Application Note

DCAN 至 MCAN 迁移指南

Sahil Deshpande and Hareesh Janakiraman *C2000 Microcontrollers*

摘要

C2000 实时 MCU 具有三种类型的控制器局域网 (CAN) 模块：eCAN、DCAN 和 MCAN。eCAN 和 DCAN 仅支持传统 CAN，而 MCAN 同时支持传统 CAN 和 CAN FD。诸如 TMS320F2838xD、TMS320F2838xS、TMS320F28003x 和 TMS320F280015x 等器件同时具有 DCAN 和 MCAN 模块。一些 C2000 器件仅具有 MCAN 模块，因为它同时支持传统 CAN 和 CAN FD。尽管上述所有 CAN 模块均符合 CAN 协议标准，但它们彼此之间软件不兼容。具体来说，DCAN 和 MCAN 模块采用完全不同的架构，因此寄存器和位结构也不同。这就要求在模块之间采用完全不同的编程方法。本文档旨在轻松实现从 DCAN 到 MCAN 模块的迁移，讨论了常见操作（例如模块初始化、位时序配置、消息 RAM 配置、缓冲器和 FIFO 配置、数据发送、接收（带过滤）和错误处理），并介绍了如何在 DCAN 和 MCAN 模块中完成这些操作。代码片段根据需要显示。

内容

商标

1 引言

在任何给定器件上，C2000 MCU 通常只具有一种类型的 CAN 模块。例如，eCAN 或 DCAN。当在 C2000 系列引入 MCAN 时，一些 MCU 同时具有 DCAN 和 MCAN。这需要用户了解两种完全不同类型的 CAN 模块并对其进行编程。为克服这一困难，已选择 MCAN 作为未来的 CAN 平台，因为 MCAN 同时支持传统 CAN 和 CAN FD。本文档列出了 DCAN 和 MCAN 模块之间的主要差异。然后，本文档继续着重介绍如何在这两个模块中执行常见操作。

要确定给定 C2000 MCU 具有哪个 CAN 模块，请参阅 C2000 实时控制 MCU 外设指南。

2 DCAN 和 MCAN 之间的主要差异

与传统 CAN 相比，CAN FD 具有两个显著优势：

- 数据段的比特率更快，提高了整体吞吐量。应用可以通过设置 *CCCR.BRSE* = 0 选择以相同的比特率发送整个帧。这样，应用仍然可以利用 CAN FD 的更高有效负载能力。
- 与传统 CAN（多达 8 字节）相比，有效负载大小更高（多达 64 字节），从而减少了协议开销。

请注意，经典 CAN 和 CAN FD 在收发器、总线终端等方面的物理层要求是相同的。如果 CAN FD 中的数据段需要更高的比特率，则必须使用专为此类比特率设计的收发器。

表 2-1 从使用和编程的角度重点介绍了 DCAN 和 MCAN 模块之间的主要差异。

表 2-1. DCAN 和 MCAN 特性差异

功能	DCAN	MCAN
比特率	整个帧使用固定比特率	可以使用两种比特率：用于*标称段*的较慢比特率和用于*数据段*的较快比特率
发送速度	上限为 1Mbps	最高 1Mbps 可用于*标称段*，而最高 5Mbps 可用于*数据段*
每帧发送的字节数（有效负载能力）	可以发送从 0 到 8 的任意数量的字节	除 0 至 8 字节外，还可以发送 12/16/20/24/32/48/64 数据字节
数据存储元素的命名规则	数据存储在消息对象中。消息对象有时也称为邮箱。	数据存储在与*过滤器元素*相关的缓冲器中
数据存储元素的数量	固定为 32（无论要发送或接收的字节数是多少）	缓冲器的数量是灵活的（具体取决于元素的配置）
CRC 字段长度	15 位	15、17 或 21 位 CRC
时间戳支持	否	有
发送器延迟补偿	不需要	在数据段实现更快的比特率时需要

3 模块初始化

对于 DCAN 和 MCAN 模块，前几个初始化步骤是相同的。可以通过以下方式进入初始化模式：软件（分别设置 *CAN_CTL.INIT* 和 *MCAN_CCCR.INIT* 位）、硬件复位、进入总线关闭状态；或者在使用 MCAN 的情况下，在消息 RAM 中检测到未校正的位错误。在此状态下，消息传输停止，CANTX 输出驱动为隐性状态（高电平），并且错误计数器保持不变。设置 *INIT* 位不会更改任何配置寄存器。

为完成软件初始化，可以将 *INIT* 位复位，并且在出现 11 个隐性位序列（总线空闲状态）之后，可以开始通信。

下面显示了每个模块的模块初始化的分步过程。

3.1 DCAN 初始化

1. 初始化消息 RAM
2. 配置位时序
3. 配置消息对象（可选 → 也可以在初始化后和模块正常运行时执行）。

3.2 MCAN 初始化

1. 配置消息 RAM（请参阅节 5）。

2. 配置 CAN 模式（传统 CAN 或 CAN FD）。
3. 配置位时序（请参阅节 4）。
4. 配置比特率切换（启用或禁用）。
5. 配置过滤器元素（可选 → 也可以在初始化后和模块运行时执行）。

请注意，在 MCAN 中切换到 *init* 模式时，与 Tx/Rx 相关的状态寄存器会复位。

3.3 初始化序列

表 3-1 中列出了初始化 DCAN 和 MCAN 模块的各个步骤以及主要差异：

表 3-1. DCAN/MCAN 初始化序列

运行	DCAN	MCAN
进入初始化模式	设置 *CAN_CTL.INIT* 位	设置 *MCAN_CCCR.INIT* 位并检查该位是否已设置
解锁受保护的寄存器	设置 *CAN_CTL.CCE* 位	设置 *MCAN_CCCR.CCE* 位
配置 CAN 模式和比特率切换	不适用	设置 CAN FD 功能的 *MCAN_CCCR.FDOE* 位 设置 *MCAN_CCCR.BRSE* 位以启用比特率切换 (BRS) （对于传统 CAN 通信，这两个位都需要为 0）
配置位时序	配置 *CAN_BTR* 寄存器	配置 *MCAN_NBTP* 寄存器
配置数据位时序	不适用	配置 *MCAN_DBTP* 寄存器（对于传统 CAN 不需要，因为 BRS 已禁用）
消息 RAM 配置	不适用	请参阅消息 RAM 配置
全局过滤器配置（如果需要）（确定模块如何处理不匹配的帧）。	不适用	设置 *MCAN_GFC* 寄存器
接收和发送配置（也可以在运行时执行）	设置消息对象	滤波器配置
锁定受保护的寄存器	清除 *CAN_CTL.CCE* 位	清除 *MCAN_CCCR.CCE* 位
使模块恢复正常运行	清除 *CAN_CTL.INIT* 位	清除 *MCAN_CCCR.INIT* 位

除了上面显示的步骤之外，对于 MCAN，可能还需要在初始化过程中设置 MCAN 时钟分频器。此配置通常通过 AUXCLKDIVSEL 寄存器执行（请参阅特定于器件的 TRM 以确定用于时钟分频的寄存器）。对于 120MHz 和 200MHz 器件，*C2000ware* 示例将 MCAN 位时钟配置为 40MHz。如果应用需要更小的时间量子 (TQ)，则可以采用其他位时钟配置。不过，需要相应地更改标称位时序和数据位时序的参数。图 3-1 显示了 DCAN 的初始化步骤。图 3-2、图 3-3 和图 3-4 显示了 MCAN 的初始化步骤。

3.4 模块初始化代码片段

以下各图展示了模块初始化代码片段。

```
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);
}
```

图 3-1. DCAN 初始化

```
//
// 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();
}
```

图 3-2. MCAN GPIO 和时钟初始化

```
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
    //***********
```

图 3-3. MCAN 工作模式和全局过滤器配置

```
    //
    // 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));
```

图 3-4. MCAN 初始化完成

4 位时序配置

传统 CAN 和 CAN FD 之间的位时序配置不同。在传统 CAN 中，该过程*相对*更简单，因为整个帧的比特率是相同的。然而，在 CAN FD 中，可以使用两种不同的比特率：较慢的“标称”比特率和较快的“数据”比特率。在 MCAN 模块中，可通过在模块初始化期间分别写入 *MCAN_NBTP* 和 *MCAN_DBTP* 寄存器来配置这两种比特率。请注意，应用可以选择仅利用可在 CAN FD 中发送的每帧更多的数据字节数，并对整个帧使用相同的比特率。数据段更快的比特率还可保证发送器延迟补偿 (TDC)，如果没有该补偿，可能发生位错误。下面是一个计算位时序参数的示例：

示例 1

假设 200MHz 的 CAN 模块时钟需要以下参数：

标称比特率 = 500kbps，数据比特率 = 2Mbps。

计算比特率的公式如下：

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

对于 500kbps 的标称比特率，*比特率预分频器 x 位时间* 的乘积必须等于 400。只要不违反 CAN 协议规定的规则，这就可以通过预分频器和位时间的多种组合来实现。例如，可以选择预分频器 20 和位时间 20TQ。预分频器为 20 (NBRP$_{reg}$ = 19) 可产生 10MHz 的位时钟，得到的时间量子 (TQ) 为 100ns。通过 TSEG1 和 TSEG2 的多种组合，可以实现 20TQ 的位时间，从而产生不同的采样点 (SP)。

位时间 = (NTSEG1$_{reg}$ + 1) + (NTSEG2$_{reg}$ + 1) + 1，其中 NTSEG1$_{reg}$ 和 NTSEG2$_{reg}$ 分别表示写入 *MCAN_NBTP.NTSEG1* 和 *MCAN_NBTP.NTSEG2* 位字段的实际值。如果 TSEG1 选择为 16 (NTSEG1$_{reg}$ = 15) 并且 TSEG2 选择为 4 (NTSEG2$_{reg}$ =3)，这些值会产生采样点 80%。通过调整 TSEG1 和 TSEG2 值，可以根据网络参数在位时间内移动采样点。

类似的计算用于 2Mbps 的数据比特率。对于 2Mbps 的数据比特率，*比特率预分频器 x 位时间* 的乘积必须等于 100。只要不违反 CAN 协议规定的规则，这就可以通过预分频器和位时间的多种组合来实现。例如，可以选择预分频器 5 和位时间 20TQ。预分频器为 5 (DBRP$_{reg}$ = 4) 可产生 40MHz 的位时钟，得到的时间量子 (TQ) 为 25ns。通过 TSEG1 和 TSEG2 的多种组合，可以实现 20TQ 的位时间，从而产生不同的采样点 (SP)。

位时间 = (DTSEG1$_{reg}$ + 1) + (DTSEG2$_{reg}$ + 1) + 1，其中 DTSEG1$_{reg}$ 和 DTSEG1$_{reg}$ 分别表示写入 *MCAN_DBTP.DTSEG1* 和 *MCAN_DBTP.DTSEG2* 位字段的实际值。如果 TSEG1 选择为 16 (DTSEG1$_{reg}$ = 15) 并且 TSEG2 选择为 4 (NTSEG2$_{reg}$ =3)，这些值会产生采样点 80%。通过调整 TSEG1 和 TSEG2 值，可以根据网络参数在位时间内移动采样点。

```
1     //
2     // Initialize bit timings.
3     //
4     bitTimes.nomRatePrescalar    = 0xBU; // Nominal Baud Rate Pre-scaler
5     bitTimes.nomTimeSeg1         = 0x2U; // Nominal Time segment before SP
6     bitTimes.nomTimeSeg2         = 0x0U; // Nominal Time segment after SP
7     bitTimes.nomSynchJumpWidth   = 0x0U; // Nominal SJW
8     bitTimes.dataRatePrescalar   = 0x1U; // Data Baud Rate Pre-scaler
9     bitTimes.dataTimeSeg1        = 0xAU; // Data Time segment before SP
10    bitTimes.dataTimeSeg2        = 0x2U; // Data Time segment after SP
11    bitTimes.dataSynchJumpWidth = 0x2U; // Data SJW
12
13    // The above is just an illustrative example. You must compute the timing values
14    // based on your network paramters such as oscillator accuracy, propagation delay
15    // introduced by the transceivers and the bus.
16
```

图 4-1. MCAN 位时序配置

5 消息 RAM 配置

在 DCAN 中，消息 RAM 只能由消息处理程序进行访问。*Driverlib* API 与消息接口 (IFx) 寄存器交互，该寄存器使用消息 RAM 执行读取或写入操作。在 MCAN 中，*Driverlib* API 可用于直接使用消息 RAM 执行读取或写入操作。

消息 RAM 结构在 DCAN 和 MCAN 中不同。在 DCAN 中，消息 RAM 中的消息对象数量固定为 32 个，每个消息对象都可配置为用于发送或接收操作。

但是，在 MCAN 中，消息 RAM 可以配置为具有以下各段：

- 标准过滤器元素
- 扩展过滤器元素
- Rx 缓冲器
- Rx FIFO
- Tx 缓冲器
- Tx FIFO 或 Tx 队列
- Tx 事件 FIFO

MCAN 消息 RAM 的设计提供了极大的灵活性，支持根据应用需要将可用内存分配至上述每个段。这些段可以按任何方式排序，并且可以为未使用的段分配零内存。请注意，消息 RAM 大小可能因器件而异。有关更多信息，请参阅特定于器件的数据表。

消息 RAM 配置涉及定义以下内容：

- 使用的每个段的起始地址。
- 每个段中的元素数量。
- 对于不同大小的数据包，元素的大小是不同的，如表 5-1 所示（过滤器元素和 Tx 事件 FIFO 具有固定大小）。

这些值写入特定的寄存器，随后由消息处理程序和 *Driverlib* API 用来与消息 RAM 进行交互。因此，消息 RAM 配置是 MCAN 在模块初始化期间的关键步骤，而 DCAN 中不需要此类配置。MCAN 模块以 32 位字对消息 RAM 进行寻址。因此，所有段的大小都是 32 位字的倍数。

表 5-1. 元素大小与数据包大小

MCAN_RXESC.RBDS/ MCAN_RXESC.F0DS/ MCAN_RXESC.F1DS/ MCAN_TXESC.TBDS （分别对应于 Rx 缓冲器、Rx FIFO 和 Tx 缓冲器）	数据字段（字节）	FIFO 元素大小（或）缓冲器元素大小 [RAM 字]
000	8	4
001	12	5
010	16	6
011	20	7
100	24	8
101	32	10
110	48	14
111	64	18

C2000ware 示例中提供了宏，当用户设置元素的数量和大小时，宏会自动计算每个段的起始地址。此配置可以成功地用于任何应用。可以使用多种有效配置，而不存在单一“正确的”配置。请注意，MCAN 模块不会检查是否有无效的配置。用户有责任验证各个段不会相互重叠或超出可用 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)))

```

图 5-1. MCAN 消息 RAM 宏

```

    //
    // 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.
```

图 5-2. MCAN 消息 RAM 初始化

6 中断处理

从 CPU 级别（PIE、IFR 和 INTM）来看，DCAN 和 MCAN 之间的中断处理是相同的。但是，中断处理在模块级别有很大不同。表 6-1 总结了 DCAN 和 MCAN 模块之间中断处理的基本差异：

表 6-1. DCAN 和 MCAN 中的中断处理

类别	DCAN	MCAN
中断源	与每个消息对象相对应的错误、状态和发送/接收中断	30 个内部中断源（在下表中指定）
全局中断寄存器	用于启用、读取和清除存在的全局中断的寄存器	对应的寄存器不存在
配置接收中断	可以根据需要，通过设置每个消息对象中的 RxIE 位来单独启用接收中断	可以针对专用 Rx 缓冲器中接收的任何新消息启用或禁用中断。
确定接收中断的源	从寄存器 CAN_INT 读取的值对应于已接收到消息的消息对象编号	中断仅表示 Rx 缓冲器中已接收到新消息。从 MCAN_NDATx 寄存器读取的值对应于已接收到消息的 Rx 缓冲器元素编号。
Rx FIFO 中断	不支持单独的中断功能	其他中断源可用，包括 FIFO 中的新消息、FIFO 已满和 FIFO 达到水线（可在消息 RAM 配置期间配置水线，以便在 FIFO 填充到特定水平时生成中断来满足应用需求）
配置发送中断	可以根据需要，通过设置每个消息对象中的 TxIE 位来单独启用发送中断	可以通过配置寄存器 MCAN_TXBTIE 来单独启用发送中断，其中每个位对应一个单独的 Tx 缓冲器元素。
确定发送中断的源	从寄存器 CAN_INT 读取的值对应于已发送消息的消息对象编号	中断仅表示发送已完成。从 MCAN_TXBTO 寄存器读取的值对应于已发送消息的 Tx 缓冲器元素编号。

6.1 MCAN 中断源

表 6-2 中介绍了 MCAN 的不同中断源：

表 6-2. MCAN 中断源

中断	说明
ARA	访问保留地址
PED	数据段中的协议错误
PEA	仲裁段中的协议错误
WDI	看门狗
BO	总线关闭
EW	警告状态
EP	错误认可
ELO	错误记录溢出
BEU	位错误未校正
BEC	位错误已校正
DRX	存储到专用 Rx 缓冲器的消息
TOO	发生超时
MRAF	消息 RAM 访问失败
TSW	时间戳绕回
TEFL	Tx 事件 FIFO 元素丢失
TEFF	Tx 事件 FIFO 已满
TEFW	Tx 事件 FIFO 达到水线

表 6-2. MCAN 中断源（续）

中断	说明
TEFN	Tx 事件 FIFO 新条目
TFE	Tx FIFO 为空
TCF	完成发送取消
TC	完成发送
HPM	高优先级消息
RF1L	Rx FIFO 1 消息丢失
RF1F	Rx FIFO 1 已满
RF1W	Rx FIFO 1 达到水线
RF1N	Rx FIFO 1 新消息
RF0L	Rx FIFO 0 消息丢失
RF0F	Rx FIFO 0 已满
RF0W	Rx FIFO 0 达到水线
RF0N	Rx FIFO 0 新消息

6.2 DCAN 中断处理

器件级中断配置：

1. 初始化 PIE 和 PIE 向量表。启用全局和实时中断。
2. 在 PIE 向量表中配置中断处理程序。在中断控制器中启用中断。

模块级中断配置

1. 使用 CAN 控制寄存器 (CAN_CTL) 启用错误和状态中断。在单独设置消息对象时启用消息对象中断。
2. 选择要使用寄存器 (CAN_IP_MUX21) 路由每个消息对象中断的中断线路，其中每个位对应于单个消息对象。
3. 中断服务例程 (ISR)：读取中断寄存器 (CAN_INT) 以确定中断源（状态/错误/特定消息对象）。通过写入 CAN 错误和状态寄存器 (CAN_ES) 或通过清除相应消息对象中的 *IntPnd* 位来清除中断。清除相应中断线路的全局中断标志。
4. 通过 PIEACK 响应中断。

```
//
// 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);

}
```

图 6-1. DCAN 中断初始化

```
//
// 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);
}
```

图 6-2. DCAN 中断处理

6.3 MCAN 中断处理

器件级中断配置：

1. 初始化 PIE 和 PIE 向量表。启用全局和实时中断。
2. 在 PIE 向量表中配置中断处理程序。在中断控制器中启用中断。

模块级中断配置

1. 使用寄存器 (MCAN_IR) 启用中断源，其中每一位对应一个中断源。根据需要使用寄存器 (MCAN_ILE) 启用中断线路。
2. 使用寄存器 (MCAN_ILS) 选择要在其中路由中断源的中断线路，其中每个位对应一个中断源。
3. 中断服务例程 (ISR)：读取中断寄存器 (MCAN_IR) 以确定中断的源（30 个独立中断源中的任何一个）。通过写入同一寄存器来清除中断。通过写入寄存器 (MCANSS_EOI) 来清除中断线路。
4. 通过 PIEACK 响应中断。

```

//
// Function Prototype.
//
static void MCANIntrConfig(void);
__interrupt void MCANIntr1ISR(void);

{
    //
    // ISR Configuration.
    //
    MCANIntrConfig();

   // Enable Interrupts.
    // (interrupts can be enabled individually by modifying the second parameter)
    //
    MCAN_enableIntr(MCANA_DRIVER_BASE, MCAN_INTR_MASK_ALL, 1U);

    //
    // Select Interrupt Line.
    // (interrupt line can be individually chosen for each interrupt source)
    //
    MCAN_selectIntrLine(MCANA_DRIVER_BASE, MCAN_INTR_MASK_ALL, MCAN_INTR_LINE_NUM_1);

    //
    // Enable Interrupt Line.
    //
    MCAN_enableIntrLine(MCANA_DRIVER_BASE, MCAN_INTR_LINE_NUM_1, 1U);

    //
    // Enable Transmission interrupt.
    // (Needs to be done individually for each Tx Buffer Element)
    // (Additionally, transmission related interrupt sources will need to be enabled above,
    // as desired by application)
    MCAN_txBufTransIntrEnable(MCANA_DRIVER_BASE, 1U, 1U);
}

//
// This function will configure X-BAR for MCAN interrupts.
//
static void MCANIntrConfig(void)
{

    Interrupt_initModule();
    Interrupt_initVectorTable();

    Interrupt_register(INT_MCANA_1,&MCANIntr1ISR);
    Interrupt_enable(INT_MCANA_1);

    Interrupt_enableGlobal();
}
```

图 6-3. MCAN 中断初始化

```
//
// 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);
}
```

图 6-4. MCAN 中断处理

7 发送数据

DCAN 和 MCAN 模块之间的整体发送过程基本相同。差异主要源于消息 RAM 的布局和使用方式。此外，MCAN 帧可以更长并采用两种不同的比特率。经过适当配置后，该模块就会负责比特率切换和处理更大的有效负载。此时不需要代码干预。

7.1 基本发送过程

本节概述的流程包括在使用 DCAN 和 MCAN 发送帧的过程中所需的操作和涉及的寄存器。

7.1.1 使用 DCAN 发送

1. 设置发送消息对象。
2. 写入 IFx 寄存器，而这些寄存器依次将消息 ID (ARBID)、DLC 和数据写入消息对象（并更新消息 ID，如有必要）。
3. 设置 IFx 寄存器 (*CAN_IFxCMD*) 中的 *TXRQST* 位，以表示消息对象已准备好发送。
4. 当总线空闲时，消息处理程序解析准备好发送的消息对象，并发送可用的最高优先级消息。

7.1.2 使用 MCAN 发送

1. 初始化发送缓冲器元素（定义为结构）。
2. 将 Tx 消息写入消息 RAM。
3. 在 *MCAN_TXBAR* 寄存器中设置与 Tx 缓冲器元素编号相对应的位（每个位代表一个单独的缓冲器元素），以表示消息已准备好发送。
4. 当总线空闲时，消息处理程序解析准备好发送的缓冲器元素，并发送可用的最高优先级消息。

7.2 MCAN 与 DCAN 发送过程差异

尽管 DCAN 和 MCAN 发送的概念过程基本相同，但表 7-1 显示了这两个模块之间的关键差异：

表 7-1. MCAN 与 DCAN 发送过程

类别	DCAN	MCAN
发送优先级	首先发送编号最低的消息对象（已准备好发送）	首先发送包含编号最低的消息 ID 的缓冲器（包括已准备好发送的缓冲器）
缓冲类型	只发送消息对象	发送缓冲器可配置为专用 Tx 缓冲器、Tx FIFO 或 Tx 队列
写入/更新发送消息	要求写入 IFx 寄存器	可以通过使用 *Driverlib* API 直接写入消息 RAM 来更新发送消息

```
//
// 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);

```

图 7-1. 使用 DCAN 发送

7.3 MCAN 发送概念

本节概述 MCAN 中的其他特性。

- 每条 Tx 消息都可以配置为在传统 CAN 或 CAN-FD 模式下发送
- 传输暂停
- 发送取消
- Tx FIFO/Tx 队列

在消息 RAM 内，Tx 缓冲器空间可以具有以下可能的配置：

1. 仅 Tx 缓冲器
2. Tx 缓冲器 + Tx FIFO
3. Tx 缓冲器 + Tx 队列

下面的表 7-2 显示了如何配置其中每一个段：

表 7-2. 各种 Tx 缓冲器选项的消息 RAM 配置

Tx 缓冲器	Tx 缓冲器 + Tx FIFO	Tx 缓冲器 + Tx 队列
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*)

下面的表 7-3 指定了每个段的功能差异和潜在用例：

表 7-3. TX 缓冲器与 Tx FIFO 与 Tx 队列特性比较

功能	Tx 缓冲器	Tx FIFO	Tx 队列
主机 (CPU) 直接可用的信息	缓冲器元素编号已知。	只能从寄存器 (*MCAN_TXFQS*) 读取 *Put* 和 *Get* 索引	只能从寄存器 (*MCAN_TXFQS*) 读取 *Put* 和 *Get* 索引
首先发送的元素	具有最低消息 ID 的元素	最早的元素	具有最低消息 ID 的元素
Put 索引/*Get* 索引	不适用	*Put 索引* 指向存储最近一帧的位置。随*添加发送请求* 而递增。*Get 索引* 指向下一个要发送的最早元素。	*Put 索引* 指向最低的空闲缓冲器元素（队列内），其中存储最近的帧。通过*添加发送请求* 进行更新。 *GET 索引* 始终为零
使用同一 ID 发送多条消息	发送缓冲器编号最低的元素	发送最早的元素	发送缓冲器编号最低的元素
完整条件	不适用	如果 FIFO 已满，除非请求的发送完成，否则无法写入消息	如果队列已满，除非请求的发送完成，否则无法写入消息
Tx 取消	可实现	无法实现	无法实现
用例	优点是应用知道哪个消息 ID 存储在哪个缓冲器元素中，因此可以在发送之前进行编辑	必须按特定顺序（而不是按消息 ID 升序）发送帧的应用	优点是缓冲器编号由 *Put 索引* 自动处理。应用不需要根据消息 ID 优先级跟踪哪个缓冲器为空

```
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
```

图 7-2. 使用 MCAN 发送

7.3.1 Tx 事件 FIFO

Tx 事件 FIFO 是存储在消息 RAM 中的已定义结构。模块可配置为具有多达 32 个元素。

虽然 Tx 缓冲器仅保存要发送的消息，但可以使用 Tx 事件 FIFO 单独来存储发送状态（包括消息 ID 和时间戳）。消息标记从 Tx 缓冲器复制到 Tx 事件 FIFO 元素，以将 Tx 事件链接到 Tx 事件 FIFO 元素。

这在具有动态管理的发送队列的应用中非常有用，在这种应用中，可以在成功发送后立即使用新消息覆盖 Tx 缓冲器，而无需保存来自 Tx 缓冲器本身的发送状态。有关如何存储 Tx 事件 FIFO 元素的更多信息，请参阅 *C2000ware* 中提供的示例。

8 接收数据

与发送类似，DCAN 和 MCAN 之间的接收也基本相同。差异主要源于消息 RAM 的布局和使用方式。此外，MCAN 帧可以更长并采用两种不同的比特率。

8.1 接收简介

在 DCAN 消息 RAM 中，有 32 个可配置的消息对象可用于发送或接收。接收消息对象用于存储接收到的数据。如果应用需要，可以为一个或多个消息对象启用接受过滤。CPU 对消息 RAM 的读写访问通过三个接口寄存器 (IFx) 集来完成。

在 MCAN 中，消息 RAM 可以划分为多个段，以包括过滤器元素、Rx 缓冲器元素和 Rx FIFO 元素。过滤器元素可配置为与接受过滤一起使用，还可以确定相应的匹配帧在消息 RAM 中的存储位置。Rx 缓冲器和 Rx FIFO 是存储接收到的帧的段，两者都有自己的一组寄存器和中断。这种结构提供了灵活性，可以更好地满足不同的应用需要。可以使用 *Driverlib* API 直接读取消息 RAM。

8.2 基本接收流程

本节概括介绍了使用 DCAN 和 MCAN 进行配置和接收帧所涉及的简要流程。

8.2.1 DCAN 接收

1. 配置接收消息对象：这涉及写入消息 ID (ARBID)，并在需要时屏蔽要接收的帧。
2. 对于每个接收到的帧，模块将按升序对照接收消息对象进行检查。当第一次匹配时，帧存储在相应的消息对象中。
3. 通过轮询或使用中断，确定新数据的接收。对于轮询，寄存器 *CAN_NDAT_21* 中的每个接收消息对象都有一个对应的位。对于使用中断，相应章节中已概述了该过程。
4. 使用其中一个 IFx 寄存器从接收的帧中读取数据。

```
1
2 #define RX_MSG_OBJ_ID1     1
3 #define RX_MSG_OBJ_ID2     2
4
5 uint16_t rxMsgData[4];
6
7     // Initialize the receive message object used for receiving CAN messages.
8     // Message Object Parameters:
9     //      Message Object ID Number: 1
10    //      Message Identifier: 0x4
11    //      Message Frame: Standard
12    //      Message Type: Receive
13    //      Message ID Mask: 0x0
14    //      Message Object Flags: Receive Interrupt
15    //      Message Data Length: 4 Bytes (Note that DLC field is a "don't care"
16    //      for a Receive mailbox
17    //
18    CAN_setupMessageObject(CANA_BASE, RX_MSG_OBJ_ID1, 0x4, CAN_MSG_FRAME_STD,
19                           CAN_MSG_OBJ_TYPE_RX, 0, CAN_MSG_OBJ_RX_INT_ENABLE,
20                           0);
21    //
22    // Initialize the receive message object used for receiving CAN messages.
23    // Possible flags: CAN_MSG_OBJ_NO_FLAGS, CAN_MSG_OBJ_USE_EXT_FILTER,
24    //                 CAN_MSG_OBJ_USE_DIR_FILTER
25    // Message Object Parameters:
26    //      Message Object ID Number: 2
27    //      Message Identifier: 0x371
28    //      Message Frame: Standard
29    //      Message Type: Receive
30    //      Message ID Mask: 0xC
31    //      Message Object Flags: UMask, MXtd, MDir
32    //      Message Object flag CAN_MSG_OBJ_USE_ID_FILTER enables usage
33    //       of msgIDMask parameter for Message Identifier based filtering
34    //      Message Data Length: "Don't care" for a Receive mailbox
35    CAN_setupMessageObject(CANA_BASE, RX_MSG_OBJ_ID2, 0x371,
36                           CAN_MSG_FRAME_STD, CAN_MSG_OBJ_TYPE_RX, 0xC,
37                           (CAN_MSG_OBJ_USE_ID_FILTER | CAN_MSG_OBJ_NO_FLAGS), 0);
38
39    while(1)
40    {
41        //
42        // Read CAN message object 1 and check for new data
43        //
44        if (CAN_readMessage(myCAN0_BASE, 1, rxMsgData))
45        {
46            rxMsgCount1++;
47        }
48        //
49        // Read CAN message object 2 and check for new data
50        //
51        else if (CAN_readMessage(myCAN0_BASE, 2, rxMsgData))
52        {
53            rxMsgCount2++;
54        }
55    }
```

图 8-1. 使用 DCAN 接收

8.2.2 MCAN 接收

1. 配置过滤器元素大小（总数）、Rx 缓冲器大小和 Rx FIFO 大小以及缓冲器和 FIFO 的元素大小。元素大小可以根据每帧的估计数据大小进行配置。这些步骤是在消息 RAM 配置过程中完成的。配置过滤器元素，其中包括设置所需的消息 ID/过滤条件，以及配置存储每个相应过滤器元素的匹配帧的位置（在 Rx 缓冲器和 Rx FIFO 0/1 之间）。
2. 对于每个接收到的帧，模块按升序检查过滤器元素（标准或扩展，取决于接收到的帧）。获得第一个匹配帧后，该帧将按照配置存储到过滤器元素中。不匹配的帧也可以配置为存储在 Rx FIFO 0/1 中。
3. 通过轮询或使用中断，确定新数据的接收。对于轮询，在寄存器 *MCAN_NDAT1* 和 *MCAN_NDAT2* 中，每个可能的 Rx 缓冲器元素都对应有一个位。因此，对于 Rx FIFO 中的新消息，可以检查 *MCAN_RXFxS.FxFL* 位以获取填充级别。*对于使用中断，相应章节中已概述了该过程。*
4. 使用 *Driverlib* API 从接收到的帧中读取数据。

8.3 过滤器元素

过滤器元素是定义的结构，需要在消息 RAM 中对其进行配置，以确定要接收哪些帧以及需要将这些帧存储在消息 RAM 中的什么位置。

标准过滤器元素用于存储标准 ID 帧，模块可以配置为具有多达 128 个元素。扩展过滤器元素用于存储扩展 ID 帧，模块可以配置为具有多达 64 个元素。标准过滤器元素和扩展过滤器元素的结构相同，但消息 ID 类型除外。以下对于标准过滤器元素的描述也适用于扩展过滤器元素。

模块具有某些全局过滤器配置（在初始化期间在 *MCAN_GFC* 寄存器中设置），用于确定是要接受还是要拒绝远程帧和不匹配帧（对于标准 ID 和扩展 ID 使用独立的配置）。

每个接收到的帧都会按顺序与配置的过滤器元素列表进行比较（标准 ID 帧与标准过滤器元素进行比较，等等）。在获得匹配项时，将根据相应过滤器元素的配置接受或拒绝帧，并按照配置的方式将其存储在消息 RAM 中（如果接受）。

注意：

MCAN 有一个单独的寄存器 (*MCAN_XIDAM*)，可用作与扩展 ID 进行“与”运算的掩码。默认情况下，寄存器（掩码）的所有位均设置为 1，这会禁用掩码。

然而，在初始化期间，启用掩码时，在执行过滤器列表之前，所有接收到的扩展 ID 都与该掩码进行“与”运算。该寄存器用于屏蔽 SAE J1939 中的 29 位 ID。

通过为特定扩展过滤器元素设置扩展过滤器类型 (*eft)* = 0x3，可以实现范围过滤器，从而不应用扩展 ID 和掩码的“与”运算。

8.3.1 过滤器元素结构

标准（或扩展）过滤器元素由以下字段定义：

- *sft*（或 *eft*）确定要实现哪种过滤器。
- *sfec*（或 *efec*）确定要存储接受的帧的位置或是否要拒绝该帧。
- *sfid1* 和 *sfid2*（或 *efid1* 和 *efid2*）确定哪些消息 ID 匹配。

各个函数可能根据过滤器类型而有所不同，如表 8-1 和表 8-2 所示：

表 8-1. 标准过滤器元素参数

参数	说明
标准过滤器类型 (SFT)	0x0：从 SFID1 到 SFID2 的范围过滤器 (SFID1<= SFID2) 0x1：用于 SFID1 或 SFID2 的双 ID 过滤器 0x2：传统过滤器：SFID1 = 过滤器；SFID2 = 掩码 0x3：过滤器元素已禁用
标准过滤器元素配置 (SFEC)	0x0：禁用过滤器元素 0x1：如果过滤器匹配，则存储在 Rx FIFO 0 中 0x2：如果过滤器匹配，则存储在 Rx FIFO 1 中 0x3：如果过滤器匹配，则拒绝 ID 0x4：如果过滤器匹配，则设置优先级 0x5：如果过滤器匹配，则设置优先级并存储在 Rx FIFO 0 中 0x6：如果过滤器匹配，则设置优先级并存储在 Rx FIFO 1 中 0x7：存储在 Rx 缓冲器中，忽略 SFT [1:0] 字段

表 8-2. 扩展过滤器元素参数

参数	说明
扩展过滤器类型 (EFT)	0x0：从 EFID1 到 EFID2 的范围过滤器 (EFID1<= EFID2) 0x1：用于 EFID1 或 EFID2 的双 ID 过滤器 0x2：传统过滤器：EFID1 = 过滤器；EFID2 = 掩码 0x3：从 EFID1 到 EFID2 的范围过滤器 (EFID1<= EFID2)，未应用 XIDAM 掩码
扩展过滤器元素配置 (EFEC)	0x0：禁用过滤器元素 0x1：如果过滤器匹配，则存储在 Rx FIFO 0 中 0x2：如果过滤器匹配，则存储在 Rx FIFO 1 中 0x3：如果过滤器匹配，则拒绝 ID 0x4：如果过滤器匹配，则设置优先级 0x5：如果过滤器匹配，则设置优先级并存储在 Rx FIFO 0 中 0x6：如果过滤器匹配，则设置优先级并存储在 Rx FIFO 1 中 0x7：存储在 Rx 缓冲器中，忽略 EFT 字段

下面显示了一个设置标准过滤器元素的示例：

如果应用需要过滤器配置，以便

- 消息 ID = 0x04 的帧必须存储在 Rx 缓冲器元素 5 中（缓冲器元素范围为 0 到 63）
- 消息 ID 为 0x371、0x375、0x379、0x37D 的帧必须存储在 Rx FIFO 0 中
- 必须拒绝消息 ID 为 0xF4 和 0x23 的帧
- 消息 ID 在 [0x734 至 0x75A] 范围内的帧必须存储在 Rx FIFO 1 中

在这种情况下，要添加的标准过滤器元素如表 8-3 所示：

表 8-3. 标准过滤器元素配置

过滤器元素编号 *(filtNum)*	标准过滤器类型 *(sft)*	标准过滤器元素配置 *(sfec)*	标准过滤器 ID 1 *(sfid1)*	标准过滤器 ID 2 *(sfid2)*
0	xx = 不用考虑	111 = 存储在 Rx 缓冲器中	0x04	0x05
1	10 = 传统位掩码过滤器	001 = 存储在 Rx FIFO 0 中	0x371（过滤器）	0x0C（掩码）
2	01 = 双 ID	011 = 拒绝	0xF4	0x23
3	00 = 范围过滤器	010 = 存储在 Rx FIFO 1 中	0x734	0x75A

当访问任何标准过滤器元素时，地址是在消息 RAM 配置期间初始化到寄存器 (*MCAN_SIDFC.FLSSA*) 的起始地址加上过滤器元素的字大小乘以过滤器元素的索引。然而，当根据过滤器列表评估任何接收到的帧时，模块检查的过滤器至多仅为在消息 RAM 配置期间初始化到寄存器 (*MCAN_SIDFC.LSS*) 的数量。

注意：确保过滤器元素索引不超过初始化的值 (*MCAN_SIDFC.LSS*)；否则过滤器元素基准可能会出现问题。

```
// Assuming that Message RAM has been configured

    MCAN_StdMsgIDFilterElement stdFiltelem;

    //
    // Initialize Rx Buffer Configuration parameters.
    //
    stdFiltelem.sfid2              = 0x5U;  // Standard Filter ID 2.
    stdFiltelem.sfid1              = 0x4U;  // Standard Filter ID 1.
    stdFiltelem.sfec               = 0x7U;  // Store into Rx Buffer
    stdFiltelem.sft                = 0x0U;  // Configuration ignored as SFEC[2:0] = 111

    //
    // Configure Standard ID filter element 0
    //
    MCAN_addStdMsgIDFilter(MCANA_DRIVER_BASE, 0U, &stdFiltelem);

    //
    // Initialize Rx Buffer Configuration parameters.
    //
    stdFiltelem.sfid2              = 0xCU;  // Standard Filter ID 2.
    stdFiltelem.sfid1              = 0x371U;// Standard Filter ID 1.
    stdFiltelem.sfec               = 0x1U;  // Store into Rx FIFO 0
    stdFiltelem.sft                = 0x2U;  // Classic Bit Mask Filter

    //
    // Configure Standard ID filter element 1
    //
    MCAN_addStdMsgIDFilter(MCANA_DRIVER_BASE, 1U, &stdFiltelem);

```

图 8-2. MCAN 过滤器配置

8.4 Rx 缓冲器

Rx 缓冲器元素是存储在消息 RAM 中的已定义结构。模块可配置为具有多达 64 个元素。

Rx 缓冲器段的起始地址存储在 *MCAN_RXBC.RBSA* 寄存器中，该段中的后续区域根据模块考虑的 Rx 缓冲器元素编号进行计算。

8.4.1 在 Rx 缓冲器中接收

在对 Rx 缓冲器进行过滤的情况下，过滤器元素可以配置为将具有由标准 ID1 定义的匹配 ID 的帧存储在 Rx 缓冲器元素（其编号由标准 ID2 定义）中。因此，每个 Rx 缓冲器元素都必须具有一个过滤器元素（标准/扩展）。无法使用任何过滤器类型将帧存储在 Rx 缓冲器中。

当在专用 Rx 缓冲器中接收到新消息时，可能会生成中断。存在两个寄存器 *MCAN_NDAT1* 和 *MCAN_NDAT2*，可能的 64 个 Rx 缓冲器元素中的每一个元素都有一个对应的位，这是在特定缓冲器元素中接收到新帧时设置的。可以使用 *Driverlib* API 从消息 RAM 读取这个新消息，之后需要清除新数据标志。只要设置了新数据标志，Rx 缓冲器元素就不会接收新数据，并且禁用相应的过滤器元素。

```
// 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);
    }
```

图 8-3. 使用 Rx 缓冲器接收

8.5 Rx FIFO

RX FIFO 元素在结构上与 Rx 缓冲器元素相同，并且也存储在消息 RAM 中。模块有两个 Rx FIFO（Rx FIFO 0 和 Rx FIFO 1），它们可单独配置为具有多达 64 个元素。Rx 缓冲器元素和 Rx FIFO 元素之间的主要区别在于模块访问它们的方式。

Rx FIFO 的行为由 *Put* 和 *Get* 索引决定。这些索引由模块在特定寄存器 (*MCAN_RXFxS*) 中维护。*Put* 索引是指需要将新接收的帧存储在消息 RAM 中的 FIFO 元素编号。*GET* 索引是指应用需要从消息 RAM 读取数据的 FIFO 元素编号。

由于这种结构，应用不需要在每次接收帧时都从 Rx 缓冲器元素检索数据，也不需要清除相应的新数据标志来在相同的 Rx 缓冲器元素中接收下一个匹配帧。相反，应用可以一次读取多个接收到的帧。

每个 FIFO 段的起始地址存储在 *MCAN_RXFxC.FxSA* 寄存器中，而该段中的后续区域根据模块的 *Put* 和 *Get* 索引进行计算。

每次在 FIFO 中接收到新消息时，*Put* 索引都会递增（由模块自动执行），而每次应用读取消息时，*Get* 索引都需要由应用来递增。FIFO 的填充级别（即 FIFO 中应用要读取的消息数）由（*Put* 索引 - *Get* 索引）确定。

FIFO 有两种模式，可根据 FIFO 已满时接收到新消息时的行为进行区分。第一种是 FIFO 阻塞模式，这意味着当 Rx FIFO 已满时，Rx FIFO 中不会存储任何消息，除非当前存储的至少一条消息已被应用读取。如果收到新消息，则会设置一个中断标志 (*MCAN_IR.RXFxL*)，表示消息丢失。第二种是 FIFO 覆盖模式，这意味着当 Rx FIFO 已满时，下一条接受的消息将覆盖最早的 FIFO 消息。

Rx FIFO 模式在初始化期间作为消息 RAM 配置的一部分进行设置。

8.5.1 在 Rx FIFO 中接收

上面介绍了将匹配帧存储到 Rx FIFO 中的过滤器配置。

注意：以下讨论可以单独应用于其中任何一个 Rx FIFO。

可以通过多种方法来读取新消息。当在 FIFO 元素中接收到任何新消息或 FIFO 已满（在消息 RAM 配置期间设置的 FIFO 大小）时，可能生成单独的中断。为避免由于 FIFO 已满而导致丢失数据，还可以设置水线（在消息 RAM 配置期间）。当 FIFO 填充级别达到设置的水线时，将生成一个中断，此中断可用于读取整个 FIFO（请参阅图 8-4）。

可以使用 *Driverlib* API 直接从消息 RAM 读取新消息（一条或多条），之后 *Get* 索引需要递增。这可以通过将最后读取的元素的索引写入寄存器 *MCAN_RXFxA* 来实现，这是使用 *Driverlib* API 完成的，如下所示。

要从 FIFO 读取多条消息，可以循环调用相同的代码。

```
// 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);
        }
    }
```

图 8-4. 使用 Rx FIFO 接收

8.6 接收高优先级消息

某些过滤器元素可以配置为将匹配帧视为高优先级消息。请注意，消息本身与其他消息是无法区分的（相同），但模块读取它们的方式略有不同。只能按照接收消息的顺序从 FIFO 读取消息。但是，可以直接读取*优先级*消息。这是可以实现的，因为有一个单独的寄存器 (*MCAN_HPM*) 存储与高优先级消息相关的信息，包括消息是标准 ID 还是扩展 ID、匹配过滤器元素的过滤器索引是什么、消息存储在哪个 FIFO 中以及 FIFO 内的相应索引。

有关如何接收高优先级消息的更多信息，请参阅 *C2000ware* 中提供的示例。

9 避免网络错误

在正确设计/配置的网络中，极少出现通信错误。错误的常见原因有：

1. 振荡器精度不足：在应用的整个工作温度范围内保持所需的精度非常重要。
2. 采样点 (SP) 选择不当：SP 必须是更佳的，既不过早也不过晚。SP 的选择必须基于振荡器精度、收发器引入的传播延迟（和任何电隔离，如使用）以及端到端总线长度。
3. 节点之间的比特率不匹配：造成这种情况的可能原因之一是振荡器容差不足。
4. 电磁干扰 (EMI)：如果噪声是瞬态的，则一旦干扰消失，总线便会自行恢复。这就是该协议的设计方式。

请注意，总线关闭是一种严重的错误情况。您必须按照上述说明调查错误的根本原因。

10 参考资料

- 德州仪器 (TI)：*信号改善功能如何释放 CAN-FD 收发器的真正潜力*
- 德州仪器 (TI)：*DCAN 模块的编程示例和调试策略*
- 德州仪器 (TI)：*MCAN (CAN FD) 模块入门*

重要声明和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、某特定用途方面的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他功能安全、信息安全、监管或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的应用。严禁对这些资源进行其他复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款或 ti.com 上其他适用条款/TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265