例程十六　SPI_Flash 的读写

在这个例程中我们将要了解一下 SPI，SPI 也是跟 I2C 一样也是有硬件来实现的。SPI 一般有 4 根线，一条时钟线 CLK，2 条数据线（MISO 和 MOSI），1 条片选信号线。

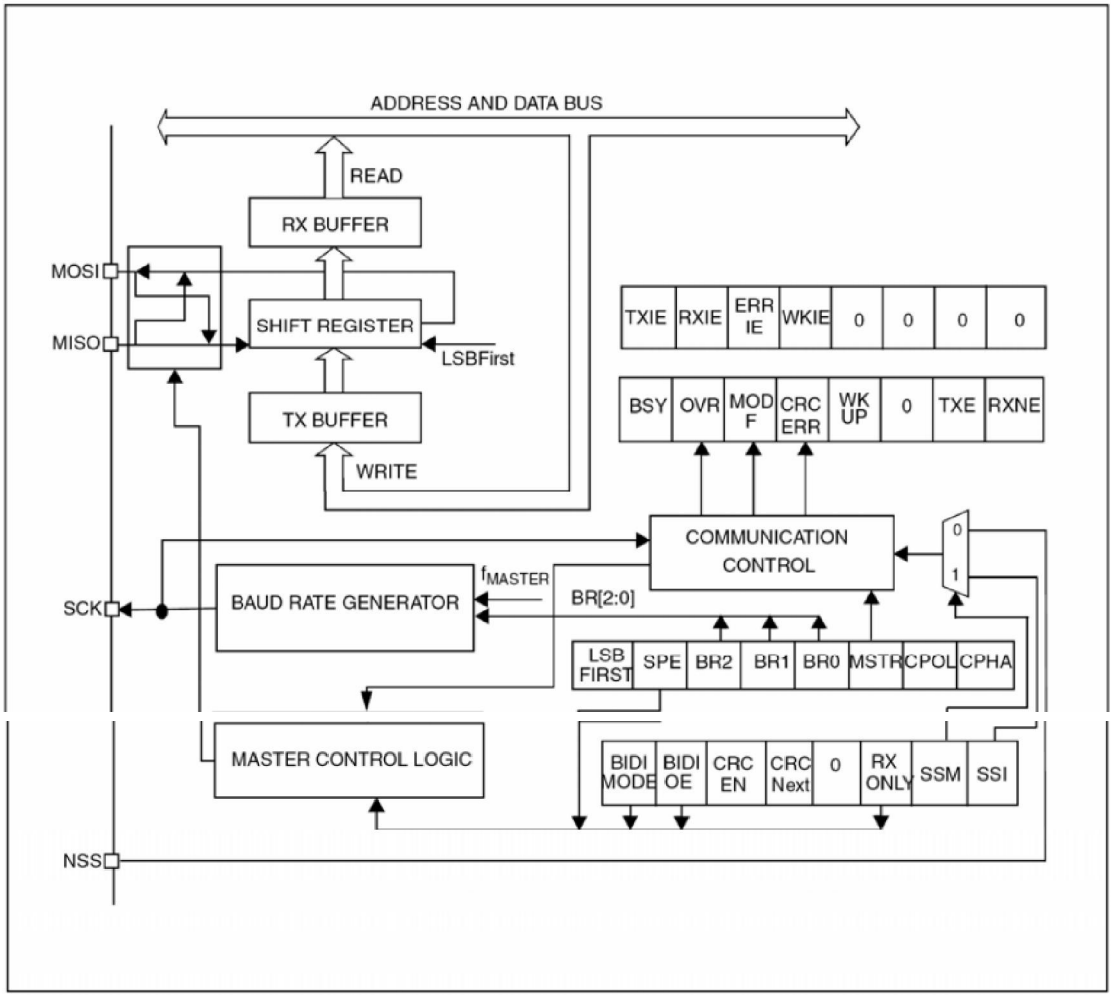
20　串行外设接口(SPI)

20.1　SPI简介

串行外设接口(SPI)允许芯片与其他设备以半/全双工、同步、串行方式通信。此接口可以被配置成主模式，并为从设备提供通信时钟(SCK)。接口还能以多主配置方式工作。

它可用于多种用途，包括带或不带第三根双向数据线的双线单工同步传输，还可使用CRC校验来进行可靠通信。

图88　SPI框图



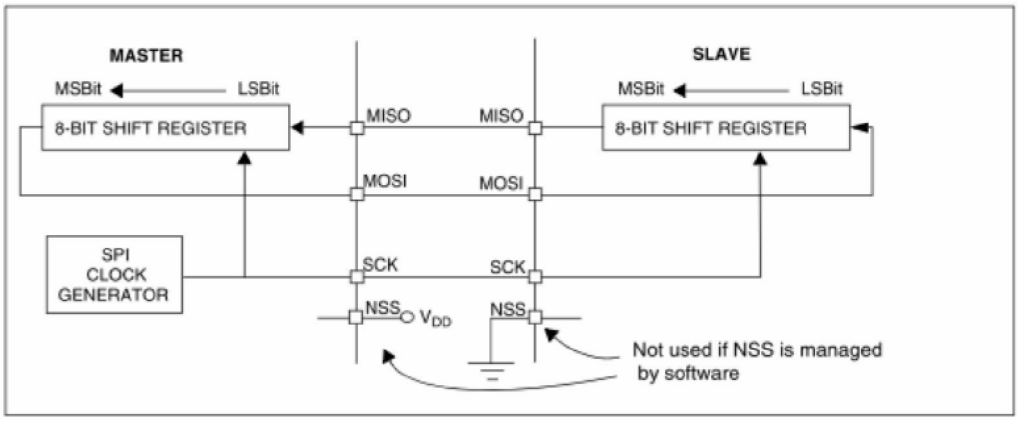
通常SPI通过4个管脚与外部器件相连：

- MISO：主设备输入/从设备输出管脚(端口C7)。该管脚在从模式下发送数据，在主模式下接收数据。
- MOSI：主设备输出/从设备输入管脚(端口C6)。该管脚在主模式下发送数据，在从模式下接收数据。
- SCK：串口时钟(端口C5)，作为主设备的输出，从设备的输入
- NSS：从设备选择(端口E5)。配置主/从模式时，这是一个可选的管脚。它的功能是用来作为“片选管脚”，让主设备可以单独地与特定的从设备通讯，避免数据线上的冲突。从设备的NSS管脚可以被主设备的标准IO来驱动。

*图89*是一个单主和单从设备互连的例子。

注意：当使用SPI的高速模式时，SPI输出端口对应的I/O必须配置为快速摆率输出，以满足要求的总线速度。

图89　单主和单从应用



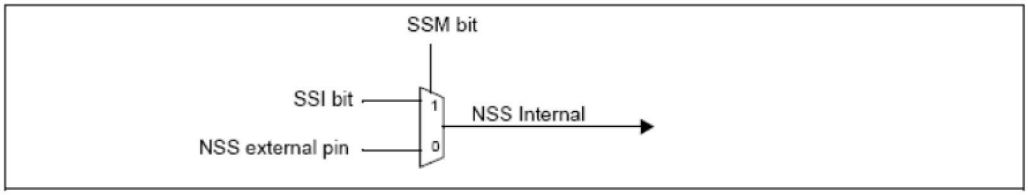
MOSI脚相互连接，MISO脚相互连接。这样，数据在主和从之间串行地传输(MSB位在前)。

通信总是由主设备发起。主设备通过MOSI脚把数据发送给从设备，从设备通过MISO引脚回传数据。这意味全双工通信的数据输出和数据输入是用同一个时钟信号同步的；时钟信号由主设备通过SCK脚提供。

从选择(NSS)脚管理

应用程序可以使用软件的方式代替使用NSS管脚(NSS引脚，端口E5)来控制从设备选择信号，参见*图90*。这样的话，外部的NSS引脚可以空出来给其他应用使用；而内部NSS信号的电平由SPI_CSR寄存器中的SSI位控制了。

图90　硬件/软件的从选择管理



时钟信号的相位和极性

使用CPOL和CPHA位，能够组合成四种可能的时序关系。CPOL(时钟极性)位控制在没有数据传输时时钟的空闲状态电平，此位对主模式和从模式下的设备都有效。如果CPOL被清'0'，SCK引脚在空闲状态保持低电平；如果CPOL被置'1'，SCK引脚在空闲状态保持高电平。

注意：确保SPI引脚配置成SPI空闲状态时的电平，以免在使能或者禁止SPI模块的时候在SPI时钟引脚上产生一个边沿。

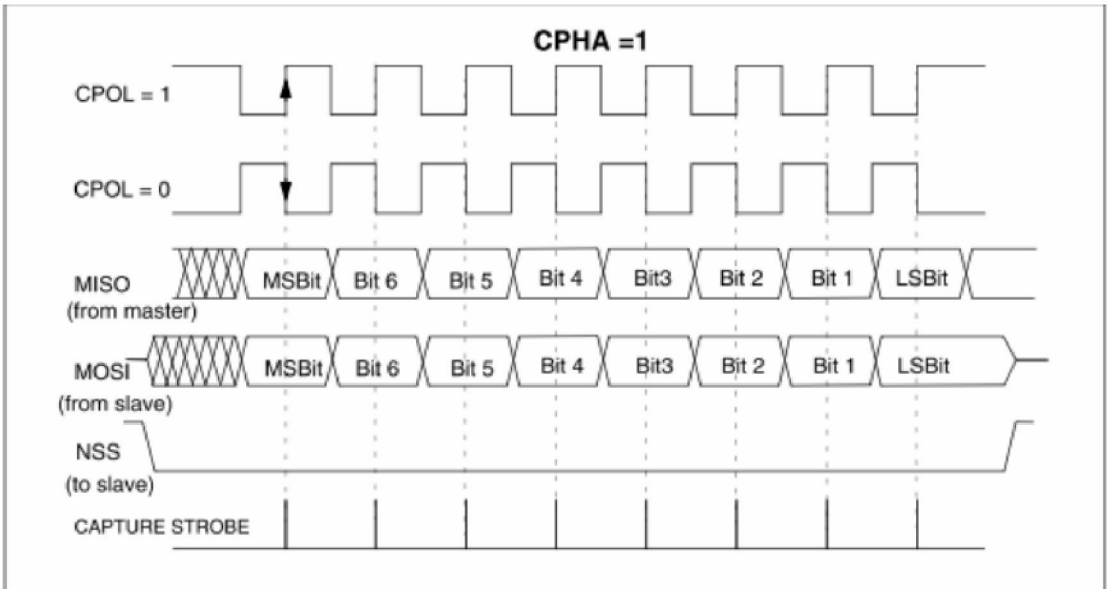
如果CPHA(时钟相位)位被置'1'，SCK时钟的第二个边沿(CPOL位为0时就是下降沿，CPOL位为1时就是上升沿)进行最高数据位的采样，数据在第一个时钟传输周期被锁存。如果CPHA位被清'0'，SCK时钟的第一边沿(CPOL位为0时就是下降沿，CPOL位为1时就是上升沿)进行数据位采样，数据在第二个时钟传输周期被锁存。

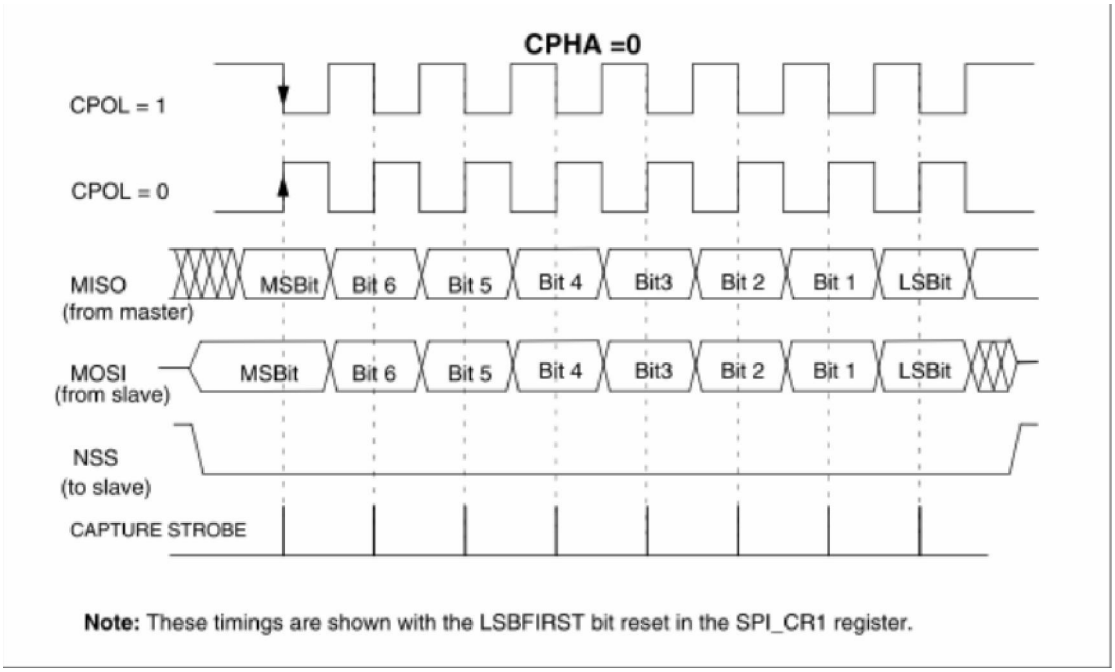
CPOL时钟极性和CPHA时钟相位的组合选择数据捕捉的时钟边沿。

*图91*显示了SPI传输的4种CPHA和CPOL位选择不同的组合主设备与从设备的SCK，MISO，MOSI引脚直接连接时，这些管脚上的时序。

注意：
1. *在改变CPOL/CPHA位之前，必须清除SPE位将SPI禁止。*
2. *主设备和从设备必须配置成相同的时序模式。*
3. *SCK的空闲状态必须和SPI_CR1寄存器指定的极性一致(CPOL为1时，应上拉SCK为高电平；CPOL为0时，应下拉SCK为低电平)。*

图91　数据时钟时序图





Note: These timings are shown with the LSBFIRST bit reset in the SPI_CR1 register.

如果大家想了解更多关于 SPI 的话，大家可以参考“STM8 寄存器.pdf”文档。里面有更详细的解释以说明。

我们平时多用到的是 SPI 主模式，跟 I2C 一样。

下面讲一下 SPI 主模式

在主配置时，串行时钟在SCK脚产生。

配置步骤

1. 通过SPI_CR1寄存器的BR[2:0]位定义串行时钟波特率。
2. 选择CPOL和CPHA位，定义数据传输和串行时钟间的相位关系(见*图91*)。
3. 配置SPI_CR1寄存器的LSBFIRST位定义帧格式。
4. 硬件模式下，在数据帧的全部传输过程中应把NSS脚连接到高电平；在软件模式下，需设置SPI_CR2寄存器的SSM和SSI位为‘1’。
5. 必须设置MSTR和SPE位(只当NSS脚被连到高电平，这些位才能保持为‘1’)。

在这个配置中，MOSI脚是数据输出，而MISO脚是数据输入。

数据传输过程

当一字节写进发送缓冲器时，发送过程开始。

在发送第一个数据位时，数据字被并行地(通过内部总线)传入移位寄存器，而后串行地移出到MOSI脚上；MSB在先还是LSB在先，取决于SPI_CR1寄存器中的LSBFIRST位。数据从发送缓冲器传输到移位寄存器时TXE标志将被置位，如果设置SPI_CR1寄存器中的TXEIE位，将产生中断。

当数据传输完成时：

- 移位寄存器里的数据传送到接收缓冲器，并且RXNE标志被置位。
- 如果SPI_ICR寄存器中的RXIE位被设置，则产生中断。

在最后一个采样时钟沿，RXNE位被设置，在移位寄存器中接收到的数据字被传送到接收缓冲器。读取SPI_DR寄存器得到这个缓冲值。读SPI_DR寄存器将清除RXNE位。

一旦传输开始，如果下一个将发送的数据被放进了发送缓冲器，就可以维持一个连续的传输流。注意在试图写发送缓冲器之前，需确认TXE标志应该是1。

一些关于常见初始化和配置的知识都讲了，下面我们从主函数看起，逐步分析：

```
void main(void)
{

  /* Infinite loop */

  /*设置内部时钟16M为主时钟*/
  CLK_HSIPrescalerConfig(CLK_PRESCALER_HSIDIV1);
  /*!<Set High speed internal clock  */
  SPI_FLASH_Init();
  Uart_Init();
  FLASH_ID = SPI_FLASH_ReadID();

  /* Perform a write in the Flash followed by a read of the written data */
  /* Erase SPI FLASH Sector to write on */
  SPI_FLASH_SectorErase(FLASH_SectorToErase);

  /* Write Tx_Buffer data to SPI FLASH memory */
  SPI_FLASH_BufferWrite(Tx_Buffer, FLASH_WriteAddress, BufferSize);

  /* Read data from SPI FLASH memory */
  SPI_FLASH_BufferRead(Rx_Buffer, FLASH_ReadAddress, BufferSize);

   __enable_interrupt();
   UART1_SendString(Rx_Buffer,BufferSize);
   Delay(0xffff);
   while (1)
   {



   }

}
```

SPI_FLASH_Init() SPI 的初始化
看看函数原型：

```
void SPI_FLASH_Init(void)
{
   SPI_Init(SPI_FIRSTBIT_MSB, SPI_BAUDRATEPRESCALER_2, SPI_MODE_MASTER,\
            SPI_CLOCKPOLARITY_HIGH, SPI_CLOCKPHASE_2EDGE, \
            SPI_DATADIRECTION_2LINES_FULLDUPLEX, SPI_NSS_SOFT, 0x07);
   SPI_Cmd(ENABLE);
   GPIO_Init(SPI_CS , SPI_Pin_CS, GPIO_MODE_OUT_PP_HIGH_FAST);
   SPI_FLASH_CS_HIGH();

}
```

SPI_Init()配置了数据位高位先发送，SPI 的传输的波特率为系统频率的 2 分频，SPI 为工作在主模式，当 SPI 处于空闲状态时时钟线处于高电平，数据在第二个时钟传输周期被锁，数据线为全双工，使用软件方式代替使用 NSS 管脚，使用 CRC 校验。
SPI_Cmd()时能 SPI
设置片选信号线，先把它置高。
初始化弄好了，下面来看看怎样往从设备读写，首先是读写一个字节。
我们开发板上的从设备上 W25X16 的，所以我们要根据 W25X16 的 Datasheet 来配置一下参数。

```
/*******************************************************************************
* Function Name  : SPI_FLASH_SendByte
* Description    : Sends a byte through the SPI interface and return the byte
*                  received from the SPI bus.
* Input          : byte : byte to send.
* Output         : None
* Return         : The value of the received byte.
*******************************************************************************/
u8 SPI_FLASH_SendByte(u8 byte)
{
  /* Loop while DR register in not emplty */
  while (SPI_GetFlagStatus( SPI_FLAG_TXE) == RESET);

  /* Send byte through the SPI1 peripheral */
  SPI_SendData(byte);

  /* Wait to receive a byte */
  while (SPI_GetFlagStatus(SPI_FLAG_RXNE) == RESET);

  /* Return the byte read from the SPI bus */
  return SPI_ReceiveData();
 }
```

这个函数就是在 SPI 总线上读写一个字节的操作，是根据 SPI 总线的时序来完成的。可以参考数据传输过程那里的说明。

```
/*******************************************************************************
* Function Name  : SPI_FLASH_WriteEnable
* Description    : Enables the write access to the FLASH.
* Input          : None
* Output         : None
* Return         : None
*******************************************************************************/
void SPI_FLASH_WriteEnable(void)
{
  /* Select the FLASH: Chip Select low */
  SPI_FLASH_CS_LOW();

  /* Send "Write Enable" instruction */
  SPI_FLASH_SendByte(WREN);

  /* Deselect the FLASH: Chip Select high */
  SPI_FLASH_CS_HIGH();
}
```

这个函数就是往 W25X16 的写数据的写使能，参考一下的时序

11.2.3 Write Enable (06h)

The Write Enable instruction (Figure 4) sets the Write Enable Latch (WEL) bit in the Status Register to a 1. The WEL bit must be set prior to every Page Program, Sector Erase, Block Erase, Chip Erase and Write Status Register instruction. The Write Enable instruction is entered by driving /CS low, shifting the instruction code "06h" into the Data Input (DI) pin on the rising edge of CLK, and then driving /CS high.

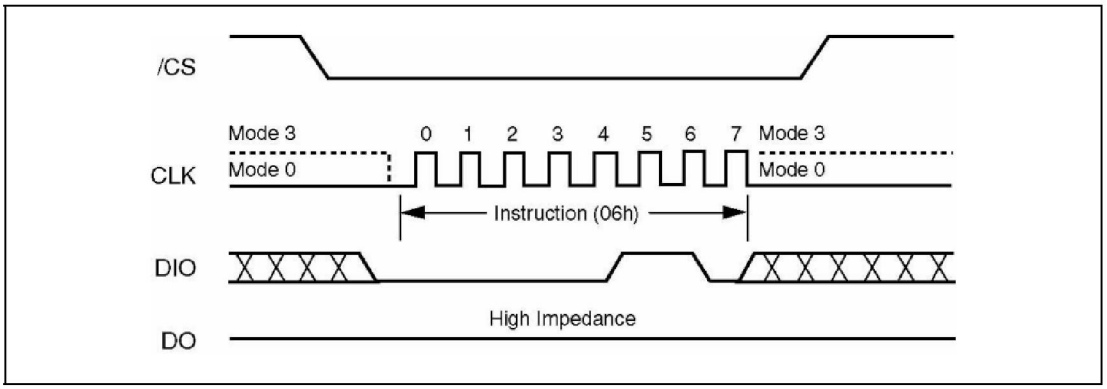


Figure 4. Write Enable Instruction Sequence Diagram

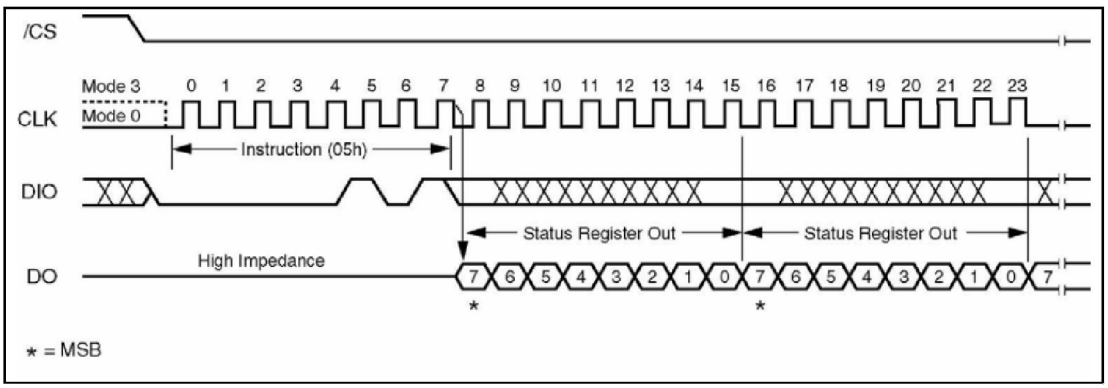
```
/*******************************************************************************
* Function Name  : SPI_FLASH_WaitForWriteEnd
* Description    : Polls the status of the Write In Progress (WIP) flag in the
*                  FLASH's status  register  and  loop  until write  opertaion
*                  has completed.
* Input          : None
* Output         : None
* Return         : None
*******************************************************************************/
void SPI_FLASH_WaitForWriteEnd(void)
{
  u8 FLASH_Status = 0;

  /* Select the FLASH: Chip Select low */
  SPI_FLASH_CS_LOW();

  /* Send "Read Status Register" instruction */
  SPI_FLASH_SendByte(RDSR);

  /* Loop as long as the memory is busy with a write cycle */
  do
  {
    /* Send a dummy byte to generate the clock needed by the FLASH
    and put the value of the status register in FLASH_Status variable */
    FLASH_Status = SPI_FLASH_SendByte(Dummy_Byte);

  }
  while ((FLASH_Status & WIP_Flag) == SET); /* Write in progress */

  /* Deselect the FLASH: Chip Select high */
  SPI_FLASH_CS_HIGH();
}
```

准备写一个数据，参考时序

11.2.5 Read Status Register (05h)

The Read Status Register instruction allows the 8-bit Status Register to be read. The instruction is entered by driving /CS low and shifting the instruction code "05h" into the DIO pin on the rising edge of CLK. The status register bits are then shifted out on the DO pin at the falling edge of CLK with most significant bit (MSB) first as shown in figure 6. The Status Register bits are shown in figure 3 and include the BUSY, WEL, BP2-BP0, TB and SRP bits (see description of the Status Register earlier in this datasheet).

The Status Register instruction may be used at any time, even while a Program, Erase or Write Status Register cycle is in progress. This allows the BUSY status bit to be checked to determine when the cycle is complete and if the device can accept another instruction. The Status Register can be read continuously, as shown in Figure 6. The instruction is completed by driving /CS high.




```
/*******************************************************************************
* Function Name  : SPI_FLASH_PageWrite
* Description    : Writes more than one byte to the FLASH with a single WRITE
*                  cycle(Page WRITE sequence). The number of byte can't exceed
*                  the FLASH page size.
* Input          : - pBuffer : pointer to the buffer  containing the data to be
*                    written to the FLASH.
*                  - WriteAddr : FLASH's internal address to write to.
*                  - NumByteToWrite : number of bytes to write to the FLASH,
*                    must be equal or less than "SPI_FLASH_PageSize" value.
* Output         : None
* Return         : None
*******************************************************************************/
void SPI_FLASH_PageWrite(u8* pBuffer, u32 WriteAddr, u16 NumByteToWrite)
{
  /* Enable the write access to the FLASH */
  SPI_FLASH_WriteEnable();

  /* Select the FLASH: Chip Select low */
  SPI_FLASH_CS_LOW();
  /* Send "Write to Memory " instruction */
  SPI_FLASH_SendByte(WRITE);
```

```
  /*******************************************************************/
  /* Send WriteAddr high nibble address byte to write to */
  SPI_FLASH_SendByte((WriteAddr & 0xFF0000) >> 16);
  /* Send WriteAddr medium nibble address byte to write to */
  SPI_FLASH_SendByte((WriteAddr & 0xFF00) >> 8);
  /* Send WriteAddr low nibble address byte to write to */
  SPI_FLASH_SendByte(WriteAddr & 0xFF);
 /*******************************************************************/

  /* while there is data to be written on the FLASH */
  while (NumByteToWrite--)
  {
    /* Send the current byte */
    SPI_FLASH_SendByte(*pBuffer);
    /* Point on the next byte to be written */
    pBuffer++;
  }

  /* Deselect the FLASH: Chip Select high */
  SPI_FLASH_CS_HIGH();

  /* Wait the end of Flash writing */
  SPI_FLASH_WaitForWriteEnd();
}
```

往 W25X16 的一页写数据，每页做多能写 4102Byte。

```
/*******************************************************************************
* Function Name  : SPI_FLASH_BufferWrite
* Description    : Writes block of data to the FLASH. In this function, the
*                  number of WRITE cycles are reduced, using Page WRITE sequence.
* Input          : - pBuffer : pointer to the buffer  containing the data to be
*                    written to the FLASH.
*                  - WriteAddr : FLASH's internal address to write to.
*                  - NumByteToWrite : number of bytes to write to the FLASH.
* Output         : None
* Return         : None
*******************************************************************************/
void SPI_FLASH_BufferWrite(u8* pBuffer, u32 WriteAddr, u16 NumByteToWrite)
{
  u8 NumOfPage = 0, NumOfSingle = 0, Addr = 0, count = 0, temp = 0;

  Addr = WriteAddr % SPI_FLASH_PageSize;
  count = SPI_FLASH_PageSize - Addr;
  NumOfPage =  NumByteToWrite / SPI_FLASH_PageSize;
  NumOfSingle = NumByteToWrite % SPI_FLASH_PageSize;

  if (Addr == 0) /* WriteAddr is SPI_FLASH_PageSize aligned  */
  {
    if (NumOfPage == 0) /* NumByteToWrite < SPI_FLASH_PageSize */
    {
      SPI_FLASH_PageWrite(pBuffer, WriteAddr, NumByteToWrite);
    }
    else /* NumByteToWrite > SPI_FLASH_PageSize */
```

```
    {
      while (NumOfPage--)
      {
        SPI_FLASH_PageWrite(pBuffer, WriteAddr, SPI_FLASH_PageSize);
        WriteAddr +=  SPI_FLASH_PageSize;
        pBuffer += SPI_FLASH_PageSize;
      }

      SPI_FLASH_PageWrite(pBuffer, WriteAddr, NumOfSingle);
    }
  }
  else /* WriteAddr is not SPI_FLASH_PageSize aligned  */
  {
    if (NumOfPage == 0) /* NumByteToWrite < SPI_FLASH_PageSize */
    {
      if (NumOfSingle > count)
      { /* (NumByteToWrite + WriteAddr) > SPI_FLASH_PageSize */
        temp = NumOfSingle - count;

        SPI_FLASH_PageWrite(pBuffer, WriteAddr, count);
        WriteAddr +=  count;
        pBuffer += count;

        SPI_FLASH_PageWrite(pBuffer, WriteAddr, temp);
      }
    else
    {
      SPI_FLASH_PageWrite(pBuffer, WriteAddr, NumByteToWrite);
    }
  }
  else /* NumByteToWrite > SPI_FLASH_PageSize */
  {
    NumByteToWrite -= count;
    NumOfPage =  NumByteToWrite / SPI_FLASH_PageSize;
    NumOfSingle = NumByteToWrite % SPI_FLASH_PageSize;

    SPI_FLASH_PageWrite(pBuffer, WriteAddr, count);
    WriteAddr +=  count;
    pBuffer += count;

    while (NumOfPage--)
    {
      SPI_FLASH_PageWrite(pBuffer, WriteAddr, SPI_FLASH_PageSize);
      WriteAddr +=  SPI_FLASH_PageSize;
      pBuffer += SPI_FLASH_PageSize;
    }

    if (NumOfSingle != 0)
    {
      SPI_FLASH_PageWrite(pBuffer, WriteAddr, NumOfSingle);
    }
  }
```

这个函数是往任何地址写任何多个 Byte 的数据，直到写满为止，最大 2Mbyte。

```
/*******************************************************************************
* Function Name  : SPI_FLASH_BufferRead
* Description    : Reads a block of data from the FLASH.
* Input          : - pBuffer : pointer to the buffer that receives the data read
*                    from the FLASH.
*                  - ReadAddr : FLASH's internal address to read from.
*                  - NumByteToRead : number of bytes to read from the FLASH.
* Output         : None
* Return         : None
*******************************************************************************/
void SPI_FLASH_BufferRead(u8* pBuffer, u32 ReadAddr, u16 NumByteToRead)
{
  /* Select the FLASH: Chip Select low */
  SPI_FLASH_CS_LOW();

  /* Send "Read from Memory " instruction */
  SPI_FLASH_SendByte(READ);

  /* Send ReadAddr high nibble address byte to read from */
  SPI_FLASH_SendByte((ReadAddr & 0xFF0000) >> 16);
  /* Send ReadAddr medium nibble address byte to read from */
  SPI_FLASH_SendByte((ReadAddr& 0xFF00) >> 8);
  /* Send ReadAddr low nibble address byte to read from */
  SPI_FLASH_SendByte(ReadAddr & 0xFF);

  while (NumByteToRead--) /* while there is data to be read */
  {
    /* Read a byte from the FLASH */
    *pBuffer = SPI_FLASH_SendByte(Dummy_Byte);
    /* Point to the next location where the byte read will be saved */
    pBuffer++;
  }

  /* Deselect the FLASH: Chip Select high */
  SPI_FLASH_CS_HIGH();
}
```

这个函数是往任何地址读任何个字节的数据缓存在 pBuffer 中。

```
/*******************************************************************************
* Function Name  : SPI_FLASH_ReadID
* Description    : Reads FLASH identification.
* Input          : None
* Output         : None
* Return         : FLASH identification
*******************************************************************************/
```

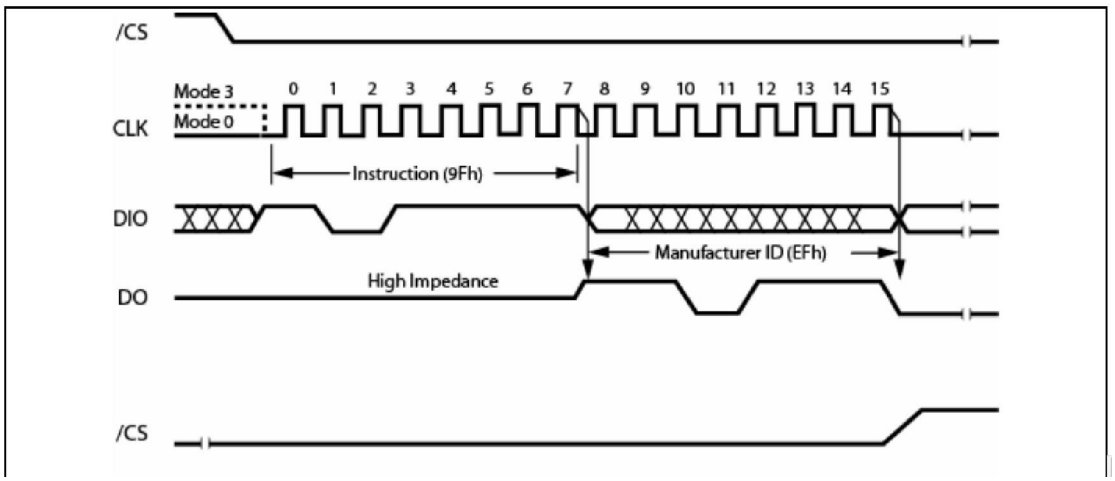
```
u32 SPI_FLASH_ReadID(void)
{
  u32 Temp = 0, Temp0 = 0, Temp1 = 0, Temp2 = 0;

  /* Select the FLASH: Chip Select low */
  SPI_FLASH_CS_LOW();

  /* Send "RDID " instruction */
  SPI_FLASH_SendByte(0x9F);

  /* Read a byte from the FLASH */
  Temp0 = SPI_FLASH_SendByte(Dummy_Byte);

  /* Read a byte from the FLASH */
  Temp1 = SPI_FLASH_SendByte(Dummy_Byte);

  /* Read a byte from the FLASH */
  Temp2 = SPI_FLASH_SendByte(Dummy_Byte);

  /* Deselect the FLASH: Chip Select high */
  SPI_FLASH_CS_HIGH();

  Temp = (Temp0 << 16) | (Temp1 << 8) | Temp2;

  return Temp;
}
```

读 W25X16 的 ID 号，参考时序

11.2.17 JEDEC ID (9Fh)

For compatibility reasons, the W25X16/32/64 provides several instructions to electronically determine the identity of the device. The Read JEDEC ID instruction is compatible with the JEDEC standard for SPI compatible serial memories that was adopted in 2003.

The instruction is initiated by driving the /CS pin low and shifting the instruction code "9Fh". The JEDEC assigned Manufacturer ID byte for Winbond (EFh) and two Device ID bytes, Memory Type (ID15-ID8) and Capacity (ID7-ID0) are then shifted out on the falling edge of CLK with most significant bit (MSB) first as shown in figure 19. For memory type and capacity values refer to Manufacturer and Device Identification table.



```
/*******************************************************************************
* Function Name  : SPI_FLASH_SectorErase
* Description    : Erases the specified FLASH sector.
* Input          : SectorAddr: address of the sector to erase.
* Output         : None
* Return         : None
*******************************************************************************/
void SPI_FLASH_SectorErase(u32 SectorAddr)
{
  /* Send write enable instruction */
  SPI_FLASH_WriteEnable();

  /* Sector Erase */
  /* Select the FLASH: Chip Select low */
  SPI_FLASH_CS_LOW();
  /* Send Sector Erase instruction */
  SPI_FLASH_SendByte(SE);
  /* Send SectorAddr high nibble address byte */
  SPI_FLASH_SendByte((SectorAddr & 0xFF0000) >> 16);
  /* Send SectorAddr medium nibble address byte */
  SPI_FLASH_SendByte((SectorAddr & 0xFF00) >> 8);
  /* Send SectorAddr low nibble address byte */
  SPI_FLASH_SendByte(SectorAddr & 0xFF);
  /* Deselect the FLASH: Chip Select high */
  SPI_FLASH_CS_HIGH();

  /* Wait the end of Flash writing */
  SPI_FLASH_WaitForWriteEnd();
}
```

擦除一个 Block 的数据，参考时序：

11.2.12 Block Erase (D8h)

The Block Erase instruction sets all memory within a specified block (64K-bytes) to the erased state of all 1s (FFh). A Write Enable instruction must be executed before the device will accept the Block Erase Instruction (Status Register bit WEL must equal 1). The instruction is initiated by driving the /CS pin low and shifting the instruction code "D8h" followed a 24-bit block address (A23-A0) (see Figure 2). The Block Erase instruction sequence is shown in figure 13.

The /CS pin must be driven high after the eighth bit of the last byte has been latched. If this is not done the Block Erase instruction will not be executed. After /CS is driven high, the self-timed Block Erase instruction will commence for a time duration of t_{BE} (See AC Characteristics). While the Block Erase cycle is in progress, the Read Status Register instruction may still be accessed for checking the status of the BUSY bit. The BUSY bit is a 1 during the Block Erase cycle and becomes a 0 when the cycle is finished and the device is ready to accept other instructions again. After the Block Erase cycle has finished the Write Enable Latch (WEL) bit in the Status Register is cleared to 0. The Block Erase instruction will not be executed if the addressed page is protected by the Block Protect (TB, BP2, BP1, and BP0) bits (see Status Register Memory Protection table).

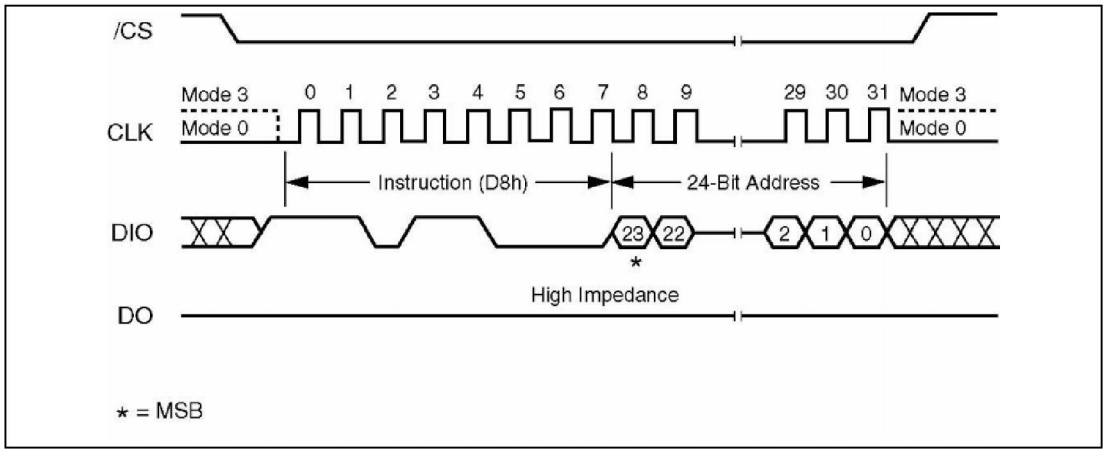


Figure 13. Block Erase Instruction Sequence Diagram

```
/*******************************************************************************
* Function Name  : SPI_FLASH_BulkErase
* Description    : Erases the entire FLASH.
* Input          : None
* Output         : None
* Return         : None
*******************************************************************************/
void SPI_FLASH_BulkErase(void)
{
  /* Send write enable instruction */
  SPI_FLASH_WriteEnable();

  /* Bulk Erase */
  /* Select the FLASH: Chip Select low */
  SPI_FLASH_CS_LOW();
  /* Send Bulk Erase instruction  */
  SPI_FLASH_SendByte(BE);
  /* Deselect the FLASH: Chip Select high */
  SPI_FLASH_CS_HIGH();

  /* Wait the end of Flash writing */
  SPI_FLASH_WaitForWriteEnd();
}
```

擦除整个 Flash， 参考时序

11.2.13 Chip Erase (C7h)

The Chip Erase instruction sets all memory within the device to the erased state of all 1s (FFh). A Write Enable instruction must be executed before the device will accept the Chip Erase Instruction (Status Register bit WEL must equal 1). The instruction is initiated by driving the /CS pin low and shifting the instruction code "C7h". The Chip Erase instruction sequence is shown in figure 14.

The /CS pin must be driven high after the eighth bit has been latched. If this is not done the Chip Erase instruction will not be executed. After /CS is driven high, the self-timed Chip Erase instruction will commence for a time duration of t_{CE} (See AC Characteristics). While the Chip Erase cycle is in progress, the Read Status Register instruction may still be accessed to check the status of the BUSY bit. The BUSY bit is a 1 during the Chip Erase cycle and becomes a 0 when finished and the device is ready to accept other instructions again. After the Chip Erase cycle has finished the Write Enable Latch (WEL) bit in the Status Register is cleared to 0. The Chip Erase instruction will not be executed if any page is protected by the Block Protect (BP2, BP1, and BP0) bits (see Status Register Memory Protection table).

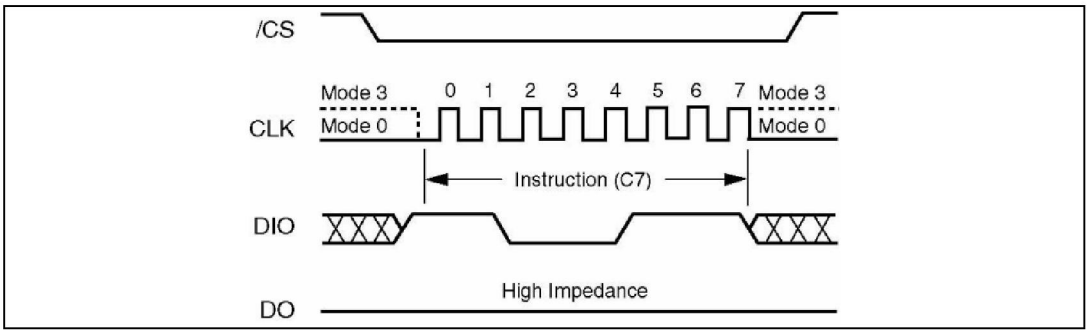


Figure 14. Chip Erase Instruction Sequence Diagram

主要的功能函数模块都讲完了，还有一些测试时候写的测试其他的函数，有兴趣的可以看下工程。

实验效果：

风驰电子祝您学习愉快～～～!!!!!