M253 transmits and receives CAN protocol through USB

Example Code Introduction for 32-bit NuMicro® Family

Document Information

Application	This example code uses the M253 series to communicate with the MCU through USB and to transmit and receive CAN protocol data.
BSP Version	M253_Series_BSP_CMSIS_V3.00.010
Hardware	NuMaker-M253LE3AE-V1.1

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

This example code demonstrates how to communicate with a host through the USB interface of the M253 series to transmit and receive CAN protocol data, and provides operating instructions for Linux environment. After the M253 receives commands from the host, it generates signals on the CAN pins. The on-board CAN transceiver (ADM3050E) on the NuMaker-M253LE converts the CAN logic signals into CANH/CANL differential signals to communicate with other CAN devices.

Similarly, for reception, when CAN signals from other devices are received, they are converted by the transceiver and passed to the MCU, which then reports the received command data back to the host.

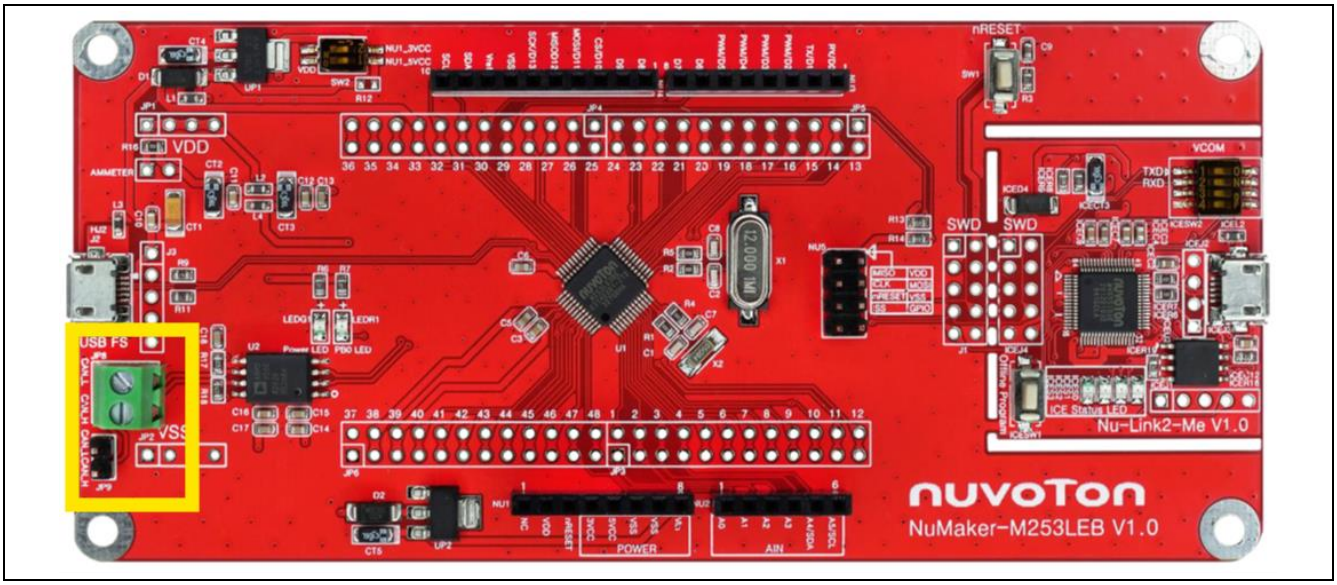


Figure 1-1 CAN pin locations on the NuMaker-M253LE development board

1.1 Principle

In Linux, SocketCAN is natively supported. It is maintained by the Linux kernel and treats the CAN bus as a network device, allowing applications to communicate using the standard BSD (Berkeley Software Distribution) Socket API without the need to write additional low-level drivers.

Linux provides ready-to-use CAN utilities. This example uses commands from can-utils to operate the CAN interface. Before using them, you need to install can-utils (installation command: sudo apt-get install can-utils).

The commands used in this example include:

candump: Monitors and displays all messages currently on the CAN bus.

cansend: Sends a single CAN frame. In this application, the maximum payload size is limited to 8 bytes.

In addition to the communication commands, the device must be connected and the communication speed configured before use. The following commands are used:

ip link set <device> down: Sets the current device state to down. This command is optional; however, when configuring the CAN device bitrate (the command below), the device must be in the down state. Therefore, this command is used to set the device state to down.

ip link set <device> type can bitrate 500000: Sets the bitrate of the current CAN device. This command cannot be executed while the device state is up.

ip link set <device> up: Sets the current device state to up. Use this command to enable the device after configuring the bitrate.

The above commands may encounter permission issues, so it is recommended to prefix them with sudo. The name depends on the actual <device> name (e.g., can0, can1, etc.). You can verify the device name using the following command:

ip link list type can: Lists the CAN devices currently connected to the host.

Figure 1-2 uses two M253 development boards: one configured as device can0 and the other as device can1.

Device can1 sends a command to can0 with ID 0x611 and packet 0x1122334455667788.

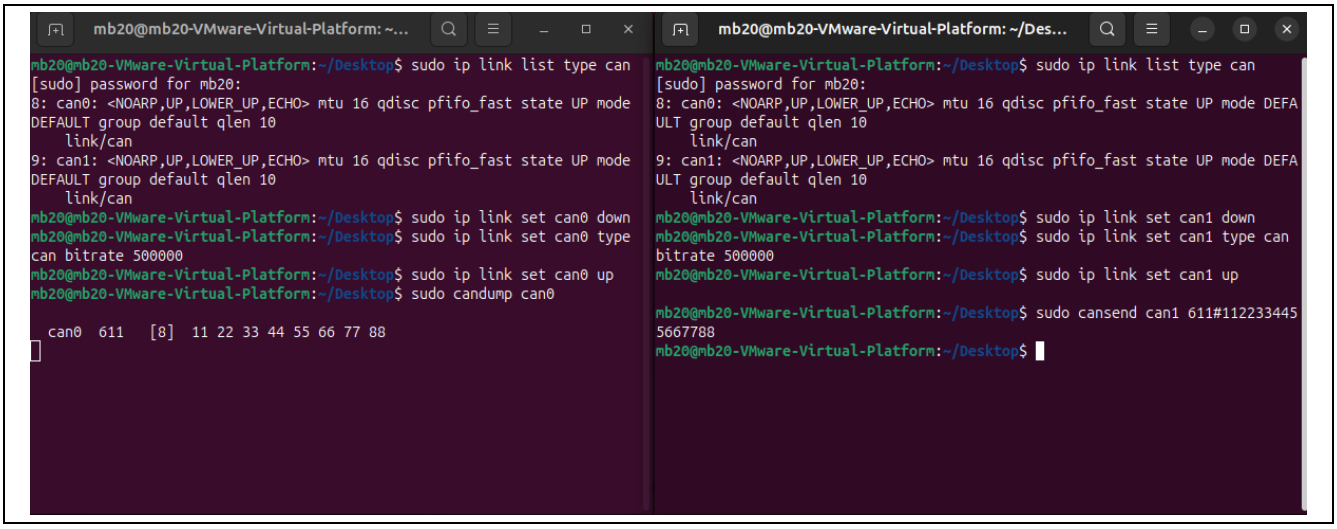


Figure 1-2 Two M253 devices communicate with each other through CAN in Linux environment

1.2 Demo Result

The compilation and programming results of this example code are shown in Figure 1-3 and Figure 1-4.

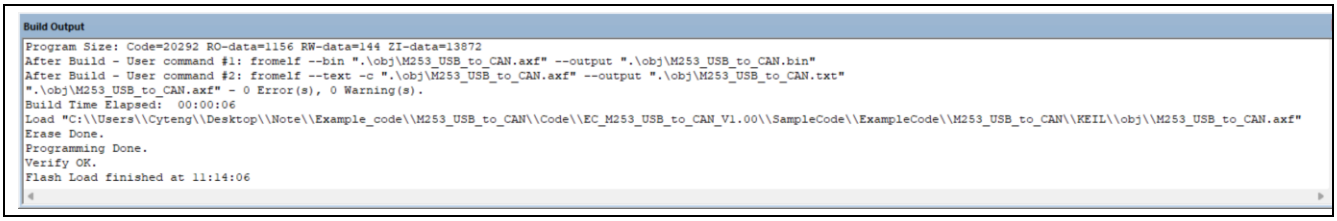


Figure 1-3 Compilation and Programming Results (Keil)



Figure 1-4 Compilation and Programming Results (VScode)

2. Code Description

The system initialization code is located in *main.c*. After completing system and USB initialization, the program further configures the CAN module and the MCU's internal timer counting mechanism.

```
int32_t main(void)
{
    /* Unlock protected registers */
    SYS_UnlockReg();

    /* Init system and multi-function I/O */
    SYS_Init();

    /* Init UART MFP */
    UartDebugInit();
```

Enable the USB function, configure the data to be returned to the host, and finally enable USB interrupts.

```
    /* Open USB controller */
    USBD_Open(&gsInfo, VCOM_ClassRequest, NULL);
    USBD_SetVendorRequest(VCOM_VendorRequest);
    /* Endpoint configuration */
    VCOM_Init();
    /* Start USB device */
    USBD_Start();
    NVIC_EnableIRQ(USBD_IRQn);
```

Following the aforementioned initialization process, the program initializes the CAN-related commands and the MCU's internal timer counting mechanism.

```
    pcan_timestamp_init();
```

The content of the pcan_timestamp_init function, located in pcanpro_timestamp.c. This function enables Timer0 for system time counting and triggers a Timer0 interrupt at a 1-millisecond interval to update system time–related variables, which are used for subsequent communication timing calculations.

```
void pcan_timestamp_init(void)
{
    TIMER0->CTL =  TIMER_PERIODIC_MODE | 47;
    TIMER0->CMP = 1000;
    TIMER_EnableInt(TIMER0);
    NVIC_EnableIRQ(TMR0_IRQn);
    TIMER_Start(TIMER0);
}
```

The interrupt service function TMR0_IRQHandler, located in pcanpro_timestamp.c. Each time the interrupt handler is entered, it increments g_au32TMRINTCount. This variable serves as a time reference for communication timeout determination.

```
void TMR0_IRQHandler(void)
{
    if (TIMER_GetIntFlag(TIMER0))
    {
        /* Clear Timer0 time-out interrupt flag */
```

```
        TIMER_ClearIntFlag(TIMER0);

        g_au32TMRINTCount++;
    }
}
```

Main program content: this section allows the system clock to be calibrated using USB, thereby reducing clock error. This code is included in all USB-related sample code. For detailed information, please refer to the M253 Technical Reference Manual M253 Series - Nuvoton Chapter 6.2.8 IRC Auto Trim.

```
        /* Start USB trim function if it is not enabled. */
        if ((SYS->HIRCTRIMCTL & SYS_HIRCTRIMCTL_FREQSEL_Msk) != 0x1)
        {
            /* Start USB trim only when USB signal arrived */
            if (USBD->INTSTS & USBD_INTSTS_SOFIF_Msk)
            {
                /* Clear SOF */
                USBD_CLR_INT_FLAG(USBD_INTSTS_SOFIF_Msk);

                /*
                    USB clock trim function:
                    HIRC Trimming with boundary function enhances robustility
                    and keeps HIRC in right frequency while receiving unstable USB signal
                */
                SYS->HIRCTRIMCTL = (0x1 << SYS_HIRCTRIMCTL_REFCKSEL_Pos)
                                 | (0x1 << SYS_HIRCTRIMCTL_FREQSEL_Pos)
                                 | (0x0 << SYS_HIRCTRIMCTL_LOOPSEL_Pos)
                                 | (0x1 << SYS_HIRCTRIMCTL_BOUNDEN_Pos)
                                 | (10  << SYS_HIRCTRIMCTL_BOUNDARY_Pos);
            }
        }

        /* Disable USB Trim when any error found */
        if (SYS->HIRCTRIMSTS & (SYS_HIRCTRIMSTS_CLKERIF_Msk |
SYS_HIRCTRIMSTS_TFAILIF_Msk))
        {
            /* Init TRIM */
            M32(TRIM_INIT) = u32TrimInit;

            /* Disable USB clock trim function */
            SYS->HIRCTRIMCTL = 0;

            /* Clear trim error flags */
            SYS->HIRCTRIMSTS = SYS_HIRCTRIMSTS_CLKERIF_Msk | SYS_HIRCTRIMSTS_TFAILIF_Msk;

            /* Clear SOF */
            USBD_CLR_INT_FLAG(USBD_INTSTS_SOFIF_Msk);
        }
```

Continuing from the above, the variable g_u8Suspend is modified within the USB interrupt. When a Suspend event is received, the system enters Power-down mode; at other times, it enters a state where it monitors CAN messages from the host.

```
        /* Enter power down when USB suspend */
        if (g_u8Suspend)
           PowerDown();
        pcan_protocol_poll();
```

The function pcan_protocol_poll, located in pcanpro_fd_protocol.c. It is the main polling function responsible for receiving, packaging, and returning data between USB and CAN.

The main processing flow includes:

- Retrieve the current system time in milliseconds and microseconds

  The two time-related functions, pcan_timestamp_millis and pcan_timestamp_us, both return the variable g_au32TMRINTCount, which is updated in the Timer0 interrupt handler to record the MCU's current count value. The only difference between them is the time unit of the returned value.

- Process CAN receive and transmit data

  The function uses the variable g_u8RxFifO0CompleteFlag to determine whether a CAN receive interrupt has been triggered. A CAN interrupt occurs when CAN data is received or when a timeout happens. When data is received, the CAN data is stored in a structure within the write_manual_rx_frame function and waits to be processed. The function pcan_can_poll checks whether a command to transmit CAN data has been received. If a transmit command is detected, response data is sent back to the host via USB.

- Return the received CAN data to the host via USB Bulk-IN

  The flush data section handles reporting CAN-received data to the host via USB Bulk-IN packets. This is processed by the pcan_flush_data function. Only Endpoint 2 Bulk-IN is handled; requests to other endpoints will return an error.

- Periodically update CAN bus load and time-synchronization information

  Every 250 ms, the internal parameters are updated to calculate the proportion of time used for data transmission relative to the total bus time. Two memory areas are allocated to store the calculation results (pointer pbs) and the status parameters (pointer pts, e.g., the last sync time). Before performing the update, the system first checks whether the device has received a Vendor Request command; only then does execution continue.

```
void pcan_protocol_poll(void)
{
    uint32_t ts_ms = pcan_timestamp_millis();
    uint32_t ts_us = pcan_timestamp_us();
    write_manual_rx_frame();
    pcan_can_poll();

    /* flush data */
    if (data_pos > 0)
    {
        /* endmark */
        *(uint32_t *)&data_buffer[data_pos] = 0x00000000;
        uint16_t flush_size = data_pos + 4;
        /* align to 64 */
        flush_size += (64 - 1);
        flush_size &= ~(64 - 1);
        int res = pcan_flush_data(&resp_fsm[1], data_buffer, flush_size);

        if (res)
        {
            data_pos = 0;
            pcan_device.last_time_flush = ts_us;
        }
    }
    /* timesync part */
    if (!pcan_device.can_drv_loaded)
```

```
        return;

    /* update bus load each 250ms */
    if ((ts_ms - pcan_device.last_bus_load_update) >= 250u)
    {
        pcan_device.last_bus_load_update = ts_ms;

        for (int i = 0; i < CAN_CHANNEL_MAX; i++)
        {
            uint32_t total_ns = pcan_device.can[i].tx_time_ns +
pcan_device.can[i].rx_time_ns;
            /* get bus in percents 0 - 100% */
            pcan_device.can[i].bus_load = total_ns / (250000000u / 100u);

            pcan_device.can[i].tx_time_ns = 0;
            pcan_device.can[i].rx_time_ns = 0;
        }
    }

    if ((ts_ms - pcan_device.last_time_sync) < 1000u)
        return;

    struct ucan_usb_ts_msg *pts = pcan_data_alloc_buffer(UCAN_USB_MSG_CALIBRATION,
sizeof(struct ucan_usb_ts_msg));

    if (!pts)
        return;

    pts->usb_frame_index = pcan_usb_frame_number();
    pts->unused = 0;
    pcan_device.last_time_sync = ts_ms;

    for (int i = 0; i < CAN_CHANNEL_MAX; i++)
    {
        if (!(pcan_device.can[i].opt_mask & UCAN_OPTION_BUSLOAD))
            continue;

        if (!pcan_device.can[i].bus_active)
            continue;

        struct ucan_bus_load_msg *pbs = pcan_data_alloc_buffer(UCAN_MSG_BUSLOAD,
sizeof(struct ucan_bus_load_msg));

        if (!pbs)
            return;

        pbs->channel = i;
        /* 0 ... 100% => 0 ... 4095 */
        pbs->bus_load = (pcan_device.can[i].bus_load * 4095u) / 100u;
    }

}
```

The function write_manual_rx_frame, first checks whether a CAN interrupt has occurred. If the interrupt has been triggered, it indicates that CAN data has been received, and g_u8RxFifO0CompleteFlag will be set to 1. This function updates the parameters of the manual_msg structure. The structure g_sRxMsgFrame is updated in the CAN interrupt handler, and within this function it is copied into manual_msg. After manual_msg is fully updated, the function calls pcan_protocol_rx_frame to process the received data.

```
void write_manual_rx_frame(void)
{
    if (g_u8RxFifO0CompleteFlag != 1)
        return ;

    g_u8RxFifO0CompleteFlag = 0;
```

```
    struct t_can_msg manual_msg = { 0 };

    manual_msg.id = g_sRxMsgFrame.u32Id;

    manual_msg.size = g_sRxMsgFrame.u32DLC;

    for (int u8Cnt = 0; u8Cnt <  g_sRxMsgFrame.u32DLC; u8Cnt++)
    {
        manual_msg.data[u8Cnt] = g_sRxMsgFrame.au8Data[u8Cnt];
    }

    if (g_sRxMsgFrame.eIdType == eCANFD_SID)
        manual_msg.flags = MSG_FLAG_STD;
    else
        manual_msg.flags = MSG_FLAG_EXT;

    manual_msg.timestamp = pcan_timestamp_us();

    uint8_t target_channel = CAN_BUS_1;
}

    pcan_protocol_rx_frame(CAN_BUS_1, &manual_msg);
```

The function pcan_protocol_rx_frame, first determines the CAN protocol DLC value based on the received result. It then stores the received CAN data into a buffer and calculates the nominal and data bit times. Finally, the data is copied into the allocated memory space (pointer pcan_msg).

```
int pcan_protocol_rx_frame(uint8_t channel, struct t_can_msg *pmsg)
{
    static const uint8_t pcan_fd_len2dlc[] =
    {
        0, 1, 2, 3, 4, 5, 6, 7, 8,  /* 0 - 8 */
        9, 9, 9, 9,     /* 9 - 12 */
        10, 10, 10, 10,     /* 13 - 16 */
        11, 11, 11, 11,     /* 17 - 20 */
        12, 12, 12, 12,     /* 21 - 24 */
        13, 13, 13, 13, 13, 13, 13, 13, /* 25 - 32 */
        14, 14, 14, 14, 14, 14, 14, 14, /* 33 - 40 */
        14, 14, 14, 14, 14, 14, 14, 14, /* 41 - 48 */
        15, 15, 15, 15, 15, 15, 15, 15, /* 49 - 56 */
        15, 15, 15, 15, 15, 15, 15, 15  /* 57 - 64 */
    };
    struct ucan_rx_msg *pcan_msg = pcan_data_alloc_buffer(UCAN_MSG_CAN_RX, sizeof(struct
ucan_rx_msg) + pmsg->size);

    if (!pcan_msg)
        return -1;

    if (!(pmsg->flags & MSG_FLAG_ECHO))
    {
        uint32_t nqt = pcan_device.can[channel].nominal_qt;
        uint32_t dqt = pcan_device.can[channel].data_qt;
        pcan_device.can[channel].rx_time_ns += pcan_can_msg_time(pmsg, nqt, dqt);

        if (!pcan_device.can[channel].led_is_busy)
        {
            // pcan_led_set_mode( channel ? LED_CH1_RX:LED_CH0_RX, LED_MODE_BLINK_FAST,
237 );
            pcan_led_set_mode(LED_CH0_RX, LED_MODE_BLINK_FAST, 237);
        }
    }

    if (pmsg->size > CAN_PAYLOAD_MAX_SIZE)
        pmsg->size = CAN_PAYLOAD_MAX_SIZE;
```

```
    pcan_msg->channel_dlc = UCAN_MSG_CHANNEL_DLC(channel, pcan_fd_len2dlc[pmsg->size]);
    pcan_msg->client = pmsg->dummy;
    pcan_msg->flags = 0;
    pcan_msg->tag_low = 0;
    pcan_msg->tag_high = 0;

    /* we support only regular frames */
    if (pmsg->flags & MSG_FLAG_RTR)
        pcan_msg->flags |= UCAN_MSG_RTR;

    if (pmsg->flags & MSG_FLAG_EXT)
        pcan_msg->flags |= UCAN_MSG_EXT_ID;

    if (pmsg->flags & MSG_FLAG_FD)
    {
        pcan_msg->flags |= UCAN_MSG_EXT_DATA_LEN;

        if (pmsg->flags & MSG_FLAG_BRS)
            pcan_msg->flags |= UCAN_MSG_BITRATE_SWITCH;

        if (pmsg->flags & MSG_FLAG_ESI)
            pcan_msg->flags |= UCAN_MSG_ERROR_STATE_IND;
    }

    if (pmsg->flags & MSG_FLAG_ECHO)
    {
        pcan_msg->flags |= UCAN_MSG_API_SRR | UCAN_MSG_HW_SRR;
        pcan_msg->ts_low = pcan_timestamp_us();
    }
    else
    {
        pcan_msg->ts_low = pmsg->timestamp;
    }

    pcan_msg->can_id = pmsg->id;
    memcpy(pcan_msg->d, pmsg->data, pmsg->size);
    return 0;
}
```

The function pcan_can_poll, in addition to performing counting operations, calls the pcan_can_flush_tx function.

```
void pcan_can_poll(void)
{
    static uint32_t err_last_check = 0;
    uint32_t ts_ms;

    ts_ms = pcan_timestamp_millis();


    pcan_can_flush_tx(CAN_BUS_1);
}
```

The function pcan_can_flush_tx sends the TX FIFO via _can_send. At the end, it updates the TX FIFO position (tx_tail).

```
static void pcan_can_flush_tx(int bus)
{
    struct t_can_dev *p_dev = &can_dev_array[bus];
    struct t_can_msg *p_msg;

    /* empty fifo */
    if (p_dev->tx_head == p_dev->tx_tail)
        return;
```

```
    p_msg = &p_dev->tx_fifo[p_dev->tx_tail];

    _can_send(p_msg);

    /* update fifo index */
    p_dev->tx_tail = (p_dev->tx_tail + 1) & (CAN_TX_FIFO_SIZE - 1);
}
```

The function _can_send configures the transmission details and then hands the data off to the function CANFD_TransmitTxMsg in canfd.c (part of the library) to transmit it. In the M253 BSP, the CAN sample code includes this configuration step before every CAN transmission.

```
static void _can_send(struct t_can_msg *p_msg)
{
    CANFD_FD_MSG_T      sTxMsgFrame;

    if (p_msg->flags & MSG_FLAG_EXT)
    {
        sTxMsgFrame.u32Id  = p_msg->id & 0x1FFFFFFF;
        sTxMsgFrame.eIdType = eCANFD_XID;
    }
    else
    {
        sTxMsgFrame.u32Id  = p_msg->id & 0x7FF;
        sTxMsgFrame.eIdType = eCANFD_SID;
    }

    /*Set the frame type*/

    /*Set the bitrate switch */
    sTxMsgFrame.bBitRateSwitch = 0;
    sTxMsgFrame.u32DLC  = p_msg->size;

    if (p_msg->flags & MSG_FLAG_RTR)
        sTxMsgFrame.eFrmType = eCANFD_REMOTE_FRM;
    else
        sTxMsgFrame.eFrmType = eCANFD_DATA_FRM;

    for (int u8Cnt = 0; u8Cnt < sTxMsgFrame.u32DLC; u8Cnt++)
        sTxMsgFrame.au8Data[u8Cnt] = p_msg->data[u8Cnt];

    if (CANFD_TransmitTxMsg(CANFD0, 0, &sTxMsgFrame) != eCANFD_TRANSMIT_SUCCESS)
    {
        printf("Failed to transmit message\n");
    }
}
```

USB interrupt handler description. When the device is connected, the USB interrupt handler declares the characteristics of the device's endpoints. This code defines two endpoints for receiving data from the host and endpoints for sending data back to the host. When the device receives a Bulk command from the host on an endpoint, it enters the USB interrupt handler. The command content includes both device initialization/configuration commands and CAN transmission data, so two endpoint interrupts are used to handle them separately.

The content of the EP3_Handler function. This function is called by the interrupt service routine. When a Bulk-OUT transfer is sent to Endpoint 1, execution is directed to EP3_Handler. Its purpose is to receive configuration and control commands from the host. ◦

```
void EP3_Handler(void)
```

```
{
    int size;
    size = USBD_GET_PAYLOAD_LEN(EP3);
    USBD_SET_PAYLOAD_LEN(EP3, EP3_MAX_PKT_SIZE);
    g_pu8RxBuf = (uint8_t *)(USBD_BUF_BASE + USBD_GET_EP_BUF_ADDR(EP3));

    for (int i = 0; i < size; i++)
        pcanpro_data.cmd_ep_buffer[i] = g_pu8RxBuf[i];

    pcan_protocol_process_data(PCAN_USB_EP_CMDOUT, pcanpro_data.cmd_ep_buffer, size);
}
```

The function EP5_Handler is invoked by an interrupt. When a Bulk-OUT transfer is sent to Endpoint 2, execution is directed to this function. Its purpose is to receive CAN transmission data from the host. Since the data is transferred in 512-byte chunks, it proceeds to call pcan_protocol_process_data.

```
void EP5_Handler(void)
{
    unsigned int data_len = 0;
    unsigned int i = 0;

    data_len = USBD_GET_PAYLOAD_LEN(EP5);
    USBD_SET_PAYLOAD_LEN(EP5, EP5_MAX_PKT_SIZE);

    if (data_len > EP5_MAX_PKT_SIZE)
    {
        DBG_PRINTF("Error! EP3 data overrun!\n");
        data_len = EP5_MAX_PKT_SIZE;
    }

    /* Read bulk-out packet from USBD SRAM to g_Bulk_Buff[] */
    for (i = 0; i < data_len; i++)
    {
        pcanpro_data.data1_ep_buffer[i + data_out_count] = inpb(USBD_BUF_BASE +
EP5_BUF_BASE + i);
    }

    data_out_count = data_out_count + data_len;

    if (data_out_count >= 512)
    {
        pcan_protocol_process_data(PCAN_USB_EP_MSGOUT_CH1, pcanpro_data.data1_ep_buffer,
data_out_count);
        data_out_count = 0;
    }
}
```

The function pcan_protocol_process_data behaves differently depending on whether it is called by EP3_Handler or EP5_Handler. If the ep is 1, the received data is a control command and the function enters pcan_protocol_process_cmd.

If the ep is 2, it enters pcan_protocol_tx_frame and updates the pcanpro_data structure.

```
void pcan_protocol_process_data(uint8_t ep, uint8_t *ptr, uint16_t size)
{
```

```
    if (ep == 1)
    {
        pcan_protocol_process_cmd(ptr, size);
        return;
    }

    /* message data  */
    struct ucan_msg *pmsg = 0;

    while (size)
    {
        if (size < 4)
            break;

        pmsg = (void *)ptr;

        if (!pmsg->size || !pmsg->type)
            break;

        if (size < pmsg->size)
            break;

        size -= pmsg->size;
        ptr += pmsg->size;

        switch (pmsg->type)
        {
            //to host only
            case UCAN_MSG_CAN_TX:
                pcan_protocol_tx_frame((struct ucan_tx_msg *)pmsg);
                break;

            case UCAN_MSG_CAN_TX_PAUSE:
                break;

            case UCAN_CMD_END_OF_COLLECTION:
            case 0xffff:
                return;

            default:
                MY_ASSERT(0);
                break;
        }
    }
}
```

The function pcan_protocol_process_cmd handles various commands used during connection setup.

The function pcan_protocol_tx_frame parses the data contained in the Bulk-OUT transfer and places it into can_dev_array[bus].tx_fifo using pcan_can_write.

```
int pcan_protocol_tx_frame(struct ucan_tx_msg *pmsg)
{
    static const uint8_t pcan_fd_dlc2len[] =
    {
```

```
        0, 1, 2, 3, 4, 5, 6, 7,
        8, 12, 16, 20, 24, 32, 48, 64
    };

    struct t_can_msg msg = { 0 };
    uint8_t channel;

    channel = UCAN_MSG_CHANNEL(pmsg);

    if (channel >= CAN_CHANNEL_MAX)
        return -1;

    msg.id = pmsg->can_id;

    /* CAN-FD frame */
    if (pmsg->flags & UCAN_MSG_EXT_DATA_LEN)
    {
        msg.flags |= MSG_FLAG_FD;

        if (pmsg->flags & UCAN_MSG_BITRATE_SWITCH)
            msg.flags |= MSG_FLAG_BRS;

        if (pmsg->flags & UCAN_MSG_ERROR_STATE_IND)
            msg.flags |= MSG_FLAG_ESI;

        msg.size = pcan_fd_dlc2len[UCAN_MSG_DLC(pmsg)];
    }
    else
    {
        msg.size = UCAN_MSG_DLC(pmsg);
    }

    if (msg.size > sizeof(msg.data))
        return -1;

    if (pmsg->flags & UCAN_MSG_RTR)
        msg.flags |= MSG_FLAG_RTR;

    if (pmsg->flags & UCAN_MSG_EXT_ID)
        msg.flags |= MSG_FLAG_EXT;

    if (pmsg->flags & (UCAN_MSG_HW_SRR))
    {
        msg.flags |= MSG_FLAG_ECHO;
        msg.dummy = pmsg->client;
    }

    memcpy(msg.data, pmsg->d, msg.size);

    msg.timestamp = pcan_timestamp_us();

    return 0;
}
```

The content of the pcan_can_write function.

```
int pcan_can_write(int bus, struct t_can_msg *p_msg)
{
    struct t_can_dev *p_dev = &can_dev_array[bus];

    if (!p_dev)
        return 0;

    if (!p_msg)
        return 0;

    uint32_t  tx_head_next = (p_dev->tx_head + 1) & (CAN_TX_FIFO_SIZE - 1);

    /* overflow ? just skip it */
    if (tx_head_next == p_dev->tx_tail)
    {
        ++p_dev->tx_ovfs;
        return -1;
    }

    p_dev->tx_fifo[p_dev->tx_head] = *p_msg;
    p_dev->tx_head = tx_head_next;

    return 0;
}
```

3. Software and Hardware Requirements

3.1 Software Requirements

- BSP version
  - M253_Series_BSP_CMSIS_V3.00.010
- IDE version
  - Keil uVision 5.42
  - Visual Studio Code 1.105.0

3.2 Hardware Requirements

- Circuit components
  - NuMaker-M253LE3AE-V1.1
- Pin Connect
  - Connect the USB pins to the PC, and connect the CAN pins on the development board to another CAN device.

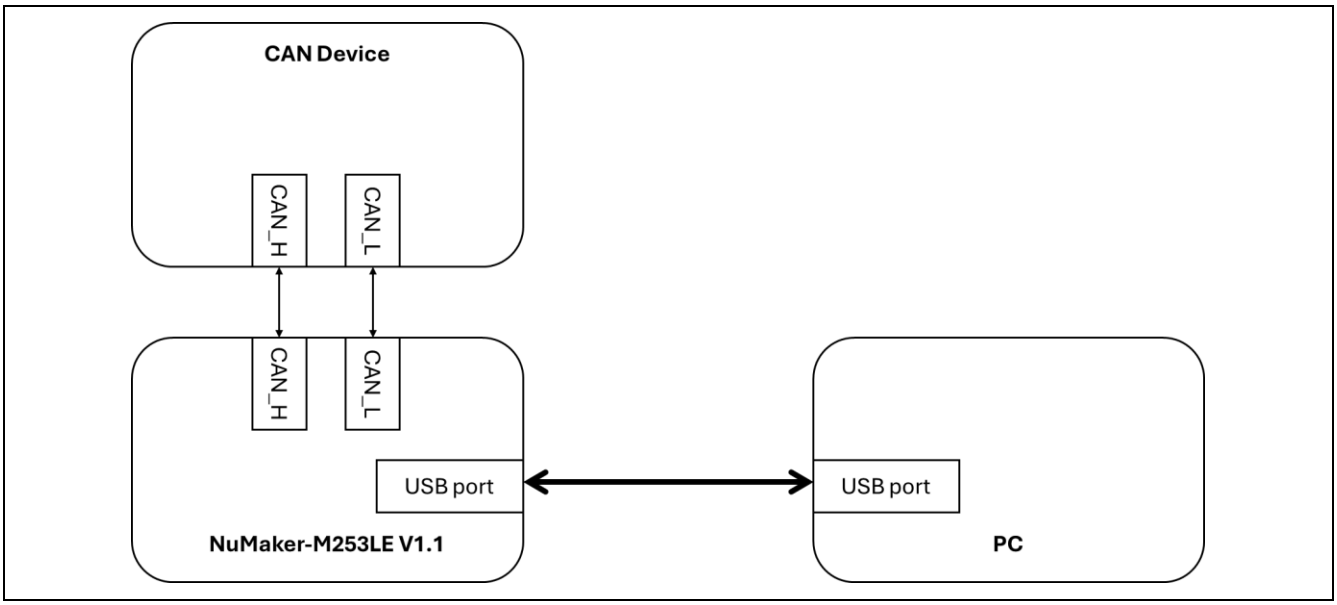


Figure 3-1 Pin Connect

4. Directory Information

The directory structure is shown below.

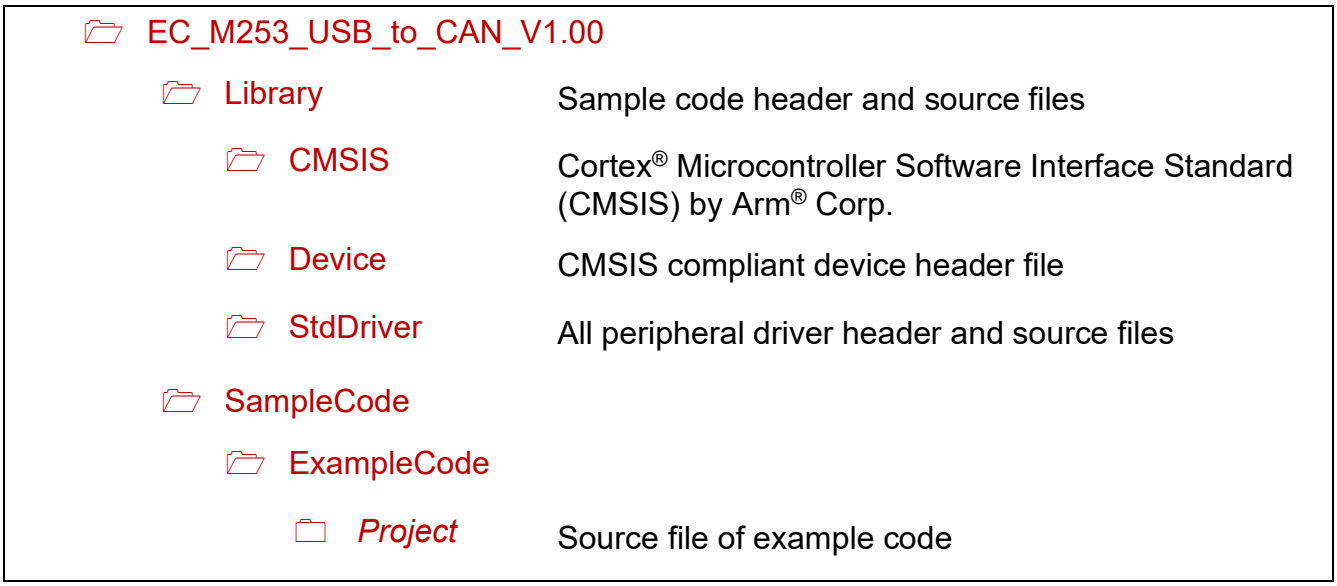


Figure 4-1 Directory Structure

5. Example Code Execution

5.1 Keil MDK

1. Navigate to the **Keil** project folder in Windows Explorer and open *M253_USB_to_CAN.uvprojx* project file.
2. Build
3. Download
4. Enter debug mode
   - Run

5.2 VS Code

1. Select **File** in **VS Code** IDE
   - Click **Open Folder**
   - Select the **VSCode** folder under the project directory
2. Select **CMSIS**
   - Click **Build solution** to compile the project
   - Click **Load & Run application** to program the target chip
   - Click **Load & Debug application** to enter debug mode

5.3 Run

1. Connect the development board's USB port to Linux, and perform the following steps in order:

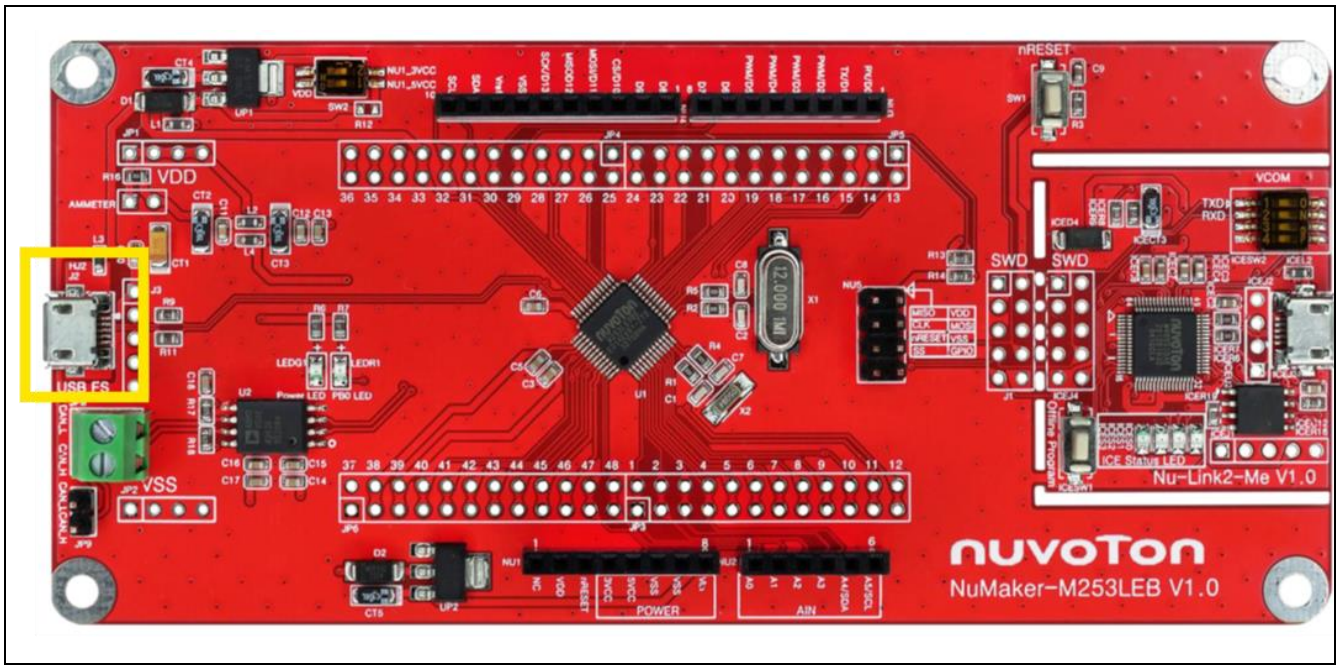


Figure 5-1 USB port location on the NuMaker-M253LE development board

2. List the currently connected CAN devices(ip link list type can).
3. Stop the device(ip link set <device> down).
4. Configure the device's transmission speed(ip link set <device> type can bitrate 500000).
5. Start the device (ip link set <device> up).
6. When the device is started, it can transmit CAN signals.(cansend <device> <can_id>#{data}), the device can also monitor the traffic on the bus (candump <device>).

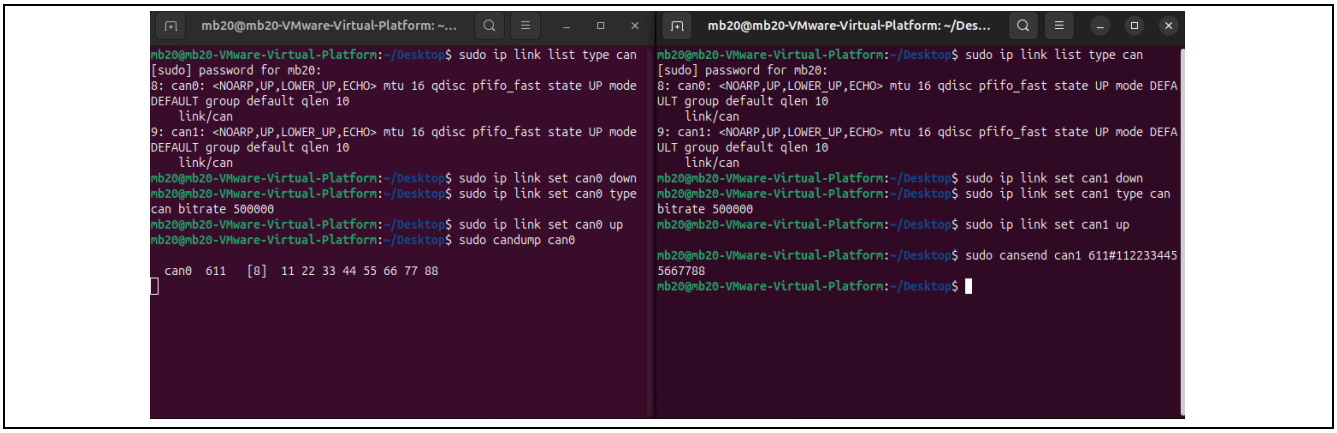


Figure 5-2 Two M253 devices communicate with each other through CAN in Linux environment

6. Revision History

Date	Revision	Description
2026.06.18	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.