Using Different Methods to Send Messages via M460's CAN FD

Example Code Introduction for 32-bit NuMicro® Family

Document Information

Application	This example code will demonstrate different transmission effects produced by several different CAN message transmission methods of the M460 series microcontroller.
BSP Version	M460_Series_BSP_CMSIS_V3.00.004
Hardware	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. Overview

The M460's CAN FD offers a significantly improved communication rate compared to CAN 2.0, optimizing data transmission and reception efficiency. Message reception provides two FIFO message buffers. In the following text, CAN FD will be referred to simply as CAN.

The message sending function provides five sending modes:

- Single TX BUFFER
- Single TX FIFO
- Single TX QUEUE
- TX BUFFER + TX FIFO
- TX BUFFER + TX QUEUE

Message sending also supports sending event logging. This example does not cover the function of sending events. Single BUFFER mode is the simplest and most used sending method and will not be described in detail in this example. The example will focus on demonstrating the differences between the other four sending methods.

The 0x1800 bytes of RAM in the example are defined in Table 1-1:

Definition	Function	Bytes Size	Number	RAM Address
SID_BUFF_SIZE	Standard Message ID Filter Element	4 bytes	20	0
XID_BUFF_SIZE	Extended Message ID Filter Element	8 bytes	16	0x50
TX_EVENT_SIZE	TX Event FIFO Element	12 bytes	32	0xD0
TX_BUFF_SIZE	TX Buffer Element	8+64 bytes	32	0x250
RX_FIFO_SIZE	RX Buffer and FIFO Element	8+64 bytes	45	0xB50

Table 1-1 RAM Allocation

1.1 CAN RX FIFOs

The receiver has two FIFOs, and each receiver can be configured with a maximum of 64 buffers. Each FIFO can be configured with different message filters, allowing for rapid hardware-based classification of received messages.

FIFO reads data in a first-in, first-out manner; new data is added to the tail of the FIFO upon arrival. The receiving method is illustrated in Figure 1-1:

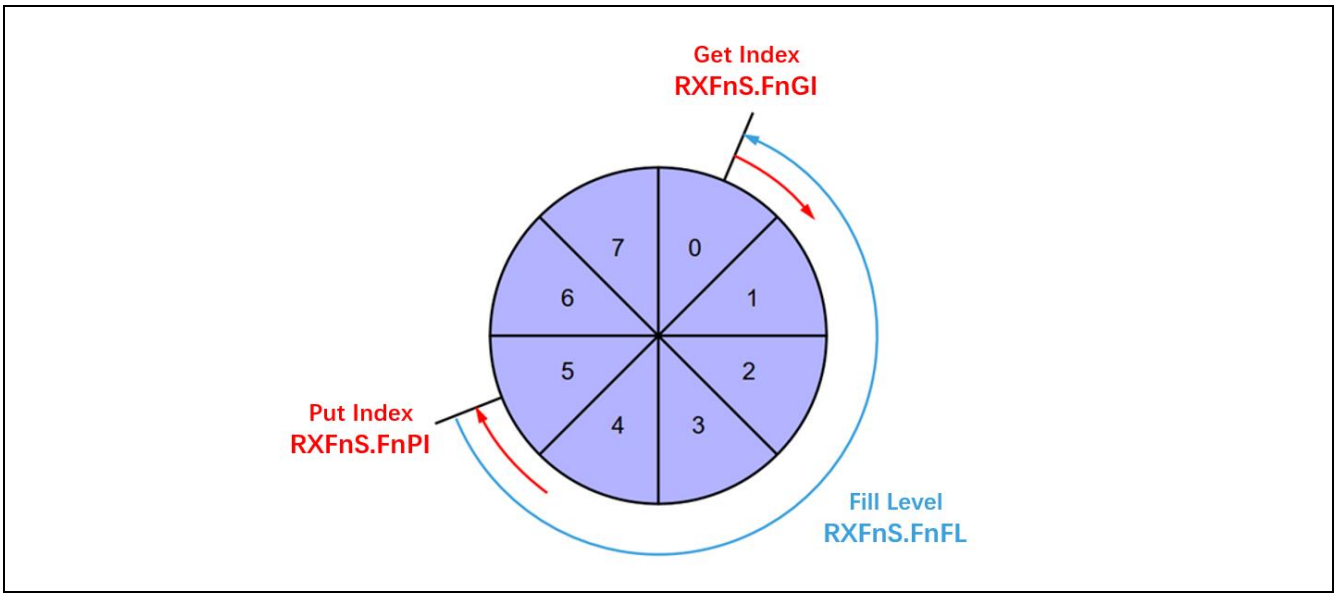


Figure 1-1 Receive FIFO

1.2 CAN Message TX

The M460 can define TX buffers for 32 messages, each of which can be configured with a message transmitted event. These events record the message transmitting status, and application logic is processed based on the different transmitting statuses.

1.2.1 Dedicated TX BUFFER

Once a message is filled into the BUFFER sent to RAM storage area and the corresponding send bit is set 1, the M460 will automatically send all messages in the BUFFER with the send bit enabled in ascending order of ID number.

This transmission method allows you to prepare commonly used command messages in advance. When you need to send them, you only need to enable the pending bit to send the data without having to fill in the message content again.

1.2.2 TX FIFO/TX QUEUE

Both FIFO and QUEUE sending methods use an internal first-in-first-out message queue. When messages are enabled and sent one by one, they are used in the same way. However, when multiple messages are pending to be sent at the same time, the order in which the messages are sent will be different.

The differences between FIFO and QUEUE when multiple messages are pending to be sent simultaneously are as Table 1-2:

	FIFO	QUEUE
Write sorting	Message written to the tail of the queue	Message written to the tail of the queue
Transmit sorting	Send the head of the queue message first	Send the message with the smallest ID number first
Order of transmitting identical IDs	according to FIFO order	Send the message that has been requested first

Table 1-2 FIFO/QUEUE Differences

The example code will end with a test analysis.

1.2.3 TX BUFFER + TX FIFO/QUEUE

This mode mixes the features of BUFFER and FIFO/QUEUE sending methods, allowing for more flexible handling of the order in which messages are sent.

The transmission order in mixed mode is shown in Table 1-3.

Mixed Mode	Sending Order	Ordering of sending with the same ID
BUFFER + FIFO	First request is the lowest ID between all dedicated TX buffers and oldest messages in TX FIFO.	Order request FIFO
BUFFER + QUEUE	First request is the lowest ID between all dedicated TX buffers and TX queue.	First request is sent first

Table 1-3 Mixed Mode Sending Differences

1.3 Test Methods

1.3.1 Data Preparation

Define 10 variables of each of the three types:

- tx_msg[10]:

Used to store CAN messages, each message sends only 1 byte of data. This 1-byte data is sorted according to the order in which it is filled into the transmit RAM. Each tx_msg corresponds to an id_dat and a buff_dat according to the array index.

- id_dat[10]:

Used to store the standard ID number of CAN, the 10 id_data entries contain standard ID numbers from 0x100 to 0x109 in random order.

- buff_dat[10]:

Used to store the corresponding storage location of the message in RAM, that is, the location of the 32 sending elements. The message storage location is stored out of order in 10

buff_dat[10], which is used to specify whether the message is stored in BUFFER, FIFO or QUEUE.

1.3.2 Initialized Transport Configuration

The M460 has four CAN ports, each configured with a different transmission mode, but all transmitting the same data. The CAN port not in transmission mode will receive transmitted messages, and the Cangaroo software will display the received messages in the order they are received on the CAN bus.

CAN0: All 32 transmitting elements are configured in FIFO mode.

CAN1: All 32 transmitting elements are configured in QUEUE mode.

CAN2: 10 transmitting elements are configured in BUFFER mode, and the remaining 22 elements are configured in FIFO mode.

CAN3: 10 transmitting elements are configured in BUFFER mode, and the remaining 22 elements are configured in QUEUE mode.

All CAN port baud rates and other parameters must be set to the same value; the default parameters are sufficient.

To facilitate data observation, only the CAN ports that need to be sent and received need to be connected during transmission.

A table has been compiled to make it easier to compare the differences between the four CAN ports.

In Table 1-4, green cells indicate message positions where FIFO/QUEUE mode is used during CAN2 and CAN3 transmission. Orange cells indicate data transmission using BUFFER mode.

CAN Message						CAN0	CAN1	CAN2	CAN3
Array SN	Pending Order	ID	Data Size	Data	Message RAM location	Write area	Write area	Write area	Write area
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

Table 1-4 Message Sending Configuration

2. Code Description

The code for this example is a modification of the previous example code. The previous example code can be found at the following link:

EC_M460_4_CAN_FD_to_SLCAN_by_VCOM_V1.00

This example is based on the previous example and the modifications are as follows:

- Optimized the CAN message sending and receiving mechanism, reducing one message format conversion and improving message sending and receiving efficiency.
- Added the function to re-divide the message RAM area.
- Added a message sending test function.

The testing process requires modifying the USB device driver and renaming different virtual VCOM names in order to locate the corresponding executing device. Another example code can be found at the following link:

EC_M460_HSUSBD_VCOM_MultiPort_V1.00

2.1 Code Introduction

The optimization of the send/receive mechanism is mainly reflected in the processing of received message data. The original routine would generate an interrupt every time a message is received, and the RAM data would be moved to the CAN message within the interrupt. When dealing with large amounts of data, interruptions can consume significant CPU resources. The optimized code uses the CAN message receiving FIFO to receive data. Only the data receiving flag is set in the interrupt code, and the outer loop reads the data received in the FIFO after detecting that a message has been received. When the system is busy, the FIFO can receive a portion of the data first, and then retrieve all of it at once when the system is idle. The *Reveive2ASCII()* function in the *slcan.c* file is responsible for extracting all data from the received FIFO and directly converting it into SLCAN data format before sending it to the PC via the USB interface.

```
/**
 * @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);
        }
    }
}
```

The RAM allocation information for CAN is defined in the *canapp.h* file, which specifies the type of data element, the quantity of each type, and its starting address in RAM. Because RAM size is limited, the number of each element cannot be maximized. This needs to be allocated on demand during actual use.

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

Two new test functions have been added; both of which are located in the *canapp.c* file.

The CAN initialization function *CANFD_Test_Start()* is responsible for configuring the four CAN transmitters to different transmission modes. CAN0 is configured to transmit entirely in FIFO mode; CAN1 is configured to transmit entirely in QUEUE mode; CAN2 is configured with 10 BUFFERs + 22 FIFOs; and CAN3 is configured with 10 BUFFERs + 22 QUEUEs.

```
/**
 * @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);
}
```

The test function *CAN_TX_Test()* defines 10 messages *tx_msg[10]* to be sent, each with its own unique ID. The order of message IDs (0x100~0x109) has been shuffled in the ID array *id_dat[10]*. The location of each message in the CAN transmit RAM has also been defined in the *buff_dat[10]* array. The data to be sent is filled into the sending RAM area in the order of the message array, and the sending enable is triggered immediately after each data is filled. Because data entry is fast but sending is slow, only the first entered data will be sent immediately, and other messages will be sent in the order of their respective sending modes.

```
/**
 * @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 Executing Example

This example uses the same baud rate setting to test one CAN transmission and another CAN receiving. The order in which the received CAN messages are viewed is observed. The message data transmitted by CAN0 through CAN3 is saved sequentially, and then the received messages are analyzed. The test firmware has been specially processed so that each CAN channel only needs to be clicked once (Send Once) and the firmware will automatically send 10 messages. Figure 2-1 shows 10 messages sent by CAN0 and received by CAN3. “SN” represents the order of message transmission and reception; a smaller number indicates an earlier message that has been received. “CAN_ID” is the ID number of each message. “DLC” represents the length of the received message data, followed by “FD” indicating a CAN FD frame, and “00~09” represent the received message data for each message.

It can be seen from Figure 2-1 that when CAN0 sends a message, the message data is arranged sequentially from 00 to 09, meaning that the message writing order and the sending order are consistent.

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

Figure 2-1 10 Messages Sent by CAN0

Figure 2-2 shows 10 messages sent by CAN1. The ID number of the first message is 0x109, and the ID order of the messages from the second message onwards is from 0x100 to 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

Figure 2-2 10 Messages Sent by CAN1

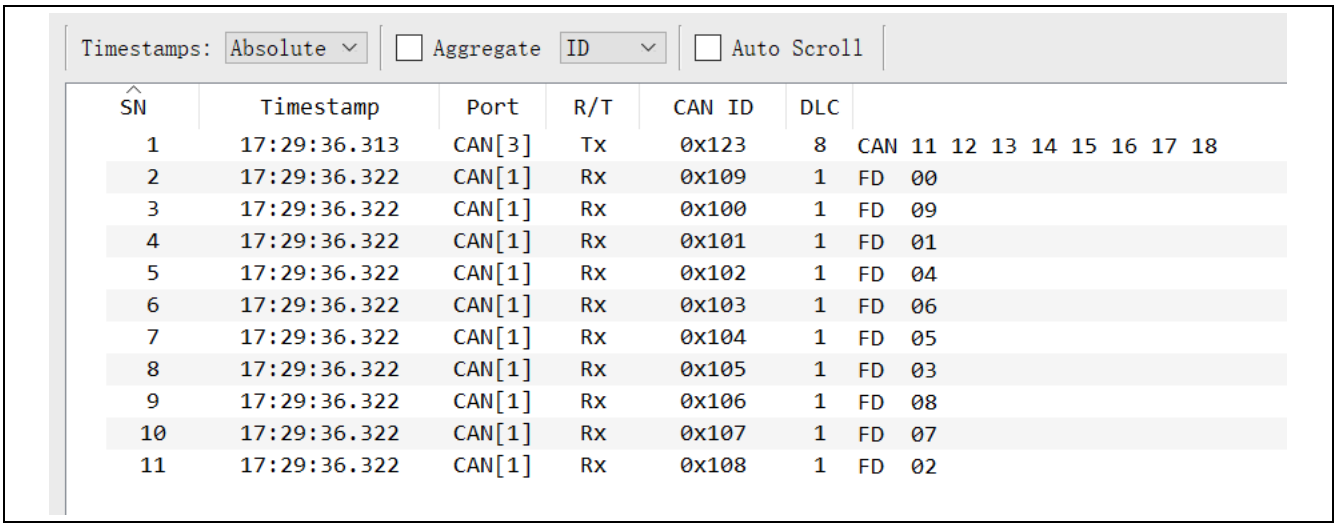
Figure 2-3 shows 10 messages sent by CAN2. The first message still has the ID number 0x109, and the other messages do not show any obvious pattern. A unified analysis and comparison will be conducted later.

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

Figure 2-3 10 Messages Sent by CAN2

Figure 2-4 shows 10 messages sent by CAN3. The first message still has an ID number of 0x109. Starting from the second message, the order of the ID numbers is consistent with the order received by CAN1.



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

Figure 2-4 10 Messages Sent by CAN3

Looking at the four CAN messages above, the first message is the first enable message with ID number 0x109. The difference lies in the nine messages sent afterward.

The received messages are sorted according to the actual number of messages received and arranged in Table 2-1:

	Write Order of 10 Messages				ID Number of 10 Messages			
Write Area	FIFO	QUEUE	BUFFER FIFO	BUFFER QUEUE	FIFO	QUEUE	BUFFER FIFO	BUFFER QUEUE
Received in Order	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

Table 2-1 Data Sorting Out

Combining Table 1-4 and Table 2-1, it can be seen that for messages sent in different ways; the first one sent is always the first enabled message with ID number 0x109.

Let's analyze the messages starting from the second one:

- CAN0 strictly follows the first-in-first-out (FIFO) transmission rule, and the transmission order is not affected by the ID number priority.

- For CAN1 and CAN3, since the sending rule for BUFFER and QUEUE is that the smaller the ID number, the higher the sending priority, the sending order for both is the same.
- There are two CAN2 sending rules, so the order in which messages are sent will be quite different from other rules. Each time a message is sent, the ID in the buffer is first retrieved and compared with the first ID in the FIFO. The message with the smaller ID is sent. After the higher-priority IDs in the BUFFER have been sent, the messages in the FIFO are then sent sequentially.
- In Table 2-1, the green cells represent the IDs of messages sent using the FIFO. The first ID (0x101) in CAN2's FIFO has a higher priority than all IDs in the BUFFER, so it is sent first. After sending, when the priority is retrieved again, the IDs in the BUFFER all have a higher priority than the first ID (0x108) in the current FIFO, so messages with IDs 0x103, 0x104, and 0x105 are sent first, and then the remaining messages in the FIFO are sent in sequence.

3. Software and Hardware Requirements

3.1 Software Requirements

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

3.2 Hardware Requirements

- Circuit components
  - 3-In-1 Analyzer Tool
- Schematic diagram of the route
  - Connect the UART0 TX pin to the PC UART RX to show the execution result of the sample code.
  - Connect the HSUSB to the PC and turn on the Cangaroo to show the sent and received CAN messages.
  - During the test, only connect the two different ports to each other.

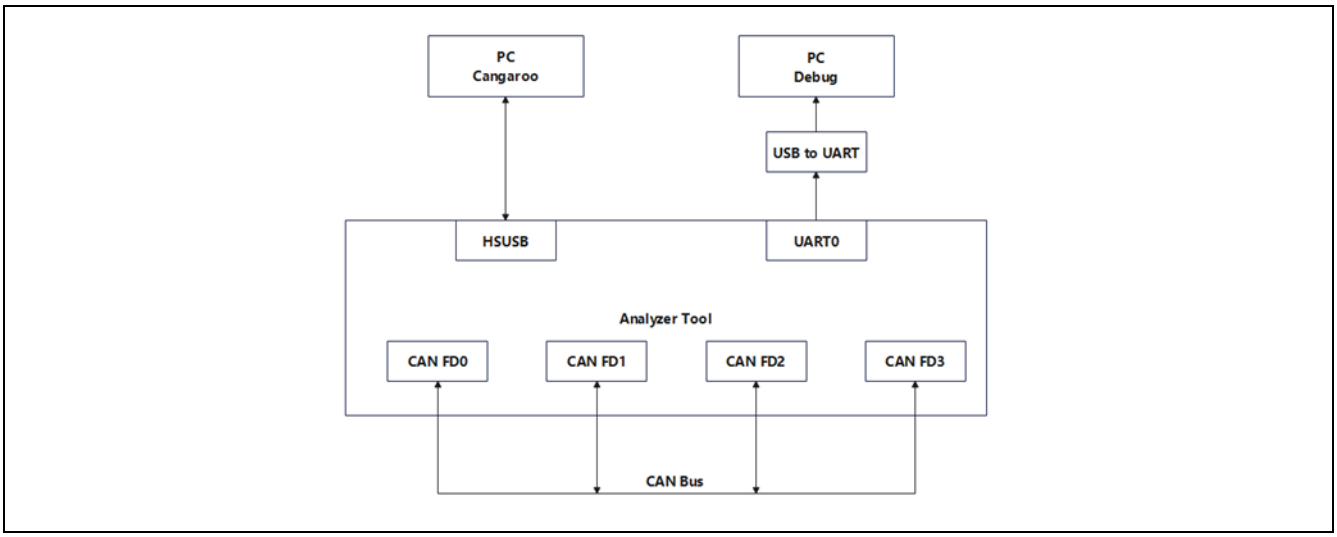


Figure 3-1 Pin Connect

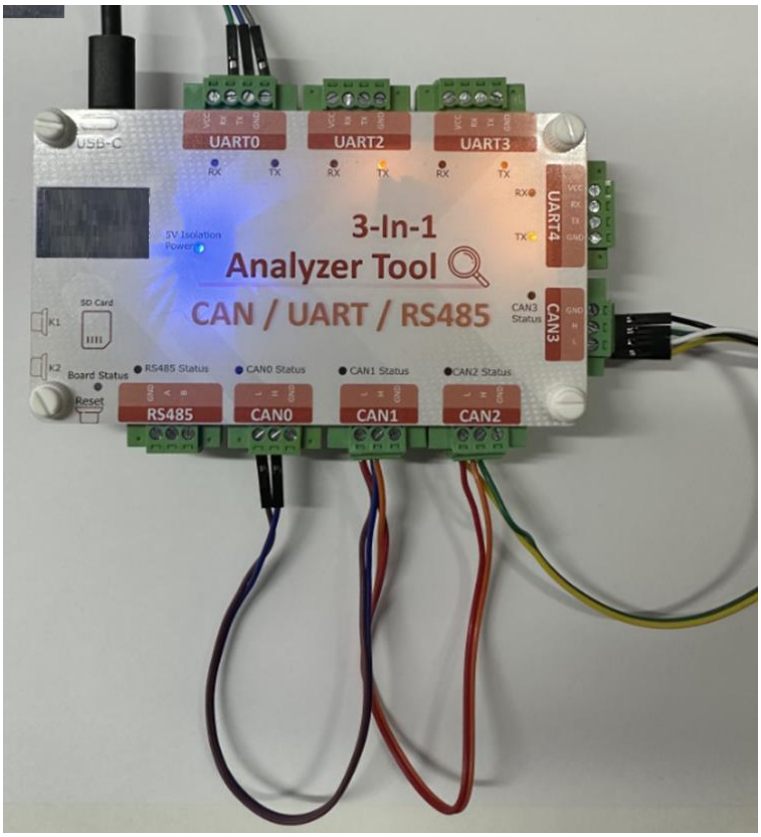


Figure 3-2 Physical Wiring

4. Directory Information

The directory structure is shown below.

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

Figure 4-1 Directory Structure

5. Excample Code Execution

1. Browse the sample code folder as described in the Directory Information section and double-click *CAN_FD_BUFF_FIFO_QUEUE.uvprojx*.
2. Enter Keil compile mode.
   - Build
   - Download
   - Start/Stop debug session
3. Enter debug mode.
   - Run

6. Revision History

Date	Revision	Description
2025.09.3	1.00	Initial version.

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.