Accessing SPI Flash with M251 PDMA

Example Code Introduction for 32-bit NuMicro® Family

Information

Application	This sample code uses SPI and PDMA to access an external SPI Flash W25Q32.
BSP Version	M251/M252/M254/M256/M258 Series CMSIS BSP V3.02.002
Hardware	NuMaker-M252SD V1.1, SPI Flash W25Q32

The information described in this document is the exclusive intellectual property of Nuvoton Technology Corporation and shall not be reproduced without permission from Nuvoton.

Nuvoton is providing this document only for reference purposes of NuMicro microcontroller and microprocessor based system design. Nuvoton assumes no responsibility for errors or omissions.

All data and specifications are subject to change without notice.

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

1 Overview

This sample code uses SPI and PDMA to access an external SPI Flash W25Q32. First, user reads SPI Flash MID and DID information through SPI protocol. Then, erase SPI Flash. After erasing is done, repeatedly write data to SPI Flash and read back data for verification for 10 times. The data writing and reading to/from SPI Flash is also processed through PDMA.

1.1 Demo Result

The demo result is shown below.

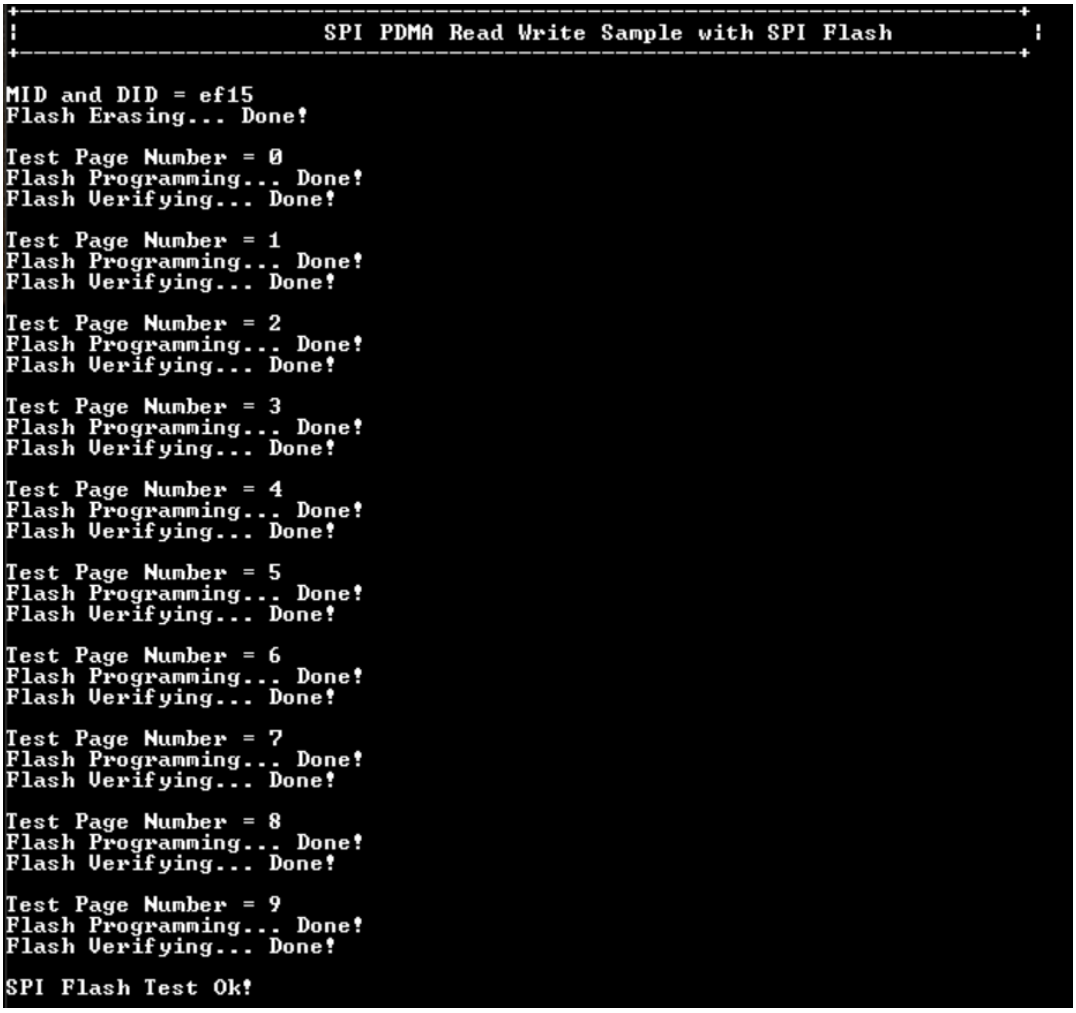

Figure 1-1 Demo Result

2 Code Description

In ***main()*** of ***main.c***, first call ***SYS_Init()*** to do system initialization.

```
/* Main */
int main(void)
{
    unsigned int u32ByteCount;
    unsigned int u32PageNumber;
    unsigned int u32ProgramFlashAddress = 0;
    unsigned int u32VerifyFlashAddress = 0;
    unsigned int MidDid;

    /* Unlock protected registers */
    SYS_UnlockReg();

    /* Init System, IP clock and multi-function I/O. */
    SYS_Init();
    …
```

In ***SYS_Init()***, settings related to SPI and PDMA include:

- Set clock source of SPI module 0: Set PCLK1 as the clock source of SPI0.
- Enable clock of SPI module 0.
- Enable clock of PDMA module.
- Reset PDMA module.

```
void SYS_Init(void)
{
    /*---------------------------------------------------------------------------------*
    /* Init System Clock                                                               */
    /*---------------------------------------------------------------------------------*/
    /* Enable HIRC clock */
    CLK_EnableXtalRC(CLK_PWRCTL_HIRCEN_Msk);

    /* Waiting for HIRC clock ready */
    CLK_WaitClockReady(CLK_STATUS_HIRCSTB_Msk);

    /* Select HCLK clock source as HIRC and and HCLK clock divider as 1 */
    CLK_SetHCLK(CLK_CLKSEL0_HCLKSEL_HIRC, CLK_CLKDIV0_HCLK(1));

    /* Select UART module clock source as HIRC and UART module clock divider as 1 */
    CLK_SetModuleClock(UART0_MODULE, CLK_CLKSEL1_UART0SEL_HIRC, CLK_CLKDIV0_UART0(1));

    /* Select PCLK0 as the clock source of SPI0 */
    CLK_SetModuleClock(SPI0_MODULE, CLK_CLKSEL2_SPI0SEL_PCLK1, MODULE_NoMsk);

    /* Enable UART peripheral clock */
    CLK_EnableModuleClock(UART0_MODULE);

    /* Enable SPI0 peripheral clock */
    CLK_EnableModuleClock(SPI0_MODULE);

    /* Enable PDMA clock */
    CLK_EnableModuleClock(PDMA_MODULE);

    /* Reset PDMA module */
    SYS_ResetModule(PDMA_RST);
    …
```

Besides, in ***SYS_Init()***, SPI related pins are also defined.

- PA0: SPI0, MOSI
- PA1: SPI0, MISO
- PA2: SPI0, CLK
- PA3: SPI0, SS

The Schmitt Trigger is also enabled on SPI0 clock pin (PA2).

```
    /* Setup SPI0 multi-function pins */
    SYS->GPA_MFPL &= ~(SYS_GPA_MFPL_PA0MFP_Msk | SYS_GPA_MFPL_PA1MFP_Msk |
                       SYS_GPA_MFPL_PA2MFP_Msk | SYS_GPA_MFPL_PA3MFP_Msk);

    SYS->GPA_MFPL = SYS_GPA_MFPL_PA0MFP_SPI0_MOSI | SYS_GPA_MFPL_PA1MFP_SPI0_MISO |
                    SYS_GPA_MFPL_PA2MFP_SPI0_CLK | SYS_GPA_MFPL_PA3MFP_SPI0_SS ;

    /* Enable SPI0 clock pin (PA2) schmitt trigger */
    PA->SMTEN |= GPIO_SMTEN_SMTEN2_Msk;
    …
```

Back to ***main()***, after ***SYS_Init()***, ***UART_Open()*** is called in order to output log messages for this demo. SPI related configurations are done by calling ***SPI_Open()***. In this demo, the SPI is configured as a master, MSB first, 8-bit transaction, SPI Mode-0 timing, and 12 MHz clock. After configuration, the SPI module is ready to transfer.

Since MCU can be the SPI master and SPI Flash can be the slave, SPI auto SS function is disabled. The SS signal can be controlled manually by SPI master. The SS signal is set to HIGH, which means that the SPI CS pin is manual mode.

```
    /* Init UART to 115200-8n1 for print message */
    UART_Open(UART0, 115200);

    /* Configure SPI_FLASH_PORT as a master, MSB first, 8-bit transaction, SPI Mode-0
    timing, clock is 12MHz */
    SPI_Open(SPI_FLASH_PORT, SPI_MASTER, SPI_MODE_0, 8, 12000000);

    /* Disable auto SS function, control SS signal manually. */
    SPI_DisableAutoSS(SPI_FLASH_PORT);
    SPI_SET_SS_HIGH(SPI_FLASH_PORT);
```

Note that all functions used to access SPI Flash in this demo include:

- ***SpiFlash_ReadMidDid()***
- ***SpiFlash_ChipErase()***
- ***SpiFlash_WaitReady()***
- ***SpiFlash_PDMA_PageProgram()***
- ***SpiFlash_PDMA_ReadData()***

The first step to interact with a SPI Flash is to read Manufacturer and Device ID from SPI Flash. The information is read by calling ***SpiFlash_ReadMidDid()***. After reading MID and DID, erase SPI Flash by ***SpiFlash_ChipErase()***. After erasing is triggered, call ***Spi_Flash_ _WaitReady()*** to check the erasing status. After the SPI Flash is erased, log message "Flash Erasing… Done!" will be shown. SPI Flash is ready to access.

```
    printf("\n+------------------------------------------------------------------------+\n");
    printf("|                      SPI PDMA Read Write Sample with SPI Flash          |\n");
    printf("+------------------------------------------------------------------------+\n");
    /* Read MID & DID */
    MidDid = SpiFlash_ReadMidDid();
    printf("\nMID and DID = %x", MidDid);

    /* Erase SPI Flash */
    SpiFlash_ChipErase();
    printf("\nFlash Erasing... ");

    /* Wait ready */
    SpiFlash_WaitReady();
    printf("Done!");…
```

After SPI Flash is ready to access, this demo will repeatedly write 256 bytes' data into SPI Flash and then read the data back from SPI Flash. The verification is done by comparing the data between. If the data verification is failed, it will show log message "*Flash Verifying... SPI Flash R/W Fail!*" and stop the program in a ***while(1)*** loop. Otherwise, it will show log message "*Flash Verifying... Done!*".

The data writing/reading and verification will be performed for 10 times in this demo. If all the verifications are passed, the final log message "*SPI Flash Test OK!*" will be shown to indicate the success of this testing.

Prepare source data to be written into SPI Flash.

```
    /* Fill the Source Data and clear Destination Data Buffer */
    for (u32ByteCount = 0; u32ByteCount < 256; u32ByteCount++)
    {
        SrcArray[u32ByteCount] = u32ByteCount;
        DestArray[u32ByteCount] = 0;
    }
```

In a for loop, repeatedly write data, read back data, and do comparison. Writing data to SPI Flash is done by calling ***SpiFlash_PDMA_PageProgram()*** and reading data from SPI Flash is done by ***SpiFlash_PDMA_ReadData()***.

```
    u32ProgramFlashAddress = 0;
    u32VerifyFlashAddress = 0;

    for (u32PageNumber = 0; u32PageNumber < TEST_NUMBER; u32PageNumber++)
    {
        printf("\n\nTest Page Number = %d", u32PageNumber);

        CLK_SysTickDelay(200000);

        /*=== Program SPI Flash ===*/
        printf("\nFlash Programming... ");

        /* Page Program */
        SpiFlash_PDMA_PageProgram((uint32_t)SrcArray, u32ProgramFlashAddress);
        SpiFlash_WaitReady();
        u32ProgramFlashAddress += 0x100;
        printf("Done!");

        /*=== Read Back and Compare Data ===*/
        printf("\nFlash Verifying... ");
```

```
        /* Page Read */
        SpiFlash_PDMA_ReadData((uint32_t)DestArray, u32VerifyFlashAddress);
        u32VerifyFlashAddress += 0x100;

        for (u32ByteCount = 0; u32ByteCount < 256; u32ByteCount++)
        {
            if (DestArray[u32ByteCount] != u32ByteCount)
            {
                /* Error */
                printf("SPI Flash R/W Fail!");

                while (1);
            }
        }

        /* Clear Destination Data Buffer */
        for (u32ByteCount = 0; u32ByteCount < 256; u32ByteCount++)
            DestArray[u32ByteCount] = 0;

        printf("Done!");
    }

    printf("\n\nSPI Flash Test Ok!");
    printf("\n\n");

    }
```

Note that the functions used to access SPI Flash in this demo are processed through PDMA. They are:

- ***SpiFlash_PDMA_PageProgram()***
- ***SpiFlash_PDMA_ReadData()***

The SPI0 Master uses PDMA Basic mode to transfer data to TX and uses PDMA Basic mode to transfer RX data to memory.
The SPI TX PDMA settings include source address must be word alignment, PDMA channel, data length, data byte counts, PDMA source is memory buffer, PDMA destination is SPI MOSI (SPI0->TX), PDMA Basic mode and PDMA signal request type. This code does not use the PDMA interrupt..

```
void Init_PDMA_CH1_for_SPI0_TX(uint32_t u32SrcAddr)
{
    /* Enable PDMA channels */
    PDMA_Open(PDMA, (1 << SPI_TX_DMA_CH));

    /* SPI PDMA TX channel configuration */
    /* Set transfer width (8 bits) and transfer count */
    PDMA_SetTransferCnt(PDMA, SPI_TX_DMA_CH, PDMA_WIDTH_8, PDMA_TEST_LENGTH);
    /* Set source/destination address and attributes */
    PDMA_SetTransferAddr(PDMA, SPI_TX_DMA_CH, (uint32_t)u32SrcAddr, PDMA_SAR_INC,
    (uint32_t)&SPI0->TX, PDMA_DAR_FIX);
    /* Set request source; set basic mode. */
    PDMA_SetTransferMode(PDMA, SPI_TX_DMA_CH, PDMA_SPI0_TX, FALSE, 0);
    /* Single request type */
    PDMA_SetBurstType(PDMA, SPI_TX_DMA_CH, PDMA_REQ_SINGLE, PDMA_BURST_128);
    /* Disable table interrupt */
    PDMA->DSCT[SPI_TX_DMA_CH].CTL |= PDMA_DSCT_CTL_TBINTDIS_Msk;
}…
```

The SPI transfer process needs to check the DMA transfer done interrupt flag. When DMA transfer done transfer flag is raised, this means that the SPI data transfer is finished.

```
    // enable SPI PDMA
    /* Initial PDMA Channel */
    Init_PDMA_CH1_for_SPI0_TX((uint32_t)u32SrcAddr);
    SPI_TRIGGER_TX_PDMA(SPI_FLASH_PORT);

    while (1)
    {
            /* Get interrupt status */
            u32RegValue = PDMA_GET_INT_STATUS(PDMA);

            /* Check the DMA transfer done interrupt flag */
            if (u32RegValue & PDMA_INTSTS_TDIF_Msk)
            {
                /* Check the PDMA transfer done interrupt flags */
                if ((PDMA_GET_TD_STS(PDMA) & (1 << SPI_TX_DMA_CH)) == (1 <<
                    SPI_TX_DMA_CH))
                {
                    /* Clear the DMA transfer done flags */
                    PDMA_CLR_TD_FLAG(PDMA, 1 << SPI_TX_DMA_CH);
```

```
                    /* Disable SPI PDMA TX function */
                    SPI_DISABLE_TX_PDMA(SPI_FLASH_PORT);
                    break;
                }
    …
```

The SPI RX PDMA settings include destination buffer start address must be word alignment, PDMA channel, data length, data byte counts, PDMA source is SPI MISO (SPI0->RX), PDMA destination is memory buffer, PDMA Basic mode and PDMA signal request type. This code does not use the PDMA interrupt.

```
void Init_PDMA_CH2_for_SPI0_RX(uint32_t u32DstAddr)
{
    /* Enable PDMA channels */
    PDMA_Open(PDMA, (1 << SPI_RX_DMA_CH));

    /* SPI PDMA RX channel configuration */
    /* Set transfer width (8 bits) and transfer count */
    PDMA_SetTransferCnt(PDMA, SPI_RX_DMA_CH, PDMA_WIDTH_8, PDMA_TEST_LENGTH);
    /* Set source/destination address and attributes */
    PDMA_SetTransferAddr(PDMA, SPI_RX_DMA_CH, (uint32_t)&SPI0->RX, PDMA_SAR_FIX,
    (uint32_t)u32DstAddr, PDMA_DAR_INC);
    /* Set request source; set basic mode. */
    PDMA_SetTransferMode(PDMA, SPI_RX_DMA_CH, PDMA_SPI0_RX, FALSE, 0);
    /* Single request type */
    PDMA_SetBurstType(PDMA, SPI_RX_DMA_CH, PDMA_REQ_SINGLE, PDMA_BURST_128);
    /* Disable table interrupt */
    PDMA->DSCT[SPI_RX_DMA_CH].CTL |= PDMA_DSCT_CTL_TBINTDIS_Msk;
}…
```

The process needs to check the DMA transfer done interrupt flag. When the DMA transfer flag done is raised, this means that the data is in destination buffer.

```
    // enable SPI PDMA
    Init_PDMA_CH2_for_SPI0_RX((uint32_t)u32DestAddr);
    SPI_TRIGGER_RX_PDMA(SPI_FLASH_PORT);

    while (1)
        {
            /* Get interrupt status */
            u32RegValue = PDMA_GET_INT_STATUS(PDMA);

            /* Check the DMA transfer done interrupt flag */
            if (u32RegValue & PDMA_INTSTS_TDIF_Msk)
            {
                /* Check the PDMA transfer done interrupt flags */
                if ((PDMA_GET_TD_STS(PDMA) & (1 << SPI_RX_DMA_CH)) == (1 <<
                   SPI_RX_DMA_CH))
                {
                    /* Clear the DMA transfer done flags */
                    PDMA_CLR_TD_FLAG(PDMA, 1 << SPI_RX_DMA_CH);
                    /* Disable SPI PDMA RX function */
                    SPI_DISABLE_RX_PDMA(SPI_FLASH_PORT);
                    break;
                }
    …
```

3 Software and Hardware Requirements

3.1 Software Requirements

- BSP version
  - M251/M252/M254/M256/M258 Series CMSIS BSP V3.02.002
- IDE version
  - Keil uVersion 5.26

3.2 Hardware Requirements

- Circuit components
  - NuMaker-M252SD Ver1.2
  - SPI FlashW25Q32
- Pin Connect

NuMaker-M252SD Ver1.2		Winbond W25Q32 Flash Memory
VCC	↔	VCC
VCC	↔	HOLD (D3) / RESET
VCC	↔	WP (D2)
VSS	↔	VSS
SPI0_SS (PA.3)	↔	CS
SPI0_CLK (PA.2)	→	CLK
SPI0_MISO (PA.1)	←	DO
SPI0_MOSI (PA.2)	←	DI

Figure 3-1 Pin Connect

4 Directory Information

The directory structure is shown below.

EC_M251_SPI_FLASH_PDMA_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 *SPI_FLASH_PDMA.uvprojx*.
2. Enter Keil compile mode.
   - Build
   - Download
   - Start/Stop debug session
3. Enter debug mode.
   - Run

6 Revision History

Date	Revision	Description
2022.3.24	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.