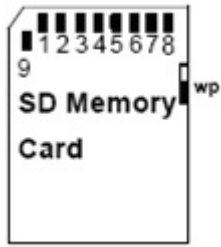


SD 卡管脚定义及 C 语言讲解

(1) **SD** 卡的引脚定义：



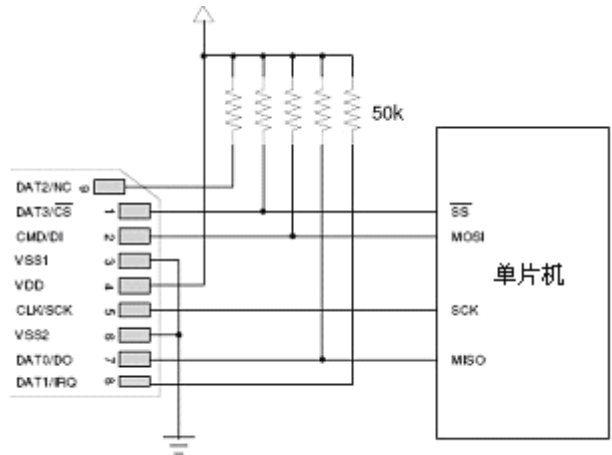
SD 卡引脚功能详述：

引脚 编号	SD 模式			SPI 模式		
	名称	类型	描述	名称	类型	描述
1	CD/DAT3	IO 或 PP	卡检测/ 数据线 3	#CS	I	片选
2	CMD	PP	命令/ 回应	DI	I	数据输入
3	V _{SS1}	S	电源地	VSS	S	电源地
4	V _{DD}	S	电源	VDD	S	电源
5	CLK	I	时钟	SCLK	I	时钟
6	V _{SS2}	S	电源地	VSS2	S	电源地
7	DAT0	IO 或 PP	数据线 0	DO	O 或 PP	数据输出
8	DAT1	IO 或 PP	数据线 1	RSV		
9	DAT2	IO 或 PP	数据线 2	RSV		

注：S：电源供给 I：输入 O：采用推拉驱动的输出

PP：采用推拉驱动的输入输出

SD 卡 SPI 模式下与单片机的连接图：



SD 卡支持两种总线方式：SD 方式与 SPI 方式。其中 SD 方式采用 6 线制，使用 CLK、CMD、DAT0~DAT3 进行数据通信。而 SPI 方式采用 4 线制，使用 CS、CLK、DataIn、DataOut 进行数据通信。SD 方式时的数据传输速度与 SPI 方式要快，采用单片机对 SD 卡进行读写时一般都采用 SPI 模式。采用不同的初始化方式可以使 SD 卡工作于 SD 方式或 SPI 方式。这里只对其 SPI 方式进行介绍。

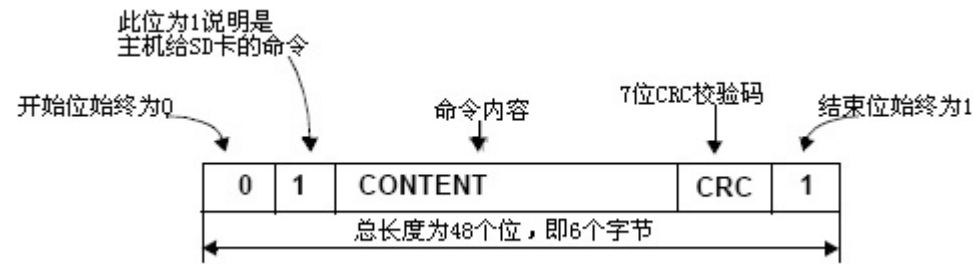
（2）SPI 方式驱动 SD 卡的方法

SD 卡的 SPI 通信接口使其可以通过 SPI 通道进行数据读写。从应用的角度来看，采用 SPI 接口的好处在于，很多单片机内部自带 SPI 控制器，不光给开发上带来方便，同时也见降低了开发成本。然而，它也有不好的地方，如失去了 SD 卡的性能优势，要解决这一问题，就要用 SD 方式，因为它提供更大的总线数据带宽。SPI 接口的选用是在上电初始时向其写入第一个命令时进行的。以下介绍 SD 卡的驱动方法，只实现简单的扇区读写。

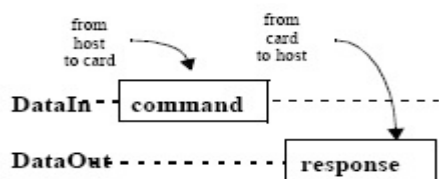
1) 命令与数据传输

1. 命令传输

SD 卡自身有完备的命令系统，以实现各项操作。命令格式如下：



命令的传输过程采用发送应答机制，过程如下：



每一个命令都有自己命令应答格式。在 **SPI** 模式中定义了三种应答格式，如下表所示：

字节	位	含义
1	7	开始位，始终为 0
	6	参数错误
	5	地址错误
	4	擦除序列错误
	3	CRC 错误
	2	非法命令
	1	擦除复位
	0	闲置状态
字节	位	含义
1	7	开始位，始终为 0
	6	参数错误
	5	地址错误
	4	擦除序列错误
	3	CRC 错误
	2	非法命令
	1	擦除复位
	0	闲置状态
2	7	溢出，CSD 覆盖
	6	擦除参数
	5	写保护非法
	4	卡 ECC 失败
	3	卡控制器错误
	2	未知错误
	1	写保护擦除跳过，锁 / 解锁失败
	0	锁卡
字节	位	含义
1	7	开始位，始终为 0
	6	参数错误
	5	地址错误
	4	擦除序列错误
	3	CRC 错误
	2	非法命令

	1	擦除复位
	0	闲置状态
2~5	全部	操作条件寄存器，高位在前

写命令的例程：

```

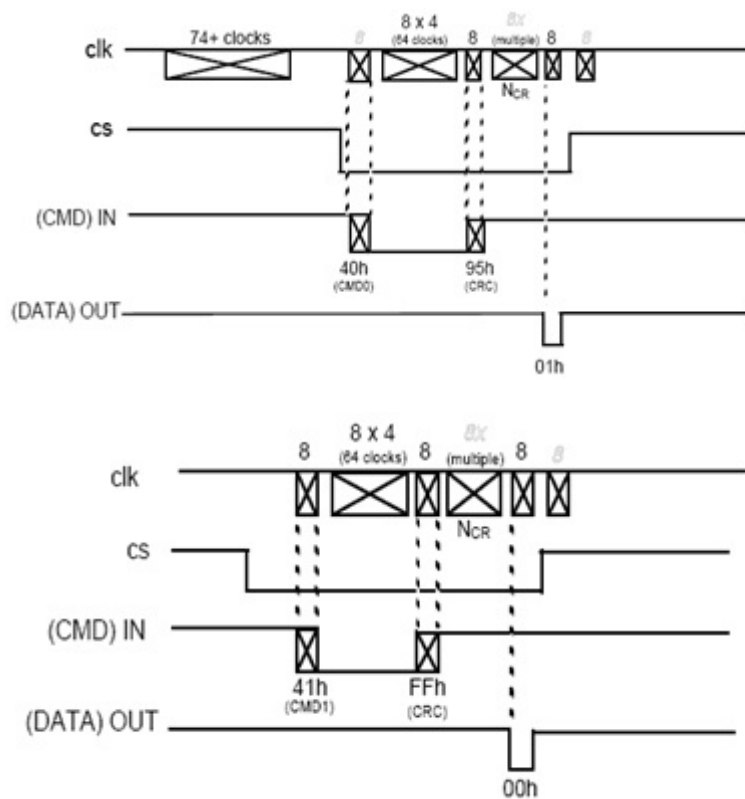
1.  //-----
   -----
2.    向 SD 卡中写入命令，并返回回应的第二个字节
3.  //-----
   -----
4.  unsigned char Write_Command_SD(unsigned char *CMD)
5.  {
6.      unsigned char tmp;
7.      unsigned char retry=0;
8.      unsigned char i;
9.
10.     //禁止 SD 卡片选
11.     SPI_CS=1;
12.     //发送 8 个时钟信号
13.     Write_Byte_SD(0xFF);
14.     //使能 SD 卡片选
15.     SPI_CS=0;
16.
17.     //向 SD 卡发送 6 字节命令
18.     for (i=0;i<0x06;i++)
19.     {
20.         Write_Byte_SD(*CMD++);
21.     }
22.
23.     //获得 16 位的回应
24.     Read_Byte_SD(); //read the first byte,ignore it.
25.     do
26.     { //读取后 8 位
27.         tmp = Read_Byte_SD();
28.         retry++;
29.     }
30.     while ((tmp==0xff) && (retry<100));
31.     return(tmp);
32. }

```

2) 初始化

SD 卡的初始化是非常重要的，只有进行了正确的初始化，才能进行后面的各项操作。在初始化过程中，SPI 的时钟不能太快，否则会造初始化失败。在初始化成功后，应尽量提高 SPI 的速率。在刚开始要先发送至少 74 个时钟信号，这是必须的。在很多读者的实验中，很多是因为疏忽了这一点，而使初始化不成功。随后就是写入两个命令 **CMD0** 与 **CMD1**，使 SD 卡进入 SPI 模式

初始化时序图：



初始化例程：

```

1.  //-----
   -----
2.      初始化 SD 卡到 SPI 模式
3.  //-----
   -----
4.  unsigned char SD_Init()
5.  {

```

```
6.     unsigned char retry,temp;
7.     unsigned char i;
8.     unsigned char CMD[] = {0x40,0x00,0x00,0x00,0x00,0x95};
9.     SD_Port_Init(); //初始化驱动端口
10.
11.     Init_Flag=1; //将初始化标志置 1
12.
13.     for (i=0;i<0x0f;i++)
14.     {
15.         Write_Byte_SD(0xff); //发送至少 74 个时钟信号
16.     }
17.
18.     //向 SD 卡发送 CMD0
19.     retry=0;
20.     do
21.     { //为了能够成功写入 CMD0, 在这里写 200 次
22.         temp=Write_Command_SD(CMD);
23.         retry++;
24.         if(retry==200)
25.         { //超过 200 次
26.             return(INIT_CMD0_ERROR); //CMD0 Error!
27.         }
28.     }
29.     while (temp!=1); //回应 01h, 停止写入
30.
31.     //发送 CMD1 到 SD 卡
32.     CMD[0] = 0x41; //CMD1
33.     CMD[5] = 0xFF;
34.     retry=0;
35.     do
36.     { //为了能成功写入 CMD1, 写 100 次
37.         temp=Write_Command_SD(CMD);
38.         retry++;
39.         if(retry==100)
40.         { //超过 100 次
41.             return(INIT_CMD1_ERROR); //CMD1 Error!
42.         }
43.     }
44.     while (temp!=0); //回应 00h 停止写入
45.
46.     Init_Flag=0; //初始化完毕, 初始化标志清零
47.
48.     SPI_CS=1; //片选无效
49.     return (0); //初始化成功
```

```
50. }
51.
52.
```

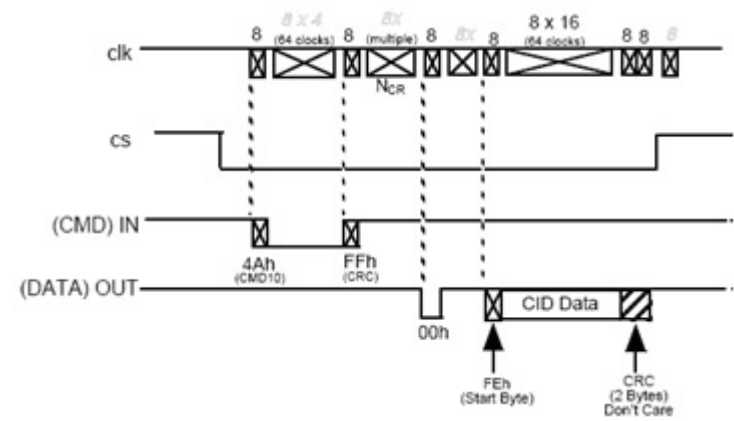
3) 读取 CID

CID 寄存器存储了 SD 卡的标识码。每一个卡都有唯一的标识码。

CID 寄存器长度为 128 位。它的寄存器结构如下：

名称	域	数据宽度	CID 划分
生产标识号	MID	8	[127:120]
OEM/应用标识	OID	16	[119:104]
产品名称	PNM	40	[103:64]
产品版本	PRV	8	[63:56]
产品序列号	PSN	32	[55:24]
保留	—	4	[23:20]
生产日期	MDT	12	[19:8]
CRC7 校验合	CRC	7	[7:1]
未使用，始终为 1	—	1	[0:0]

它的读取时序如下：



与此时序相对应的程序如下：

```
1.  //-----
   -----
2.      读取 SD 卡的 CID 寄存器    16 字节    成功返回 0
3.  //-----
   -----
4.  unsigned char Read_CID_SD(unsigned char *Buffer)
```

```

5. {
6.     //读取 CID 寄存器的命令
7.     unsigned char CMD[] = {0x4A, 0x00, 0x00, 0x00, 0x00, 0xFF};
8.     unsigned char temp;
9.     temp=SD_Read_Block(CMD, Buffer, 16); //read 16 bytes
10.    return(temp);
11. }

```

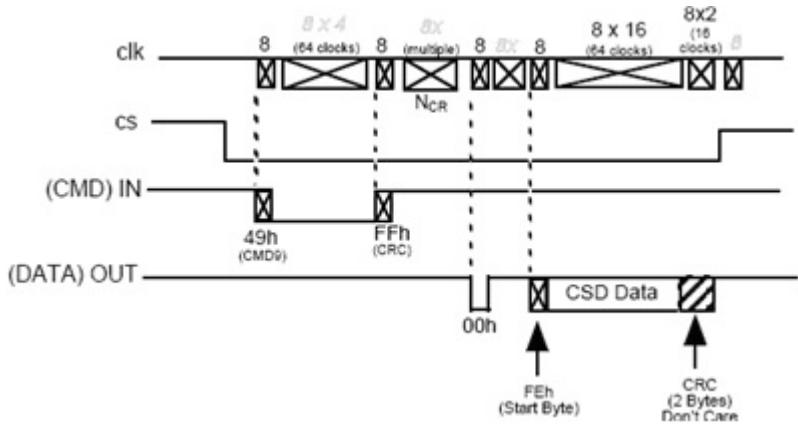
4) 读取 CSD

CSD (Card-Specific Data) 寄存器提供了读写 SD 卡的一些信息。其中的一些单元可以由用户重新编程。具体的 CSD 结构如下：

名称	域	数据宽度	单元类型	CSD 划分
CSD 结构	CSD_STRUCTURE	2	R	[127:126]
保留	-	6	R	[125:120]
数据读取时间 1	TAAC	8	R	[119:112]
数据在 CLK 周期内读取时间 2 (NSAC*100)	NSAC	8	R	[111:104]
最大数据传输率	TRAN_SPEED	8	R	[103:96]
卡命令集合	CCC	12	R	[95:84]
最大读取数据块长	READ_BL_LEN	4	R	[83:80]
允许读的部分块	READ_BL_PARTIAL	1	R	[79:79]
非线写块	WRITE_BLK_MISALIGN	1	R	[78:78]
非线读块	READ_BLK_MISALIGN	1	R	[77:77]
DSR 条件	DSR_IMP	1	R	[76:76]
保留	-	2	R	[75:74]
设备容量	C_SIZE	12	R	[73:62]
最大读取电流@V _{DD} min	VDD_R_CURR_MIN	3	R	[61:59]
最大读取电流@V _{DD} max	VDD_R_CURR_MAX	3	R	[58:56]
最大写电流@V _{DD} min	VDD_W_CURR_MIN	3	R	[55:53]
最大写电流@V _{DD} max	VDD_W_CURR_MAX	3	R	[52:50]
设备容量乘子	C_SIZE_MULT	3	R	[49:47]
擦除单块使能	ERASE_BLK_EN	1	R	[46:46]
擦除扇区大小	SECTOR_SIZE	7	R	[45:39]
写保护群大小	WP_GRP_SIZE	7	R	[38:32]
写保护群使能	WP_GRP_ENABLE	1	R	[31:31]
保留	-	2	R	[30:29]
写速度因子	R2W_FACTOR	3	R	[28:26]
最大写数据块长度	WRITE_BL_LEN	4	R	[25:22]

允许写的部分	WRITE_BL_PARTIAL	1	R	[21:21]
保留	-	5	R	[20:16]
文件系统群	FILE_OFRMAT_GRP	1	R/W	[15:15]
拷贝标志	COPY	1	R/W	[14:14]
永久写保护	PERM_WRITE_PROTECT	1	R/W	[13:13]
暂时写保护	TMP_WRITE_PROTECT	1	R/W	[12:12]
文件系统	FIL_FORMAT	2	R/W	[11:10]
保留	-	2	R/W	[9:8]
CRC	CRC	7	R/W	[7:1]
未用，始终为 1	-	1		[0:0]

读取 CSD 的时序：



相应的程序例程如下：

```
1.  //-----  
2.      读 SD 卡的 CSD 寄存器    共 16 字节    返回 0 说明读取成功  
3.  //-----  
4.  unsigned char Read_CSD_SD(unsigned char *Buffer)  
5.  {  
6.      //读取 CSD 寄存器的命令  
7.      unsigned char CMD[] = {0x49,0x00,0x00,0x00,0x00,0xFF};  
8.      unsigned char temp;  
9.      temp=SD_Read_Block(CMD,Buffer,16); //read 16 bytes  
10.     return(temp);  
11. }
```

4) 读取 SD 卡信息

综合上面对 CID 与 CSD 寄存器的读取，可以知道很多关于 SD 卡的信息，以下程序可以获取这些信息。如下：

```
1.  //-----  
    -----  
2.  //返回  
3.  // SD 卡的容量，单位为 M  
4.  // sector count and multiplier MB are in  
5.  u08 == C_SIZE / (2^(9-C_SIZE_MULT))  
6.  // SD 卡的名称  
7.  //-----  
    -----  
8.  void SD_get_volume_info()  
9.  {  
10.     unsigned char i;  
11.     unsigned char c_temp[5];  
12.     VOLUME_INFO_TYPE SD_volume_Info,*vinf;  
13.     vinf=&SD_volume_Info; //Init the pointoer;  
14. /读取 CSD 寄存器  
15.     Read_CSD_SD(sectorBuffer.dat);  
16. //获取总扇区数  
17.     vinf->sector_count = sectorBuffer.dat[6] & 0x03;  
18.     vinf->sector_count <= 8;  
19.     vinf->sector_count += sectorBuffer.dat[7];  
20.     vinf->sector_count <= 2;  
21.     vinf->sector_count += (sectorBuffer.dat[8] & 0xc0) >> 6;  
22. // 获取 multiplier  
23.     vinf->sector_multiply = sectorBuffer.dat[9] & 0x03;  
24.     vinf->sector_multiply <= 1;  
25.     vinf->sector_multiply += (sectorBuffer.dat[10] & 0x80) >> 7;  
26. //获取 SD 卡的容量  
27.     vinf->size_MB = vinf->sector_count >> (9-vinf->sector_multiply);  
28. // get the name of the card  
29.     Read_CID_SD(sectorBuffer.dat);  
30.     vinf->name[0] = sectorBuffer.dat[3];  
31.     vinf->name[1] = sectorBuffer.dat[4];  
32.     vinf->name[2] = sectorBuffer.dat[5];  
33.     vinf->name[3] = sectorBuffer.dat[6];  
34.     vinf->name[4] = sectorBuffer.dat[7];  
35.     vinf->name[5] = 0x00; //end flag  
36. }
```

```

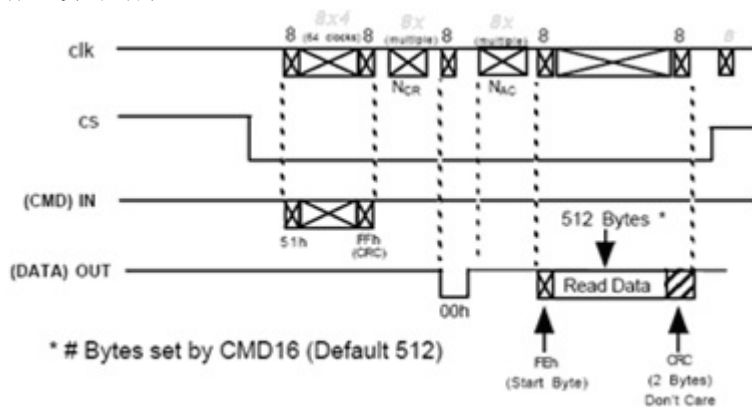
37.          以上程序将信息装载到一个结构体中，这个结构体的定义如下：
38. typedef struct SD_VOLUME_INFO
39. { //SD/SD Card info
40.     unsigned int    size_MB;
41.     unsigned char  sector_multiply;
42.     unsigned int    sector_count;
43.     unsigned char  name[6];
44. } VOLUME_INFO_TYPE;

```

5) 扇区读

扇区读是对 SD 卡驱动的目的之一。SD 卡的每一个扇区中有 512 个字节，一次扇区读操作将把某一个扇区内的 512 个字节全部读出。过程很简单，先写入命令，在得到相应的回应后，开始数据读取。

扇区读的时序：



扇区读的程序例程：

```

1. unsigned char SD_Read_Sector(unsigned long sector,unsigned char *buffer)
2. {
3.     unsigned char retry;
4.     //命令 16
5.     unsigned char CMD[] = {0x51,0x00,0x00,0x00,0x00,0xFF};
6.     unsigned char temp;
7.
8.     //地址变换    由逻辑块地址转为字节地址
9.     sector = sector << 9; //sector = sector * 512
10.
11.     CMD[1] = ((sector & 0xFF000000) >>24 );
12.     CMD[2] = ((sector & 0x00FF0000) >>16 );
13.     CMD[3] = ((sector & 0x0000FF00) >>8 );
14.

```

```

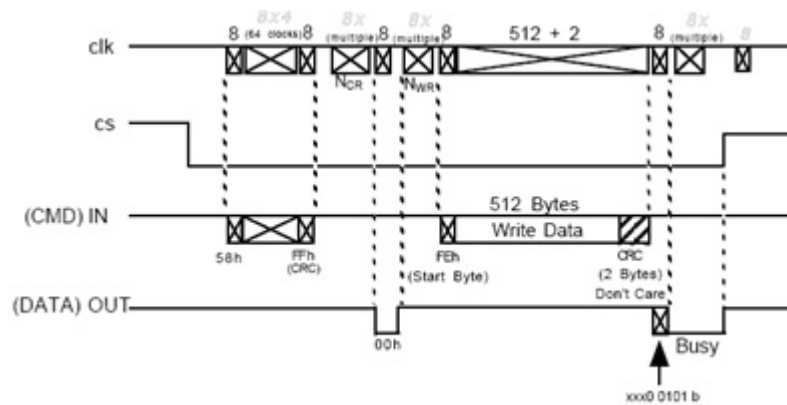
15.    //将命令 16 写入 SD 卡
16.    retry=0;
17.    do
18.    {    //为了保证写入命令  一共写 100 次
19.        temp=Write_Command_MMC(CMD);
20.        retry++;
21.        if(retry==100)
22.        {
23.            return(READ_BLOCK_ERROR); //block write Error!
24.        }
25.    }
26.    while(temp!=0);
27.
28.    //Read Start Byte form MMC/SD-Card (FEh/Start Byte)
29.    //Now data is ready,you can read it out.
30.    while (Read_Byte_MMC() != 0xfe);
31.    readPos=0;
32.    SD_get_data(512,buffer) ;    //512 字节被读出到 buffer 中
33.    return 0;
34. }
35. 其中 SD_get_data 函数如下:
36. //-----
37.     获取数据到 buffer 中
38. //-----
39. void SD_get_data(unsigned int Bytes,unsigned char *buffer)
40. {
41.     unsigned int j;
42.     for (j=0;j<Bytes;j++)
43.         *buffer++ = Read_Byte_SD();
44. }
45.

```

6) 扇区写

扇区写是 SD 卡驱动的另一目的。每次扇区写操作将向 SD 卡的某个扇区中写入 512 个字节。过程与扇区读相似，只是数据的方向相反与写入命令不同而已。

扇区写的时序：



扇区写的程序例程：

```

1.  //-----
   -----
2.      写 512 个字节到 SD 卡的某一个扇区中去    返回 0 说明写入成功
3.  //-----
   -----
4.  unsigned char SD_write_sector(unsigned long addr,unsigned char *Buffer)
5.  {
6.      unsigned char tmp,retry;
7.      unsigned int i;
8.      //命令 24
9.      unsigned char CMD[] = {0x58,0x00,0x00,0x00,0x00,0xFF};
10.     addr = addr << 9; //addr = addr * 512
11.
12.     CMD[1] = ((addr & 0xFF000000) >>24 );
13.     CMD[2] = ((addr & 0x00FF0000) >>16 );
14.     CMD[3] = ((addr & 0x0000FF00) >>8 );
15.
16.     //写命令 24 到 SD 卡中去
17.     retry=0;
18.     do
19.     { //为了可靠写入, 写 100 次
20.         tmp=Write_Command_SD(CMD);
21.         retry++;
22.         if(retry==100)
23.         {
24.             return(tmp); //send commamd Error!
25.         }

```

```

26.     }
27.     while (tmp!=0);
28.
29.
30.     //在写之前先产生 100 个时钟信号
31.     for (i=0;i<100;i++)
32.     {
33.         Read_Byte_SD();
34.     }
35.
36.     //写入开始字节
37.     Write_Byte_MMC(0xFE);
38.
39.     //现在可以写入 512 个字节
40.     for (i=0;i<512;i++)
41.     {
42.         Write_Byte_MMC(*Buffer++);
43.     }
44.
45.     //CRC-Byte
46.     Write_Byte_MMC(0xFF); //Dummy CRC
47.     Write_Byte_MMC(0xFF); //CRC Code
48.
49.
50.     tmp=Read_Byte_MMC(); // read response
51.     if((tmp & 0x1F)!=0x05) // 写入的 512 个字节是未被接受
52.     {
53.         SPI_CS=1;
54.         return(WRITE_BLOCK_ERROR); //Error!
55.     }
56.     //等到 SD 卡不忙为止
57. //因为数据被接受后，SD 卡在向储存阵列中编程数据
58.     while (Read_Byte_MMC() !=0xff) {};
59.
60.     //禁止 SD 卡
61.     SPI_CS=1;
62.     return(0); //写入成功
63. }
64.

```

此上内容在笔者的实验中都已调试通过。单片机采用 STC89LE 单片机（SD 卡的初始化电压为 2.0V~3.6V，操作电压为 3.1V~3.5V，因此不能用 5V 单片机，或进行分压处理），工作于 22.1184M 的时钟下，由于所采用的单片机中没硬件 SPI，采用软件模拟 SPI，因

此读写速率都较慢。如果要半 SD 卡应用于音频、视频等要求高速场合，则需要选用有硬件 SPI 的控制器，或使用 SD 模式，当然这就需要各位读者对 SD 模式加以研究，有了 SPI 模式的基础，SD 模式应该不是什么难事。

