Application Note

32-bit Cortex™-M0 MCU
NuMicro® Family

How to use SPI 2-Bit Mode?

Application Note

Table of Contents-

1 INTRODUCTION

This document describes how to use the 2-bit mode function of SPI controller in NUC100 series chips.

1.1 Scope

In this application, the SPI controller will be configured as a SPI master in 2-bit mode, and use two SPI flash devices (W25Q16BV, the S*piFlash* of Winbond) as slave devices. The SPI master uses a serial clock (SPICLKx), a slave select (SPISSx0) and 2 channels data in/out pins (MISOx0/1, MOSIx0/1) to control these two SPI flash devices.

1.2 Features

- Supports master and slave mode.
- Supports 1-bit/2-bit data in/out during a serial clock cycle.
- Configurable bit length (1~32 bits) of a word and configurable word numbers (1~2 words) during a transfer.
- MSB or LSB first transfer.
- Two slave select signals in master mode.
- Supports word and byte suspend.
- Supports transmit and receive DMA.

1.3 Structure

1.3.1 SPI Block Diagram

The SPI controller block diagram is shown as below. The SPICLKx is the serial clock from the SPI master. These SPISSx0 and SPISSx1 two pins are the slave select output pins in master mode, but only SPISSx0 is the slave select input pin in slave mode. These MOSIx0 and MISOx0 pins combine to form the first channel data in/out pins, and the MOSIx1 and MISOx1 pins combine to form the second channel data in/out pins.

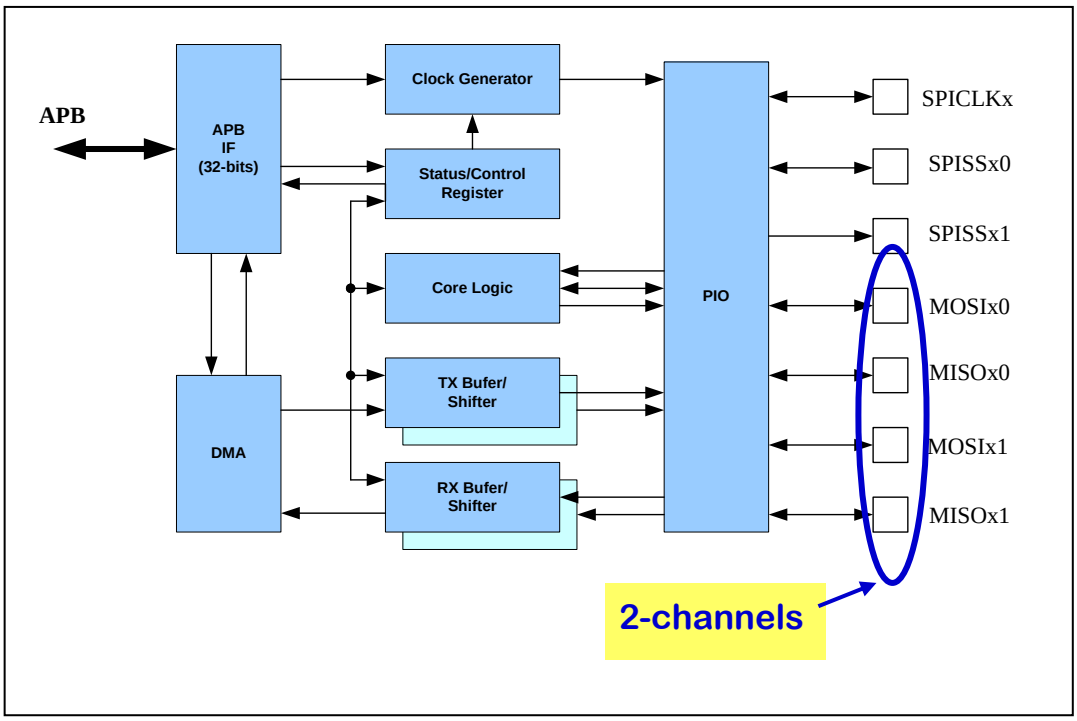


Figure 1 SPI Block Diagram

1.3.2 SPI Application Block Diagram

The application block diagram is shown as below. We configure the SPI controller 2 (SPI2) of NUC100 series as a SPI master in 2-bit mode. The SPICLK2 is serial clock output from the SPI2 master to control the data in/out with two SPI flash devices. The SPISS20 is the slave select output from the SPI2 master to enable/disable these two SPI flash devices. The first channel, combined by MISO20 and MOSI20 pins, is connected to SPI flash device 0. The second channel, combined by MISO21 and MOSI21 pins, is connected to SPI flash device 1.

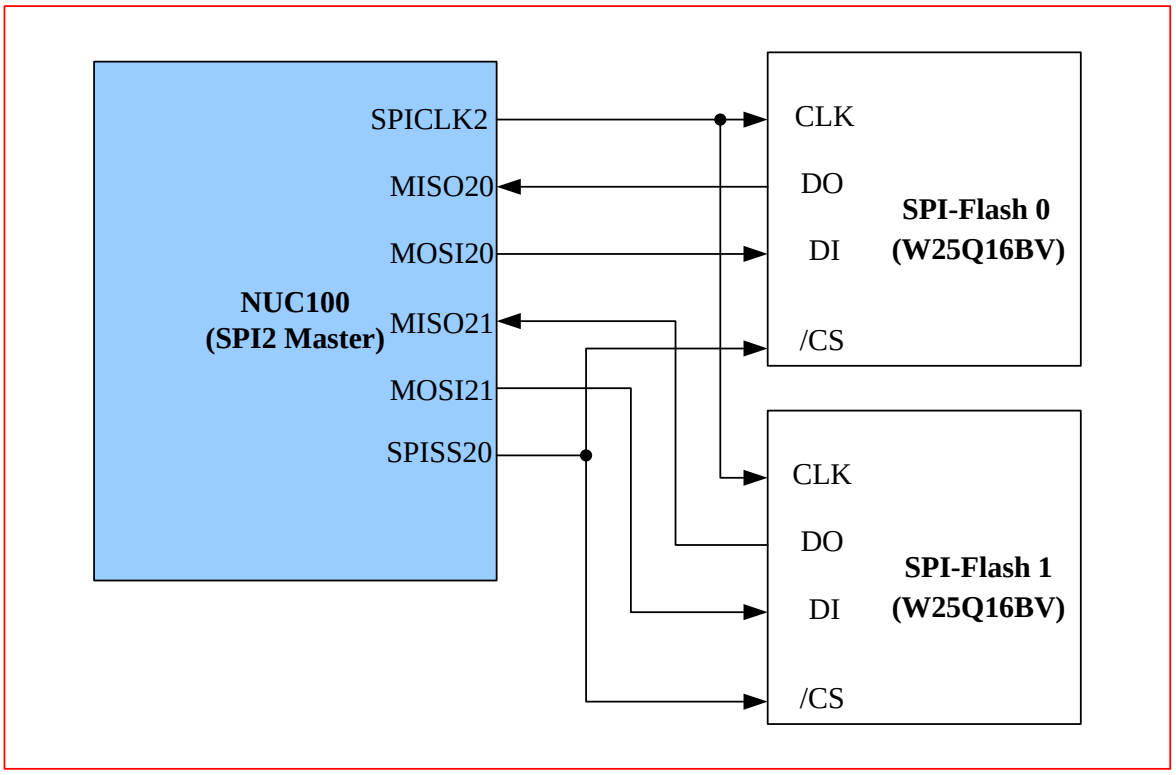


Figure 2 SPI Application Block Diagram in 2-Bit Transfer Mode

1.3.3 Tx/Rx Registers and I/O Pins in 2-Bit Transfer Mode

In 2-bit transfer mode, it only supports up to 32 bit length can be transmitted out and received in during a transfer. These MOSIx0 and MISOx0 pins are to combine to form the channel 0 to transmit data from the SPI_Tx0 register and receive data into the SPI_Rx0 register. These MOSIx1 and MISOx1 pins are to combine to form channel 1 to transmit data from SPI_Tx1 register and receive data into the SPI_Rx1 register. The diagram about the relationship of registers and IO is shown as below.

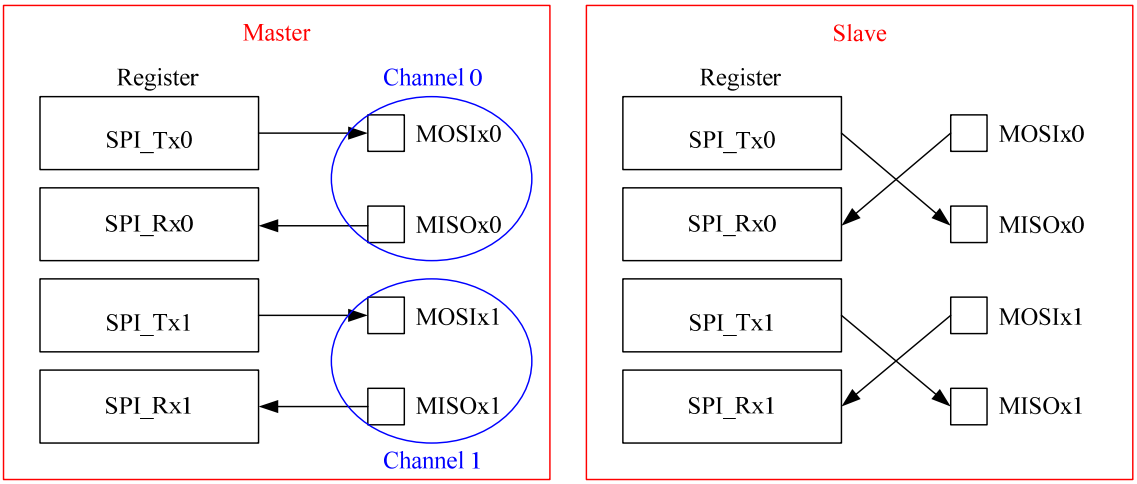


Figure 3 Tx/Rx Registers and I/O Diagram in 2-Bit Transfer Mode

1.3.4 Timing Diagram in 2-Bit Transfer Mode

The timing diagram for SPI 2-bit transfer mode is shown as below.

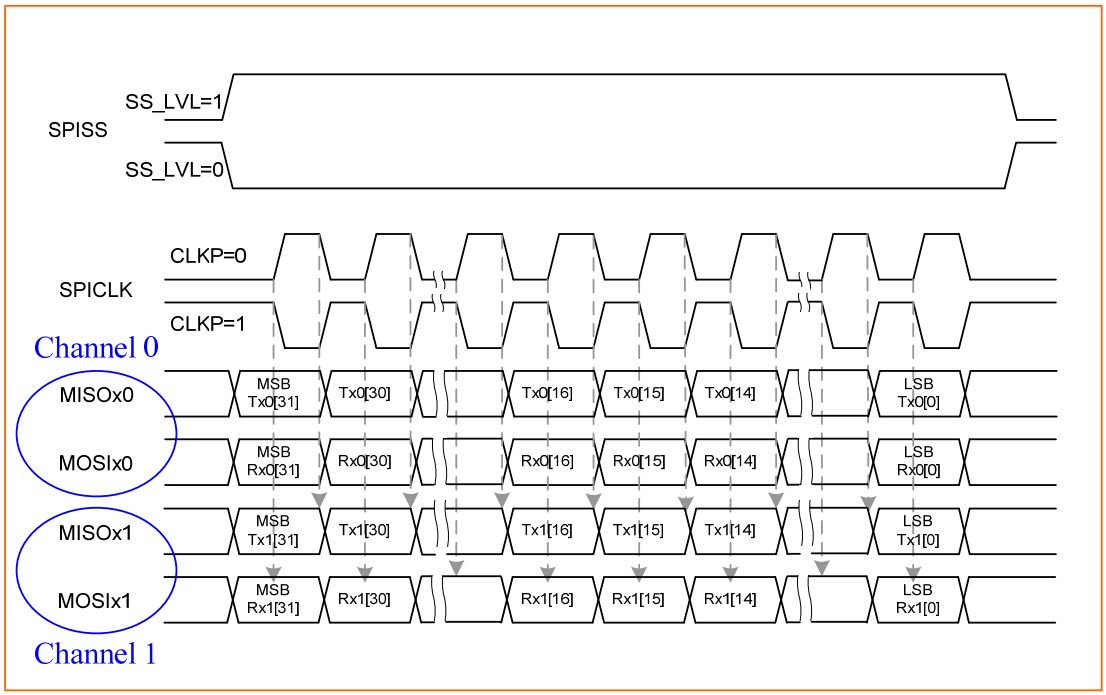


Figure 4 Timing Diagram in 2-Bit Transfer Mode

2 CODE SECTION

2.1 Main Function

In the main function, it do the MID and DID check first for these two SPI flash, erase these two flash, program 0x00~0xff data to the first page of flash 0 and program 0xff~0x00 data to the first page of flash 1, and finally verify the data from these two flash.

```
#define TEST_NUMBER 1
int main(void)
{
        uint32_t u32ByteCount, u32FlashAddress, u32PageNumber;
        uint8_t DataBuffer0[256];
        uint8_t DataBuffer1[256];

        /* Unlock the protected registers */
        UNLOCKREG();

        /* Enable the 12MHz oscillator oscillation */
        DrvSYS_SetOscCtrl(E_SYS_XTL12M, 1);

        /* HCLK clock source. 0: external 12MHz; 4:internal 22MHz RC oscillator */
        DrvSYS_SetHCLKSource(0);

        LOCKREG();

        /* HCLK clock frequency = HCLK clock source / (HCLK_N + 1) */
        DrvSYS_SetClockDivider(E_SYS_HCLK_DIV, 0);

        /* Configure SPI2 as a master, Type1 waveform, 32-bit transaction */
        DrvSPI_Open(eDRVSPI_PORT2, eDRVSPI_MASTER, eDRVSPI_TYPE1, 32);

        /* MSB first */
        DrvSPI_SetEndian(eDRVSPI_PORT2, eDRVSPI_MSB_FIRST);

        /* Disable the automatic slave select function of SS0. */
        DrvSPI_DisableAutoCS(eDRVSPI_PORT2);
```

```
/* Set the active level of slave select. */
DrvSPI_SetSlaveSelectActiveLevel(eDRVSPI_PORT2, eDRVSPI_ACTIVE_LOW_FALLING);

/* Configure SPI2 as 2-bit transfer mode */
DrvSPI_Set2BitSerialDataIOMode(eDRVSPI_PORT2, TRUE);

/* SPI clock rate 1MHz */
DrvSPI_SetClock(eDRVSPI_PORT2, 1000000, 0);

/* Check MID & DID */
SpiFlashx2_ReadMidDid();

/* Erase SPI flash */
SpiFlashx2_ChipErase();

/* Wait ready */
SpiFlashx2_WaitReady();

/* initial source data */
for(u32ByteCount=0; u32ByteCount<256; u32ByteCount++)
{
        DataBuffer0[u32ByteCount] = u32ByteCount;
}

/* Program SPI flash */
u32FlashAddress = 0;
for(u32PageNumber=0; u32PageNumber<TEST_NUMBER; u32PageNumber++)
{
        /* page program */
        SpiFlashx2_PageProgram(DataBuffer0, u32FlashAddress, 256);
        SpiFlashx2_WaitReady();
        u32FlashAddress += 0x100;
}
```

```
        /* clear data buffer */
        for(u32ByteCount=0; u32ByteCount<256; u32ByteCount++)
        {
                DataBuffer0[u32ByteCount] = 0;
                DataBuffer1[u32ByteCount] = 0;
        }

        /* Verify SPI flash */
        u32FlashAddress = 0;
        for(u32PageNumber=0; u32PageNumber<TEST_NUMBER; u32PageNumber++)
        {
                SpiFlashx2_ReadData(DataBuffer0, DataBuffer1, u32FlashAddress, 256);
                u32FlashAddress += 0x100;

                for(u32ByteCount=0; u32ByteCount<256; u32ByteCount++)
                {
                        if(DataBuffer0[u32ByteCount]!=u32ByteCount)
                                while(1); /* Verify Error! */

                        if((uint8_t)DataBuffer1[u32ByteCount]!=(uint8_t)~DataBuffer0[u32ByteCount])
                                 while(1); /* Verify Error! */
                }

                /* clear data buffer */
                for(u32ByteCount=0; u32ByteCount<256; u32ByteCount++)
                {
                        DataBuffer0[u32ByteCount] = 0;
                        DataBuffer1[u32ByteCount] = 0;
                }
        }

        DrvSPI_Close(eDRVSPI_PORT2);

        return 1;
}
```

2.2 Check MID and DID of SPI Flash

The SPI master sends 0x90 command data to these two SPI flash to read back the individual manufacturer ID (MID) and Device ID (DID), check the each read back data whether is equal to 0xEF14. If these ID are correct, the interface between SPI master (NUC100) and two slave flash devices is connected successfully.

```
// ***************************************
// For W25Q16BV, Manufacturer ID: 0xEF; Device ID: 0x14
void SpiFlashx2_ReadMidDid(void)
{
    uint32_t au32SourceData[2];
    uint32_t au32DestinationData[2];

    /* configure transaction length as 8 bits */
    DrvSPI_SetBitLength(eDRVSPI_PORT2, 8);

    /* /CS: active */
    DrvSPI_SetCS(eDRVSPI_PORT2, eDRVSPI_SS0);

    /* send Command: 0x90, Read Manufacturer/Device ID */
    au32SourceData[0] = 0x90;
    au32SourceData[1] = 0x90;
    DrvSPI_BurstWrite(eDRVSPI_PORT2, &au32SourceData[0]);

    /* wait */
    while (DrvSPI_Busy(eDRVSPI_PORT2)) {}

    /* configure transaction length as 24 bits */
    DrvSPI_SetBitLength(eDRVSPI_PORT2, 24);

    /* send 24-bit '0', dummy */
    au32SourceData[0] = 0x0;
    au32SourceData[1] = 0x0;
   DrvSPI_BurstWrite(eDRVSPI_PORT2, &au32SourceData[0]);

    /* wait */
    while (DrvSPI_Busy(eDRVSPI_PORT2)) {}
```

```
        /* configure transaction length as 16 bits */
        DrvSPI_SetBitLength(eDRVSPI_PORT2, 16);

        /* receive */
        au32SourceData[0] = 0x0;
        au32SourceData[1] = 0x0;
        DrvSPI_BurstWrite(eDRVSPI_PORT2, &au32SourceData[0]);

        /* wait */
        while (DrvSPI_Busy(eDRVSPI_PORT2)) {}

        /* /CS: de-active */
        DrvSPI_ClrCS(eDRVSPI_PORT2, eDRVSPI_SS0);

        /* dump Rx register */
        DrvSPI_DumpRxRegister(eDRVSPI_PORT2, &au32DestinationData[0], 2);

        /* check the MID and DID */
        if ((au32DestinationData[0] & au32DestinationData[1] & 0xffff) != 0xEF14)
                while (1);  /* MID & DID Error */

}
```

2.3 Erase SPI Flash

The SPI master sends 0xC7 command data to these two SPI flash to erase these two flash chips.

```
void SpiFlashx2_ChipErase(void)
{
    uint32_t au32SourceData[2];

    /* configure transaction length as 8 bits */
    DrvSPI_SetBitLength(eDRVSPI_PORT2, 8);

    /* /CS: active */
    DrvSPI_SetCS(eDRVSPI_PORT2, eDRVSPI_SS0);

    /* send Command: 0x06, Write enable */
    au32SourceData[0] = 0x06;
    au32SourceData[1] = 0x06;
    DrvSPI_BurstWrite(eDRVSPI_PORT2, &au32SourceData[0]);

    /* wait */
    while (DrvSPI_Busy(eDRVSPI_PORT2)) {}

    /* /CS: de-active */
    DrvSPI_ClrCS(eDRVSPI_PORT2, eDRVSPI_SS0);

    /* /CS: active */
    DrvSPI_SetCS(eDRVSPI_PORT2, eDRVSPI_SS0);

    /* send Command: 0xC7, Chip Erase */
    au32SourceData[0] = 0xc7;
    au32SourceData[1] = 0xc7;
    DrvSPI_BurstWrite(eDRVSPI_PORT2, &au32SourceData[0]);

    /* wait */
    while (DrvSPI_Busy(eDRVSPI_PORT2)) {}

    /* /CS: de-active */
```

```
        DrvSPI_ClrCS(eDRVSPI_PORT2, eDRVSPI_SS0);
}
```

2.4 Read the Status Register 1 of SPI Flash

The SPI master sends 0x05 command data to these two SPI flash to read back the individual Status Register 1 of these two flash devices.

```
uint32_t SpiFlash2_ReadStatusReg1(void)
{
        uint32_t au32SourceData[2];
        uint32_t au32DestinationData[2];

        /* configure transaction length as 16 bits */
        DrvSPI_SetBitLength(eDRVSPI_PORT2, 16);

        /* /CS: active */
        DrvSPI_SetCS(eDRVSPI_PORT2, eDRVSPI_SS0);

        /* send Command: 0x05, Read status register 1 */
        au32SourceData[0] = 0x0500;
        au32SourceData[1] = 0x0500;
        DrvSPI_BurstWrite(eDRVSPI_PORT2, &au32SourceData[0]);

        /* wait */
        while (DrvSPI_Busy(eDRVSPI_PORT2)) {}

        /* /CS: de-active */
        DrvSPI_ClrCS(eDRVSPI_PORT2, eDRVSPI_SS0);

        /* dump Rx register */
        DrvSPI_DumpRxRegister(eDRVSPI_PORT2, &au32DestinationData[0], 2);

        return ((au32DestinationData[0] | au32DestinationData[1]) & 0xFF);

}
```

2.5 Check the BUSY Bit of SPI Flash

The SPI master reads back the status register 1 from these two SPI flash and checks these two flash whether are busy or not.

```
void SpiFlashx2_WaitReady(void)
{
        uint32_t ReturnValue;

        do{
                ReturnValue = SpiFlash2_ReadStatusReg1();
                ReturnValue = ReturnValue & 1;
        }while(ReturnValue!=0); // check the BUSY bit

}
```

2.6 Program SPI Flash

The SPI master sends 0x02 command data to these two SPI flash to program data to these two flash chips.

```
void SpiFlashx2_PageProgram(uint8_t *DataBuffer, uint32_t StartAddress, uint32_t ByteCount)
{
    uint32_t au32SourceData[2];
    uint32_t Counter;

    /* configure transaction length as 8 bits */
    DrvSPI_SetBitLength(eDRVSPI_PORT2, 8);

    /* /CS: active */
    DrvSPI_SetCS(eDRVSPI_PORT2, eDRVSPI_SS0);

    /* send Command: 0x06, Write enable */
    au32SourceData[0] = 0x06;
    au32SourceData[1] = 0x06;
    DrvSPI_BurstWrite(eDRVSPI_PORT2, &au32SourceData[0]);

    /* wait */
    while (DrvSPI_Busy(eDRVSPI_PORT2)) {}

    /* /CS: de-active */
    DrvSPI_ClrCS(eDRVSPI_PORT2, eDRVSPI_SS0);

    /* /CS: active */
    DrvSPI_SetCS(eDRVSPI_PORT2, eDRVSPI_SS0);

    /* send Command: 0x02, Page program */
    au32SourceData[0] = 0x02;
    au32SourceData[1] = 0x02;
    DrvSPI_BurstWrite(eDRVSPI_PORT2, &au32SourceData[0]);

    /* wait */
    while (DrvSPI_Busy(eDRVSPI_PORT2)) {}
```

```
        /* configure transaction length as 24 bits */
        DrvSPI_SetBitLength(eDRVSPI_PORT2, 24);

        /* send 24-bit start address */
        au32SourceData[0] = StartAddress;
        au32SourceData[1] = StartAddress;
        DrvSPI_BurstWrite(eDRVSPI_PORT2, &au32SourceData[0]);

        /* wait */
        while (DrvSPI_Busy(eDRVSPI_PORT2)) {}

        /* configure transaction length as 8 bits */
        DrvSPI_SetBitLength(eDRVSPI_PORT2, 8);

        for(Counter=0; Counter<ByteCount; Counter++)
        {
                /* send data to program */
                 au32SourceData[0] = DataBuffer[Counter];
                 au32SourceData[1] = ~au32SourceData[0];
                DrvSPI_BurstWrite(eDRVSPI_PORT2, &au32SourceData[0]);

                 /* wait */
                 while (DrvSPI_Busy(eDRVSPI_PORT2)) {}
        }

        /* /CS: de-active */
        DrvSPI_ClrCS(eDRVSPI_PORT2, eDRVSPI_SS0);

}
```

2.7 Read Data from SPI Flash

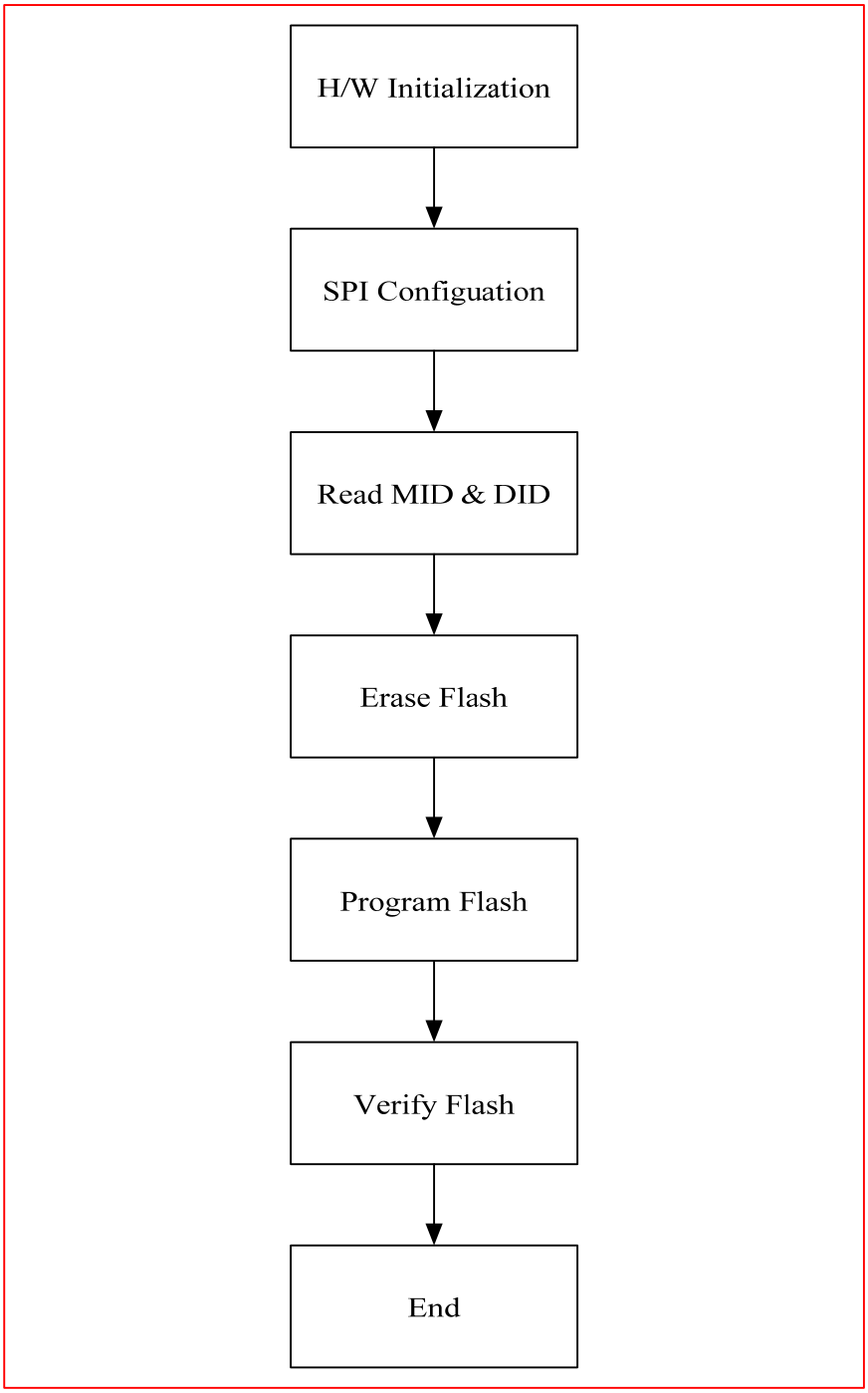
The SPI master sends 0x03 command data to these two SPI flash to read data from these two flash chips.

```
void SpiFlashx2_ReadData(uint8_t *DataBuffer0, uint8_t *DataBuffer1, uint32_t StartAddress, uint32_t ByteCount)
{
    uint32_t au32SourceData[2];
    uint32_t au32DestinationData[2];
    uint32_t Counter;

    /* configure transaction length as 8 bits */
    DrvSPI_SetBitLength(eDRVSPI_PORT2, 8);

    /* /CS: active */
    DrvSPI_SetCS(eDRVSPI_PORT2, eDRVSPI_SS0);

    /* send Command: 0x03, Read data */
    au32SourceData[0] = 0x03;
    au32SourceData[1] = 0x03;
    DrvSPI_BurstWrite(eDRVSPI_PORT2, &au32SourceData[0]);

    /* wait */
    while (DrvSPI_Busy(eDRVSPI_PORT2)) {}

    /* configure transaction length as 24 bits */
    DrvSPI_SetBitLength(eDRVSPI_PORT2, 24);

    /* send 24-bit start address */
    au32SourceData[0] = StartAddress;
    au32SourceData[1] = StartAddress;
    DrvSPI_BurstWrite(eDRVSPI_PORT2, &au32SourceData[0]);

    /* wait */
    while (DrvSPI_Busy(eDRVSPI_PORT2)) {}
```

```
        /* configure transaction length as 8 bits */
        DrvSPI_SetBitLength(eDRVSPI_PORT2, 8);

        for(Counter=0; Counter<ByteCount; Counter++)
        {
                /* receive */
                au32SourceData[0] = 0x0;
                au32SourceData[1] = 0x0;
                DrvSPI_BurstWrite(eDRVSPI_PORT2, &au32SourceData[0]);

                /* wait */
                while (DrvSPI_Busy(eDRVSPI_PORT2)) {}

                /* dump Rx register */
                DrvSPI_DumpRxRegister(eDRVSPI_PORT2, &au32DestinationData[0], 2);
                DataBuffer0[Counter] = (uint8_t) au32DestinationData[0];
                DataBuffer1[Counter] = (uint8_t) au32DestinationData[1];
        }

        /* /CS: de-active */
        DrvSPI_ClrCS(eDRVSPI_PORT2, eDRVSPI_SS0);

}
```

3 CALLING SEQUENCE

3.1 SPI Controller and SPI Flash Integration



3.2 NUC100 Settings

1. Hardware initialization: Enable the 12MHz oscillator, set HCLK clock source and divider.
2. Configure the SPI controller 2 as a master, 2-bit transfer mode, data transmitted at negative edge, data received at positive edge, disable auto slave select output, slave select is low active and the bit rate is 1MHz.
3. Read back the MID and DID to check whether the interface is connected successfully.
4. Erase all these two SPI flash chips.
5. Program the first page of these two SPI flash chips with the complementary data.
6. Read back the data from these two SPI flash chips to verify the complementary data.
7. Close the SPI Controller 2.

3.3 API Usage Reference

1. SPI Driver Reference Guide.doc

4 EXECUTION ENVIRONMENT SETUP AND RESULT

4.1 Test Smpl_SPI_Flashx2

The SPI sample code using 2-bit transfer mode to control two SPI flash chips, Smpl_SPI_Flashx2, could be built by Keil MDK tool and download to NUC100 series Learning Board through ICE. Then user is able to execute the code in ICE environment or reset the Learning Board to execute the code which had been programmed in on-chip Program Flash.

4.2 Result

Erase these two flash chips, and program 0x00~0xff data to the first page of flash 0 and program 0xff~0x00 data to the first page of flash 1.

5 REVISION HISTORY

REV.	DATE	DESCRIPTION
1.00	May 12, 2010	1. Initially issued.

Important Notice

Nuvoton products are not designed, intended, authorized or warranted for use as components in systems or equipment intended for surgical implantation, atomic energy control instruments, airplane or spaceship instruments, transportation instruments, traffic signal instruments, combustion control instruments, or for other applications intended to support or sustain life. Further more, Nuvoton products are not intended for applications wherein failure of Nuvoton products could result or lead to a situation wherein personal injury, death or severe property or environmental damage could occur.

Nuvoton customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify Nuvoton for any damages resulting from such improper use or sales.

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.