M460 HSUSB Virtual Multiple Serial Port

Example Code Introduction for 32-bit NuMicro® Family

Document Information

Application	This example code describes how to use the M460 HSUSB virtual multiple serial port.
BSP version	M460_Series_BSP_CMSIS_V3.00.002
Hardware	NuMaker-M467HJ_V1.0

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 introduces the use of M460 high-speed USB to virtual multi-channel serial port (VCOM) for high-speed data acquisition of multi-channel serial port, and describes the differences between high-speed and full-speed USB, and how to identify the correspondence between device names and hardware.

- Similarities and differences between high-speed USB and full-speed USB (why to use HSUSB to virtual serial ports)
- The type of USB serial port device
- How to identify the correspondence between device names and hardware

1.1 Similarities and Differences Between High-speed USB and Full-speed USB

Both high-speed USB and full-speed USB fall under USB 2.0. The USB 2.0 technical specification was jointly developed and released by Compaq, Hewlett Packard, Intel, Lucent, Microsoft, NEC, and Philips, and the specification improved the peripheral data transmission speed. According to the original plan, the new USB 2.0 standard was only intended to set this standard at 240 Mbps, and later it was raised to 480 Mbps.

Due to the changes in the standards developed at the time, the USB specification resulted in three speed options: 480 Mbps, 12 Mbps, and 1.5 Mbps. In June 2003, when the USB 2.0 standard began to gain popularity, the USB Association renamed the USB specification and standard.

The USB standard was renamed, and the original USB 1.1 was changed to USB 2.0 Full Speed (FSUSB for short). At the same time, the original USB 2.0 was changed to USB 2.0 High-Speed (HSUSB for short), and a new logo was announced. The high-speed version of USB 2.0 is certainly faster than the full-speed version of USB 2.0. The differences between the three versions of the USB 2.0 protocol are as Figure 1-1:

Current name	Old name	Speed
USB 2.0 High Speed	USB 2.0	480Mbps
USB 2.0 Full Speed	USB 1.1	12Mbps
USB 2.0 Low Speed	USB 1.0	1.5Mbps

Figure 1-1 Different Speed of USB 2.0 Versions

In the USB specification, there is one of the most basic devices, called endpoints, which are a uniquely addressable part of a USB device, which is the source or reception point of information in the communication flow between the host and the device.

- **Control endpoint**

  The control endpoint or endpoint 0 is used to shake hands with the device, and it consists of endpoint IN and endpoint OUT. The USB specification requires that every USB device must support a control endpoint because it is a necessary component for device enumeration and basic communication.

  M467 HSUSB has an independent CEP endpoint for basic communication, while FSUSB needs use endpoint 0 and endpoint 1 to implement transmission and reception.

- **Number of endpoints**

  With the exception of endpoint 0, the number of other endpoints is based on their design requirements, and different controllers have different number of endpoints.

- **Endpoint direction**

  Data transmission needs bidirectional, in this case, each endpoint can only be configured as a single receive or single send.

- **Endpoint verifies**

  The endpoint uses CRC to detect errors in transmission. The CRC is verified by USB hardware, without the need to pay attention during use.

  The USB system defines four types of endpoint transmissions:

- **Control Transfer**

  Mainly used for enumerate devices when they are connected and other device-specific operations.

- **Interrupt Transfer**

  Used for reliable transmission of small amounts of data with strict latency requirements.

- **Bulk Transfer**

  Used for reliable transmission of large amounts of data with loose latency requirements.

- **Synchronous Transfer**

  Used for real-time data transmission where reliability is not required.

The serial port device in this example uses three of the above types of transmission:

- **Control Transfer:** To set up the USB and serial port configurations, a separate bidirectional USB control endpoint is used.

- **Interrupt Transfer:** Each VCOM uses one interrupt, which occupies one USB endpoint, to interrupt the signal of serial port packets.
- **Bulk Transfer:** Each VCOM uses two batch transfer, occupy two USB endpoints. Each endpoint respectively for receive and send the data from serial port.

M460 EndPoint	EndPoint number	SRAM Size	DMA	EndPoint Maximum Packet	Control EndPoint
HSUSB	12	4kbytes	✓	1024bytes	independent
FSUSB	25	1.5kbyte	×	64bytes	share

Figure 1-2 Difference Between M460 High-speed and Low-speed USB

Figure 1-2 shows the hardware differences between the two types of the M460 USB, from that the maximum packet size of HSUSB can send more data one time. Refer to Figure 1-1 Different Speed of USB 2.0 Versions The HSUSB can be used to respond to data sent by VCOM more quickly. HSUSB has only 12 endpoints and independent transceiver control endpoint, and it can only be virtual out 4 VCOMs, while FSUSB can be virtual out up to 7 VCOMs.

VCOM endpoint assignment for HSUSB (independent control endpoint is bidirectional transmission endpoint for transmitting and receiving):

Number of VCOMs = 12 ÷ (1 interrupt transfer + 1 receive bulk transfer + 1 transmit bulk transfer)

= 12 ÷ 3

= 4

VCOM endpoint assignment for FSUSB (one-way endpoint):

Number of VCOMs = (25-2 control endpoints (1 receive + 1 transmit)) ÷ (1 interrupt transfer + 1 receive bulk transfer + 1 transmit bulk transfer)

= (25-2) ÷ 3

= 7 (maximum divisor is 7)

There is another important concept in the USB control transfer protocol: descriptors.

USB descriptors contain a variety of information about the device that describes the functionality, features, and capabilities of the device to the host. USB descriptors include device descriptors, config descriptors, interface descriptors, endpoint descriptors, string descriptors, and among others.

- Device descriptor contains basic information about the device, such as the device type, vendor ID, product ID, and device version number.
- String descriptors provide some vendor and product character descriptions.

- Configure descriptors describe the configuration information of the device, including the number of configurations, the properties of each configuration, and so on.
- Interface descriptors describes the interface information of a device, including the number of interfaces, the type of each interface, and the protocol.
- Endpoint descriptor describes the endpoint information of the device, including the address, direction, type, maximum packet size of the endpoint, and so on.

The USB host communicates and interacts with the device by reading these descriptors to understand the capabilities and features of the device and load the appropriate drivers based on this information.

Descriptors have some specific relationships in USB interactions, and different descriptions have different functions. Figure 1-3 shows some of the descriptor hierarchies used in this example.

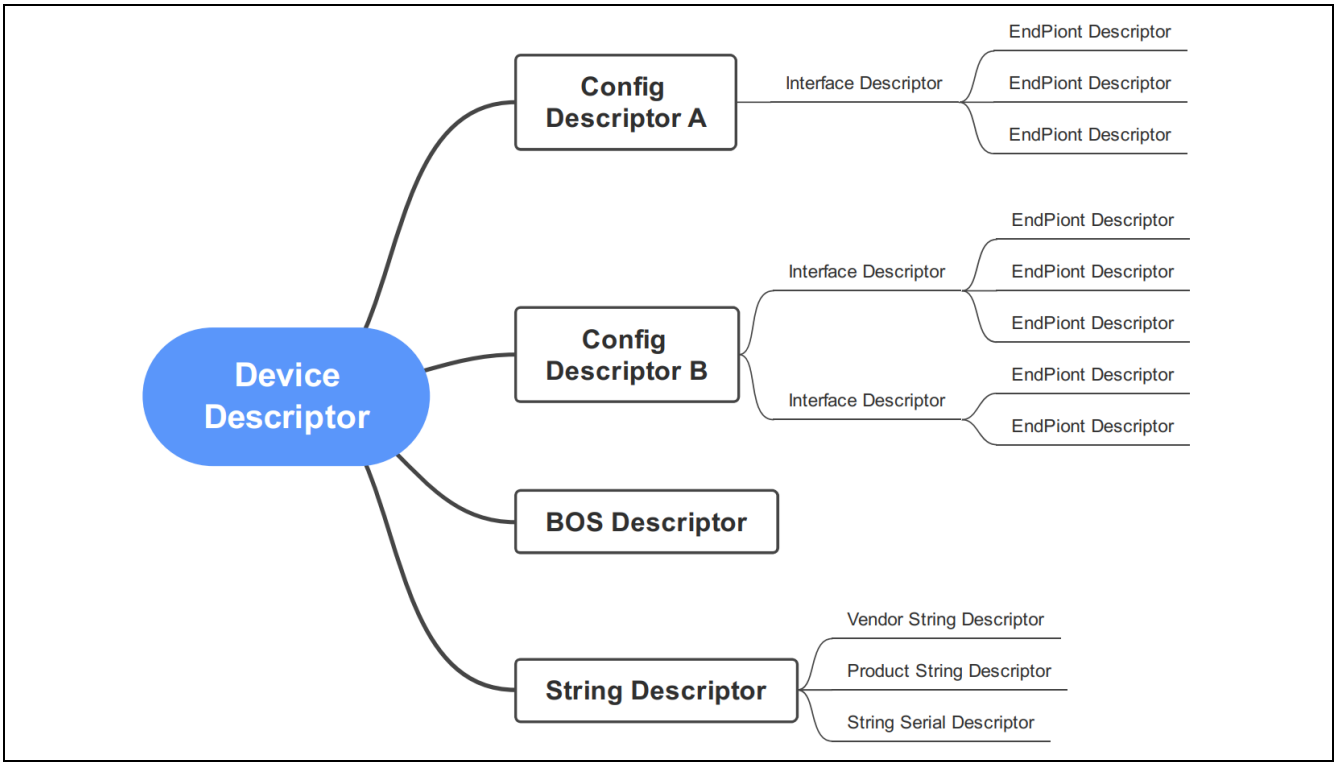

Figure 1-3 USB Descriptor Hierarchy

There are also some differences descriptors between the HSUSB and FSUSB, and the types of descriptors in FSUSB are a little less than those in HSUSB, mainly differences caused by the interface baud rates. Except for the difference in the maximum packet size of the endpoint, the configuration descriptor for other baud rates is the same as the content of the main configuration descriptor. Figure 1-4 is a comparison of the descriptor types of M460.

Device Descriptor	Config Descriptor	String Descriptor	BOS Descriptor	HID Report Descriptor	HID Report Descriptor Size	HID Descriptor Start Index	Qualifier Descriptor	Full Speed Config Descriptor	HSUSB Other Speed Config Descriptor	FSUSB Other Speed Config Descriptor
FSUSB Descriptor										
HSUSB Descriptor										

Figure 1-4 HSUSB and FSUSB Descriptor Differences

1.2 The Type of USB Serial Port Device

USB to VCP serial port

- It supports high-speed communication and hardware flow control.
- It is fully functional, and supports GPIO and other expansion functions.
- Emulates a standard serial port for upgrading the original serial port peripherals, or adding additional serial ports via USB.
- Computer interface applications are fully compatible and require no modifications.
- The driver only needs to be installed once, and it can also be installed automatically on the Internet.

USB to CDC serial port

- Windows10 and OS versions above have built-in CDC serial port drivers, but older versions need to be installed.
- Software and hardware flow control is not supported.
- It is not recommended to continuously transmit data at high baud rates, and it is recommended to increase the delay between transmissions.
- It is recommended to close the serial port before the system sleeps; otherwise, data will lose after wake up.

USB to HID serial port

- Use the USB HID device driver that comes with the operating system; the USB HID serial port is recognized as a standard HID device on the system.
- Built-in HID drivers for the operating system are supported.
- The system does not generate serial port nodes, so there is no need for manual selection, and a certain extent of information hiding is also realized.
- Data transmission is using USB interrupt endpoint.
- You need to use the native HID API of the system or the dynamic library provided by a third party to operate the serial port.

In this example, the CDC serial port is used, which can be used without a driver on common windows system, which greatly facilitates the user's practicability, although the CDC serial port does not support flow control, but the real-time response of data can also be greatly improved through independent interrupt endpoints.

1.3 How to Find the Correspondence between Multiple Serial Ports

Take a look at the default driver in Figure 1-5 that is virtualized on the computer after the firmware runs:

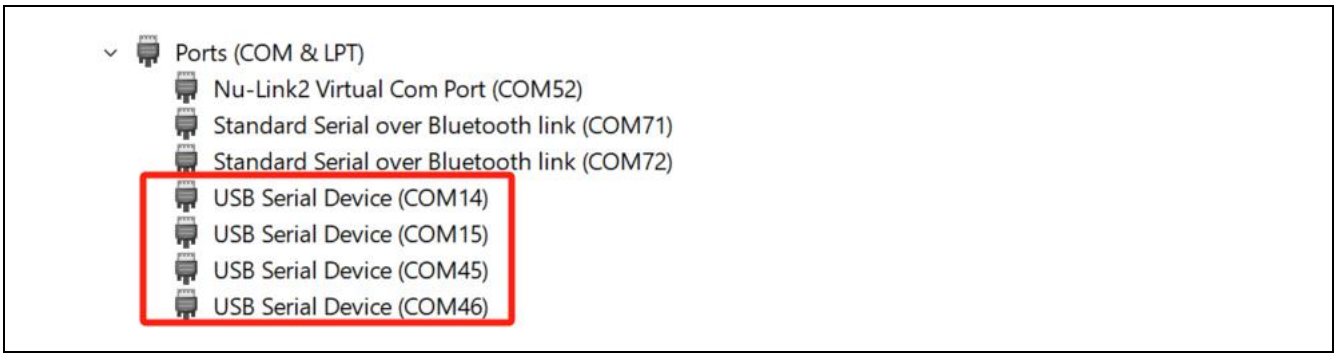

Figure 1-5 Default Driver VCOM Name

After connected to the device, the system will assign the original serial port number or the new serial port number of the device according to the serial port number (COM number) of the registered device, but the M467 firmware is set according to the interface number (Interface number) and physical serial port (UART number) of the USB device. This configuration method leads to the fact that the string slogan (the system is assigned randomly) does not correspond well to the physical serial port in the actual use process.

The fixed correspondence between the USB interface number and the UART number in the firmware is as follows:

- Interface0 <==> UART0
- Interface2 <==> UART2
- Interface4 <==> UART3
- Interface6 <==> UART4

The COM number on the computer is random, and the device name "USB Serial Device" is the same, so it is impossible to quickly select the physical UART port that needs to be operated, and you can only view the interface number of the corresponding USB device from the device properties.

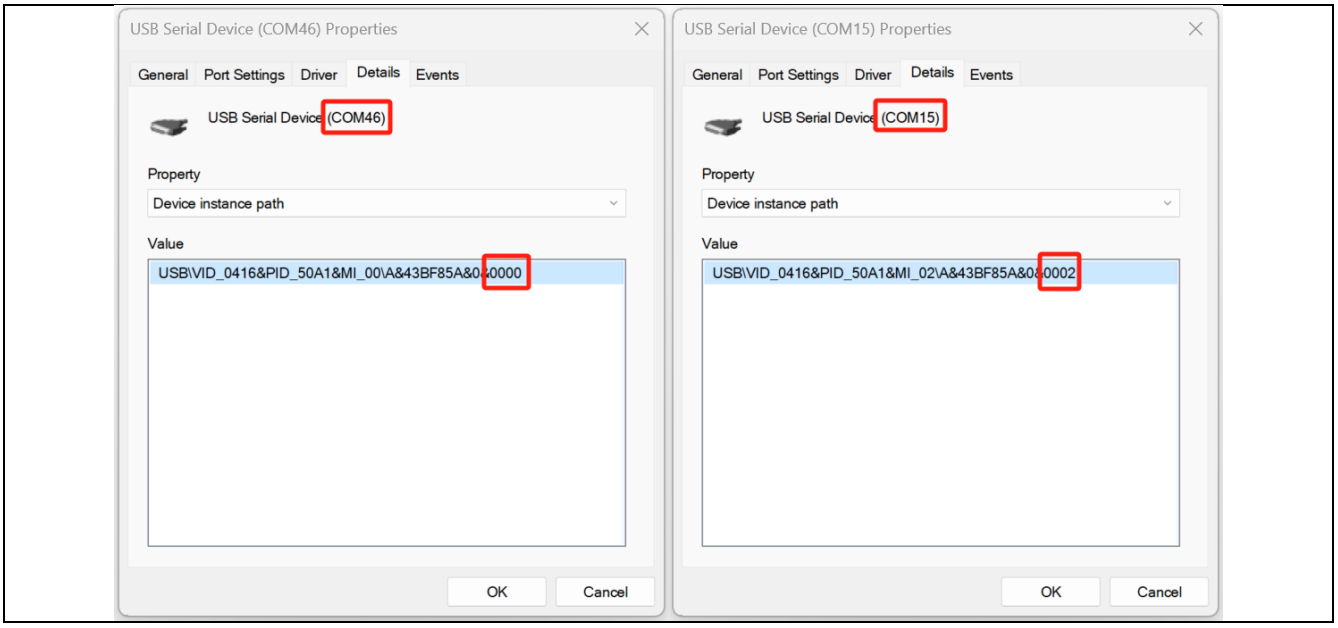

Figure 1-6 Interfaces for Different COMs

As shown in Figure 1-6, the numbers in the red box show the interface number, then the COM number and the interface number relations are as follows:

- COM46 <==> Interface0
- COM15 <==> Interface2

To solve above problems, adjust the USB driver using different device names (VCOM numbers) to distinguish different interface numbers, so that users can quickly select the physical UART port need to be operated.

The following steps can be used to install the tuned drivers:

1. Install the Nu-Link USB driver from the Download Link:

   https://www.nuvoton.com.cn/export/resource-files/en-us--Nu-Link_USB_Driver_V1.11.zip

   Replace the original driver files *nuvcom.cat* and *nuvcom.inf*.

2. The default installation path is C:\Program Files\Nuvoton Tools\Nu-Link_USB_Driver.

3. Update the drivers.

   Select the device - > Right-click Update Driver -> Browse My Computer for Drivers -> Let me select -> from the list of available drivers available on my PC, Figure 1-7 Select the Corresponding Driver by selecting the appropriate VCOMx driver -> Next -> Done.

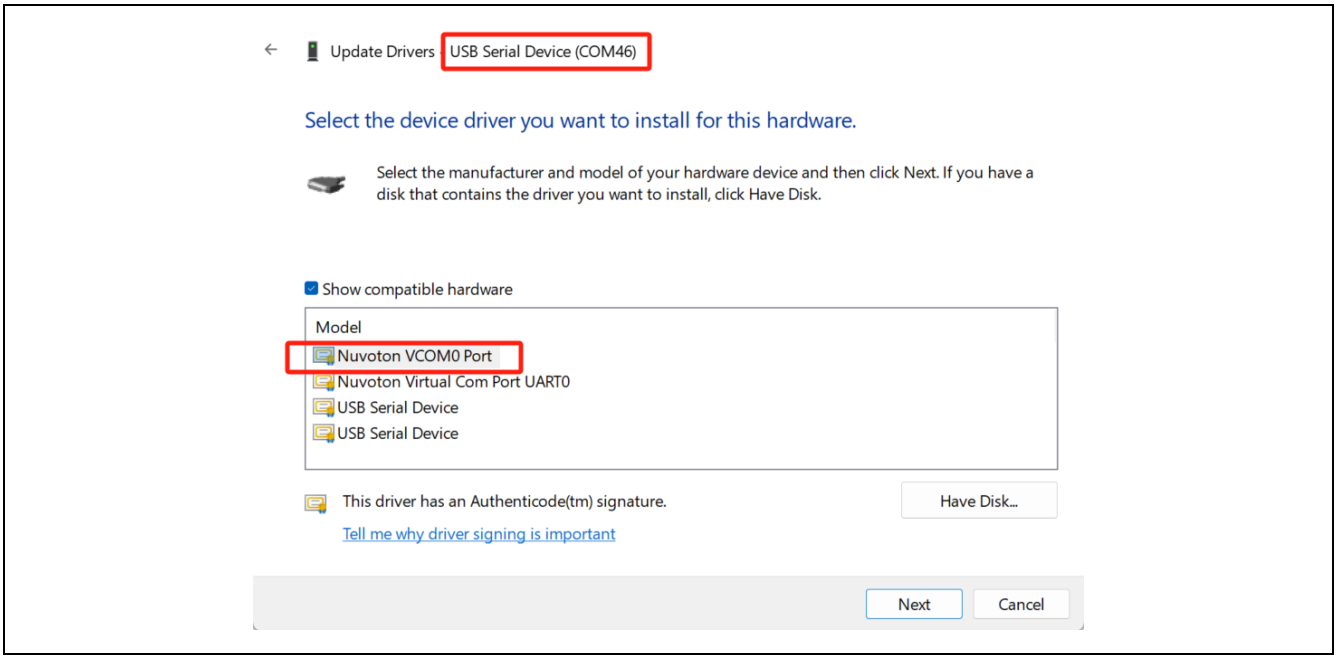

Figure 1-7 Select the Corresponding Driver

As shown in Figure 1-8, the 4 VCOM is displayed after the driver is updated:

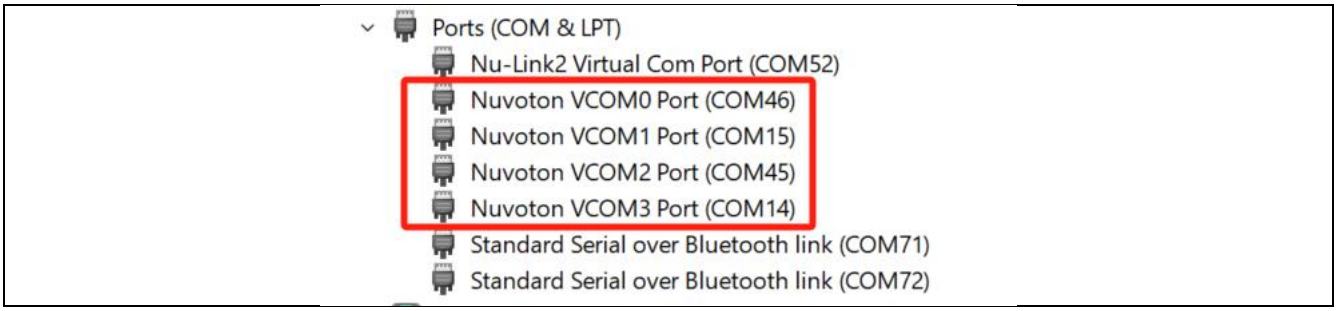

Figure 1-8 Name of the Modified Device

From Figure 1-8 Name of the Modified Device combined with the default interface and UART bind by the firmware, the physical UART port can be easily found from the COM port:

- Interface0 <==> VCOM0 <==> UART0 <==> COM46
- Interface2 <==> VCOM1 <==> UART2 <==> COM15
- Interface4 <==> VCOM2 <==> UART3 <==> COM45
- Interface6 <==> VCOM3 <==> UART4 <==> COM14

The interface with UART is bind by the firmware. Please refer to instructions in Figure 1-5.

2. Code Description

There are many USB-related codes, and some customizable content commonly used by users is selected to introduce, including the following points:

- USB parameter macro definition
- Device descriptor
- The serial port configuration
- Data sending and receiving
- Test case

2.1 USB Parameter Macro Definition

The definition of the USB parameters is located in the *vcom_serial.h* file.
Define the number of VCOM_CNT :1~4. If the range is exceeded, the compilation error will be reported.

```
#define VCOM_CNT 4

#if ((VCOM_CNT>=5) || (VCOM_CNT<=0))
#error VCOM number error, VCOM_CNT must be 1 ~ 4
#endif......
```

USB PID and VID need to be able to make the driver recognize that it is a VCOM device, so it cannot be changed; otherwise, the above device name correspondence will not be realized.

```
//------------------device descriptor define---------------------//
/* Define the vendor id and product id */
#define USBD_VID        0x0416
#define USBD_PID        0x50A1
```

The size of the RAM allocated to each endpoint is 4 Kbytes, and the amount of RAM allocated to each VCOM is 256 Bytes.
The control endpoint CEP is generally defined as 64 Bytes.

```
//------------------ram buffer define---------------------//
/* Define DMA Maximum Transfer length */
#define USBD_MAX_DMA_LEN    0x1000
#define EP_MAX_DATA_PKT_SIZE   256

/* Define EP maximum packet size */
/*4096 Bytes Configurable able RAM used as endpoint buffer*/
#define CEP_MAX_PKT_SIZE        64
#define CEP_OTHER_MAX_PKT_SIZE  64
```

2.2 Device Descriptor

Each USB device has its own device description, and in this case, each VCOM device has the same description type. The difference is that the interface number, maximum packet, endpoint number, and endpoint type of the respective device are defined. The description of the device is in the *descriptors.c* file. Only the device description for VCOM0 is shown here.

```
//-----------------VCOM0------------------//
#if (VCOM_CNT >= 1)
    THEM
    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 */

    /* 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   */
    INTF_NUM_VCOM0,                   /* bMasterInterface     */
    INTF_NUM_VCOM0 + 1,               /* bSlaveInterface0 */

    /* ENDPOINT descriptor */
    LEN_ENDPOINT,                     /* bLength          */
    DESC_ENDPOINT,                    /* bDescriptorType */
```

```
    (EP_INPUT | INT_IN_EP_NUM_0),       /* bEndpointAddress */
    EP_INT,                             /* bmAttributes     */
    GET_BYTE0(EPC_OTHER_MAX_PKT_SIZE),
    GET_BYTE1(EPC_OTHER_MAX_PKT_SIZE),  /* wMaxPacketSize   */
    0x01,                               /*bInterval */

    /* INTERFACE descriptor */
    LEN_INTERFACE,      /* bLength              */
    DESC_INTERFACE,     /* bDescriptorType      */
    INTF_NUM_VCOM0 + 1, /* bInterfaceNumber     */
    0x00,               /* bAlternateSetting    */
    0x02,               /* bNumEndpoints        */
    0x0A,               /* bInterfaceClass      */
    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_OTHER_MAX_PKT_SIZE),
    GET_BYTE1(EPA_OTHER_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_OTHER_MAX_PKT_SIZE),
    GET_BYTE1(EPB_OTHER_MAX_PKT_SIZE),  /* wMaxPacketSize   */
    0x00,                               /*bInterval */
#endif
```

2.3 Serial Port Configuration

The serial port uses an array to store the serial port config parameters, with a default value of 115200 bsp, 1 stop bit, 8 bits of data, and no parity. This array is defined in a *vcom_serical.c* file, and when the PC opens the serial port, the USB receives the config parameters sent by the PC through the control endpoint and reconfigures the serial port.

```
/*-------------------------------------------------------------------------*/
/* Stop bit     */
/* parity       */
/* data bits    */
STR_VCOM_LINE_CODING gLineCoding[VCOM_CNT] =
#if (VCOM_CNT>=1)
{   {115200, 0, 0, 8},   /* Baud rate : 115200    */
#endif
#if (VCOM_CNT>=2)
    {115200, 0, 0, 8},   /* Baud rate : 115200    */
#endif
#if (VCOM_CNT>=3)
    {115200, 0, 0, 8},   /* Baud rate : 115200    */
```

```
#endif
#if (VCOM_CNT>=4)
    {115200, 0, 0, 8},   /* Baud rate : 115200    */
#endif
};
```

The function *VCOM_ClassRequest()* in the *vcom_serical.c* file is used to read and set the serial port settings data sent by USB.

```
/**
 *  @brief      USB Control Endpoint to ConFig UART port
 *  @param[in]  None
 *  @return     None
 *  @details    The function is used to:
                  Read UART ConFig information
                  Write UART ConFig information
*/
void VCOM_ClassRequest(void)
{   static uint8_t intface;
    if(gUsbCmd.bmRequestType & 0x80)    /* request data transfer direction */
    {
        // Device to host
        switch(gUsbCmd.bRequest)
        {
        case GET_LINE_CODE:
        {
            intface = gUsbCmd.wIndex & 0xff;
            HSUSBD_PrepareCtrlIn((uint8_t *)&gLineCoding[intface>>1], 7);

            HSUSBD_CLR_CEP_INT_FLAG(HSUSBD_CEPINTSTS_INTKIF_Msk);
            HSUSBD_ENABLE_CEP_INT(HSUSBD_CEPINTEN_INTKIEN_Msk);
            break;
        }
        default:
        {
            /* Setup error, stall the device */
            HSUSBD_SET_CEP_STATE(HSUSBD_CEPCTL_STALLEN_Msk);
            break;
        }
        }
    }
    else
    {
        // Host to device
        switch(gUsbCmd.bRequest)
        {
        case SET_CONTROL_LINE_STATE:
        {
            intface = gUsbCmd.wIndex & 0xff;
            gCtrlSignal[intface>>1] = gUsbCmd.wValue;

            // DATA IN for end of setup
            /* Status stage */
            HSUSBD_CLR_CEP_INT_FLAG(HSUSBD_CEPINTSTS_STSDONEIF_Msk);
            HSUSBD_SET_CEP_STATE(HSUSBD_CEPCTL_NAKCLR);
            HSUSBD_ENABLE_CEP_INT(HSUSBD_CEPINTEN_STSDONEIEN_Msk);
            break;
```

```
        }
        case SET_LINE_CODE:
        {
            intface = gUsbCmd.wIndex & 0xff;
            HSUSBD_CtrlOut((uint8_t *)&gLineCoding[intface>>1], 7);

            /* Status stage */
            HSUSBD_CLR_CEP_INT_FLAG(HSUSBD_CEPINTSTS_STSDONEIF_Msk);
            HSUSBD_SET_CEP_STATE(HSUSBD_CEPCTL_NAKCLR);
            HSUSBD_ENABLE_CEP_INT(HSUSBD_CEPINTEN_STSDONEIEN_Msk);

            /* UART setting */
            VCOM_LineCoding(intface>>1);
            break;
        }
        default:
        {
            /* Setup error, stall the device */
            HSUSBD_SET_CEP_STATE(HSUSBD_CEPCTL_STALLEN_Msk);
            break;
        }
        }
    }
}
```

After receiving the serial port configuration sent by USB, the hardware serial port set parameters by *VCOM_LineCoding ()* in the *uart_cfg.c* file.

```
/**
 * @brief USB VCOM set to become config
 *  @param[in]  UART COM Index:
 *              - \ref IDX_VCOM0
 *              - \ref IDX_VCOM1
 *              - \ref IDX_VCOM2
 *              - \ref IDX_VCOM3
 *  @return     None
 *  @details
*/
void VCOM_LineCoding(uint8_t port)
{
    uint32_t u32Reg;
    uint32_t u32Baud_Div;
    volatile struct vcom_var_t *vtp=&vcom_dat[port];

    switch(port)
    {
#if (VCOM_CNT>=1)
    case IDX_VCOM0 :
        NVIC_DisableIRQ(UART0_IRQn);
        break;
#endif
#if (VCOM_CNT>=2)
    case IDX_VCOM1 :
        NVIC_DisableIRQ(UART2_IRQn);
        break;
#endif
#if (VCOM_CNT>=3)
```

```
    case IDX_VCOM2 :
        NVIC_DisableIRQ(UART3_IRQn);
        break;
#endif
#if (VCOM_CNT>=4)
    case IDX_VCOM3 :
        NVIC_DisableIRQ(UART4_IRQn);
        break;
#endif
    }

    // Reset software fifo
    vtp->comRbytes = 0;
    vtp->comRhead = 0;
    vtp->comRtail = 0;

    vtp->comTbytes = 0;
    vtp->comThead = 0;
    vtp->comTtail = 0;

    // Reset hardware fifo
    uart_port[port]->FIFO = 0x3;

    // Set baudrate
    u32Baud_Div = UART_BAUD_MODE2_DIVIDER(__HIRC, gLineCoding[port].u32DTERate);

    if(u32Baud_Div > 0xFFFF)
        uart_port[port]->BAUD = (UART_BAUD_MODE0 | UART_BAUD_MODE0_DIVIDER(__HIRC,
gLineCoding[port].u32DTERate));
    else
        uart_port[port]->BAUD = (UART_BAUD_MODE2 | u32Baud_Div);

    // Set parity
    if(gLineCoding[port].u8ParityType == 0)
        u32Reg = 0;        // none parity
    else if(gLineCoding[port].u8ParityType == 1)
        u32Reg = 0x08;     ODD Parity
    else if(gLineCoding[port].u8ParityType == 2)
        u32Reg = 0x18;     // even parity
    else
        u32Reg = 0;

    // bit width
    switch(gLineCoding[port].u8DataBits)
    {
    case 5:
        u32Reg |= 0;
        break;
    case 6:
        u32Reg |= 1;
        break;
    case 7:
        u32Reg |= 2;
        break;
    case 8:
        u32Reg |= 3;
```

```
        break;
    default:
        break;
    }

    // stop bit
    if(gLineCoding[port].u8CharFormat > 0)
        u32Reg |= 0x4; // 2 or 1.5 bits

    uart_port[port]->LINE = u32Reg;

    // Re-enable UART interrupt
    switch(port)
    {
#if (VCOM_CNT>=1)
    case IDX_VCOM0 :
        NVIC_EnableIRQ(UART0_IRQn);
        break;
#endif
#if (VCOM_CNT>=2)
    case IDX_VCOM1 :
        NVIC_EnableIRQ(UART2_IRQn);
        break;
#endif
#if (VCOM_CNT>=3)
    case IDX_VCOM2 :
        NVIC_EnableIRQ(UART3_IRQn);
        break;
#endif
#if (VCOM_CNT>=4)
    case IDX_VCOM3 :
        NVIC_EnableIRQ(UART4_IRQn);
        break;
#endif
    }
}
```

2.4 Data Sending and Receiving

The transmission of data is duplex, and each interface has a corresponding sending and receiving processing mechanism.

When the USB send buffer is filled with data, and after the transmission is completed, the corresponding endpoint will produce an interrupt, and the send flag will be cleared. When there is data in the USB receive buffer, the corresponding endpoint will generate a receive interrupt, and the receive flag will be set to indicate that the data reception is complete.

```
//-----------------VCOM0-------------------//
void EPA_Handle(void) //Send complete endpoint
{   //data out mcu
    volatile uint32_t IrqSt = HSUSBD->EP[EPA]. EPINTSTS & HSUSBD->EP[EPA]. EPINTEN;
    volatile struct vcom_var_t *vtp=&vcom_dat[IDX_VCOM0];

    vtp->gu32TxSize = 0;
    HSUSBD_ENABLE_EP_INT(EPA, 0);
```

```
    HSUSBD_CLR_EP_INT_FLAG(EPA, IrqSt);
}

void EPB_Handle(void) //Receive complete endpoint
{   //data in mcu
    volatile uint32_t IrqSt = HSUSBD->EP[EPB]. EPINTSTS & HSUSBD->EP[EPB]. EPINTS;
    volatile struct vcom_var_t *vtp=&vcom_dat[IDX_VCOM0];
    int volatile i;

    vtp->gu32RxSize = HSUSBD->EP[EPB]. EPDATCNT & 0xffff;
    for(i = 0; i < vtp->gu32RxSize; i++)
        vtp->gUsbRxBuf[i] = HSUSBD->EP[EPB]. EPDAT_BYTE;

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

The UART receives data from the physical interface through interrupts and stores it in the buffer. The main function continuously detects whether there is data in the UART receive buffer, and if it detects data, it moves the data to the USB send buffer and triggers the corresponding USB endpoint to send data. In the same way, the main function constantly checks whether there is data in the USB receive buffer, and if there have data, it also triggers the UART to send the USB data. The function *VCOM_TransferData ()* in the *uart_cfg.c* file needs to be executed *loop in the main()* function.

```
/**
 *  @brief      UART & USB Receive Buffer Transmit
 *  @param[in]  None
 *  @return     None
 *  @details    Put it in main Loop to receive and transmit uart data
*/
void VCOM_TransferData(void)
{
    int32_t i, i32Len,cnt;
    volatile struct vcom_var_t *vtp;

    for(cnt=0; cnt<VCOM_CNT; cnt++)
    {   vtp=&vcom_dat[cnt];

        /* Check if any data to send to USB & USB is ready to send them out */
        if(vtp->comRbytes && (vtp->gu32TxSize == 0))
        {
            i32Len = vtp->comRbytes;
            if(i32Len > EP_MAX_DATA_PKT_SIZE)
                i32Len = EP_MAX_DATA_PKT_SIZE;

            for(i = 0; i < i32Len; i++)
            {
                vtp->gRxBuf[i] = vtp->comRbuf[vtp->comRhead++];
                if(vtp->comRhead >= RXBUFSIZE)
                    vtp->comRhead = 0;
            }

            switch(cnt)
```

```
            {
#if (VCOM_CNT>=1)
            case IDX_VCOM0 :
                NVIC_DisableIRQ(UART0_IRQn);
                vtp->comRbytes -= i32Len;
                NVIC_EnableIRQ(UART0_IRQn);
                break;
#endif
#if (VCOM_CNT>=2)
            case IDX_VCOM1 :
                NVIC_DisableIRQ(UART2_IRQn);
                vtp->comRbytes -= i32Len;
                NVIC_EnableIRQ(UART2_IRQn);
                break;
#endif
#if (VCOM_CNT>=3)
            case IDX_VCOM2 :
                NVIC_DisableIRQ(UART3_IRQn);
                vtp->comRbytes -= i32Len;
                NVIC_EnableIRQ(UART3_IRQn);
                break;
#endif
#if (VCOM_CNT>=4)
            case IDX_VCOM3 :
                NVIC_DisableIRQ(UART4_IRQn);
                vtp->comRbytes -= i32Len;
                NVIC_EnableIRQ(UART4_IRQn);
                break;
#endif
            }

            vtp->gu32TxSize = i32Len;

            switch(cnt)
            {
#if (VCOM_CNT>=1)
            case IDX_VCOM0 :
                for(i = 0; i < i32Len; i++)
                    HSUSBD->EP[EPA]. EPDAT_BYTE = vtp->gRxBuf[i];
                HSUSBD->EP[EPA]. EPRSPCTL = HSUSBD_EP_RSPCTL_SHORTTXEN;      // packet end
                HSUSBD->EP[EPA]. EPTXCNT = i32Len;
                HSUSBD_ENABLE_EP_INT(EPA, HSUSBD_EPINTEN_INTKIEN_Msk);
                break;
#endif
#if (VCOM_CNT>=2)
            case IDX_VCOM1 :
                for(i = 0; i < i32Len; i++)
                    HSUSBD->EP[EPD]. EPDAT_BYTE = vtp->gRxBuf[i];
                HSUSBD->EP[EPD]. EPRSPCTL = HSUSBD_EP_RSPCTL_SHORTTXEN;     // packet end
                HSUSBD->EP[EPD]. EPTXCNT = i32Len;
                HSUSBD_ENABLE_EP_INT(EPD, HSUSBD_EPINTEN_INTKIEN_Msk);
                break;
#endif
#if (VCOM_CNT>=3)
            case IDX_VCOM2 :
                for(i = 0; i < i32Len; i++)
```

```
                    HSUSBD->EP[EPG]. EPDAT_BYTE = vtp->gRxBuf[i];
                HSUSBD->EP[EPG]. EPRSPCTL = HSUSBD_EP_RSPCTL_SHORTTXEN;    // packet end
                HSUSBD->EP[EPG]. EPTXCNT = i32Len;
                HSUSBD_ENABLE_EP_INT(EPG, HSUSBD_EPINTEN_INTKIEN_Msk);
                break;
#endif
#if (VCOM_CNT>=4)
            case IDX_VCOM3 :
                for(i = 0; i < i32Len; i++)
                    HSUSBD->EP[EPJ]. EPDAT_BYTE = vtp->gRxBuf[i];
                HSUSBD->EP[EPJ]. EPRSPCTL = HSUSBD_EP_RSPCTL_SHORTTXEN;    // packet end
                HSUSBD->EP[EPJ]. EPTXCNT = i32Len;
                HSUSBD_ENABLE_EP_INT(EPJ, HSUSBD_EPINTEN_INTKIEN_Msk);
                break;
#endif
            }
        }

        /* Process the Bulk out data when bulk out data is ready. */
        if(vtp->gi8BulkOutReady && (vtp->gu32RxSize <= TXBUFSIZE - vtp->comTbytes))
        {
            for(i = 0; i < vtp->gu32RxSize; i++)
            {
                vtp->comTbuf[vtp->comTtail++] = vtp->gUsbRxBuf[i];
                if(vtp->comTtail >= TXBUFSIZE)
                    vtp->comTtail = 0;
            }

            switch(cnt)
            {
#if (VCOM_CNT>=1)
            case IDX_VCOM0 :
                NVIC_DisableIRQ(UART0_IRQn);
                vtp->comTbytes += vtp->gu32RxSize;
                NVIC_EnableIRQ(UART0_IRQn);
                break;
#endif
#if (VCOM_CNT>=2)
            case IDX_VCOM1 :
                NVIC_DisableIRQ(UART2_IRQn);
                vtp->comTbytes += vtp->gu32RxSize;
                NVIC_EnableIRQ(UART2_IRQn);
                break;
#endif
#if (VCOM_CNT>=3)
            case IDX_VCOM2 :
                NVIC_DisableIRQ(UART3_IRQn);
                vtp->comTbytes += vtp->gu32RxSize;
                NVIC_EnableIRQ(UART3_IRQn);
                break;
#endif
#if (VCOM_CNT>=4)
            case IDX_VCOM3 :
                NVIC_DisableIRQ(UART4_IRQn);
                vtp->comTbytes += vtp->gu32RxSize;
                NVIC_EnableIRQ(UART4_IRQn);
```

```
                break;
#endif
            }

            vtp->gu32RxSize = 0;
            vtp->gi8BulkOutReady = 0; /* Clear bulk out ready flag */
        }

        /* Process the software Tx FIFO */
        if(vtp->comTbytes)
        {
            /* Check if Tx is working */
            if((uart_port[cnt]->INTEN & UART_INTEN_THREIEN_Msk) == 0)
            {
                /* Send one bytes out */
                UART_WRITE(uart_port[cnt], vtp->comTbuf[vtp->comThead++]);
                if(vtp->comThead >= TXBUFSIZE)
                    vtp->comThead = 0;

                vtp->comTbytes--;

                /* Enable Tx Empty Interrupt. (Trigger first one) */
                uart_port[cnt]->INTEN |= UART_INTEN_THREIEN_Msk;
            }
        }
    }
}
}
```

2.5 Test Case

Test the data transmission between 2 sets of different port configurations, and see the results:

1. Connect the transceiver pins of UART0 and UART2 with wires.
2. The port config is 2 Mbps, 8-bit data, no parity, and 1-bit stop bit.
3. VCOM0 and VCOM1 send data to each other.

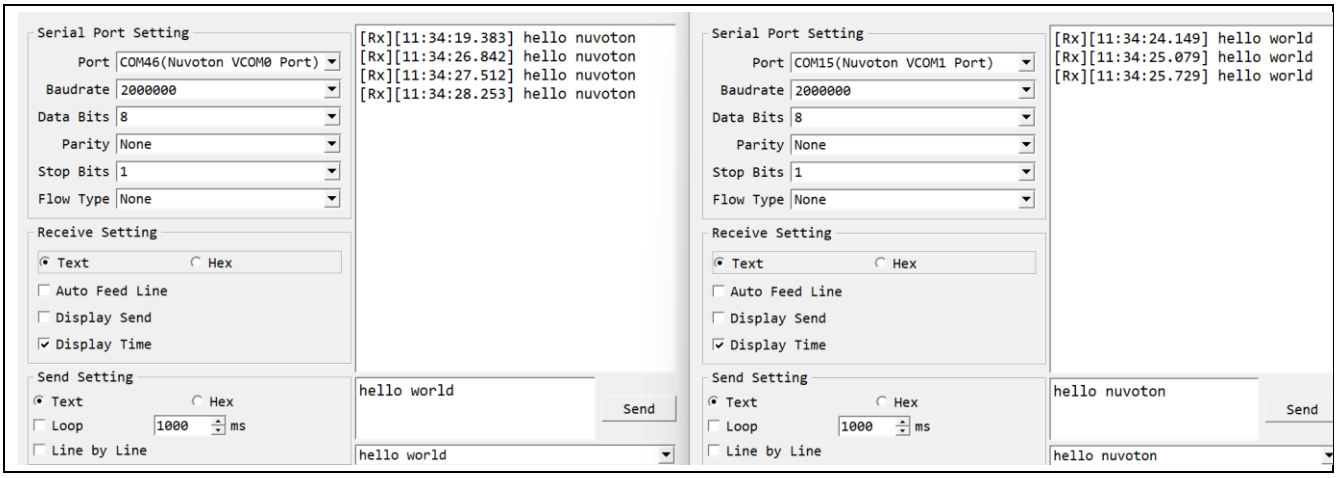

Figure 2-1 VCOM0 and VCOM1 Send Data to Each Other

1. The transceiver pins of UART3 and UART4 are interconnected with wires.

2. The port config is 9.6 kbps, 8 bits of data, no parity, and 1 stop bit.
3. VCOM2 and VCOM3 are connected and send data to each other.

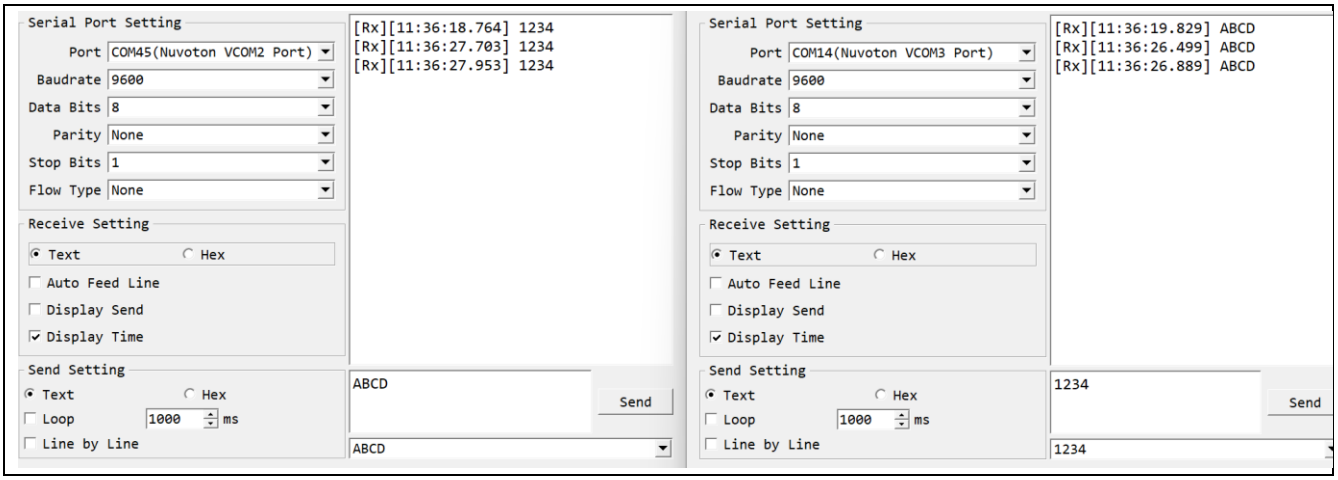

Figure 2-2 VCOM2 and VCOM3 Send Data to Each Other

3. Software & Hardware Requirements

3.1 Software Requirements

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

3.2 Hardware Requirements

- Circuit components
  - NuMaker-M467HJ_V1.0
- Pin Connect
  - Connect HSUSB of the M467HJ evaluation board HSUSB to the PC to virtualize the 4VCOM.
  - Connect the UART pins on the board that need to be tested to display the execution results of the example code.

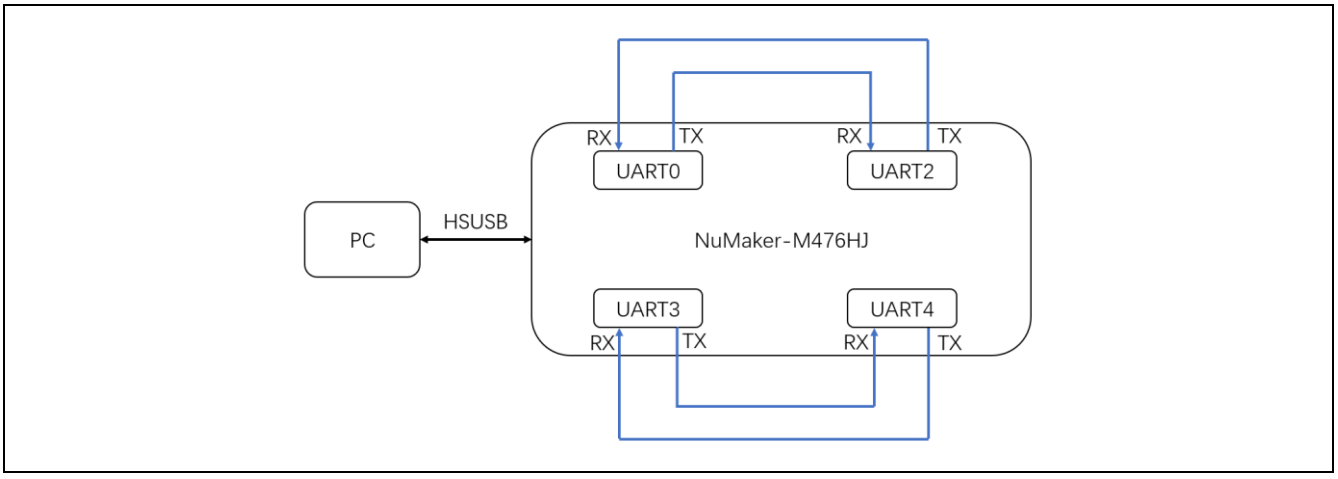

Figure 3–1 Pin Connect

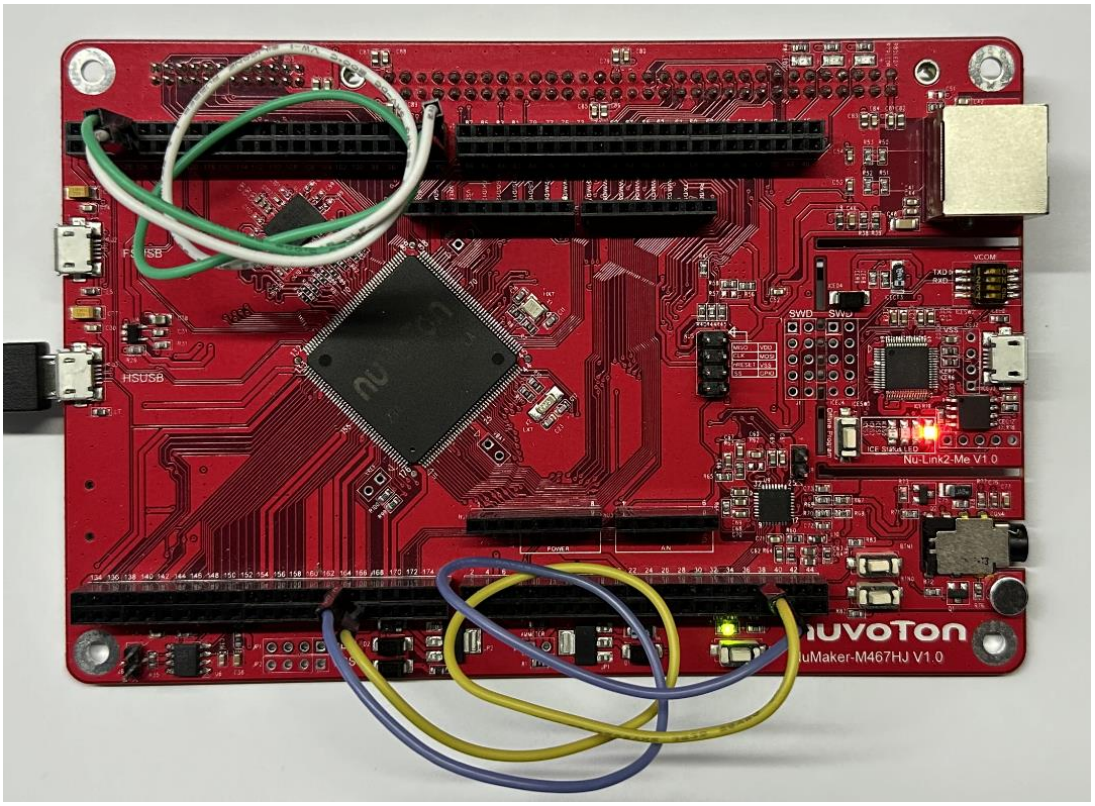

Figure 3‑2 Physical Wiring

4. Directory Information

The directory structure is shown below.

EC_M460_HSUSBD_VCOM_MultiPort_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 *HSUSBD_VCOM_MultiPort.uvprojx*
2. Enter Keil compile mode.
   - Build
   - Download
   - Start/Stop debug session
3. Enter debug mode.
   - Run

6. Revision History

Date	Revision	Description
2024.04.28	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.