Application Note

DCAN to MCAN Migration Guide

Sahil Deshpande and Hareesh Janakiraman *C2000 Microcontrollers*

ABSTRACT

C2000 Real-time MCUs feature three types of Controller Area Network (CAN) modules: eCAN, DCAN and MCAN. While eCAN and DCAN only support classic CAN, MCAN supports both classic CAN and CAN FD. Devices such as TMS320F2838xD, TMS320F2838xS, TMS320F28003x and TMS320F280015x feature both DCAN and MCAN modules. Some C2000 devices only feature the MCAN module since it supports both classic CAN and CAN FD. Although all of the aforementioned CAN modules are compliant with the CAN protocol standard, none of them are software-compatible with each other. Specifically, the DCAN and MCAN module employ a completely different architecture and hence, register and bit structure. This warrants a very different programming approach between the modules. This document is intended to ease the migration from DCAN to the MCAN module. It discusses the most common operations such as module initialization, bit-timing configuration, message RAM configuration, buffer and FIFO configuration, data transmission, reception (with filtering) and error handling. It explains how these operations are done in DCAN and MCAN modules. Code snippets are shown as warranted.

Table of Contents

Trademarks

All trademarks are the property of their respective owners.

1 Introduction

On any given device, C2000 MCUs have typically featured only one type of CAN module. For example, either eCAN or DCAN. When MCAN was introduced in the C2000 family, some MCUs featured both DCAN and MCAN. This necessitated the user to understand and program two completely different types of CAN modules. To eliminate this hardship, MCAN has been chosen as the CAN platform moving forward, since MCAN supports both classic CAN as well as CAN FD. This document lists the primary differences between the DCAN and MCAN modules. It then proceeds to highlight how common operations are done in both modules.

To determine which CAN module is featured in a given C2000 MCU, see C2000 Real-time Control MCU Peripherals Guide.

2 Key Differences Between DCAN and MCAN

CAN FD offers two significant advantages over classic CAN:

- Faster bit-rate in the data-phase, increasing the overall throughput. An application can choose to transmit the entire frame at the same bit-rate by setting *CCCR.BRSE* = 0. This way, the application still takes advantage of the higher payload capability of CAN FD.
- Higher payload size (up to 64 bytes) compared to classic CAN (up to 8 bytes), reducing protocol overhead.

Note that the physical-layer requirements in terms of transceiver, bus termination and so forth is identical between Classic CAN and CAN FD. If higher bit-rates are desired for the data phase in CAN FD, then transceivers designed for such bit-rates must be used.

Table 2-1 highlights the key differences between the DCAN and MCAN modules from a usage and programming perspective.

Table 2-1. DCAN and MCAN Feature Differences

Feature	DCAN	MCAN
Bit-rate	Fixed bit-rate for the entire frame	Two bit-rates can be used: a slower bit-rate for the *nominal phase* and a faster bit-rate for the *data phase*
Transmission speed	Capped at 1Mbps	Up to 1Mbps can be used for the *nominal phase* and up to 5Mbps for the *data phase*
Number of bytes transmitted per frame (Payload capability)	Any number of bytes from 0 to 8 can be transmitted	In addition to 0 to 8 bytes, transmission of 12/16/20/24/32/48/64 data bytes is possible
Nomenclature of data storage elements	Data is stored in Message Objects. Message objects are also sometimes referred to as Mailboxes.	Data is stored in buffers tied to *filter elements*
Number of data storage elements	Fixed at 32, regardless of the number of bytes to be transmitted or received	Depending on the configuration of the element, the number of buffers is flexible
CRC-field length	15 bits	15, 17 or 21-bit CRC
Time-stamping support	No	Yes
Transmitter delay compensation	Not required	Required for faster bit-rates in the data phase

3 Module Initialization

The first few initialization steps are identical for both DCAN and MCAN modules. The initialization mode is entered either by software (setting the *CAN_CTL.INIT* and *MCAN_CCCR.INIT* bits, respectively), by a hardware reset, by going to a bus-off state or in the case of MCAN, on the detection of an uncorrected bit error in the Message RAM. In this state, the message transfer is stopped, the CANTX output is driven recessive (high) and the error counters remain unchanged. Setting the *INIT* bit does not change any configuration registers.

To complete the software initialization, the *INIT* bit can be reset, and after an occurrence of a sequence of 11 recessive bits (Bus-idle state), communication can commence.

The step-by-step process for module initialization is shown below for each module.

3.1 DCAN Initialization

1. Initialize Message RAM
2. Configure bit-timing
3. Configure Message Objects (optional → can also be done after initialization and while module is operational).

3.2 MCAN Initialization

1. Configure Message RAM (See Section 5).
2. Configure CAN mode (Classic CAN or CAN FD).
3. Configure bit-timing (See Section 4).
4. Configure bit-rate switching (enable or disable).
5. Configure Filter Elements (optional → can also be done after initialization and while module is operational).

Note that the status registers related to Tx/Rx are reset on switching to *init* mode in MCAN.

3.3 Initialization sequence

The individual steps for initialization of DCAN and MCAN modules, along with the key differences have been listed in Table 3-1:

Table 3-1. DCAN/MCAN Initialization Sequence

Operation	DCAN	MCAN
Enter Initialization Mode	Set *CAN_CTL.INIT* bit	Set *MCAN_CCCR.INIT* bit and check that the bit has been set
Unlock Protected Registers	Set *CAN_CTL.CCE* bit	Set *MCAN_CCCR.CCE* bit
Configure CAN Mode and Bit Rate Switching	Not applicable	Set *MCAN_CCCR.FDOE* bit for CAN FD function Set *MCAN_CCCR.BRSE* bit to enable Bit Rate Switching (BRS) (Both bits need to be 0 for Classic CAN Communication)
Configure bit-timing	Configure *CAN_BTR* register	Configure *MCAN_NBTP* register
Configure data bit-timing	Not applicable	Configure *MCAN_DBTP* register (Not needed for Classic CAN as BRS is disabled)
Message RAM Configuration	Not applicable	See Message RAM configuration
Global Filter Configuration, if required (determines how the module handles non-matching frames).	Not applicable	Set *MCAN_GFC* Register
Receive and Transmit Configuration (can be done at run time as well)	Setup Message Object	Filter Configuration
Lock Protected Registers	Clear *CAN_CTL.CCE* bit	Clear *MCAN_CCCR.CCE* bit
Return module to normal operation	Clear *CAN_CTL.INIT* bit	Clear *MCAN_CCCR.INIT* bit

In addition to the steps shown above, for MCAN, the MCAN Clock Divider may need to be set up as part of the initialization process. This configuration is typically done via the AUXCLKDIVSEL register (refer to the device-specific TRM to determine the register for clock division). For 120 MHz and 200 MHz devices, *C2000ware* examples configure the MCAN bit-clock to 40 MHz. If an application desires a smaller time quanta (TQ), other configurations for the bit-clock are possible. However, the parameters for the Nominal and Data Bit Timings need to be changed accordingly. Figure 3-1 shows the initialization steps for DCAN. Figure 3-2, Figure 3-3, and Figure 3-4 show the initialization steps for MCAN.

3.4 Code Snippets for Module Initialization

Code snippets for module initialization is shown in the following figures.

```
main()
{
    //
    // Initialize device clock and peripherals
    //
    Device_init();

    //
    // Initialize GPIO and configure GPIO pins for CANTX/CANRX
    //
    Device_initGPIO();

    //
    // Configuring the GPIOs for DCAN.
    //
    GPIO_setPinConfig(DEVICE_GPIO_CFG_CANRXA);
    GPIO_setPinConfig(DEVICE_GPIO_CFG_CANTXA);

    //
    // Initialize the CAN controller
    //
    CAN_initModule(CANA_BASE);

    //
    // Set up the CAN bus bit rate to 500 kbps
    //
    CAN_setBitRate(CANA_BASE, DEVICE_SYSCLK_FREQ, 500000, 16);

    //
    // Initialize the transmit message object used for sending CAN messages.
    //
    CAN_setupMessageObject(CANA_BASE, TX_MSG_OBJ_ID, 0x1,
                           CAN_MSG_FRAME_STD, CAN_MSG_OBJ_TYPE_TX, 0,
                           CAN_MSG_OBJ_NO_FLAGS, MSG_DATA_LENGTH);

    //
    // Initialize the receive message object used for receiving CAN messages.
    //
    CAN_setupMessageObject(CANA_BASE, RX_MSG_OBJ_ID, 0x1,
                           CAN_MSG_FRAME_STD, CAN_MSG_OBJ_TYPE_RX, 0,
                           CAN_MSG_OBJ_NO_FLAGS, MSG_DATA_LENGTH);

    //
    // Start CAN module operations
    //
    CAN_startModule(CANA_BASE);
}
```

Figure 3-1. DCAN Initialization

```
//
// Function Prototype.
//
static void MCANConfig(void);

main()
{
    //
    // Initialize device clock and peripherals
    //
    Device_init();

    //
    // Initialize GPIO and unlock the GPIO configuration registers
    //
    Device_initGPIO();

    //
    // Configuring the GPIOs for MCAN.
    //
    GPIO_setPinConfig(DEVICE_GPIO_CFG_MCANRXA);
    GPIO_setPinConfig(DEVICE_GPIO_CFG_MCANTXA);

    #ifdef F2838x
    //
    // Allocate shared peripheral to C28x (Applicable only for 320F2838x)
    //
    SysCtl_allocateSharedPeripheral(SYSCTL_PALLOCATE_MCAN_A,0x0U);

    //
    // Configure the divisor for the MCAN bit-clock
    //
    SysCtl_setMCANClk(SYSCTL_MCANCLK_DIV_5);
    #else
    //
    // Configure the divisor for the MCAN bit-clock
    //
    SysCtl_setMCANClk(SYSCTL_MCANCLK_DIV_3);
    #endif

    //
    // Configure MCAN (shown separately)
    //
    MCANConfig();
}
```

Figure 3-2. MCAN GPIO and Clock Initialization

```
static void MCANConfig(void)
{
    MCAN_InitParams initParams;
    MCAN_ConfigParams configParams;
    MCAN_MsgRAMConfigParams msgRAMConfigParams;
    MCAN_MsgRAMConfigParams    msgRAMConfigParams;
    MCAN_BitTimingParams       bitTimes;

    //
    //  Initializing all structs to zero to prevent stray values
    //
    memset(&initParams, 0, sizeof(initParams));
    memset(&configParams, 0, sizeof(configParams));
    memset(&msgRAMConfigParams, 0, sizeof(msgRAMConfigParams));
    memset(&bitTimes, 0, sizeof(bitTimes));

    //
    // Initialize MCAN Init parameters.
    //
    initParams.fdMode            = 0x0U; // FD operation disabled.
    initParams.brsEnable         = 0x0U; // Bit rate switching for
                                         // transmissions disabled.
    initParams.txpEnable         = 0x0U; // Transmit pause disabled.
    initParams.darEnable         = 0x1U; // Disable Automatic retransmission of
                                         // messages not transmitted successfully
    initParams.tdcEnable         = 0x1U; // Transmitter Delay Compensation is
                                         // enabled.
    initParams.wdcPreload        = 0xFFU;// Start value of the Message RAM
                                         // Watchdog Counter preload.

    //
    // Initialize MCAN Config parameters.
    //
    configParams.asmEnable         = 0x0U;  // Normal CAN operation.
    configParams.tsPrescalar       = 0xFU;  // Prescaler Value.
    configParams.tsSelect          = 0x0U;  // Timestamp counter value.
    configParams.timeoutSelect     = MCAN_TIMEOUT_SELECT_CONT;
    // Time-out counter source select.
    configParams.timeoutPreload    = 0xFFFFU; // Start value of the Timeout
                                              // Counter.
    configParams.timeoutCntEnable  = 0x0U; // Time-out Counter is disabled.
    configParams.filterConfig.rrfs = 0x1U; // Reject all remote frames with
                                           // 29-bit extended IDs.
    configParams.filterConfig.rrfe = 0x1U; // Reject all remote frames with
                                           // 11-bit standard IDs.
    configParams.filterConfig.anfe = 0x2U; // Reject Non-Matching Frames Extended
    configParams.filterConfig.anfs = 0x2U; // Reject Non-Matching Frames Standard

    //***********
    // Message RAM Configuration Section Here
    //***********

    //***********
    // Bit Timing Configuration Section Here
    //***********
```

Figure 3-3. MCAN Operating Mode and Global Filter Configuration

```
    //
    // Wait for Memory initialization to be completed.
    //
    while(FALSE == MCAN_isMemInitDone(MCANA_DRIVER_BASE));

    //
    // Put MCAN in SW initialization mode.
    //
    MCAN_setOpMode(MCANA_DRIVER_BASE, MCAN_OPERATION_MODE_SW_INIT);

    //
    // Wait till MCAN is not initialized.
    //
    while (MCAN_OPERATION_MODE_SW_INIT != MCAN_getOpMode(MCANA_DRIVER_BASE));

    //
    // Initialize MCAN module.
    //
    MCAN_init(MCANA_DRIVER_BASE, &initParams);

    //
    // Configure MCAN module.
    //
    MCAN_config(MCANA_DRIVER_BASE, &configParams);

    //
    // Configure Bit timings.
    //
    MCAN_setBitTime(MCANA_DRIVER_BASE, &bitTimes);

    //
    // Configure Message RAM Sections
    //
    MCAN_msgRAMConfig(MCANA_DRIVER_BASE, &msgRAMConfigParams);

    //
    // Take MCAN out of the SW initialization mode
    //
    MCAN_setOpMode(MCANA_DRIVER_BASE, MCAN_OPERATION_MODE_NORMAL);

    while (MCAN_OPERATION_MODE_NORMAL != MCAN_getOpMode(MCANA_DRIVER_BASE));
```

Figure 3-4. MCAN Initialization completion

4 Bit Timing Configuration

Bit timing configuration differs between classic CAN and CAN FD. The process is *relatively* easier in classic CAN, since the bit-rate is the same for the entire frame. However, in CAN FD, two different bit-rates can be used: a slower “nominal” bit-rate and a faster “data” bit-rate. In the MCAN module, these two bit-rates are configured by writing to the *MCAN_NBTP* and *MCAN_DBTP* registers respectively during the module initialization. Note that an application can choose to only take advantage of the higher number of data bytes per frame that can be transmitted in CAN FD and use the same bit-rate for the entire frame. The faster bit-rate for the data phase also warrants Transmitter Delay Compensation (TDC) without which bit errors can occur. Below is an example for calculating bit-timing parameters:

Example 1

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

Nominal bit-rate = 500kbps, Data bit-rate = 2Mbps.

The formula for computing the bit-rate is given by:

$$Bit - rate = \frac{\text{CAN module clock}}{\text{Bit} - \text{rate prescaler} \times \text{Bit} - \text{time}} \quad (1)$$

For a nominal bit-rate of 500kbps, the product of *Bit-rate prescaler x Bit-time* must be equal to 400. This can be achieved with multiple combinations of the prescaler and bit-time, as long as the rules laid out by the CAN protocol are not violated. For example, a prescaler of 20 and a bit-time of 20 TQ can be chosen. A prescaler of 20 ($NBRP_{reg}$ = 19) yields a bit-clock of 10 MHz with a resultant time-quanta (TQ) of 100 ns. A bit-time of 20 TQ can be achieved with multiple combinations for TSEG1 and TSEG2 resulting in varying sample-points (SP).

Bit-time = ($NTSEG1_{reg}$ + 1) + ($NTSEG2_{reg}$ + 1) + 1, where $NTSEG1_{reg}$ and $NTSEG2_{reg}$ represent the actual values written into *MCAN_NBTP.NTSEG1* and *MCAN_NBTP.NTSEG2* bit-fields respectively. If TSEG1 is chosen to be 16 ($NTSEG1_{reg}$ = 15) and TSEG2 is chosen to be 4 ($NTSEG2_{reg}$ =3), those values yield a sampling-point of 80%. By adjusting the TSEG1 and TSEG2 values, the sampling-point can be moved within the bit-time based on the network parameters.

Similar calculation is used for data bit-rate of 2Mbps. For a data bit-rate of 2Mbps, the product of *Bit-rate prescaler x Bit-time* must be equal to 100. This can be achieved with multiple combinations of the prescaler and bit-time, as long as the rules laid out by the CAN protocol are not violated. For example, a prescaler of 5 and a bit-time of 20 TQ can be chosen. A prescaler of 5 ($DBRP_{reg}$ = 4) yields a bit-clock of 40 MHz with a resultant time-quanta (TQ) of 25ns. A bit-time of 20 TQ can be achieved with multiple combinations for TSEG1 and TSEG2 resulting in varying sample-points (SP).

Bit-time = ($DTSEG1_{reg}$ + 1) + ($DTSEG2_{reg}$ + 1) + 1, where $DTSEG1_{reg}$ and $DTSEG1_{reg}$ represent the actual values written into *MCAN_DBTP.DTSEG1* and *MCAN_DBTP.DTSEG2* bit-fields respectively. If TSEG1 is chosen to be 16 ($DTSEG1_{reg}$ = 15) and TSEG2 is chosen to be 4 ($NTSEG2_{reg}$ =3), those values yield a sampling-point of 80%. By adjusting the TSEG1 and TSEG2 values, the sampling-point can be moved within the bit-time based on the network parameters.

```
//
// Initialize bit timings.
//
bitTimes.nomRatePrescalar   = 0xBU; // Nominal Baud Rate Pre-scaler
bitTimes.nomTimeSeg1        = 0x2U; // Nominal Time segment before SP
bitTimes.nomTimeSeg2        = 0x0U; // Nominal Time segment after SP
bitTimes.nomSynchJumpWidth  = 0x0U; // Nominal SJW
bitTimes.dataRatePrescalar  = 0x1U; // Data Baud Rate Pre-scaler
bitTimes.dataTimeSeg1       = 0xAU; // Data Time segment before SP
bitTimes.dataTimeSeg2       = 0x2U; // Data Time segment after SP
bitTimes.dataSynchJumpWidth = 0x2U; // Data SJW

// The above is just an illustrative example. You must compute the timing values
// based on your network paramters such as oscillator accuracy, propagation delay
// introduced by the transceivers and the bus.

```

Figure 4-1. MCAN Bit-timing Configuration

5 Message RAM Configuration

In DCAN, Message RAM can only be accessed by the Message Handler. The *Driverlib* APIs interact with the Message Interface (IFx) registers which carry out the read or write operations with the Message RAM. In MCAN, *Driverlib* APIs can be used to carry out the read or write operations with the Message RAM directly.

The Message RAM is structured differently in MCAN as compared to DCAN. In DCAN, the number of message objects in the Message RAM is fixed at 32 and each message object can be configured for either transmit or receive operation.

However, in MCAN, Message RAM can be configured to have the following sections:

- Standard Filter Element
- Extended Filter Element
- Rx Buffer
- Rx FIFO
- Tx Buffer
- Tx FIFO or Tx Queue
- Tx Event FIFO

The design of MCAN Message RAM offers tremendous flexibility, enabling allocation of the available memory to each of the sections mentioned above based on the application needs. The sections can be ordered in any manner and the unused sections can be allocated zero memory. Note that Message RAM size can vary from one device to another. Refer the device-specific data sheet for more information.

Message RAM configuration involves defining the following:

- Starting address for every section used.
- The number of elements in each section.
- The size of the elements which is different for different size of data packets as can be seen in Table 5-1 (Filter Elements and Tx Event FIFO have fixed sizes).

These values are written to specific registers and are subsequently used by both the Message Handler and *Driverlib* APIs to interact with the Message RAM. Hence, Message RAM configuration is a crucial step for MCAN during module initialization, while such configuration is not required in DCAN. The MCAN module addresses the Message RAM in 32-bit words. Consequently, all sections are of sizes that are multiples of 32-bit words.

Table 5-1. Element Size Vs Size of Data Packets

MCAN_RXESC.RBDS/ MCAN_RXESC.F0DS/ MCAN_RXESC.F1DS/ MCAN_TXESC.TBDS (Corresponding to Rx Buffer, Rx FIFOs and Tx Buffer, respectively)	Data Field (bytes)	FIFO Element Size (or) Buffer Element Size [RAM words]
000	8	4
001	12	5
010	16	6
011	20	7
100	24	8
101	32	10
110	48	14
111	64	18

Macros are available in *C2000ware* examples which automatically calculate the starting address for each section when the number and size of elements are set by the user. This configuration can be successfully utilized in any application. Multiple valid configurations are possible and there is not a single "correct" configuration. Note that the MCAN module does not check for invalid configurations. It is the responsibility of the user to verify that sections do not overlap each other or exceed the available RAM.

```
//
// Defines
//
#define MCAN_STD_ID_FILTER_NUM          (1U)
#define MCAN_EXT_ID_FILTER_NUM          (0U)
#define MCAN_FIFO_0_NUM                 (5U)
#define MCAN_FIFO_0_ELEM_SIZE           (MCAN_ELEM_SIZE_64BYTES)
#define MCAN_FIFO_1_NUM                 (10U)
#define MCAN_FIFO_1_ELEM_SIZE           (MCAN_ELEM_SIZE_64BYTES)
#define MCAN_RX_BUFF_NUM                (10U)
#define MCAN_RX_BUFF_ELEM_SIZE          (MCAN_ELEM_SIZE_64BYTES)
#define MCAN_TX_BUFF_SIZE               (10U)
#define MCAN_TX_FQ_SIZE                 (0U)
#define MCAN_TX_BUFF_ELEM_SIZE          (MCAN_ELEM_SIZE_64BYTES)
#define MCAN_TX_EVENT_SIZE              (10U)

//
//  Defining Starting Addresses for Message RAM Sections,
//  (Calculated from Macros based on User defined configuration above)
//
#define MCAN_STD_ID_FILT_START_ADDR     (0x0U)
#define MCAN_EXT_ID_FILT_START_ADDR     (MCAN_STD_ID_FILT_START_ADDR + ((MCAN_STD_ID_FILTER_NUM * MCANSS_STD_ID_FILTER_SIZE_WORDS * 4U)))
#define MCAN_FIFO_0_START_ADDR          (MCAN_EXT_ID_FILT_START_ADDR + ((MCAN_EXT_ID_FILTER_NUM * MCANSS_EXT_ID_FILTER_SIZE_WORDS * 4U)))
#define MCAN_FIFO_1_START_ADDR          (MCAN_FIFO_0_START_ADDR + (MCAN_getMsgObjSize(MCAN_FIFO_0_ELEM_SIZE) * 4U * MCAN_FIFO_0_NUM))
#define MCAN_RX_BUFF_START_ADDR         (MCAN_FIFO_1_START_ADDR + (MCAN_getMsgObjSize(MCAN_FIFO_1_ELEM_SIZE) * 4U * MCAN_FIFO_1_NUM))
#define MCAN_TX_BUFF_START_ADDR         (MCAN_RX_BUFF_START_ADDR + (MCAN_getMsgObjSize(MCAN_RX_BUFF_ELEM_SIZE) * 4U * MCAN_RX_BUFF_NUM))
#define MCAN_TX_EVENT_START_ADDR        (MCAN_TX_BUFF_START_ADDR + (MCAN_getMsgObjSize(MCAN_TX_BUFF_ELEM_SIZE) * 4U * (MCAN_TX_BUFF_SIZE + MCAN_TX_FQ_SIZE)))

```

Figure 5-1. MCAN Message RAM Macros

```

    //
    // Initialize Message RAM Sections Configuration Parameters.
    //
    msgRAMConfigParams.flssa                = MCAN_STD_ID_FILT_START_ADDR;
    // Standard ID Filter List Start Address.
    msgRAMConfigParams.lss                  = MCAN_STD_ID_FILTER_NUM;
    // List Size: Standard ID.
    msgRAMConfigParams.flesa                = MCAN_EXT_ID_FILT_START_ADDR;
    // Extended ID Filter List Start Address.
    msgRAMConfigParams.lse                  = MCAN_EXT_ID_FILTER_NUM;
    // List Size: Extended ID.
    msgRAMConfigParams.txStartAddr          = MCAN_TX_BUFF_START_ADDR;
    // Tx Buffers Start Address.
    msgRAMConfigParams.txBufNum             = MCAN_TX_BUFF_SIZE;
    // Number of Dedicated Transmit Buffers.
    msgRAMConfigParams.txFIFOSize           = MCAN_TX_FQ_SIZE;
    // Number of Tx FIFO or Tx Queue Elements
    msgRAMConfigParams.txBufMode            = 0U; //Tx FIFO operation
    msgRAMConfigParams.txBufElemSize        = MCAN_TX_BUFF_ELEM_SIZE;
    // Tx Buffer Element Size.
    msgRAMConfigParams.txEventFIFOStartAddr = MCAN_TX_EVENT_START_ADDR;
    // Tx Event FIFO Start Address.
    msgRAMConfigParams.txEventFIFOSize      = MCAN_TX_BUFF_SIZE;
    // Event FIFO Size.
    msgRAMConfigParams.txEventFIFOWaterMark = 3U;
    // Level for Tx Event FIFO watermark interrupt.
    msgRAMConfigParams.rxFIFO0startAddr     = MCAN_FIFO_0_START_ADDR;
    // Rx FIFO0 Start Address.
    msgRAMConfigParams.rxFIFO0size          = MCAN_FIFO_0_NUM;
    // Number of Rx FIFO elements.
    msgRAMConfigParams.rxFIFO0waterMark     = 3U; // Rx FIFO0 Watermark.
    msgRAMConfigParams.rxFIFO0OpMode        = 0U; // FIFO blocking mode.
    msgRAMConfigParams.rxFIFO1startAddr     = MCAN_FIFO_1_START_ADDR;
    // Rx FIFO1 Start Address.
    msgRAMConfigParams.rxFIFO1size          = MCAN_FIFO_1_NUM;
    // Number of Rx FIFO elements.
    msgRAMConfigParams.rxFIFO1waterMark     = 3U; // Level for Rx FIFO 1
                                                  // watermark interrupt.
    msgRAMConfigParams.rxFIFO1OpMode        = 0U; // FIFO blocking mode.
    msgRAMConfigParams.rxBufStartAddr       = MCAN_RX_BUFF_START_ADDR;
    // Rx Buffer Start Address.
    msgRAMConfigParams.rxBufElemSize        = MCAN_RX_BUFF_ELEM_SIZE;
    // Rx Buffer Element Size.
    msgRAMConfigParams.rxFIFO0ElemSize      = MCAN_FIFO_0_ELEM_SIZE;
    // Rx FIFO0 Element Size.
    msgRAMConfigParams.rxFIFO1ElemSize      = MCAN_FIFO_1_ELEM_SIZE;
    // Rx FIFO1 Element Size.
```

Figure 5-2. MCAN Message RAM Initialization

6 Interrupt handling

From a CPU level (PIE, IFR and INTM), interrupt handling is identical between DCAN and MCAN. However, interrupt handling differs significantly at the module level. Table 6-1 summarizes the basic differences in interrupt handling between DCAN and MCAN modules:

Table 6-1. Interrupt Handling in DCAN and MCAN

Category	DCAN	MCAN
Interrupt sources	Error, Status and Transmission/Reception interrupts corresponding to each message object	30 internal interrupt sources (specified in table below)
Global interrupt registers	Registers to enable, read and clear global interrupts present	Corresponding register is absent
Configuring reception interrupt	Reception interrupt can be separately enabled by setting RxIE bit in each Message Object as required	Interrupt can be enabled or disabled for any new message being received in dedicated Rx Buffer.
Determining source of receive interrupt	Value read from register CAN_INT corresponds to Message Object Number where message has been received in	Interrupt only denotes that a new message has been received in Rx Buffer. Value read from MCAN_NDATx registers corresponds to Rx Buffer element number where message has been received.
Rx FIFO interrupts	No separate interrupt functionality supported	Additional interrupt sources available including New Message in FIFO, FIFO being full and FIFO Watermark Reached (Watermark can be configured during Message RAM configuration to generate an interrupt when FIFO is filled to a certain level to serve the application needs)
Configuring transmission interrupts	Transmission interrupt can be separately enabled by setting TxIE bit in each Message Object as required	Transmission interrupt can be separately enabled by configuring the register MCAN_TXBTIE, where each bit corresponds to a separate Tx Buffer Element.
Determining Source of transmission interrupt	Value read from register CAN_INT corresponds to Message Object Number where message has been transmitted from	Interrupt only denotes that a transmission has been completed. Value read from the MCAN_TXBTO register corresponds to the Tx Buffer element number from which the transmission has occurred.

6.1 MCAN Interrupt Sources

The different interrupt sources for MCAN have been documented in Table 6-2:

Table 6-2. MCAN Interrupt Sources

Interrupt	Description
ARA	Access to Reserved Address
PED	Protocol Error in Data Phase
PEA	Protocol Error in Arbitration Phase
WDI	Watchdog
BO	Bus-off
EW	Warning Status
EP	Error Passive
ELO	Error Logging Overflow
BEU	Bit Error Uncorrected

Table 6-2. MCAN Interrupt Sources (continued)

Interrupt	Description
BEC	Bit Error Corrected
DRX	Message Stored to Dedicated Rx Buffer
TOO	Timeout Occurred
MRAF	Message RAM Access Failure
TSW	Timestamp Wraparound
TEFL	Tx Event FIFO Element Lost
TEFF	Tx Event FIFO Full
TEFW	Tx Event FIFO Watermark Reached
TEFN	Tx Event FIFO New Entry
TFE	Tx FIFO Empty
TCF	Transmission Cancellation Finished
TC	Transmission Completed
HPM	High Priority Message
RF1L	Rx FIFO 1 Message Lost
RF1F	Rx FIFO 1 Full
RF1W	Rx FIFO 1 Watermark Reached
RF1N	Rx FIFO 1 New Message
RF0L	Rx FIFO 0 Message Lost
RF0F	Rx FIFO 0 Full
RF0W	Rx FIFO 0 Watermark Reached
RF0N	Rx FIFO 0 New Message

6.2 DCAN Interrupt Handling

Device-level Interrupt Configurations:

1. Initialize PIE and PIE Vector Table. Enable Global and Real-time Interrupts.
2. Configure the interrupt handler in the PIE Vector Table. Enable interrupt in the interrupt controller.

Module-level Interrupt Configurations

1. Enable Error and Status interrupts, using the CAN Control Register (CAN_CTL). Enable Message Object interrupts while setting up Message Objects individually.
2. Select interrupt line where each message object interrupt is to be routed using the register (CAN_IP_MUX21), where each bit corresponds to a single message object.
3. Interrupt Service Routine (ISR) : Read Interrupt Register (CAN_INT) to determine the source of the interrupt (status/error/particular message object). Clear the Interrupt by writing to CAN Error and Status Register (CAN_ES) or by clearing the *IntPnd* bit in the corresponding Message Object. Clear the Global Interrupt Flag for the corresponding Interrupt Line.
4. Acknowledge the interrupt via PIEACK.

```
//
// Function Prototypes
//
__interrupt void canISR(void);

{
    //
    // Initialize PIE and clear PIE registers. Disables CPU interrupts.
    //
    Interrupt_initModule();

    //
    // Initialize the PIE vector table with pointers to the shell Interrupt
    // Service Routines (ISR).
    //
    Interrupt_initVectorTable();

    // Enable Global Interrupt (INTM) and realtime interrupt (DBGM)
    //
    EINT;
    ERTM;

    // Interrupts that are used in this example are re-mapped to
    // ISR functions found within this file.
    // This registers the interrupt handler in PIE vector table.
    //
    Interrupt_register(INT_CANA0,&canaISR);

    //
    // Enable the CAN-A interrupt signal
    //
    Interrupt_enable(INT_CANA0);

    //
    // Enable interrupts on the CAN A peripheral. (error, status, line 0/1)
    //
    CAN_enableInterrupt(CANA_BASE, CAN_INT_IE0 | CAN_INT_ERROR |
                        CAN_INT_STATUS);

    // Enable Global Interrupt for corresponding interrupt line
    //
    CAN_enableGlobalInterrupt(CANA_BASE, CAN_GLOBAL_INT_CANINT0);

    //
    //  Transmit/Receive Interrupt enabled during setup Message object Function
    //
    CAN_setupMessageObject(CANA_BASE, TX_MSG_OBJ_ID, 0x15555555,
                           CAN_MSG_FRAME_EXT, CAN_MSG_OBJ_TYPE_TX, 0,
                           CAN_MSG_OBJ_TX_INT_ENABLE, MSG_DATA_LENGTH);

    //  Select Interrupt Line for each mailbox
    //
    CAN_setInterruptMux(CANA_BASE, mux);

}
```

Figure 6-1. DCAN Interrupt Initialization

```
//
// CAN A ISR - The interrupt service routine called when a CAN interrupt is
//             triggered on CAN module A.
//
__interrupt void
canaISR(void)
{
    uint32_t status;

    // Read the CAN-A interrupt status to find the cause of the interrupt

    status = CAN_getInterruptCause(CANA_BASE);

    // If the cause is a controller status interrupt, then get the status

    if(status == CAN_INT_INT0ID_STATUS)     // Read the controller status.
    {
        status = CAN_getStatus(CANA_BASE);

        // Check to see if an error occurred.

        if(((status  & ~(CAN_STATUS_TXOK)) != CAN_STATUS_LEC_MSK) &&
           ((status  & ~(CAN_STATUS_TXOK)) != CAN_STATUS_LEC_NONE))
        {
            errorFlag = 1;       // Set a flag to indicate some errors may have occurred.
        }
    }
    else if(status == TX_MSG_OBJ_ID)
    {
        // Transmit Message handling will go here

        CAN_clearInterruptStatus(CANA_BASE, TX_MSG_OBJ_ID);  // Clear the message object interrupt
    }
    else if(status == RX_MSG_OBJ_ID)
    {
        // Receive message handling will go here

        CAN_clearInterruptStatus(CANA_BASE, RX_MSG_OBJ_ID);  // Clear the message object interrupt
    }

    else
    {
        //
        // Spurious interrupt handling can go here.
        //
    }

    //
    // Clear the global interrupt flag for the CAN interrupt line
    //
    CAN_clearGlobalInterruptStatus(CANA_BASE, CAN_GLOBAL_INT_CANINT0);

    //
    // Acknowledge this interrupt located in group 9
    //
    Interrupt_clearACKGroup(INTERRUPT_ACK_GROUP9);
}
```

Figure 6-2. DCAN Interrupt handling

6.3 MCAN Interrupt Handling

Device-level Interrupt Configurations:

1. Initialize PIE and PIE Vector Table. Enable Global and Real-time Interrupts.
2. Configure the interrupt handler in the PIE Vector Table. Enable interrupt in the interrupt controller.

Module-level Interrupt Configurations

1. Enable interrupt sources using the register (MCAN_IR), where each bit corresponds to a single interrupt source. Enable interrupt lines as required using the register (MCAN_ILE).
2. Select interrupt lines where interrupt source is to be routed using the register (MCAN_ILS), where each bit corresponds to a single interrupt source.
3. Interrupt Service Routine (ISR) : Read Interrupt Register (MCAN_IR) to determine the source of the interrupt (any of the 30 individual interrupt sources). Clear the interrupt by writing to the same register. Clear the interrupt line by writing to the register (MCANSS_EOI).
4. Acknowledge the interrupt via PIEACK.

```
1
2 //
3 // Function Prototype.
4 //
5 static void MCANIntrConfig(void);
6 __interrupt void MCANIntr1ISR(void);
7
8 {
9     //
10    // ISR Configuration.
11    //
12    MCANIntrConfig();
13
14   // Enable Interrupts.
15    // (interrupts can be enabled individually by modifying the second parameter)
16    //
17    MCAN_enableIntr(MCANA_DRIVER_BASE, MCAN_INTR_MASK_ALL, 1U);
18
19    //
20    // Select Interrupt Line.
21    // (interrupt line can be individually chosen for each interrupt source)
22    //
23    MCAN_selectIntrLine(MCANA_DRIVER_BASE, MCAN_INTR_MASK_ALL, MCAN_INTR_LINE_NUM_1);
24
25    //
26    // Enable Interrupt Line.
27    //
28    MCAN_enableIntrLine(MCANA_DRIVER_BASE, MCAN_INTR_LINE_NUM_1, 1U);
29
30    //
31    // Enable Transmission interrupt.
32    // (Needs to be done individually for each Tx Buffer Element)
33    // (Additionally, transmission related interrupt sources will need to be enabled above,
34    // as desired by application)
35    MCAN_txBufTransIntrEnable(MCANA_DRIVER_BASE, 1U, 1U);
36 }
37
38 //
39 // This function will configure X-BAR for MCAN interrupts.
40 //
41 static void MCANIntrConfig(void)
42 {
43
44    Interrupt_initModule();
45    Interrupt_initVectorTable();
46
47    Interrupt_register(INT_MCANA_1,&MCANIntr1ISR);
48    Interrupt_enable(INT_MCANA_1);
49
50    Interrupt_enableGlobal();
51 }
```

Figure 6-3. MCAN Interrupt Initialization

```
//
// This is Interrupt Service Routine for MCAN interrupt 1.
//
__interrupt void MCANIntr1ISR(void)
{
    uint32_t intrStatus;

    intrStatus = MCAN_getIntrStatus(MCANA_DRIVER_BASE);

    //  Clearing the corresponding interrupt line
    //
    HW_WR_FIELD32(MCANA_DRIVER_BASE + MCAN_MCANSS_EOI, MCAN_MCANSS_EOI, 0x2);

    // Clear the interrupt Status.
    //
    MCAN_clearIntrStatus(MCANA_DRIVER_BASE, intrStatus);

    //
    //  Check to see if the interrupt is caused by
    //  reception of new message in dedicated Rx Buffer
    //
    if((MCAN_INTR_SRC_DEDICATED_RX_BUFF_MSG & intrStatus) == MCAN_INTR_SRC_DEDICATED_RX_BUFF_MSG)
    {
        //
        //  Receive Message Handling will go here
    }
    //
    //  Check to see if the interrupt is caused by
    //  reception of new message in Rx FIFO 1
    //
    if((MCAN_INTR_SRC_RX_FIFO1_NEW_MSG & intrStatus) == MCAN_INTR_SRC_RX_FIFO1_NEW_MSG)
    {
        //
        //  Receive Message Handling will go here
    }
    //
    //  Check to see if the interrupt is caused by
    //  completion of transmission of message
    //
    if((MCAN_INTR_SRC_TRANS_COMPLETE & intrStatus) == MCAN_INTR_SRC_TRANS_COMPLETE)
    {
        //
        //  Transmit Message Handling will go here
    }
    //
    // Similar logic can be implemented to check for other interrupt sources and determine
    // the actions based on the needs of the application
    //

    // Acknowledge this interrupt located in group 9
    //
    Interrupt_clearACKGroup(INTERRUPT_ACK_GROUP9);
}
```

Figure 6-4. MCAN Interrupt Handling

7 Transmitting data

The overall transmission process is largely identical between DCAN and MCAN modules. The differences stem primarily from how the Message RAM is laid out and utilized. Also, MCAN frames can be longer and employ two different bit-rates. Once configured appropriately, the module takes care of bit-rate switching and handling the larger payload. No code intervention is needed at that point.

7.1 Basic Transmission Process

This section outlines the flow including the actions required and the registers involved in the process of transmitting a frame using DCAN and MCAN.

7.1.1 Transmission with DCAN

1. Setup a transmit Message Object.
2. Write to the IFx registers which in turn writes the Message ID (ARBID), DLC and data to the Message Object (and update Message ID, if warranted).
3. Set *TXRQST* bit in the IFx register (*CAN_IFxCMD*) to signal that Message Object is ready for transmission.
4. When bus is idle, Message Handler parses Message Objects ready for transmission and transmits the highest priority Message available.

7.1.2 Transmission with MCAN

1. Initialize Transmit Buffer Element (defined as a structure).
2. Write Tx Message to Message RAM.
3. Set the bit corresponding to the Tx Buffer Element Number in the *MCAN_TXBAR* register (each bit represents a separate buffer element) to signal that message is ready for transmission.
4. When bus is idle, Message Handler parses Buffer Elements ready for transmission and transmits the highest priority Message available.

7.2 MCAN Vs DCAN Transmit Procedural Differences

Although the conceptual procedure for transmission is largely identical for DCAN and MCAN, Table 7-1 captures the key differences between both modules:

Table 7-1. MCAN Vs DCAN Transmit Procedure

Category	DCAN	MCAN
Transmit Priority	Numerically lowest Message Object (that is ready for transmission) is transmitted first	Buffer containing numerically lowest Message ID (among those ready for transmission) is transmitted first
Buffer Type	Only Transmit Message Object	Transmit Buffers can be configured as Dedicated Tx Buffers, Tx FIFO or Tx Queue
Writing/Updating Transmit Message	Requires Writes to IFx registers	Transmit Message can be updated by writing directly to Message RAM using *Driverlib* APIs

```
//
// Defines
//
#define MSG_DATA_LENGTH    4
#define TX_MSG_OBJ_ID      1

    uint16_t txMsgData[4];

    //
    // Initialize the transmit message object used for sending CAN messages.
    // Message Object Parameters:
    //      Message Object ID Number: 1
    //      Message Identifier: 0x1
    //      Message Frame: Standard
    //      Message Type: Transmit
    //      Message ID Mask: 0x0
    //      Message Object Flags: Transmit Interrupt
    //      Message Data Length: 4 Bytes
    //
    CAN_setupMessageObject(CANA_BASE, TX_MSG_OBJ_ID, 0x1, CAN_MSG_FRAME_STD,
                           CAN_MSG_OBJ_TYPE_TX, 0, CAN_MSG_OBJ_TX_INT_ENABLE,
                           MSG_DATA_LENGTH);

    //
    // Initialize the transmit message object data buffer to be sent
    //
    txMsgData[0] = 0x12;
    txMsgData[1] = 0x34;
    txMsgData[2] = 0x56;
    txMsgData[3] = 0x78;

    CAN_sendMessage(CANA_BASE, TX_MSG_OBJ_ID, MSG_DATA_LENGTH, txMsgData);

    .
```

Figure 7-1. Transmission with DCAN

7.3 MCAN Transmit Concepts

This section outlines the additional features in MCAN.

- Each Tx Message can be configured to transmit in Classic CAN or CAN-FD mode
- Transmit Pause
- Transmit Cancellation
- Tx FIFO / Tx Queue

Within the Message RAM, the Tx Buffer space can have the following possible configurations:

1. Only Tx Buffers
2. Tx Buffers + Tx FIFO
3. Tx Buffers + Tx Queue

How to configure each of these sections has been displayed in Table 7-2 below:

Table 7-2. Message RAM Configuration for Various Tx Buffer Options

Tx Buffers	Tx Buffers + Tx FIFO	Tx Buffers + Tx Queue
txBufNum = BUFF_SIZE (*MCAN_TXBC.NDTB*) txFIFOSize = 0 (*MCAN_TXBC.TFQS*)	txBufNum = BUFF_SIZE (*MCAN_TXBC.NDTB*) txFIFOSize = FIFO_SIZE (*MCAN_TXBC.TFQS*) txBufMode = 0 (*MCAN_TXBC.TFQM*)	txBufNum = BUFF_SIZE (*MCAN_TXBC.NDTB*) txFIFOSize = QUE_SIZE (*MCAN_TXBC.TFQS*) txBufMode = 1 (*MCAN_TXBC.TFQM*)

The differences in functionality and potential use cases for each of the sections have been specified in Table 7-3 below:

Table 7-3. Tx Buffers Vs Tx FIFO Vs Tx Queue Feature Comparison

Feature	Tx Buffers	Tx FIFO	Tx Queue
Information directly available to host (CPU)	Buffer Element number is known.	Only *Put* and *Get* indices can be read from a register (*MCAN_TXFQS*)	Only *Put* and *Get* indices can be read from a register (*MCAN_TXFQS*)
Element transmitted first	Element with lowest message ID	Oldest Element	Element with lowest message ID
Put Index / *Get* Index	Not applicable	*Put index* points to where the most recent frame is stored. Incremented with *Add Transmit Request*.*Get index* points to the oldest element, which is be transmitted next.	*Put index* points to the lowest, free buffer element (within the queue), where the most recent frame is stored. Updated with *Add Transmit Request*. *Get index* is always Zero
Transmission of multiple messages with same ID	Element with lowest buffer number is transmitted	Oldest Element is transmitted	Element with lowest buffer number is transmitted
Full Condition	Not applicable	In case FIFO is full, no message can be written unless a requested transmission is completed	In case queue is full, no message can be written unless a requested transmission is completed
Tx Cancellation	Possible	Not Possible	Not Possible
Use Cases	Advantage is that the application knows which message ID is stored in which buffer element and hence, can be edited before sending	Applications where frames have to be transmitted in a specific order, not following increasing order of message IDs	Advantage is that buffer number is automatically handled by the *Put index*. Application need not track which buffer is empty based on the message ID priority

```
1
2
3 // Assuming Message RAM has been configured
4
5     MCAN_TxBufElement    txMsg;               // Initialising Structure to store Transmit Message
6     uint32_t bufNum;
7
8     //
9     // Initialize message to transmit.
10    //
11    txMsg.id       = ((uint32_t)(0x1)) << 18U; // Identifier Value.
12    txMsg.rtr      = 0U; // Transmit data frame.
13    txMsg.xtd      = 0U; // 11-bit standard identifier. (Bit needs to be set in the case of extended identifier)
14    txMsg.esi      = 0U; // ESI bit in CAN FD format depends only on error
15                         // passive flag.
16    txMsg.dlc      = 4U; // CAN + CAN FD: transmit frame has 0-8 data bytes.
17    txMsg.brs      = 0U; // CAN FD frames transmitted with bit rate
18                         // switching disabled
19    txMsg.fdf      = 0U; // Frame transmitted in Classic CAN format.
20    txMsg.efc      = 1U; // Store Tx events. (Generates Tx Event FIFO element on successful transmission)
21    txMsg.mm       = 0xAAU; // Message Marker. (Used to match with corresponding Tx Event FIFO Element)
22
23    //
24    // Data bytes.
25    //
26    txMsg.data[0]  = 0x12;
27    txMsg.data[1]  = 0x34;
28    txMsg.data[2]  = 0x56;
29    txMsg.data[3]  = 0x78;
30
31    //
32    // Write Tx Message to a dedicated Tx Buffer in the Message RAM.
33    // Note: Parameter bufNum corresponds to desired buffer number
34    //
35    MCAN_writeMsgRam(MCANA_DRIVER_BASE, MCAN_MEM_TYPE_BUF, bufNum, &txMsg);
36
37    //
38    // Enable Transmission interrupt. (MCAN_TXBTIE)
39    // Each Tx Buffer has a corresponding Interrupt Enable Bit
40    // Note: Need not be enabled for every transmission
41    //
42    MCAN_txBufTransIntrEnable(MCANA_DRIVER_BASE, bufNum, 1U);
43
44    //
45    // Add request for transmission.
46    // Note: Parameter bufNum corresponds to desired buffer number
47    //
48    MCAN_txBufAddReq(MCANA_DRIVER_BASE, bufNum);
49
50    //
51    // To check that the transmission has been completed (non-essential step)
52    // Bit is set in the MCAN_TXBTO register corresponding to each buffer element
53    //
54    while((HWREG(MCANA_DRIVER_BASE + MCAN_TXBTO) & (uint32_t)(1 << bufNum)) = (uint32_t)(1 << bufNum))
55    {
56    }
57
```

Figure 7-2. Transmission with MCAN

7.3.1 Tx Event FIFO

Tx Event FIFOs are defined structures which are stored within the Message RAM. The module can be configured to have up to 32 elements.

While the Tx Buffer holds only the message to be transmitted, the transmit status (including the message ID and timestamp) can be stored separately using the Tx Event FIFO. The Message Marker is copied from the Tx Buffer to the Tx Event FIFO element to link the Tx Event to the Tx Event FIFO Element.

This is useful in applications with a dynamically managed transmit queue, where a Tx Buffer can be overwritten with the new message immediately after a successful transmission without needing to save the transmit status from the Tx Buffer itself. For more information on how to store Tx Event FIFO elements, see the example available in *C2000ware*.

8 Receiving Data

Like transmission, reception is largely identical between DCAN and MCAN. The differences stem primarily from how the Message RAM is laid out and utilized. Also, MCAN frames can be longer and employ two different bit-rates.

8.1 Introduction to Reception

In DCAN Message RAM, there are 32 configurable Message Objects that can be used for either transmission or reception. Receive Message Objects are used for storing received data. If needed by the application, acceptance filtering can be enabled for one or more message object(s). CPU read and write accesses to the Message RAM are done through three Interface Register (IFx) sets.

In MCAN, the Message RAM can be divided into sections to include Filter Elements, Rx Buffer Elements and Rx FIFO Elements. Filter Elements can be configured to be used with acceptance filtering and also determine where the corresponding matching frame is to be stored within the Message RAM. Rx Buffer and Rx FIFO are sections where the received frames are stored, with both having their own set of registers and interrupts. This structure provides flexibility to better serve different application needs. Message RAM can be read from directly using *Driverlib* APIs.

8.2 Basic Reception Process

This section outlines the high level processes involved in configuration and receiving frames using both DCAN and MCAN.

8.2.1 DCAN Reception

1. Configure Receive Message Object: This involves writing the Message IDs (ARBID) and if needed, masking for frames that are to be received.
2. For each received frame, the module checks against the Receive Message Objects in ascending order. On the first match, the frame is stored in the corresponding Message Object.
3. Either by polling or using interrupts, ascertain the reception of new data. For polling, there is a bit corresponding to each receive message object in the register *CAN_NDAT_21*. For using Interrupts, the procedure has been outlined in the corresponding section.
4. Use one of the IFx Registers to read the data from the received frame.

```
#define RX_MSG_OBJ_ID1     1
#define RX_MSG_OBJ_ID2     2

uint16_t rxMsgData[4];

    // Initialize the receive message object used for receiving CAN messages.
    // Message Object Parameters:
    //      Message Object ID Number: 1
    //      Message Identifier: 0x4
    //      Message Frame: Standard
    //      Message Type: Receive
    //      Message ID Mask: 0x0
    //      Message Object Flags: Receive Interrupt
    //      Message Data Length: 4 Bytes (Note that DLC field is a "don't care"
    //      for a Receive mailbox
    //
    CAN_setupMessageObject(CANA_BASE, RX_MSG_OBJ_ID1, 0x4, CAN_MSG_FRAME_STD,
                           CAN_MSG_OBJ_TYPE_RX, 0, CAN_MSG_OBJ_RX_INT_ENABLE,
                           0);
    //
    // Initialize the receive message object used for receiving CAN messages.
    // Possible flags: CAN_MSG_OBJ_NO_FLAGS, CAN_MSG_OBJ_USE_EXT_FILTER,
    //                 CAN_MSG_OBJ_USE_DIR_FILTER
    // Message Object Parameters:
    //      Message Object ID Number: 2
    //      Message Identifier: 0x371
    //      Message Frame: Standard
    //      Message Type: Receive
    //      Message ID Mask: 0xC
    //      Message Object Flags: UMask, MXtd, MDir
    //      Message Object flag CAN_MSG_OBJ_USE_ID_FILTER enables usage
    //       of msgIDMask parameter for Message Identifier based filtering
    //      Message Data Length: "Don't care" for a Receive mailbox
    CAN_setupMessageObject(CANA_BASE, RX_MSG_OBJ_ID2, 0x371,
                           CAN_MSG_FRAME_STD, CAN_MSG_OBJ_TYPE_RX, 0xC,
                           (CAN_MSG_OBJ_USE_ID_FILTER | CAN_MSG_OBJ_NO_FLAGS), 0);

    while(1)
    {
        //
        // Read CAN message object 1 and check for new data
        //
        if (CAN_readMessage(myCAN0_BASE, 1, rxMsgData))
        {
            rxMsgCount1++;
        }
        //
        // Read CAN message object 2 and check for new data
        //
        else if (CAN_readMessage(myCAN0_BASE, 2, rxMsgData))
        {
            rxMsgCount2++;
        }
    }
```

Figure 8-1. Reception with DCAN

8.2.2 MCAN Reception

1. Configure Filter Element Size (total number), Rx Buffer size and Rx FIFO size as well as the Element Size for both Buffer and FIFO. Element Size can be configured based on the estimated data size per frame. These steps are completed as part of the Message RAM configuration. Configure Filter Elements which includes setting the Message IDs / filtering conditions desired, along with configuring where the matching frame for each corresponding Filter Element is stored (among Rx Buffer and Rx FIFO 0/1).
2. For each received frame, the module checks against filter elements (standard or extended, depending on the received frame) in ascending order. On getting the first match, the frame is stored as configured into the filter element. Non-matching frames can also be configured to be stored in Rx FIFO 0/1.
3. Either by polling or using interrupts, ascertain the reception of new data. For polling, there is a bit corresponding to each possible Rx Buffer Element in the registers *MCAN_NDAT1* and *MCAN_NDAT2*. Consequently, for a new message in Rx FIFO, the *MCAN_RXFxS.FxFL* bits can be checked to get the fill level. *For using Interrupts, the procedure has been outlined in the corresponding section.*
4. Use *Driverlib* APIs to read the data from the received frame.

8.3 Filter Elements

Filter Elements are defined structures, which need to be configured in the Message RAM to determine which frames are to be received and where these frames need to be stored within the Message RAM.

Standard Filter Elements are used to store Standard ID frames, and the module can be configured to have up to 128 elements. Extended Filter Elements are used to store Extended ID frames, and the module can be configured to have up to 64 elements. The structure for both Standard and Extended Filter Elements is identical except for the Message ID type. The following description given for Standard Filter Elements is valid for Extended Filter Elements as well.

The module has certain Global Filter Configurations (set in the *MCAN_GFC* register during initialization) which determines whether to accept or reject remote frames and non-matching frames (independent configurations for both Standard and Extended IDs).

Each received frame is sequentially compared to the list of configured Filter Elements (Standard ID frames are compared to Standard Filter Elements and so on). On getting a match, based on the configuration of the corresponding filter element, the frame is accepted or rejected, and is stored in the Message RAM as configured (if accepted).

Note:

MCAN has a separate register (*MCAN_XIDAM*) that can be used as an Extended ID "AND" Mask. By default, the register (mask) has all bits set to one, which disables the mask.

However, during initialization, on enabling the mask, all received extended IDs are ANDed with this mask before the filter list is executed. This register is intended for masking of 29-bit IDs in SAE J1939.

By setting Extended Filter Type (*eft)* = 0x3 for a particular Extended Filter Element, a Range Filter can be implemented such that the Extended ID AND Mask is not applied.

8.3.1 Filter Element Structure

Standard (or Extended) Filter Elements are defined by the following fields:

- *sft* (or *eft*) determines what kind of filter is to be implemented.
- *sfec* (or *efec*) determines where the accepted frame is to be stored or if the frame is to be rejected.
- *sfid1* and *sfid2* (or *efid1* and *efid2*) determine which message IDs are matching.

The individual functions can vary based on the filter type as shown in Table 8-1 and Table 8-2:

Table 8-1. Standard Filter Element Parameters

Parameter	Description
Standard Filter Type (SFT)	0x0 : Range Filter from SFID1 to SFID2 (SFID1<= SFID2) 0x1 : Dual ID Filter for SFID1 or SFID2 0x2 : Classic Filter: SFID1 = Filter; SFID2 = Mask 0x3 : Filter Element Disabled
Standard Filter Element Configuration (SFEC)	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 Rx FIFO 0, if filter matches 0x6 : Set Priority and store in Rx FIFO 1, if filter matches 0x7 : Store in Rx Buffer, ignore SFT [1:0] field

Table 8-2. Extended Filter Element Parameters

Parameter	Description
Extended Filter Type (EFT)	0x0 : Range Filter from EFID1 to EFID2 (EFID1<= EFID2) 0x1 : Dual ID Filter for EFID1 or EFID2 0x2 : Classic Filter: EFID1 = Filter; EFID2 = Mask 0x3 : Range Filter from EFID1 to EFID2 (EFID1<= EFID2), XIDAM mask not applied
Extended Filter Element Configuration (EFEC)	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 Rx FIFO 0, if filter matches 0x6 : Set Priority and store in Rx FIFO 1, if filter matches 0x7 : Store in Rx Buffer, ignore EFT field

An example for setting up Standard Filter Elements is shown below:

If the application requires filter configuration such that

- Frame with Message ID = 0x04 must be stored in Rx Buffer Element 5 (buffer element range is from 0 to 63)
- Frame with Message IDs 0x371, 0x375, 0x379, 0x37D must be stored in Rx FIFO 0
- Frame with Message IDs 0xF4 and 0x23 must be rejected
- Frame with Message IDs in the range [0x734 to 0x75A] must be stored in Rx FIFO 1

In that case, the Standard Filter Elements to be added is shown in Table 8-3 :

Table 8-3. Standard Filter Element Configuration

Filter Element Number *(filtNum)*	Standard Filter Type *(sft)*	Standard Filter Element Configuration *(sfec)*	Standard Filter ID 1 *(sfid1)*	Standard Filter ID 2 *(sfid2)*
0	xx = Don't care	111 = Store in Rx Buffer	0x04	0x05
1	10 = Classic Bit Mask Filter	001 = Store in Rx FIFO 0	0x371 (filter)	0x0C (mask)
2	01 = Dual ID	011 = Reject	0xF4	0x23
3	00 = Range Filter	010 = Store in Rx FIFO 1	0x734	0x75A

When accessing any Standard Filter Element, the address is the starting address initialized to the register (*MCAN_SIDFC.FLSSA*) during Message RAM configuration plus the word size of the filter element times the index of the filter element. However, while evaluating any received frame against the filter list, the module only checks filters up to the number initialized to the register (*MCAN_SIDFC.LSS*) during Message RAM Configuration.

Note: Ensure that the Filter Element index does not exceed the initialized value (*MCAN_SIDFC.LSS*); otherwise the filter element reference can malfunction.

```
1
2 // Assuming that Message RAM has been configured
3
4     MCAN_StdMsgIDFilterElement stdFiltelem;
5
6     //
7     // Initialize Rx Buffer Configuration parameters.
8     //
9     stdFiltelem.sfid2              = 0x5U;  // Standard Filter ID 2.
10    stdFiltelem.sfid1              = 0x4U;  // Standard Filter ID 1.
11    stdFiltelem.sfec               = 0x7U;  // Store into Rx Buffer
12    stdFiltelem.sft                = 0x0U;  // Configuration ignored as SFEC[2:0] = 111
13
14    //
15    // Configure Standard ID filter element 0
16    //
17    MCAN_addStdMsgIDFilter(MCANA_DRIVER_BASE, 0U, &stdFiltelem);
18
19    //
20    // Initialize Rx Buffer Configuration parameters.
21    //
22    stdFiltelem.sfid2              = 0xCU;  // Standard Filter ID 2.
23    stdFiltelem.sfid1              = 0x371U;// Standard Filter ID 1.
24    stdFiltelem.sfec               = 0x1U;  // Store into Rx FIFO 0
25    stdFiltelem.sft                = 0x2U;  // Classic Bit Mask Filter
26
27    //
28    // Configure Standard ID filter element 1
29    //
30    MCAN_addStdMsgIDFilter(MCANA_DRIVER_BASE, 1U, &stdFiltelem);
31
```

Figure 8-2. MCAN Filter Configuration

8.4 Rx Buffer

Rx Buffer Elements are defined structures which are stored within the Message RAM. The module can be configured to have up to 64 elements.

The starting address of the Rx Buffer Section is stored in *MCAN_RXBC.RBSA* register and subsequent regions within this section are calculated based on the Rx Buffer Element number in consideration by the module.

8.4.1 Receiving in Rx Buffer

In case of filtering for Rx Buffers, a filter element can be configured to store a frame with matching ID defined by Standard ID1, in a Rx Buffer Element for which the number is defined by Standard ID2. Consequently, there has to be one Filter Element (std/ext) for each Rx Buffer Element. It is NOT possible to use any filter types to store frames in the Rx Buffer.

An interrupt can be generated when a new message is received in a dedicated Rx Buffer. There are two registers *MCAN_NDAT1* and *MCAN_NDAT2*, with a bit corresponding to each of the possible 64 Rx Buffer elements, which is set on receiving a new frame in the particular buffer element. This new message can be read from the Message RAM using *Driverlib* API after which the New Data Flag needs to be cleared. As long as the New Data Flag is set, the Rx Buffer Element does not receive new data, and the corresponding filter element is disabled.

```
// Assuming Message RAM Configuration and Filter Configuration has been completed

    MCAN_RxBufElement    rxMsg;
    MCAN_RxNewDataStatus newData;

    uint32_t bufNum;

    while(1)
    {
        //
        // Get the New Data Status.
        //
        MCAN_getNewDataStatus(MCANA_DRIVER_BASE, &newData);

        // If message is received in Rx buffer element represented by variable bufNum
        if((newData.statusLow & (1UL << bufNum)) != 0)
        {
            MCAN_readMsgRam(MCANA_DRIVER_BASE, MCAN_MEM_TYPE_BUF, bufNum,
                            0, &rxMsg);
        }

        //
        //  Clearing the NewData registers
        //
        MCAN_clearNewDataStatus(MCANA_DRIVER_BASE, &newData);
    }
```

Figure 8-3. Reception using Rx Buffer

8.5 Rx FIFO

Rx FIFO Elements are structurally identical to Rx Buffer Elements, and are also stored in the Message RAM. The module has two Rx FIFOs (Rx FIFO 0 and Rx FIFO 1) which can be individually configured to have up to 64 elements. The primary difference between Rx Buffer Elements and Rx FIFO Elements is in how the module accesses them.

The behaviour of the Rx FIFOs is determined by the *Put* and *Get* indices. These indices are maintained by the module in specific registers (*MCAN_RXFxS*). The *Put* index refers to the FIFO Element number where a newly received frame needs to be stored in the Message RAM. The *Get* index refers to the FIFO Element number from where the application needs to read the data from the Message RAM.

As a result of this structure, it is not necessary for the application to retrieve the data from a Rx Buffer Element each time a frame is received and clear the corresponding New Data Flag to receive the next matching frame in the same Rx Buffer Element. Instead, the application can read multiple received frames in one go.

The starting address for each FIFO section is stored in *MCAN_RXFxC.FxSA* register and the subsequent regions within this section are calculated based on the *Put* and *Get* indices by the module.

The *Put* index is incremented (automatically by the module) every time a new message is received into the FIFO, whereas, the *Get* index needs to be incremented by the application every time a message is read by the application. The Fill level of the FIFO, which translates to the number of messages in the FIFO to be read by the application, is determined by (*Put* Index - *Get* Index).

There are two modes for FIFOs which are differentiated on the basis of their behaviour when a new message is received when the FIFO is full. First is FIFO blocking mode, which means that when the Rx FIFO is full, no messages are stored in the Rx FIFO, unless at least one of the messages currently stored has been read by the application. In case a new message is received, there is an interrupt flag (*MCAN_IR.RXFxL*) that is set denoting a lost message. Second is the FIFO overwrite mode, which means that when the Rx FIFO is full, the next accepted message overwrites the oldest FIFO message.

The Rx FIFO mode is set during initialization as part of the Message RAM configuration.

8.5.1 Receiving in Rx FIFO

Filter configurations to store a matching frame into Rx FIFO have been described above.

Note: The following discussion can be applied separately to either of the Rx FIFOs.

There are a number of ways to read a new message. Separate interrupts can be generated when any new message is received in a FIFO element or when the FIFO becomes full (size of the FIFO set during Message RAM configuration). To avoid losing data on account of the FIFO being full, it is also possible to set a watermark (during Message RAM configuration). When the FIFO fill level reaches the set watermark, an interrupt is generated which can be used to read the entire FIFO (see Figure 8-4).

The new message (or messages) can be read directly from the Message RAM using *Driverlib* API after which the *Get* index needs to be incremented. This can be done by writing the index of the last element read to the register *MCAN_RXFxA*, which is done using a *Driverlib* API as shown below.

To read multiple messages from the FIFO, the same code can be called in a loop.

```
// Assuming Message RAM Configuration and Filter Configuration has been completed

    MCAN_RxBufElement rxMsg[NUM_OF_MSG], rxMsg1;
    MCAN_RxFIFOStatus RxFS;

    while(1)
    {
        RxFS.num = MCAN_RX_FIFO_NUM_1;

        MCAN_getRxFIFOStatus(MCANA_DRIVER_BASE, &RxFS);

        if((RxFS.fifoFull) != 0U)
        {
            MCAN_readMsgRam(MCANA_DRIVER_BASE, MCAN_MEM_TYPE_FIFO, 0U,
                            MCAN_RX_FIFO_NUM_1, &rxMsg1);

            rxMsg[count] = rxMsg1;
            //variable count is the array index where newest message is to be stored

            MCAN_writeRxFIFOAck(MCANA_DRIVER_BASE, MCAN_RX_FIFO_NUM_1,
                                RxFS.getIdx);
        }
    }
```

Figure 8-4. Reception with Rx FIFO

8.6 Receiving High Priority Messages

Certain Filter Elements can be configured to treat matching frames as high priority messages. Note that the messages themselves are indisinguishable (identical) from other messages, but the module reads them slightly differently. Messages can only be read from a FIFO in the order in which the messages were received. However, *priority* messages can be read directly. This is possible because there is a separate register (*MCAN_HPMS*) that stores information related to the High Priority Message including whether the message is Standard ID or Extended ID, what is the filter index for the matching filter element, in which FIFO the message is stored and the corresponding index within the FIFO.

For more information on how to receive High Priority Messages, please refer to the example available in *C2000ware*.

9 Avoiding network errors

In a properly designed/configured network, communication errors are rare. Common reasons for errors are :

1. Inadequate oscillator accuracy: It is important that the required accuracy is maintained in the entire operating temperature range of the application.
2. Improper sampling-point (SP) selection: SP must be optimal and neither too-early nor too late. The SP must be chosen based on the oscillator accuracy and propagation delay introduced by the transceivers (and any galvanic isolation, if used) and the end-to-end bus length.
3. Mismatched bit-rates between nodes: This can happen, among other things, due to inadequate oscillator tolerance.
4. Electromagnetic interference (EMI): If the noise is transient, the bus recovers on its own once the disturbances vanish. That is how the protocol is designed.

Note that bus-off is a severe error condition. You must investigate the root-cause of the errors as explained above.

10 References

- Texas Instruments: *How Signal Improvement Capability Unlocks the Real Potential of CAN-FD Transceivers*
- Texas Instruments: *Programming Examples and Debug Strategies for the DCAN Module*
- Texas Instruments: *Getting Started with the MCAN (CAN FD) 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 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