M031 Data Transfer between SPI and 4 UARTs

Example Code Introduction for 32-bit NuMicro® Family

Document Information

Application	This example code is used for the M031 series microcontroller (MCU) to transfer data between SPI and 4 UARTs.
BSP Version	M031_Series_BSP_CMSIS_V3.05.000
Hardware	NuMaker-M031KI V1.0 NuMaker-M031SE V1.1

The information described in this document is the exclusive intellectual property of Hugoton 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 is used for the M031 series microcontroller (MCU) to send and receive data between SPI and 4 UARTs.

There are two projects in the example. A master project in the Master directory communicates with slave through SPI, sets SPI bus clock frequency to 16 MHz, sends data to slave at intervals, reads and parses slave data after detecting the slave request, and records and prints the length of the transmitted data. A slave project in the Slave directory parses and sends data to other devices through UARTs after receiving SPI data from master, and similarly encodes data and notifies master to read data after receiving UART data from other devices. This document introduces the example code of SPI clock frequency and role setting, PDMA Basic operation mode and Scatter-Gather operation mode setting, and SPI+PDMA transmission and judgment of transmission completion.

1.1 Principle

Master and slave communicate through SPI. As shown in Figure 1-1, the data frame format is:

Head (1 byte) + LEN (2 bytes) + Channel (1 byte) + Data (128 bytes) + CRC(1 byte) + Tail(1 byte)

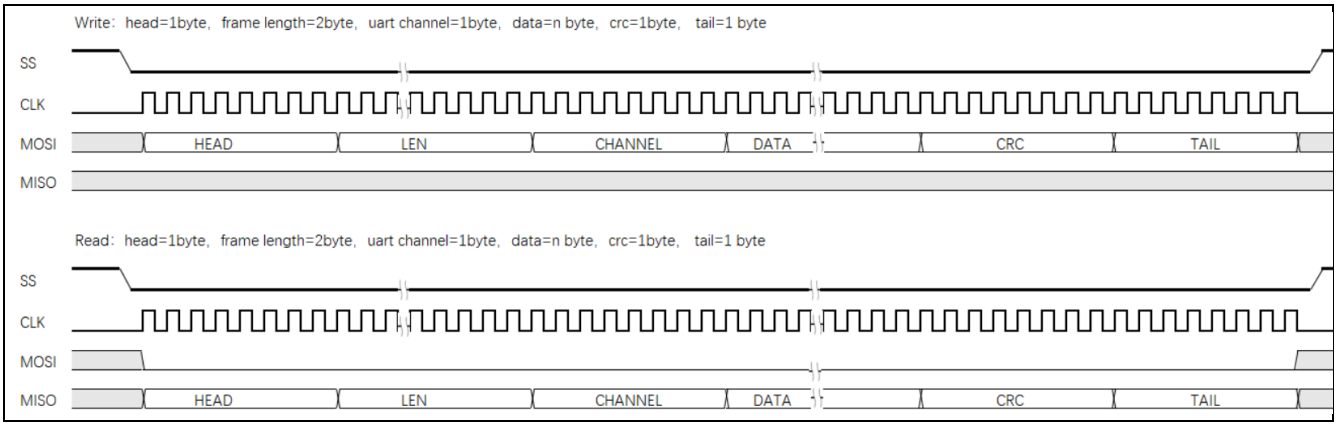

Figure 1-1 SPI Transfer Timing

Slave and other devices communicate through UARTs, using transparent data transmission.

When master is idle, it sends data to slave at regular interval. The interval is affected by the length of the slave's receiving buffer, the speed of data parsing and forwarding. If the sending speed of master is faster than the forwarding speed of slave, the receiving buffer of slave will overflow. In this example, the interval is set to 15 ms. Then it takes 15*4=60 ms for SPI of master to send a frame (128 bytes) of data to each UART. It takes 128*4*10/115200=44 ms for UARTs（115200 bps）of slave to forward the data. Therefore, the forwarding speed of slave is faster than the sending speed of master, and no slave receiving buffer overflow will occur. When the slave request is detected, master reads slave data. Both SPI TX and SPI RX of master use PDMA Basic operation mode to transfer data, and PDMA will be updated to idle state after the transfer task is finished. The PDMA needs to be restarted for next transfer.

When slave receives SPI data from master, it parses and sends data to other devices through UART. However, after slave receiving UART data from other devices, it cannot actively send data to master by SPI. It needs to notify master through GPIO and wait for master to read data. The frame length of the host read data is fixed. If the slave data length is not enough, the end of the frame will be filled with 0. To avoid triggering SPI transmission when UART receives little data, slave checks whether there is data to be sent to master every 2 ms. SPI TX and SPI RX of slave also use PDMA to transfer data. SPI TX uses PDMA Basic operation mode to transfer data, and PDMA will be updated to idle state after the transfer task is finished. SPI RX uses PDMA Scatter-Gather operation mode to transfer data to the specified memory in turn by using two descriptor tables. As a slave, in order to ensure no data loss, it is necessary to set a large buffer for the receiving data and send the received data to the corresponding device as soon as possible. The size of the receiving buffer is affected by the receiving speed, data length, and forwarding speed. SPI of slave needs to receive the data sent by master to 4 UARTs, and the transmission speed of SPI is faster than the speed of UART, so the buffer of SPI needs to be larger than the buffer of UART.

The master and slave flow chart is shown in Figure 1-2.

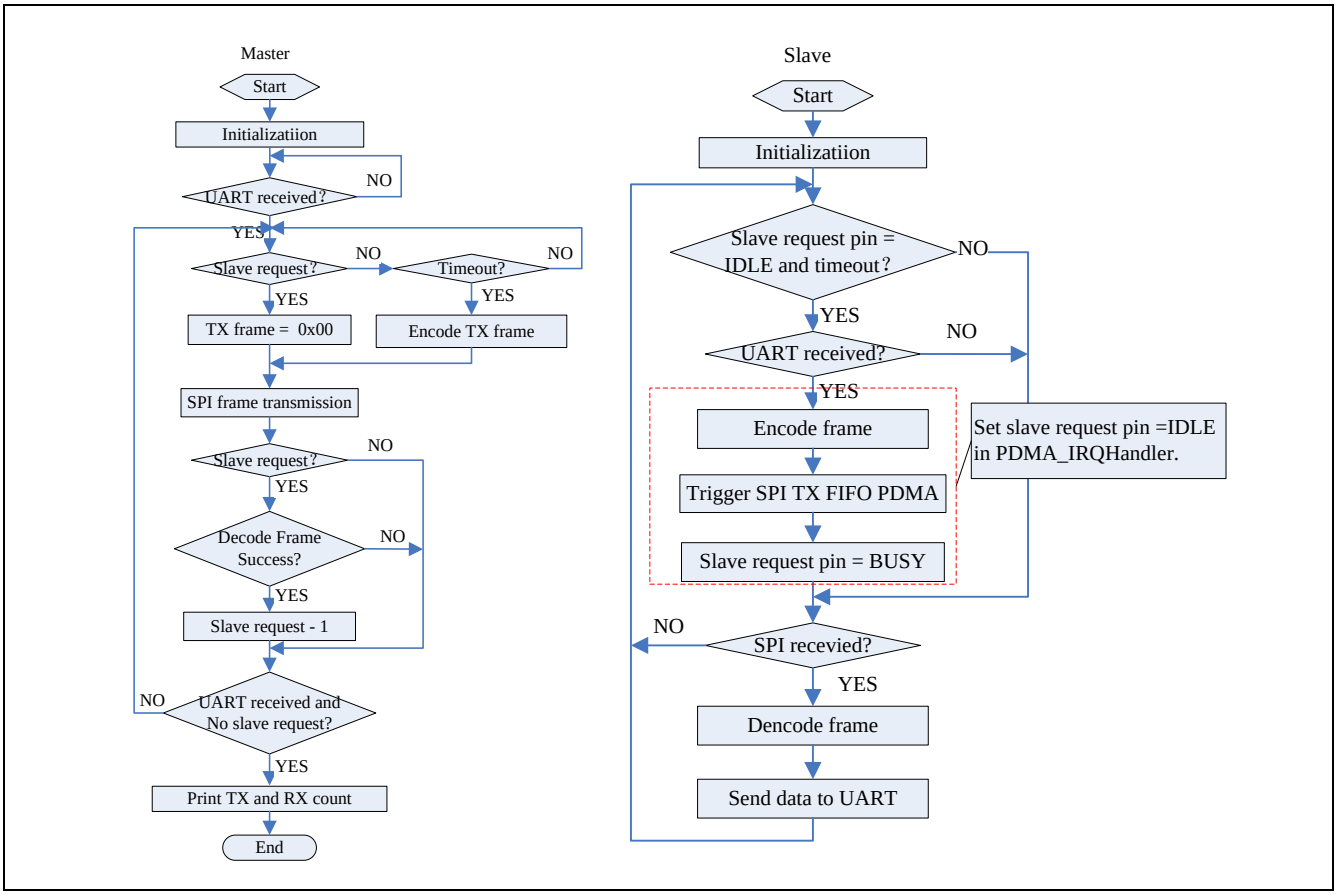

Figure 1-2 Mast and Slave Flow Chart

1.1 Demo Result

If the master debug UART and Slave's 4 UARTs are connected to PC when executing the program, the master debug UART tool can display execution result as shown in Figure 1-3. Also, data transmission results of Slave's 4 UARTs are shown in Figure 1-4.

```
+-----------------------------------------------------------------------+
|                  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.
```

Figure 1-3 Master Execution Results

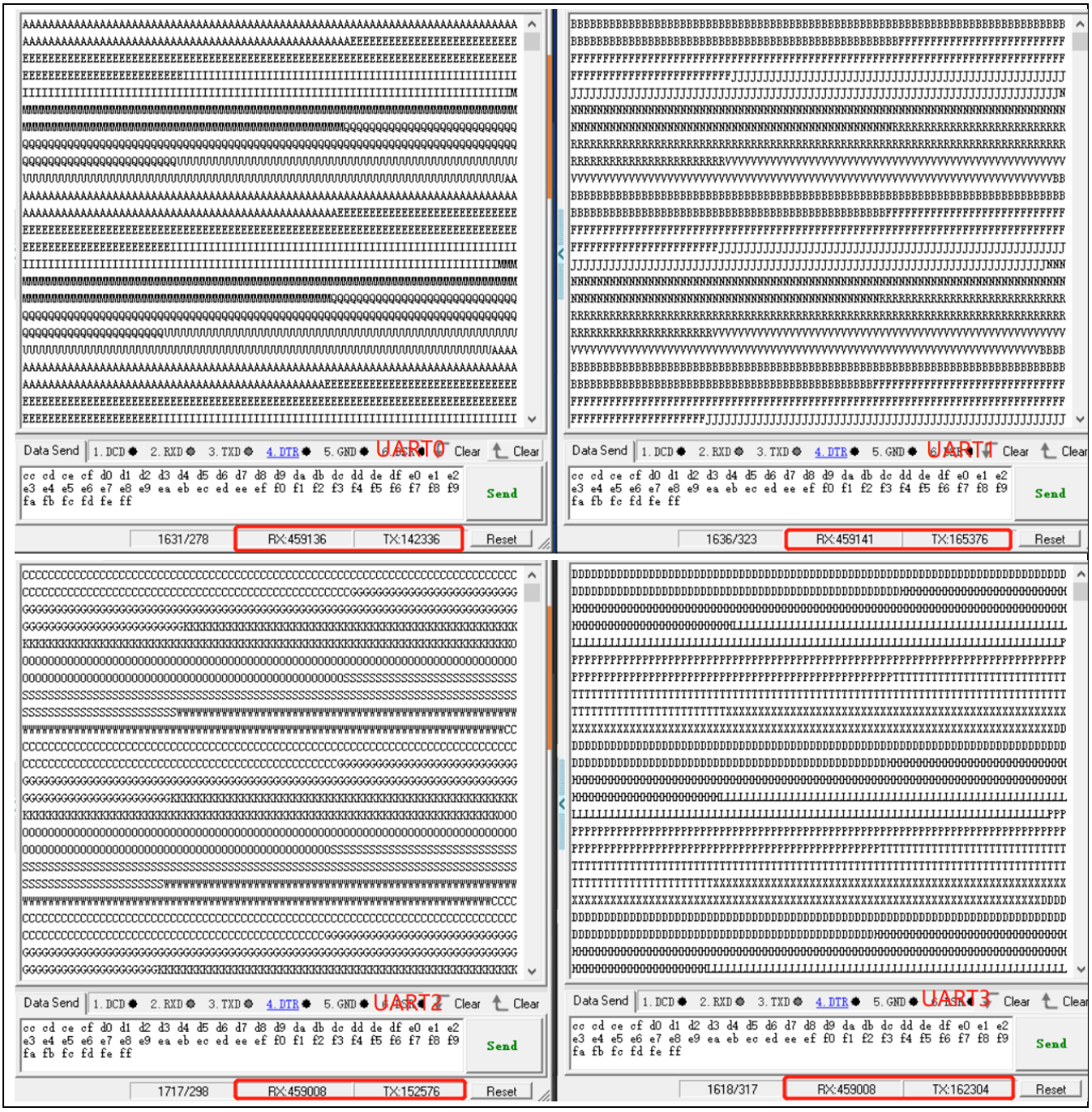

Figure 1-4 Data Transfer Results of Slave's 4 UARTs

2. Code Description

- Master code in *Master/main.c*

Define SPI parameters:

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

Define SPI communication protocol:

```
#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)
```

Define PDMA channel:

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

Define UART:

Include UART number, buffer size, and buffer struct.

```
/*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;
```

Define master send interval:

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

SPI initialization:

Master needs to set SPI clock and automatic slave selection.

```
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 initialization:

Both SPI TX and SPI RX use PDMA Basic operation mode and specify the transferred data length.

```
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 transfer:

```
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);
}
```

Data frame encoding:

Encode data to be sent according to format.

```
/**
  * @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;
}
```

Data Frame decoding:

Decode the received slave data.

```
/**
  * @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 interrupt hander:

Record the number of slave requests in the GPIO interrupt hander function.

```
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++;
    }
    …
}
```

Main function:

```
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);
}
```

- Slave code in *Slave/main.c*

Define buffer:

```
/* 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 initialization:

```
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 initialization:

Slave does not need to set SPI clock and slave selection.

```
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 initialization:

SPI TX uses PDMA Basic operation mode. SPI RX uses PDMA Scatter-Gather operation mode.

```
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 interrupt handler:

```
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 receives data:

```
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++;
        }
    }
}
```

Main function:

```
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. Software and Hardware Requirements

3.1 Software Requirements

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

3.2 Hardware Requirements

- Circuit components
  - NuMaker-M031KI V1.0
  - NuMaker-M031SE V1.1
- Pin Connect
  - SPI0_MOSI (PA.0), SPI0_MISO (PA.1), SPI0_CLK (PA.2), SPI0_SS (PA.3) and GND of the two evaluation boards are connected together.
  - UART0 of NuMaker-M031KI V1.0 is connected to PC to control the execution of the sample code and display the execution results.
  - UART0, UART1, UART2, UUART0 of NuMaker-M031SE V1.1 are connected to PC to send and receive data.

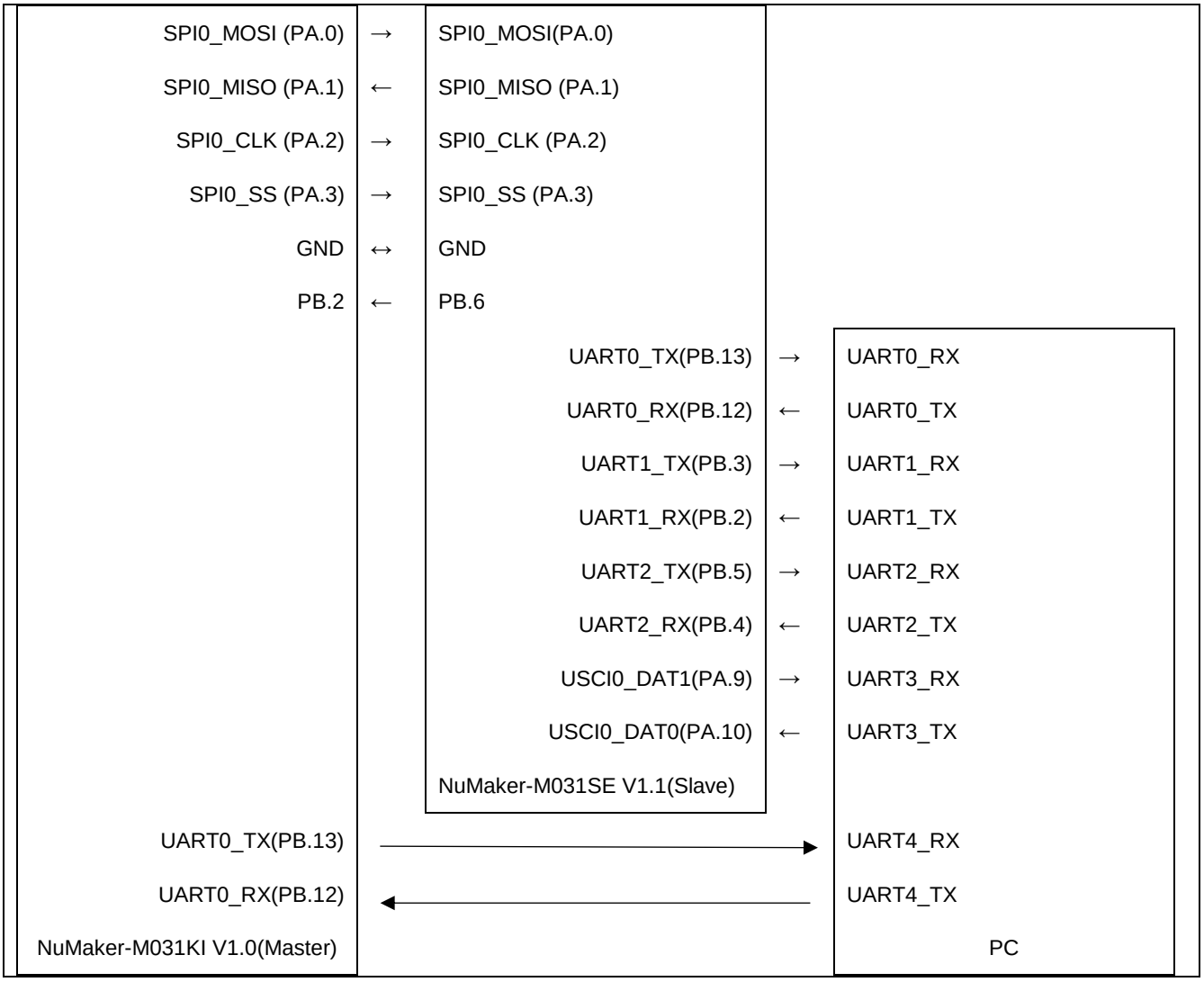

Figure 3-1 Pin Connect

4. Directory Information

The directory structure is shown below.

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

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 *M031_SPI_TO_4_UART.uvmpw*.
2. Set Master as active project, and enter Keil compile mode
   - Set Master as Active project
   - Build
   - Download
3. Set Slave as active project, and enter Keil compile mode
   - Set Master as Active project
   - Build
   - Download
4. Print master debug information
   - Open UART Assistant
   - Baud rate: 115200 bps
5. Open Slave's 4 UARTS
   - Open UART Assistant
   - Baud rate: 115200 bps
6. Run
   - Press reset key on NuMaker-M031SE V1.1, and execute slave code.
   - Press reset key on NuMaker-M031KI V1.0, and execute master code.
7. Data Transfer
   - Send any character in master UART Assistant to start test.
   - Set 512 bytes of data to be sent every 500 ms in Slave's 4 UART Assistant .
   - Then stop the sending of Slave's 4 UARTs.
   - Send any character in master UART Assistant to stop test.

6. Revision History

Date	Revision	Description
2022.10.20	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.