NUC123 SPI Dual I/O Mode

Example Code Introduction for 32-bit NuMicro® Family

Information

Application	This sample code using the SPI dual I/O mode .
BSP Version	NUC123SeriesBSP_CMSIS_V3.00.001
Hardware	NuTiny-EVB-NUC123 V2.0 X 2

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 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 Function Description

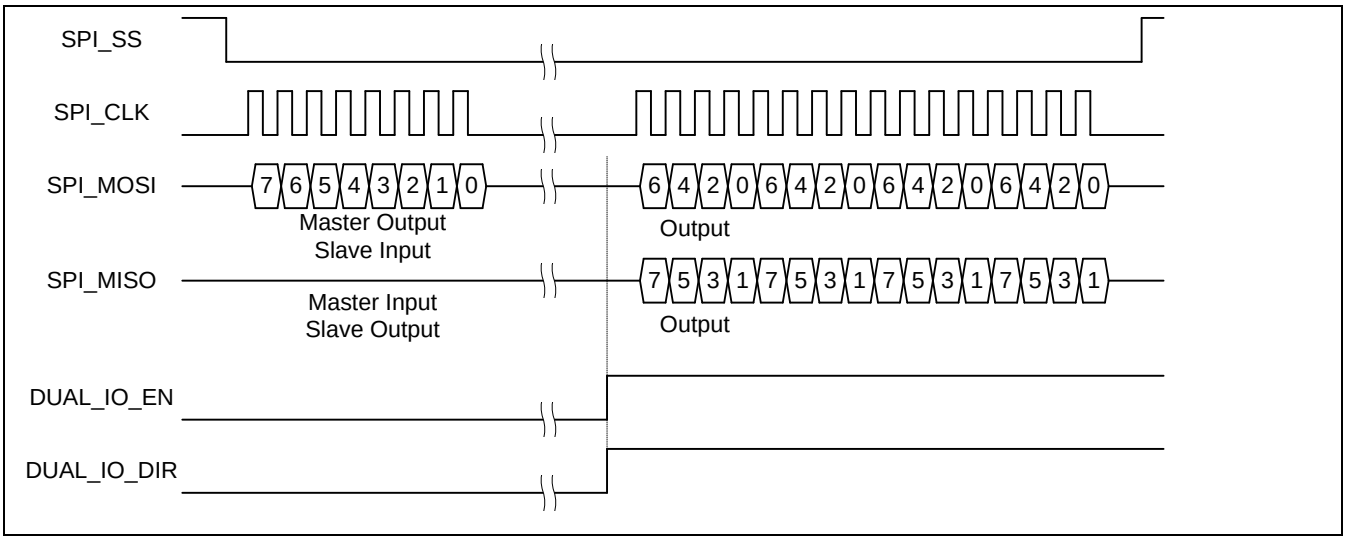
1.1 Introduction

This example code will demo how to use the SPI dual IO function. A multi I/O SPI device is capable of supporting increased bandwidth or throughput from a single device.

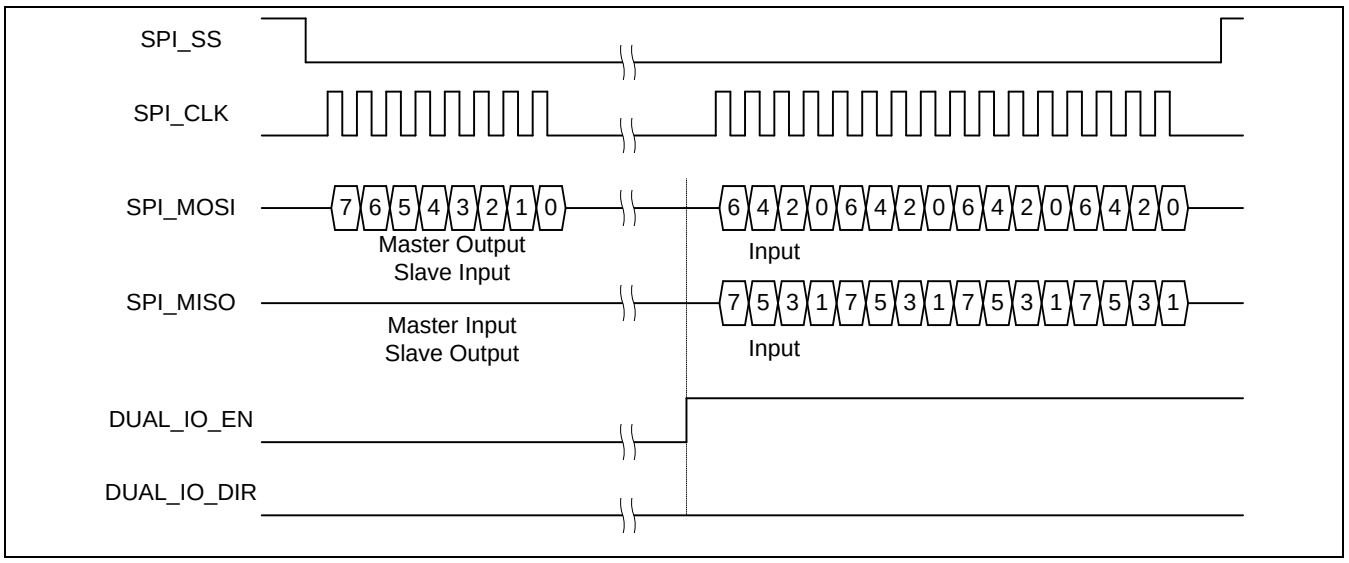
1.2 SPI dual IO operation

The SPI controller also supports Dual I/O transfer when set the DUAL_IO_EN (SPI_CNTRL2[13]) to 1. Many general SPI flashes support Dual I/O transfer. The DUAL_IO_DIR (SPI_CNTRL2[12]) is used to define the direction of the transfer data. When set the DUAL_IO_DIR to 1, the controller will send the data to external device. When the DUAL_IO_DIR is cleared to 0, the controller will read the data from the external device. This function is only available when the transfer bit length is even number.
The Dual I/O mode is not supported when the Slave 3-wire mode or the Byte Reorder function is enabled.

For Dual I/O mode, if both the DUAL_IO_EN (SPI_CNTRL2[13]) and DUAL_IO_DIR (SPI_CNTRL2[12]) are set as 1, the MOSI0 is the even bit data output and the MISO0 will be set as the odd bit data output. If the DUAL_IO_EN (SPI_CNTRL2[13]) is set as 1 and DUAL_IO_DIR (SPI_CNTRL2[12]) is set as 0, both the MISO0 and MOSI0 will be set as data input ports.



Bit Sequence of Dual Output Mode



Bit Sequence of Dual Input Mode

1.3 Demo Result

First, in Dual I/O mode, the Master will transmit 16 bytes of data first. When the Slave receives it, it will use the serial port to print out.

```
Received data:

0:  0x550000

1:  0x550001

2:  0x550002

3:  0x550003

4:  0x550004

5:  0x550005

6:  0x550006

7:  0x550007

8:  0x550008

9:  0x550009

10: 0x55000A

11: 0x55000B

12: 0x55000C

13: 0x55000D

14: 0x55000E

15: 0x55000F
```

Then, the Slave will transfer 16 bytes of data. When the Master receives it, the master will print out the receiver data from slave.

```
Received data:
0:  0xAA0000
1:  0xAA0001
2:  0xAA0002
3:  0xAA0003
4:  0xAA0004
5:  0xAA0005
6:  0xAA0006
7:  0xAA0007
8:  0xAA0008
9:  0xAA0009
10: 0xAA000A
11: 0xAA000B
12: 0xAA000C
13: 0xAA000D
14: 0xAA000E
15: 0xAA000F
```

2 Code Description

SPI Master initial:

```
void void SPI_Init(void)
{
    /*---------------------------------------------------------------------------------*/
    /* Init SPI                                                                        */
    /*---------------------------------------------------------------------------------*/
    /* Configure SPI0 as a master, clock idle low, 32-bit transaction, drive output on
       falling clock edge and latch input on rising edge. */
    /* Set IP clock divider. SPI clock rate = 2 MHz */
    SPI_Open(SPI0, SPI_MASTER, SPI_MODE_0, 32, 2000000);
    /* Enable the automatic hardware slave select function. Select the SPI0_SS0 pin and
    configure as low-active. */
    SPI_EnableAutoSS(SPI0, SPI_SS0, SPI_SS_ACTIVE_LOW);
}
```

SPI Master Main Function:

```
int main(void)
{
    ……
    /* Init SPI */
    SPI_Init();
    /* Enable Dual I/O output mode */
    SPI_ENABLE_DUAL_OUTPUT_MODE(SPI0);
    /* Enable FIFO mode, set TX FIFO threshold, enable TX FIFO threshold interrupt and
    RX FIFO time-out interrupt */
    SPI_EnableFIFO(SPI0, 2, 2);
    SPI_EnableInt(SPI0, SPI_FIFO_TX_INT_MASK | SPI_FIFO_TIMEOUT_INT_MASK);
    g_u32TxDataCount = 0;
    g_u32RxDataCount = 0;
    NVIC_EnableIRQ(SPI0_IRQn);

    /* Wait for transfer done */
    while(g_u32RxDataCount < TEST_COUNT);

    /* Enable Dual I/O input mode */
    SPI_ENABLE_DUAL_INPUT_MODE(SPI0);
    printf("Dual I/O output transfer is done. Before starting the Dual I/O input
```

```
                transfer, make sure the slave device is ready.\n");
        printf("Press any key to start the Dual I/O input transfer.");
        getchar();
        printf("\n");

        g_u32TxDataCount = 0;
        g_u32RxDataCount = 0;
        SPI_EnableInt(SPI0, SPI_FIFO_TX_INT_MASK | SPI_FIFO_TIMEOUT_INT_MASK);

        /* Wait for transfer done */
        while(g_u32RxDataCount < TEST_COUNT);

        /* Print the received data */
        printf("Received data:\n");
        for(u32DataCount = 0; u32DataCount < TEST_COUNT; u32DataCount++)
        {
            printf("%d:\t0x%X\n", u32DataCount, g_au32DestinationData[u32DataCount]);
        }
        /* Disable TX FIFO threshold interrupt and RX FIFO time-out interrupt */
        SPI_DisableInt(SPI0, SPI_FIFO_TX_INT_MASK | SPI_FIFO_TIMEOUT_INT_MASK);
        NVIC_DisableIRQ(SPI0_IRQn);

        printf("\n\nExit SPI driver sample code.\n");

        /* Disable SPI0 peripheral clock */
        SPI_Close(SPI0);
        while(1);
    }
```

SPI Master Interrupt

```
    void SPI0_IRQHandler(void)
    {
        /* Check RX EMPTY flag */
        while(SPI_GET_RX_FIFO_EMPTY_FLAG(SPI0) == 0)
        {
            /* Read RX FIFO */
            g_au32DestinationData[g_u32RxDataCount++] = SPI_READ_RX0(SPI0);
        }
        /* Check TX FULL flag and TX data count */
```

```
        while((SPI_GET_TX_FIFO_FULL_FLAG(SPI0) == 0) && (g_u32TxDataCount < TEST_COUNT))
        {
            /* Write to TX FIFO */
            SPI_WRITE_TX0(SPI0, g_au32SourceData[g_u32TxDataCount++]);
        }
        if(g_u32TxDataCount >= TEST_COUNT)
            SPI_DisableInt(SPI0, SPI_FIFO_TX_INT_MASK); /*Disable TX FIFO threshold*/

        /* Check the RX FIFO time-out interrupt flag */
        if(SPI_GetIntFlag(SPI0, SPI_FIFO_TIMEOUT_INT_MASK))
        {
            /* If RX FIFO is not empty, read RX FIFO. */
            while(SPI_GET_RX_FIFO_EMPTY_FLAG(SPI0) == 0)
                g_au32DestinationData[g_u32RxDataCount++] = SPI_READ_RX0(SPI0);
        }
    }
```

SPI Slave initial:

```
    void void SPI_Init(void)
    {
        /*---------------------------------------------------------------------------------------*/
        /* Init SPI                                                                              */
        /*---------------------------------------------------------------------------------------*/
        /* Configure SPI0 as a slave, clock idle low, 32-bit transaction, drive output on
        falling clock edge and latch input on rising edge. */
        /* Configure SPI0 as a low level active device. SPI peripheral clock rate is equal
        to system clock rate. */
        SPI_Open(SPI0, SPI_MASTER, SPI_MODE_0, 32, 2000000);
    }
```

SPI Slave Main Function:

```
    int main(void)
    {
        ……
        /* Init SPI */
        SPI_Init();
        /* Enable Dual I/O input mode */
        SPI_ENABLE_DUAL_INPUT_MODE(SPI0);
```

```
    /* Enable FIFO mode and set FIFO threshold. */
    SPI_EnableFIFO(SPI0, 2, 2);

    printf("\n\n");
    printf("+-----------------------------------------------------+\n");
    printf("|          SPI Slave Mode Sample Code                 |\n");
    printf("+-----------------------------------------------------+\n");
    printf("\n");
    printf("Configure SPI0 as a slave.\n");
    printf("Bit length of a transaction: 32\n");
    printf("The I/O connection for SPI0:\n");
    printf("    SPI0_SS0 (PC.0)\n    SPI0_CLK (PC.1)\n");
    printf("    SPI0_MISO0 (PC.2)\n    SPI0_MOSI0 (PC.3)\n\n");
    printf("SPI controller will enable Dual I/O mode and FIFO mode.\n");
    printf("In the first stage, the SPI controller will receive %d data from a off-chip
     master device.\n", TEST_COUNT);
    printf("After the transfer is done, the %d received data will be printed out.\n",
    TEST_COUNT);
    printf("In the second stage, the SPI controller will transfer %d data to the off-
    chip master device.\n", TEST_COUNT);

    for(u32TxDataCount = 0; u32TxDataCount < TEST_COUNT; u32TxDataCount++)
    {
        /* Write the initial value to source buffer */
        g_au32SourceData[u32TxDataCount] = 0x00AA0000 + u32TxDataCount;
        /* Clear destination buffer */
        g_au32DestinationData[u32TxDataCount] = 0;
    }

    u32RxDataCount = 0;
    printf("< Stage 1 >\n");
    printf("Press any key to start the Dual I/O input transfer if the master device
    configuration is ready.");
    getchar();
    printf("\n");

    /* Access RX FIFO */
    while(u32RxDataCount < TEST_COUNT)
    {
        /* Check RX EMPTY flag */
        if(SPI_GET_RX_FIFO_EMPTY_FLAG(SPI0) == 0)
```

```
            {
             /* Read RX FIFO */
             g_au32DestinationData[u32RxDataCount++] = SPI_READ_RX0(SPI0);
            }
    }

    /* Print the received data */
    printf("Received data:\n");
    for(u32RxDataCount = 0; u32RxDataCount < TEST_COUNT; u32RxDataCount++)
    {
        printf("%d:\t0x%X\n", u32RxDataCount, g_au32DestinationData[u32RxDataCount]);
    }

    /* Enable Dual I/O output mode */
    SPI_ENABLE_DUAL_OUTPUT_MODE(SPI0);
    u32TxDataCount = 0;
    u32RxDataCount = 0;
    printf("< Stage 2 >\n");
    printf("Enable Dual I/O output mode. Slave is ready to transfer.\n");

    /* Access TX and RX FIFO */
    while(u32RxDataCount < TEST_COUNT)
    {
        /* Check TX FULL flag and TX data count */
        if((SPI_GET_TX_FIFO_FULL_FLAG(SPI0) == 0) && (u32TxDataCount < TEST_COUNT))
            SPI_WRITE_TX0(SPI0, g_au32SourceData[u32TxDataCount++]);
           /* Write to TX FIFO */
        /* Check RX EMPTY flag */
        if(SPI_GET_RX_FIFO_EMPTY_FLAG(SPI0) == 0)
            g_au32DestinationData[u32RxDataCount++] = SPI_READ_RX0(SPI0);
           /* Read RX FIFO */
    }

    printf("\n\nExit SPI driver sample code.\n");

    /* Disable SPI0 peripheral clock */
    SPI_Close(SPI0);
    while(1);
}
```

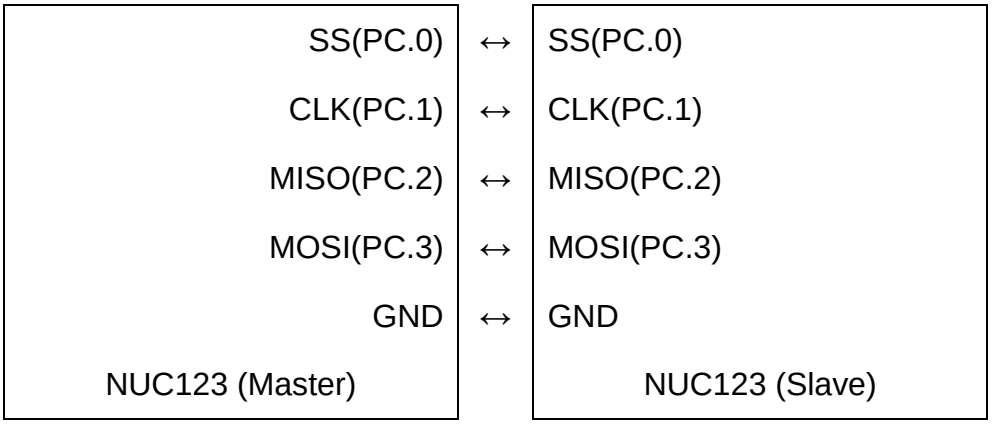
3 Software and Hardware Environment

Software Environment

- BSP version
  - NUC123 Series BSP CMSIS v3.00.003
- IDE version
  - Keil uVersion 5.15

Hardware Environment

- Circuit components
  - NuTiny-EVB-NUC123 V2.0
- Diagram



4 Directory Information

- EC_NUC123_SPI_DUALIO_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
  - SamlpeCode
    - ExampleCode — Source file of example code

5 How to Execute Example Code

1. Browsing into sample code folder by Directory Information (section 4) and double click SPI_MasterDualIOMode.uvproj and SPI_SlaveDualIOMode.uvproj.
2. Enter Keil compile mode
   a. Build
   b. Download
   c. Start/Stop debug session
3. Enter debug mode
   a. Run

6 Revision History

Date	Revision	Description
June. 17, 2019	1.00	1. Initially issued.

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.