第1章　SPI—读写串行 FLASH

本章参考资料：《STM32 中文参考手册》SPI 章节及《SPI 总线协议介绍》。

若对 SPI 通讯协议不了解，可先阅读《SPI 总线协议介绍》文档的内容学习。

关于 FLASH 存储器，请参考“常用存储器介绍”章节，实验中 FLASH 芯片的具体参数，请参考其规格书《W25Q64》来了解。

1.1 SPI 协议简介

SPI 协议是由摩托罗拉公司提出的通讯协议(Serial Peripheral Interface)，即串行外围设备接口，是一种高速全双工的通信总线。它被广泛地使用在 ADC、LCD 等设备与 MCU 间，要求通讯速率较高的场合。

学习本章时，可与 I2C 章节对比阅读，体会两种通讯总线的差异以及 EEPROM 存储器与 FLASH 存储器的区别。下面我们分别对 SPI 协议的物理层及协议层进行讲解。

1.1.1 SPI 物理层

SPI 通讯设备之间的常用连接方式见图 1-1。

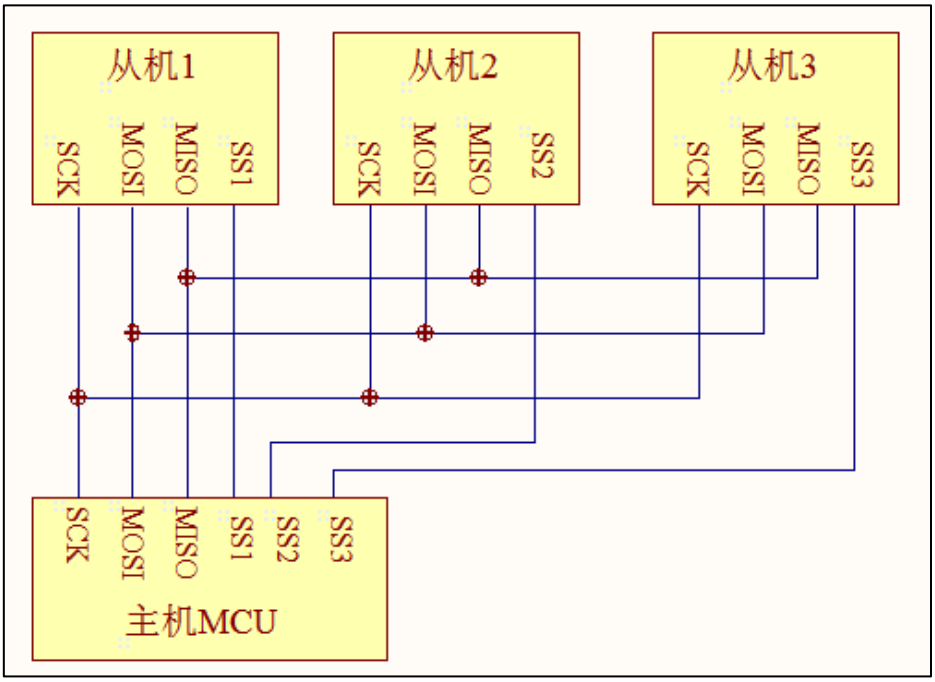

图 1-1 常见的 SPI 通讯系统

SPI 通讯使用 3 条总线及片选线，3 条总线分别为 SCK、MOSI、MISO，片选线为 $\overline{SS}$，它们的作用介绍如下：

(1) $\overline{SS}$ (Slave Select)：从设备选择信号线，常称为片选信号线，也称为 NSS、CS，以下用 NSS 表示。当有多个 SPI 从设备与 SPI 主机相连时，设备的其它信号线 SCK、MOSI 及 MISO 同时并联到相同的 SPI 总线上，即无论有多少个从设备，都共同只使用这 3 条总线；而每个从设备都有独立的这一条 NSS 信号线，本信号线独占主机的一个引脚，即有多少个从设备，就有多少条片选信号线。I2C 协议中通过设备地址来寻址、选中总线上的某个设备并与其进行通讯；而 SPI 协议中没有设备地址，它使用 NSS 信号线来寻址，当主机要选择从设备时，把该从设备的 NSS 信号线设置为低电平，该从设备即被选中，即片选有效，接着主机开始与被选中的从设备进行 SPI 通讯。所以 SPI 通讯以 NSS 线置低电平为开始信号，以 NSS 线被拉高作为结束信号。。

(2) SCK (Serial Clock)：时钟信号线，用于通讯数据同步。它由通讯主机产生，决定了通讯的速率，不同的设备支持的最高时钟频率不一样，如 STM32 的 SPI 时钟频率最大为 $f_{pclk}/2$，两个设备之间通讯时，通讯速率受限于低速设备。

(3) MOSI (Master Output， Slave Input)：主设备输出/从设备输入引脚。主机的数据从这条信号线输出，从机由这条信号线读入主机发送的数据，即这条线上数据的方向为主机到从机。

(4) MISO(Master Input,，Slave Output)：主设备输入/从设备输出引脚。主机从这条信号线读入数据，从机的数据由这条信号线输出到主机，即在这条线上数据的方向为从机到主机。

1.1.2 协议层

与 I2C 的类似，SPI 协议定义了通讯的起始和停止信号、数据有效性、时钟同步等环节。

1. SPI 基本通讯过程

先看看 SPI 通讯的通讯时序，见图 1-2。

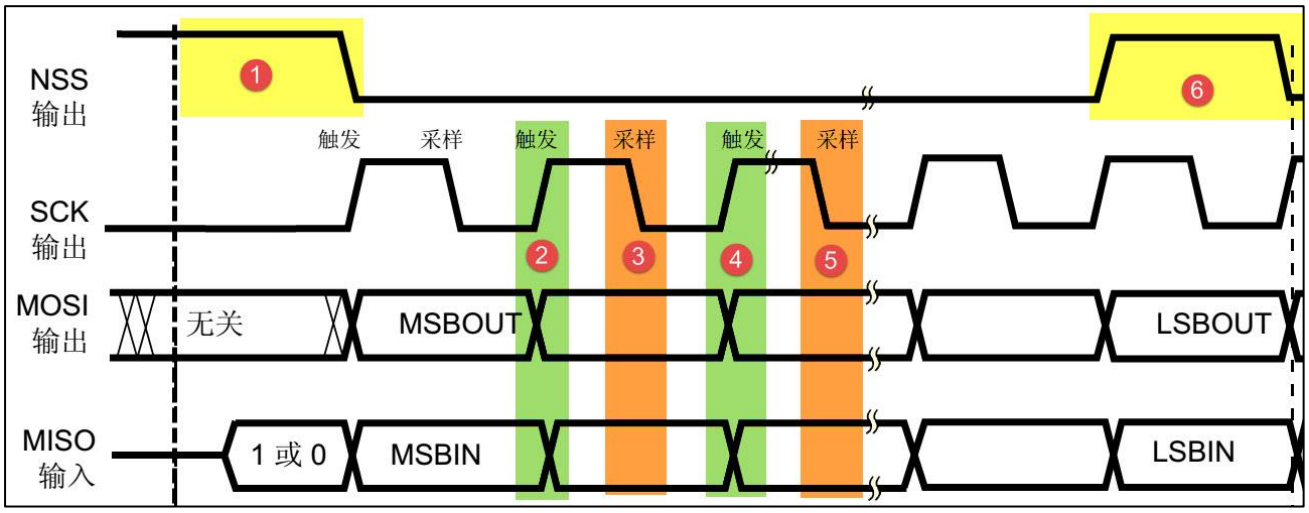

图 1-2 SPI 通讯时序

这是一个主机的通讯时序。NSS、SCK、MOSI 信号都由主机控制产生，而 MISO 的信号由从机产生，主机通过该信号线读取从机的数据。MOSI 与 MISO 的信号只在 NSS 为低电平的时候才有效，在 SCK 的每个时钟周期 MOSI 和 MISO 传输一位数据。

以上通讯流程中包含的各个信号分解如下：

1. 通讯的起始和停止信号

在图 1-2 中的标号①处，NSS 信号线由高变低，是 SPI 通讯的起始信号。NSS 是每个从机各自独占的信号线，当从机在自己的 NSS 线检测到起始信号后，就知道自己被主机选中了，开始准备与主机通讯。在图中的标号⑥处，NSS 信号由低变高，是 SPI 通讯的停止信号，表示本次通讯结束，从机的选中状态被取消。

2. 数据有效性

SPI 使用 MOSI 及 MISO 信号线来传输数据，使用 SCK 信号线进行数据同步。MOSI 及 MISO 数据线在 SCK 的每个时钟周期传输一位数据，且数据输入输出是同时进行的。数据传输时，MSB 先行或 LSB 先行并没有作硬性规定，但要保证两个 SPI 通讯设备之间使用同样的协定，一般都会采用图 1-2 中的 MSB 先行模式。

观察图中的②③④⑤标号处，MOSI 及 MISO 的数据在 SCK 的上升沿期间变化输出，在 SCK 的下降沿时被采样。即在 SCK 的下降沿时刻，MOSI 及 MISO 的数据有效，高电平时表示数据“1”，为低电平时表示数据“0”。在其它时刻，数据无效，MOSI 及 MISO 为下一次表示数据做准备。

SPI 每次数据传输可以 8 位或 16 位为单位，每次传输的单位数不受限制。

3. CPOL/CPHA 及通讯模式

上面讲述的图 1-2 中的时序只是 SPI 中的其中一种通讯模式，SPI 一共有四种通讯模式，它们的主要区别是总线空闲时 SCK 的时钟状态以及数据采样时刻。为方便说明，在此引入“时钟极性 CPOL”和“时钟相位 CPHA”的概念。

时钟极性 CPOL 是指 SPI 通讯设备处于空闲状态时，SCK 信号线的电平信号(即 SPI 通讯开始前、 NSS 线为高电平时 SCK 的状态)。CPOL=0 时， SCK 在空闲状态时为低电平，CPOL=1 时，则相反。

时钟相位 CPHA 是指数据的采样的时刻，当 CPHA=0 时，MOSI 或 MISO 数据线上的信号将会在 SCK 时钟线的“奇数边沿”被采样。当 CPHA=1 时，数据线在 SCK 的“偶数边沿”采样。见图 1-3 及图 1-4。

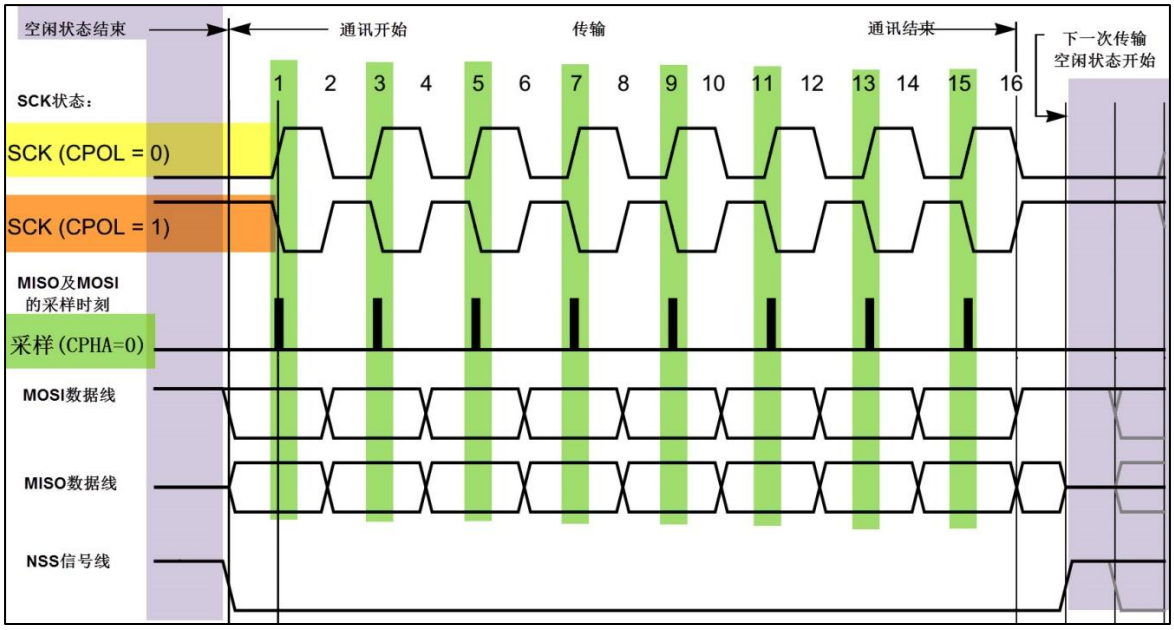

图 1-3 CPHA=0 时的 SPI 通讯模式

我们来分析这个 CPHA=0 的时序图。首先，根据 SCK 在空闲状态时的电平，分为两种情况。SCK 信号线在空闲状态为低电平时，CPOL=0；空闲状态为高电平时，CPOL=1。

无论 CPOL=0 还是=1，因为我们配置的时钟相位 CPHA=0，在图中可以看到，采样时刻都是在 SCK 的奇数边沿。注意当 CPOL=0 的时候，时钟的奇数边沿是上升沿，而 CPOL=1 的时候，时钟的奇数边沿是下降沿。所以 SPI 的采样时刻不是由上升/下降沿决定的。MOSI 和 MISO 数据线的有效信号在 SCK 的奇数边沿保持不变，数据信号将在 SCK 奇数边沿时被采样，在非采样时刻，MOSI 和 MISO 的有效信号才发生切换。

类似地，当 CPHA=1 时，不受 CPOL 的影响，数据信号在 SCK 的偶数边沿被采样，见图 1-4。

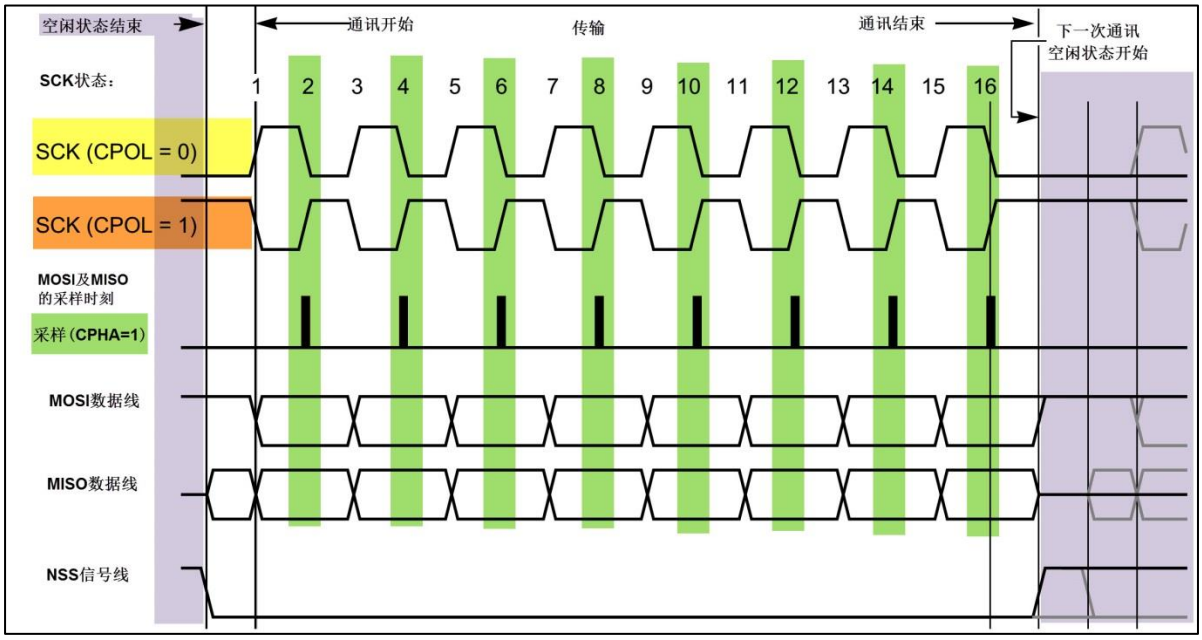

图 1-4 CPHA=1 时的 SPI 通讯模式

由 CPOL 及 CPHA 的不同状态，SPI 分成了四种模式，见表 1-1，主机与从机需要工作在相同的模式下才可以正常通讯，实际中采用较多的是“模式 0”与“模式 3”。

表 1-1 SPI 的四种模式

SPI 模式	CPOL	CPHA	空闲时 SCK 时钟	采样时刻
0	0	0	低电平	奇数边沿
1	0	1	低电平	偶数边沿
2	1	0	高电平	奇数边沿
3	1	1	高电平	偶数边沿

1.2 STM32 的 SPI 特性及架构

与 I2C 外设一样，STM32 芯片也集成了专门用于 SPI 协议通讯的外设。

1.2.1 STM32 的 SPI 外设简介

STM32 的 SPI 外设可用作通讯的主机及从机，支持最高的 SCK 时钟频率为 $f_{pclk}/2$ (STM32F103 型号的芯片默认 f_{pclk1} 为 72MHz，f_{pclk2} 为 36MHz)，完全支持 SPI 协议的 4 种模式，数据帧长度可设置为 8 位或 16 位，可设置数据 MSB 先行或 LSB 先行。它还支持双线全双工(前面小节说明的都是这种模式)、双线单向以及单线模式。其中双线单向模式可以同时使用 MOSI 及 MISO 数据线向一个方向传输数据，可以加快一倍的传输速度。而单线模式则可以减少硬件接线，当然这样速率会受到影响。我们只讲解双线全双工模式。

1.2.2 STM32 的 SPI 架构剖析

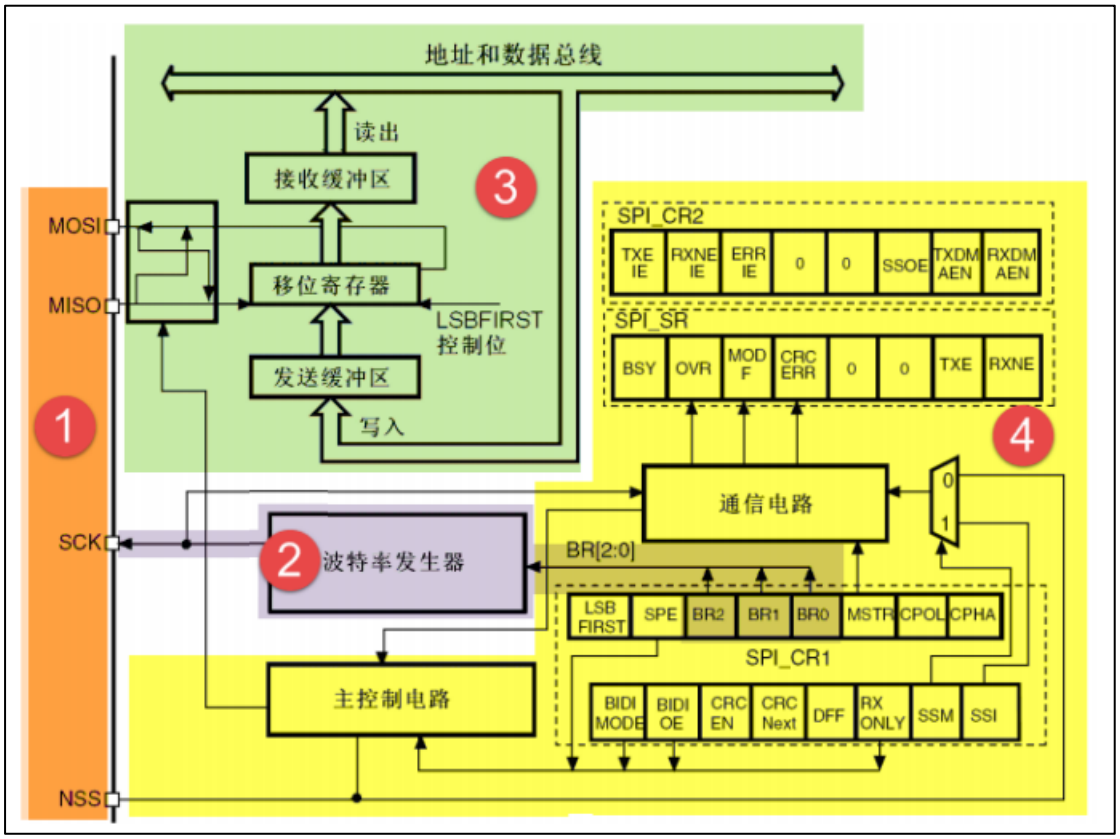

图 1-5 SPI 架构图

1. 通讯引脚

SPI 的所有硬件架构都从图 1-5 中左侧 MOSI、MISO、SCK 及 NSS 线展开的。STM32 芯片有多个 SPI 外设，它们的 SPI 通讯信号引出到不同的 GPIO 引脚上，使用时必须配置到这些指定的引脚，见表 1-2。关于 GPIO 引脚的复用功能，可查阅《STM32F10x 规格书》，以它为准。

表 1-2 STM32F4xx 的 SPI 引脚(整理自《STM32F10x 规格书》)

引脚	SPI 编号		
	SPI1	SPI2	SPI3
NSS	PA4	PB12	PA15 下载口的 TDI
CLK	PA5	PB13	PB3 下载口的 TDO
MISO	PA6	PB14	PB4 下载口的 NTRST
MOSI	PA7	PB15	PB5

其中 SPI1 是 APB2 上的设备，最高通信速率达 36Mbtis/s，SPI2、SPI3 是 APB1 上的设备，最高通信速率为 18Mbits/s 其它功能上没有差异。其中 SPI3 用到了下载接口的引脚，这几个引脚默认功能是下载，第二功能才是 IO 口，如果想使用 SPI3 接口，则程序上必须先禁用掉这几个 IO 口的下载功能先。一般的情况下，这几个 IO 口我们是专门用来下载和调试程序用，不会复用为 SPI3。

2. 时钟控制逻辑

SCK 线的时钟信号，由波特率发生器根据“控制寄存器 CR1”中的 BR[0:2]位控制，该位是对 f_{pclk} 时钟的分频因子，对 f_{pclk} 的分频结果就是 SCK 引脚的输出时钟频率，计算方法见表 1-3。

表 1-3 BR 位对 fpclk 的分频

BR[0:2]	分频结果(SCK 频率)	BR[0:2]	分频结果(SCK 频率)
000	$f_{pclk}/2$	100	$f_{pclk}/32$
001	$f_{pclk}/4$	101	$f_{pclk}/64$
010	$f_{pclk}/8$	110	$f_{pclk}/128$
011	$f_{pclk}/16$	111	$f_{pclk}/256$

其中的 f_{pclk} 频率是指 SPI 所在的 APB 总线频率，APB1 为 f_{pclk1}，APB2 为 f_{pckl2}。

通过配置“控制寄存器 CR”的“CPOL 位”及“CPHA”位可以把 SPI 设置成前面分析的 4 种 SPI 模式。

3. 数据控制逻辑

SPI 的 MOSI 及 MISO 都连接到数据移位寄存器上，数据移位寄存器的数据来源及目标接收、发送缓冲区以及 MISO、MOSI 线。当向外发送数据的时候，数据移位寄存器以“发送缓冲区”为数据源，把数据一位一位地通过数据线发送出去；当从外部接收数据的时候，数据移位寄存器把数据线采样到的数据一位一位地存储到“接收缓冲区”中。通过写 SPI 的“数据寄存器 DR”把数据填充到发送缓冲区中，通讯读“数据寄存器 DR”，可以获取接收缓冲区中的内容。其中数据帧长度可以通过“控制寄存器 CR1”的“DFF 位”配置成 8 位及 16 位模式；配置“LSBFIRST 位”可选择 MSB 先行还是 LSB 先行。

4. 整体控制逻辑

整体控制逻辑负责协调整个 SPI 外设，控制逻辑的工作模式根据我们配置的“控制寄存器(CR1/CR2)”的参数而改变，基本的控制参数包括前面提到的 SPI 模式、波特率、LSB 先行、主从模式、单双向模式等等。在外设工作时，控制逻辑会根据外设的工作状态修改“状态寄存器(SR)”，我们只要读取状态寄存器相关的寄存器位，就可以了解 SPI 的工作状态了。除此之外，控制逻辑还根据要求，负责控制产生 SPI 中断信号、DMA 请求及控制 NSS 信号线。

实际应用中，我们一般不使用 STM32 SPI 外设的标准 NSS 信号线，而是更简单地使用普通的 GPIO，软件控制它的电平输出，从而产生通讯起始和停止信号。

1.2.3 通讯过程

STM32 使用 SPI 外设通讯时，在通讯的不同阶段它会对“状态寄存器 SR”的不同数据位写入参数，我们通过读取这些寄存器标志来了解通讯状态。

图 1-6 中的是“主模式”流程，即 STM32 作为 SPI 通讯的主机端时的数据收发过程。

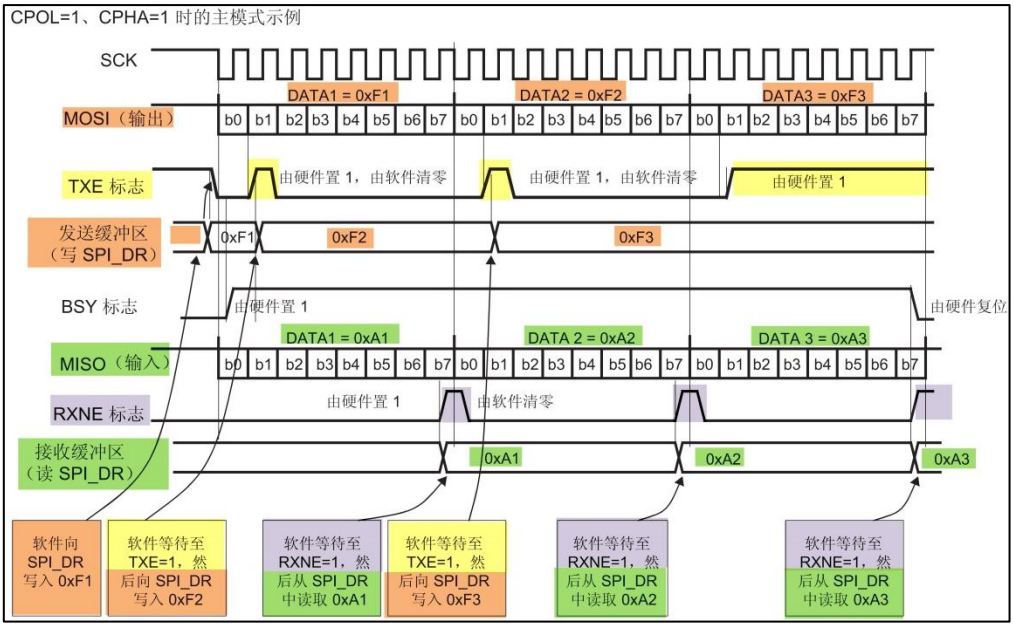

图 1-6 主发送器通讯过程

主模式收发流程及事件说明如下：

(1) 控制 NSS 信号线，产生起始信号(图中没有画出)；

(2) 把要发送的数据写入到“数据寄存器 DR”中，该数据会被存储到发送缓冲区；

(3) 通讯开始，SCK 时钟开始运行。MOSI 把发送缓冲区中的数据一位一位地传输出去；MISO 则把数据一位一位地存储进接收缓冲区中；

(4) 当发送完一帧数据的时候，“状态寄存器 SR”中的“TXE 标志位”会被置 1，表示传输完一帧，发送缓冲区已空；类似地，当接收完一帧数据的时候，“RXNE 标志位”会被置 1，表示传输完一帧，接收缓冲区非空；

(5) 等待到“TXE 标志位”为 1 时，若还要继续发送数据，则再次往“数据寄存器 DR”写入数据即可；等待到“RXNE 标志位”为 1 时，通过读取“数据寄存器 DR”可以获取接收缓冲区中的内容。

假如我们使能了 TXE 或 RXNE 中断，TXE 或 RXNE 置 1 时会产生 SPI 中断信号，进入同一个中断服务函数，到 SPI 中断服务程序后，可通过检查寄存器位来了解是哪一个事件，再分别进行处理。也可以使用 DMA 方式来收发“数据寄存器 DR”中的数据。

1.3 SPI 初始化结构体详解

跟其它外设一样，STM32 标准库提供了 SPI 初始化结构体及初始化函数来配置 SPI 外设。初始化结构体及函数定义在库文件“stm32f4xx_spi.h”及“stm32f4xx_spi.c”中，编程时我们可以结合这两个文件内的注释使用或参考库帮助文档。了解初始化结构体后我们就能对 SPI 外设运用自如了，见代码清单 1-1。

代码清单 1-1 SPI 初始化结构体

```
1 typedef struct
 2 {
 3     uint16_t SPI_Direction;           /*设置 SPI 的单双向模式 */
 4     uint16_t SPI_Mode;                /*设置 SPI 的主/从机端模式 */
 5     uint16_t SPI_DataSize;            /*设置 SPI 的数据帧长度，可选 8/16 位 */
```

```
 6     uint16_t SPI_CPOL;                 /*设置时钟极性 CPOL，可选高/低电平*/
 7     uint16_t SPI_CPHA;                 /*设置时钟相位，可选奇/偶数边沿采样 */
 8     uint16_t SPI_NSS;                 /*设置 NSS 引脚由 SPI 硬件控制还是软件控制*/
 9     uint16_t SPI_BaudRatePrescaler;   /*设置时钟分频因子，fpclk/分频数=fSCK */
10     uint16_t SPI_FirstBit;             /*设置 MSB/LSB 先行 */
11     uint16_t SPI_CRCPolynomial;        /*设置 CRC 校验的表达式 */
12 } SPI_InitTypeDef;
```

这些结构体成员说明如下，其中括号内的文字是对应参数在 STM32 标准库中定义的宏：

(1) SPI_Direction

本成员设置 SPI 的通讯方向，可设置为双线全双工(SPI_Direction_2Lines_FullDuplex)，双线只接收(SPI_Direction_2Lines_RxOnly)，单线只接收(SPI_Direction_1Line_Rx)、单线只发送模式(SPI_Direction_1Line_Tx)。

(2) SPI_Mode

本成员设置 SPI 工作在主机模式(SPI_Mode_Master)或从机模式(SPI_Mode_Slave)，这两个模式的最大区别为 SPI 的 SCK 信号线的时序，SCK 的时序是由通讯中的主机产生的。若被配置为从机模式，STM32 的 SPI 外设将接受外来的 SCK 信号。

(3) SPI_DataSize

本成员可以选择 SPI 通讯的数据帧大小是为 8 位(SPI_DataSize_8b)还是 16 位(SPI_DataSize_16b)。

(4) SPI_CPOL 和 SPI_CPHA

这两个成员配置 SPI 的时钟极性 CPOL 和时钟相位 CPHA，这两个配置影响到 SPI 的通讯模式，关于 CPOL 和 CPHA 的说明参考前面“通讯模式”小节。

时钟极性 CPOL 成员，可设置为高电平(SPI_CPOL_High)或低电平(SPI_CPOL_Low)。

时钟相位 CPHA 则可以设置为 SPI_CPHA_1Edge(在 SCK 的奇数边沿采集数据) 或 SPI_CPHA_2Edge (在 SCK 的偶数边沿采集数据) 。

(5) SPI_NSS

本成员配置 NSS 引脚的使用模式，可以选择为硬件模式(SPI_NSS_Hard)与软件模式(SPI_NSS_Soft)，在硬件模式中的 SPI 片选信号由 SPI 硬件自动产生，而软件模式则需要我们亲自把相应的 GPIO 端口拉高或置低产生非片选和片选信号。实际中软件模式应用比较多。

(6) SPI_BaudRatePrescaler

本成员设置波特率分频因子，分频后的时钟即为 SPI 的 SCK 信号线的时钟频率。这个成员参数可设置为 fpclk 的 2、4、6、8、16、32、64、128、256 分频。

(7) SPI_FirstBit

所有串行的通讯协议都会有 MSB 先行(高位数据在前)还是 LSB 先行(低位数据在前)的问题，而 STM32 的 SPI 模块可以通过这个结构体成员，对这个特性编程控制。

(8) SPI_CRCPolynomial

这是 SPI 的 CRC 校验中的多项式，若我们使用 CRC 校验时，就使用这个成员的参数(多项式)，来计算 CRC 的值。

配置完这些结构体成员后，我们要调用 SPI_Init 函数把这些参数写入到寄存器中，实现 SPI 的初始化，然后调用 SPI_Cmd 来使能 SPI 外设。

1.4 SPI—读写串行 FLASH 实验

FLSAH 存储器又称闪存，它与 EEPROM 都是掉电后数据不丢失的存储器，但 FLASH 存储器容量普遍大于 EEPROM，现在基本取代了它的地位。我们生活中常用的 U 盘、SD 卡、SSD 固态硬盘以及我们 STM32 芯片内部用于存储程序的设备，都是 FLASH 类型的存储器。在存储控制上，最主要的区别是 FLASH 芯片只能一大片一大片地擦写，而在“I2C 章节”中我们了解到 EEPROM 可以单个字节擦写。

本小节以一种使用 SPI 通讯的串行 FLASH 存储芯片的读写实验为大家讲解 STM32 的 SPI 使用方法。实验中 STM32 的 SPI 外设采用主模式，通过查询事件的方式来确保正常通讯。

1.4.1 硬件设计

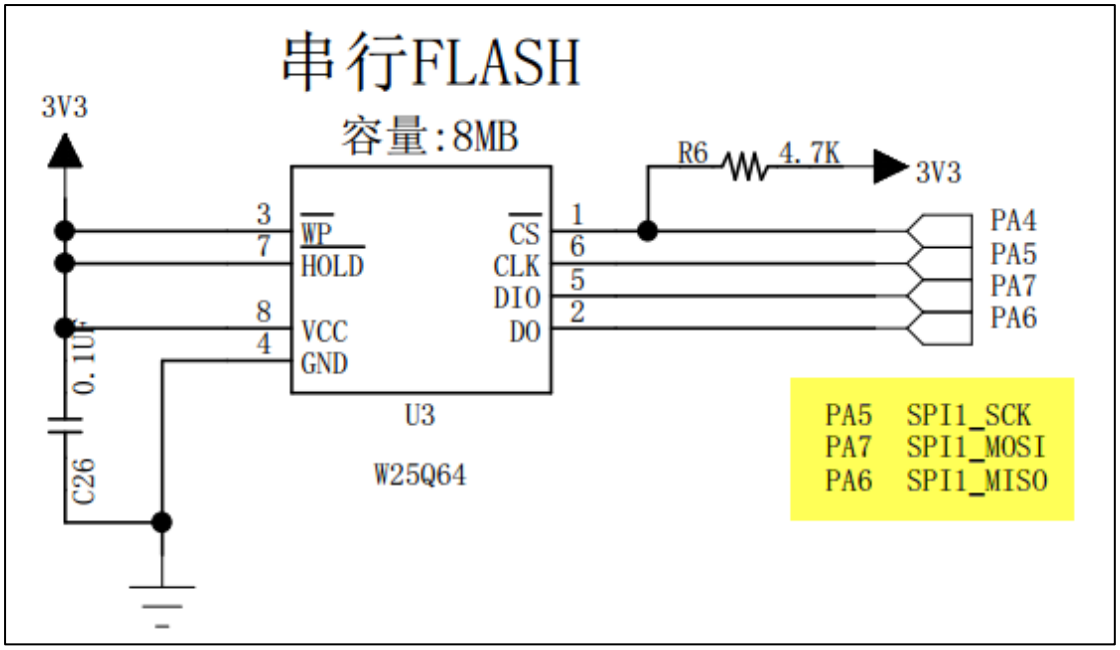

图 1-7 SPI 串行 FLASH 硬件连接图

本实验板中的 FLASH 芯片(型号：W25Q64)是一种使用 SPI 通讯协议的 NOR FLASH 存储器，它的 CS/CLK/DIO/DO 引脚分别连接到了 STM32 对应的 SPI 引脚

NSS/SCK/MOSI/MISO 上，其中 STM32 的 NSS 引脚是一个普通的 GPIO，不是 SPI 的专用 NSS 引脚，所以程序中我们要使用软件控制的方式。

FLASH 芯片中还有 WP 和 HOLD 引脚。WP 引脚可控制写保护功能，当该引脚为低电平时，禁止写入数据。我们直接接电源，不使用写保护功能。HOLD 引脚可用于暂停通讯，该引脚为低电平时，通讯暂停，数据输出引脚输出高阻抗状态，时钟和数据输入引脚无效。我们直接接电源，不使用通讯暂停功能。

关于 FLASH 芯片的更多信息，可参考其数据手册《W25Q64》来了解。若您使用的实验板 FLASH 的型号或控制引脚不一样，只需根据我们的工程修改即可，程序的控制原理相同。

1.4.2 软件设计

为了使工程更加有条理，我们把读写 FLASH 相关的代码独立分开存储，方便以后移植。在“工程模板”之上新建“bsp_spi_flash.c”及“bsp_spi_ flash.h”文件，这些文件也可根据您的喜好命名，它们不属于 STM32 标准库的内容，是由我们自己根据应用需要编写的。

1. 编程要点

(1) 初始化通讯使用的目标引脚及端口时钟；
(2) 使能 SPI 外设的时钟；
(3) 配置 SPI 外设的模式、地址、速率等参数并使能 SPI 外设；
(4) 编写基本 SPI 按字节收发的函数；
(5) 编写对 FLASH 擦除及读写操作的的函数；
(6) 编写测试程序，对读写数据进行校验。

2. 代码分析

SPI 硬件相关宏定义

我们把 SPI 硬件相关的配置都以宏的形式定义到 “bsp_spi_ flash.h”文件中，见代码清单 1-2。

代码清单 1-2 SPI 硬件配置相关的宏

```
/*SPI 接口定义-开头****************************/
#define      SPIx                            SPI1
#define      SPI_APBxClock_FUN               RCC_APB2PeriphClockCmd
#define      SPI_CLK                         RCC_APB2Periph_SPI1
//CS(NSS)引脚 片选选普通 GPIO 即可
#define      SPI_CS_APBxClock_FUN            RCC_APB2PeriphClockCmd
#define      SPI_CS_CLK                      RCC_APB2Periph_GPIOA
#define      SPI_CS_PORT                     GPIOA
#define      SPI_CS_PIN                      GPIO_Pin_4
//SCK 引脚
#define      SPI_SCK_APBxClock_FUN           RCC_APB2PeriphClockCmd
#define      SPI_SCK_CLK                     RCC_APB2Periph_GPIOA
#define      SPI_SCK_PORT                    GPIOA
#define      SPI_SCK_PIN                     GPIO_Pin_5
//MISO 引脚
```

```
#define      SPI_MISO_APBxClock_FUN      RCC_APB2PeriphClockCmd
#define      SPI_MISO_CLK                RCC_APB2Periph_GPIOA
#define      SPI_MISO_PORT               GPIOA
#define      SPI_MISO_PIN                GPIO_Pin_6
//MOSI 引脚
#define      SPI_MOSI_APBxClock_FUN      RCC_APB2PeriphClockCmd
#define      SPI_MOSI_CLK                RCC_APB2Periph_GPIOA
#define      SPI_MOSI_PORT               GPIOA
#define      SPI_MOSI_PIN                GPIO_Pin_7

#define  SPI_FLASH_CS_LOW()     GPIO_ResetBits( SPI_CS_PORT, SPI_CS_PIN )
#define  SPI_FLASH_CS_HIGH()    GPIO_SetBits( SPI_CS_PORT, SPI_CS_PIN )

/*SPI 接口定义-结尾****************************/
```

以上代码根据硬件连接，把与 FLASH 通讯使用的 SPI 号 、GPIO 等都以宏封装起来，并且定义了控制 CS(NSS)引脚输出电平的宏，以便配置产生起始和停止信号时使用。

初始化 SPI 的 GPIO

利用上面的宏，编写 SPI 的初始化函数，见代码清单 1-3。

代码清单 1-3 SPI 的初始化函数(GPIO 初始化部分)

```
/**
 * @brief  SPI_FLASH 初始化
 * @param  无
 * @retval 无
 */
void SPI_FLASH_Init(void)
{
    SPI_InitTypeDef  SPI_InitStructure;
    GPIO_InitTypeDef GPIO_InitStructure;

    /* 使能 SPI 时钟 */
    SPI_APBxClock_FUN ( SPI_CLK, ENABLE );

    /* 配置 SPI 的 CS 引脚，普通 IO 即可 */
    SPI_CS_APBxClock_FUN ( SPI_CS_CLK, ENABLE );
    GPIO_InitStructure.GPIO_Pin = SPI_CS_PIN;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Init(SPI_CS_PORT, &GPIO_InitStructure);

    /* 配置 SPI 的 SCK 引脚*/
    SPI_SCK_APBxClock_FUN ( SPI_SCK_CLK, ENABLE );
    GPIO_InitStructure.GPIO_Pin = SPI_SCK_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init(SPI_SCK_PORT, &GPIO_InitStructure);

    /* 配置 SPI 的 MISO 引脚*/
    SPI_MISO_APBxClock_FUN ( SPI_MISO_CLK, ENABLE );
    GPIO_InitStructure.GPIO_Pin = SPI_MISO_PIN;
    GPIO_Init(SPI_MISO_PORT, &GPIO_InitStructure);

    /* 配置 SPI 的 MOSI 引脚*/
    SPI_MOSI_APBxClock_FUN ( SPI_MOSI_CLK, ENABLE );
    GPIO_InitStructure.GPIO_Pin = SPI_MOSI_PIN;
    GPIO_Init(SPI_MOSI_PORT, &GPIO_InitStructure);

    /* 停止信号 FLASH: CS 引脚高电平*/
    SPI_FLASH_CS_HIGH();

    //为方便讲解，以下省略 SPI 模式初始化部分
}
```

与所有使用到 GPIO 的外设一样，都要先把使用到的 GPIO 引脚模式初始化，配置好复用功能。GPIO 初始化流程如下：

(1) 使用 GPIO_InitTypeDef 定义 GPIO 初始化结构体变量，以便下面用于存储 GPIO 配置；

(2) 调用库函数 RCC_APB2PeriphClockCmd 来使能 SPI 引脚使用的 GPIO 端口时钟。

(3) 向 GPIO 初始化结构体赋值，把 SCK/MOSI/MISO 引脚初始化成复用推挽模式。而 CS(NSS)引脚由于使用软件控制，我们把它配置为普通的推挽输出模式。

(4) 使用以上初始化结构体的配置，调用 GPIO_Init 函数向寄存器写入参数，完成 GPIO 的初始化。

配置 SPI 的模式

以上只是配置了 SPI 使用的引脚，对 SPI 外设模式的配置。在配置 STM32 的 SPI 模式前，我们要先了解从机端的 SPI 模式。本例子中可通过查阅 FLASH 数据手册《W25Q64》获取。根据 FLASH 芯片的说明，它支持 SPI 模式 0 及模式 3，支持双线全双工，使用 MSB 先行模式，支持最高通讯时钟为 104MHz，数据帧长度为 8 位。我们要把 STM32 的 SPI 外设中的这些参数配置一致。见代码清单 1-4。

代码清单 1-4 配置 SPI 模式

```
/**
 * @brief  SPI_FLASH 引脚初始化
 * @param  无
 * @retval 无
 */
void SPI_FLASH_Init(void)
{
    /*为方便讲解，省略了 SPI 的 GPIO 初始化部分*/
    //......

    SPI_InitTypeDef  SPI_InitStructure;
    /* SPI 模式配置 */
    // FLASH 芯片 支持 SPI 模式 0 及模式 3，据此设置 CPOL CPHA
    SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
    SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
    SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
    SPI_InitStructure.SPI_CPOL = SPI_CPOL_High;
    SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge;
    SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
    SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_4;
    SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
    SPI_InitStructure.SPI_CRCPolynomial = 7;
    SPI_Init(FLASH_SPI, &SPI_InitStructure);

    /* 使能 SPI  */
    SPI_Cmd(SPIx, ENABLE);
}
```

这段代码中，把 STM32 的 SPI 外设配置为主机端，双线全双工模式，数据帧长度为 8 位，使用 SPI 模式 3(CPOL=1，CPHA=1)，NSS 引脚由软件控制以及 MSB 先行模式。最后

一个成员为 CRC 计算式，由于我们与 FLASH 芯片通讯不需要 CRC 校验，并没有使能 SPI 的 CRC 功能，这时 CRC 计算式的成员值是无效的。

赋值结束后调用库函数 SPI_Init 把这些配置写入寄存器，并调用 SPI_Cmd 函数使能外设。

使用 SPI 发送和接收一个字节的数据

初始化好 SPI 外设后，就可以使用 SPI 通讯了，复杂的数据通讯都是由单个字节数据收发组成的，我们看看它的代码实现，见代码清单 1-5。

代码清单 1-5 使用 SPI 发送和接收一个字节的数据

```
/**
 * @brief  使用 SPI 发送一个字节的数据
 * @param  byte：要发送的数据
 * @retval 返回接收到的数据
 */
u8 SPI_FLASH_SendByte(u8 byte)
{
    /* 等待发送缓冲区为空，TXE 事件 */
    while (SPI_I2S_GetFlagStatus(SPIx,SPI_I2S_FLAG_TXE) == RESET);

    /* 写入数据寄存器，把要写入的数据写入发送缓冲区 */
    SPI_I2S_SendData(SPIx , byte);

    /* 等待接收缓冲区非空，RXNE 事件 */
    while (SPI_I2S_GetFlagStatus(SPIx,SPI_I2S_FLAG_RXNE) == RESET);

    /* 读取数据寄存器，获取接收缓冲区数据 */
    return SPI_I2S_ReceiveData(SPIx );
}

#define Dummy_Byte     0xFF
/**
 * @brief  使用 SPI 读取一个字节的数据
 * @param  无
 * @retval 返回接收到的数据
 */
u8 SPI_FLASH_ReadByte(void)
{
    return (SPI_FLASH_SendByte(Dummy_Byte));
}
```

SPI_FLASH_SendByte 函数实现了前面讲的“SPI 通讯过程”：

(1) 本函数中不包含 SPI 起始和停止信号，只是收发的主要过程，所以在调用本函数前后要做好起始和停止信号的操作；

(2) 通过检测 TXE 标志，获取发送缓冲区的状态，若发送缓冲区为空，则表示可能存在的上一个数据已经发送完毕；

(3) 等待至发送缓冲区为空后，调用库函数 SPI_I2S_SendData 把要发送的数据“byte”写入到 SPI 的数据寄存器 DR，写入 SPI 数据寄存器的数据会存储到发送缓冲区，由 SPI 外设发送出去；

(4) 写入完毕后等待 RXNE 事件，即接收缓冲区非空事件。由于 SPI 双线全双工模式下 MOSI 与 MISO 数据传输是同步的(请对比“SPI 通讯过程”阅读)，当接收缓冲区非空时，表示上面的数据发送完毕，且接收缓冲区也收到新的数据；

(5) 等待至接收缓冲区非空时，通过调用库函数 SPI_I2S_ReceiveData 读取 SPI 的数据寄存器 DR，就可以获取接收缓冲区中的新数据了。代码中使用关键字“return”把接收到的这个数据作为 SPI_FLASH_SendByte 函数的返回值，所以我们可以看到在下面定义的 SPI 接收数据函数 SPI_FLASH_ReadByte，它只是简单地调用了 SPI_FLASH_SendByte 函数发送数据“Dummy_Byte”，然后获取其返回值(因为不关注发送的数据，所以此时的输入参数“Dummy_Byte”可以为任意值)。可以这样做的原因是 SPI 的接收过程和发送过程实质是一样的，收发同步进行，关键在于我们的上层应用中，关注的是发送还是接收的数据。

控制 FLASH 的指令

搞定 SPI 的基本收发单元后，还需要了解如何对 FLASH 芯片进行读写。FLASH 芯片自定义了很多指令，我们通过控制 STM32 利用 SPI 总线向 FLASH 芯片发送指令，FLASH 芯片收到后就会执行相应的操作。

而这些指令，对主机端(STM32)来说，只是它遵守最基本的 SPI 通讯协议发送出的数据，但在设备端(FLASH 芯片)把这些数据解释成不同的意义，所以才成为指令。查看 FLASH 芯片的数据手册《W25Q128》，可了解各种它定义的各种指令的功能及指令格式，见表 1-4。

表 1-4 FLASH 常用芯片指令表(摘自规格书《W25Q128》)

指令	第一字节(指令编码)	第二字节	第三字节	第四字节	第五字节	第六字节	第七-N 字节
Write Enable	06h						
Write Disable	04h						
Read Status Register	05h	(S7–S0)					
Write Status Register	01h	(S7–S0)					
Read Data	03h	A23–A16	A15–A8	A7–A0	(D7–D0)	(Next byte)	continuous
Fast Read	0Bh	A23–A16	A15–A8	A7–A0	dummy	(D7–D0)	(Next Byte) continuous
Fast Read Dual Output	3Bh	A23–A16	A15–A8	A7–A0	dummy	I/O = (D6,D4,D2,D0) O = (D7,D5,D3,D1)	(one byte per 4 clocks, continuous)
Page Program	02h	A23–A16	A15–A8	A7–A0	D7–D0	Next byte	Up to 256 bytes

Block Erase(64KB)	D8h	A23–A16	A15–A8	A7–A0			
Sector Erase(4KB)	20h	A23–A16	A15–A8	A7–A0			
Chip Erase	C7h						
Power-down	B9h						
Release Power-down / Device ID	ABh	dummy	dummy	dummy	(ID7-ID0)		
Manufacturer/ Device ID	90h	dummy	dummy	00h	(M7-M0)	(ID7-ID0)	
JEDEC ID	9Fh	(M7-M0) 生产厂商	(ID15-ID8) 存储器类型	(ID7-ID0) 容量			

该表中的第一列为指令名，第二列为指令编码，第三至第 N 列的具体内容根据指令的不同而有不同的含义。其中带括号的字节参数，方向为 FLASH 向主机传输，即命令响应，不带括号的则为主机向 FLASH 传输。表中“A0~A23”指 FLASH 芯片内部存储器组织的地址；“M0~M7”为厂商号（MANUFACTURER ID）；“ID0-ID15”为 FLASH 芯片的 ID；“dummy”指该处可为任意数据；“D0~D7”为 FLASH 内部存储矩阵的内容。

在 FLSAH 芯片内部，存储有固定的厂商编号(M7-M0)和不同类型 FLASH 芯片独有的编号(ID15-ID0)，见表 1-5。

表 1-5 FLASH 数据手册的设备 ID 说明

FLASH 型号	厂商号(M7-M0)	FLASH 型号(ID15-ID0)
W25Q64	EF h	4017 h
W25Q128	EF h	4018 h

通过指令表中的读 ID 指令“JEDEC ID”可以获取这两个编号，该指令编码为“9F h”，其中“9F h”是指 16 进制数“9F” (相当于 C 语言中的 0x9F)。紧跟指令编码的三个字节分别为 FLASH 芯片输出的“(M7-M0)”、“(ID15-ID8)”及“(ID7-ID0)” 。

此处我们以该指令为例，配合其指令时序图进行讲解，见图 1-8。

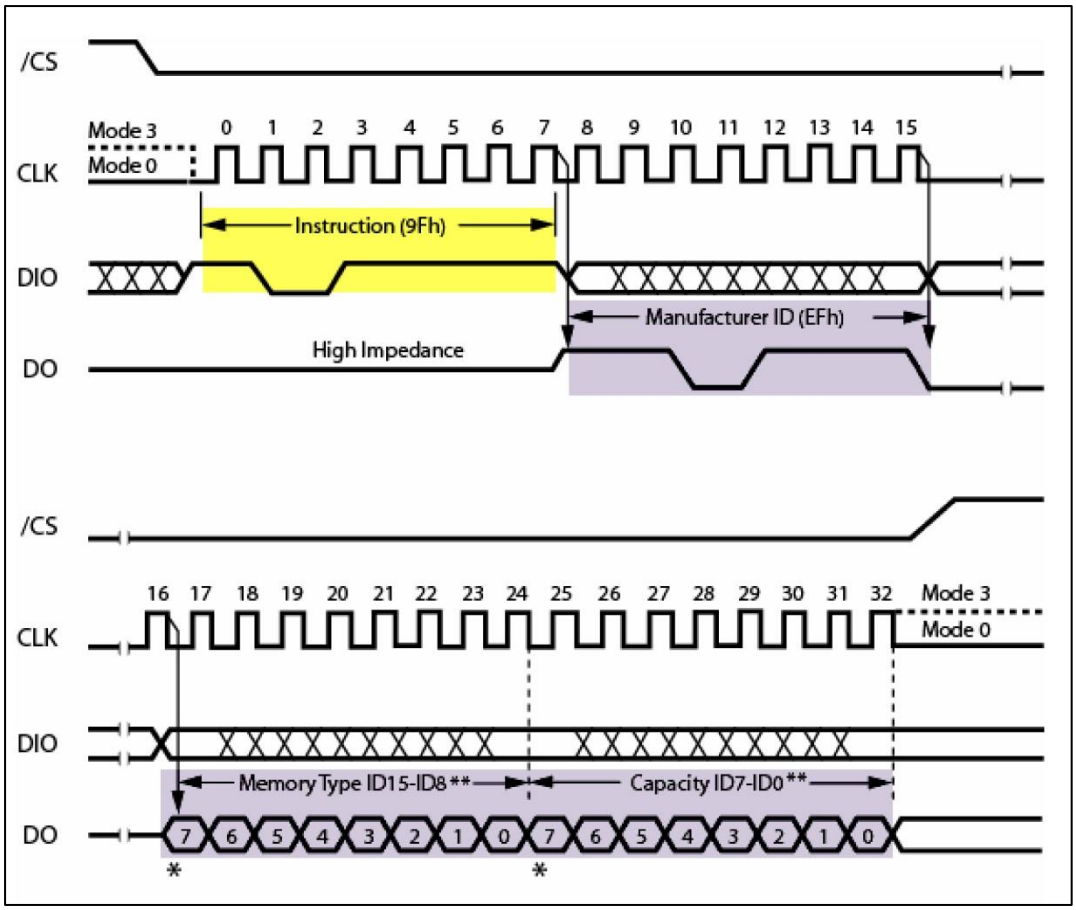

图 1-8 FLASH 读 ID 指令“JEDEC ID”的时序(摘自规格书《W25Q64》)

主机首先通过 MOSI 线向 FLASH 芯片发送第一个字节数据为“9F h”，当 FLASH 芯片收到该数据后，它会解读成主机向它发送了“JEDEC 指令”，然后它就作出该命令的响应：通过 MISO 线把它的厂商 ID(M7-M0)及芯片类型(ID15-0)发送给主机，主机接收到指令响应后可进行校验。常见的应用是主机端通过读取设备 ID 来测试硬件是否连接正常，或用于识别设备。

对于 FLASH 芯片的其它指令，都是类似的，只是有的指令包含多个字节，或者响应包含更多的数据。

实际上，编写设备驱动都是有一定的规律可循的。首先我们要确定设备使用的是什么通讯协议。如上一章的 EEPROM 使用的是 I^2C，本章的 FLASH 使用的是 SPI。那么我们就先根据它的通讯协议，选择好 STM32 的硬件模块，并进行相应的 I^2C 或 SPI 模块初始化。接着，我们要了解目标设备的相关指令，因为不同的设备，都会有相应的不同的指令。如 EEPROM 中会把第一个数据解释为内部存储矩阵的地址(实质就是指令)。而 FLASH 则定义了更多的指令，有写指令，读指令，读 ID 指令等等。最后，我们根据这些指令的格式要求，使用通讯协议向设备发送指令，达到控制设备的目标。

定义 FLASH 指令编码表

为了方便使用，我们把 FLASH 芯片的常用指令编码使用宏来封装起来，后面需要发送指令编码的时候我们直接使用这些宏即可，见代码清单 1-6。

代码清单 1-6 FLASH 指令编码表

```
/*FLASH 常用命令*/
#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
/*其它*/
#define  sFLASH_ID                      0XEF4017
#define Dummy_Byte                      0xFF
```

读取 FLASH 芯片 ID

根据“JEDEC”指令的时序，我们把读取 FLASH ID 的过程编写成一个函数，见代码清单 1-7。

代码清单 1-7 读取 FLASH 芯片 ID

```
/**
  * @brief  读取 FLASH ID
  * @param  无
  * @retval FLASH ID
  */
u32 SPI_FLASH_ReadID(void)
{
    u32 Temp = 0, Temp0 = 0, Temp1 = 0, Temp2 = 0;

    /* 开始通讯：CS 低电平 */
    SPI_FLASH_CS_LOW();

    /* 发送 JEDEC 指令，读取 ID */
    SPI_FLASH_SendByte(W25X_JedecDeviceID);

    /* 读取一个字节数据 */
    Temp0 = SPI_FLASH_SendByte(Dummy_Byte);

    /* 读取一个字节数据 */
    Temp1 = SPI_FLASH_SendByte(Dummy_Byte);

    /* 读取一个字节数据 */
    Temp2 = SPI_FLASH_SendByte(Dummy_Byte);

    /* 停止通讯：CS 高电平 */
    SPI_FLASH_CS_HIGH();

    /*把数据组合起来，作为函数的返回值*/
    Temp = (Temp0 << 16) | (Temp1 << 8) | Temp2;

    return Temp;
}
```

这段代码利用控制 CS 引脚电平的宏“SPI_FLASH_CS_LOW/HIGH”以及前面编写的单字节收发函数 SPI_FLASH_SendByte，很清晰地实现了“JEDEC ID”指令的时序：发送一个字节的指令编码“W25X_JedecDeviceID”，然后读取 3 个字节，获取 FLASH 芯片对

该指令的响应，最后把读取到的这 3 个数据合并到一个变量 Temp 中，然后作为函数返回值，把该返回值与我们定义的宏“sFLASH_ID”对比，即可知道 FLASH 芯片是否正常。

FLASH 写使能以及读取当前状态

在向 FLASH 芯片存储矩阵写入数据前，首先要使能写操作，通过“Write Enable”命令即可写使能，见代码清单 1-8。

代码清单 1-8 写使能命令

```
/**
 * @brief  向 FLASH 发送 写使能 命令
 * @param  none
 * @retval none
 */
void SPI_FLASH_WriteEnable(void)
{
    /* 通讯开始：CS 低 */
    SPI_FLASH_CS_LOW();

    /* 发送写使能命令*/
    SPI_FLASH_SendByte(W25X_WriteEnable);

    /*通讯结束：CS 高 */
    SPI_FLASH_CS_HIGH();
}
```

与 EEPROM 一样，由于 FLASH 芯片向内部存储矩阵写入数据需要消耗一定的时间，并不是在总线通讯结束的一瞬间完成的，所以在写操作后需要确认 FLASH 芯片“空闲”时才能进行再次写入。为了表示自己的工作状态，FLASH 芯片定义了一个状态寄存器，见图 1-9。

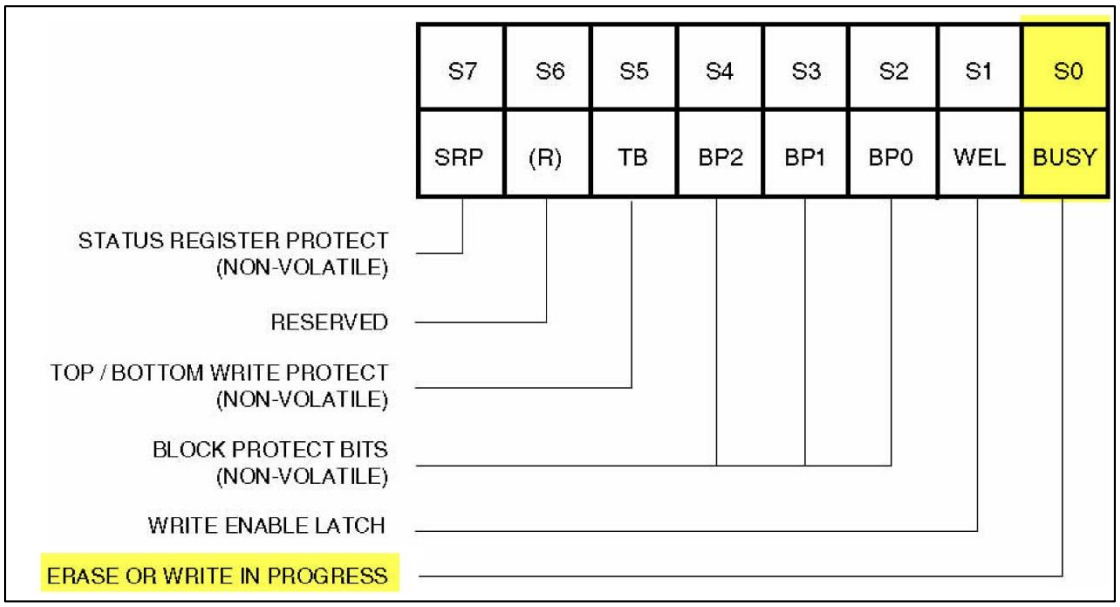

图 1-9 FLASH 芯片的状态寄存器

我们只关注这个状态寄存器的第 0 位“BUSY”，当这个位为“1”时，表明 FLASH 芯片处于忙碌状态，它可能正在对内部的存储矩阵进行“擦除”或“数据写入”的操作。

利用指令表中的“Read Status Register”指令可以获取 FLASH 芯片状态寄存器的内容，其时序见图 1-10。

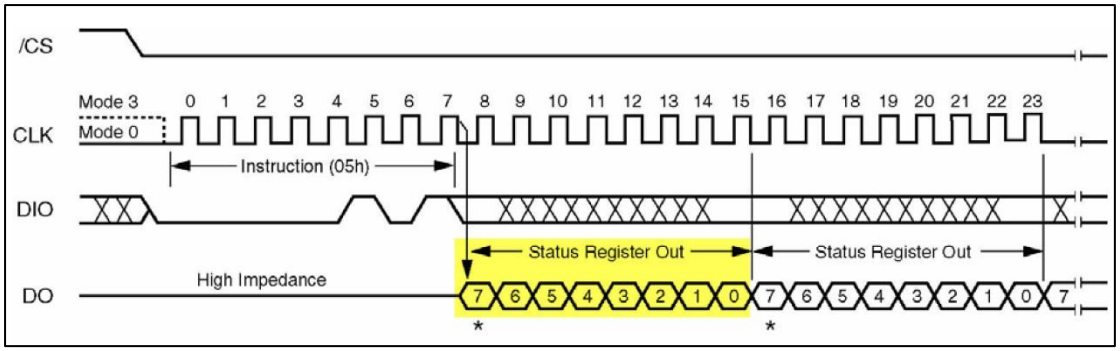

图 1-10 读取状态寄存器的时序

只要向 FLASH 芯片发送了读状态寄存器的指令，FLASH 芯片就会持续向主机返回最新的状态寄存器内容，直到收到 SPI 通讯的停止信号。据此我们编写了具有等待 FLASH 芯片写入结束功能的函数，见代码清单 1-9。

代码清单 1-9 通过读状态寄存器等待 FLASH 芯片空闲

```
/* WIP(busy)标志，FLASH 内部正在写入 */
#define WIP_Flag                  0x01

/**
 * @brief  等待 WIP(BUSY)标志被置 0，即等待到 FLASH 内部数据写入完毕
 * @param  none
 * @retval none
 */
void SPI_FLASH_WaitForWriteEnd(void)
{
    u8 FLASH_Status = 0;

    /* 选择 FLASH: CS 低 */
    SPI_FLASH_CS_LOW();

    /* 发送 读状态寄存器 命令 */
    SPI_FLASH_SendByte(W25X_ReadStatusReg);

    /* 若 FLASH 忙碌，则等待 */
    do
    {
        /* 读取 FLASH 芯片的状态寄存器 */
        FLASH_Status = SPI_FLASH_SendByte(Dummy_Byte);
    }
    while ((FLASH_Status & WIP_Flag) == SET);  /* 正在写入标志 */

    /* 停止信号  FLASH: CS 高 */
    SPI_FLASH_CS_HIGH();
}
```

这段代码发送读状态寄存器的指令编码“W25X_ReadStatusReg”后，在 while 循环里持续获取寄存器的内容并检验它的“WIP_Flag 标志”(即 BUSY 位)，一直等待到该标志表示写入结束时才退出本函数，以便继续后面与 FLASH 芯片的数据通讯。

FLASH 扇区擦除

由于 FLASH 存储器的特性决定了它只能把原来为“1”的数据位改写成“0”，而原来为“0”的数据位不能直接改写为“1”。所以这里涉及到数据“擦除”的概念，在写入前，必须要对目标存储矩阵进行擦除操作，把矩阵中的数据位擦除为“1”，在数据写入的时候，如果要存储数据“1”，那就不修改存储矩阵 ，在要存储数据“0”时，才更改该位。

通常，对存储矩阵擦除的基本操作单位都是多个字节进行，如本例子中的 FLASH 芯片支持“扇区擦除”、“块擦除”以及“整片擦除”，见表 1-6。

表 1-6 本实验 FLASH 芯片的擦除单位

擦除单位	大小
扇区擦除 Sector Erase	4KB
块擦除 Block Erase	64KB
整片擦除 Chip Erase	整个芯片完全擦除

FLASH 芯片的最小擦除单位为扇区(Sector)，而一个块(Block)包含 16 个扇区，其内部存储矩阵分布见图 1-11。。

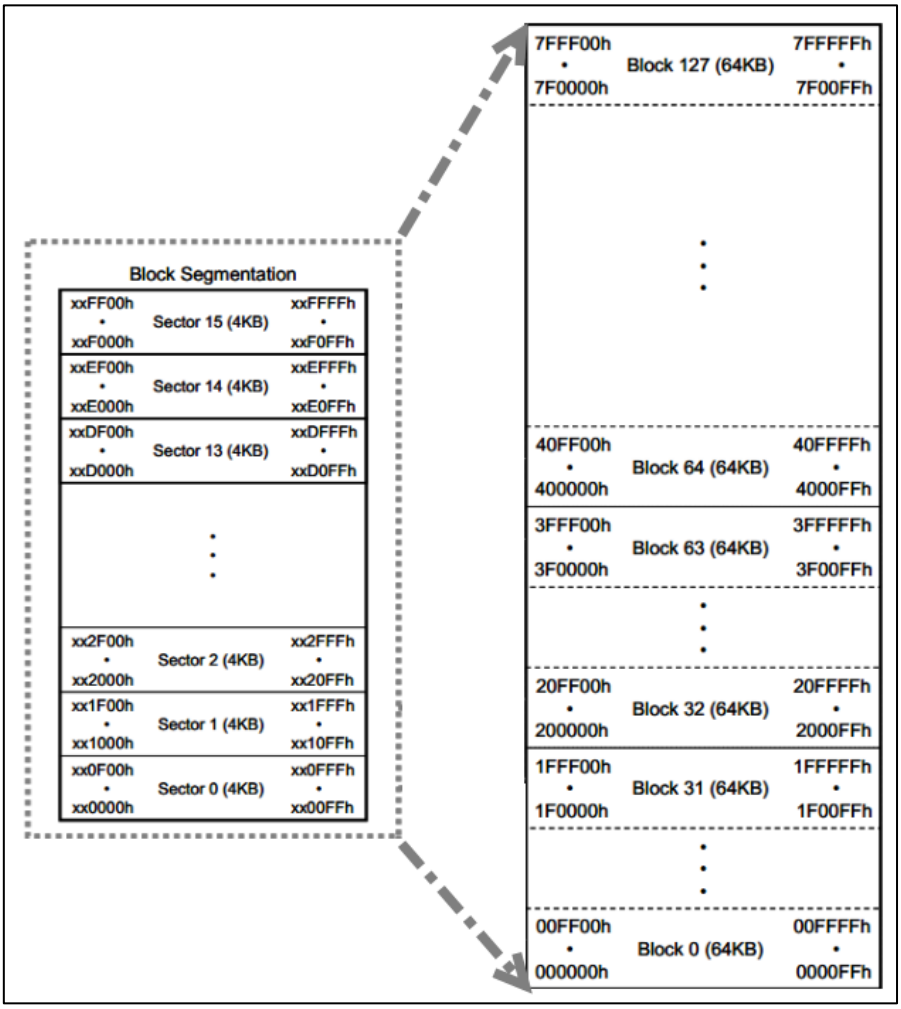

图 1-11 FLASH 芯片的存储矩阵

使用扇区擦除指令“Sector Erase”可控制 FLASH 芯片开始擦写，其指令时序见图 1-14。

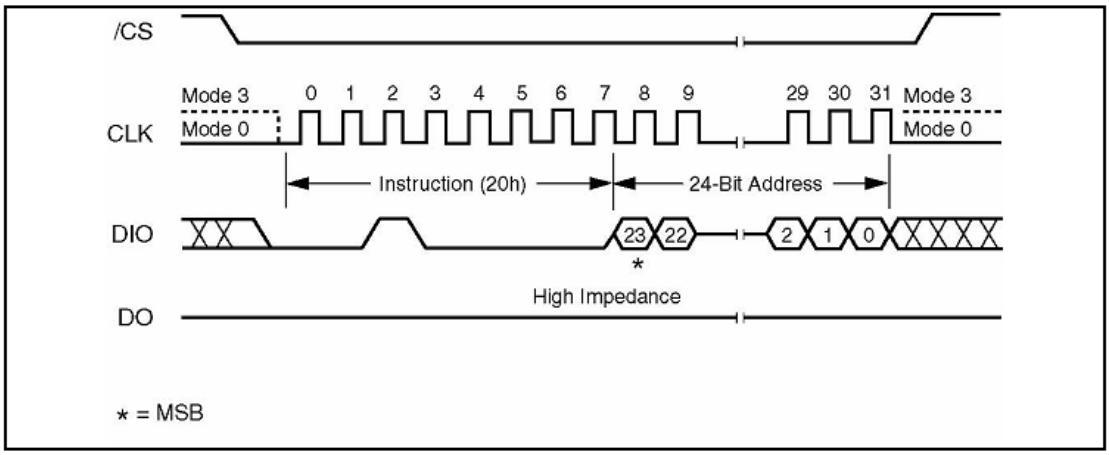

图 1-12 扇区擦除时序

扇区擦除指令的第一个字节为指令编码，紧接着发送的 3 个字节用于表示要擦除的存储矩阵地址。要注意的是在扇区擦除指令前，还需要先发送“写使能”指令，发送扇区擦除指令后，通过读取寄存器状态等待扇区擦除操作完毕，代码实现见代码清单 1-10。

代码清单 1-10 擦除扇区

```
/**
 * @brief  擦除 FLASH 扇区
 * @param  SectorAddr：要擦除的扇区地址
 * @retval 无
 */
void SPI_FLASH_SectorErase(u32 SectorAddr)
{
    /* 发送 FLASH 写使能命令 */
    SPI_FLASH_WriteEnable();
    SPI_FLASH_WaitForWriteEnd();
    /* 擦除扇区 */
    /* 选择 FLASH: CS 低电平 */
    SPI_FLASH_CS_LOW();
    /* 发送扇区擦除指令*/
    SPI_FLASH_SendByte(W25X_SectorErase);
    /*发送擦除扇区地址的高位*/
    SPI_FLASH_SendByte((SectorAddr & 0xFF0000) >> 16);
    /* 发送擦除扇区地址的中位 */
    SPI_FLASH_SendByte((SectorAddr & 0xFF00) >> 8);
    /* 发送擦除扇区地址的低位 */
    SPI_FLASH_SendByte(SectorAddr & 0xFF);
    /* 停止信号 FLASH: CS 高电平 */
    SPI_FLASH_CS_HIGH();
    /* 等待擦除完毕*/
    SPI_FLASH_WaitForWriteEnd();
}
```

这段代码调用的函数在前面都已讲解，只要注意发送擦除地址时高位在前即可。调用扇区擦除指令时注意输入的地址要对齐到 4KB。

FLASH 的页写入

目标扇区被擦除完毕后，就可以向它写入数据了。与 EEPROM 类似，FLASH 芯片也有页写入命令，使用页写入命令最多可以一次向 FLASH 传输 256 个字节的数据，我们把这个单位为页大小。FLASH 页写入的时序见图 1-13。

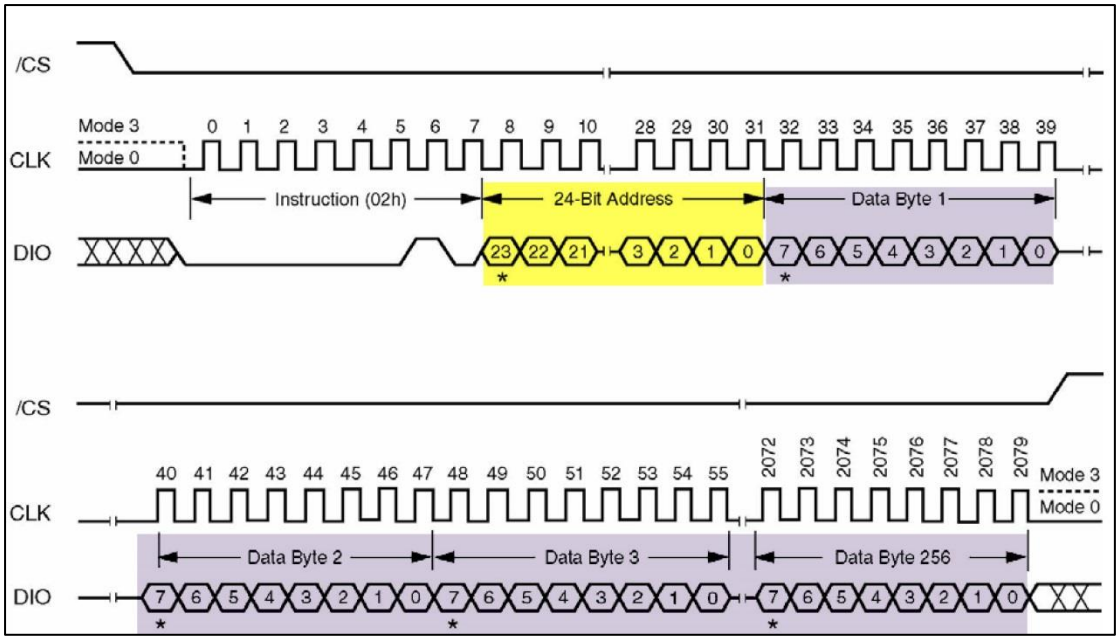

图 1-13 FLASH 芯片页写入

从时序图可知，第 1 个字节为“页写入指令”编码，2-4 字节为要写入的“地址 A”，接着的是要写入的内容，最多个可以发送 256 字节数据，这些数据将会从“地址 A”开始，按顺序写入到 FLASH 的存储矩阵。若发送的数据超出 256 个，则会覆盖前面发送的数据。

与擦除指令不一样，页写入指令的地址并不要求按 256 字节对齐，只要确认目标存储单元是擦除状态即可(即被擦除后没有被写入过)。所以，若对“地址 x”执行页写入指令后，发送了 200 个字节数据后终止通讯，下一次再执行页写入指令，从“地址(x+200)”开始写入 200 个字节也是没有问题的(小于 256 均可)。 只是在实际应用中由于基本擦除单元是 4KB，一般都以扇区为单位进行读写，想深入了解，可学习我们的“FLASH 文件系统”相关的例子。

把页写入时序封装成函数，其实现见代码清单 1-11。

代码清单 1-11 FLASH 的页写入

```
/**
  * @brief  对 FLASH 按页写入数据，调用本函数写入数据前需要先擦除扇区
  * @param  pBuffer，要写入数据的指针
  * @param WriteAddr，写入地址
  * @param  NumByteToWrite，写入数据长度，必须小于等于页大小
  * @retval 无
  */
void SPI_FLASH_PageWrite(u8* pBuffer, u32 WriteAddr, u16 NumByteToWrite)
{
    /* 发送 FLASH 写使能命令 */
    SPI_FLASH_WriteEnable();

    /* 选择 FLASH: CS 低电平 */
    SPI_FLASH_CS_LOW();
    /* 写送写指令*/
    SPI_FLASH_SendByte(W25X_PageProgram);
    /*发送写地址的高位*/
    SPI_FLASH_SendByte((WriteAddr & 0xFF0000) >> 16);
    /*发送写地址的中位*/
```

```
    SPI_FLASH_SendByte((WriteAddr & 0xFF00) >> 8);
    /*发送写地址的低位*/
    SPI_FLASH_SendByte(WriteAddr & 0xFF);

    if (NumByteToWrite > SPI_FLASH_PerWritePageSize)
    {
        NumByteToWrite = SPI_FLASH_PerWritePageSize;
        FLASH_ERROR("SPI_FLASH_PageWrite too large!");
    }

    /* 写入数据*/
    while (NumByteToWrite--)
    {
        /* 发送当前要写入的字节数据 */
        SPI_FLASH_SendByte(*pBuffer);
        /* 指向下一字节数据 */
        pBuffer++;
    }

    /* 停止信号 FLASH: CS 高电平 */
    SPI_FLASH_CS_HIGH();

    /* 等待写入完毕*/
    SPI_FLASH_WaitForWriteEnd();
}
```

这段代码的内容为：先发送“写使能”命令，接着才开始页写入时序，然后发送指令编码、地址，再把要写入的数据一个接一个地发送出去，发送完后结束通讯，检查 FLASH 状态寄存器，等待 FLASH 内部写入结束。

不定量数据写入

应用的时候我们常常要写入不定量的数据，直接调用“页写入”函数并不是特别方便，所以我们在它的基础上编写了“不定量数据写入”的函数，基实现见代码清单 1-12。

代码清单 1-12 不定量数据写入

```
/**
 * @brief  对 FLASH 写入数据，调用本函数写入数据前需要先擦除扇区
 * @param   pBuffer，要写入数据的指针
 * @param  WriteAddr，写入地址
 * @param  NumByteToWrite，写入数据长度
 * @retval 无
 */
void SPI_FLASH_BufferWrite(u8* pBuffer, u32 WriteAddr, u16 NumByteToWrite)
{
    u8 NumOfPage = 0, NumOfSingle = 0, Addr = 0, count = 0, temp = 0;

    /*mod 运算求余，若 writeAddr 是 SPI_FLASH_PageSize 整数倍，
      运算结果 Addr 值为 0*/
    Addr = WriteAddr % SPI_FLASH_PageSize;

    /*差 count 个数据值，刚好可以对齐到页地址*/
    count = SPI_FLASH_PageSize - Addr;
    /*计算出要写多少整数页*/
    NumOfPage =  NumByteToWrite / SPI_FLASH_PageSize;
    /*mod 运算求余，计算出剩余不满一页的字节数*/
    NumOfSingle = NumByteToWrite % SPI_FLASH_PageSize;

    /* Addr=0,则 WriteAddr 刚好按页对齐 aligned  */
    if (Addr == 0)
    {
        /* NumByteToWrite < SPI_FLASH_PageSize */
```

```
27          if (NumOfPage == 0)
28          {
29              SPI_FLASH_PageWrite(pBuffer, WriteAddr,
30                                  NumByteToWrite);
31          }
32          else /* NumByteToWrite > SPI_FLASH_PageSize */
33          {
34              /*先把整数页都写了*/
35              while (NumOfPage--)
36              {
37                  SPI_FLASH_PageWrite(pBuffer, WriteAddr,
38                                      SPI_FLASH_PageSize);
39                  WriteAddr +=  SPI_FLASH_PageSize;
40                  pBuffer += SPI_FLASH_PageSize;
41              }
42              /*若有多余的不满一页的数据，把它写完*/
43              SPI_FLASH_PageWrite(pBuffer, WriteAddr,
44                                  NumOfSingle);
45          }
46      }
47      /* 若地址与 SPI_FLASH_PageSize 不对齐  */
48      else
49      {
50          /* NumByteToWrite < SPI_FLASH_PageSize */
51          if (NumOfPage == 0)
52          {
53              /*当前页剩余的 count 个位置比 NumOfSingle 小，一页写不完*/
54              if (NumOfSingle > count)
55              {
56                  temp = NumOfSingle - count;
57                  /*先写满当前页*/
58                  SPI_FLASH_PageWrite(pBuffer, WriteAddr, count);
59
60                  WriteAddr +=  count;
61                  pBuffer += count;
62                  /*再写剩余的数据*/
63                  SPI_FLASH_PageWrite(pBuffer, WriteAddr, temp);
64              }
65              else /*当前页剩余的 count 个位置能写完 NumOfSingle 个数据*/
66              {
67                  SPI_FLASH_PageWrite(pBuffer, WriteAddr,
68                                      NumByteToWrite);
69              }
70          }
71          else /* NumByteToWrite > SPI_FLASH_PageSize */
72          {
73              /*地址不对齐多出的 count 分开处理，不加入这个运算*/
74              NumByteToWrite -= count;
75              NumOfPage =  NumByteToWrite / SPI_FLASH_PageSize;
76              NumOfSingle = NumByteToWrite % SPI_FLASH_PageSize;
77
78              /* 先写完 count 个数据，为的是让下一次要写的地址对齐 */
79              SPI_FLASH_PageWrite(pBuffer, WriteAddr, count);
80
81              /* 接下来就重复地址对齐的情况 */
82              WriteAddr +=  count;
83              pBuffer += count;
84              /*把整数页都写了*/
85              while (NumOfPage--)
86              {
87                  SPI_FLASH_PageWrite(pBuffer, WriteAddr,
88                                      SPI_FLASH_PageSize);
89                  WriteAddr +=  SPI_FLASH_PageSize;
90                  pBuffer += SPI_FLASH_PageSize;
91              }
92              /*若有多余的不满一页的数据，把它写完*/
```

```
93              if (NumOfSingle != 0)
94              {
95                  SPI_FLASH_PageWrite(pBuffer, WriteAddr,
96                                      NumOfSingle);
97              }
98          }
99      }
100 }
```

很多读者觉得这段代码的运算很复杂，看不懂，其实主要抓住最主要的两点就很好理解了，Buffer 写的时候我们分两种情况来讨论 。

地址对齐

地址对齐的时候，代码比较好理解。通过“整除”计算要写入的数据 NumByteToWrite 能写满多少“完整的页”，计算得的值存储在 NumOfPage 中，但有时数据不是刚好能写满完整页的，会多一点出来，通过“求余”计算得出“不满一页的数据个数”就存储在 NumOfSingle 中。然后调用 I2C_EE_PageWrite()函数把数据写到 FLASH 中。

地址不对齐

当地址不对齐的时候，会需要一个 count 值，用于存储从该首地址开始到写满该地址所在的页，还能写多少个数据。实际传输时，先把这部分 count 个数据先写入，填满该页，然后把剩余的数据(NumByteToWrite-count)，再重复上述求出 NumOPage 及 NumOfSingle 的过程，按页对齐的方式传输到 FLASH。

在实际调用这个“不定量数据写入”函数时，要确保目标扇区处于擦除状态。

从 FLASH 读取数据

相对于写入，FLASH 芯片的数据读取要简单得多，使用读取指令“Read Data”即可，其指令时序见图 1-14。

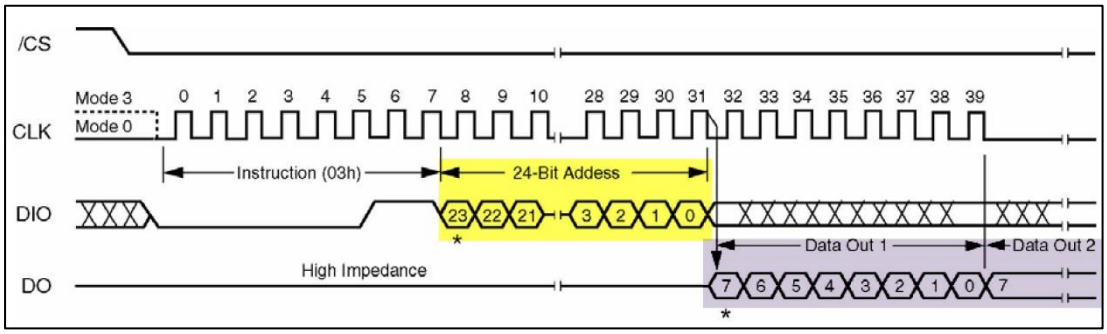

图 1-14 EEPROM 页写入时序(摘自《AT24C02》规格书)

发送了指令编码及要读的起始地址后，FLASH 芯片就会按地址递增的方式返回存储矩阵的内容，读取的数据量没有限制，只要没有停止通讯，FLASH 芯片就会一直返回数据。代码实现见代码清单 1-13。

代码清单 1-13 从 FLASH 读取数据

```
1 /**
2   * @brief  读取 FLASH 数据
3   * @param  pBuffer，存储读出数据的指针
4   * @param   ReadAddr，读取地址
5   * @param   NumByteToRead，读取数据长度
6   * @retval 无
7   */
8 void SPI_FLASH_BufferRead(u8* pBuffer, u32 ReadAddr, u16 NumByteToRead)
```

```
{
    /* 选择 FLASH: CS 低电平 */
    SPI_FLASH_CS_LOW();

    /* 发送 读 指令 */
    SPI_FLASH_SendByte(W25X_ReadData);

    /* 发送 读 地址高位 */
    SPI_FLASH_SendByte((ReadAddr & 0xFF0000) >> 16);
    /* 发送 读 地址中位 */
    SPI_FLASH_SendByte((ReadAddr& 0xFF00) >> 8);
    /* 发送 读 地址低位 */
    SPI_FLASH_SendByte(ReadAddr & 0xFF);

    /* 读取数据 */
    while (NumByteToRead--)
    {
        /* 读取一个字节*/
        *pBuffer = SPI_FLASH_SendByte(Dummy_Byte);
        /* 指向下一个字节缓冲区 */
        pBuffer++;
    }

    /* 停止信号 FLASH: CS 高电平 */
    SPI_FLASH_CS_HIGH();
}
```

由于读取的数据量没有限制，所以发送读命令后一直接收 NumByteToRead 个数据到结束即可。

3. main 函数

最后我们来编写 main 函数，进行 FLASH 芯片读写校验，见代码清单 1-14。

代码清单 1-14 main 函数

```
int main(void)
{
    LED_GPIO_Config();
    LED_BLUE;

    /* 配置串口 1 为: 115200 8-N-1 */
    USART_Config();
    printf("\r\n 这是一个 8Mbyte 串行 flash(W25Q64)实验 \r\n");

    /* 8M 串行 flash W25Q64 初始化 */
    SPI_FLASH_Init();

    /* 获取 Flash Device ID */
    DeviceID = SPI_FLASH_ReadDeviceID();
    Delay( 200 );

    /* 获取 SPI Flash ID */
    FlashID = SPI_FLASH_ReadID();
    printf("\r\n FlashID is 0x%X,\
    Manufacturer Device ID is 0x%X\r\n", FlashID, DeviceID);

    /* 检验 SPI Flash ID */
    if (FlashID == sFLASH_ID)
    {
        printf("\r\n 检测到串行 flash W25Q64 !\r\n");

        /* 擦除将要写入的 SPI FLASH 扇区，FLASH 写入前要先擦除 */
        // 这里擦除 4K，即一个扇区，擦除的最小单位是扇区
```

```
29          SPI_FLASH_SectorErase(FLASH_SectorToErase);
30
31          /* 将发送缓冲区的数据写到 flash 中 */
32          // 这里写一页，一页的大小为 256 个字节
33          SPI_FLASH_BufferWrite(Tx_Buffer, FLASH_WriteAddress, BufferSize);
34          printf("\r\n 写入的数据为：%s \r\t", Tx_Buffer);
35
36          /* 将刚刚写入的数据读出来放到接收缓冲区中 */
37          SPI_FLASH_BufferRead(Rx_Buffer, FLASH_ReadAddress, BufferSize);
38          printf("\r\n 读出的数据为：%s \r\n", Rx_Buffer);
39
40          /* 检查写入的数据与读出的数据是否相等 */
41          TransferStatus1 = Buffercmp(Tx_Buffer, Rx_Buffer, BufferSize);
42
43          if ( PASSED == TransferStatus1 )
44          {
45              LED_GREEN;
46              printf("\r\n 8M 串行 flash(W25Q64)测试成功!\n\r");
47          }
48          else
49          {
50              LED_RED;
51              printf("\r\n 8M 串行 flash(W25Q64)测试失败!\n\r");
52          }
53      }// if (FlashID == sFLASH_ID)
54      else// if (FlashID == sFLASH_ID)
55      {
56          LED_RED;
57          printf("\r\n 获取不到 W25Q64 ID!\n\r");
58      }
59
60      while (1);
61  }
```

函数中初始化了 LED、串口、SPI 外设，然后读取 FLASH 芯片的 ID 进行校验，若 ID 校验通过则向 FLASH 的特定地址写入测试数据，然后再从该地址读取数据，测试读写是否正常。

注意：

由于实验板上的 FLASH 芯片默认已经存储了特定用途的数据，如擦除了这些数据会影响到某些程序的运行。所以我们预留了 FLASH 芯片的“第 0 扇区(0-4096 地址)”专用于本实验，如非必要，请勿擦除其它地址的内容。如已擦除，可在配套资料里找到“刷外部 FLASH 内容”程序，根据其说明给 FLASH 重新写入出厂内容。

1.4.3 下载验证

用 USB 线连接开发板“USB TO UART”接口跟电脑，在电脑端打开串口调试助手，把编译好的程序下载到开发板。在串口调试助手可看到 FLASH 测试的调试信息。

1.5 每课一问

1. 在 SPI 外设初始化部分，MISO 引脚可以设置为输入模式吗？为什么？实际测试现象如何？
2. 尝试使用 FLASH 芯片存储 int 整型变量，float 型浮点变量，编写程序写入数据，并读出校验。*提示：FLASH 存储的数据都是 0 和 1，至于你存储的是整数还是浮点型的数，*

FLASH 是完全不知道的，只是你解析的时候解析成什么类型的数而已。当你存储的是浮点型的数据的时候，那么解析的时候就是 4 个字节才是一个数，如果存储的是字符型的，那么解析的时候一个字节就是一个数。

3. 如果扇区未经擦除就写入，会有什么后果？请做实验验证。
4. 简述 FLASH 存储器与 EEPROM 存储器的区别。