

## SPI 总线和 SPI 接口的 Flash 芯片



### 目录

1. SPI 总线概述 .....	3
1.1 定义 .....	3
1.2 特点 .....	3
1.2.1 采用主-从模式(Master-Slave) 的控制方式 .....	3
1.2.2 采用同步方式(Synchronous)传输数据 .....	4
1.2.3 数据交换(Data Exchanges) .....	4
1.3 工作机制 .....	5
1.3.1 概述 .....	5
1.3.2 Timing .....	6
1.3.3 SSPSR .....	7
1.3.4 SSPBUF .....	9
1.3.5 Controller .....	10
2. SPI 接口的 Flash 芯片 M25P80 .....	11
2.1 概述 .....	11
2.2 SPI 模式 .....	11
2.3 所支持的操作 .....	12
2.4 指令集 .....	13
2.4.1 WREN(Write Enable)写使能 .....	13
2.4.2 WRDI(Write Disable) .....	13

2.4.3 RDSR (Read Status Register) .....	14
2.4.4 WRSR(Write Status Register) .....	15
2.4.5 READ(Read Data Byte) .....	15
2.4.6 FAST_READ (Fast Read).....	15
2.4.7 PP(Page Program) .....	16
2.4.8 SE (Sector Erase).....	17
2.4.9 BE(Bulk Erase).....	18
2.4.10 DP (Deep Power Down) .....	18
2.4.11 RES(Read Electronic Signature and Release from Deep power down).....	19
2.4.12 RES(Release from Deep power-down ) .....	19
3. M25P80 硬件连接单片机的硬件原理图 .....	20
3.1 STC8A8K64S4A12 开发板上面的 SPI Flash 连接 .....	20
3.2 H 系列核心板中 H-STC15 板上面的 SPI Flash.....	21
4. STC 单片机的 SPI 模块 .....	21
4.1 控制寄存器 .....	22
4.2 状态寄存器 .....	23
4.3 数据寄存器 .....	23
4.4 辅助寄存器 AUXR1.....	23
5. STC 单片机的 SPI 底层操作 .....	24
6. M25P80 的驱动程序 .....	25
7. M25P80 的上层应用 demo 程序 .....	30

7.1 test-m25p80.....	30
7.2 xmodem 协议传输文件到 Flash.....	31
8.源代码下载 .....	51

## 1. SPI 总线概述

### 1.1 定义

SPI(Serial Peripheral Interface), 串行外围设备接口, 是 Motorola 公司推出的一种同步串行接口技术. STC12 STC15 等单片机内部集成了 SPI 模块, 通过对此模块相关的寄存器进行操作启动通信、交换数据等。SPI 允许 MCU 以全双工的同步串行方式, 与各种外围设备进行高速数据通信。和 UART I2C 类似, SPI 是单片机系统中最为常见的外设接口之一, 主要应用在 EEPROM, Flash, 实时时钟(RTC), 数模转换器(ADC), 数字信号处理器 (DSP) 以及数字信号解码器之间。它在芯片中只占用四根管脚 (Pin) 用于控制以及数据传输,节约了芯片的 pin 脚数目, 同时为 PCB 在布局上节省了空间. 正是出于这种简单易用的特性, 现在越来越多的芯片上都集成了 SPI 技术.

### 1.2 特点

#### 1.2.1 采用主-从模式(Master-Slave) 的控制方式

SPI 规定了两个 SPI 设备之间通信必须由主设备 (Master) 来控制从设备 (Slave). 一个 Master 设备可以通过提供 Clock 以及对

Slave 设备进行片选 (Slave Select) 来控制多个 Slave 设备, SPI 协议还规定 Slave 设备的 Clock 由 Master 设备通过 SCK 管脚提供给 Slave 设备, Slave 设备本身不能产生或控制 Clock, 没有 Clock 则 Slave 设备不能正常工作.

### 1.2.2 采用同步方式(Synchronous)传输数据

Master 设备会根据将要交换的数据来产生相应的时钟脉冲(Clock Pulse), 时钟脉冲组成了时钟信号(Clock Signal), 时钟信号通过时钟极性 (CPOL) 和 时钟相位 (CPHA) 控制着两个 SPI 设备间何时数据交换以及何时对接收到的数据进行采样, 来保证数据在两个设备之间是同步传输的.

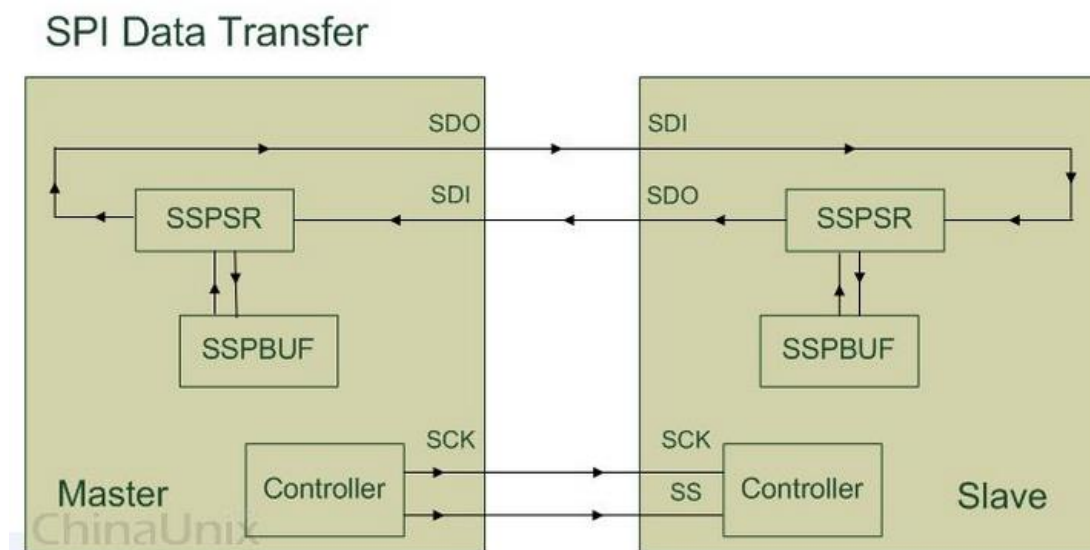
### 1.2.3 数据交换(Data Exchanges)

SPI 设备间的数据传输之所以又被称为数据交换, 是因为 SPI 协议规定一个 SPI 设备不能在数据通信过程中仅仅只充当一个 "发送者(Transmitter)" 或者 "接收者(Receiver)". 在每个 Clock 周期内, SPI 设备都会发送并接收一个 bit 大小的数据, 相当于该设备有一个 bit 大小的数据被交换了. 一个 Slave 设备要想能够接收到 Master 发过来的控制信号, 必须在此之前能够被 Master 设备进行访问 (Access). 所以, Master 设备必须首先通过 SS/CS pin 对 Slave 设备进行片选, 把想要访问的 Slave 设备选上. 在数据传输的过程中, 每次接收到的数据必须在下一次数据传输之前被采样. 如果之前接收到的数据没有被读取, 那么这些已经接收完成的数据将有可能被丢弃, 导致 SPI 物理模块最终失效. 因此, 在程序中一般都会在 SPI 传输完数据

后, 去读取 SPI 设备里的数据, 即使这些数据(Dummy Data)在我们的程序里是无用的.

## 1.3 工作机制

### 1.3.1 概述



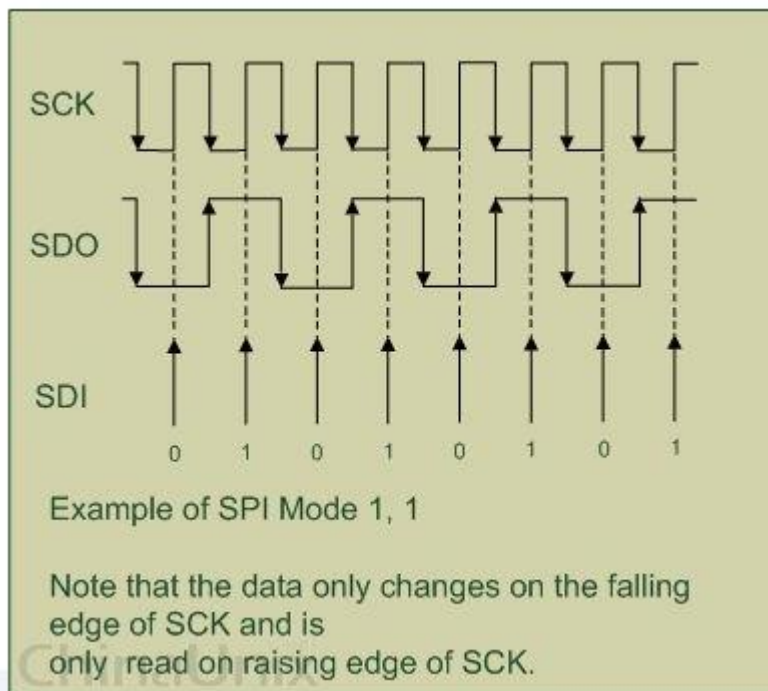
上图只是对 SPI 设备间通信的一个简单的描述, 下面就来解释一下图中所示的几个组件 (Module): SSPBUF, Synchronous Serial Port Buffer, 泛指 SPI 设备里面的内部缓冲区, 一般在物理上是以 FIFO 的形式, 保存传输过程中的临时数据; SSPSR, Synchronous Serial Port Register, 泛指 SPI 设备里面的移位寄存器(Shift Register), 它的作用是根据设置好的数据位宽(bit-width) 把数据移入或者移出 SSPBUF; Controller, 泛指 SPI 设备里面的控制寄存器, 可以通过配置它们来设置 SPI 总线的传输模式.通常情况下, 我们只需要对上图所描述四个管脚(pin) 进行编程即可控制整个 SPI 设备之间的数据通信:

SCK, Serial Clock, 主要的作用是 Master 设备往 Slave 设备传输时钟

信号, 控制数据交换的时机以及速率;

SS/CS, Slave Select/Chip Select, 用于 Master 设备片选 Slave 设备, 使被选中的 Slave 设备能够被 Master 设备所访问;SDO/MOSI, Serial Data Output/Master Out Slave In, 在 Master 上面也被称为 Tx-Channel, 作为数据的出口, 主要用于 SPI 设备发送数据;SDI/MISO, Serial Data Input/Master In Slave Out, 在 Master 上面也被称为 Rx-Channel, 作为数据的入口, 主要用于 SPI 设备接收数据;SPI 设备在进行通信的过程中, Master 设备和 Slave 设备之间会产生一个数据链路回环(Data Loop), 就像上图所画的那样, 通过 SDO 和 SDI 管脚, SSPSR 控制数据移入移出 SSPBUF, Controller 确定 SPI 总线的通信模式, SCK 传输时钟信号.

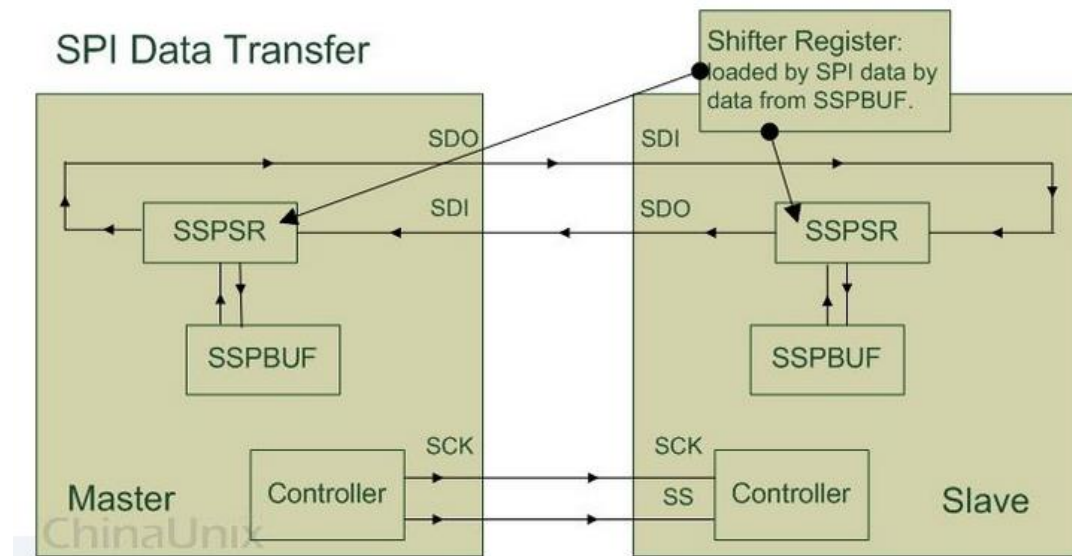
### 1.3.2 Timing.



上图通过 Master 设备与 Slave 设备之间交换 1 Byte 数据来说明

SPI 协议的工作机制.首先, 在这里解释一下两个概念: CPOL: 时钟极性, 表示 SPI 在空闲时, 时钟信号是高电平还是低电平. 若 CPOL 被设为 1, 那么该设备在空闲时 SCK 管脚下的时钟信号为高电平. 当 CPOL 被设为 0 时则正好相反. CPHA: 时钟相位, 表示 SPI 设备是在 SCK 管脚上的时钟信号变为上升沿时触发数据采样, 还是在时钟信号变为下降沿时触发数据采样. 若 CPHA 被设置为 1, 则 SPI 设备在时钟信号变为下降沿时触发数据采样, 在上升沿时发送数据. 当 CPHA 被设为 0 时也正好相反.上图里的 "Mode 1, 1" 说明了本例所使用的 SPI 数据传输模式被设置成 CPOL = 1, CPHA = 1. 这样, 在一个 Clock 周期内, 每个单独的 SPI 设备都能以全双工(Full-Duplex)的方式, 同时发送和接收 1 bit 数据, 即相当于交换了 1 bit 大小的数据. 如果 SPI 总线的 Channel-Width 被设置成 Byte, 表示 SPI 总线上每次数据传输的最小单位为 Byte, 那么挂载在该 SPI 总线的设备每次数据传输的过程至少需要 8 个 Clock 周期(忽略设备的物理延迟).因此, SPI 总线的频率越快, Clock 周期越短, 则 SPI 设备间数据交换的速率就越快.

### 1.3.3 SSPSR.



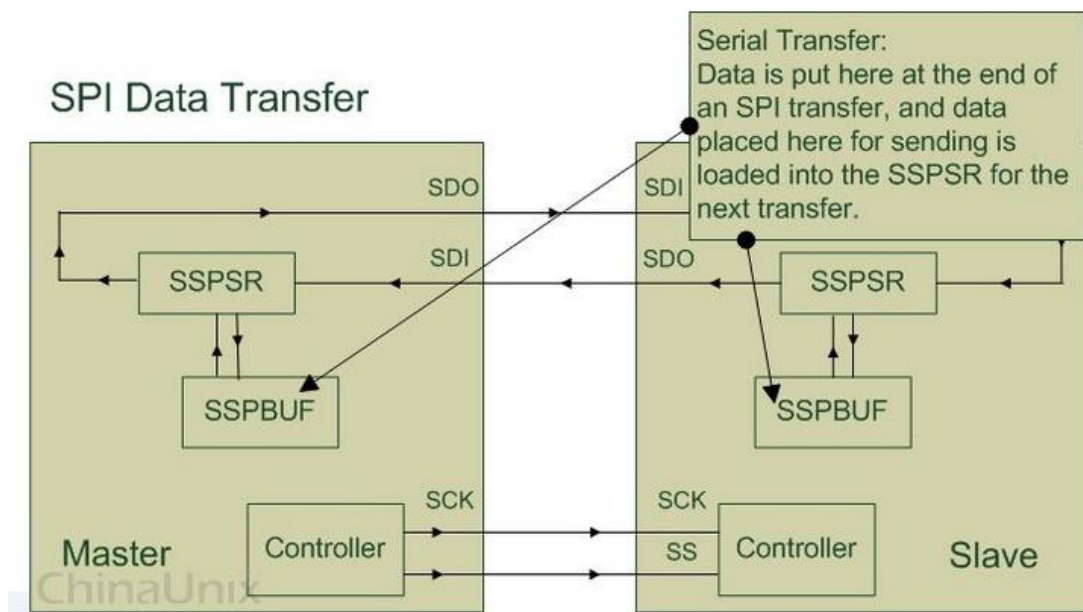
SSPSR 是 SPI 设备内部的移位寄存器(Shift Register). 它的主要作用是根据 SPI 时钟信号状态, 往 SSPBUF 里移入或者移出数据, 每次移动的数据大小由 Bus-Width 以及 Channel-Width 所决定.

Bus-Width 的作用是指定地址总线到 Master 设备之间数据传输的单位.例如, 我们想要往 Master 设备里面的 SSPBUF 写入 16 Byte 大小的数据: 首先, 给 Master 设备的配置寄存器设置 Bus-Width 为 Byte; 然后往 Master 设备的 Tx-Data 移位寄存器在地址总线的入口写入数据, 每次写入 1 Byte 大小的数据(使用 writeb 函数); 写完 1 Byte 数据之后, Master 设备里面的 Tx-Data 移位寄存器会自动把从地址总线传来的 1 Byte 数据移入 SSPBUF 里; 上述动作一共需要重复执行 16 次. Channel-Width 的作用是指定 Master 设备与 Slave 设备之间数据传输的单位. 与 Bus-Width 相似, Master 设备内部的移位寄存器会依据 Channel-Width 自动地把数据从 Master-SSPBUF 里通过 Master-SDO 管脚搬运到 Slave 设备里的 Slave-SDI 引脚, Slave—SSPSR 再把每次接收的数据移入 Slave-SSPBUF 里.通常情况下,



Bus-Width 总是会大于或等于 Channel-Width, 这样能保证不会出现因 Master 与 Slave 之间数据交换的频率比地址总线与 Master 之间的数据交换频率要快, 导致 SSPBUF 里面存放的数据为无效数据这样的情况.

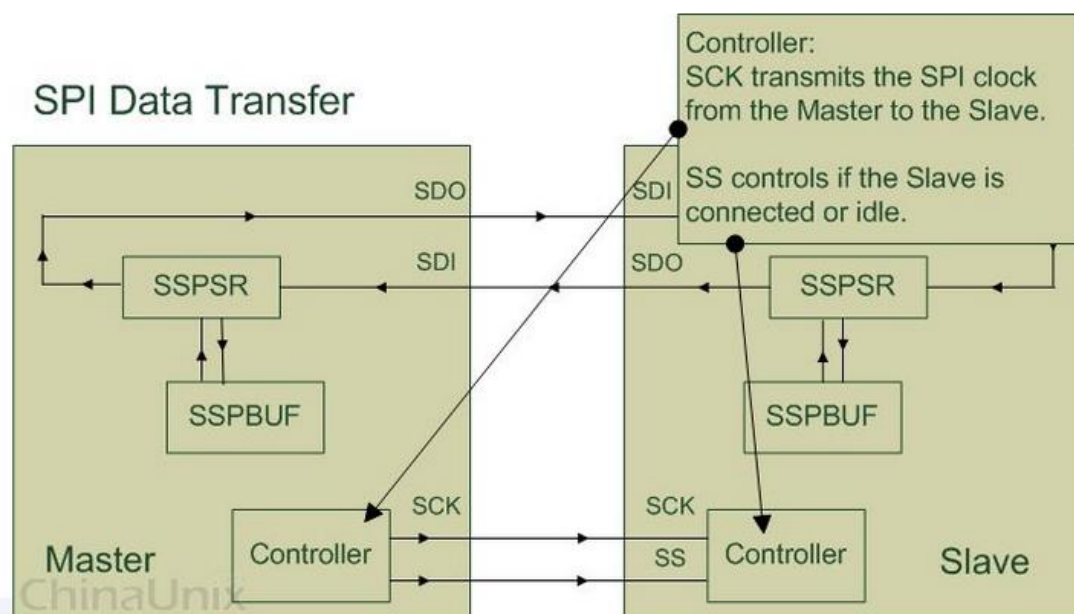
#### 1.3.4 SSPBUF



我们知道, 在每个时钟周期内, Master 与 Slave 之间交换的数据其实都是 SPI 内部移位寄存器从 SSPBUF 里面拷贝的. 我们可以通过往 SSPBUF 对应的寄存器 (Tx-Data / Rx-Data register) 里读写数据, 间接地操控 SPI 设备内部的 SSPBUF. 例如, 在发送数据之前, 我们应该先往 Master 的 Tx-Data 寄存器写入将要发送出去的数据, 这些数据会被 Master-SSPSR 移位寄存器根据 Bus-Width 自动移入 Master-SSPBUF 里, 然后这些数据又会被 Master-SSPSR 根据 Channel-Width 从 Master-SSPBUF 中移出, 通过 Master-SDO 管脚传给 Slave-SDI 管脚, Slave-SSPSR 则把从 Slave-SDI 接收到的数据移

入 Slave-SSPBUF 里。与此同时, Slave-SSPBUF 里面的数据根据每次接收数据的大小(Channel-Width), 通过 Slave-SDO 发往 Master-SDI, Master-SSPSR 再把从 Master-SDI 接收的数据移入 Master-SSPBUF.在单次数据传输完成之后, 用户程序可以通过从 Master 设备的 Rx-Data 寄存器读取 Master 设备数据交换得到的数据。

### 1.3.5 Controller.



Master 设备里面的 Controller 主要通过时钟信号(Clock Signal)以及片选信号 (Slave Select Signal)来控制 Slave 设备. Slave 设备会一直等待, 直到接收到 Master 设备发过来的片选信号, 然后根据时钟信号来工作. Master 设备的片选操作必须由程序所实现. 例如: 由程序把 SS/CS 管脚的时钟信号拉低电平, 完成 SPI 设备数据通信的前期工作; 当程序想让 SPI 设备结束数据通信时, 再把 SS/CS 管脚上的时钟信号拉高电平。

## 2. SPI 接口的 Flash 芯片 M25P80

### 2.1 概述

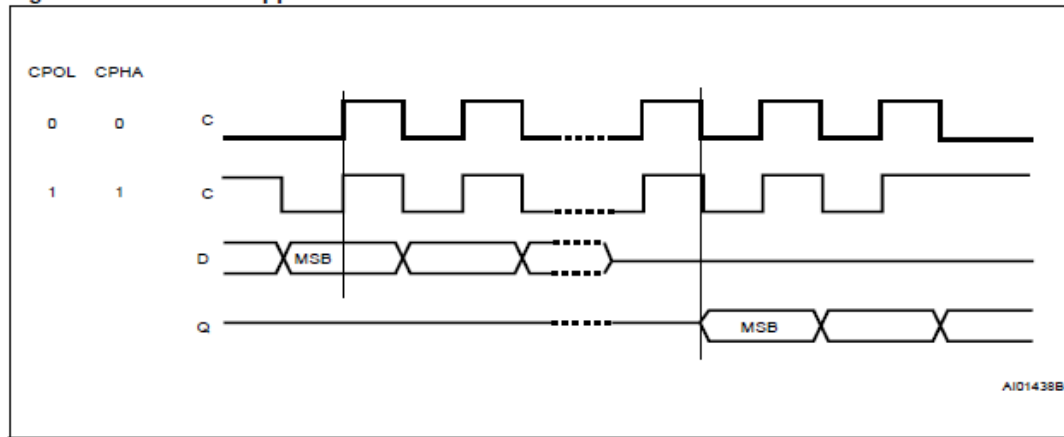
Flash 芯片是嵌入式系统中非常常用的一种存储外设，它可以永久保存数据，掉电后数据不会丢失，价格适中，体积小巧，SPI 接口的 Flash/EEPROM 芯片访问速度远远超过 I2C 接口的 EEPROM。是一种性价比很高的存储设备。在有图形界面的系统中存储汉字库、存储位图图片、存储字符串资源，在数据采集系统中存放采集的历史数据等用途广泛。

《[M25P80](#)》是 ST 公司的一款 Flash 芯片，存储容量是 8Mbit，也就是 1M 字节，这 1M 字节的数据分成 16 个扇区，每个扇区分成 256 页，每页 256 个字节。

### 2.2 SPI 模式

M25P80 支持两种 SPI 模式，分别是  $CPOL = 0, CPHA = 1$  和  $CPOL = 1, CPHA = 1$ 。这两种模式都是在 SCK 的下降沿输出，在 SCK 的上升沿采样。不同之处在于 standby 模式下面， $CPOL = 0$  模式，SCK 为低电平。 $CPOL = 1$  为高电平。

Figure 6. SPI Modes Supported



## 2.3 所支持的操作

SPI flash 目前所支持的操作有：

**Page Programming** 对一页内最多 256 个字节编程写入。

**Sector Erase and Bulk Erase** 擦除一个扇区或者所有扇区的内容

**Polling During Write, Program or Erase Cycle.** 在执行编程操作,或者擦除期间查询芯片状态。

**Active Power, Standby and deep power-down** 进入低功耗等省电模式

**Protection mode** 进入保护模式

## 2.4 指令集

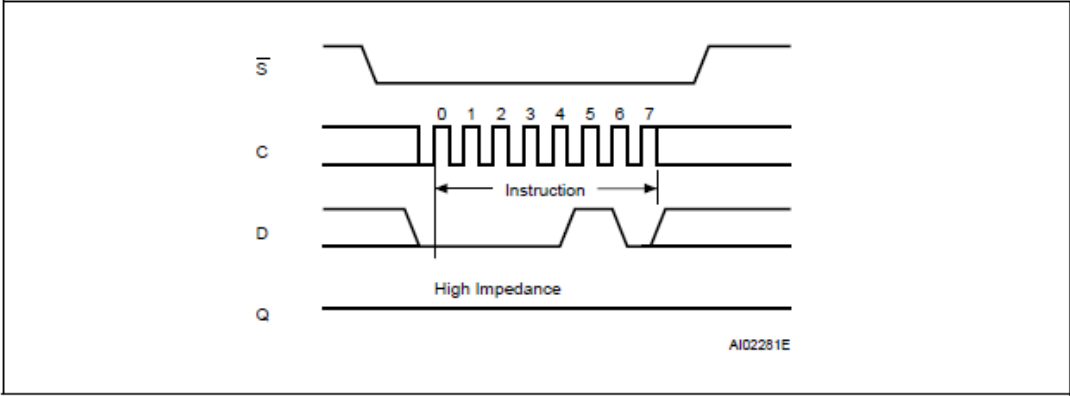
Table 4. Instruction Set

Instruction	Description	One-byte Instruction Code		Address Bytes	Dummy Bytes	Data Bytes
WREN	Write Enable	0000 0110	06h	0	0	0
WRDI	Write Disable	0000 0100	04h	0	0	0
RDSR	Read Status Register	0000 0101	05h	0	0	1 to ∞
WRSR	Write Status Register	0000 0001	01h	0	0	1
READ	Read Data Bytes	0000 0011	03h	3	0	1 to ∞
FAST_READ	Read Data Bytes at Higher Speed	0000 1011	0Bh	3	1	1 to ∞
PP	Page Program	0000 0010	02h	3	0	1 to 256
SE	Sector Erase	1101 1000	D8h	3	0	0
BE	Bulk Erase	1100 0111	C7h	0	0	0
DP	Deep Power-down	1011 1001	B9h	0	0	0
RES	Release from Deep Power-down, and Read Electronic Signature	1010 1011	ABh	0	3	1 to ∞
	Release from Deep Power-down			0	0	0

以上是 M25P80 支持的指令集，各种操作例如擦除扇区，写入数据、读取数据等就是通过一个或者多个指令序列组合来实现的。

### 2.4.1 WREN(Write Enable)写使能

Figure 9. Write Enable (WREN) Instruction Sequence

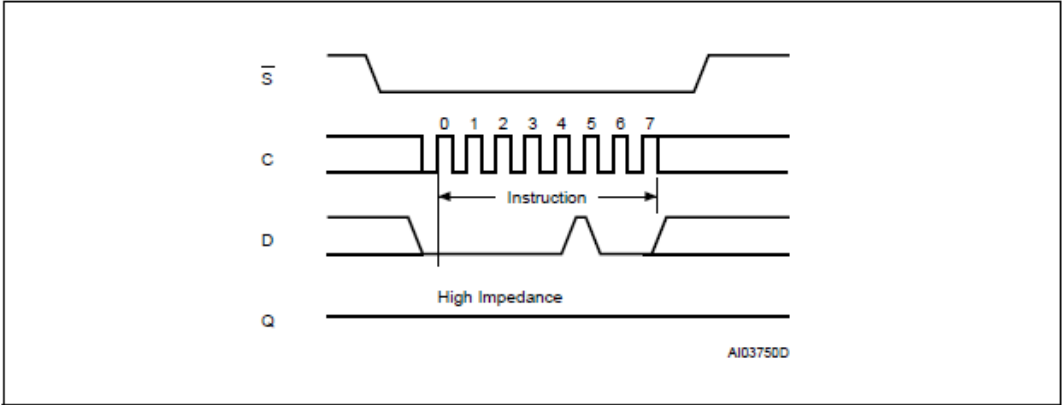


CS 片选信号变低，然后写入一字节 WREN 指令(0x06),然后 CS 片选信号变高。在执行编程 PP(Page Program)、SE(Sector Erase)、BE(Bulk Erase) WRSR(Write Status Register)之前必须执行此指令以使能相关功能。

### 2.4.2 WRDI(Write Disable)

此命令释放 WEL(Write Enable Latch)位,和 WREN 操作功能相反。

Figure 10. Write Disable (WRDI) Instruction Sequence



除了使用此命令显式释放 WEL 位之外，PP SE BEWRSR 操作之后会总  
动释放 WEL 位。

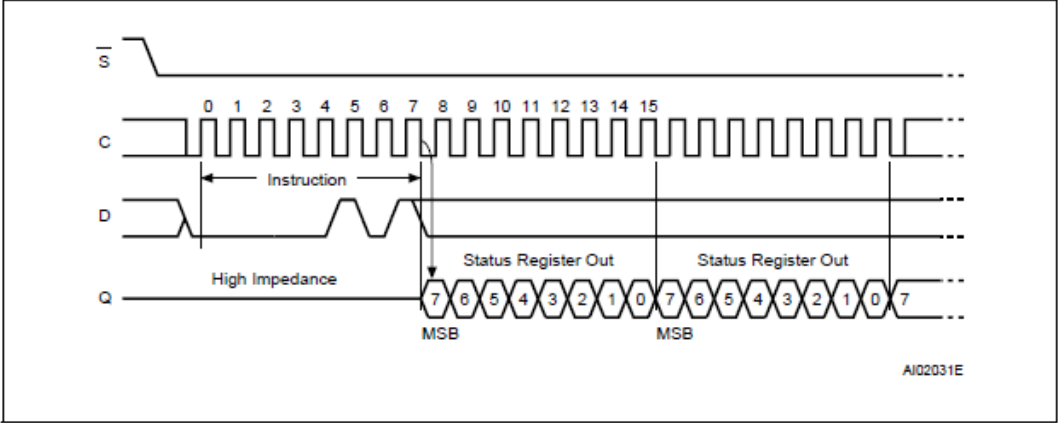
2.4.3 RDSR (Read Status Register)

Table 5. Status Register Format

b7				b0			
SRWD	0	0	BP2	BP1	BP0	WEL	WIP
Status Register Write Protect				Block Protect Bits			Write Enable Latch Bit
							Write In Progress Bit

在执行写操作、擦除操作之前通常需要读取此寄存器的值以确保上一  
个写操作已经完成，当前处在空闲状态。

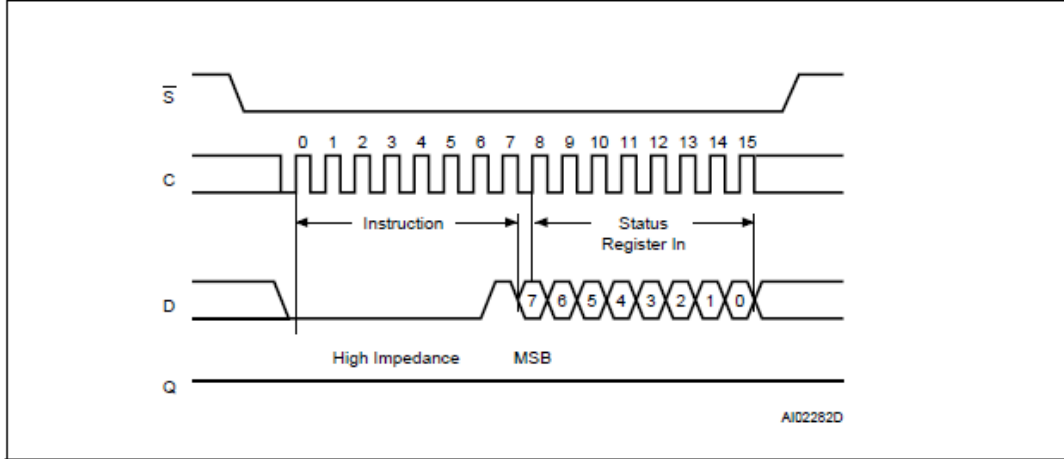
Figure 11. Read Status Register (RDSR) Instruction Sequence and Data-Out Sequence



#### 2.4.4 WRSR(Write Status Register)

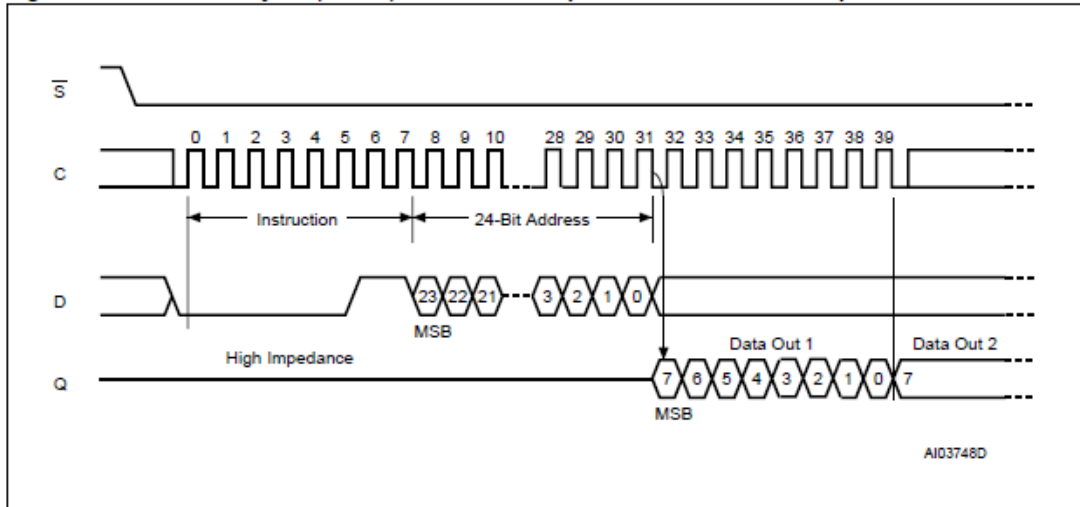
此命令通常用于更改 Block Protection 设置。

Figure 12. Write Status Register (WRSR) Instruction Sequence



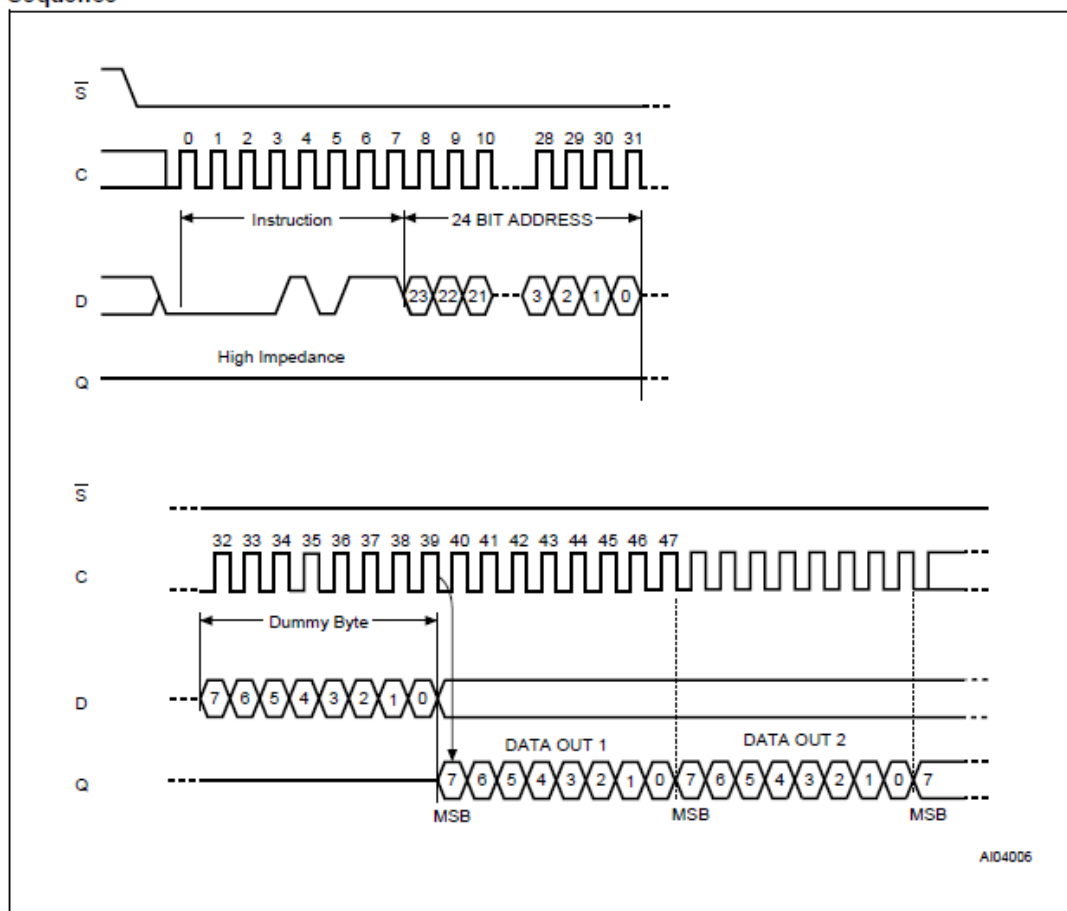
#### 2.4.5 READ(Read Data Byte)

Figure 13. Read Data Bytes (READ) Instruction Sequence and Data-Out Sequence



此命令用于从存储区域读取数据.操作时序上，首先写入指令字节 0x03.然后写入三字节的片内地址，然后一次读取存储内容。

#### 2.4.6 FAST\_READ (Fast Read)

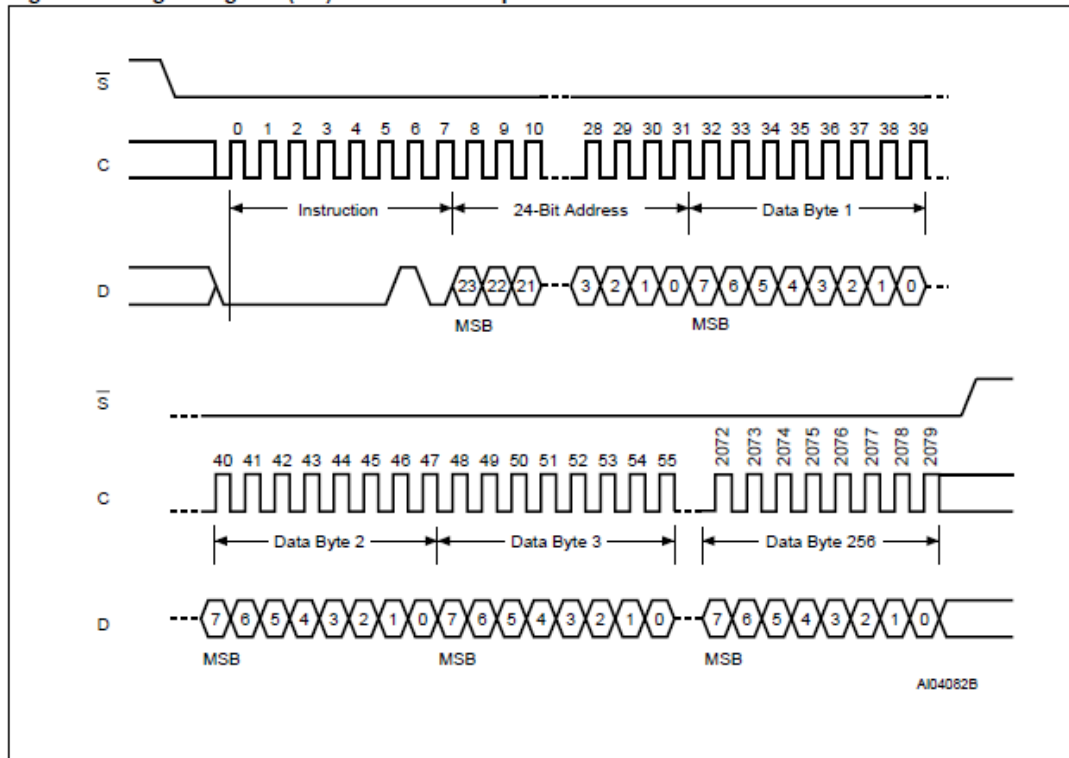
**Figure 14. Read Data Bytes at Higher Speed (FAST\_READ) Instruction Sequence and Data-Out Sequence**

这个指令和 READ 的功能是一样的。

#### 2.4.7 PP(Page Program)



Figure 15. Page Program (PP) Instruction Sequence

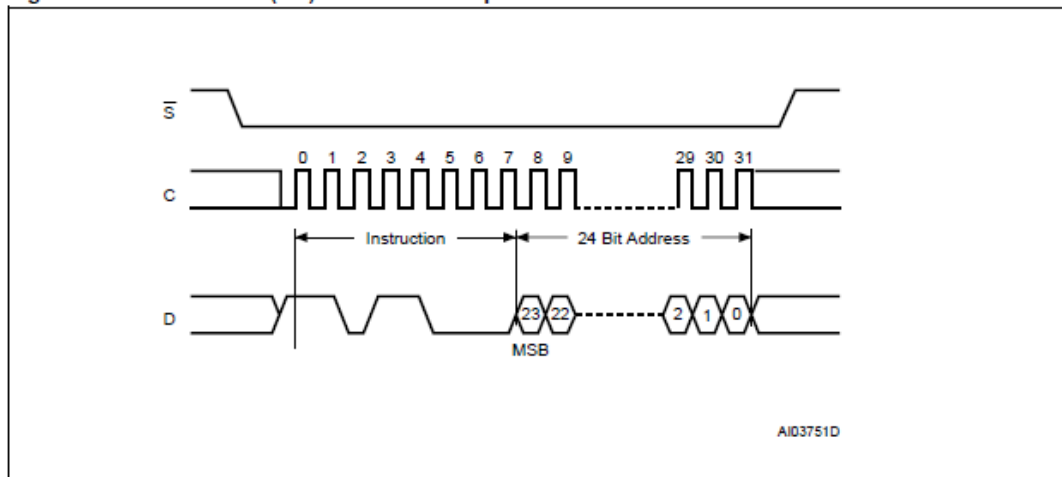


Note: Address bits A23 to A20 are Don't Care.

此指令用于对芯片进行编程，在执行此指令之前必须首先执行 **WREN** 指令以使能写操作。此指令首先是写入指令字节，然后写入三字节的地址，然后把需要编程写入的数据依次写入。

#### 2.4.8 SE (Sector Erase)

Figure 16. Sector Erase (SE) Instruction Sequence



Note: Address bits A23 to A20 are Don't Care.

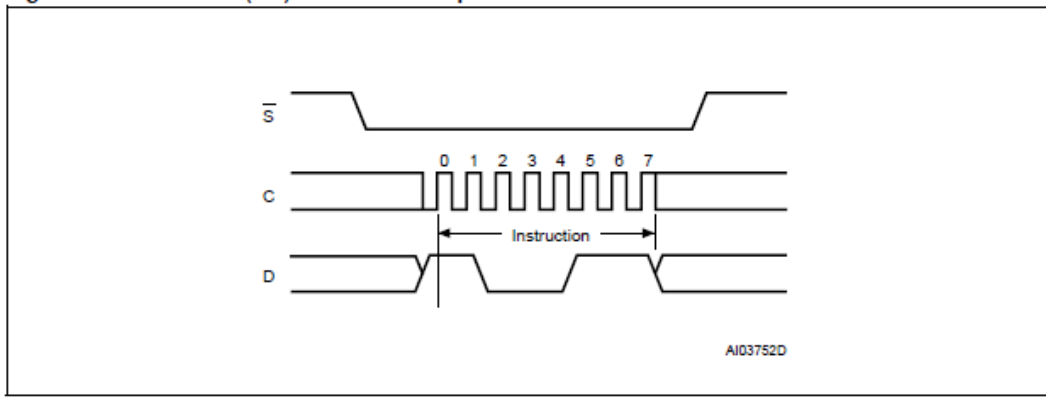
擦除扇区的操作，首先把 **CS** 置低片选，然后写入指令字节，然后写

入 3 字节的地址，此地址的低 16 位被忽略。写入在此扇区范围内的任一地址值都会导致此扇区被擦除。

需要注意的是，此函数执行前，需要先执行 WREN 操作，使能 WEL 位。当此指令序列中 CS 变高以后，芯片内部将启动一个自擦除的动作，在此期间，芯片除了可以读取状态字节外，不接受其他命令。

#### 2.4.9 BE(Bulk Erase)

Figure 17. Bulk Erase (BE) Instruction Sequence

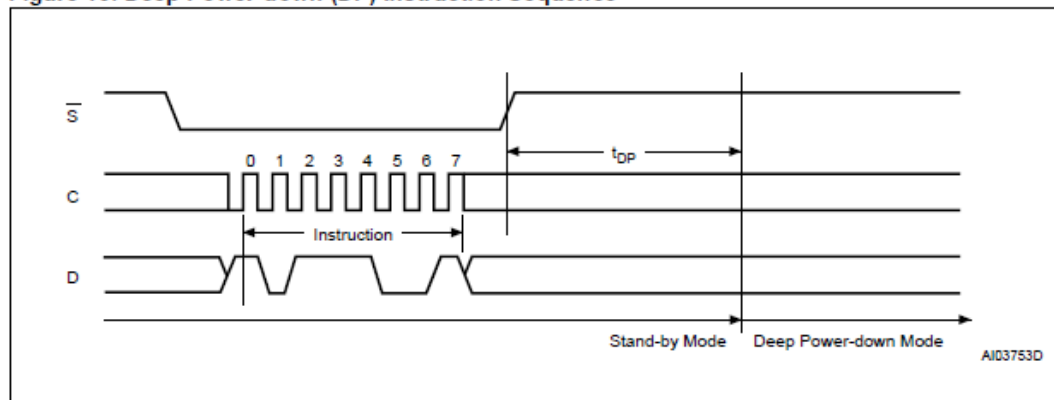


此指令用于擦除整个存储区。此指令序列最简单，将 CS 置地后写入指令字节，然后将 CS 置高。CS 置高后内部启动自擦除操作，和 SE 一样，在内部擦除动作没有执行完以前，除了读取状态字节外，不接收其他任何指令。

此指令执行之前同样需要首先执行 WREN 操作。以使能 WEL 位。

#### 2.4.10 DP (Deep Power Down)

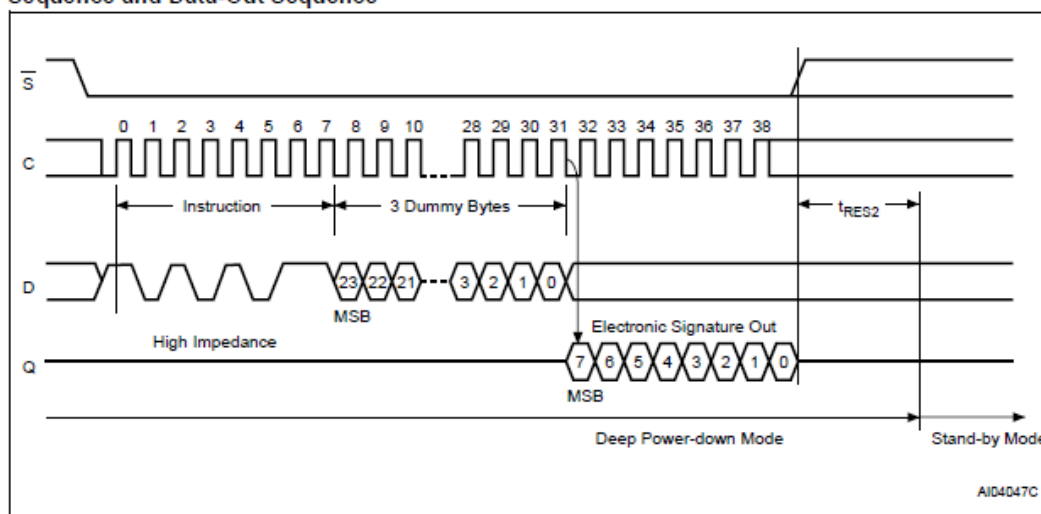
Figure 18. Deep Power-down (DP) Instruction Sequence



此指令把芯片置位低功耗模式。在低功耗模式下仅仅接受 RES(Read Electronic Signature)指令和 Release from Deep power down 指令。

#### 2.4.11 RES(Read Electronic Signature and Release from Deep power down)

Figure 19. Release from Deep Power-down and Read Electronic Signature (RES) Instruction Sequence and Data-Out Sequence



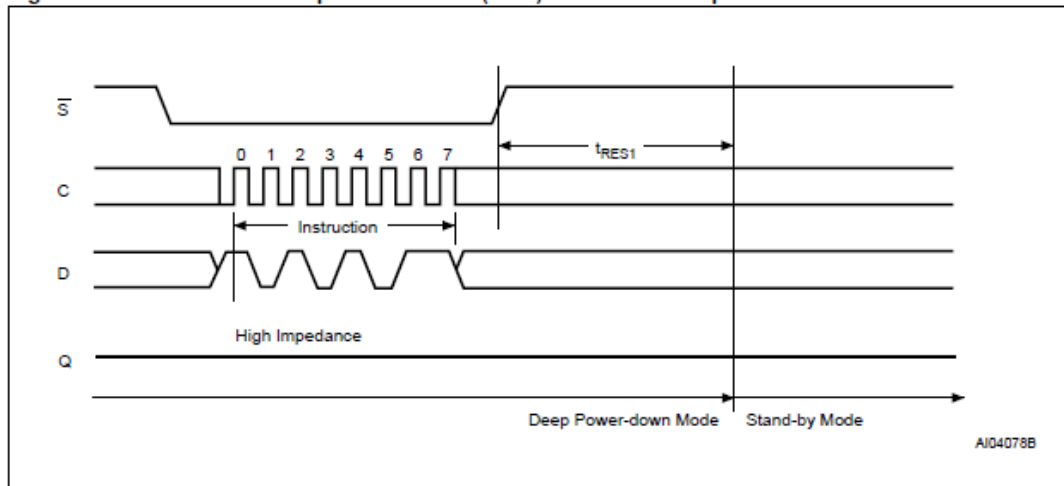
Note: The value of the 8-bit Electronic Signature, for the M25P80, is 13h.

此指令读取电子标识，并且从低功耗模式唤醒。此序列首先写入指令字节，然后跟 3 字节 dummy 数据，然后电子标识将输出。

#### 2.4.12 RES(Release from Deep power-down )

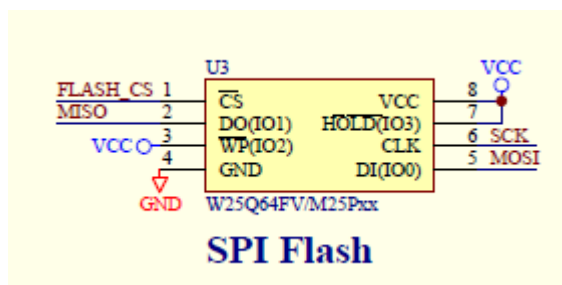
此指令使芯片从低功耗模式退出。

Figure 20. Release from Deep Power-down (RES) Instruction Sequence



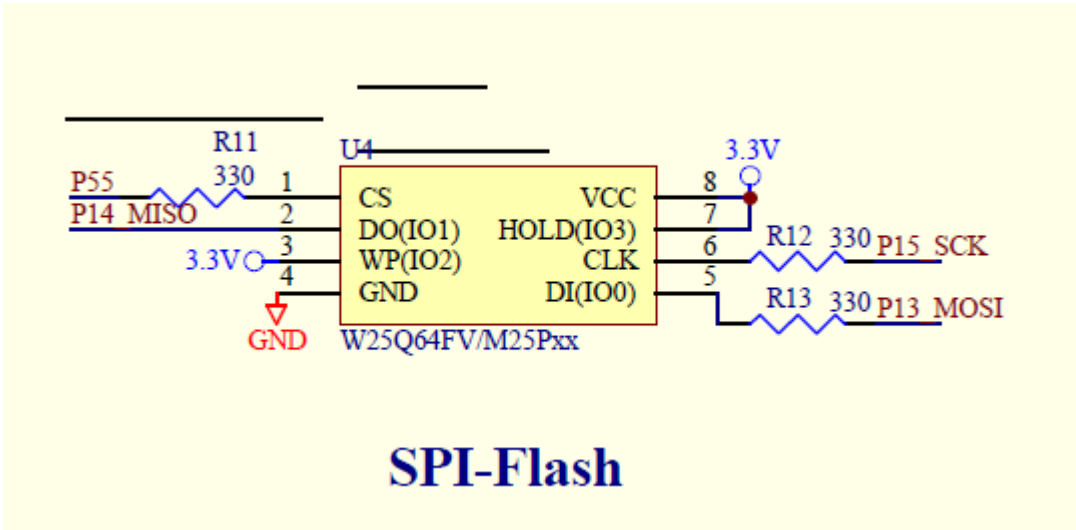
### 3. M25P80 硬件连接单片机的硬件原理图

#### 3.1 STC8A8K64S4A12 开发板上面的 SPI Flash 连接



Flash 芯片的片选管脚连接在单片机的 IO 管脚上面，P1.5 P1.6 P1.7 分别复用作 MOSI MISO 和 SCK 管脚。

### 3.2 H 系列核心板中 H-STC15 板上面的 SPI Flash



和 STC8A8K64S4A12 开发板的连接的区别仅在于片选线的不同，此处用 P5.5 作为片选线。

## 4. STC 单片机的 SPI 模块

STC 早期的单片机例如 STC89C52 是没有内置 SPI 功能模块的，如果要操作 SPI 外设，需要使用软件模拟 SPI 的时序。STC12 STC15 内置了 SPI 模块后，操作的便利性和通信速率大大提升。

STC12C5A60S2系列 1T 8051单片机SPI功能模块特殊功能寄存器 SPI Management SFRs

符号	描述	地址	位地址及其符号								复位值
			B7	B6	B5	B4	B3	B2	B1	B0	
SPCTL	SPI Control Register	CEH	SSIG	SPEN	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	0000,0100
SPSTAT	SPI Status Register	CDH	SPIF	WCOL	-	-	-	-	-	-	00xx,xxxx
SPDAT	SPI Data Register	CFH									0000,0000
AUXR1	Auxiliary Register 1	A2H	-	PCA_P4	SPI_P4	S2_P4	GF2	ADRJ	-	DPS	x000,00x0

和 SPI 功能相关的寄存器共有 4 个，简洁但功能强大。下面仅对四个寄存器进行描述，更详细的通信细节可以参考 STC 单片机的数据手册。

## 4.1 控制寄存器

SPCTL：SPI控制寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
SPCTL	CEH	name	SSIG	SPEN	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

SPCTL 是控制寄存器。SSIG 是 SS 引脚忽略控制位，当 SSIG = 1 的时候，B4 MSTR 确定是主机还是从机，SSIG = 0 的时候 SS 脚用于确定器件是主机还是从机。

SPEN 是 SPI 使能位，为 1 时 SPI 使能。为 0 时 SPI 功能禁用，此时 SPI 相关的引脚可以作为 IO 口使用。

DORD 是设定发送和接收数据的顺序，为 1 则 LSB 先发送，为 0 则是 MSB 先发送。

MSTR 是从主模式选择位，后有详述。

CPOL 是时钟极性位，为 1 时 SCK 空闲时为高电平，为 0 则 SCK 空闲时为低电平。

CPHA 是时钟相位选择位，为 1 则数据在前时钟沿驱动，为 0 则在 SS 为低时被驱动，在 SCK 的后时钟沿被改变，在前时钟沿被采样。

SPR1/0 是时钟速率选择控制位。

SPI时钟频率的选择

SPR1	SPR0	时钟( SCLK )
0	0	CPU_CLK/4
0	1	CPU_CLK/16
1	0	CPU_CLK/64
1	1	CPU_CLK/128

## 4.2 状态寄存器

SPSTAT: SPI状态寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
SPSTAT	CDH	name	SPIF	WCOL	-	-	-	-	-	-

SPIF 是 SPI 传输完成标志,当一次传输完成的时候 SPIF 被置位, 如果 ESPI 位置位而且 EA 也为 1, 则会产生 SPI 中断。当 SPI 处于主模式且 SSIG 为 0 时, 如果 SS 为输入而且被驱动为低电平则 SPIF 也将被置位表示模式改变。

WCOL: 写冲突标志, 在数据传输的过程中, 如果对 SPI 寄存器 SPDAT 执行了写操作, WCOL 将置位。WCOL 位通过软件向其写入 1 清零。

## 4.3 数据寄存器

SPDAT: SPI数据寄存器

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
SPDAT	CFH	name								

## 4.4 辅助寄存器 AUXR1

AUXR1: 辅助寄存器1

SFR name	Address	bit	B7	B6	B5	B4	B3	B2	B1	B0
AUXR1	A2H	name	-	PCA_P4	SPI_P4	S2_P4	GF2	ADRJ	-	DPS

SPI\_P4 为 0 的时候, 缺省 SPI 在 P1 口。为 1 的时候 SPI 从 P1 口切换到 P4 口. SPICLK-P4.3 MISO-P4.2 MOSI-P4.1 SS-P4.0。

## 5. STC 单片机的 SPI 底层操作

```

void spi_initialization(uint8_t speed)
{
    speed &= 0x03u;
    /*Configure SPI to drive M25P80*/
    SPCTL = 0xdc | speed; /*SSIG SPEN DORD MSTR CPOL CPHA SPR1 SPR0*/
                        /* 1   1   0   1   0   0   0   0 */
                        /*Ignore SS pin,Enable SPI, MSB first,CPOL =0,CPHA = 0,
                        |clock rate = 1/4 CPU_clk*/

    SPSTAT = 0xc0; /*clear SPIF WCOL flags.*/
}

```

SPI 接口的初始化操作非常简单，设置 SPICL 寄存器，使能 SPI 功能，MSB first,CPOL=0 CPHA=0，时钟工作在 1/4 系统时钟。然后清除 SPSTAT 中的 WCOL 标志位。此处的时序时钟等是将单片机设置为主模式，为操作 M25P80 芯片所设置的。如果要操作其他的 SPI 接口外设，需要查看其 datasheet，设置合理的 CPOL 和 CPHA 位。

```

57 uint8_t spi_ExchangeByte(uint8_t val)
58 {
59     SPDAT = val;
60     while((SPSTAT & 0x80) == 0);
61     SPSTAT = 0x80;
62     //if(SPSTAT & 0x40) printf("SPI Err");
63     return SPDAT;
64 }

```

spi\_ExchangeByte()函数用来在主机和从机之间交换数据。发送数据和接收数据都是使用此函数来实现。如果要向从设备写入数据，则将待写入字节作为入口参数传入，SPDAT = val；把数据写入 SPDAT 寄存器后，SPI 模块就会产生正确的时序把数据发送出去。如果要执行读取操作，入口参数的 val 可以写任何数据，通过一次 dummy 写操作产生 SCK 上面的时钟，从而把从机发出的数据读进来。



## 6. M25P80 的驱动程序

```

65 static void StartCommand(uint8_t cmd)
66 {
67     SpiFlashSelect();
68     Delaylus();
69     spi_ExchangeByte(cmd);
70 }
71
72 static void EndCommand(void)
73 {
74     SpiFlashDeselect();
75     Delaylus();
76 }

```

StartCommand()此函数选中 SPI Flash 并把 cmd 所指定的指令值发送出去。EndCommand()取消 SPI flash 的片选，并延时后结束一条执行的序列。

```

78 static uint8_t xdata g_eep_error = 0;
79 static void BusyWait(void)
80 {
81     uint8_t val;
82     uint32_t i;
83     for(i=0; i<4000ul;i++)
84     {
85         StartCommand(EPPCMD_RDSR);
86         val = spi_ExchangeByte(0);
87         EndCommand();
88         if((val & 0x01) == 0) return;
89         time_HoggingDelayMs(1);
90         if(i % 10 == 0) { /*sys_FeedingWatchdog();*/ }
91     }
92     if(g_eep_error < 10) g_eep_error++;
93     printf("EEP fatal error.");
94 }

```

此函数循环读取 M25P80 的 Status 寄存器,如果 WIP(Write In Progress) 位为 1,则表示当前芯片正忙,此时会继续重试,直到最长重试 4 秒后仍然没有变成空闲模式,则表示可能出现了严重问题。如果发现 Status 寄存器的位已经为 0,则表示已经处在空闲状态了。

```

96 static void WriteEnable(void)
97 {
98     uint8_t val;
99     uint32_t i;
100
101     BusyWait();
102     StartCommand(EPPCMD_WREN);
103     EndCommand();
104     for(i=0;i<4000ul;i++)
105     {
106         StartCommand(EPPCMD_RDSR);
107         val = spi_ExchangeByte(0);
108         EndCommand();
109         if(val & 0x02) return;
110         time_HoggingDelayMs(1);
111         if(i % 10 == 0){/*sys_FeedingWatchdog();*/}
112     }
113     if(g_eep_error < 10) g_eep_error++;
114     printf("WriteEnable:EPP fatal error.");
115 }

```

WREN 指令的代码实现，首先 BusyWait 检查是否处在空闲状态，然后片选后发送 WREN 指令字节，然后读取 status 寄存器，检查 WEL 位是否为 1。

```

127 static void SectorErase(uint32_t addr)
128 {
129     BusyWait();
130     WriteEnable();
131
132     StartCommand(EPPCMD_SE);
133     spi_ExchangeByte(addr >> 16);
134     spi_ExchangeByte(addr >> 8);
135     spi_ExchangeByte(addr >> 0);
136     EndCommand();
137 }

```

擦除一个扇区的操作，首先是等待设备空闲，然后执行 WREN 指令，然后执行 SE 指令，依次写入指令字，以及三个字节的扇区地址。

```

146 static void BulkErase(void)
147 {
148     BusyWait();
149     WriteEnable();
150
151     StartCommand(EPPCMD_BE);
152     EndCommand();
153 }

```

擦除整个的存储区域，此指令序列首先检查芯片是否空闲，然后 WREN 指令，然后片选后发送 BE 指令，然后取消片选。此命令执行后芯片大约有 10 秒左右的时间会处于忙状态，在此时间内将无法执行除查询状态之外的其他命令。

```

155 static uint8_t ReadSignature(void)
156 {
157     uint8_t val;
158     StartCommand(EEPCMD_RES);
159     spi_ExchangeByte(0);
160     spi_ExchangeByte(0);
161     spi_ExchangeByte(0);
162     val = spi_ExchangeByte(0);
163     EndCommand();
164     return val;
165 }

```

此函数读取芯片的电子标识。

```

167 static void PageProgram(uint16_t page,uint16_t offset,
168     const uint8_t *buffer, uint16_t size) large
169 {
170     uint16_t i;
171     uint32_t addr;
172
173     addr = (((uint32_t)page)<<8) + offset;
174     if((addr & 0xffff) == 0) SectorErase(addr);
175     BusyWait();
176     WriteEnable();
177     StartCommand(EEPCMD_PP);
178     spi_ExchangeByte(addr >> 16);
179     spi_ExchangeByte(addr >> 8);
180     spi_ExchangeByte(addr >> 0);
181     for(i=0 ; i<size; i++) spi_ExchangeByte(buffer[i]);
182     EndCommand();
183 }

```

此函数实现烧录编程一页数据的功能，入口参数指定页序号，页内偏移起始地址，所要写入的数据，以及所要写入的数据的长度。指令执行时首先是查询系统的忙状态，然后 WREN 命令，然后写入 PP 指令，然后写入地址，然后把数据依次写入。此函数作为一个辅助子函数被 void m25p80\_Write(const void \*vbuf, uint16\_t len, uint32\_t eep\_addr) 函数调用。

```

static char xdata g_virtual_eep[10];
void m25p80_Write(const void *vbuf, uint16_t len, uint32_t eep_addr) large
{
    const uint8_t * buffer = (const uint8_t *)vbuf;
    uint16_t page,startpos;
    uint16_t nleft,nwrite;

    if(g_eep_error >= 10)
    {
        if(len < sizeof(g_virtual_eep)) memcpy(g_virtual_eep,vbuf,len);
        return;
    }
    page = eep_addr / 256;
    startpos = eep_addr % 256;
    nleft=len;
    for(;nleft;)
    {
        if(nleft <= 256 - startpos)
            nwrite = nleft;
        else
            nwrite = 256 - startpos;
        PageProgram(page,startpos,buffer,nwrite);
        buffer += nwrite;
        nleft -= nwrite;
        page ++;
        startpos=0;
    }
}

```

这个是 m25P80 软件模块给出的对外 API 接口之一，用于写数据。这个函数把按照扇区和页操作的非线性操作整合成为线性操作，对于此函数的调用者来说，他可以把 M25P80 简单地看作一个 1M 字节的线性文件，可以在任意起始地址，写入任意长度的数据。而在函数内部则把调用者传入的数据，根据页边界进行组织分页写入。

```

/*for MSP430 sizeof(void *) == 2 !!! 16 bit pointer */
void m25p80_Read(void *vbuf, uint16_t len, /*void * */uint32_t eep_addr)
{
    uint16_t i;
    uint8_t *buf = (uint8_t *)vbuf;

```

```

if(g_eep_error >= 10)
{
    if(len < sizeof(g_virtual_eep)) memcpy(vbuf,g_virtual_eep,len);
    return;
}
BusyWait();
StartCommand(EEPCMD_READ);
spi_ExchangeByte(eep_addr >> 16);
spi_ExchangeByte(eep_addr >> 8);
spi_ExchangeByte(eep_addr >> 0);
for(i=0 ; i<len; i++)
{
    buf[i]=spi_ExchangeByte(0);
}
EndCommand();
}

```

**m25p80\_Read()**函数是另一个此模块的导出函数，用于从m25p80中读取数据，调用此函数指定读取数据的存放地址，所要读取的数据长度，以及片内起始地址即可。

```

void m25p80_EraseChip(void)
{
    uint32_t secaddr;
    if(g_eep_error >= 10) return;
    for(secaddr=0;secaddr<0xffffful;secaddr+=0x10000ul)
    {
        SectorErase(secaddr);
    }
}

```

此函数擦除整个芯片的存储内容。

```

void m25p80_Initialization(void)
{
    spi_Initialization(SPI_SPEED_MOSTFAST);
    WriteStatusReg(0x00);
    spi_SetSpeed(SPI_SPEED_MOSTFAST);
}

```

初始化函数，初始化 SPI 接口，设置 SPI 接口的时钟速率。

## 7. M25P80 的上层应用 demo 程序

在代码中对于 M25P80 的上层应用是通过在 console 控制台上敲入命令执行的方式实现的。《串口-底层驱动和上层应用》文中详细描述了控制台的代码实现细节，此处不再赘述。

```
39         else if(strcmp(argv[0], "test-m25p80") == 0)
40         {
41             m25p80_TestMain(argc,argv);
42         }
43         else if(strcmp(argv[0], "xmodem") == 0)
44         {
45             xmodem_Main(argc,argv);
46         }
```

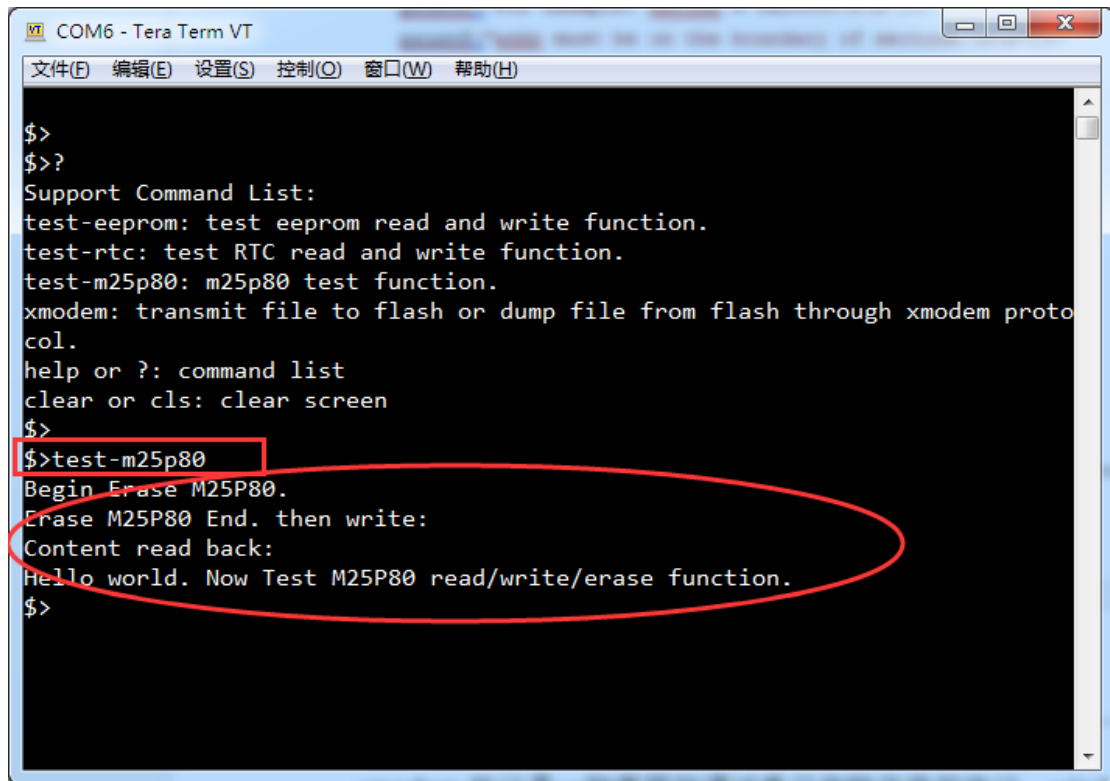
console 中的代码片段，有两个命令可以测试 SPI Flash 相关的功能，分别是 test-m25p80 和 xmodem.

### 7.1 test-m25p80

```
void m25p80_TestMain(uint8_t argc, char *argv[])
{
    const char code *str = "Hello world. Now Test M25P80 read/write/erase function.";
    char xdata buf[64];

    argc = argc;
    argv = argv;
    memset(buf,0,sizeof(buf));
    printf("Begin Erase M25P80.\r\n");
    m25p80_EraseChip();
    printf("Erase M25P80 End. then write:\r\n");
    m25p80_Write(str,strlen(str)+1,0x500);
    m25p80_Read(buf,sizeof(buf),0x500);
    buf[sizeof(buf)-1] = 0;
    printf("Content read back:\r\n");
    printf("%s\r\n",buf);
}
```

test-m25p80 函数比较简单，它首先擦除整个 flash 的内容，然后写入一个字符串，然后再读出来进行对比。



```
COM6 - Tera Term VT
文件(F) 编辑(E) 设置(S) 控制(O) 窗口(W) 帮助(H)

$>
$>?
Support Command List:
test-eeeprom: test eeprom read and write function.
test-rtc: test RTC read and write function.
test-m25p80: m25p80 test function.
xmodem: transmit file to flash or dump file from flash through xmodem protocol.
help or ?: command list
clear or cls: clear screen
$>
$>test-m25p80
Begin Erase M25P80.
Erase M25P80 End. then write:
Content read back:
Hello world. Now Test M25P80 read/write/erase function.
$>
```

## 7.2 xmodem 协议传输文件到 Flash

xmodem 协议稍显复杂。

```
void xmodem_Main(uint8_t argc, char *argv[])
{
    int32_t addr;
    int32_t len;

    if(argc == 1) goto _print_usage_;
    if(strcmp(argv[1], "w") == 0)
    {
        if(argc != 3) goto _print_usage_;
        addr = atol(argv[2]);
        if(addr < 0 || addr > 0xf0000 || (addr & 0xffff) != 0 )
            goto _print_usage_;
        xmodem_ReceiveFile(addr);
        return;
    }
    else if(strcmp(argv[1], "r") == 0)
```

```

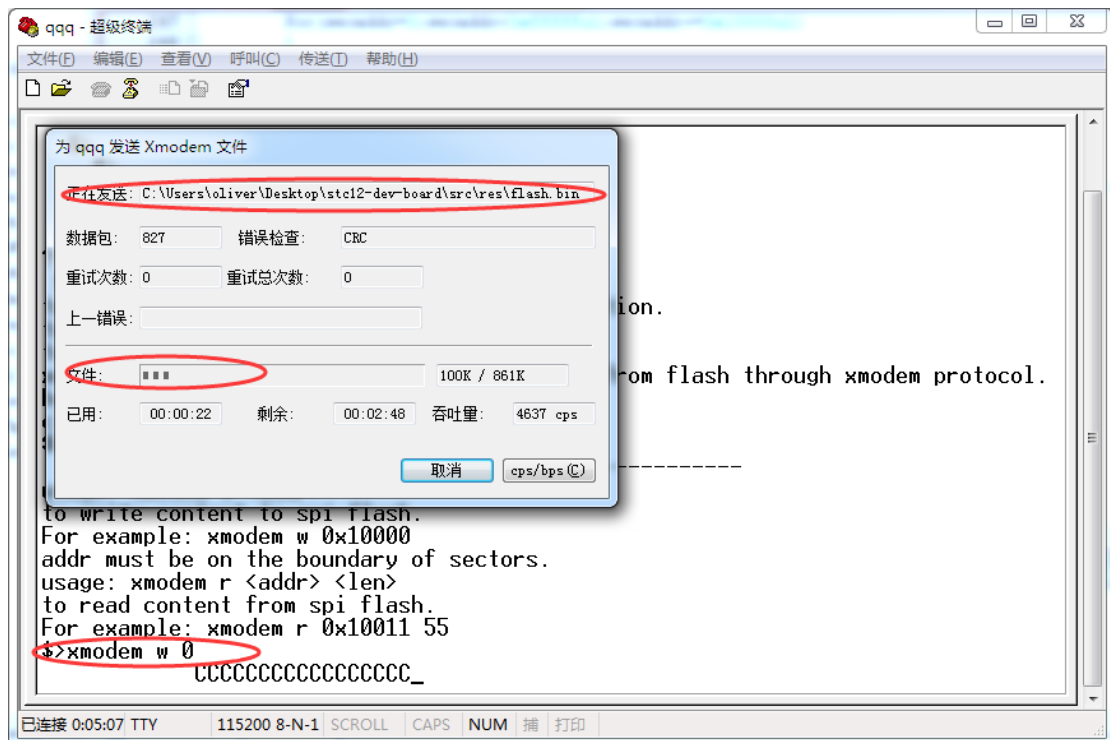
{
    if(argc != 4) goto _print_usage_;
    addr = atol(argv[2]);
    if(addr < 0 || addr > 0xfffff)goto _print_usage_;
    len = atol(argv[3]);
    if(len <= 0)goto _print_usage_;
    xmodem_SendFile(addr,len);
    return;
}

_print_usage_:
printf("-----usage-----\r\n");
printf("usage: %s w <addr>\r\n",argv[0]);
printf("to write content to spi flash.\r\n");
printf("For example: xmodem w 0x10000\r\n");
printf("addr must be on the boundary of sectors.\r\n");

printf("usage: %s r <addr> <len>\r\n",argv[0]);
printf("to read content from spi flash.\r\n");
printf("For example: xmodem r 0x10011 55\r\n");
}

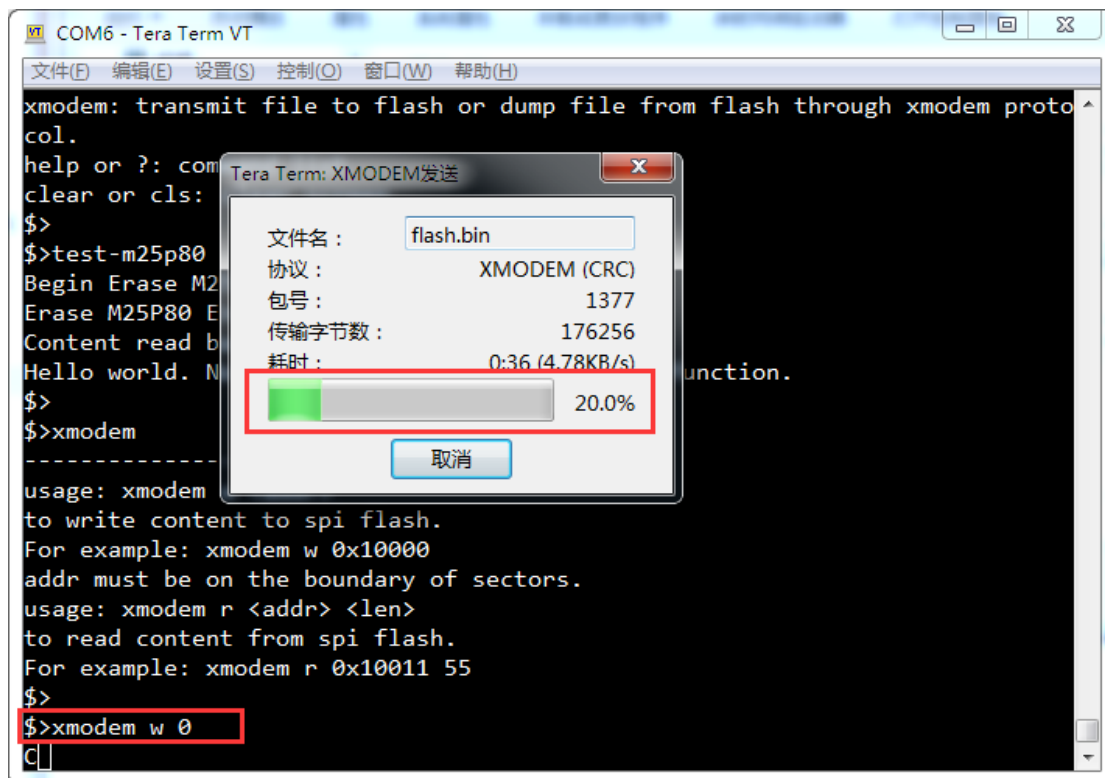
```

xmodem 协议是一种常用的通过串口传输文件的协议，windows XP 上自带的超级终端工具，TeraTerm 串口工具等都支持此协议的传输。

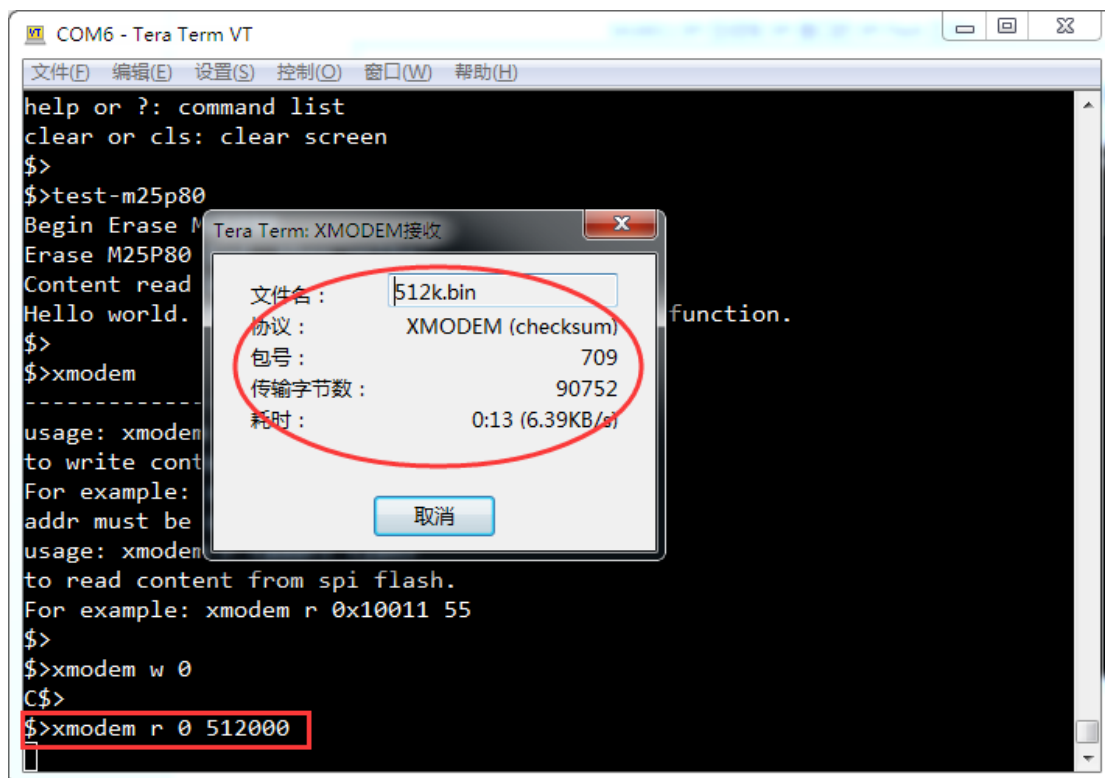


上面是 windows 自带的超级终端的 xmodem 协议传输文件的界面。





通过 teraTerm 工具使用 xmodem 协议发送文件给单片机。



通过 teraTerm 工具使用 xmodem 协议接收文件。

此处的演示代码通过 **xmodem** 协议把 PC 机上面的文件通过串口传输给单片机，单片机把接收到的文件写入到 **SPI flash** 之中，也可以把 **flash** 之中的内容通过 **xmodem** 读取到 PC 机上面的一个文件中。这是一个非常方便的功能，在单片机上面实现了 **xmodem** 协议之后就可以通过终端工具在线升级 **Flash** 的内容，而不需要事先使用编程器烧录好 **FLASH** 的内容然后再焊接到电路板上。另外后面也可以使用 **xmodem** 协议实现升级单片机等功能。

### **xmodem** 协议的具体内容

**Xmodem** 协议是一种使用拨号调制解调器的个人计算机通信中广泛使用的异步文件运输协议。这种协议以 **128** 字节块的形式传输数据，并且每个块都使用一个校验和过程来进行错误检测。如果接收方关于一个块的校验和与它在发送方的校验和相同时，接收方就向发送方发送一个认可字节。

**Xmodem** 协议相关控制字符:

SOH	0x01
STX	0x02
EOT	0x04
ACK	0x06
NAK	0x15
CAN	0x18
CTRLZ	0x1A

标准 **Xmodem** 协议帧格式（每个数据包含有 **128** 字节数据）

标准Xmodem协议帧格式（每个数据包含有128字节数据）

SOH	信息包序号	信息包序号的补码	数据区段	校验和	
_____	_____	_____	_____	_____	

### 说明：

SOH： 帧的开头字节，代表信息包中的第一个字节；

信息包序号： 对 256 取模所得当前包号，第一个信息包的序号为 1 ，而信息包序号范围 0~255；

信息包序号的补码： 当前信息包号的补码；

数据区段： 数据区段的长度固定为 128 字节；

校验和： 1 字节的算术校验和，只对数据区段计算后对 256 取模而得。

### 数据包说明

对于标准 Xmodem 协议来说，如果传送的文件不是 128 的整数倍，那么最后一个数据包的有效内容肯定小于帧长，不足的部分需要用 CTRL-Z(0x1A)来填充。如果传送的是 bootloader 工程生成的.bin 文件，mcu 收到后遇到 0x1A 字符会怎么处理？其实如果传送的是文本文件，那么接收方对于接收的内容是很容易识别的，因为 CTRL-Z 不是前 128 个 ascii 码，不是通用可见字符，如果是二进制文件，mcu 其实也不会把它当作代码来执行。哪怕是 excel 文件等，由于其内部会有些结构表示各个字段长度等，所以不会读取多余的填充字符。**启动传输** 传输由接收方启动，方法是向发送方发送"C"或者 NAK(这里提到的 NAK 是用来启动传输的，它也可用来对数据产生重传的机制)。接收

方发送 **NAK** 信号表示接收方打算用累加和校验；发送字符"**C**"则表示接收方想打算使用 **CRC** 校验。

### 传输过程

当接收方发送的第一个"**C**"或者 **NAK** 到达发送方，发送方认为可以发送第一个数据包，传输已经启动。发送方接着应该将数据以每次 **128** 字节的数据加上包头，包号，包号补码，末尾加上校验和，打包成帧格式传送。

发送方发了第一包后就等待接收方的确认字节 **ACK**，收到接收方传来的 **ACK** 确认，就认为数据包被接收方正确接收，并且接收方要求发送方继续发送下一个包；如果发送方收到接收方传来的 **NAK**(这里，**NAK** 用来告诉发送方重传，不是用来启动传输)字节，则表示接收方请求重发刚才的数据包；如果发送方收到接收方传来的 **CAN** 字节，则表示接收方请求无条件停止传输。

### 结束传输

如果发送方正常传输完全部数据，需要结束传输，正常结束需要发送方发送 **EOT** 字节通知接收方。接收方回以 **ACK** 进行确认。当然接收方也可强制停止传输，当接收方发送 **CAN** 字节给发送方，表示接收方想无条件停止传输，发送方收到 **CAN** 后，不需要再发送 **EOT** 确认。

### 特殊处理

虽然数据包是以 **SOH** 来标志一个信息包的起始的，但在 **SOH** 位置上如果出现 **EOT** 则表示数据传输结束，再也没有数据传过来。接收方首先应确认数据包序号的完整性，通过对数据包序号取补，然后和数

据包序号的补码异或，结果为 0 表示正确，结果不为 0 则发送 NAK 请求重传。

接收方确认数据包序号正确后，然后检查是否期望的序号。如果不是期望得到的数据包序号，说明发生严重错误，应该发送一个 CAN 来中止传输。如果接收到的数据包的包序号和前一包相同，那么收方会忽略这个重复包，向发送方发出 ACK，准备接收下一个包。

接收方确认了信息包序号的完整性和是正确期望的后，只对 128 字节的数据区段进行算术和校验，结果与帧中最后一个字节（算术校验和）比较，相同发送 ACK，不同发送 NAK。

### xmodem 协议的实现

源代码 xmodem.c 是 xmodem 协议的具体实现内容。可以把源代码和上述协议的具体内容对照理解。

```
void xmodem_ReceiveFile(uint32_t addr)
{
    uint8_t state = 0, old_state = 0xff;
    uint8_t xdata buf[134];
    uint8_t count = 0;
    uint32_t oldticks, ticks;
    uint8_t nread;
    uint8_t packno;
    uint8_t retry_count = 0;
    uint8_t packlen;

    if( g_xmodem_crc_mode == 0)
        packlen = 132;
    else
        packlen = 133;

    while(1)
    {
        tick_Task();
        ticks = time_GetTicks();
```

```
    if(old_state != state)
    {
        old_state = state;

        //uart2_Printf("-----%bd-----\r\n",old_state);
    }
    switch(state)
    {
    case 0:
        if(g_xmodem_crc_mode==0)
            SendAck(CHAR_NAK,1);
        else
            SendAck(CHAR_C,1);
        oldticks = ticks;
        state = 1;
        break;
    case 1:
        if(ticks - oldticks < 3*1000ul)
        {
            nread = uart1_Read(buf,1);
            if(nread == 1)
            {
                if(buf[0] == CHAR_SOH)
                {
                    oldticks = ticks;
                    packno = 1;
                    count = 1;
                    state = 2;
                    break;
                }
            }
            break;
        }
        state = 0;
        break;
    case 2:
        if(ticks - oldticks < 10*1000ul)
        {
            nread = uart1_Read(buf+count,packlen-count);
            count += nread;
            if(count == packlen)
            {
                if(PackIsValid(buf,packno))
```

```
        {
            WriteFlash(addr,buf+3,128);
            addr += 128;
            SendAck(CHAR_ACK,2);
            count = 0;
            packno++;
            state = 3;
            retry_count = 0;
            oldticks = ticks;
        }
        else
        {
            retry_count ++;
            SendAck(CHAR_NAK,3);
            state = 1;
            if(retry_count >= 10)
            {
                SendAck(CHAR_CAN,4);
                goto errexit;
            }
        }
        break;
    }
    break;
}
SendAck(CHAR_NAK,5);
retry_count++;
state = 1;
if(retry_count >= 10)
{
    SendAck(CHAR_CAN,6);
    goto errexit;
}
break;
case 3: /*receive next package head.*/
    if(ticks - oldticks < 10*1000ul)
    {
        nread = uart1_Read(buf,1);
        if(nread == 1)
        {
            if(buf[0] == CHAR_SOH)
            {
                oldticks = ticks;
                count = 1;
            }
        }
    }
}
```

```
        state = 4;
        break;
    }
    else if(buf[0] == CHAR_EOT)
    {
        SendAck(CHAR_ACK, 7);
        //uart2_Printf("file receive finished ok.\r\n");
        goto errexit;
    }
}
break;
}
retry_count++;
if(retry_count == 10)
{
    SendAck(CHAR_CAN, 8);
    goto errexit;
}
SendAck(CHAR_NAK, 9);
oldticks = ticks;
break;
case 4: /*receive remained part of next package.*/
    if(ticks - oldticks < 10*1000ul)
    {
        nread = uart1_Read(buf+count, packlen-count);
        count += nread;
        if(count == packlen)
        {
            if(PackIsValid(buf, packno))
            {
                WriteFlash(addr, buf+3, 128);
                addr += 128;
                SendAck(CHAR_ACK, 10);
                count = 0;
                packno++;
                state = 3;
                retry_count = 0;
                oldticks = ticks;
            }
            else
            {
                retry_count ++;
                SendAck(CHAR_NAK, 11);
                state = 3;
            }
        }
    }
}
```



```

        if(retry_count >= 10)
        {
            SendAck(CHAR_CAN,12);
            goto errexit;
        }
    }
    break;
}
break;
}
SendAck(CHAR_NAK,0);
retry_count++;
state = 3;
if(retry_count >= 10)
{
    SendAck(CHAR_CAN,0);
    goto errexit;
}
break;
default:
    break;
}
}
errexit:
    return;
}

```

上面的代码是接收文件的代码，通过一个 **while** 循环中驱动一个 **switch(state)** 状态机来实现，状态 0 是初始状态，在此状态下面发送 **NAK** 或者 **C** 字符，然后在状态 1 等待 PC 机发来的第一个数据包，如果在 3 秒内未收到数据包则重复发送，如果收到 **SOH** 字符则说明传输开始，转换到状态 2 接收剩余部分，然后对一个完整的数据包进行分析，如果正确则将数据写入 **flash**，然后继续在状态 3 和状态 4 接收剩余的数据包，在接收过程中如果出现错误，超时或者接受到错误字符或者数据包校验错误，则会要求重传或者其他的错误处理。

```

void xmodem_SendFile(uint32_t addr,uint32_t len)
{

```

```
uint8_t state = 0, old_state = 0xff;
uint8_t xdata buf[134];
uint8_t i, sum, count = 0;
uint32_t oldticks, ticks;
uint8_t nwrite, nread;
uint8_t packno = 1;
uint8_t retry_count = 0;
uint8_t packlen = 132;
uint8_t crc_mode = 0;
uint16_t crc;
while(1)
{
    tick_Task();
    ticks = time_GetTicks();
    if(old_state != state)
    {
        old_state = state;

        //uart2_Printf("-----%bd-----\r\n", old_state);
    }
    switch(state)
    {
    case 0:
        retry_count = 0;
        oldticks = ticks;
        state = 1;
        break;
    case 1:
        if(ticks - oldticks < 10*1000ul)
        {
            nread = uart1_Read(buf, 1);
            if(nread == 1)
            {
                if(buf[0] == CHAR_C)
                {
                    crc_mode = 1;
                    state = 2;
                    packlen = 133;
                    packno = 1;
                }
                else if(buf[0] == CHAR_NAK)
                {
                    crc_mode = 0;
                }
            }
        }
    }
```

```
        state = 2;
        packlen = 132;
        packno = 1;
    }
    else if(buf[0] == CHAR_CAN)
    {
        printf("user cancels communication.\r\n");
        state = 100;
    }
    else
    {
        printf("error communication.\r\n");
        state = 100;
    }
}
else
{
    oldticks = ticks;
    retry_count++;
    if(retry_count>= 5)
    {
        printf("can not receive response more than 50
second.exit\r\n");
        state = 100;
    }
}
break;
case 2: /*send packets.*/
    if(len == 0)
    {
        SendAck(CHAR_EOT,0);
        retry_count = 0;
        oldticks = ticks;
        state = 4;
    }
    else
    {
        if(len >= 128) nwrite = 128;
        else
        {
            nwrite = len;
            memset(buf,0x1a,sizeof(buf));
        }
    }
}
```

```
    buf[0] = CHAR_SOH;
    buf[1] = packno;
    buf[2] = packno ^ 0xffu;
    m25p80_Read(&buf[3], nwrite, addr);
    if(crc_mode)
    {
        crc = crc_CalcCCITT16NibbleVer(&buf[3], 128);
        buf[131] = crc >> 8;
        buf[132] = crc;
    }
    else
    {
        for(i=0, sum=0; i<128; i++)
        {
            sum += buf[3+i];
        }
        buf[131] = sum;
    }
    uart1_Write(buf, packlen);
    oldticks = ticks;
    retry_count = 10;
    state = 3;
}
break;
case 3: /*check response*/
    if(ticks - oldticks < 10*1000ul)
    {
        nread = uart1_Read(buf, 1);
        if(nread == 1)
        {
            if(buf[0] == CHAR_ACK)
            {
                addr += nwrite;
                len -= nwrite;
                packno++;
                state = 2;
            }
            else if(buf[0] == CHAR_NAK)
            {
                state = 2;
            }
            else if(buf[0] == CHAR_CAN)
            {
                printf("user cancels communication.\r\n");
            }
        }
    }
}
```

```
        state = 100;
    }
    else
    {
        printf("error communication.\r\n");
        state = 100;
    }
}
else
{
    oldticks = ticks;
    retry_count++;
    if(retry_count>= 5)
    {
        printf("can not receive response more than 50
second.exit\r\n");
        state = 100;
    }
}
break;
case 4: /*Check the last ack.*/
    if(ticks - oldticks < 10*1000ul)
    {
        nread = uart1_Read(buf,1);
        if(nread == 1)
        {
            if(buf[0] == CHAR_ACK)
            {
                state = 100;
            }
            else if(buf[0] == CHAR_CAN)
            {
                printf("user cancels communication.\r\n");
                state = 100;
            }
            else
            {
                printf("error communication.\r\n");
                state = 100;
            }
        }
    }
}
else
```

```

        {
            oldticks = ticks;
            retry_count++;
            if(retry_count>= 5)
            {
                printf("can not receive response more than 50
second.exit\r\n");
                state = 100;
            }
        }
        break;
    case 100:
        goto errexit;
        break;
    default:
        break;
    }
}
errexit:
;
}

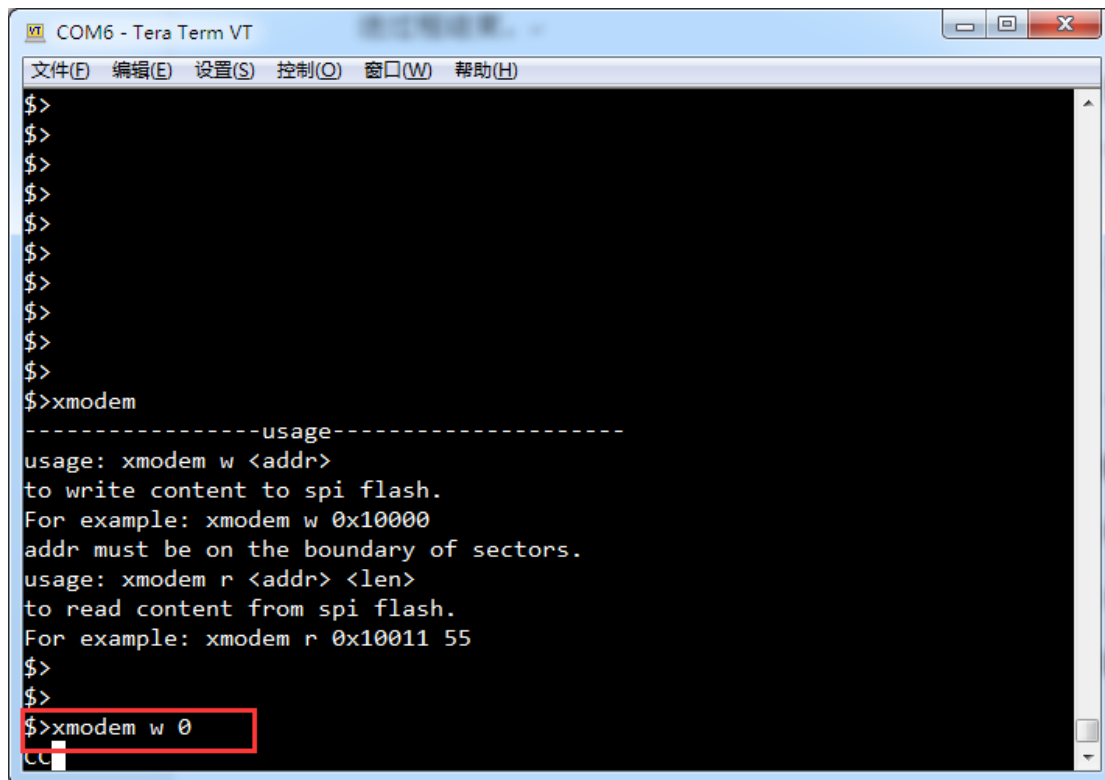
```

上面是发送数据到 PC 机文件的代码，同样也是 `while()` 循环驱动 `switch(state)` 状态机的方式来实现的，状态 0 为初始化状态，状态 1 等待对方发来的启动字符 **NAK** 或者 **C**，收到第一个字符后进入状态 2 发送数据包给 PC 机上面的终端工具程序，然后状态 3 下面等待对方的反馈字符，如果反馈 **ACK** 则发送下一个数据包，如果 **NAK** 则重传，如果 **CAN** 则取消传输等等。当所有数据发送完毕后则发送 **EOT** 字符，然后进入状态 4 等待最后回复 **ACK** 确认，收到确认后整个发送过程结束。

此处 `xmodem` 属于上层的串口协议的应用，底层的串口驱动，收发机制在《串口-底层驱动和上层应用》一文中详细解释。

具体命令执行步骤，以 `teraTerm` 为例,如果要发送文件到单片机，则

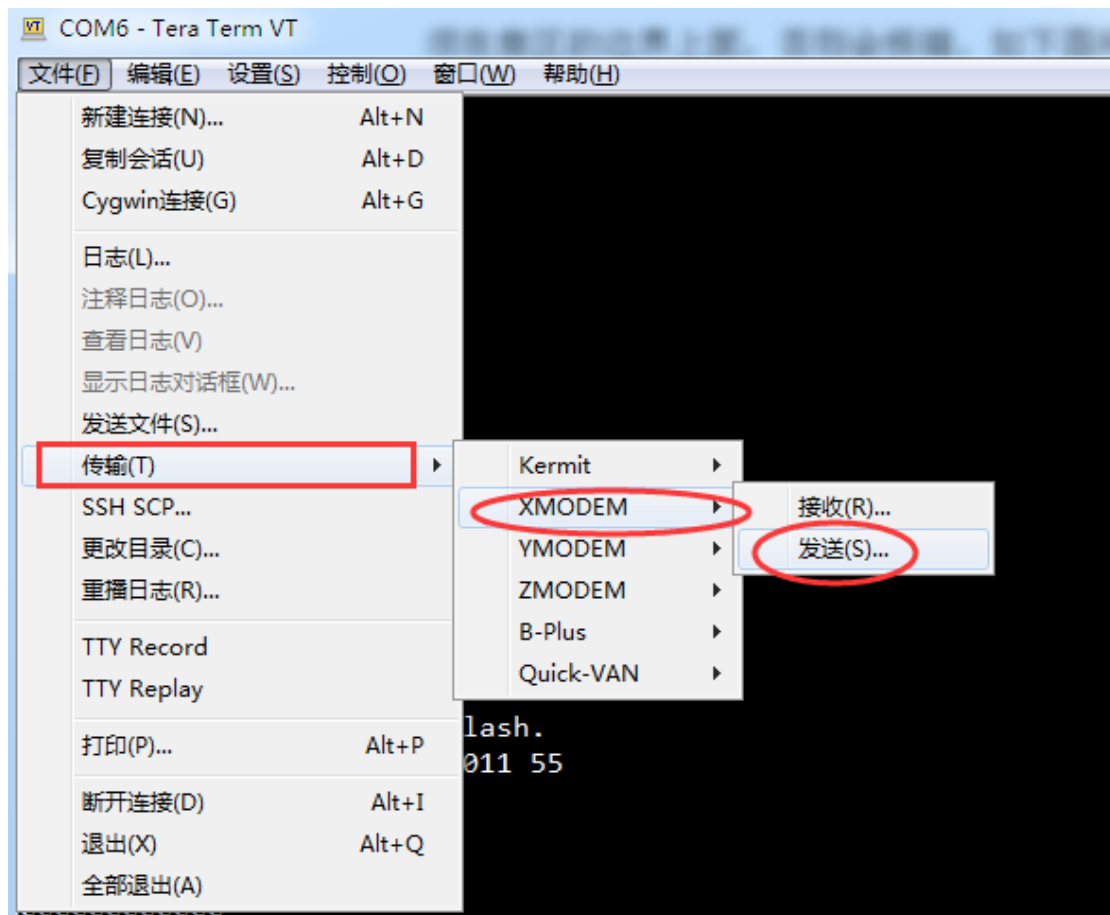
敲入命令 `xmodem w 0` 其中 `w` 表示写数据到单片机系统的 Flash 芯片中，`0` 是写入 Flash 芯片的起始地址，也可以是其他的地址，但必须在扇区的边界上面，否则会报错。如下图所示：



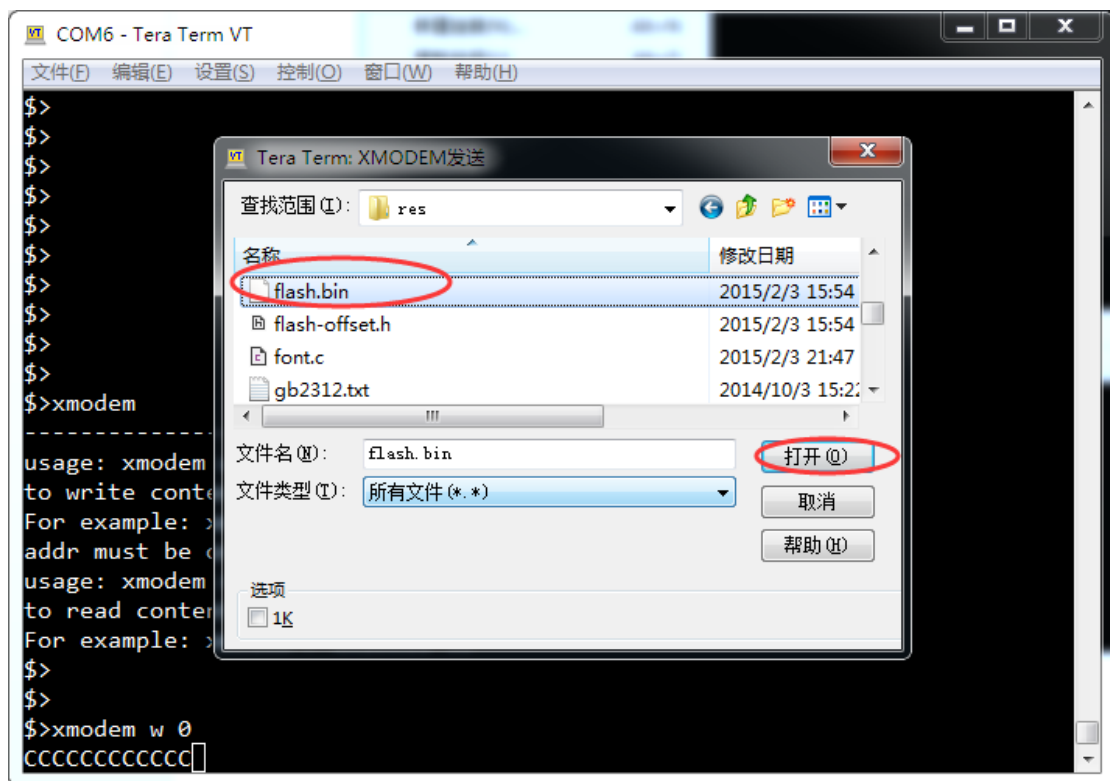
The screenshot shows a Tera Term VT terminal window titled "COM6 - Tera Term VT". The menu bar includes "文件(F)", "编辑(E)", "设置(S)", "控制(O)", "窗口(W)", and "帮助(H)". The terminal output shows a series of "\$>" prompts. After typing "xmodem", a usage message is displayed: "-----usage-----", "usage: xmodem w <addr>", "to write content to spi flash.", "For example: xmodem w 0x10000", "addr must be on the boundary of sectors.", "usage: xmodem r <addr> <len>", "to read content from spi flash.", "For example: xmodem r 0x10011 55". The prompt "\$>" appears twice more. Finally, the command "\$>xmodem w 0" is entered and highlighted with a red rectangle.

```
$>
$>
$>
$>
$>
$>
$>
$>
$>
$>
$>xmodem
-----usage-----
usage: xmodem w <addr>
to write content to spi flash.
For example: xmodem w 0x10000
addr must be on the boundary of sectors.
usage: xmodem r <addr> <len>
to read content from spi flash.
For example: xmodem r 0x10011 55
$>
$>
$>xmodem w 0
```

然后再执行菜单命令：

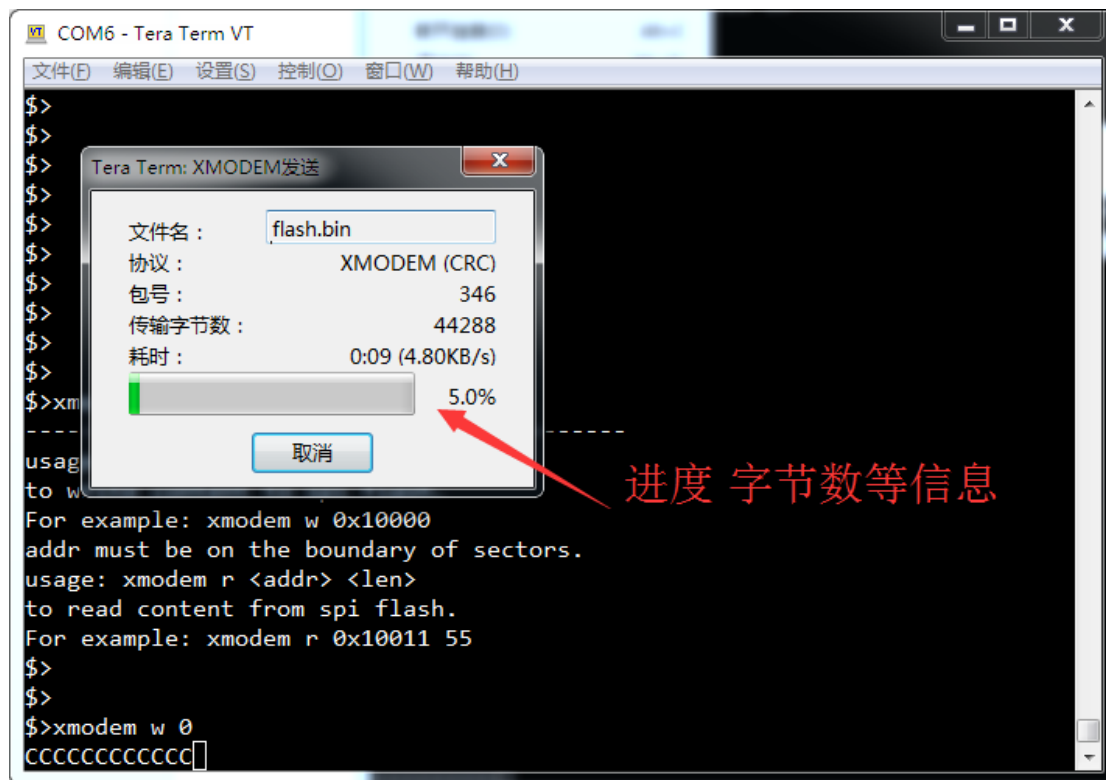


然后选择待发送的文件：



然后则开始传输了：

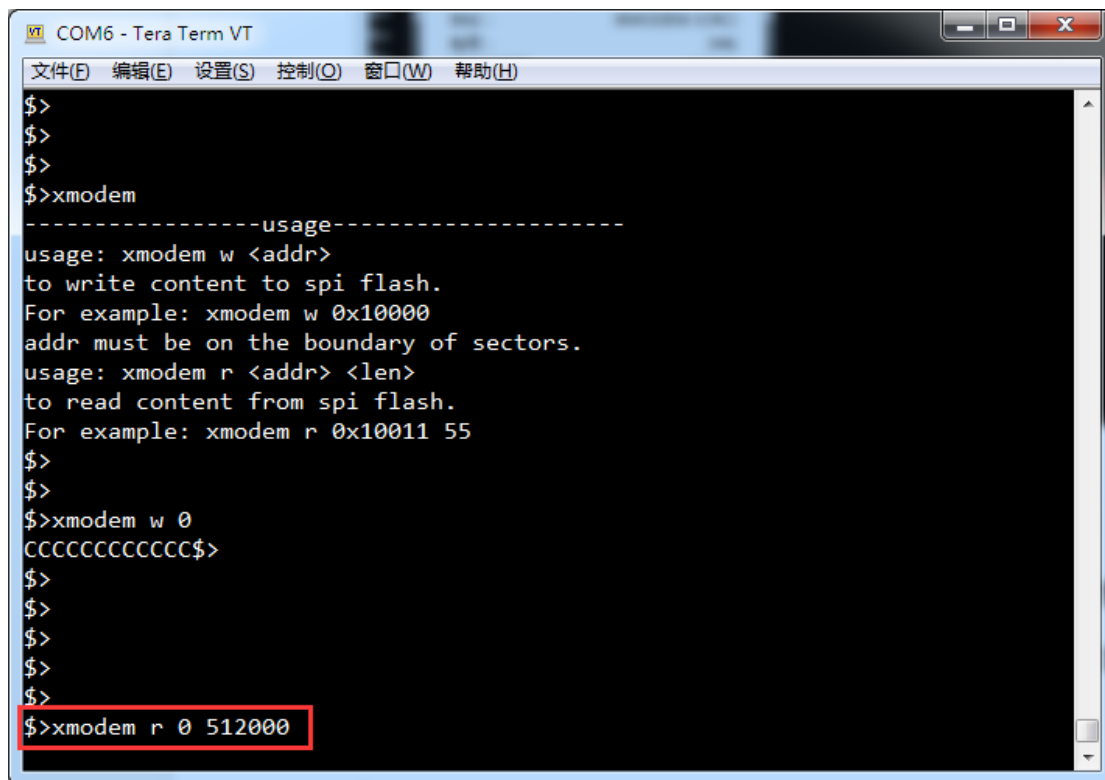




传输过程中会显示进度，传输的字节数等信息。

如果要把 Flash 中内容发送给 PC 机文件，则需要执行：

xmodem r 0 512000 这样的命令，r 表示从 Flash 中读取数据，而 0 表示所要读取的 Flash 的内容的起始地址。512000 则是要读取的数据长度。

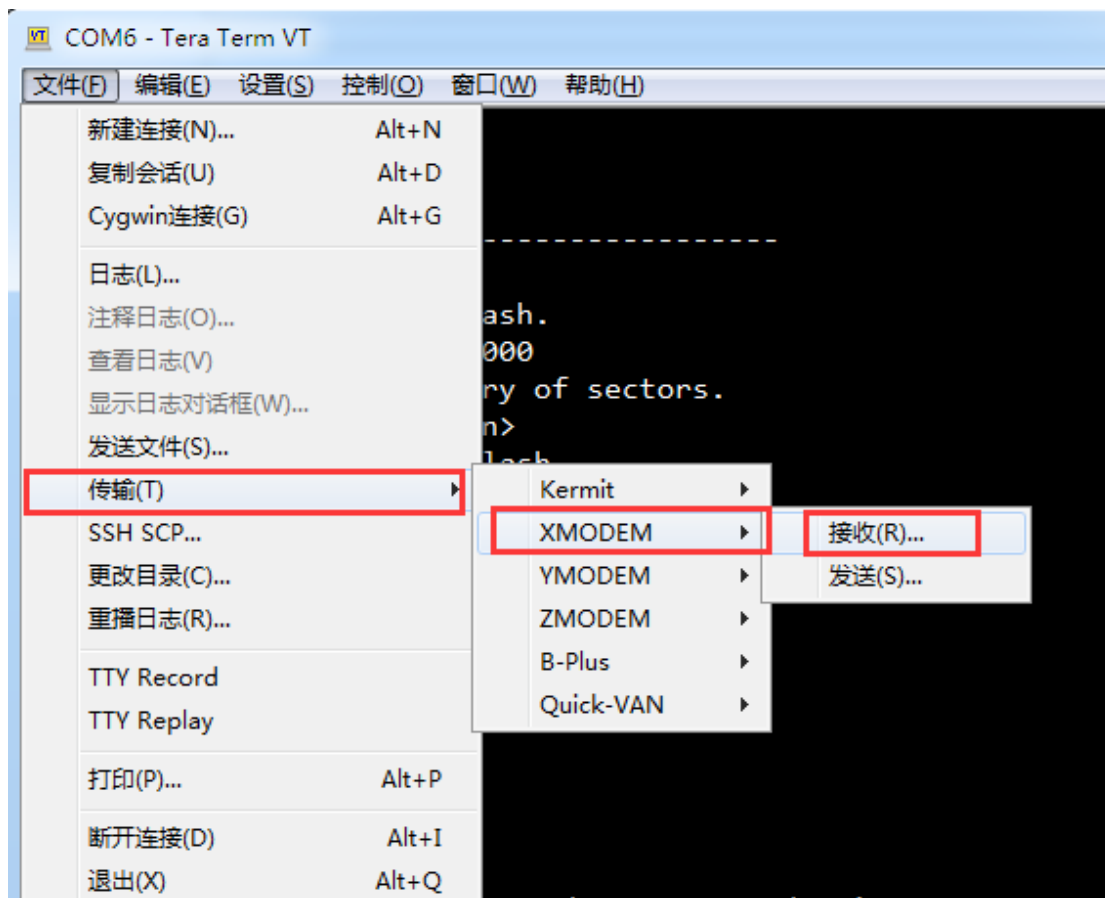


COM6 - Tera Term VT

文件(F) 编辑(E) 设置(S) 控制(O) 窗口(W) 帮助(H)

```
$>  
$>  
$>  
$>xmodem  
-----usage-----  
usage: xmodem w <addr>  
to write content to spi flash.  
For example: xmodem w 0x10000  
addr must be on the boundary of sectors.  
usage: xmodem r <addr> <len>  
to read content from spi flash.  
For example: xmodem r 0x10011 55  
$>  
$>  
$>xmodem w 0  
CCCCCCCCCCCC$>  
$>  
$>  
$>  
$>  
$>  
$>xmodem r 0 512000
```

敲入命令后，单片机进入等待传输状态，然后执行菜单项：



同样需要选择要存放数据的文件，然后开始传输数据的过程。

## 8.源代码下载

本文配套的 demo 工程、源代码在产品随附资料中。



深东电子： 专注 STC 单片机应用

深东电子网站 <http://www.shendongmcu.com>

深东电子淘宝 <http://shendongmcu.taobao.com>