Converting M460 4 CAN FD Messages to SLCAN by VCOM

Example Code Introduction for 32-bit NuMicro® Family

Document Information

Application	This example code converts 4 CAN FD messages of the M460 series microcontroller (MCU) into SLCAN protocol through the VCOM interface and sends them to the computer.
BSP Version	M460_Series_BSP_CMSIS_V3.00.002
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 CAN FD messages are stored in the microcontroller (MCU) with binary format, which makes it difficult to observe and operate the data on the CAN FD bus. This example converts CAN FD messages into ASCII code with SLCAN protocol through the virtual serial port VCOM to communicate with the debugging assistant on the PC.

1.1 The following four basic contents are introduced in the overview.

- Introduction to CAN FD
- Introduction to VCOM
- Introduction to SLCAN protocol
- CAN debugging assistant

1.2 Introduction to CAN FD

CAN FD (CAN with Flexible Data-rate) was introduced by Bosch in 2012 to overcome the Classical CAN's bit rate limitation to 1 Mbps and to expand the number of data bytes per CAN frame from up to 8 to up to 64, thereby closing the gap between Classical CAN and other protocols. This is achieved by a modified CAN frame format where the bit rate can be switched to faster value inside the CAN frame and by a new data length coding. CAN FD is standardized as ISO11898-1:2015. CAN FD protocol controllers are also able to perform Classical CAN communication.

Key features of CAN FD vs. classic CAN:

- Higher baud rate: The maximum baud rate in the data domain can reach 8 Mbps.
- More data quantity: Up to 64 bytes.

1.3 Introduction to VCOM

Virtual communication port is referred to as VCOM, which uses the USB interface to replace the hardware serial communication port of the Industrial Computer. In this example, one high-speed USB interface of the M467 high-performance microcontroller is used to virtualize up to 4 VCOMs, which can send and receive 4 CAN FD messages with different baud rates at the same time.

1.4 Introduction to SLCAN Protocol

Serial Line CAN (SLCAN) is a serial communication port protocol based on ASCII code, and it is also the protocol of LAWICEL hardware devices. The SLCAN communication protocol used in this article is compatible with some CAN232 devices, and the latest version of CAN232 is Version 3.0B in 2023.

The protocol can be downloaded from LAWICEL CAN232 | canusb.com.

SLCAN protocol base format: command + parameter + [CR].

This example supports the following commands:

Command	Code	Param1	Param2	Param3	Param4
Open Port	O	[CR]			
Close Port	C	[CR]			
Set the Baud Rate of the Arbitration Domain	S	1~8	[CR]		
Set the Baud Rate of the Data Domain	Y	2,5	[CR]		
CAN Standard Remote Frame	r	11bit Frame ID	Data Length	[CR]	
CAN Standard Frame	t	11bit Frame ID	Data Length	Data	[CR]
CAN FD Standard Frame	d	11bit Frame ID	Data Length	Data	[CR]
CAN FD BRS Standard Frame	b	11bit Frame ID	Data Length	Data	[CR]
CAN Extended Remote Frame	R	29bit Frame ID	Data Length	[CR]	[CR]
CAN Extended Format	T	29bit Frame ID	Data Length	Data	[CR]
CAN FD Extended Frame	D	29bit Frame ID	Data Length	Data	[CR]
CAN FD BRS Extended Frame	B	29bit Frame ID	Data Length	Data	[CR]
Software Version	V	[CR]			
Hardware Version	v	[CR]			
Serial Number	N	[CR]			

Figure 1-1 SLCAN Supported Commands

Arbitration domain supported baud rate parameters:

Code: S	Param1 & Baud Rate of the Arbitration								
Param1	0	1	2	3	4	5	6	7	8
Baud Rate of the Arbitration	10kbps	20kbps	50kbps	100kbps	125kbps	250kbps	500kbps	800kbps	1Mbps

Figure 1-2 Arbitration Domain Parameter Baud Rate Comparison

Baud rate parameters supported by the data domain:

Code: Y	Param1 & Baud Rate of the Data	
Param1	2	5
Baud Rate of the Data	2Mbps	5Mbps

Figure 1-3 Data Domain Parameter Baud Rate Comparison

Index code for data quantity:

Data Length	Param2 & Data Length						
Param2	9	A	B	C	D	E	F
Data Length	12Bytes	16Bytes	20Bytes	24Bytes	32Bytes	48Bytes	64Bytes

Figure 1-4 Data Length Parameters and Index Code Comparisons

1.5 CAN Debugging Assistant

There are two types of CAN debugging assistants: serial port debugging assistant and Cangaroo debugging assistant.

- Serial port debugging assistant:

  To send and receive data through a standard serial port, you need to decode the received SLCAN data and encode the sending CAN FD messages. This example uses the serial debugging assistant to monitor the running logs of the firmware.

- Cangaroo debugging assistant:

  Cangaroo, an open-source CAN bus data analysis tool, is developed and maintained by Hubert Denkmair. This small and powerful software is designed to help electronics engineers and hobbyists to interpret and understand in-vehicle network communication data. Built on the Qt framework, Cangaroo provides an intuitive graphical user interface that supports multiple operating systems.

  The Cangaroo used in this example is optimized for the Nuvoton hardware platform, and the software includes parsing function, which can easily send the required CAN FD messages directly, and can intuitively observe the data received on the CAN bus. You can open 4 debug ports at one time, quickly select the data ports to be sent, and monitor the messages received by the 4 CAN FD ports at the same time.

2. Code Description

This section first introduces the program workflow to make it easier for user to understand the operation mechanism and better to introduce the program, and then analyze an operation instance of the program.

- Flow chart: Data sending and receiving process and CAN debugging operation process.
- Code introduction: USB Virtual COM, Data Sending and Receiving, and Protocol Conversion.
- Operation instance: 1 CAN port sends data while 3 CAN ports receive, monitor the execution log, and analyze and compare the received data.

2.1 Flow Chart

The data sending and receiving process is as follows:

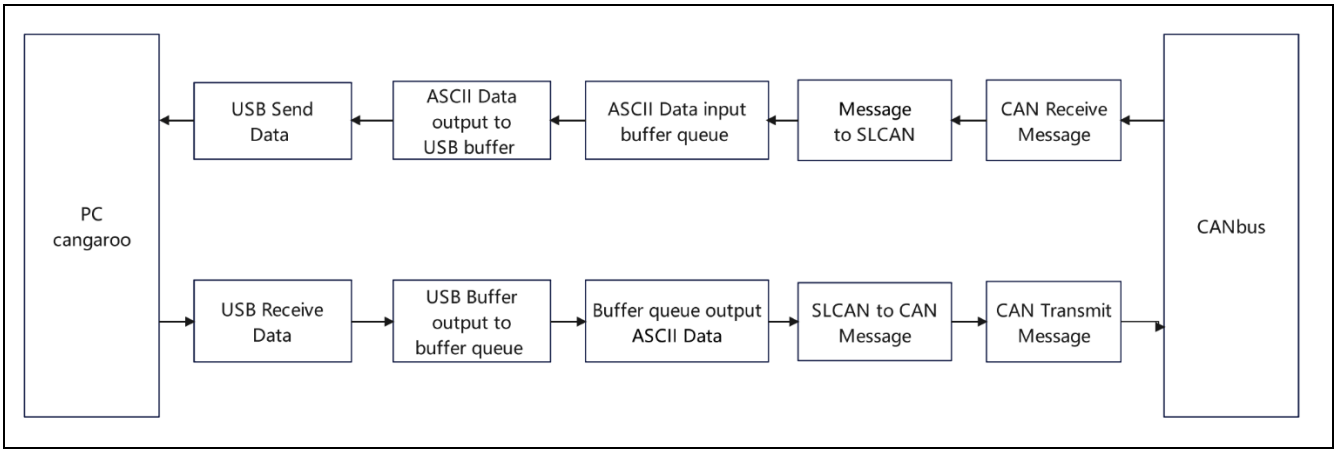

Figure 2-1 Flowchart of Data Sending and Receiving

There are four main cache areas involved in data transmission and reception:

- USB cache: Stores real-time packet data for USB transmission and receiving.
- RAM cache: Used for bidirectional data transmission, improving the real-time performance of data response.
- SCLAN cache: Memory area for converting CAN messages.
- CAN cache: Stores real-time message data for CAN to send and receive.

The CAN debugging operation process is as follows:

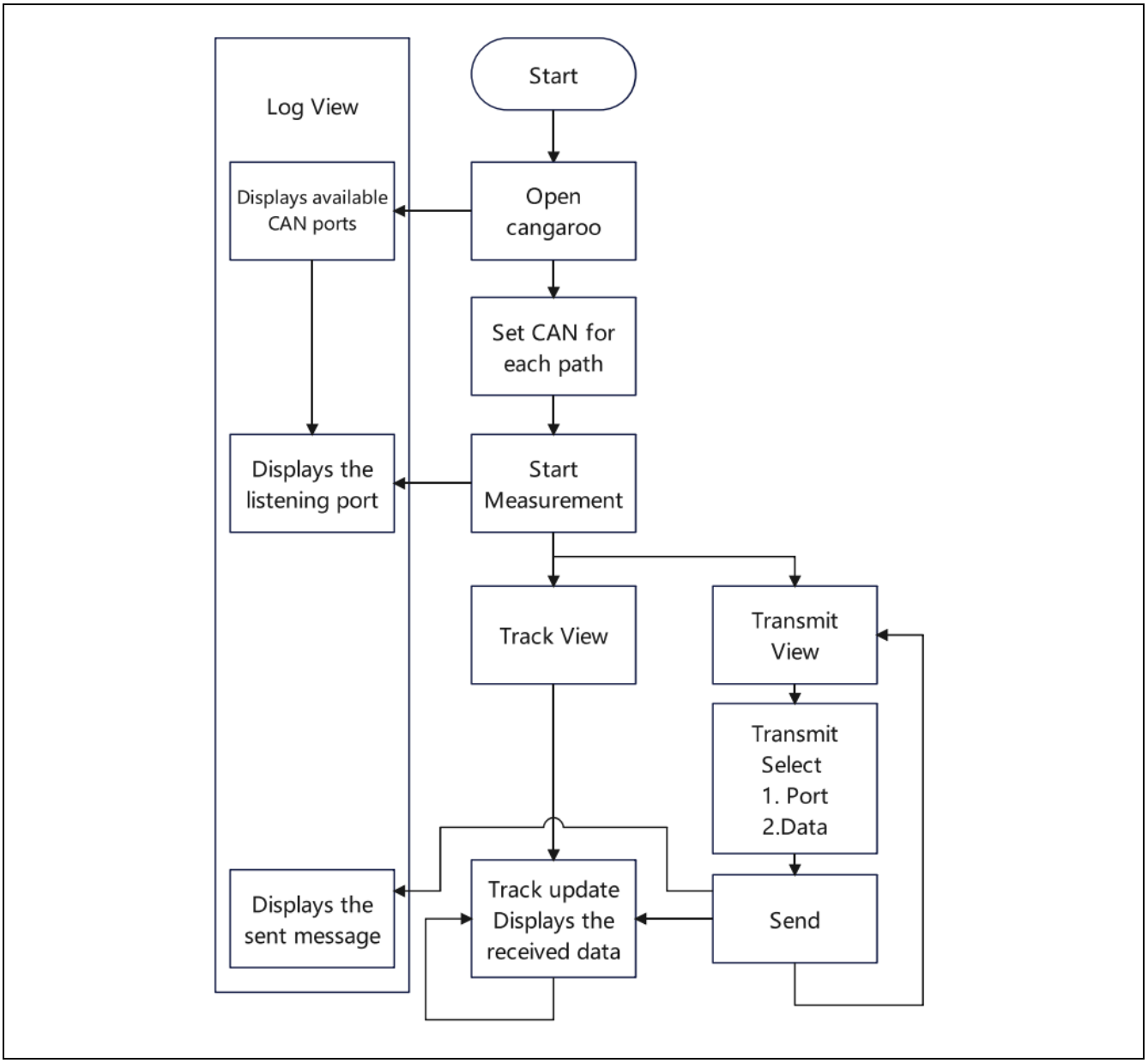

Figure 2-2 Computer Debugging Operation Flow Chart

The CAN debugging assistant has three windows (Figure 2-3):

- Log window: Displays the status of the previous device and the CAN messages that have been sent.
- Tracking window: Displays the messages received by the current CAN, and the display mode can be adjusted according to the timestamp.
- Transmitting window: Selects the port, format and type of message to be sent.

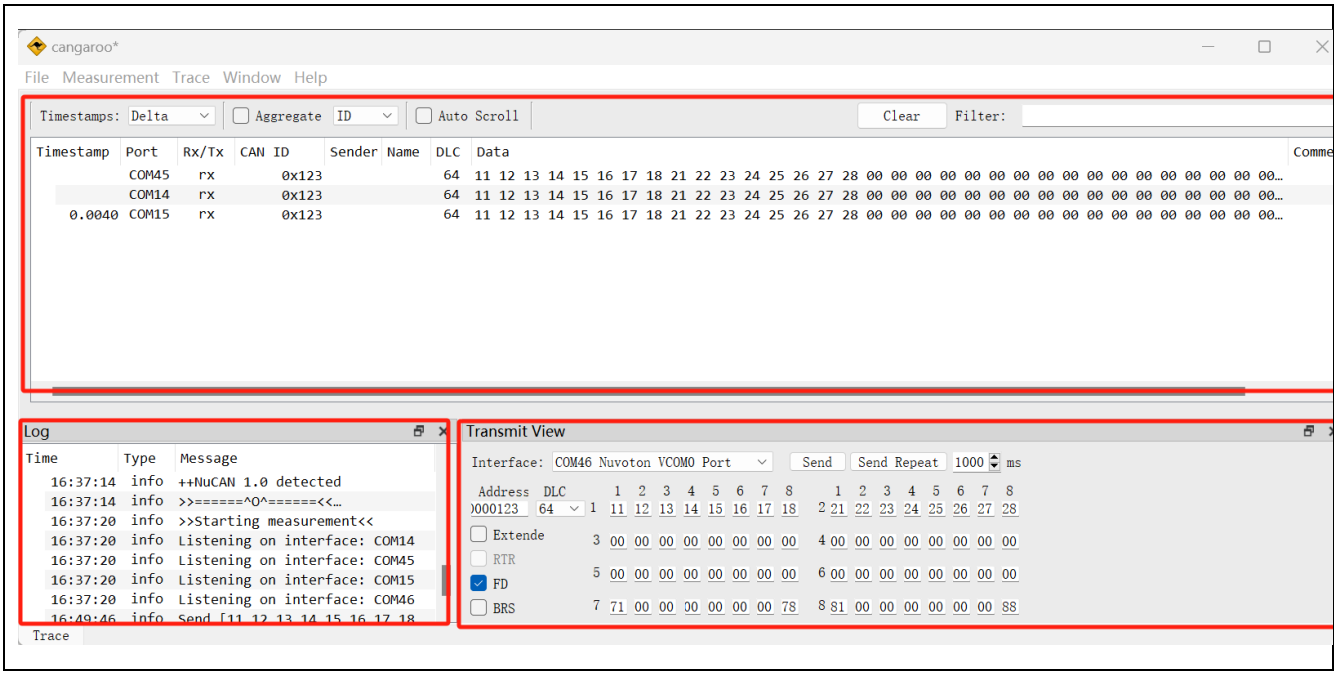

Figure 2-3 CAN Debug Assistant Windows

As Figure 2-3, the top window is the tracking window, the small window in the lower left is the log window, and the lower right is the message transmitting window.

2.2 Code Introduction

USB Virtual COM is a virtual 4 ports VCOM that describes the type of USB device and node information by the *descriptors.c* descriptor file. Each VCOM has its own description; VCOM0 is described as follows.

```
//-----------------VCOM0------------------//
#if (VCOM_CNT >= 1)
    // IAD
    LEN_IAD,        // bLength        : Interface Descriptor size
    DESC_IAD,       // bDescriptorType: IAD
    INTF_NUM_VCOM0, // bFirstInterface
    0x02,           // bInterfaceCount
    0x02,           // bFunctionClass     : CDC
    0x02,           // bFunctionSubClass
    0x01,           // bFunctionProtocol
    0x02,           // iFunction

    /* INTERFACE descriptor */
    LEN_INTERFACE,                          /* bLength             */
    DESC_INTERFACE,                         /* bDescriptorType     */
    INTF_NUM_VCOM0,                         /* bInterfaceNumber    */
    0x00,                                   /* bAlternateSetting   */
    0x01,                                   /* bNumEndpoints       */
    DESC_CDC_COMMUNICATION_INTERFACE_CLASS, /* bInterfaceClass     */
    DESC_CDC_ABSTRACT_CONTROL_MODEL,        /* bInterfaceSubClass  */
    0x01,                                   /* bInterfaceProtocol  */
    0x00,                                   /* iInterface          */
//LEN_VCOM_COMMAND_INTR
```

```
    /* Communication Class Specified INTERFACE descriptor */
    0x05,                           /* Size of the descriptor, in bytes */
    DESC_CDC_CS_INTERFACE,          /* CS_INTERFACE descriptor type */
    DESC_CDC_HEADER,                /* Header functional descriptor subtype */
    0x10, 0x01,  /* Communication device compliant to the communication spec. ver. 1.10 */

    /* Communication Class Specified INTERFACE descriptor */
    0x05,                           /* Size of the descriptor, in bytes */
    DESC_CDC_CS_INTERFACE,          /* CS_INTERFACE descriptor type */
    DESC_CDC_CALL_MANAGEMENT,       /* Call management functional descriptor */
    0x00,                      /* BIT0: Whether device handle call management itself. */
    /* BIT1: Whether device can send/receive call management information over a Data Class
Interface 0 */
    INTF_NUM_VCOM0 + 1,           /* Interface number of data class interface optionally
used for call management */

    /* Communication Class Specified INTERFACE descriptor */
    0x04,                             /* Size of the descriptor, in bytes */
    DESC_CDC_CS_INTERFACE,            /* CS_INTERFACE descriptor type */
    DESC_CDC_ABSTRACT_CONTROL_MODEL,  /* Abstract control management functional
descriptor subtype */
    0x00,                             /* bmCapabilities       */

    /* Union Functional descriptor */
    0x05,                             /* bLength              */
    DESC_CDC_CS_INTERFACE,            /* bDescriptorType: CS_INTERFACE descriptor type */
    DESC_CDC_UNION,                   /* bDescriptorSubType:Union Functional Descriptor*/
    INTF_NUM_VCOM0,                   /* bMasterInterface     */
    INTF_NUM_VCOM0 + 1,               /* bSlaveInterface0     */
//END  ILEN_VCOM_COMMAND_INTR

    /* ENDPOINT descriptor */
    LEN_ENDPOINT,                     /* bLength          */
    DESC_ENDPOINT,                    /* bDescriptorType  */
    (EP_INPUT | INT_IN_EP_NUM_0),     /* bEndpointAddress */
    EP_INT,                           /* bmAttributes     */
    GET_BYTE0(EPC_MAX_PKT_SIZE),
    GET_BYTE1(EPC_MAX_PKT_SIZE),      /* wMaxPacketSize   */
    0x01,                             /* bInterval:1 ms   */

    /* INTERFACE descriptor */
    LEN_INTERFACE,      /* bLength              */
    DESC_INTERFACE,     /* bDescriptorType      */
    INTF_NUM_VCOM0 + 1, /* bInterfaceNumber     */
    0x00,               /* bAlternateSetting    */
    0x02,               /* bNumEndpoints        */
    0x0A,               /* bInterfaceClass:CDC-Data */
    0x00,               /* bInterfaceSubClass   */
    0x00,               /* bInterfaceProtocol   */
    0x00,               /* iInterface           */

    /* ENDPOINT descriptor */
    LEN_ENDPOINT,                     /* bLength          */
    DESC_ENDPOINT,                    /* bDescriptorType  */
    (EP_INPUT | BULK_IN_EP_NUM_0),    /* bEndpointAddress */
    EP_BULK,                          /* bmAttributes     */
```

```
    GET_BYTE0(EPA_MAX_PKT_SIZE),
    GET_BYTE1(EPA_MAX_PKT_SIZE),        /* wMaxPacketSize   */
    0x00,                               /* bInterval        */

    /* ENDPOINT descriptor */
    LEN_ENDPOINT,                       /* bLength          */
    DESC_ENDPOINT,                      /* bDescriptorType  */
    (EP_OUTPUT | BULK_OUT_EP_NUM_0),    /* bEndpointAddress */
    EP_BULK,                            /* bmAttributes     */
    GET_BYTE0(EPB_MAX_PKT_SIZE),
    GET_BYTE1(EPB_MAX_PKT_SIZE),        /* wMaxPacketSize   */
    0x00,                               /* bInterval        */
#endif
}
```

The *vcom_serial.c* file is responsible for the handshake between USB device and computer, and for transmitting the USB data. The sending interrupt *EPA_Handle ()* and receiving interrupt *EPB_Handle ()* for VCOM0 are as follows. The sending interrupt indicates that the USB cache data has been sent, and the next packet data can be sent. The receiving interrupt is responsible for copying the USB cache data to the RAM cache.

```
//-----------------VCOM0------------------//
void EPA_Handle(void)
{   //data out mcu
    volatile uint32_t IrqSt = HSUSBD->EP[EPA]. EPINTSTS & HSUSBD->EP[EPA]. EPINTEN;
#if (VCOM_CNT>=1)
    volatile struct vcom_var_t *vtp=&vcom_dat[IDX_VCOM0];
    vtp->gu32TxSize = 0;
#endif
    HSUSBD_ENABLE_EP_INT(EPA, 0);
    HSUSBD_CLR_EP_INT_FLAG(EPA, IrqSt);
}

void EPB_Handle(void)
{   //data in mcu
    volatile uint32_t IrqSt = HSUSBD->EP[EPB]. EPINTSTS & HSUSBD->EP[EPB]. EPINTEN;
#if (VCOM_CNT>=1)
    volatile struct vcom_var_t *vtp=&vcom_dat[IDX_VCOM0];
    int volatile i;
    vtp->gu32RxSize = HSUSBD->EP[EPB]. EPDATCNT & 0xffff;

    //pc_vcom -> usb -> mcu
    for(i = 0; i < vtp->gu32RxSize; i++)
        com_putchar(HSUSBD->EP[EPB]. EPDAT_BYTE,IDX_VCOM0);

    /* Set a flag to indicate bulk out ready */
    vtp->gi8BulkOutReady = 1;
#endif
    HSUSBD_CLR_EP_INT_FLAG(EPB, IrqSt);
}
```

The *VCOMapp.c* file contains the initialization of the USB interface and the processing functions of sending and receiving data through the RAM cache. The main way to send and receive data is to use functions like *putchar()* and *getchar()* to drop data into and retrieve data from the cache area. The file contains 4 types of cache function operations from USB cache to RAM cache,

RAM cache to USB cache, SLCAN cache to RAM cache and RAM cache to SLCAN cache. The *com_putchar()* and *com_getchar()* functions of the COM port in this example are as follows.

```
/**
 * @brief       pc com send to usb recv in mcu
 * @param[in]   bInChar    : the data will be send
                vcom       : com index
 * @return      1 byte send data,if error return -1
 * @details       pc_vcom -> usb -> mcu
 */
int32_t com_putchar(char bInChar,uint8_t vcom)
{
    volatile struct vcom_var_t *vtp;
    vtp=&vcom_dat[vcom];
    if(vtp->comTbytes < TXBUFSIZE)
    {
        /* Enqueue the character */
        vtp->comTbuf[vtp->comTtail++] = bInChar;
        if(vtp->comTtail >= TXBUFSIZE)
            vtp->comTtail = 0;
        vtp->comTbytes++;
    }
    else
    {
        return -1;
    }
    return bInChar;
}

/**
 * @brief       get 1 byte data from mcu send buffer to vcom
 * @param[in]     vcom :com index
 * @return      1 byte data,if error return -1
 * @details       mcu -> usb -> pc_vcom
 */
int32_t  com_getchar(uint8_t vcom)
{
    uint8_t bInChar;
    volatile struct vcom_var_t *vtp;
    vtp=&vcom_dat[vcom];
    if (vtp->comRbytes)
    {
        bInChar = vtp->comRbuf[vtp->comRhead++];
        if(vtp->comRhead >= RXBUFSIZE)
            vtp->comRhead = 0;
        vtp->comRbytes--;
    }
    else return -1;

    return bInChar;
}
```

The protocol conversion is mainly performed by the *slcan.c* file, which converts the data received by CAN into SLCAN protocol and sends it to VCOM, and converts the SLCAN data

form VCOM into CAN messages and sends them to CAN bus. The *slcan.c* file is also responsible for parsing SLCAN commands.

```
/**
 * @brief       both can and process
 * @param[in]   slcan_port : one can port will to be operated
 * @return
 * @details
 */
void slcan_process_task(struct slcan_t* slcan_port)
{
    slcan_process_can(slcan_port);      //data from CAN FD to VCOM
    slcan_process_uart(slcan_port);     //data from VCOM  to CAN FD
}
```

2.3 Operation Instance

In this example, use the same baud rate setting, and test 1 CAN send and 3 CAN receive. COM46 is responsible for sending data, and COM45, COM14, and COM15 ports are responsible for receiving data.

Before executing, you need to confirm that the driver name of the VCOM device has been modified according to the Nuvoton VCOM name. Please refer to the *EC_M460_HSUSBD_VCOM_MultiPort_V1.00* example readme file for details.

Open the APP to set the CAN baud rate, and the execution log is as follows:

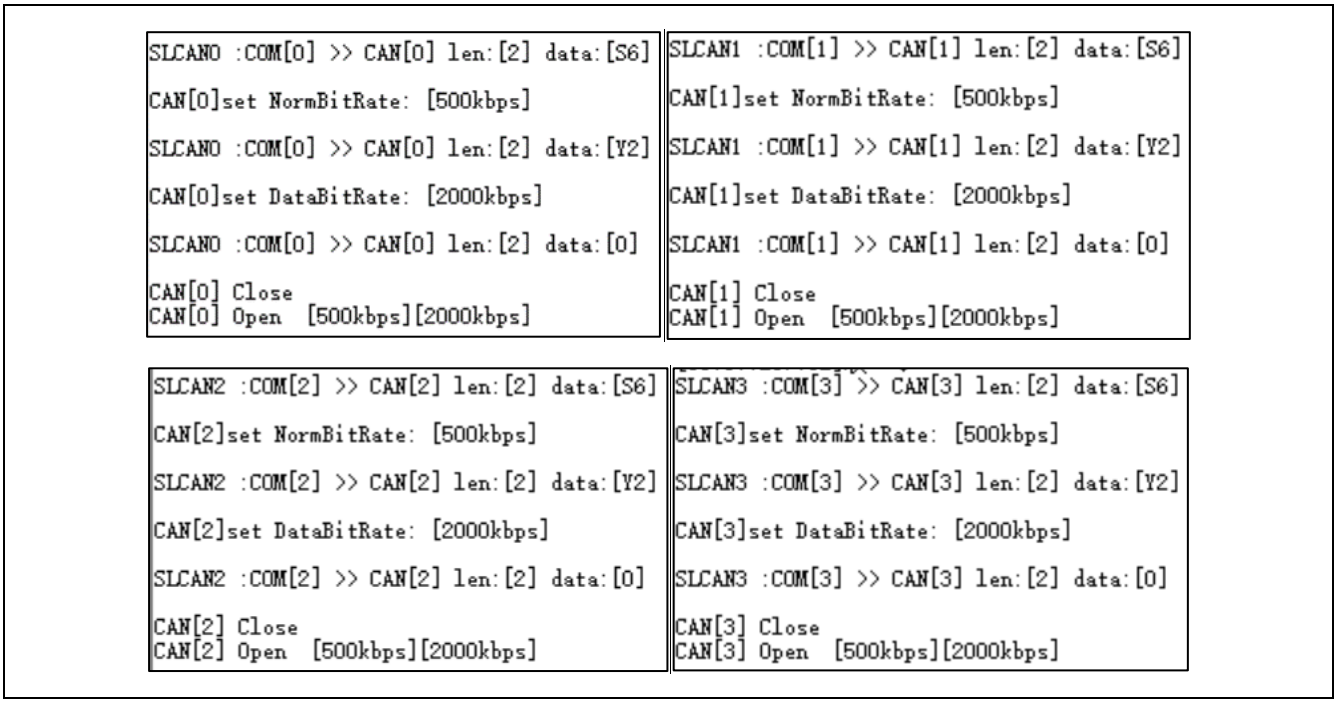

Figure 2-4 CAN Port Baud Rate Setting Log

The baud rate of the 4 sets of CAN arbitration domains is 500 kbps, and the baud rate of the variable speed frame data segment is 2 Mbps.

The APP selects “COM46 Nuvoton VCOM0 Port” to send a frame of data:

CAN FD standard frame, 64 bytes data, non-variable baud rate frames, data transmission and receiving are shown as Figure 2-5:

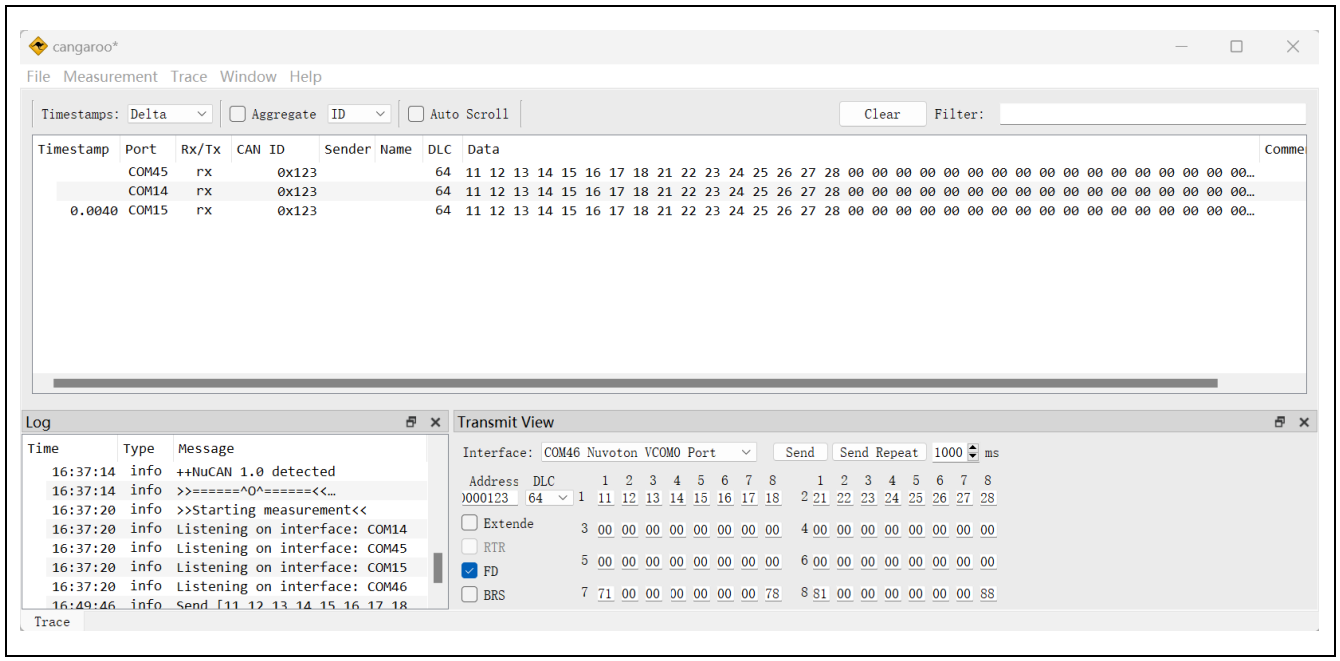

Figure 2-5 Upper Computer Data Transmission and Reception Diagram

From Figure 2-5, it can be seen that each of the COM45, COM14, and COM15 ports receives a frame of data, and the content is the same as the data sent by COM46.

The log of data sent and received is as follows:

```
SLCAN0 :COM[0] >> CAN[0] len:[134] data:
[b123F111213141516171821222324252627280000000000000000000000000000000000000000000000000000000000000000000000000000000007100
000000000078810000000000000088]

SLCAN2 :COM[2] << CAN[2] len:[134] data:
[b123F111213141516171821222324252627280000000000000000000000000000000000000000000000000000000000000000000000000000000007100
000000000078810000000000000088]

SLCAN3 :COM[3] << CAN[3] len:[134] data:
[b123F111213141516171821222324252627280000000000000000000000000000000000000000000000000000000000000000000000000000000007100
000000000078810000000000000088]

SLCAN1 :COM[1] << CAN[1] len:[134] data:
[b123F111213141516171821222324252627280000000000000000000000000000000000000000000000000000000000000000000000000000000007100
000000000078810000000000000088]
```

Figure 2–6 Execution Logs for Sent and Received Data

- COM[0] >> CAN[0] means that data is sent from VCOM0 to the CAN0 bus.
- COM[2] << CAN[2] means that CAN2 received data from the bus and sent it to the VCOM2 port.
- COM[3] << CAN[3] means that CAN3 received data from the bus and sent it to the VCOM3 port.

- COM[1] << CAN[1] means that CAN1 received data from the bus and sent it to the VCOM1 port.

Protocol Analysis: (For more commands refer to Figure 1-1)

“len:[134]” indicates that the data quantity of the SLCAN protocol is 134 bytes.

The 134 Bytes SLCAN protocol includes:

Command Code:	b	->	CAN FD standard frame data (Reference Figure 1-1)
Frame ID：	123	->	0x123
Data quantity:	F	->	64 bytes (Reference Figure 1-1)

Frame data (64 Bytes):

0x 11 12 13 14 15 16 17 18
21 22 23 24 25 26 27 28
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
71 00 00 00 00 00 00 78
81 00 00 00 00 00 00 88

The data parsed in the log is the same as the data sent and received. The test is completed.

3. Software & Hardware Requirements

3.1 Software Requirements

- BSP version
  - M460_Series_BSP_CMSIS_V3.00.002
- IDE version
  - Keil uVersion 5.30
- 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.

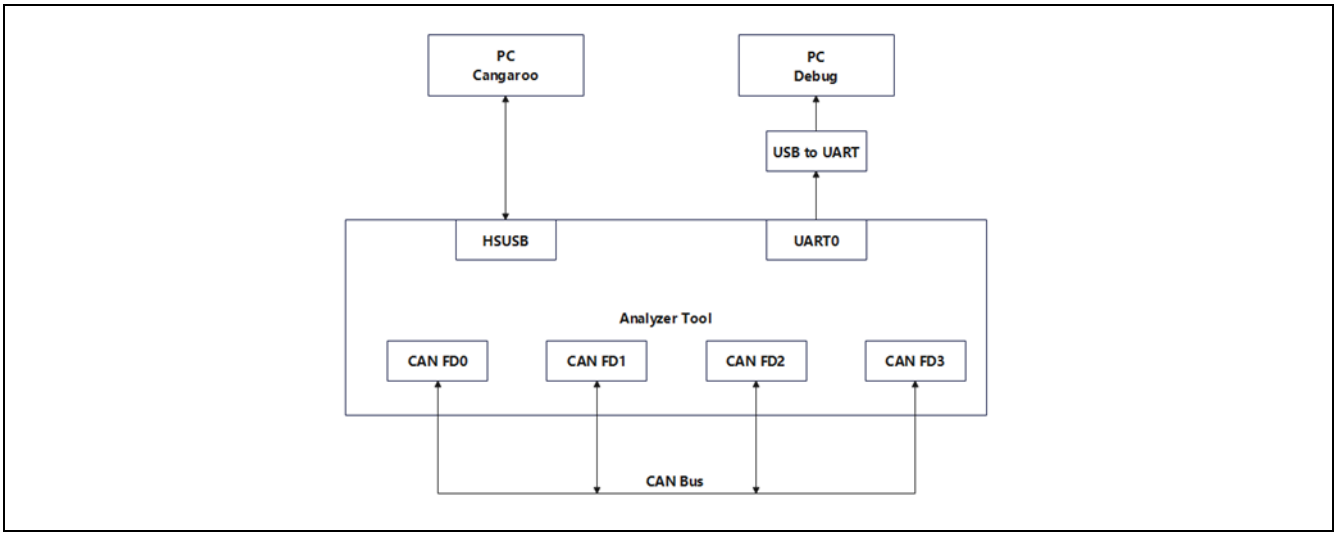

Figure 3-1 Pin Connect

Figure 3-2 Physical Wiring

4. Directory information

EC_M460_4_CAN_FD_to_SLCAN_by_VCOM_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	
Project	Source file of example code

Figure 4-1 Directory Structure

5. Example Code Execution

1. Browse the sample code folder as described in the Directory Information section and double-click *4_CAN_FD_to_SLCAN_by_VCOM.uvprojx*.
2. Enter Keil compile mode.

- Build
- Download
- Start/Stop debug session

3. Enter debug mode.

- Run

6. Revision History

Date	Revision	Description
2024.06.18	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.