M031 SPI和4个UART之间的数据传输

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 M031 来实现 1 个 SPI 和 4 个 UART 之间传输数据
BSP 版本	M031_Series_BSP_CMSIS_V3.05.000
开发平台	NuMaker-M031KI V1.0 NuMaker-M031SE 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

本范例代码主要演示 M031 实现 1 个 SPI 和 4 个 UART 之间进行数据交换。

范例里有两个工程。Master 文件夹下是主机程序，它通过 SPI 和从机通讯，设置 SPI 总线时钟为 16MHZ，定时给从机发送数据、检测到从机请求后读取和解析从机数据、记录并打印传输数据的长度。Slave 文件夹下是从机程序，它收到主机的 SPI 数据后，解析并通过 UART 发送给其他设备；收到其他设备的 UART 数据后，对数据进行编码并通知主机来读取数据。本文介绍了 SPI 速率和角色的设置、PDMA 基本操作模式和 Scatter-Gather 操作模式的配置、SPI+PDMA 的传输和传完判断。

1.1 原理

主机和从机通过 SPI 通讯，如图 1-1 所示数据帧格式为：

帧头（1 字节）+ 帧长度（2 字节）+ 通道（1 字节）+ 数据（128 字节）+ CRC(1 字节) + 帧尾(1 字节)

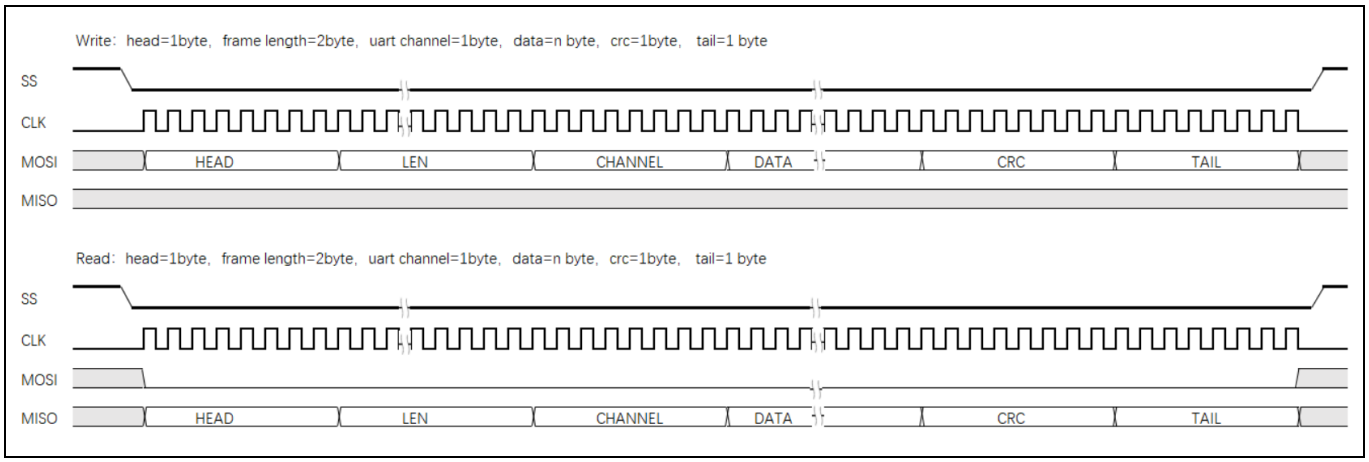

图 1-1 SPI 收发时序

从机和其他设备通过 UART 通讯，采用数据透传。

当主机空闲时定时给从机发送数据，发送间隔受到从机接收 buffer 长度、数据解析和转发速度的影响。如果主机发送速度比从机转发速度快，会导致从机接收 buffer 溢出。本范例设置主机发送的间隔是 15 ms，那么主机 SPI 向 4 个 UART 都发送一帧(128 字节)数据需要 15*4=60 ms。从机 UART 以 115200 bps 波特率再将数据转发出去需要 128*4*10/115200=44 ms。因此从机转发速度比主机发送速度快，不会发生从机接收 buffer 溢出。当检查到从机请求后，读取从机

数据，读取的帧长度固定。主机的 SPI TX 和 RX 都使用 PDMA 基本操作模式搬运数据，完成搬运任务后 PDMA 会回到空闲状态，下一次搬运需要重新启动 PDMA。

当从机收到主机(SPI)数据，解析后通过 UART 主动发送给其他设备。但当从机收到其他设备(UART)数据，不能主动发送给主机(SPI)，需要通过 GPIO 通知主机，等待主机来读取。主机读数据的帧长度固定，如果从机数据不足，帧结尾会用 0 补齐。为了避免 UART 收到少量数据就触发一次 SPI 传输，从机每隔 2 ms 检查是否有数据需要发送给主机。从机的 SPI TX 和 RX 也使用 PDMA 搬运数据。SPI TX 使用 PDMA 的基本操作模式，完成搬运任务后回到空闲状态；SPI RX 使用 PDMA 的 Scatter-Gather 操作模式，采用两张描述符表轮流将数据搬运到指定内存。作为从机，为了保证不丢失数据，需要设置较大的数据接收 buffer，收到的数据尽快发送给对应的设备。buffer 的大小跟接收速度、数据量以及转发速度都有关系。从机 SPI 需要接收主机发送给 4 个串口的数据，并且 SPI 传输速度比 UART 快，所以 SPI 的 buffer 需要比 UART 的 buffer 大。

主机和从机的流程图如图 1-2 所示。

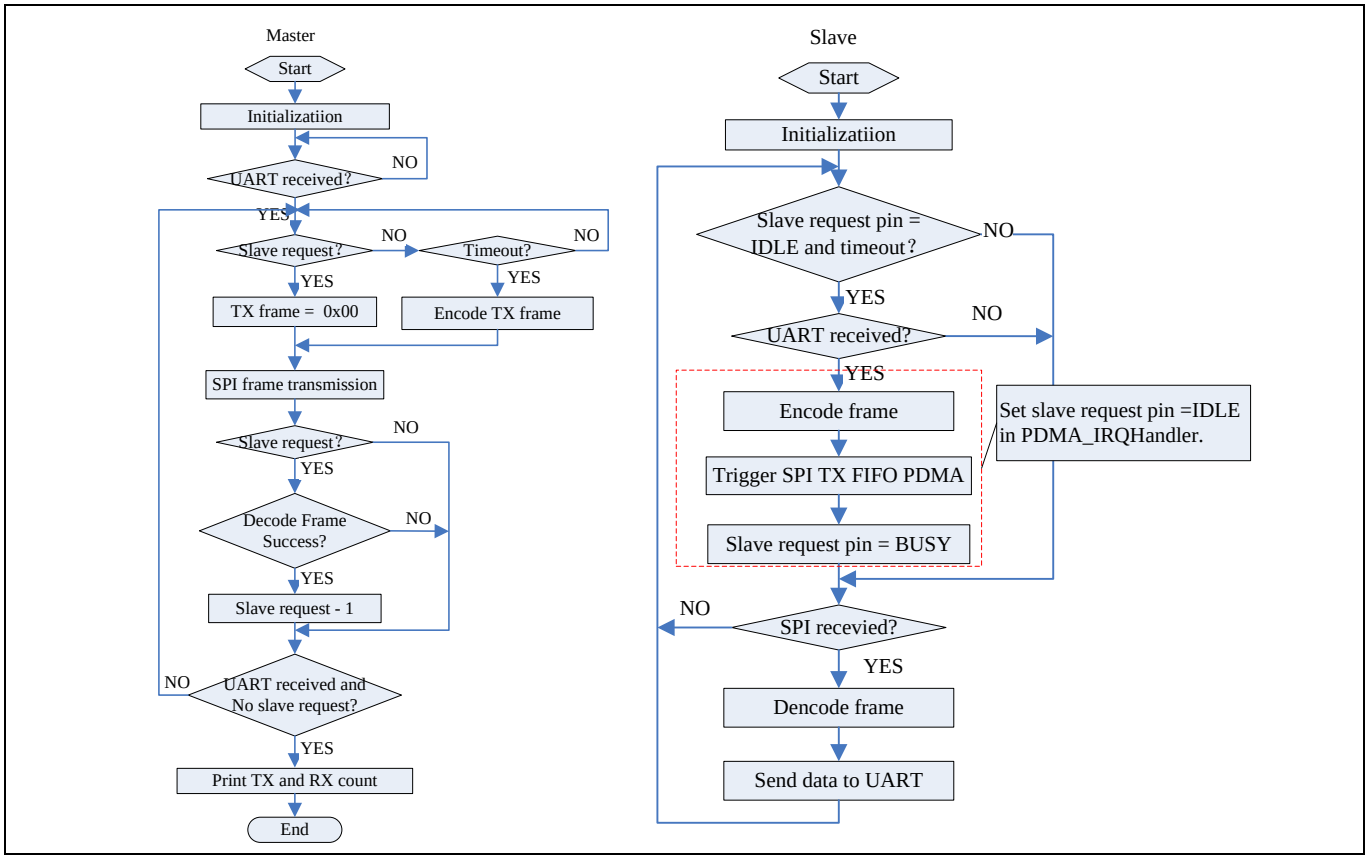

图 1-2 流程图

1.2 执行结果

如果将主机调试串口和从机的 4 个串口都连接到电脑上，程序执行时主机调试串口里将显示执行结果如图 1-3 所示，从机 4 个串口数据传输结果如图 1-4 所示。

```
+-----------------------------------------------------------------------+
|                  M031 SPI To 4 UART Sample                            |
+-----------------------------------------------------------------------+
The I/O connection:
   Master                    Slave
SPI0_SS(PA.3)      ----->  SPI0_SS(PA.3)
SPI0_CLK(PA.2)     ----->  SPI0_CLK(PA.2)
SPI0_MISO(PA.1)    <-----  SPI0_MISO(PA.1)
SPI0_MOSI(PA.0)    ----->  SPI0_MOSI(PA.0)
PB.2               <-----  PB.6
   Slave                     PC
UART0_RX(PB.12)    <-----  UART0_TX
UART0_TX(PB.13)    ----->  UART0_RX
UART1_RX(PB.2)     <-----  UART1_TX
UART1_TX(PB.3)     ----->  UART1_RX
UART2_RX(PB.4)     <-----  UART2_TX
UART2_TX(PB.5)     ----->  UART2_RX
USCI0_DAT1(PA.9)   <-----  UART3_TX
USCI0_DAT0(PA.10)  ----->  UART3_RX

Master sends a frame data to slave every 15 ms.
Slave will send the data to master if UARTs receive data.

SPI Data Format:
START(1) + FRAME LEN(2) + CHANNEL(1) + DATA(128) + CRC(1) + STOP(1)

The SPI master configuration is ready.

If slave has been started, press any key to continue.

Press any key to stop.

The transmission has stopped. Print results:

Master to UART0: sent 459136 bytes, received 142336 bytes.
Master to UART1: sent 459136 bytes, received 165376 bytes.
Master to UART2: sent 459008 bytes, received 152576 bytes.
Master to UART3: sent 459008 bytes, received 162304 bytes.
```

图 1-3 主机的执行结果

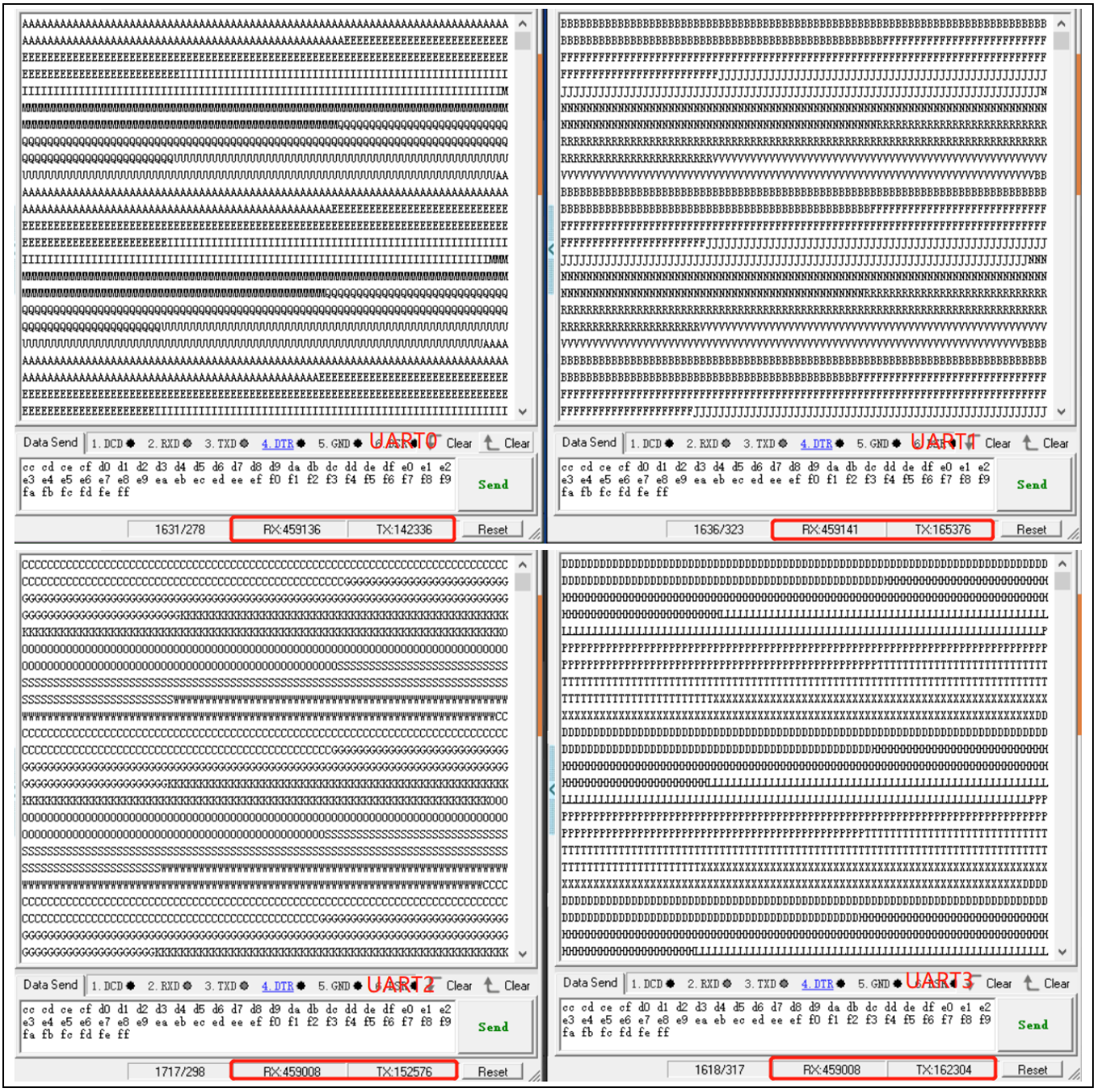

图 1-4 从机 4 个 UART 数据传输结果

2. 代码介绍

- 主机，代码位于 *Master/main.c*

定义 SPI 参数：

```
#define SPI_CLK_FREQ           16000000   /* SPI clock: HZ */
#define DATA_WIDTH             8          /* data width: bit */
#define FIFO_LEVEL             8          /* FIFO level */
```

定义 SPI 通讯协议：

```
#define SPI_FRAME_MAX          (128+6)    /* The length of frame */
#define SPI_HEAD_LEN           1
#define SPI_FRAME_LEN          2
#define SPI_CHANNEL_LEN        1
#define SPI_CRC_LEN            1
#define SPI_TAIL_LEN           1
#define SPI_FRAME_FORMAT_LEN   (SPI_HEAD_LEN+SPI_FRAME_LEN+\
                                SPI_CHANNEL_LEN+SPI_CRC_LEN+SPI_TAIL_LEN)
#define SPI_PAYLOAD_MAX        (SPI_FRAME_MAX - SPI_FRAME_FORMAT_LEN)

#define SPI_FRAME_HEAD         (0xA5)
#define SPI_FRAME_TAIL         (0x55)
```

定义 PDMA 通道：

```
#define SPI_TX_PDMA_CH         0
#define SPI_RX_PDMA_CH         1
```

定义 UART：

包含 UART 的数量、buffer 大小、buffer 的结构

```
/*UART channel */
#define UART_CH_NUM            4            /* The number of uart channels */

/* Buffer size */
#define UART_BUFFER_SIZE 20480

typedef struct
{
    uint8_t buf[UART_BUFFER_SIZE];
    uint32_t in;
    uint32_t out;
} UartBuffer_t;
```

定义主机发送间隔：

```
#define SPI_TX_INTERVAL        15           /* SPI transmission interval ms */
```

SPI 初始化：

主机需要设置 SPI 时钟和 AutoSS。

```
void SPI_Init(void)
{
    /* Configure SPI0 as a master, MSB first, 8-bit transaction, SPI Mode-0 timing,
       clock is 16MHz */
    SPI_Open(SPI_PORT, SPI_MASTER, SPI_MODE_0, DATA_WIDTH, SPI_CLK_FREQ);

    /* Enable the automatic hardware slave select function.
       Select the SS pin and configure as low-active. */
    SPI_EnableAutoSS(SPI0, SPI_SS, SPI_SS_ACTIVE_LOW);
}
```

PDMA 初始化：

SPI TX 和 SPI RX 都采用 PDMA 基本操作模式。搬运指定长度的数据。

```
void PDMA_Init(void)
{
    /* Enable PDMA channels */
    PDMA_Open(PDMA, (1 << SPI_TX_PDMA_CH) | (1 << SPI_RX_PDMA_CH));

    /*=============================================================================
      SPI master PDMA TX channel configuration:
      -----------------------------------------------------------------------
        Word length = 8 bits
        Transfer Count = SPI_FRAME_MAX
        Source = g_au8SPITXBuf
        Source Address = Increasing
        Destination = SPI0->TX
        Destination Address = Fixed
        Burst Type = Single Transfer
    =============================================================================*/
    /* Set transfer width (8 bits) and transfer count */
    PDMA_SetTransferCnt(PDMA, SPI_TX_PDMA_CH, PDMA_WIDTH_8, SPI_FRAME_MAX);
    /* Set source/destination address and attributes */
    PDMA_SetTransferAddr(PDMA,
                         SPI_TX_PDMA_CH,
                         (uint32_t)g_au8SPITXBuf, PDMA_SAR_INC,
                         (uint32_t)&SPI0->TX, PDMA_DAR_FIX);
    /* Set request source; set basic mode. */
    PDMA_SetTransferMode(PDMA, SPI_TX_PDMA_CH, PDMA_SPI0_TX, FALSE, 0);
    /* Single request type. SPI only support PDMA single request type. */
    PDMA_SetBurstType(PDMA, SPI_TX_PDMA_CH, PDMA_REQ_SINGLE, 0);
    /* Disable table interrupt */
    PDMA->DSCT[SPI_TX_PDMA_CH].CTL |= PDMA_DSCT_CTL_TBINTDIS_Msk;

    /*=============================================================================
      SPI master PDMA RX channel configuration:
      -----------------------------------------------------------------------
        Word length = 8 bits
        Transfer Count = SPI_FRAME_MAX
```

```
        Source = SPI0->RX
        Source Address = Fixed
        Destination = g_au8SPIRXBuf
        Destination Address = Increasing
        Burst Type = Single Transfer
    ===========================================================================*/
    /* Set transfer width (8 bits) and transfer count */
    PDMA_SetTransferCnt(PDMA, SPI_RX_PDMA_CH, PDMA_WIDTH_8, SPI_FRAME_MAX);
    /* Set source/destination address and attributes */
    PDMA_SetTransferAddr(PDMA,
                         SPI_RX_PDMA_CH,
                         (uint32_t)&SPI0->RX, PDMA_SAR_FIX,
                         (uint32_t)g_au8SPIRXBuf, PDMA_DAR_INC);
    /* Set request source; set basic mode. */
    PDMA_SetTransferMode(PDMA, SPI_RX_PDMA_CH, PDMA_SPI0_RX, FALSE, 0);
    /* Single request type. SPI only support PDMA single request type. */
    PDMA_SetBurstType(PDMA, SPI_RX_PDMA_CH, PDMA_REQ_SINGLE, 0);
    /* Disable table interrupt */
    PDMA->DSCT[SPI_RX_PDMA_CH].CTL |= PDMA_DSCT_CTL_TBINTDIS_Msk;
}
```

SPI 传输：

```
void SPIFrame_Transfer(void)
{
   /* Trigger SPI TX and RX PDMA function. */
   SPI_TRIGGER_TX_PDMA(SPI0);
   SPI_TRIGGER_RX_PDMA(SPI0);

   /* Wait transfer done */
   while ((!(PDMA_GET_INT_STATUS(PDMA) & PDMA_INTSTS_TDIF_Msk))||/* PDMA transfer done
    interrupt */
   (!(PDMA_GET_TD_STS(PDMA) & (1 << SPI_TX_PDMA_CH)))||/* TX channel transfer done flag */
   (!(PDMA_GET_TD_STS(PDMA) & (1 << SPI_RX_PDMA_CH)))||/* RX Channel transfer done flag */
   (SPI_IS_BUSY(SPI0))); /* Check SPI transfer done flag */

    /* Clear the PDMA transfer done flags */
    PDMA_CLR_TD_FLAG(PDMA, (1 << SPI_TX_PDMA_CH) | (1 << SPI_RX_PDMA_CH));

    /* Disable SPI master's PDMA transfer function */
    SPI_DISABLE_TX_PDMA(SPI0);
    SPI_DISABLE_RX_PDMA(SPI0);
}
```

数据帧编码：

将需要发送的数据按照格式进行编码。

```
/**
  * @brief      Encode raw data to SPI frame.
  * @param      pSrcBuf:     Raw data.
  * @param      dataLen:     Length of raw data.
  * @param      ch:          UART channel [0~3].
  * @param      pFrameBuf:   Encoded data.
  * @param      FrameLen:    Length of Encoded data.
```

```
  * @return     0 successful, other failure.
  */
int8_t SPIFrame_Encode(uint8_t *pSrcBuf, uint16_t dataLen,
                       uint8_t ch,
                       uint8_t *pFramebuf, uint16_t FrameLen)
{
    uint16_t index = 0;
    uint16_t len = 0;
    uint8_t crc8 = 0;

    /* Source data */
    if (!pFramebuf || !pSrcBuf || dataLen == 0)
    {
        return -1;
    }

    len = (dataLen + SPI_FRAME_FORMAT_LEN);

    if (len > FrameLen)
    {
        return -2;
    }

    pFramebuf[index++] = SPI_FRAME_HEAD;

    /* Pack frame length */
    pFramebuf[index++] = (len & 0xFF); /* Low 8 bit */
    pFramebuf[index++] = ((len >> 8) & 0xFF); /* High 8 bit */

    /* Channel */
    pFramebuf[index++] = (char)ch;

    /* Data */
    memcpy(&pFramebuf[index], pSrcBuf, dataLen);
    index += dataLen;

    /* CRC */
    crc8 = Cal_CRC8(pFramebuf, index);
    pFramebuf[index++] = crc8;

    /* Tail */
    pFramebuf[index++] = SPI_FRAME_TAIL;
    return 0;
}
```

数据帧解码：

将收到的从机数据帧解码。

```
/**
  * @brief      Decode SPI frame to data.
  * @param      pFrameBuf:    SPIFrame data to be decoded.
  * @param      FrameLen:     Length of SPIFrame.
  * @param      UartBuffer:  Decoded data buffer struct.
  * @return     0 successful, other failure.
```

```
  */
int SPIFrame_Decode(uint8_t *pFramebuf, uint16_t FrameLen, UartBuffer_t UartBuffer[])
{
    uint16_t index = 0;
    uint8_t ch = 0;
    uint16_t dlen = 0;
    uint8_t crc8 = 0;
    uint8_t *d = 0;
    uint16_t len = 0;
    uint32_t i;

    /* Data exist */
    if (!pFramebuf || FrameLen == 0)
    {
        return -1;
    }

    /* Frame head */
    if (pFramebuf[index++] != SPI_FRAME_HEAD)
    {
        return -2;
    }

    /* Unpack frame length */
    len = ((uint16_t)((pFramebuf[index] & 0x00FF) + ((pFramebuf[index + 1] & 0x00FF) <<
     8)));

    if (len == 0 ||
            len < (SPI_HEAD_LEN + SPI_FRAME_LEN +
                   SPI_CHANNEL_LEN + SPI_CRC_LEN + SPI_TAIL_LEN) ||
            len > FrameLen)
    {
        return -3;
    }

    /* Skip length */
    index += 2;

    /* Channel */
    ch = pFramebuf[index];

    index++;
    dlen = len - SPI_FRAME_FORMAT_LEN;
    d = &pFramebuf[index];
    index += dlen;

    /* Checksum */
    crc8 = Cal_CRC8(&pFramebuf[0], index);

    if (crc8 != pFramebuf[index])
    {
        return -4;
    }

    index++;
```

```
    /* Tail */
    if (pFramebuf[index] != SPI_FRAME_TAIL)
    {
        return -5;
    }

    /* Save data */
    for (i = 0; i < dlen; i++)
    {
        UartBuffer[ch].buf[UartBuffer[ch].in % UART_BUFFER_SIZE] = d[i];
        UartBuffer[ch].in++;
    }

    return 0;
}
```

GPIO 中断：

中断函数里记录从机请求的次数。

```
void GPABGH_IRQHandler(void)
{
    volatile uint32_t temp;

    /* To check if PB.2 interrupt occurred */
    if (GPIO_GET_INT_FLAG(PB, BIT2))
    {
        GPIO_CLR_INT_FLAG(PB, BIT2);
        g_au32SalveReadReq++;
    }
    …
}
```

主函数：

```
int main(void)
{
    …
    j = 'A';
    while (1)
    {
        /* Prepare data frame */
        if (g_au32SalveReadReq) /* SPI master read */
        {
            /* Clear TX frame buffer */
            memset(&g_au8SPITXBuf[0], 0, SPI_FRAME_MAX);
        }
        /* SPI master send */
        else if ((g_au32Delay_ms == 0) &&   /* Time out */
                 (SPI_RX_INT== 1))          /* Slave is ready to receive */
        {
            /* Send interval */
            g_au32Delay_ms = SPI_TX_INTERVAL;
```

```
            /* Fill TX frame buffer */
            memset(payload, j, SPI_PAYLOAD_MAX);


           /* This is the test data.
              Please change it to the data that needs to be transmitted as needed. */
            j++;
            if( j > 'X')
            j = 'A';

            /* Frame encode */
            SPIFrame_Encode(payload, SPI_PAYLOAD_MAX, send_ch, g_au8SPITXBuf,
                            SPI_FRAME_MAX);
            g_au32Sndcnt[send_ch] += SPI_PAYLOAD_MAX;

            /* Change UART number */
            send_ch = (send_ch < UART_CH_NUM - 1) ? (send_ch + 1) : 0;
        }
        else
            continue;

        /* PDMA config */
        SPI_PDMA_ReConfig();

        /* Complete a transfer */
        SPIFrame_Transfer();

        /* Data Parser */
        if (g_au32SalveReadReq)
        {
            flag = SPIFrame_Decode(g_au8SPIRXBuf, SPI_FRAME_MAX, UartBuf);

            /* If the parsing is successful, the slave request flag decreases. */
            if (flag == 0)
            {
                __disable_irq();
                g_au32SalveReadReq--;
                __enable_irq();
            }
        }

     /* Exit: Stop command is received, and no data needs to be read from slave.
        The reason why the pin is used as the judgment condition here is that although
        master has completed the reception processing, the slave may not have executed to
        the place where the pin is set to 1 to indicate idle.
         */
        if ((g_au8Wait == 0) && (SPI_RX_INT == 1)) break;
    }

    /* Disable PDMA channel */
    PDMA_Close(PDMA);

    /* Close SPI0 */
    SPI_Close(SPI0);
```

```
/*=============================================================================================
        Add data processing code
  ============================================================================================*/
    printf("\nThe transmission has stopped. Print results:\n\n");

    for (i = 0; i < UART_CH_NUM; i++)
    {
        printf("Master to UART%d: sent %d bytes, received %d bytes.\n",
               i, g_au32Sndcnt[i], UartBuf[i].in);
    }

    while (1);
}
```

- SPI 从机，代码位于 *Slave/main.c*

Buffer 定义：

```
/* Buffer size */
#define UART_BUFFER_SIZE 1024
#define SPI_BUFFER_SIZE (SPI_FRAME_MAX*72)

typedef struct
{
    uint8_t buf[UART_BUFFER_SIZE];
    uint32_t in;
    uint32_t out;
} UartBuffer_t;

typedef struct
{
    uint8_t buf[SPI_BUFFER_SIZE];
    uint32_t in;
    uint32_t out;
} SpiBuffer_t;
```

UART 初始化：

```
void UART_Init(void)
{
    /* Configure UART: 115200, 8-bit word, no parity bit, 1 stop bit. */
    /* UART0 init */
    UART_Open(UART0, 115200);
    /* Set Rx Time-out counter */
    UART_SetTimeoutCnt(UART0, 0x10);
    /* Set RX FIFO Interrupt Trigger Level as 8 bytes */
    UART0->FIFO &= ~ UART_FIFO_RFITL_Msk;
    UART0->FIFO |= UART_FIFO_RFITL_8BYTES;

    /* UART1 init */
    UART_Open(UART1, 115200);
    /* Set Rx Time-out counter */
    UART_SetTimeoutCnt(UART1, 0x10);
    /* Set RX FIFO Interrupt Trigger Level as 8 bytes */
```

```
    UART1->FIFO &= ~ UART_FIFO_RFITL_Msk;
    UART1->FIFO |= UART_FIFO_RFITL_8BYTES;

    /* UART2 */
    UART_Open(UART2, 115200);
    /* UUART0 */
    UUART_Open(UUART0, 115200);

    /* Enable RDA\Time-out Interrupt */
    UART_EnableInt(UART0, (UART_INTEN_RDAIEN_Msk | UART_INTEN_RXTOIEN_Msk));
    UART_EnableInt(UART2, (UART_INTEN_RDAIEN_Msk));
    NVIC_EnableIRQ(UART02_IRQn);

    UART_EnableInt(UART1, (UART_INTEN_RDAIEN_Msk | UART_INTEN_RXTOIEN_Msk));
    NVIC_EnableIRQ(UART13_IRQn);

    UUART_ENABLE_TRANS_INT(UUART0, UUART_INTEN_RXENDIEN_Msk);
    NVIC_EnableIRQ(USCI_IRQn);
}
```

SPI 初始化：

从机不需要设置时钟和 SS。

```
void SPI_Init(void)
{
    /* Configure SPI0 as a slave, MSB first, 8-bit transaction, SPI Mode-0 timing*/
    SPI_Open(SPI_PORT, SPI_SLAVE, SPI_MODE_0, DATA_WIDTH, (uint32_t) NULL);
}
```

PDMA 初始化：

SPI TX 使用 PDMA 基本操作模式，SPI RX 使用 PDMA Scatter-Gather 操作模式。

```
void PDMA_Init(void)
{
    /* Enable PDMA channels */
    PDMA_Open(PDMA, (1 << SPI_TX_PDMA_CH) | (1 << SPI_RX_PDMA_CH));


/*=============================================================================================
      SPI slave PDMA TX channel configuration:
      -----------------------------------------------------------------------------------------
        Word length = 8 bits
        Transfer Count = SPI_FRAME_MAX
        Source = g_u8SpiTxBuf
        Source Address = Increasing
        Destination = SPI0->TX
        Destination Address = Fixed
        Burst Type = Single Transfer
          Operation Mode = Basic mode

=============================================================================================*/
    /* Set transfer width (8 bits) and transfer count */
```

```
    PDMA_SetTransferCnt(PDMA, SPI_TX_PDMA_CH, PDMA_WIDTH_8, SPI_FRAME_MAX);
    /* Set source/destination address and attributes */
    PDMA_SetTransferAddr(PDMA,
                         SPI_TX_PDMA_CH,
                         (uint32_t)g_u8SpiTxBuf, PDMA_SAR_INC,
                         (uint32_t)&SPI0->TX, PDMA_DAR_FIX);
    /* Set request source; set basic mode. */
    PDMA_SetTransferMode(PDMA, SPI_TX_PDMA_CH, PDMA_SPI0_TX, FALSE, 0);
    /* Single request type. SPI only support PDMA single request type. */
    PDMA_SetBurstType(PDMA, SPI_TX_PDMA_CH, PDMA_REQ_SINGLE, 0);
    /* Disable table interrupt */
    PDMA->DSCT[SPI_TX_PDMA_CH].CTL |= PDMA_DSCT_CTL_TBINTDIS_Msk;


/*==============================================================================================
      SPI slave PDMA RX channel configuration:
      ----------------------------------------------------------------------------------------
        Channel = SPI_RX_PDMA_CH
        Operation mode = scatter-gather mode
        First scatter-gather descriptor table = DMA_DESC[0]
        Request source = PDMA_MEM(memory to memory)

        Transmission flow:
                                      loop around
                                  SPI_BUFFER_SIZE/2 times
     ------------------------                                 ------------------------
    |                        | ------------------------> |                        |
    |  DMA_DESC[0]           |                           |  DMA_DESC[1]           |
    |  (Descriptor table 1)  |                           |  (Descriptor table 2)  |
    |                        | <------------------------ |                        |
     ------------------------                                 ------------------------

    Note: Each configuration table in SRAM need to be reloaded after transmission
    finished.

==============================================================================================*/
    /* Enable Scatter Gather mode, assign the first scatter-gather descriptor table is
       table 1, and set transfer mode as SPI RX to memory */
    PDMA_SetTransferMode(PDMA, SPI_RX_PDMA_CH, PDMA_SPI0_RX, TRUE,
    (uint32_t)&DMA_DESC[0]);

    /* Scatter-Gather descriptor table configuration in SRAM */
    /*------------------------------------------------------------------------------------------
      Descriptor table 1 configuration:

                          SPI0_RX                    SpiRxBuf.buf[0]
                    ----------------------------->-----------------------
                  /| [0] | [1] | [2] | [3] | | [0] | [1] | [2] | [3] |\
                   |     |     |     |     | |     |     |     |     |
SPI_BUFFER_SIZE/2  |          ...          | |          ...          | SPI_BUFFER_SIZE/2
                   |     |     |     |     | |     |     |     |     |
                  \|[508]|[509]|[510]|[511]| |[508]|[509]|[510]|[511]|/
                   ------------------------  -----------------------
                    \                       /   \                     /
                         8bits(one byte)            8bits(one byte)
```

```
        Operation mode = scatter-gather mode
        Next descriptor table = DMA_DESC[1](Descriptor table 2)
        transfer done and timeout interrupt = enable

        Transfer count = SPI_BUFFER_SIZE/2
        Transfer width = 8 bits
        Source address = SPI0_RX
        Source address increment size = fixed address(no increment)
        Destination address = SpiRxBuf.buf[0]
        Destination address increment size = 8 bits(one byte)
        Transfer type = single transfer

        Total transfer length = (SPI_BUFFER_SIZE/2) * 8 bits
    ------------------------------------------------------------------------------------*/
    DMA_DESC[0].ctl = \
             ((SPI_BUFFER_SIZE / 2 - 1)
              << PDMA_DSCT_CTL_TXCNT_Pos) |/* Transfer count is SPI_BUFFER_SIZE/2 */ \
              PDMA_WIDTH_8 |    /* Transfer width is 8 bits */ \
              PDMA_SAR_FIX |    /* Source increment size is  fixed(no increment) */ \
              PDMA_DAR_INC |    /* Destination increment size is 8 bits */ \
              PDMA_REQ_SINGLE | /* Transfer type is single transfer type */ \
              PDMA_BURST_1 |    /* Burst size is 1. No effect in single transfer type */ \
                 PDMA_OP_SCATTER;  /* Operation mode is scatter-gather mode */
    /* Configure source address */
    DMA_DESC[0].src = (uint32_t)(&SPI0->RX);
    /* Configure destination address */
    DMA_DESC[0].dest = (uint32_t)&SpiRxBuf.buf[0];/* Buf 1 */
    /* Configure next descriptor table address */
    DMA_DESC[0].offset = (uint32_t)&DMA_DESC[1] - (PDMA->SCATBA);/* Next operation table
    is table2*/

    /*-------------------------------------------------------------------------------------
      Descriptor table 2 configuration:
                      SPI0_RX                 SpiRxBuf.buf[SPI_BUFFER_SIZE/2]
                    ------------------------->-----------------------
                  /| [0] | [1] | [2] | [3] | | [0] | [1] | [2] | [3] |\
                   |     |     |     |     | |     |     |     |     |
    SPI_BUFFER_SIZE/2 |          ...          | |          ...          | SPI_BUFFER_SIZE/2
                   |     |     |     |     | |     |     |     |     |
                  \|[508]|[509]|[510]|[511]| |[508]|[509]|[510]|[511]|/
                    ----------------------   ----------------------
                    \                     /   \                     /
                       8bits(one byte)             8bits(one byte)

        Operation mode = scatter-gather mode
        Next descriptor table = DMA_DESC[0](Descriptor table 1)
        transfer done and timeout interrupt = enable

        Transfer count = SPI_BUFFER_SIZE/2
        Transfer width = 8 bits
        Source address = SPI0_RX
        Source address increment size = fixed address(no increment)
        Destination address = SpiRxBuf.buf[SPI_BUFFER_SIZE/2]
        Destination address increment size = 8 bits(one byte)
        Transfer type = single transfer
```

```
        Total transfer length = (SPI_BUFFER_SIZE/2) * 8 bits
    ------------------------------------------------------------------------------*/
    DMA_DESC[1].ctl = \
            ((SPI_BUFFER_SIZE / 2 - 1)
             << PDMA_DSCT_CTL_TXCNT_Pos) | /* Transfer count is SPI_BUFFER_SIZE/2 */ \
             PDMA_WIDTH_8 |    /* Transfer width is 8 bits */ \
             PDMA_SAR_FIX |    /* Source increment size is fixed(no increment) */ \
             PDMA_DAR_INC |    /* Destination increment size is 8 bits */ \
             PDMA_REQ_SINGLE | /* Transfer type is single transfer type */ \
             PDMA_BURST_1 |    /* Burst size is 1. No effect in single transfer type */ \
             PDMA_OP_SCATTER;  /* Operation mode is scatter-gather mode */
    /* Configure source address */
    DMA_DESC[1].src = (uint32_t)(&SPI0->RX);
    /* Configure destination address */
    DMA_DESC[1].dest = (uint32_t)&SpiRxBuf.buf[SPI_BUFFER_SIZE / 2] ; /* Buf 2 */
    /* Configure next descriptor table address */
    DMA_DESC[1].offset = (uint32_t)&DMA_DESC[0] - (PDMA->SCATBA);/* Next operation table
    is table1*/

    /* PDMA channel timeout */
    PDMA_SetTimeOut(PDMA, SPI_RX_PDMA_CH, 1, PDMA_TIME); /* Channel 0 timeout period = CLK
     * TOC0 */

    /* Enable interrupt */
    PDMA_EnableInt(PDMA, SPI_TX_PDMA_CH, PDMA_INT_TRANS_DONE); /* Channel0 transfer done*/
    PDMA_EnableInt(PDMA, SPI_RX_PDMA_CH, PDMA_INT_TRANS_DONE); /* Channel1 transfer done*/
    PDMA_EnableInt(PDMA, SPI_RX_PDMA_CH, PDMA_INT_TIMEOUT); /* Channel timeout */
    NVIC_EnableIRQ(PDMA_IRQn);
}
```

- PDMA 中断处理：

```
void PDMA_IRQHandler(void)
{
    uint32_t status = PDMA_GET_INT_STATUS(PDMA); /* Interrupt status */
    uint32_t TDSTS;
    uint32_t cnt = PDMA_GET_TRANS_CNT(PDMA, SPI_RX_PDMA_CH); /* Channel transfer count */
    static  uint32_t cntbak  = SPI_BUFFER_SIZE / 2 - 1;

    if (status & PDMA_INTSTS_TDIF_Msk)     /* Channel transfer done */
    {
        TDSTS = PDMA_GET_TD_STS(PDMA);

        /* Check channel transfer done status */
        if (TDSTS & (1 << SPI_TX_PDMA_CH)) /* SPI TX channel done*/
        {
            /* Wait for SPI transmission to complete */
            while (SPI_IS_BUSY(SPI0));

            SPI_PDMA_Disable(SPI_TX_PDMA_CH);

            /* Clear transfer done flag of channel SPI_TX_PDMA_CH */
            PDMA_CLR_TD_FLAG(PDMA, (1 << SPI_TX_PDMA_CH));
        }

        if (TDSTS & (1 << SPI_RX_PDMA_CH)) /* SPI RX channel finish a descriptor table*/
```

```
        {
            /* Update receive count */
            SpiRxBuf.in = (SpiRxBuf.in + cntbak + 1); /* Count */

            cntbak = SPI_BUFFER_SIZE / 2 - 1; /* Channel cnt backup */

            /* Clear transfer done flag of channel SPI_RX_PDMA_CH */
            PDMA_CLR_TD_FLAG(PDMA, (1 << SPI_RX_PDMA_CH));
        }
    }
    else if (status & (1 << (8 + SPI_RX_PDMA_CH)))   /* Channel SPI_RX_PDMA_CH timeout */
    {
        /* Update receive count */
        SpiRxBuf.in = (SpiRxBuf.in + cntbak - cnt);  /* Count */
        cntbak = cnt; /* channel cnt backup */

        /* Restart timeout */
        PDMA_SetTimeOut(PDMA, SPI_RX_PDMA_CH, 0, 0);
        PDMA_CLR_TMOUT_FLAG(PDMA, SPI_RX_PDMA_CH);
        PDMA_SetTimeOut(PDMA, SPI_RX_PDMA_CH, 1, PDMA_TIME);
    }
}
```

- UART 接收数据：

```
void UART02_IRQHandler(void)
{
    /* UART0 */
    if (UART_GET_INT_FLAG(UART0, UART_INTSTS_RDAINT_Msk | UART_INTSTS_RXTOINT_Msk))
    {
        /* Get all the input characters */
        while (!UART_GET_RX_EMPTY(UART0))
        {
            /* Get the character from UART Buffer */
            UartBuf[UART0_INDEX].buf[UartBuf[UART0_INDEX].in % UART_BUFFER_SIZE] =
            UART_READ(UART0);
            UartBuf[UART0_INDEX].in++;
        }
    }

    /* UART2 */
    if (UART_GET_INT_FLAG(UART2, UART_INTSTS_RDAINT_Msk))
    {
        /* Get all the input characters */
        while (!UART_GET_RX_EMPTY(UART2))
        {
            /* Get the character from UART Buffer */
            UartBuf[UART2_INDEX].buf[UartBuf[UART2_INDEX].in % UART_BUFFER_SIZE] =
            UART_READ(UART2);
            UartBuf[UART2_INDEX].in++;
        }

    }
}

void UART13_IRQHandler(void)
```

```
{
    /* UART1 */
    if (UART_GET_INT_FLAG(UART1, UART_INTSTS_RDAINT_Msk | UART_INTSTS_RXTOINT_Msk))
    {
        /* Get all the input characters */
        while (!UART_GET_RX_EMPTY(UART1))
        {
            /* Get the character from UART Buffer */
            UartBuf[UART1_INDEX].buf[UartBuf[UART1_INDEX].in % UART_BUFFER_SIZE] =
            UART_READ(UART1);
            UartBuf[UART1_INDEX].in++;
        }
    }
}

void USCI01_IRQHandler(void)
{
    if (UUART_GET_PROT_STATUS(UUART0) & UUART_PROTSTS_RXENDIF_Msk)
    {
        /* Clear RX end interrupt flag */
        UUART_CLR_PROT_INT_FLAG(UUART0, UUART_PROTSTS_RXENDIF_Msk);

        /* Get all the input characters */
        while (!UUART_IS_RX_EMPTY(UUART0))
        {
            /* Get the character from UART Buffer */
            UartBuf[UART3_INDEX].buf[UartBuf[UART3_INDEX].in % UART_BUFFER_SIZE]  =
            UUART_READ(UUART0);
            UartBuf[UART3_INDEX].in++;
        }
    }
}
```

- 主函数：

```
int main(void)
{
   …
   while (1)
   {
       /* UART ==> SPI */
       if ((SPI_TX_REQ_PIN == TX_IDLE) && (g_u32Delay_ms == 0)) /* TX idle and timeout */
       {
           /* Transfer every 2 ms */
           g_u32Delay_ms = SPI_TX_INTERVAL;

           /* Read the data received by UART */
           rLength = UartBuffer_Read(&UartBuf[index], buf, SPI_PAYLOAD_MAX);

           if (rLength > 0)
           {
               /* Encode data into SPI frame */
               if (SPIFrame_Encode(buf, rLength, index, g_u8SpiTxBuf, SPI_FRAME_MAX) == 0)
               {
                   SPI_PDMA_Enable(SPI_TX_PDMA_CH);     /* Start PDMA ==> SPI */
               }
```

```
            }

            /* UART channel index */
            index = (index == (UART_CH_NUM - 1)) ? UART0_INDEX : (index + 1);
        }

        /* SPI ==> UART */
        /* Read the data received by SPI */
        rLength = SpiBuffer_Read(&SpiRxBuf, spiFrame, SPI_FRAME_MAX);

        if (rLength > 0)
        {
            /* Parse the data received by SPI */
            if (SPIFrame_Decode(spiFrame, rLength, buf, &rlen, &channel) == 0)
            {
                Uart_Send_Data(channel, buf, rlen);
            }
        }
    }
}
```

3. 软件与硬件需求

3.1 软件需求

- BSP 版本
  - M031_Series_BSP_CMSIS_V3.05.000
- IDE 版本
  - Keil uVersion 5.26

3.2 硬件需求

- 电路组件
  - NuMaker-M031KI V1.0
  - NuMaker-M031SE V1.1
- 示意图
  - 将两块开发板的 SPI0_MOSI (PA.0)、SPI0_MISO (PA.1)、SPI0_CLK (PA.2)、SPI0_SS (PA.3)、GND 分别连接在一起。
  - NuMaker-M031KI 的 UART0 连接到电脑上。控制范例代码的执行和显示执行结果。
  - NuMaker-M031SE 的 UART0、UART1、UART2、UUART0 连接到电脑上。发送数据和显示串口收到的数据。

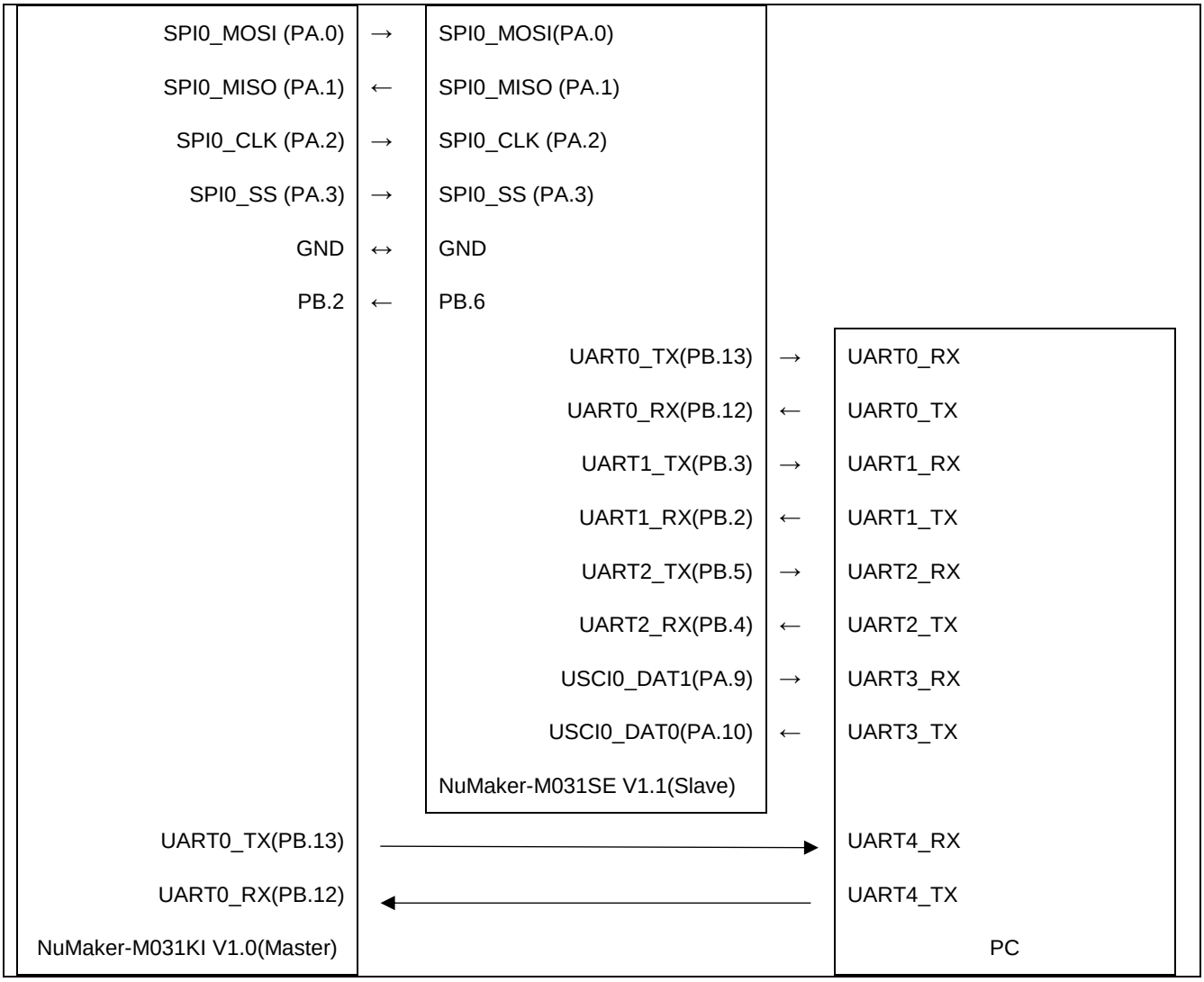
SPI0_MOSI (PA.0)
SPI0_MISO (PA.1)
SPI0_CLK (PA.2)
SPI0_SS (PA.3)
GND
PB.2
NuMaker-M031KI V1.0(Master)
SPI0_MOSI(PA.0)
SPI0_MISO (PA.1)
SPI0_CLK (PA.2)
SPI0_SS (PA.3)
GND
PB.6
UART0_TX(PB.13)
UART0_RX(PB.12)
UART1_TX(PB.3)
UART1_RX(PB.2)
UART2_TX(PB.5)
UART2_RX(PB.4)
USCI0_DAT1(PA.9)
USCI0_DAT0(PA.10)
NuMaker-M031SE V1.1(Slave)
UART0_RX
UART0_TX
UART1_RX
UART1_TX
UART2_RX
UART2_TX
UART3_RX
UART3_TX
UART0_TX(PB.13)
UART0_RX(PB.12)
UART4_RX
UART4_TX
PC

图 3-1 引脚连接

4. 目录信息

EC_M031_SPI_TO_4_UART_V1.00	
Library	Sample code header and source files
CMSIS	Cortex® Microcontroller Software Interface Standard (CMSIS) by Arm® Corp.
Device	CMSIS compliant device header file
StdDriver	All peripheral driver header and source files
SampleCode	
ExampleCode	Source file of example code

图 4-1 目录信息

5. 范例程序执行

1. 根据目录信息章节进入 ExampleCode 路径，双击 *M031_SPI_TO_4_UART.uvmpw*。
2. 进入 Master 编译模式界面
   - 设置 Master 为 Active 工程
   - 编译
   - 下载代码至内存
3. 进入 Slave 编译模式界面
   - 设置 Slaver 为 Active 工程
   - 编译
   - 下载代码至内存
4. 打印主机调试信息
   - PC 上打开串口调试助手
   - 波特率：115200bps
5. 打开从机 4 个通讯串口
   - PC 上打开串口调试助手
   - 波特率：115200bps
6. 运行
   - 从机按复位键，执行代码
   - 主机按复位键，执行代码
7. 数据通讯
   - 主机调试串口助手里发送任意字符开始测试
   - 在从机 4 个串口助手里设置每 500 ms 发送 512 字节数据

- 停止从机 4 个串口的发送
- 主机串口调试助手里发送任意字符结束测试

6. 修订纪录

Date	Revision	Description
2022.10.20	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.