Lesson 9 SPI_Flash Write and Read

1. Project Purpose

Learn the principle of FLASH memory and SPI bus communication and realize to read and write SPI FLASH on the controller, and display the written characters in the serial port assistant.

2. Project Principle

Flash memory is a type of nonvolatile memory that can retain data for an extended period of time even without current supply, and its storage characteristic is equivalent to the hard disk drive. This feature is the basis for flash memory to become the storage medium for various portable digital devices.

The Serial Peripheral Interface (SPI), developed by Motorola, is a high-speed full duplex interface. It is widely used in ADCs, LCDs and MCUs and suitable for occasions with higher communication speed requirements.

SPI FLASH is a type of flash memory that reads and writes through SPI interface. The general SPI FLASH has two characteristics for reading and writing:

1) When writing, only 1 can be written, not 0.
2) When erasing, it is erased by sector (that is, all data becomes 0), and the sector size varies according to different chips (the chip we chose has 4096 bytes per sector).

Based on the above two points, we can know that the data of a byte is to change the corresponding data bit in the chip from 0 to 1 or from 1 to zero.

Because FLASH does not support writing 0 when writing so we are required to erase the corresponding sector to 0. However, the original data will be lost after erasing, so it is generally read first, and then the sector is erased. At the end, rewrite the modified data into Flash.

3. Program Analyst

1) The STM32 has a SPI hardware interface inside. Communication with SPI_FLASH can be completed as long as it is properly configured.

2) After configuring the clock and each I/O port and then setting the SPI, the communication can be started.

```
void InitSpi(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    GPIO_SetBits(GPIOA,GPIO_Pin_4);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;  //SPI CS
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);


    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA|RCC_APB2Periph_SPI1, ENABLE);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;  //Multiplexed push-pull output
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    GPIO_SetBits(GPIOA,GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7);

    SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;  //Set SPI one-way or two-way data mode: SPI is set to two-wire two-way full duplex
    SPI_InitStructure.SPI_Mode = SPI_Mode_Master;       //Set SPI work mode: set to master SPI
    SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;       //Set the data size of SPI: SPI send and receive 8-bit frame structure
    SPI_InitStructure.SPI_CPOL = SPI_CPOL_High;     //The steady state of the serial clock is selected: the clock is floating high
    SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge;    //Data is captured on the second clock edge
    SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;       //NSS signal is managed by hardware (NSS pin) or software (using SSI bit): the internal NSS signal is controlled by SSI bit
    SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_4;      //Define the value of the baud rate prescaler: the baud rate prescaler value is 256
    SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;  //Specify whether the data transmission starts from the MSB bit or the LSB bit: Data transmission starts from the MSB bit
    SPI_InitStructure.SPI_CRCPolynomial = 7;    //The polynomial of CRC value calculation
    SPI_Init(SPI1, &SPI_InitStructure);  //Initialize the peripheral SPIx register according to the parameters specified in SPI_InitStruct

    SPI_Cmd(SPI1, ENABLE); //Enable SPI peripheral

    SPIx_ReadWriteByte(0xff);//start transfer

}
```

3) As shown in the figure above, the code in green box is the chip select pin for configuring Flash. The code in red box is for turning on the related clock. The code in yellow box is for configuring the three pins SPI_MISO、SPI_MOSI、SPI_CLK to the multiplexed push-pull output. The code in purple box is for configuring the working parameters of SPI.

4) After configuring the SPI interface, we can write the data to be sent into the register, the hardware processing will be automatically completed. According to the feature of SPI, each time a byte is sent, the microcontroller will also receive a byte. The following figure is the function of data interaction:

5) When the SPI communication function is available, we can enable communication between SPI Flash. Writing and reading SPI Flash requires sending corresponding commands to SPI Flash and these commands are usually described in the chip manual.

The figure shown below is the command table of SPI FLASH:

INSTRUCTION NAME	BYTE 1 (CODE)	BYTE 2	BYTE 3	BYTE 4	BYTE 5	BYTE 6
Write Enable	06h					
Write Disable	04h					
Read Status Register-1	05h	(S7–S0) (2)				
Read Status Register-2	35h	(S15-S8) (2)				
Write Status Register	01h	(S7–S0)	(S15-S8)			
Page Program	02h	A23–A16	A15–A8	A7–A0	(D7–D0)	
Quad Page Program	32h	A23–A16	A15–A8	A7–A0	(D7–D0, …)(3)	
Block Erase (64KB)	D8h	A23–A16	A15–A8	A7–A0		
Block Erase (32KB)	52h	A23–A16	A15–A8	A7–A0		
Sector Erase (4KB)	20h	A23–A16	A15–A8	A7–A0		
Chip Erase	C7h/60h					
Erase Suspend	75h					
Erase Resume	7Ah					
Power-down	B9h					
High Performance Mode	A3h	dummy	dummy	dummy		
Mode Bit Reset (4)	FFh	FFh				
Release Power down or HPM / Device ID	ABh	dummy	dummy	dummy	(ID7-ID0) (5)	
Manufacturer/ Device ID(6)	90h	dummy	dummy	00h	(M7-M0)	(ID7-ID0)
Read Unique ID(7)	4Bh	dummy	dummy	dummy	Dummy	(ID63-ID0)
JEDEC ID	9Fh	(M7-M0) Manufacturer	(ID15-ID8) Memory Type	(ID7-ID0) Capacity		

6) According to the table above, define the commands that may be used are defined in the header files with micro definition for later use.

```
//指令表
#define W25X_WriteEnable     0x06
#define W25X_WriteDisable    0x04
#define W25X_ReadStatusReg     0x05
#define W25X_WriteStatusReg    0x01
#define W25X_ReadData      0x03
#define W25X_FastReadData    0x0B
#define W25X_FastReadDual    0x3B
#define W25X_PageProgram     0x02
#define W25X_BlockErase      0xD8
#define W25X_SectorErase     0x20
#define W25X_ChipErase       0xC7
#define W25X_PowerDown       0xB9
#define W25X_ReleasePowerDown 0xAB
#define W25X_DeviceID      0xAB
#define W25X_ManufactDeviceID 0x90
#define W25X_JedecDeviceID     0x9F
```

7) Then check whether Flash is busy or not. When Flash is busy, it will not read or write commands. Therefore, the status of Flash should be checked first before reading and writing. When Flash is idle, it can write and read. In addition, FlashErase and FLashEraseSector perform all ease and sector erase respectively.

```
u8 SPI_Flash_ReadSR(void)
{
    u8 byte=0;
    SPI_FLASH_CS=0;                              //Enable device
    SPIx_ReadWriteByte(W25X_ReadStatusReg);      //Send read status register command
    byte=SPIx_ReadWriteByte(0Xff);               //Read byte
    SPI_FLASH_CS=1;                              //Cancel chip selection
    return byte;
}

//wait idle
void SPI_Flash_Wait_Busy(void)
{
    while((SPI_Flash_ReadSR()&0x01)==0x01);      // Wait for the BUSY bit to be cleared
}
// /**************************************************
```

```
void SPI_FLASH_Write_Enable(void)
{
    SPI_FLASH_CS=0;                              //enable device
    SPIx_ReadWriteByte(W25X_WriteEnable);        //write enable
    SPI_FLASH_CS=1;                              //cancel chip selection
}
```

```
/**************************************************
Erase sector, sector size is 4096, the minimum erase size of Flash is erased in units of sectors
Entrance parameters:
        addr
Exit parameters: None
**************************************************/
void FlashEraseSector(DWORD addr)
{
    SPI_FLASH_Write_Enable();                    //SET WEL
    SPI_Flash_Wait_Busy();
    SPI_FLASH_CS=0;                              //enable device
    SPIx_ReadWriteByte(W25X_SectorErase);        //send sector erase command
    SPIx_ReadWriteByte((u8)((addr)>>16));    //send 24bit address
    SPIx_ReadWriteByte((u8)((addr)>>8));
    SPIx_ReadWriteByte((u8)addr);
    SPI_FLASH_CS=1;                              //cancel chip selection
    SPI_Flash_Wait_Busy();                       //Wait for the erase to complete
}
```

8) The following two functions is to read and write the data in Flash:

```
/*************************************************
Read data from Flash
Entrance parameter:
    addr   : address parameter
    size   : data block size
    buffer : buffer data read from Flash
Exit parameter: Null
*************************************************/
void FlashRead(DWORD addr, DWORD size, BYTE *buffer)
{
    u16 i;
    SPI_FLASH_CS=0;                              //enable device
    SPIx_ReadWriteByte(W25X_ReadData);           //send read command
    SPIx_ReadWriteByte((u8)((addr)>>16));  //send 24bit address
    SPIx_ReadWriteByte((u8)((addr)>>8));
    SPIx_ReadWriteByte((u8)addr);
    for(i=0; i<size; i++)
    {
        buffer[i]=SPIx_ReadWriteByte(0XFF);   //loop to read
    }
    SPI_FLASH_CS=1;                              //cancel chip selection
}


/*************************************************
Write data into Flash
Entrance parameter :
    addr   : address parameter
    size   : data block size
    buffer : Buffer data that needs to be written to Flash
Exit parameter: Null
*************************************************/
void FlashWrite(DWORD addr, DWORD size, BYTE *buffer)
{
    u16 i;
    SPI_FLASH_Write_Enable();                    //SET WEL
    SPI_FLASH_CS=0;                              //enable device
    SPIx_ReadWriteByte(W25X_PageProgram);        //send page command
    SPIx_ReadWriteByte((u8)((addr)>>16)); //send 24bit address
    SPIx_ReadWriteByte((u8)((addr)>>8));
    SPIx_ReadWriteByte((u8)addr);
    for(i=0; i<size; i++)SPIx_ReadWriteByte(buffer[i]); //loop to write
    SPI_FLASH_CS=1;                              //cancel the chip selection
    SPI_Flash_Wait_Busy();                       //wait for the writing to the end
}
```

Instead of reading, modifying, then erasing, and finally rewriting as described above, the above operation writes the function directly. Why does this situation occur?

It's not that these operations are not required, but we need to finish them ourselves in programming. Because the operations to be performed on Flash are not complex, the task in Flash is relatively single and the situation is relatively fixed. Therefore, we can manually erase the number of sectors and the number of the bytes written, which can solve this problem.

For example, we modify only one byte but cause the program to read and write all 4KB data of the entire sector once, which can improve the efficiency and reduce the RAM usage. Through the combination of the above several functions, you can keep your own data on the SPI Flash.