M460 的 CAN FD 使用不同方式發送消息

NuMicro® 32 位元系列微控制器範例代碼介紹

文檔資訊

應用簡述	本範例代碼將演示M460的CAN消息幾種不同發送方式產生的不同發送效果
BSP 版本	M460_Series_BSP_CMSIS_V3.00.004
開發平臺	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 的通訊速率相比較 CAN2.0 有了較大的提升，並且優化了消息的收發方式。消息接收提供了 2 個 FIFO 的訊息暫存區。後續文中均將 CAN FD 簡稱為 CAN。

消息發送提供了 5 種發送模式：

- 單 BUFFER
- 單 FIFO
- 單 QUEUE
- BUFFER + FIFO
- BUFFER + QUEUE

消息發送同時支援發送事件記錄，本次範例不介紹發送事件功能。BUFFER 模式是最簡單常用的一種發送方式，本範例不再重點敘述。範例將重點演示其他 4 種發送方式的差異。

範例的 0x1800 個位元組的 RAM 定義如下表 1-1：

定義	功能	位元組數	數量	RAM 地址
SID_BUFF_SIZE	標準幀 ID 篩檢程式	4 bytes	20	0
XID_BUFF_SIZE	擴展幀 ID 篩檢程式	8 bytes	16	0x50
TX_EVENT_SIZE	發送事件 FIFO	12 bytes	32	0xD0
TX_BUFF_SIZE	發送暫存區	8+64 bytes	32	0x250
RX_FIFO_SIZE	接收暫存區	8+64 bytes	45	0xB50

表 1-1 RAM 分配

1.1 CAN 消息接收 FIFO

消息接收設置了 2 個接收 FIFO，每個 FIFO 最大設置為接收 64 條消息。每個 FIFO 可以設置不同的消息過濾，這樣可以通過硬體方式快速的將接收到的消息進行分類。

FIFO 遵循先進先出的方式讀取，收到新的消息會添加到 FIFO 尾部。發送示意如圖 1-1：

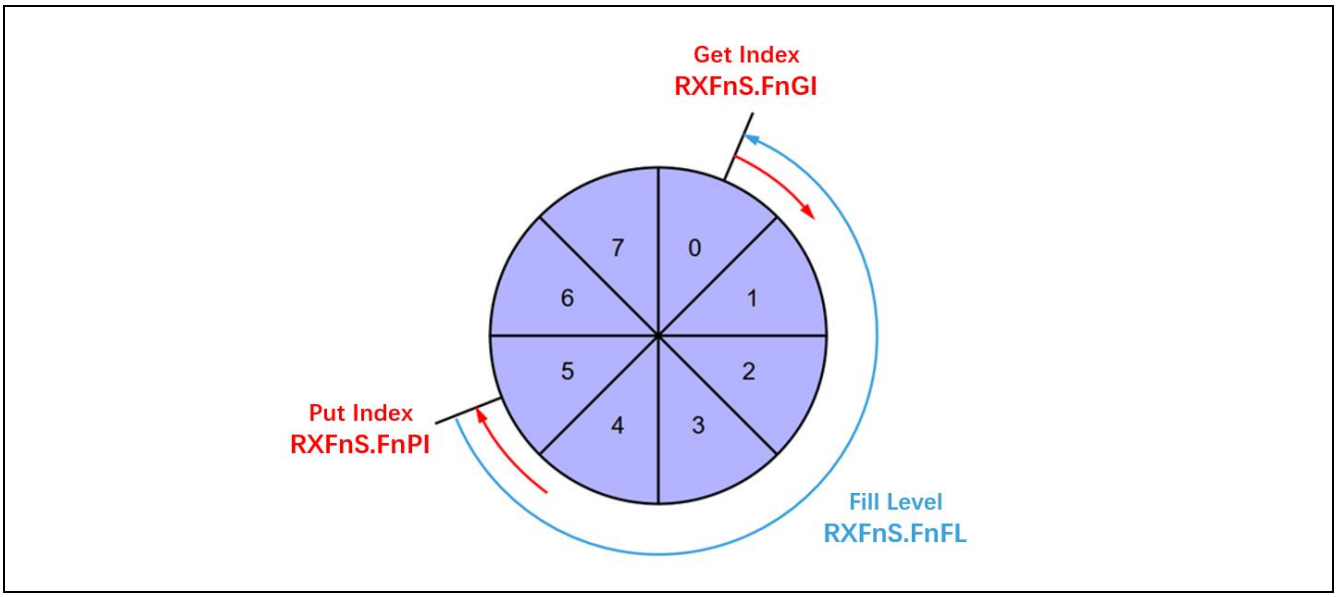


圖 1-1 先入先出佇列

1.2 CAN 消息發送

M460 可以定義 32 個消息發送暫存區，每個暫存區可以配置一個消息發送事件，事件可以記錄消息的發送情。根據不同的發送情況進行業務邏輯處理。

1.2.1 BUFFER 發送方式

當消息被填入 BUFFER 發送 RAM 暫存區，並且使能對應發送位後，M460 會自動將所有 BUFFER 中的使能發送位的消息按照 ID 號從小到大的順序依次發送出去。

這種發送方式可以將常用的命令類消息先準備好，當需要發送的時候只要使能發送位元，消息就能發送出去，不需要再次進行消息內容的填寫。

1.2.2 FIFO/QUEUE 發送方式

FIFO 和 QUEUE 的發送方式均使用了內部的一個先進先出的訊息佇列，當消息一個一個依次使能發送的時候兩者的使用方式一樣，但在多個消息同時被使能發送的時候，發送消息的順序就會有差異。

多個消息同時使能發送時兩者的差異如下表 1-2：

Type	FIFO	QUEUE
寫入順序	消息寫入隊尾	消息寫入隊尾
發送順序	優先發送隊首消息	優先發送 ID 號最小的消息
相同 ID 發送順序	按照 FIFO 順序	優先發送最先請求的消息

表 1-2 FIFO/QUEUE 發送差異

範例結尾會有實例進行觀測分析。

1.2.3 BUFFER+FIFO/QUEUE 發送方式

該模式下結合了 BUFFER 和 FIFO/QUEUE 發送方式的特點，可以更加靈活的處理發送消息的先後順序。

混合模式下的發送先後順序如下表 1-3：

組合模式	發送排序	相同 ID 發送排序
BUFFER+FIFO	優先發送所有專用 Tx 緩衝區中具有最小 ID 的元素和 Tx FIFO 中最早的元素	按照 FIFO 順序
BUFFER+QUEUE	優先發送所有專用 Tx 緩衝區和 Tx 佇列中具有最小 ID 的元素	優先發送最先請求的消息

表 1-3 組合模式發送差異

1.3 測試方法

1.3.1 數據準備

定義 3 種類型各 10 個數據：

- tx_msg[10]：

用於存放 CAN 消息，每個消息僅發送 1 個位元組的消息，這 1 個位元組的消息按照發送 RAM 的填入使能順序進行排序。每個 tx_msg 按照陣列的序號都各自對應一個 id_dat 和 buff_dat。

- id_dat[10] ：

用於存放 CAN 的標準 ID 號，10 個 id_data 中亂序存放了 0x100~0x109 的標準 ID 號。

- buff_dat[10] ：

用於存放消息在 RAM 種對應的存放位置，即 32 個發送元素的位置。10 個 buff_dat[10]中亂序存放了消息的存放位置，用於指定消息存放到 BUFFER 中還是 FIFO 或者 QUEUE 中。

1.3.2 初始化發送配置

M460 有 4 個 CAN，每個 CAN 為其配置不同的發送方式，發送相同的消息。不處於發送狀態的 CAN 會接收到發出的消息，cangroo 軟體會按照匯流排接收到的先後順序依次顯示收到的消息。

CAN0 ：32 個發送元素全部配置成 FIFO 模式。

CAN1 ：32 個發送元素全部配置成 QUEUE 模式。

CAN2 ：10 個發送元素配置成 BUFFER 模式，剩下 22 個元素配置成 FIFO 模式。

CAN3 ：10 個發送元素配置成 BUFFER 模式，剩下 22 個元素配置成 QUEUE 模式。

所有 CAN 口串列傳輸速率等參數均需要設置成一樣，即使用默認的參數就可以。

為了方便觀察消息，發送的時候只對接其中的需要發送和需要接收的 CAN 口即可。

整理了一個表格方便對比 4 個 CAN 口的差異。

表 1-4 中綠色格子表示 CAN2 和 CAN3 發送時使用了 FIFO/QUEUE 模式的消息位置。橘色格子表示使用了 BUFFER 模式進行消息的發送。

CAN 消息数据						CAN0	CAN1	CAN2	CAN3
陣列序號	使能順序	標準 ID	數據位元組	數據內容	Message RAM 位置	寫入區域	寫入區域	寫入區域	寫入區域
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 消息發送配置表

2. 代碼介紹

範例的代碼基於上一個範例代碼修改而來，關於上一個範例代碼可以參考如下連結：

EC_M460_4_CAN_FD_to_SLCAN_by_VCOM_V1.00

本範例基於上一個範例修改內容如下：

- 優化了 CAN 消息的收發機制，減少一次消息格式轉換，提高了消息的收發效率。
- 重新劃分消息 RAM 區域的功能。
- 增加消息發送的測試程式。

測試過程需要修改 USB 設備驅動程式，重命名不同的虛擬 VCOM 名稱，以便找到相應的運行設備。具體的範例可以參考如下連結：

EC_M460_HSUSBD_VCOM_MultiPort_V1.00

2.1 代碼簡介

收發機制的優化主要體現在對接收消息的處理上，原範例每次收到消息後均會產生一次中斷，並且在中斷內將 RAM 消息搬移到 CAN 的消息內。消息量較大的情況下中斷會佔用大量 CPU 資源。優化後的範例利用 CAN 接收消息的 FIFO 接收消息，在中斷程式內僅設置消息接收標誌位元，外部程式判斷到有消息後再去讀取 FIFO 中收到的消息。當系統較繁忙時 FIFO 可以先接收一部分消息，然後在空閒時一次性全部取出。*Reveive2ASCII()* 程式在 slca*n.c* 檔中，負責將接收 FIFO 中的所有消息提取出來，並直接轉換成 SLCAN 的消息格式通過 USB 介面發送給 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 分配資訊在 *canapp.h* 檔中定義了消息元素的類型，每種類型的數量，以及其在 RAM 中的開始地址。因為 RAM 大小有限，所以並不能將每個元素的數量最大化。這個需要在實際使用過程中按需分配。

```
/*-----------------------------------------------------------------------------*/
/* 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)))
```

新增加的測試程式有 2 個，2 個程式均在 *canapp.c* 文件中。

CAN 初始化程式 *CANFD_Test_Start()*，負責將 4 個 CAN 發送配置為不同的發送方式。CAN0 的發送全部配置為 FIFO 模式；CAN1 的發送全部配置為 QUEUE 模式；CAN2 的發送配置為 10 個 BUFFER + 22 個 FIFO；CAN3 的發送配置為 10 個 BUFFER + 22 個 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);
}
```

發送的測試程式 *CAN_TX_Test()* 定義了 10 個發送的消息 *tx_msg[10]*，每個消息有各自不同的 ID。消息 ID（0x100~0x109）的順序在 ID 陣列 *id_dat[10]*中已經打亂。每個消息在 CAN 發送 RAM 中的位置也已經定義在 *buff_dat[10]*陣列中。按照消息陣列的順序依次將需要發送的消息填入到發送 RAM 區域，每當填入一個消息完成後即刻觸發發送使能。因為消息填入速度較快，發送較慢，只有第一個填入的消息會立即進行發送，其他的消息會按照不同發送模式的先後順序進行發送。

```
/**
 * @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 運行實例

本例使用相同的串列傳輸速率設置，測試 1 路 CAN 發送，另外 1 路 CAN 接收。查看接收 CAN 的收到消息的先後順序。依次將 CAN0~CAN3 發送的消息保存下來，然後再來分析一下接收到的消息。測試程式進行了特殊處理，每一路 CAN 只需要點擊 1 次 Send Once，固件就會自動發送 10 個消息。圖 2-1 是 CAN0 發出的 10 個消息，通過 CAN3 來接收。“SN”代表消息收發的先後順序，數越小代表越早收到消息。“CAN_ID”是每個消息的 ID 號。“DLC”代表接收到消息數據的長度，後面跟的“FD”代表是 CANFD 的幀，“FD”右邊“00~09”是每個消息接收到的消息數據。

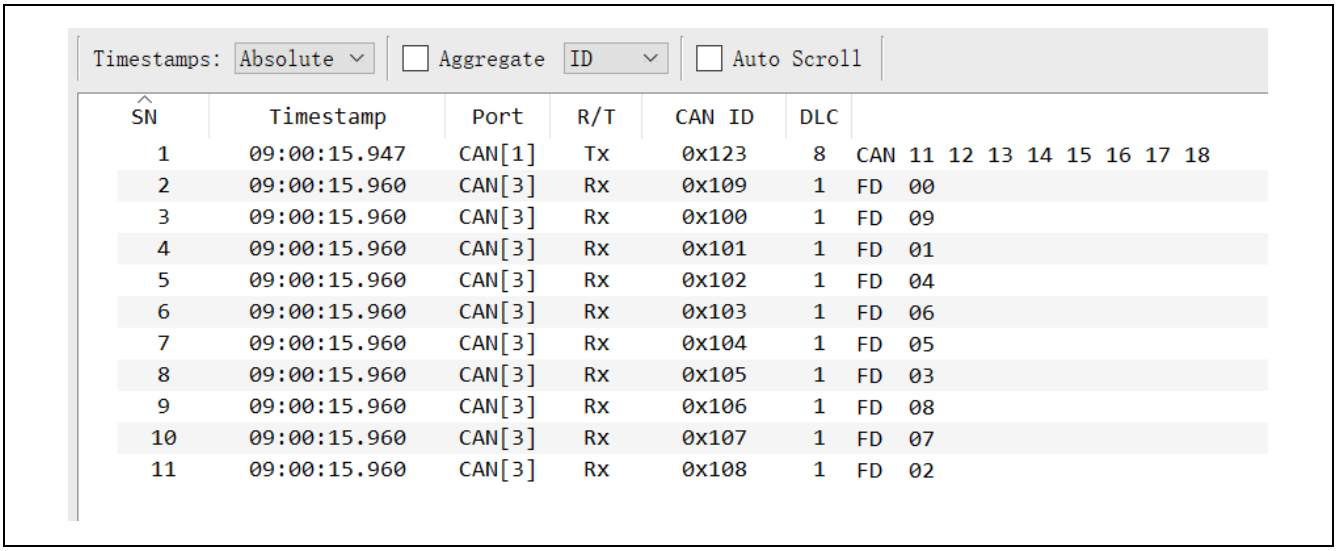
從圖 2-1 上可以看出 CAN0 發出消息，消息數據是從 00 依次排到 09 的，即消息寫入順序和發出順序是一致的。

Timestamps: Absolute ☐ Aggregate ID ☐ Auto Scroll

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

圖 2-1 CAN0 發出的 10 個消息

圖 2-2 是 CAN1 發出的 10 個消息，第 1 個消息 ID 號為 0x109，從第 2 個開始消息的 ID 順序是從 0x100 依次排到 0x108。

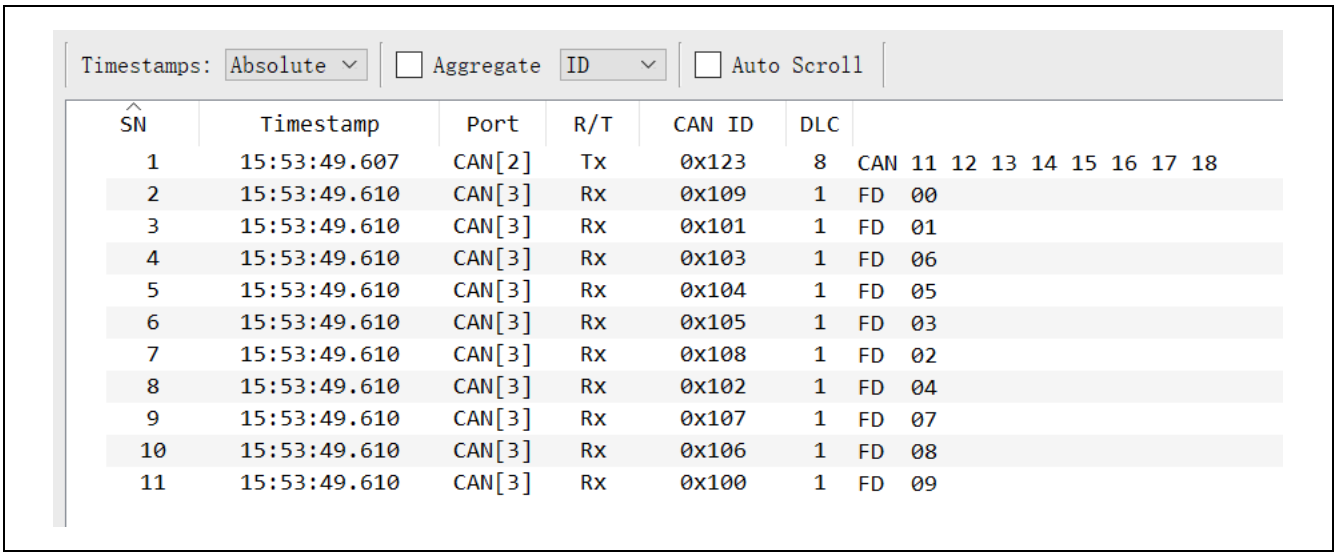


Timestamps: Absolute | Aggregate | ID | Auto Scroll

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

圖 2-2 CAN1 發出的 10 個消息

圖 2-3 是 CAN2 發出的 10 個消息，第 1 個消息依然是 ID 號為 0x109，其他的消息沒有很明顯的規律。後續會統一分析對比。

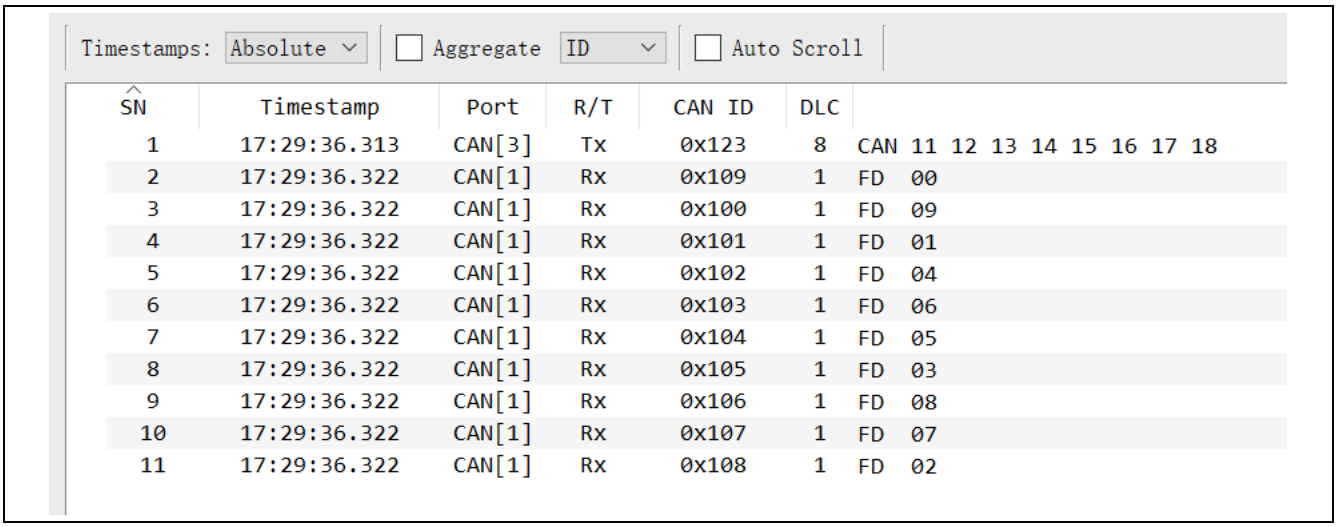


Timestamps: Absolute | Aggregate | ID | Auto Scroll

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

圖 2-3 CAN2 發出的 10 個消息

圖 2-4 是 CAN3 發出的 10 個消息，第 1 個消息依然是 ID 號為 0x109，從第 2 個消息開始 ID 號的順序和 CAN1 接收到的順序是一致的。



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

圖 2-4 CAN3 發出的 10 個消息

從上面 4 個 CAN 發出的消息看，第 1 個消息均為 ID 號為 0x109 的第 1 個使能發送的消息，差異體現在後續發送的 9 個消息上。

將接收到的消息按照實際接收排序整理到表 2-1 如下：

	10 個消息寫入順序				10 個消息的 ID 號			
寫入區域	FIFO	QUEUE	BUFFER FIFO	BUFFER QUEUE	FIFO	QUEUE	BUFFER FIFO	BUFFER QUEUE
實際接收排序	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

結合表 1-4 和表 2-1 可以看出，不同方式發送的消息，第 1 個發出的都是第 1 個被使能的 ID 號為 0x109 的消息。

從第 2 個發出的消息來分析：

- CAN0 完全遵守先入先出的發送規律，發送順序不受 ID 號優先順序的影響。
- CAN1 和 CAN3，因為 BUFFER 和 QUEUE 的發送規律是 ID 號越小，發送優先順序越高，所以兩者的發送順序是一樣的。
- CAN2 發送規則有 2 種，所以發送消息的順序相比其他的會有較大的差異。每次發送的時候會先檢索 BUFFER 中的 ID，將其和 FIFO 中第 1 個 ID 進行對比，發送 ID 值較小的消息。當 BUFFER 中優先順序較高的 ID 發送完成後，然後再將 FIFO 中的消息依次發出。
- 表 2-1 中綠色的格子表示使用 FIFO 發送消息的 ID。CAN2 的 FIFO 第 1 个 ID（0x101）優先順序比 BUFFER 中的 ID 都高，所以先進行了發送。發送完成後再次檢索優先順序的時候 BUFFER 中的 ID 優先順序都比當前 FIFO 第 1 个 ID（0x108）要高，所以將 ID 為 0x103，0x104，0x105 消息優先發出，發出後再將剩下 FIFO 中的消息依次發出。

3. 軟體與硬體需求

3.1 軟體需求

- 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 硬體需求

- 電路元件
  - 3-In-1 Analyzer Tool
- 線路示意圖
  - 將 UART0 TX pin 腳連接到 PC UART RX，以顯示範例程式的執行結果。
  - 將 HSUSB 連接到 PC，並打開 Cangaroo 上位機，以顯示收發的 CAN 消息。
  - 測試的時候只需要將不同 2 個的 CAN 口相互連接。

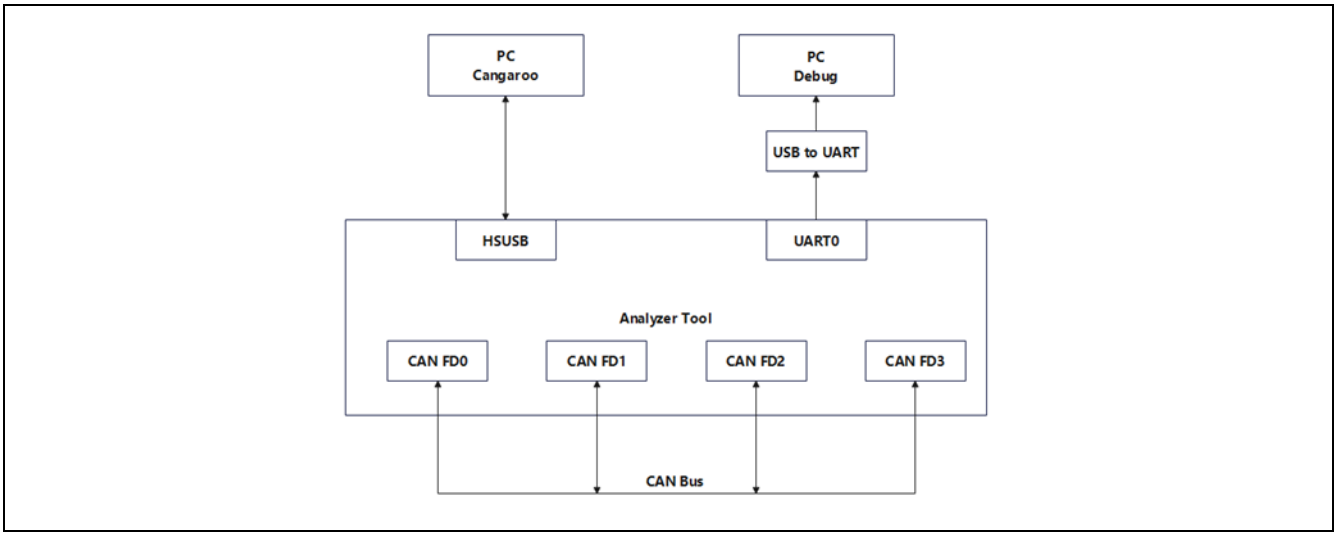


圖 3-1 線路示意圖

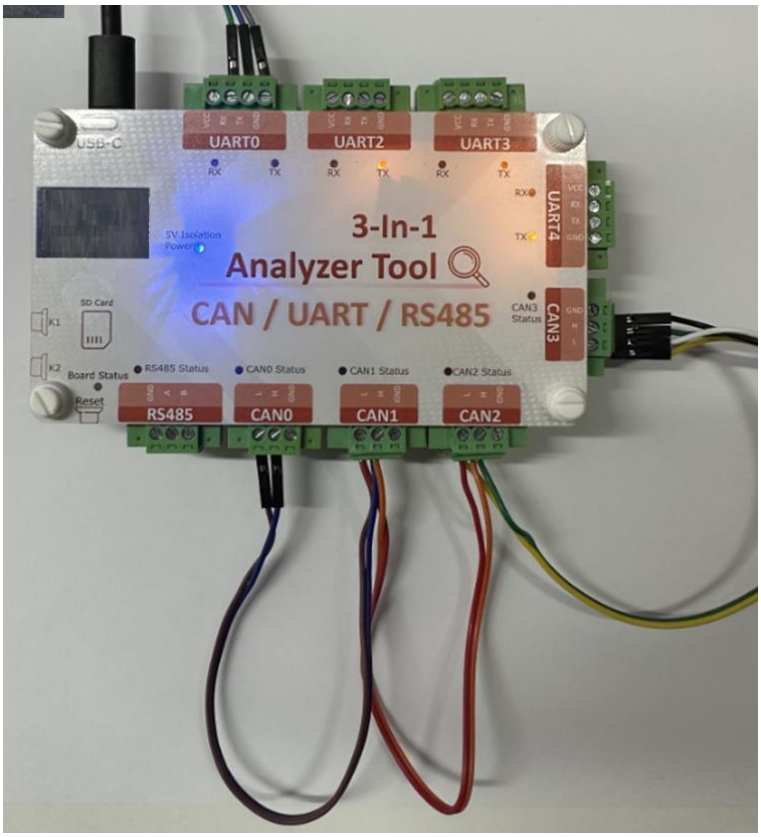

USB-C
UART0
UART2
UART3
UART4
3-In-1
Analyzer Tool
CAN / UART / RS485
5V Isolation Power
SD Card
K1
K2
Board Status
Reset
RS485 Status
CAN0 Status
CAN1 Status
CAN2 Status
CAN3 Status
CAN3
RS485
CAN0
CAN1
CAN2

图 3-2 實物接線

4. 目錄資訊

目錄結構如下圖所示。

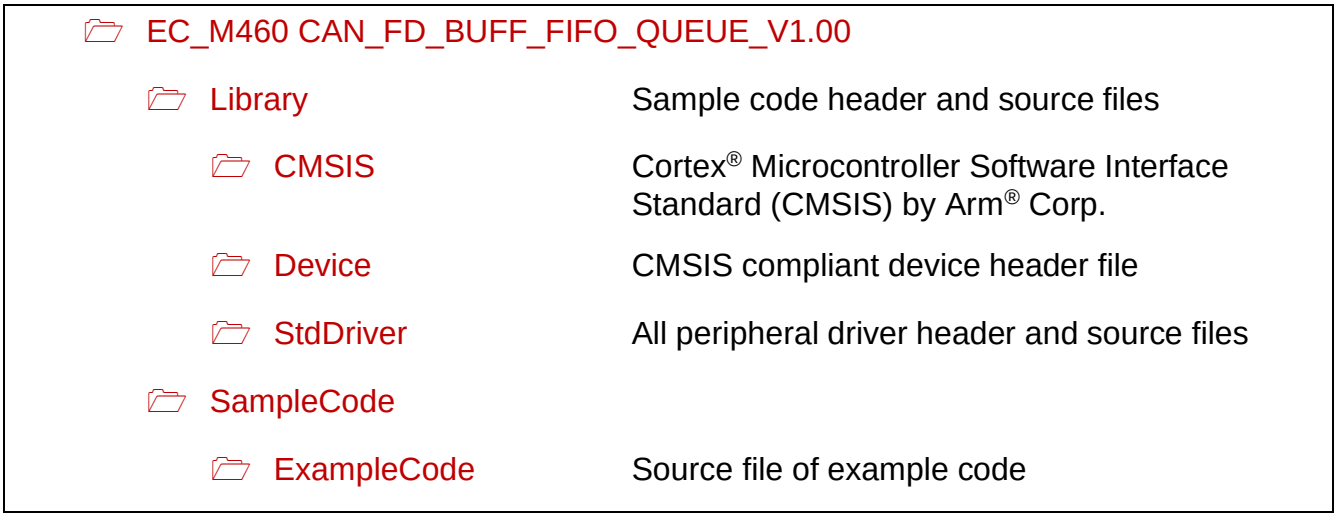


圖 4-1 目錄資訊

5. 例程式執行

1. 根據目錄資訊章節進入 ExampleCode 路徑中的 KEIL 資料夾，雙擊 *CAN_FD_BUFF_FIFO_QUEUE.uvprojx*。
2. 進入編譯模式介面
   - 編譯
   - 下載代碼至記憶體
   - 進入 / 離開除錯模式
3. 進入模擬模式介面
   - 執行程式

6. 修訂紀錄

Date	Revision	Description
2025.09.3	1.00	初始發布。

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.