M460 的 CAN FD 使用不同方式发送消息

NuMicro® 32 位系列微控制器范例代码介绍

文件信息

应用简述	本范例代码将演示M460的CAN消息几种不同发送方式产生的不同发送效果
BSP 版本	M460_Series_BSP_CMSIS_V3.00.004
开发平台	3-In-1 Analyzer Tool

The information described in this document is the exclusive intellectual property of Nuvoton Technology Corporation and shall not be reproduced without permission from Nuvoton.

Nuvoton is providing this document only for reference purposes of NuMicro microcontroller and microprocessor based system design. Nuvoton assumes no responsibility for errors or omissions.

All data and specifications are subject to change without notice.

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

1. 概述

M460 的 CAN FD 的通讯速率相比较 CAN2.0 有了较大的提升，并且优化了数据的收发方式。消息接收提供了 2 个 FIFO 的消息缓冲区。后续文中均将 CAN FD 简称为 CAN。

消息发送提供了 5 种发送模式：

- 单 BUFFER
- 单 FIFO
- 单 QUEUE
- BUFFER + FIFO
- BUFFER + QUEUE

消息发送同时支持发送事件记录，本次例程不介绍发送事件功能。BUFFER 模式是最简单常用的一种发送方式，本例程不再重点叙述。例程将重点演示其他 4 种发送方式的差异。

例程的 0x1800 个字节的 RAM 定义如下表 1-1：

定义	功能	字节数	数量	RAM 地址
SID_BUFF_SIZE	标准帧 ID 过滤器	4 bytes	20	0
XID_BUFF_SIZE	扩展帧 ID 过滤器	8 bytes	16	0x50
TX_EVENT_SIZE	发送事件 FIFO	12 bytes	32	0xD0
TX_BUFF_SIZE	发送暂存区	8+64 bytes	32	0x250
RX_FIFO_SIZE	接收暂存区	8+64 bytes	45	0xB50

表 1-1 RAM 分配

1.1 CAN 消息接收 FIFO

消息接收设置了 2 个接收 FIFO，每个 FIFO 最大设置为接收 64 条消息。每个 FIFO 可以设置不同的消息过滤，这样可以通过硬件方式快速的将接收到的消息进行分类。

FIFO 遵循先进先出的方式读取，收到新的数据会添加到 FIFO 尾部。接收示意如图 1-1：

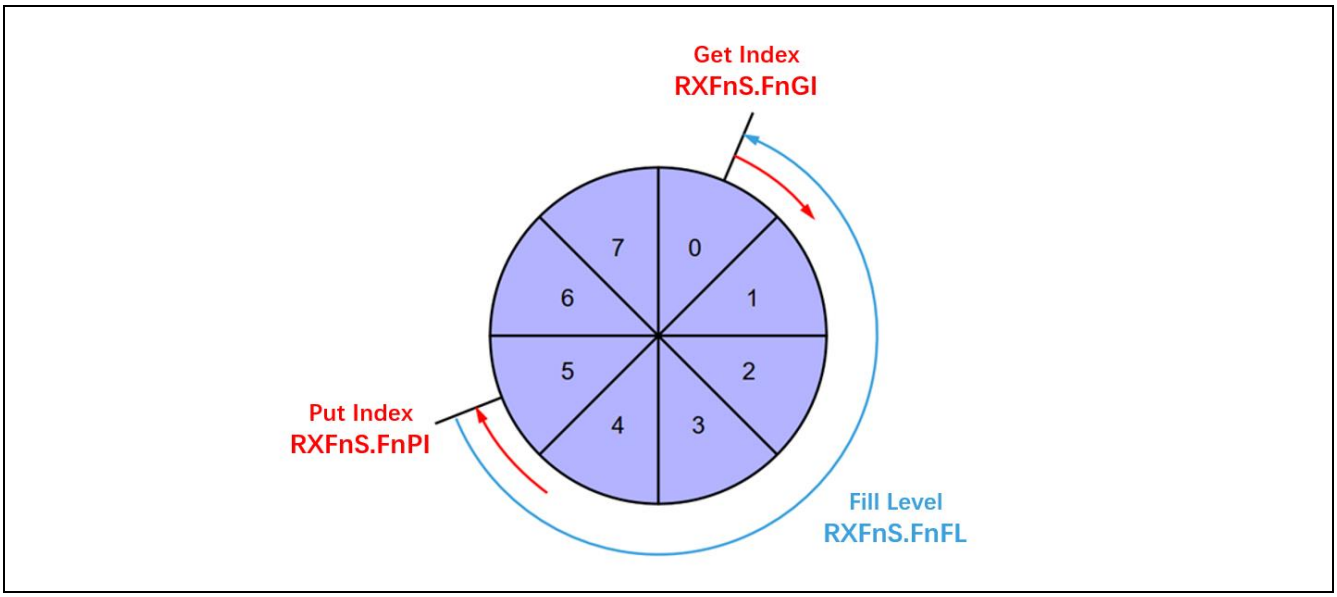


图 1-1 先入先出队列

1.2 CAN 消息发送

M460 可以定义 32 个消息发送缓冲，每个缓冲可以配置一个消息发送事件，事件可以记录消息的发送情况。根据不同的发送情况进行业务逻辑处理。

1.2.1 BUFFER 发送方式

当消息被填入 BUFFER 发送 RAM 暂存区，并且使能对应发送位后，M460 会自动将所有 BUFFER 中的使能发送位的消息按照 ID 号从小到大的顺序依次发送出去。

这种发送方式可以将常用的命令类消息先准备好，当需要发送的时候只要使能发送位，数据就能发送出去，不需要再次进行消息内容的填写。

1.2.2 FIFO/QUEUE 发送方式

FIFO 和 QUEUE 的发送方式均使用了内部的一个先进先出的消息队列，当消息一个一个依次使能发送的时候两者的使用方式一样，但在多个消息同时被使能发送的时候，发送消息的顺序就会有差异。

多个消息同时使能发送时两者的差异如下表 1-2：

Type	FIFO	QUEUE
写入顺序	消息写入队尾	消息写入队尾
发送顺序	优先发送队首消息	优先发送 ID 号最小的消息
相同 ID 发送顺序	按照 FIFO 顺序	优先发送最先请求的消息

表 1-2 FIFO/QUEUE 发送差异

例程结尾会有实例进行观测分析。

1.2.3 BUFFER+FIFO/QUEUE 发送方式

该模式下结合了 BUFFER 和 FIFO/QUEUE 发送方式的特点，可以更加灵活的处理发送消息的先后顺序。

混合模式下的发送先后顺序如下表 1-3：

组合模式	发送排序	相同 ID 发送排序
BUFFER+FIFO	优先发送所有专用 Tx 缓冲区中具有最小 ID 的元素和 Tx FIFO 中最早的元素	按照 FIFO 顺序
BUFFER+QUEUE	优先发送所有专用 Tx 缓冲区和 Tx 队列中具有最小 ID 的元素	优先发送最先请求的消息

表 1-3 组合模式发送差异

1.3 测试方法

1.3.1 数据准备

定义 3 种类型各 10 个变量：

- tx_msg[10]：

用于存放 CAN 消息，每个消息仅发送 1 个字节的数据，这 1 个字节的数据按照发送 RAM 的填入使能顺序进行排序。每个 tx_msg 按照数组的序号都各自对应一个 id_dat 和 buff_dat。

- id_dat[10] ：

用于存放 CAN 的标准 ID 号，10 个 id_data 中乱序存放了 0x100~0x109 的标准 ID 号。

- buff_dat[10] ：

用于存放消息在 RAM 种对应的存放位置，即 32 个发送元素的位置。10 个 buff_dat[10]中乱序存放了消息的存放位置，用于指定消息存放到 BUFFER 中还是 FIFO 或者 QUEUE 中。

1.3.2 初始化发送配置

M460 有 4 个 CAN，每个 CAN 为其配置不同的发送方式，发送相同的数据。不处于发送状态的 CAN 会接收到发出的消息，cangroo 软件会按照总线接收到的先后顺序依次显示收到的消息。

CAN0 ：32 个发送元素全部配置成 FIFO 模式。

CAN1 ：32 个发送元素全部配置成 QUEUE 模式。

CAN2 ：10 个发送元素配置成 BUFFER 模式，剩下 22 个元素配置成 FIFO 模式。

CAN3 ：10 个发送元素配置成 BUFFER 模式，剩下 22 个元素配置成 QUEUE 模式。

所有 CAN 口波特率等参数均需要设置成一样，即使用默认的参数就可以。

为了方便观察数据，发送的时候只对接其中的需要发送和需要接收的 CAN 口即可。

整理了一个表格方便对比 4 个 CAN 口的差异。

表 1-4 中绿色格子表示 CAN2 和 CAN3 发送时使用了 FIFO/QUEUE 模式的消息位置。橘色格子表示使用了 BUFFER 模式进行数据的发送。

CAN 消息数据						CAN0	CAN1	CAN2	CAN3
数组序号	使能顺序	标准 ID	数据字节	数据内容	Message RAM 位置	写入区域	写入区域	写入区域	写入区域
1	1	0x109	1	0	0	FIFO	QUEUE	BUFFER	BUFFER
2	2	0x101	1	1	10	FIFO	QUEUE	FIFO	QUEUE
3	3	0x108	1	2	11	FIFO	QUEUE	FIFO	QUEUE
4	4	0x105	1	3	2	FIFO	QUEUE	BUFFER	BUFFER
5	5	0x102	1	4	12	FIFO	QUEUE	FIFO	QUEUE
6	6	0x104	1	5	5	FIFO	QUEUE	BUFFER	BUFFER
7	7	0x103	1	6	9	FIFO	QUEUE	BUFFER	BUFFER
8	8	0x107	1	7	12	FIFO	QUEUE	FIFO	QUEUE
9	9	0x106	1	8	14	FIFO	QUEUE	FIFO	QUEUE
10	10	0x100	1	9	15	FIFO	QUEUE	FIFO	QUEUE

表 1-4 消息发送配置表

2. 代码介绍

例程的代码基于上一个示例代码修改而来，关于上一个示例代码可以参考如下链接：

EC_M460_4_CAN_FD_to_SLCAN_by_VCOM_V1.00

本例程基于上一个例程修改内容如下：

- 优化了 CAN 消息的收发机制，减少一次消息格式转换，提高了消息的收发效率。
- 重新划分消息 RAM 区域的功能。
- 增加消息发送的测试函数。

测试过程需要修改 USB 设备驱动程序，重命名不同的虚拟 VCOM 名称，以便找到相应的运行设备。具体的范例可以参考如下链接：

EC_M460_HSUSBD_VCOM_MultiPort_V1.00

2.1 代码简介

收发机制的优化主要体现在对接收消息的数据的处理上，原例程每次收到消息后均会产生一次中断，并且在中断内将 RAM 数据搬移到 CAN 的消息内。数据量较大的情况下中断会占用大量 CPU 资源。优化后的例程利用 CAN 接收消息的 FIFO 接收数据，在中断程序内仅设置数据接收标志位，外部循环判断到有消息后再去读取 FIFO 中收到的数据。当系统较繁忙时 FIFO 可以先接收一部分数据，然后在空闲时一次性全部取出。*Reveive2ASCII()* 函数在 slca*n.c* 文件中，负责将接收 FIFO 中的所有数据提取出来，并直接转换成 SLCAN 的数据格式通过 USB 接口发送给 PC。

```
/**
 * @brief       Receive all CAN FD Message from Rx FIFO.
 *
 * @param[in]   slcan_port  : The pointer of the CANFD port.
 * @details     This function get all CAN ascii message from CANFD Rx RAM .
 */
void Reveive2ASCII(struct slcan_t* slcan_port)
{
    CANFD_T * psCanfd = canfd[slcan_port->candev_sn];

    CANFD_BUF_T *pRxBuffer;
    uint8_t GetIndex;
    uint32_t res = 0;

    __I  uint32_t *pRXFS;
    __IO uint32_t *pRXFC, *pRXFA;
    uint8_t msgLostBit;
```

```
    pRXFS = &(psCanfd->RXF1S);
    pRXFC = &(psCanfd->RXF1C);
    pRXFA = &(psCanfd->RXF1A);
    msgLostBit = 7;

    /* if FIFO is not empty */
    while((CANFD_ReadReg(pRXFS) & 0x7F) > 0) //Extract all messages in FIFO.
    {
        GetIndex = (uint8_t)((CANFD_ReadReg(pRXFS) >> 8) & 0x3F);
        pRxBuffer = (CANFD_BUF_T *)(CANFD_SRAM_BASE_ADDR(psCanfd) + (CANFD_ReadReg(pRXFC)
        & 0xFFFF) + (GetIndex * sizeof(CANFD_BUF_T)));

        res = slcan_buf2ascii((char *)slcan_port->uart_tx_buffer,pRxBuffer);
        slcan_port->recv_count++;
        slcan_uart_write(slcan_port, slcan_port->uart_tx_buffer, res);//Data write out

        /* we got the message */
        *pRXFA = GetIndex;

        /* check for overflow */
        if (CANFD_ReadReg(pRXFS) & CANFD_RXFS_RFL)
        {
            /* clear overflow flag */
            psCanfd->IR = (1UL << msgLostBit);
        }
    }
}
```

CAN 的 RAM 分配信息在 *canapp.h* 文件中定义了数据元素的类型，每种类型的数量，以及其在 RAM 中的开始地址。因为 RAM 大小有限，所以并不能将每个元素的数量最大化。这个需要在实际使用过程中按需分配。

```
/*-----------------------------------------------------------------------------*/
/* Message RAM elements                                                        */
/*-----------------------------------------------------------------------------*/
//Note: AUTOSAR requires at least three Tx Queue Buffers and support of transmit
cancellation

#define SID_BUFF_SIZE  20       //20 elements
#define XID_BUFF_SIZE  16       //16 elements
#define TX_EVENT_SIZE  32       //32 elements
#define TX_BUFF_SIZE   32       //32 elements
#define RX_FIFO_SIZE   45       //45 elements

/*
The maximum Message RAM size is 1536 words = 0x1800 bytes

sizeof(SID elements)       = 4 bytes
sizeof(XID elements)       = 8 bytes
sizeof(TX EVENT elements)  = 12 bytes
sizeof(TX BUFF  elements)  = 8 + 64 bytes
sizeof(RX FIFO  elements)  = 8 + 64 bytes

SID_BUFF_ADDR_OFFSET = 0
XID_BUFF_ADDR_OFFSET = SID_BUFF_SIZE                                *4      = 0x50
TX_EVENT_ADDR_OFFSET = XID_BUFF_ADDR_OFFSET  + XID_BUFF_SIZE        *8      = 0xD0
TX_BUFF_ADDR_OFFSET  = TX_EVENT_ADDR_OFFSET  + TX_EVENT_SIZE        *12     = 0x250
RX_FIFO_ADDR_OFFSET  = TX_BUFF_ADDR_OFFSET   + TX_BUFF_SIZE         *(8+64) = 0xB50
END_ADD              = RX_FIFO_ADDR_OFFSET   + RX_FIFO_SIZE         *(8+64) = 0x17F8 < 0x1800
*/

#define SID_BUFF_ADDR_OFFSET                                                (0)
#define XID_BUFF_ADDR_OFFSET                                        (SID_BUFF_SIZE*4)
```

```
#define TX_EVENT_ADDR_OFFSET            (XID_BUFF_ADDR_OFFSET + XID_BUFF_SIZE*8)
#define TX_BUFF_ADDR_OFFSET             (TX_EVENT_ADDR_OFFSET + TX_EVENT_SIZE*12)
#define RX_FIFO_ADDR_OFFSET         (TX_BUFF_ADDR_OFFSET +
TX_BUFF_SIZE*(sizeof(CANFD_BUF_T)))
```

新增加的测试函数有 2 个，2 个函数均在 *canapp.c* 文件中。

CAN 初始化函数 *CANFD_Test_Start()*，负责将 4 个 CAN 发送配置为不同的发送方式。CAN0 的发送全部配置为 FIFO 模式；CAN1 的发送全部配置为 QUEUE 模式；CAN2 的发送配置为 10 个 BUFFER + 22 个 FIFO；CAN3 的发送配置为 10 个 BUFFER + 22 个 QUEUE。

```
/**
 * @brief       CAN FD Start Test Config
 * @param[in]
 * @return
*/
void CANFD_Test_Start(void)
{
    uint8_t u8Cnt;
    CANFD_FD_T *psCanfdStr;
    DEBUG_MSG("\nStart CAN FD bus function.\n");

    SYS_ResetModule(CANFD0_RST);
    SYS_ResetModule(CANFD1_RST);
    SYS_ResetModule(CANFD2_RST);
    SYS_ResetModule(CANFD3_RST);

    for(u8Cnt=0;u8Cnt<CANFD_NUM;u8Cnt++)
    {
    //Basic settings.
    sCANFD_Config[u8Cnt].sBtConfig.sNormBitRate.u32BitRate  = DEF_NormBitRate;
    sCANFD_Config[u8Cnt].sBtConfig.sDataBitRate.u32BitRate  = DEF_DataBitRate;//1M;2M;5M
    sCANFD_Config[u8Cnt].sBtConfig.bFDEn                    = TRUE;
    sCANFD_Config[u8Cnt].sBtConfig.bBitRateSwitch           = TRUE;
    /* Disable the Internal Loopback mode */
    sCANFD_Config[u8Cnt].sBtConfig.bEnableLoopBack          = FALSE;
    /* Disable the Monitor mode */
    sCANFD_Config[u8Cnt].sBtConfig.bEnableMonitor           = FALSE;
    }

    u8Cnt = 0;psCanfdStr = &sCANFD_Config[u8Cnt];
    //CANFD0 TX buffer settings.
    sCANFD_Config[u8Cnt].sTxConfig.eDataFieldSize           = 64;
    sCANFD_Config[u8Cnt].sTxConfig.eModeSel                 = eCANFD_FIFO_MODE;
    /* Need (u32DBufNumber + u32ElemCnt)  <= TX_BUFF_SIZE */
    sCANFD_Config[u8Cnt].sTxConfig.u32DBufNumber            = 0;
    sCANFD_Config[u8Cnt].sTxConfig.u32ElemCnt               = TX_BUFF_SIZE - 0;
    CANFD_Initial(u8Cnt,psCanfdStr);

    u8Cnt = 1;psCanfdStr = &sCANFD_Config[u8Cnt];
    //CANFD1 TX buffer settings.
    sCANFD_Config[u8Cnt].sTxConfig.eDataFieldSize           = 64;
    sCANFD_Config[u8Cnt].sTxConfig.eModeSel                 = eCANFD_QUEUE_MODE;
    /* Need (u32DBufNumber + u32ElemCnt)  <= TX_BUFF_SIZE */
    sCANFD_Config[u8Cnt].sTxConfig.u32DBufNumber            = 0;
    sCANFD_Config[u8Cnt].sTxConfig.u32ElemCnt               = TX_BUFF_SIZE - 0;
```

```
    CANFD_Initial(u8Cnt,psCanfdStr);

    u8Cnt = 2;psCanfdStr = &sCANFD_Config[u8Cnt];
    //CANFD2 TX buffer settings.
    sCANFD_Config[u8Cnt].sTxConfig.eDataFieldSize           = 64;
    sCANFD_Config[u8Cnt].sTxConfig.eModeSel                 = eCANFD_FIFO_MODE;
    /* Need (u32DBufNumber + u32ElemCnt)  <= TX_BUFF_SIZE */
    sCANFD_Config[u8Cnt].sTxConfig.u32DBufNumber            = DEF_DBufNumber;
    sCANFD_Config[u8Cnt].sTxConfig.u32ElemCnt               = TX_BUFF_SIZE -
     DEF_DBufNumber;
    CANFD_Initial(u8Cnt,psCanfdStr);

    u8Cnt = 3;psCanfdStr = &sCANFD_Config[u8Cnt];
    //CANFD3 TX buffer settings.
    sCANFD_Config[u8Cnt].sTxConfig.eDataFieldSize           = 64;
    sCANFD_Config[u8Cnt].sTxConfig.eModeSel                 = eCANFD_QUEUE_MODE;
    /* Need (u32DBufNumber + u32ElemCnt)  <= TX_BUFF_SIZE */
    sCANFD_Config[u8Cnt].sTxConfig.u32DBufNumber            = DEF_DBufNumber;
    sCANFD_Config[u8Cnt].sTxConfig.u32ElemCnt               = TX_BUFF_SIZE -
     DEF_DBufNumber;
    CANFD_Initial(u8Cnt,psCanfdStr);
}
```

发送的测试函数 *CAN_TX_Test()*定义了 10 个发送的消息 *tx_msg[10]*，每个消息有各自不同的 ID。消息 ID（0x100~0x109）的顺序在 ID 数组 *id_dat[10]*中已经打乱。每个消息在 CAN 发送 RAM 中的位置也已经定义在 *buff_dat[10]*数组中。按照消息数组的顺序依次将需要发送的数据填入到发送 RAM 区域，每当填入一个数据完成后即刻触发发送使能。因为数据填入速度较快，发送较慢，只有第一个填入的数据会立即进行发送，其他的消息会按照不同发送模式的先后顺序进行发送。

```
/**
 * @brief       CAN FD send test messages.
 * @param[in]   slcan_port  : The pointer of the CANFD port.
 * @return
*/
void CAN_TX_Test(struct slcan_t* slcan_port)
{
    uint32_t msg_num,msg_cnt,dat_cnt=0;
    CANFD_T * g_pCanfd = canfd[slcan_port->candev_sn];
    CANFD_FD_MSG_T    tx_msg[10] ;                      //TX message.
    uint32_t        buff_dat[10] =
   {  0, 10, 10,  2, 10,  5,  9, 10, 10, 10};    //TX buffer ram address index.
    uint32_t          id_dat[10] =
   {0x109,0x101,0x108,0x105,0x102,0x104,0x103,0x107,0x106,0x100};  //TX message ID.
    uint32_t           buff_mask = 0;

    while(CANFD_GET_COMMUNICATION_STATE(g_pCanfd) != eCANFD_IDLE);

    //Prepare the message to send.
    for(msg_cnt = 0 ; msg_cnt < 10 ;msg_cnt++)
    {
    tx_msg[msg_cnt].u32Id          = id_dat[msg_cnt];
    tx_msg[msg_cnt].eIdType        = eCANFD_SID;
```

```
    tx_msg[msg_cnt].u32DLC          = 1;
    tx_msg[msg_cnt].bFDFormat       = 1;
    tx_msg[msg_cnt].bBitRateSwitch  = 0;
    tx_msg[msg_cnt].eFrmType        = eCANFD_DATA_FRM;

    {//The order in which the messages are filled into the TX ram.
    tx_msg[msg_cnt].au8Data[dat_cnt] = msg_cnt ;
    }

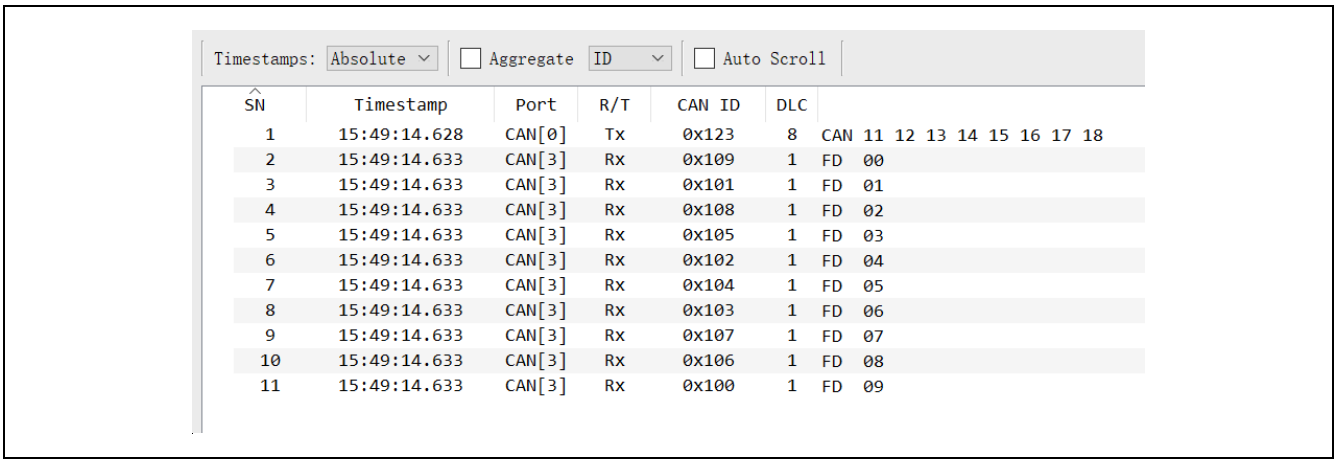
    CANFD_BUFSet(slcan_port->candev_sn, buff_dat[msg_cnt],tx_msg + msg_cnt);

    //The buffer area needs to add a send request.
    if(buff_dat[msg_cnt] < sCANFD_Config[slcan_port->candev_sn].sTxConfig.u32DBufNumber)
            g_pCanfd->TXBAR =  0x01 << buff_dat[msg_cnt];
    }
}
```

2.2 运行实例

本例使用相同的波特率设置，测试 1 路 CAN 发送，另外 1 路 CAN 接收。查看接收 CAN 的收到消息的先后顺序。依次将 CAN0~CAN3 发送的消息保存下来，然后再来分析一下接收到的消息。测试固件进行了特殊处理，每一路 CAN 只需要点击 1 次 Send Once，固件就会自动发送 10 个消息。图 2-1 是 CAN0 发出的 10 个消息，通过 CAN3 来接收。“SN”代表消息收发的先后顺序，数字越小代表越早收到消息。“CAN_ID”是每个消息的 ID 号。“DLC”代表接收到消息数据的长度，后面跟的“FD”代表是 CANFD 的帧，“00~09”是每个消息接收到的消息数据。

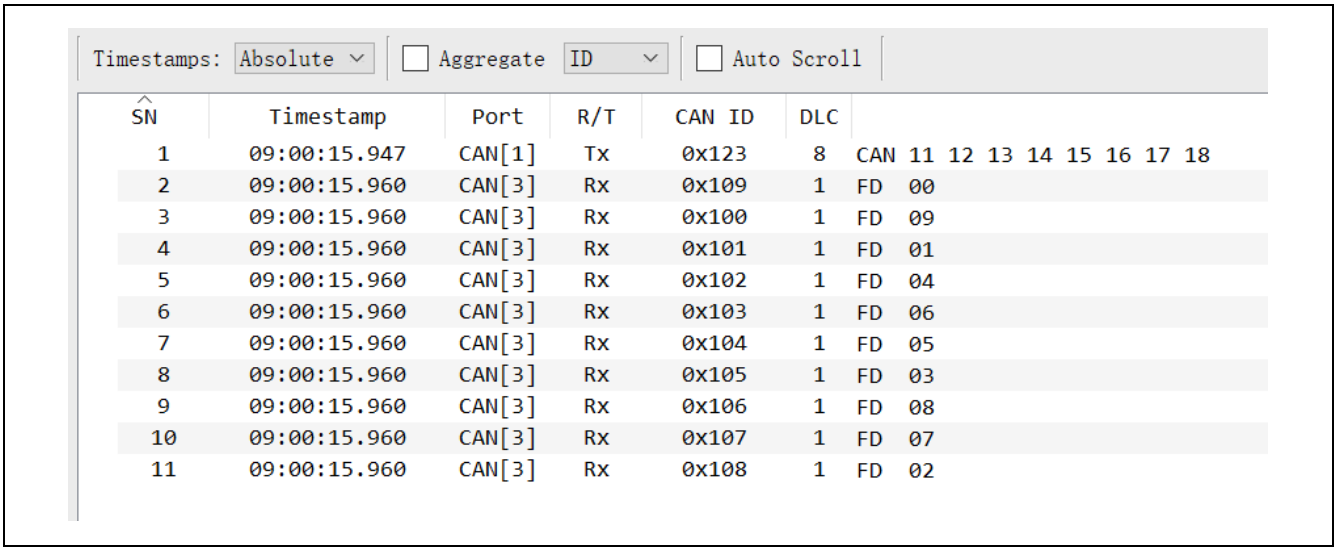
从图 2-1 上可以看出 CAN0 发出消息，消息数据是从 00 依次排到 09 的，即消息写入顺序和发出顺序是一致的。



SN	Timestamp	Port	R/T	CAN ID	DLC	
1	15:49:14.628	CAN[0]	Tx	0x123	8	CAN 11 12 13 14 15 16 17 18
2	15:49:14.633	CAN[3]	Rx	0x109	1	FD 00
3	15:49:14.633	CAN[3]	Rx	0x101	1	FD 01
4	15:49:14.633	CAN[3]	Rx	0x108	1	FD 02
5	15:49:14.633	CAN[3]	Rx	0x105	1	FD 03
6	15:49:14.633	CAN[3]	Rx	0x102	1	FD 04
7	15:49:14.633	CAN[3]	Rx	0x104	1	FD 05
8	15:49:14.633	CAN[3]	Rx	0x103	1	FD 06
9	15:49:14.633	CAN[3]	Rx	0x107	1	FD 07
10	15:49:14.633	CAN[3]	Rx	0x106	1	FD 08
11	15:49:14.633	CAN[3]	Rx	0x100	1	FD 09

图 2-1 CAN0 发出的 10 个消息

图 2-2 是 CAN1 发出的 10 个消息，第 1 个消息 ID 号为 0x109，从第 2 个开始消息的 ID 顺序是从 0x100 依次排到 0x108。

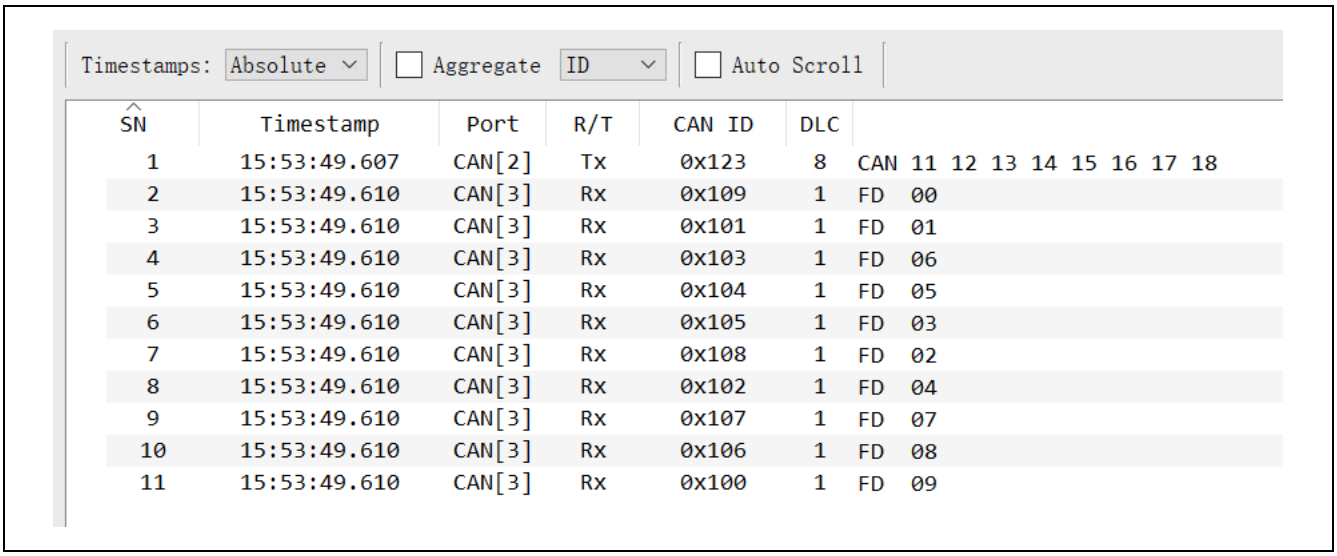


Timestamps: Absolute | Aggregate | ID | Auto Scroll

SN	Timestamp	Port	R/T	CAN ID	DLC	
1	09:00:15.947	CAN[1]	Tx	0x123	8	CAN 11 12 13 14 15 16 17 18
2	09:00:15.960	CAN[3]	Rx	0x109	1	FD 00
3	09:00:15.960	CAN[3]	Rx	0x100	1	FD 09
4	09:00:15.960	CAN[3]	Rx	0x101	1	FD 01
5	09:00:15.960	CAN[3]	Rx	0x102	1	FD 04
6	09:00:15.960	CAN[3]	Rx	0x103	1	FD 06
7	09:00:15.960	CAN[3]	Rx	0x104	1	FD 05
8	09:00:15.960	CAN[3]	Rx	0x105	1	FD 03
9	09:00:15.960	CAN[3]	Rx	0x106	1	FD 08
10	09:00:15.960	CAN[3]	Rx	0x107	1	FD 07
11	09:00:15.960	CAN[3]	Rx	0x108	1	FD 02

图 2-2 CAN1 发出的 10 个消息

图 2-3 是 CAN2 发出的 10 个消息，第 1 个消息依然是 ID 号为 0x109，其他的消息没有很明显的规律。后续会统一分析对比。

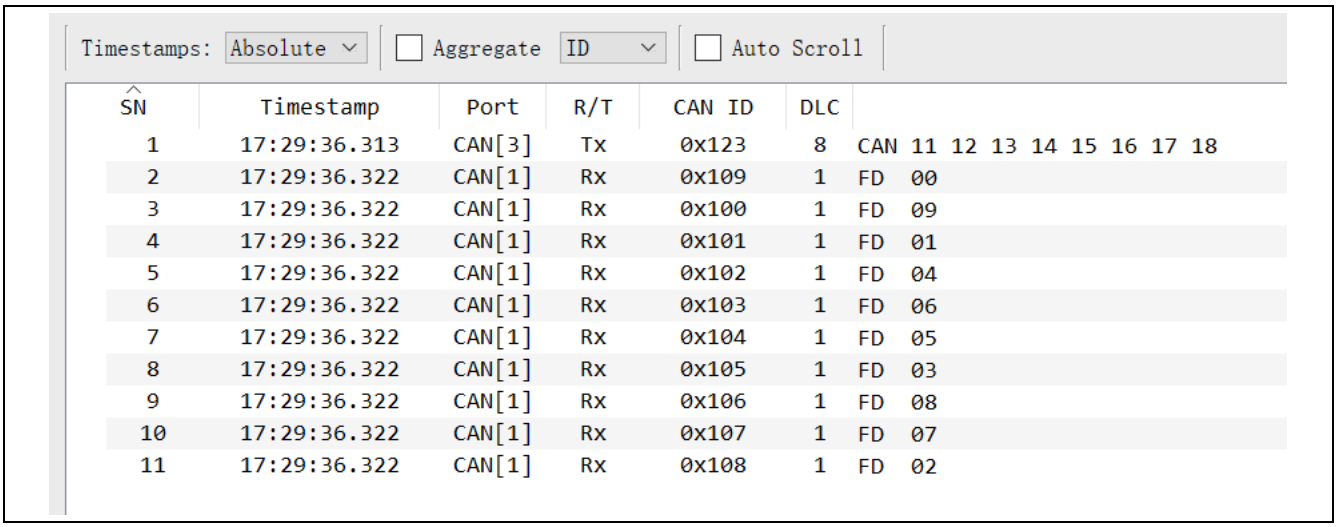


Timestamps: Absolute | Aggregate | ID | Auto Scroll

SN	Timestamp	Port	R/T	CAN ID	DLC	
1	15:53:49.607	CAN[2]	Tx	0x123	8	CAN 11 12 13 14 15 16 17 18
2	15:53:49.610	CAN[3]	Rx	0x109	1	FD 00
3	15:53:49.610	CAN[3]	Rx	0x101	1	FD 01
4	15:53:49.610	CAN[3]	Rx	0x103	1	FD 06
5	15:53:49.610	CAN[3]	Rx	0x104	1	FD 05
6	15:53:49.610	CAN[3]	Rx	0x105	1	FD 03
7	15:53:49.610	CAN[3]	Rx	0x108	1	FD 02
8	15:53:49.610	CAN[3]	Rx	0x102	1	FD 04
9	15:53:49.610	CAN[3]	Rx	0x107	1	FD 07
10	15:53:49.610	CAN[3]	Rx	0x106	1	FD 08
11	15:53:49.610	CAN[3]	Rx	0x100	1	FD 09

图 2-3 CAN2 发出的 10 个消息

图 2-4 是 CAN3 发出的 10 个消息，第 1 个消息依然是 ID 号为 0x109，从第 2 个消息开始 ID 号的顺序和 CAN1 接收到的顺序是一致的。



Timestamps: Absolute | Aggregate ID | Auto Scroll

SN	Timestamp	Port	R/T	CAN ID	DLC	
1	17:29:36.313	CAN[3]	Tx	0x123	8	CAN 11 12 13 14 15 16 17 18
2	17:29:36.322	CAN[1]	Rx	0x109	1	FD 00
3	17:29:36.322	CAN[1]	Rx	0x100	1	FD 09
4	17:29:36.322	CAN[1]	Rx	0x101	1	FD 01
5	17:29:36.322	CAN[1]	Rx	0x102	1	FD 04
6	17:29:36.322	CAN[1]	Rx	0x103	1	FD 06
7	17:29:36.322	CAN[1]	Rx	0x104	1	FD 05
8	17:29:36.322	CAN[1]	Rx	0x105	1	FD 03
9	17:29:36.322	CAN[1]	Rx	0x106	1	FD 08
10	17:29:36.322	CAN[1]	Rx	0x107	1	FD 07
11	17:29:36.322	CAN[1]	Rx	0x108	1	FD 02

图 2-4 CAN3 发出的 10 个消息

从上面 4 个 CAN 发出的消息看，第 1 个消息均为 ID 号为 0x109 的第 1 个使能发送的消息，差异体现在后续发送的 9 个消息上。

将接收到的消息按照实际接收排序整理到表 2-1 如下：

	10 个消息写入顺序				10 个消息的 ID 号			
写入区域	FIFO	QUEUE	BUFFER FIFO	BUFFER QUEUE	FIFO	QUEUE	BUFFER FIFO	BUFFER QUEUE
实际接收排序	CAN0	CAN1	CAN2	CAN3	CAN0	CAN1	CAN2	CAN3
1	1	1	1	1	0x109	0x109	0x109	0x109
2	2	10	2	10	0x101	0x100	0x101	0x100
3	3	2	7	2	0x108	0x101	0x103	0x101
4	4	5	6	5	0x105	0x102	0x104	0x102
5	5	7	4	7	0x102	0x103	0x105	0x103
6	6	6	3	6	0x104	0x104	0x108	0x104
7	7	4	5	4	0x103	0x105	0x102	0x105
8	8	9	8	9	0x107	0x106	0x107	0x106
9	9	8	9	8	0x106	0x107	0x106	0x107
10	10	3	10	3	0x100	0x108	0x100	0x108

表 2-1 接收消息整理

结合表 1-4 和表 2-1 可以看出，不同方式发送的消息，第 1 个发出的都是第 1 个被使能的 ID 号为 0x109 的消息。

从第 2 个发出的消息来分析：

- CAN0 完全遵守先入先出的发送规律，发送顺序不受 ID 号优先级的影响。
- CAN1 和 CAN3，因为 BUFFER 和 QUEUE 的发送规律是 ID 号越小，发送优先级越高，所以两者的发送顺序是一样的。
- CAN2 发送规则有 2 种，所以发送消息的顺序相比其他的会有较大的差异。每次发送的时候会先检索 BUFFER 中的 ID，将其和 FIFO 中第 1 个 ID 进行对比，发送 ID 号较小的消息。当 BUFFER 中优先级较高的 ID 发送完成后，然后再将 FIFO 中的消息依次发出。
- 表 2-1 中绿色的格子表示使用 FIFO 发送消息的 ID。CAN2 的 FIFO 第 1 个 ID（0x101）优先级比 BUFFER 中的 ID 都高，所以先进行了发送。发送完成后再次检索优先级的时候 BUFFER 中的 ID 优先级都比当前 FIFO 第 1 个 ID（0x108）要高，所以将 ID 为 0x103，0x104，0x105 消息优先发出，发出后再将剩下 FIFO 中的消息依次发出。

3. 软件与硬件需求

3.1 软件需求

- BSP 版本
  - M460_Series_BSP_CMSIS_V3.00.004
- IDE 版本
  - Keil uVision 5.37@ Arm Compiler 6
- APP 版本
  - Cangaroo V0.3.0

3.2 硬件需求

- 电路组件
  - 3-In-1 Analyzer Tool
- 线路示意图
  - 将 UART0 TX pin 脚连接到 PC UART RX，以显示范例代码的执行结果。
  - 将 HSUSB 连接到 PC，并打开 Cangaroo 上位机，以显示收发的 CAN 消息。
  - 测试的时候只需要将不同 2 个的 CAN 口相互连接。

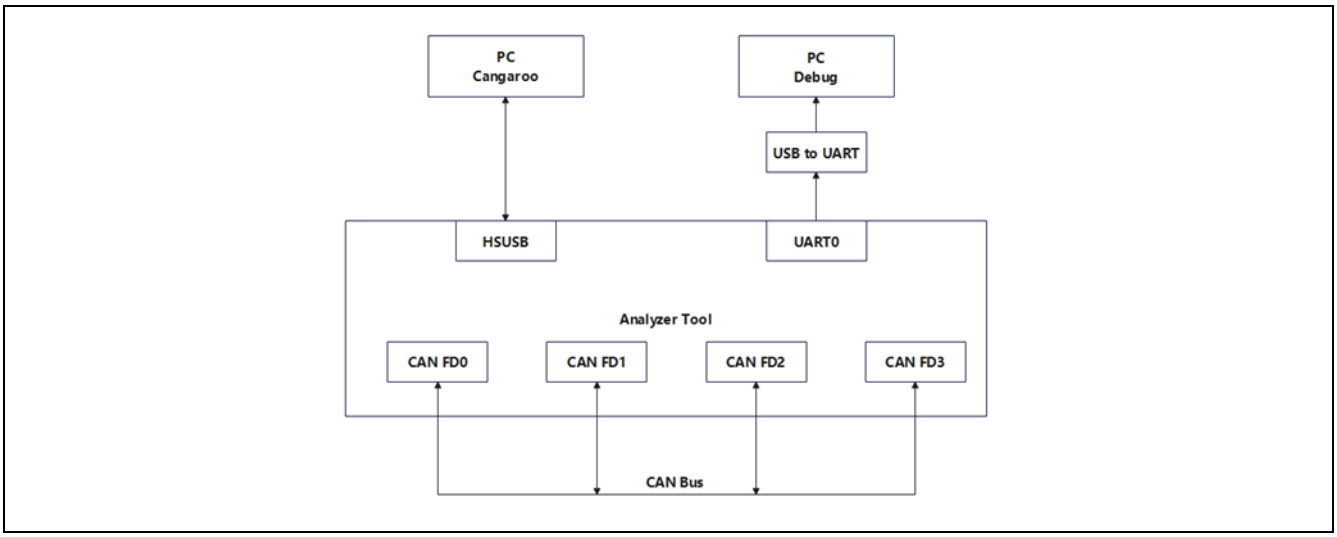


图 3-1 线路示意图

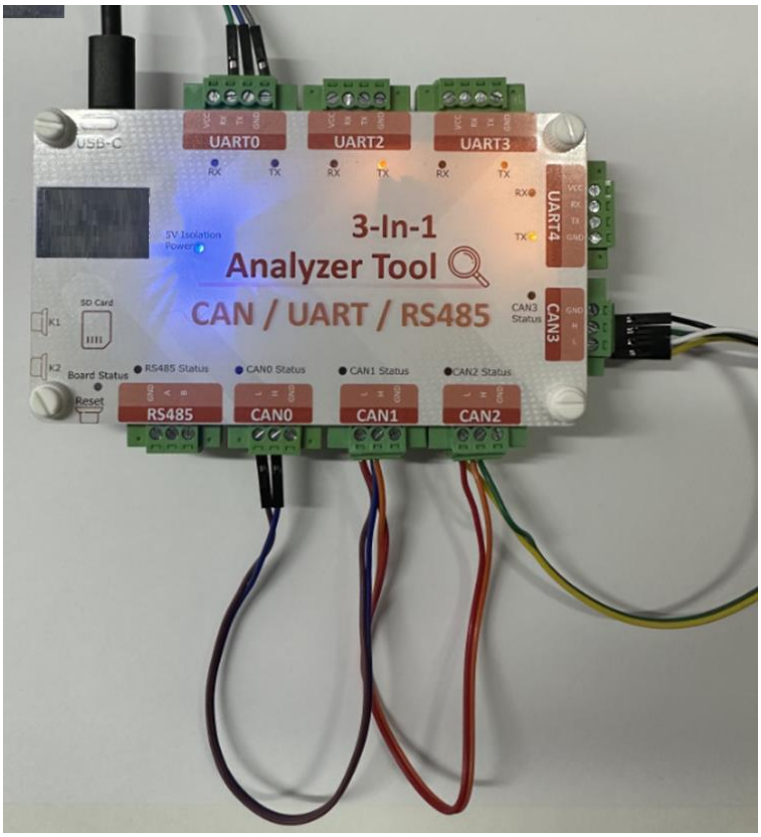

USB-C
UART0
UART2
UART3
UART4
3-In-1
Analyzer Tool
CAN / UART / RS485
5V Isolation Power
SD Card
K1
K2
Board Status
Reset
RS485 Status
CAN0 Status
CAN1 Status
CAN2 Status
CAN3 Status
RS485
CAN0
CAN1
CAN2
CAN3

图 3-2 实物接线

4. 目录信息

目录结构如下图所示。

EC_M460 CAN_FD_BUFF_FIFO_QUEUE_V1.00	
Library	Sample code header and source files
CMSIS	Cortex® Microcontroller Software Interface Standard (CMSIS) by Arm® Corp.
Device	CMSIS compliant device header file
StdDriver	All peripheral driver header and source files
SampleCode	
ExampleCode	Source file of example code

图 4-1 目录信息

5. 例程序执行

1. 根据目录信息章节进入 ExampleCode 路径中的 KEIL 文件夹，双击 *CAN_FD_BUFF_FIFO_QUEUE.uvprojx*。
2. 进入编译模式界面
   - 编译
   - 下载代码至内存
   - 进入 / 离开仿真模式
3. 进入仿真模式界面
   - 执行代码

6. 修订纪录

Date	Revision	Description
2025.09.3	1.00	初始发布。

Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.

Please note that all data and specifications are subject to change without notice.
All the trademarks of products and companies mentioned in this datasheet belong to their respective owners.