M253 透过 USB 传送与接收 CAN 协议

NuMicro® 32 位系列微控制器范例代码介绍

文件信息

应用简述	本范例代码使用 M253 系列使用 USB 与 MCU 沟通，并传送与接收 CAN 协议。
BSP 版本	M253_Series_BSP_CMSIS_V3.00.010
开发平台	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. 概述

本范例代码展示如何透过 M253 系列的 USB 接口与主机 (Host) 沟通，藉此传送与接收 CAN 协议数据，并提供操作方式于 Linux 环境。在 M253 接收到 Host 的指令后，于 CAN 脚位产生讯号，并利用 Numaker-M253LE 上带的 CAN 收发器(ADM3050E)将 CAN 的逻辑讯号转换成 CANH/CANL 的差动讯号与其他 CAN 装置进行通讯。接收讯号同理，在收到其他装置的 CAN 讯号后，经收发器转换给 MCU 后，会回报 Host 收到的指令内容。

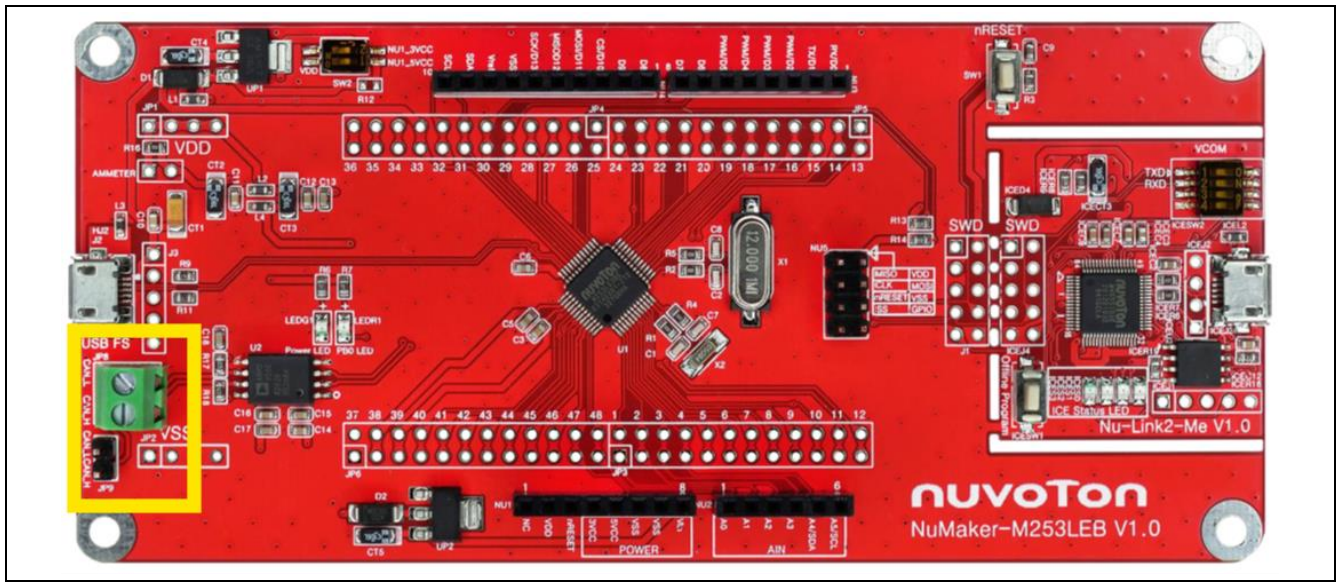


图 1-1 NuMaker-M253LE 开发板 CAN 脚位位置

1.1 原理

在 Linux 底下，原生支援 SocketCAN。它由 Linux 核心维护，会将 CAN 总线视为一个网络设备，使应用程序可以用标准的 BSD(Berkeley Software Distribution) Socket API 进行通讯，不需要额外撰写底层驱动。

Linux 中有现成的 CAN 工具，本范例使用 can-utils 中的指令来对 CAN 进行操作，使用前需要先安装 can-utils(安装指令: sudo apt-get install can-utils)。

会使用到的指令有:

candump: 监测并显示当前在 CAN 总线上的所有讯息。

cansend: 传送一笔 CAN frame，在此应用中，锁定上限为 8 bytes。

除了通讯指令以外，在使用前需要先和设备进行连接并设定通讯速度，会使用到的指令如下:

ip link set <device> down: 更改当前装置状态为 down。此指令为非必要指令，在设定 can 装置的 bitrate(下行指令)时装置必须要为 down 状态，因此用这条指令把装置状态设为 down。

ip link set <device> type can bitrate 500000: 更改当前 CAN 装置的 bitrate。该指令无法在装置状态为 up 时使用。

ip link set <device> up: 更改当前装置状态为 up。在设定完 bitrate 后启动。

上述指令会有权限不足问题，建议加上 sudo。而<device> 会根据装置的名称而有所不同(can0, can1…)，可以透过以下指令确认:

ip link list type can: 列出当前有连接在 Host 的 CAN 装置。

图 1-2 使用两个 M253 开发板，一个为装置 can0，另一个为装置 can1。

由 can1 传送指令给 can0，ID:0x611，Packet:0x1122334455667788。

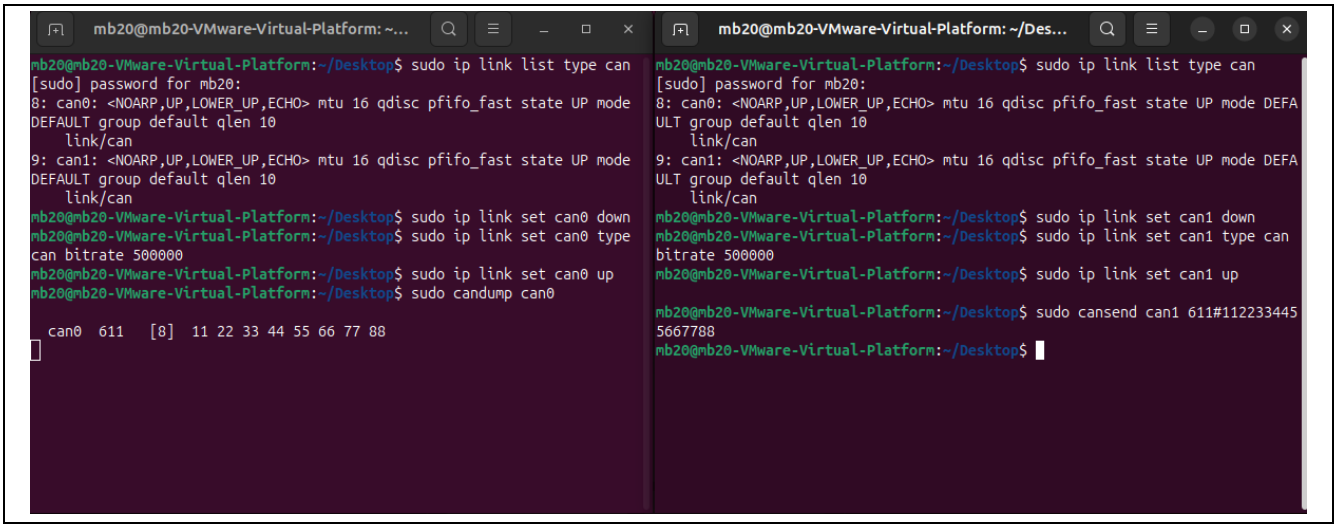


图 1-2 Linux 环境下用两个 M253 装置以 CAN 做通讯

1.2 执行结果

本范例代码执行编译和烧录的结果如图 1-3 和图 1-4 所示。

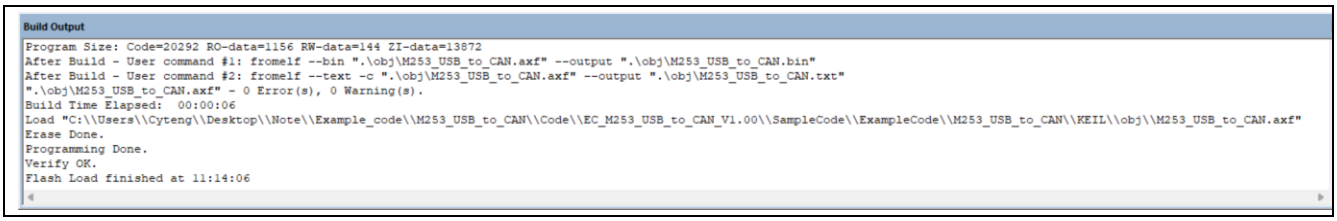


图 1-3 执行编译和烧录结果(Keil)



图 1-4 执行编译和烧录结果(VScode)

2. 代码介绍

系统初始化相关程序代码位于 *main.c*。在完成系统与 USB 初始化后，程序将进一步设定 CAN 模块以及 MCU 内部的 Timer 计算器制。

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

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

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

开启 USB 功能，并且设定回传给 Host 的数据，最后开启 USB 中断功能。

```
    /* 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);
```

接续前述初始化流程，程序会进行 CAN 相关指令与 MCU 内部 Timer 计算器制的初始化。

```
    pcan_timestamp_init();
```

函式 pcan_timestamp_init 内容，代码位于 *pcanpro_timestamp.c*。此函式透过启用 Timer0 进行系统时间计数，并以 1 毫秒为周期触发 Timer0 中断，更新系统时间相关变量，作为后续通讯时间计算。

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

中断程序 TMR0_IRQHandler，代码位于 *pcanpro_timestamp.c*。每次进入中断程序后会递增 g_au32TMRINTCount，该变量作为时间基准，提供通讯 timeout 判断。

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

```
        TIMER_ClearIntFlag(TIMER0);

        g_au32TMRINTCount++;
    }
}
```

主程序内容，此处为让系统的 Clock 可以利用 USB 校正，藉此让误差缩小。在 USB 的相关 Sample code 中都会有此段程序代码。详细说明可以阅读 M253 技术参考手册 M253 Series - Nuvoton 第 6.2.8 IRC Auto Trim 章节。

```
        /* 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);
        }
```

接续上面，g_u8Suspend 变数会在 USB 中断内被更动，在收到 Suspend 时会进入 Powerdown，其余时间会进入监测 Host 的 CAN 讯息。

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

函式 pcan_protocol_poll 内容，代码位于 pcanpro_fd_protocol.c。为主要轮询函式，负责 USB 与 CAN 数据的接收、封装与回传处理。

主要处理流程包含：

- 撷取当前系统时间-毫秒与微秒

  两个跟时间有关的函数 pcan_timestamp_millis 和 pcan_timestamp_us，皆是回传 Timer0 中断程序内更新的 g_au32TMRINTCount 变量以纪录 MCU 当前的计数值，两者仅回传的单位不相同。

- 处理 CAN 接收与传送数据

  函式透过变量 g_u8RxFifO0CompleteFlag 判断是否已触发 CAN 接收中断。CAN 中断会发生于收到 CAN 的数据以及 Time out 的时候。收到数据的时候会在 write_manual_rx_frame 函式内将 CAN 收到的数据存放到结构体中等待呼叫。pcan_can_poll 则判断是否收到要传送 CAN 数据的指令。若有收到送指令，用 USB 向 Host 发出回传数据。

- 将接收到的 CAN 数据透过 USB Bulk IN 回传至 Host

  flush data 部分为处理 CAN 收到数据后向 Host 透过 USB Bulk in 封包报告给 Host，透过 pcan_flush_data 函式作处理(仅有 Endpoint 2 的 Bulk in 会被处理，其他 Endpoint 会回传错误)

- 周期性更新 CAN bus load 与 time sync 信息

  每 250ms 更新内部的参数，计算传输数据的时间占总总线的时间，并配置两块空间分别存放计算结果(指标 pbs)以及状态参数(指标 pts, 参数举例:上次 sync 时间)。在进入更新前，会先确认装置是否已收到 Vendor Request 指令后继续往下执行。

```
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;
    }

}
```

函式 write_manual_rx_frame 內容，先检查是否有经过中断程序，有经过中断程序代表有 CAN 数据接收，g_u8RxFifO0CompleteFlag 才会被设为 1。该函数用来更新结构体 manual_msg 中的各个参数，结构体 g_sRxMsgFrame 在 CAN 中断里更新，并在此函数内将 g_sRxMsgFrame 放到 manual_msg 中。最后更新完 manual_msg 后呼叫 pcan_protocol_rx_frame 进行接收。

```
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);
```

函式 pcan_protocol_rx_frame 內容，先将 CAN 协议的 DLC 码根据接收结果做判断，将 CAN 收到的数据存到 Buffer 中，并计算 nominal, data bit time，最后把数据搬到配置的空间(指针 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;
}
```

函式 pcan_can_poll 內容。除了计数以外，呼叫 pcan_can_flush_tx 函式。

```
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);
```

```
}
```

函式 pcan_can_flush_tx 内容，将 TXFIFO 透过_can_send 发送。结尾更新 TXFIFO 的位置(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);
}
```

函式_can_send 内容，设定传输的细节，最后交给 Library 中 canfd.c 的函数 CANFD_TransmitTxMsg 来将数据送出。在 M253 BSP 中有包含 CAN 的 Sample code 在 CAN 传输前都要包含这个设定。

```
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 中断的呼叫程序，在装置接上时，会宣告此装置的 Endpoint 的特性，此程序会宣告两个接收 Host 装置的 Endpoint 和发送给 Host 装置的 Endpoint。在装置收到 Endpoint 上由 Host 发送的 Bulk 指令时，会进入 USB 的中断程序。指令内容包含初始化设定装置以及对装置发送 CAN 向外传送数据内容，因此分为两个 Endpoint 中断。

函式 EP3_Handler 内容，由中断程序呼叫，在对 Endpoint1 传送 Bulk out 讯号时会导向此函式。此函式目的在接收来自 Host 的设定指令。

```
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);
}
```

函式 EP5_Handler 内容，由中断程序呼叫，在对 Endpoint2 传送 Bulk out 讯号时会导向此函式，此函式目的在接收来自 Host 的发送 CAN 讯号，由于对于传讯号是以 512 bytes 去传送的，所以会进到 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;
    }
}
```

函式 pcan_protocol_process_data 内容，根据是由 EP3_Handler 还是 EP5_Handler 呼叫此函数而不同。若 ep 为 1，表示收到的数据为控制指令，进入 pcan_protocol_process_cmd。ep 为 2 时，进入 pcan_protocol_tx_frame，更新 pcanpro_data 数据。

```
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;
            }
        }
    }
```

函式 pcan_protocol_process_cmd，包含各种连接时会用到的指令。

函式 pcan_protocol_tx_frame 内容，解析 Bulk out 中的数据内容，并用 pcan_can_write 放到 can_dev_array[bus].tx_fifo 中。

```
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;
}
```

函式 pcan_can_write 內容。

```
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. 软件与硬件需求

3.1 软件需求

- BSP 版本

- M253_Series_BSP_CMSIS_V3.00.010

- IDE 版本

- Keil uVision 5.42
- Visual Studio Code 1.105.0

3.2 硬件需求

- 电路组件

- NuMaker-M253LE3AE-V1.1

- 引脚连接

- 将 USB 脚位接上 PC 上，并将开发版上的 CAN 脚位接到其他 CAN 装置上。

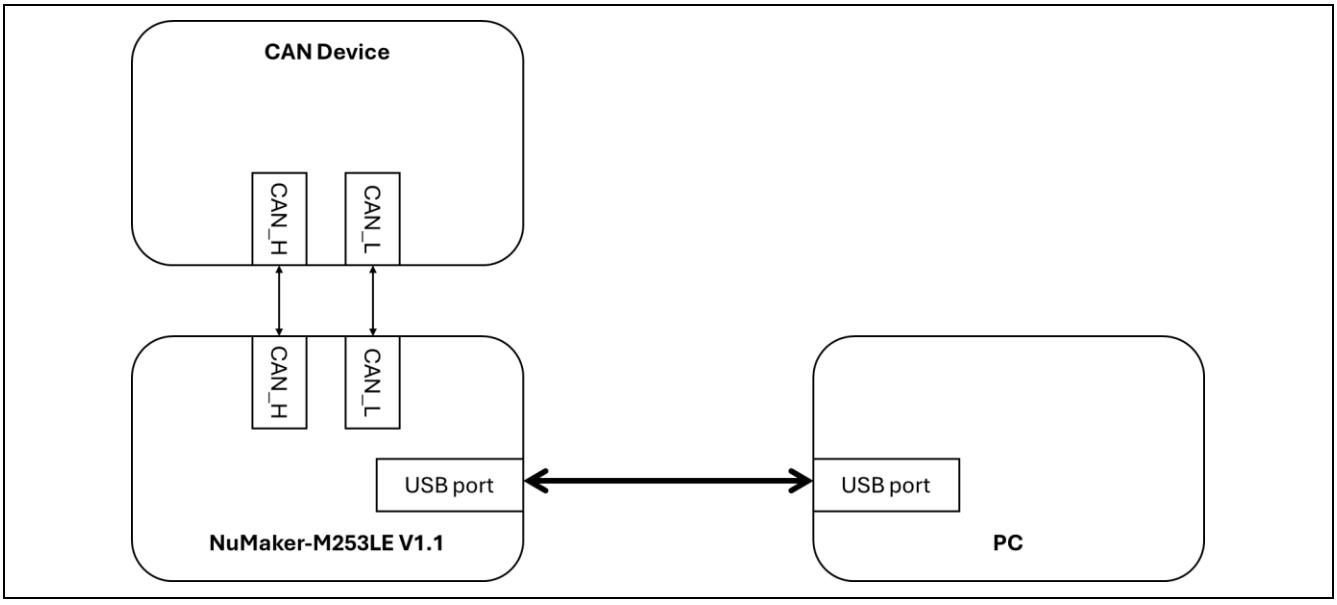


图 3-1 引脚连接

4. 目录信息

目录结构如下图所示。

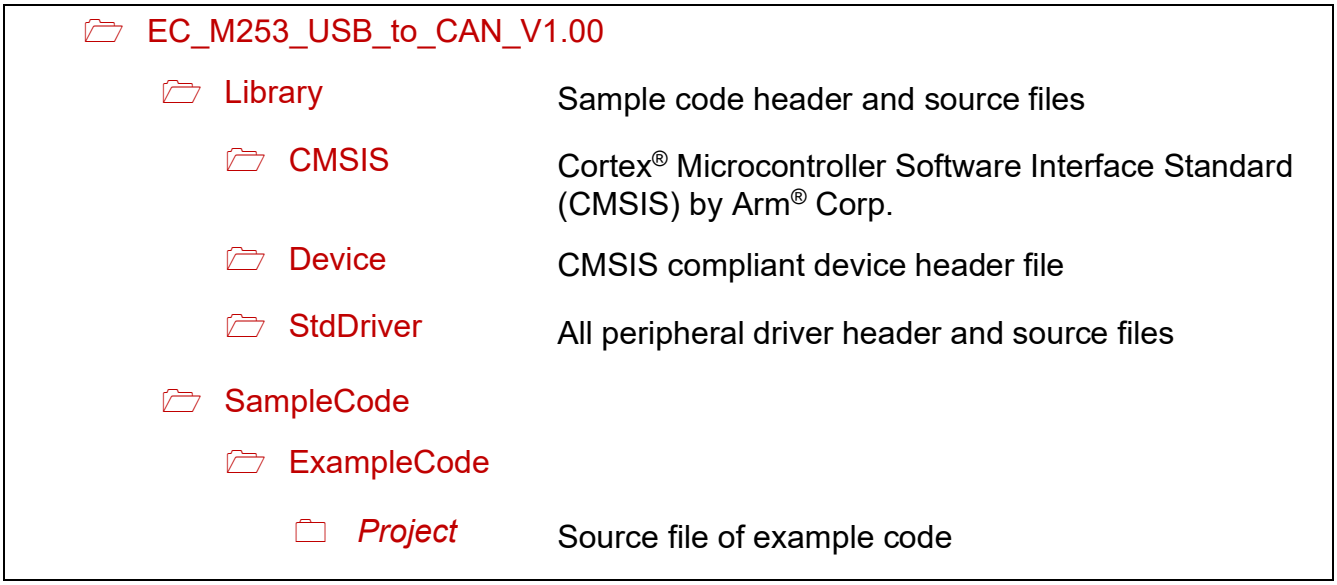


图 4-1 目录信息

5. 范例程序执行

5.1 Keil MDK

1. 在 Windows 档案总管中进入 **Keil** 工程文件夹，再开启 *M253_USB_to_CAN.uvprojx* 工程文件。
2. 编译代码
3. 下载代码
4. 进入仿真模式
    - 执行代码

5.2 VS Code

1. 在 **VS Code** IDE 中开启 **File** 选单
    - 单击 **Open Folder**
    - 选取工程目录下的 **VSCode** 文件夹
2. 选择 **CMSIS**
    - 单击 **Build solution** 编译工程
    - 单击 **Load & Run application** 下载映像文件到目标芯片
    - 单击 **Load & Debug application** 进入仿真模式

5.3 执行

1. 使用 Linux 连接开发版的 USB port，并按照以下顺序操作:

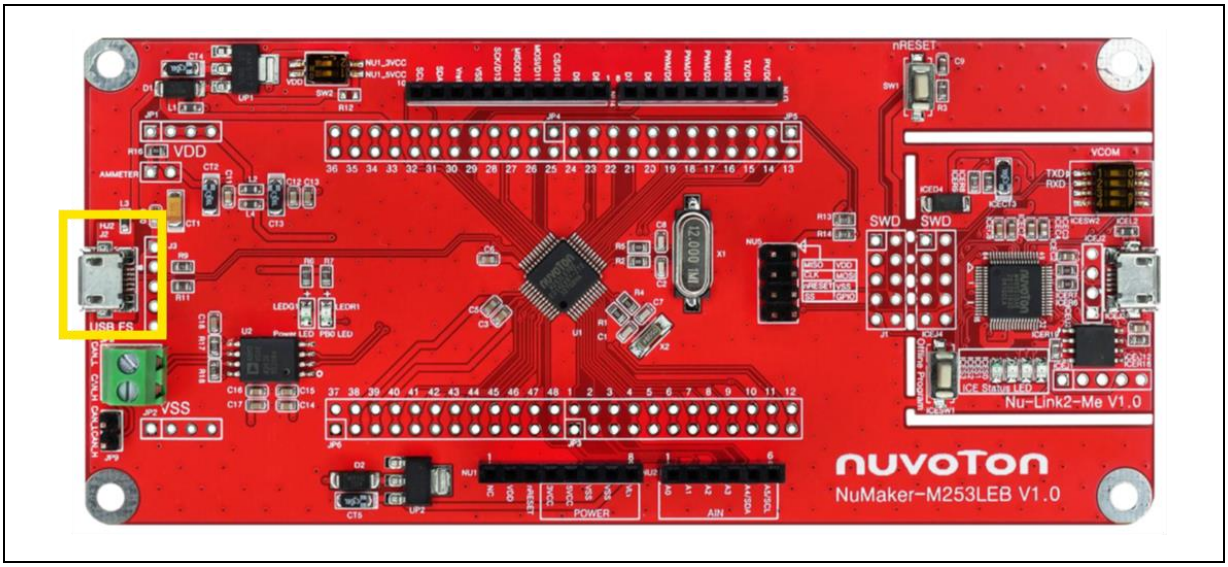


图 5-1 NuMaker-M253LE 开发板 USB port 位置

2. 列出当前连接的 CAN 装置(ip link list type can)

3. 停止装置(ip link set <device> down)

4. 设定装置的传输速度(ip link set <device> type can bitrate 500000)

5. 开启装置(ip link set <device> up)

6. 当装置开启时，可以让装置发送 CAN 讯号(cansend <device> <can_id>#{data})，也可以让装置监看讯号在线的传输内容(candump <device>)。

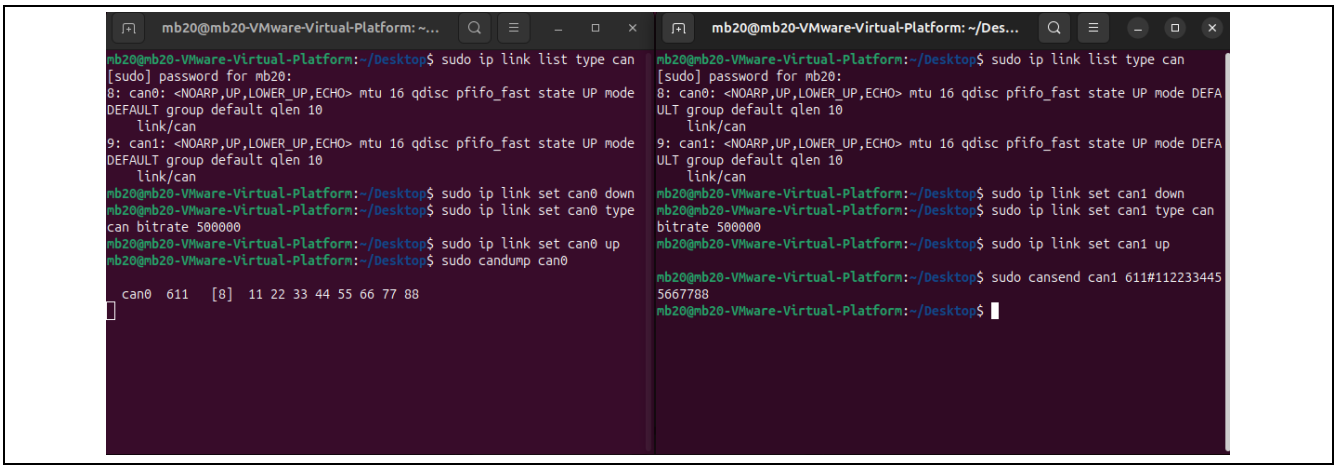


图 5-2 Linux 环境下用两个 M253 装置以 CAN 做通讯

6. 修订纪录

Date	Revision	Description
2026.06.18	1.00	初始发布。

Important Notice

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

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

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

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