华中科技大学瑞萨实验室

技术报告

K60 的 SD 卡开发总结

Ver0.2

胡春旭

华中科技大学控制科学与工程系

2012 年 5 月 7 日

目录

1 SD 卡简介

SD 卡在现在的日常生活与工作中使用非常广泛，时下已经成为最为通用的数据存储卡。在诸如 MP3、数码相机等设备上也都采用 SD 卡作为其存储设备。SD 卡之所以得到如此广泛的使用，是因为它价格低廉、存储容量大、使用方便、通用性与安全性强等优点。既然它有着这么多优点，那么如果将它加入到单片机应用开发系统中来，将使系统变得更加出色。这就要求对 SD 卡的硬件与读写时序进行研究。对于 SD 卡的硬件结构，在官方的文档上有很详细的介绍，如 SD 卡内的存储器结构、存储单元组织方式等内容。要实现对它的读写，最核心的是它的时序。

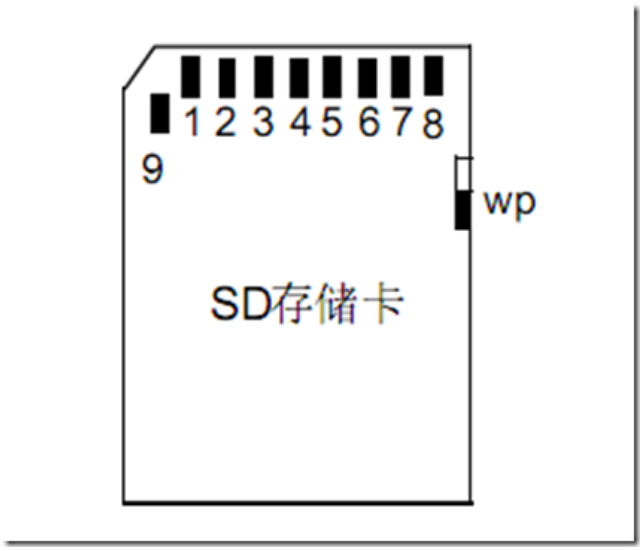
1.1 SD 卡工作方式

SD 卡有两个可选的通讯协议：SD 模式和 SPI 模式 SD 模式是 SD 卡标准的读写方式，但是在选用 SD 模式时，往往需要选择带有 SD 卡控制器接口的 MCU，或者必须加入额外的 SD 卡控制单元以支持 SD 卡的读写。然而，大多数 MCU 都没有集成 SD 卡控制器接口，若选用 SD 模式通讯就无形中增加了产品的硬件成本。在 SD 卡数据读写时间要求不是很严格的情况下，选用 SPI 模式可以说是一种最佳的解决方案。因为在 SPI 模式下，通过四条线就可以完成所有的数据交换，并且目前市场上很多 MCU 都集成有现成的 SPI 接口电路，采用 SPI 模式对 SD 卡进行读写操作可大大简化硬件电路的设计。

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。

1.2 SD 卡物理接口

我们看到的 SD 卡接口如下所示，包含 9 个引脚和一个写保护开关：



其引脚定义如下：

引脚	SD 模式			SPI 模式		
	名称[1]	类型	描述	名称	类型	描述
1	CD/DAT3[2]	I/O/PP[3]	卡的检测/数据线[Bit 3]	CS	I	片选（低电平有效）
2	CMD	PP[4]	命令／响应	DI	I[5]	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	I/O/PP	数据线[Bit 0]	DO	O/PP	数据输出
8	DAT1	I/O/PP	数据线[Bit 1]	RSV		
9	DAT2	I/O/PP	数据线[Bit 2]	RSV		

注：

1. S：电源；I：输入；O：推挽输出；PP：推挽 I/O。

2. 扩展的 DAT 线（DAT1 ~ DAT3）在上电后处于输入状态。它们在执行 SET_BUS_WIDTH 命令后作为 DAT 线操作。当不使用 DAT1 ~ DAT3 线时，主机应使自己的 DAT1~DAT3 线处于输入模式。这样定义是为了与 MMC 卡保持兼容。

3. 上电后，这条线为带 50KΩ 上拉电阻的输入线（可以用于检测卡是否存在或选择 SPI 模式）。用户可以在正常的数据传输中用 SET_CLR_CARD_DETECT

（ACMD42）命令断开上拉电阻的连接。MMC 卡的该引脚在 SD 模式下为保留引脚，在 SD 模式下无任何作用。

4. MMC 卡在 SD 模式下为：I/O/PP/OD。

5. MMC 卡在 SPI 模式下为：I/PP。

1.3 SD 卡控制流程

1.3.1 SD 卡的 SPI 工作模式

SD 卡在上电初期自动进入 SD 总线模式，在此模式下向 SD 卡发送复位命令 CMD0。如果 SD 卡在接收复位命令过程中 CS 低电平有效，则进入 SPI 模式，否则工作在 SD 总线模式。

为了使 SD 卡初始化进入 SPI 模式，我们需要使用的命令有 3 个：CMD0，ACMD41，CMD55（使用 ACMD 类的指令前应先发 CMD55，CMD55 起到一个切换到 ACMD 类命令的作用）。

为什么在使用 CMD0 以后不使用 CMD1？CMD1 是 MMC 卡使用的指令，虽然本文并不想讨论 MMC 卡的问题，但是我还是要说：为了实现兼容性，上电或者发送 CMD0 后，应该首先发送 CMD55+ACMD41 确认是否有回应，如果有回应则为 SD 卡，如果等回应超时，则可能是 MMC 卡，再发 CMD1 确认。

正确的回应内容应该是：

CMD0——0x01（SD 卡处于 in-idle-state）

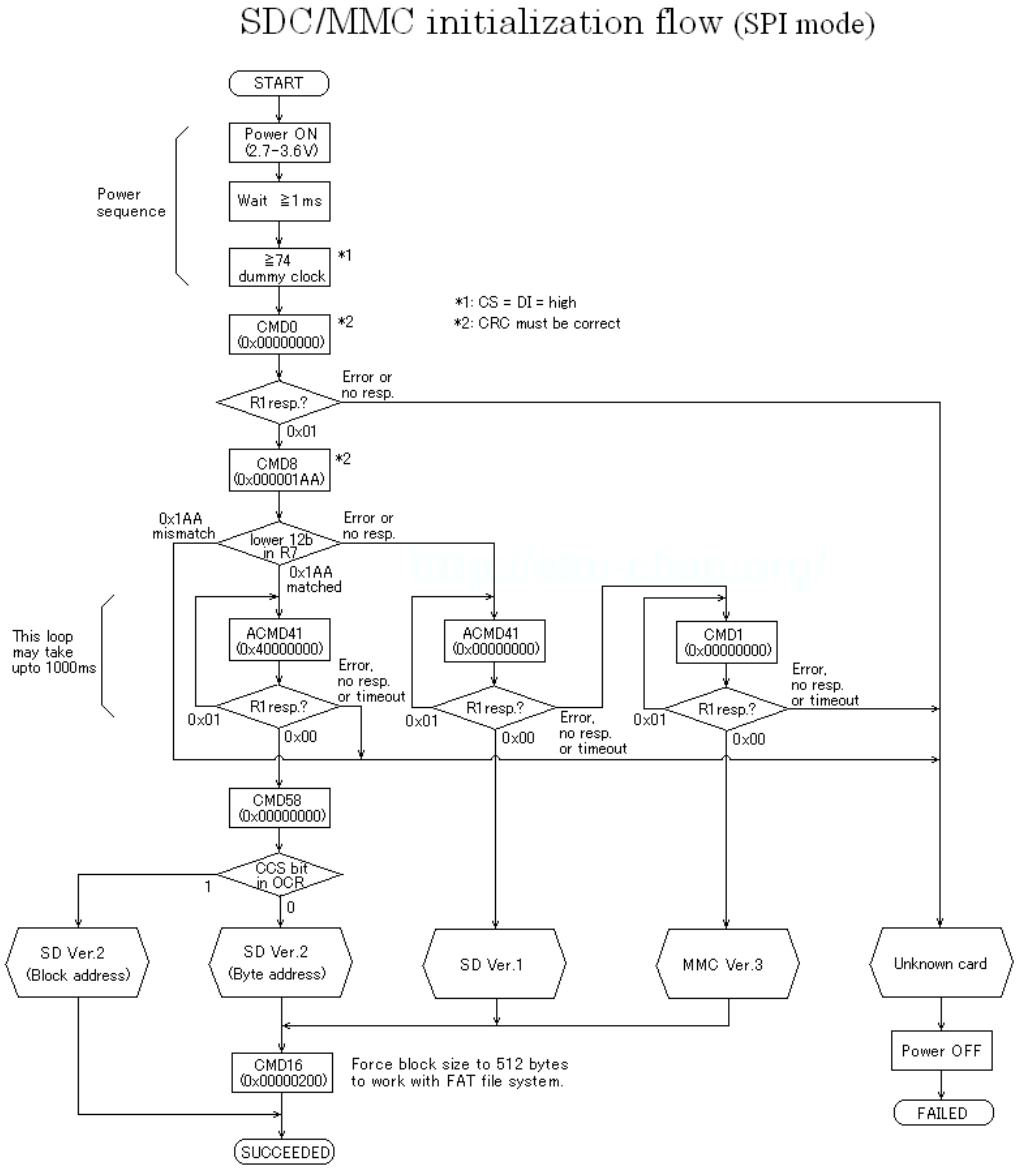
CMD55——0x01（SD 卡处于 in-idle-state）

ACMD41——0x00（SD 卡跳出 in-idle-state，完成初始化准备接受下一条指令）

这里要说的是如果最后的回应内容还是 0x01 的话，可以循环发送 CMD55+ACMD41，直到回应的内容 0x00。

在所有的指令中，唯独 CMD0 特殊，在向 SD 卡发送以前需要向 SD 卡发送 74+个时钟。那么为什么要 74 个 CLK 呢？因为在上电初期，电压的上升过程据 SD 卡组织的计算约合 64 个 CLK 周期才能到达 SD 卡的正常工作电压他们管这个叫做 Supply ramp up time，其后的 10 个 CLK 是为了与 SD 卡同步，之后开始 CMD0

的操作，严格按照此项操作，一定没有问题。



在复位成功之后可以通过 CMD55 和 ACMD41 判断当前电压是否在工作范围内 主机还可以继续通过 CMD10 读取 SD 卡的 CID 寄存器，通过 CMD16 设置数据 Block 长度，通过 CMD9 读取卡的 CSD 寄存器 从 CSD 寄存器中，主机可获知卡容量，支持的命令集等重要参数。

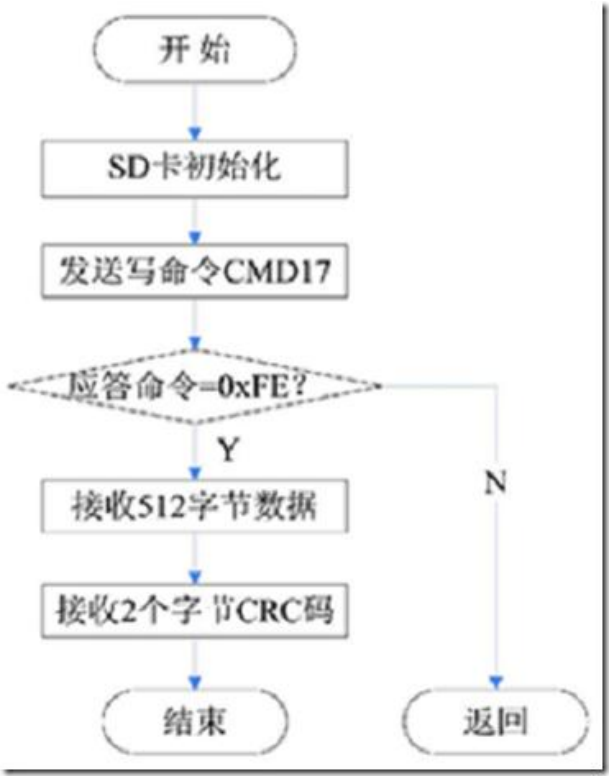
1.3.2 SD 卡读写流程

完成 SD 卡的初始化之后即可进行它的读写操作 SD 卡的读写操作都是通过发送 SD 卡命令完成的 SPI 总线模式支持单块（CMD24）和多块（CMD25）写操作，多块操作是指从指定位置开始写下去，直到 SD 卡收到一个停止命令 CMD12 才停止单块写操作的数据块长度只能是 512 字节单块写入时，命令为 CMD24，当应答为 0 时说明可以写入数据，大小为 512 字节 SD 卡对每个发送给自己的数据块都通过一个应答命令确认，它为 1 个字节长，当低 5 位为 00101 时，表明数据块被正确写入 SD 卡。

在需要读取 SD 卡中数据的时候，读 SD 卡的命令字为 CMD17，接收正确的第一个响应命令字节为 0xFE，随后是 512 个字节的用户数据块，最后为 2 个字节的 CRC 验证码。可见，读写 SD 卡的操作都是在初始化后基于 SD 卡命令和响应完成操作的，写、读 SD 卡的程序流程图如下所示 ：

（1） 写 SD 卡流程

（2） 读 SD 卡流程



1.3.3 SD 卡的操作命令

```
/* 命令响应定义 define command's response */
#define R1 1
#define R1B 2
#define R2 3
#define R3 4

/**********************************************

      SD 卡 SPI 模式下命令集

**********************************************/

/**********  基本命令集 Basic command set ***********/
/* 复位 SD 卡 Reset cards to idle state */
#define CMD0 0
```

```
#define CMD0_R R1

/* 读 OCR 寄存器 Read the OCR (MMC mode, do not use for SD cards) */
#define CMD1 1
#define CMD1_R R1

/* 读 CSD 寄存器 Card sends the CSD */
#define CMD9 9
#define CMD9_R R1

/* 读 CID 寄存器 Card sends CID */
#define CMD10 10
#define CMD10_R R1

/* 停止读多块时的数据传输 Stop a multiple  block (stream) read/write operation */
#define CMD12 12
#define CMD12_R R1B

/* 读 Card_Status 寄存器 Get the addressed card's status register */
#define CMD13 13
#define CMD13_R R2

/*********** 块读命令集 Block read commands ***********/

/* 设置块的长度 Set the block length */
#define CMD16 16
#define CMD16_R R1

/* 读单块 Read a single  block */
#define CMD17 17
#define CMD17_R R1

/* 读多块,直至主机发送 CMD12 为止 Read multiple  blocks until  a CMD12 */
#define CMD18 18
```

```
#define CMD18_R R1

/************  块写命令集 Block write commands **************/
/*  写单块 Write a block of the size selected with CMD16 */
#define CMD24 24
#define CMD24_R R1

/*  写多块 Multiple  block  write until  a CMD12 */
#define CMD25 25
#define CMD25_R R1

/*  写 CSD 寄存器 Program the programmable  bits  of the CSD */
#define CMD27 27
#define CMD27_R R1

/*****************  写保护 Write protection ***************/
/* Set the write protection bit of the addressed group */
#define CMD28 28
#define CMD28_R R1B

/* Clear the write protection bit of the addressed group */
#define CMD29 29
#define CMD29_R R1B

/* Ask the card for the status of the write protection bits */
#define CMD30 30
#define CMD30_R R1

/**************  擦除命令 Erase commands ***************/
/*  设置擦除块的起始地址(只用于 SD 卡) Set the address of the first write block to be
erased(only for SD) */
#define CMD32 32
#define CMD32_R R1
```

```
/* 设置擦除块的终止地址(只用于 SD 卡) Set the address of the last write block to be erased(only for SD) */
#define CMD33 33
#define CMD33_R R1

/* 设置擦除块的起始地址(只用于 MMC 卡) Set the address of the first write block to be erased(only for MMC) */
#define CMD35 35
#define CMD35_R R1

/* 设置擦除块的终止地址(只用于 MMC 卡) Set the address of the last write block to be erased(only for MMC) */
#define CMD36 36
#define CMD36_R R1

/* 擦除所选择的块 Erase the selected write blocks */
#define CMD38 38
#define CMD38_R R1B

/************ 锁卡命令 Lock Card commands **************/
/* 设置/复位密码或上锁/解锁卡 Set/reset the password or lock/unlock the card */
#define CMD42 42
#define CMD42_R      R1B
/* Commands from 42 to 54, not defined here */

/*********** 应用命令 Application-specific commands ***********/
/* 禁止下一个命令为应用命令 Flag that the next command is application-specific */
#define CMD55 55
#define CMD55_R R1

/* 应用命令的通用 I/O General purpose I/O for application-specific commands */
#define CMD56 56
#define CMD56_R R1
```

```
/* 读 OCR 寄存器  Read the OCR (SPI mode only) */
#define CMD58 58
#define CMD58_R R3

/* 使能或禁止 CRC Turn CRC on or off */
#define CMD59 59
#define CMD59_R R1

/***************  应用命令 Application-specific  commands *********/
/* 获取 SD Status 寄存器 Get the SD card's status */
#define ACMD13 13
#define ACMD13_R R2

/* 得到已写入卡中的块的个数 Get the number of written write blocks (Minus errors ) */
#define ACMD22 22
#define ACMD22_R R1

/* 在写之前,设置预先擦除的块的个数 Set the number of write blocks to be pre-erased before writing */
#define ACMD23 23
#define ACMD23_R R1

/* 读取 OCR 寄存器 Get the card's OCR (SD mode) */
#define ACMD41 41
#define ACMD41_R R1

/* 连接/断开 CD/DATA[3]引脚上的上拉电阻 Connect or disconnect the 50kOhm internal pull-up  on CD/DAT[3] */
#define ACMD42 42
#define ACMD42_R R1

/* 读取 SCR 寄存器 Get the SD configuration  register */
#define ACMD51 51
```

```
#define ACMD51_R R1
```

2 SPI 接口简介

SPI(Serial Peripheral Interface--串行外设接口)总线系统是一种同步串行外设接口，它可以使 MCU 与各种外围设备以串行方式进行通信以交换信息。SPI 有三个寄存器分别为：控制寄存器 SPCR，状态寄存器 SPSR，数据寄存器 SPDR。外围设备 FLASHRAM、网络控制器、LCD 显示驱动器、A/D 转换器和 MCU 等。SPI 总线系统可直接与各个厂家生产的多种标准外围器件直接接口，该接口一般使用 4 条线：串行时钟线（SCLK）、主机输入/从机输出数据线 MISO、主机输出/从机输入数据线 MOSI 和低电平有效的从机选择线 SS(有的 SPI 接口芯片带有中断信号线 INT、有的 SPI 接口芯片没有主机输出/从机输入数据线 MOSI)。

SPI 接口是在 CPU 和外围低速器件之间进行同步串行数据传输，在主器件的移位脉冲下，数据按位传输，高位在前，低位在后，为全双工通信，数据传输速度总体来说比 I2C 总线要快，速度可达到几 Mbps。

2.1 四种信号

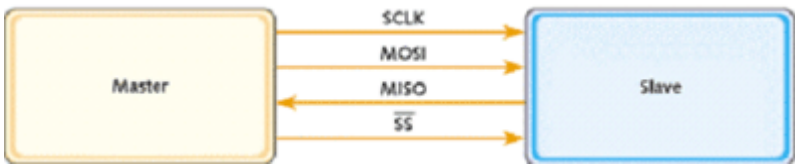
（1）MOSI – 主器件数据输出，从器件数据输入

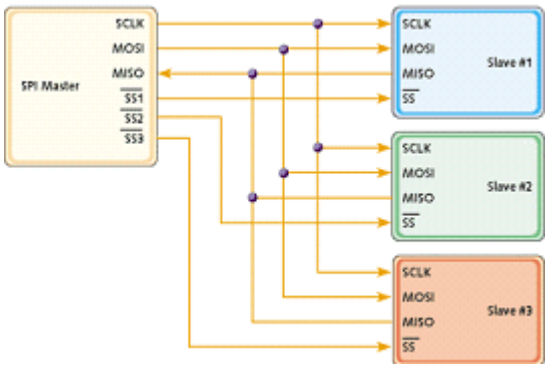
（2）MISO – 主器件数据输入，从器件数据输出

（3）SCLK – 时钟信号，由主器件产生

（4）/SS – 从器件使能信号，由主器件控制,有的 IC 会标注为 CS(Chip select)

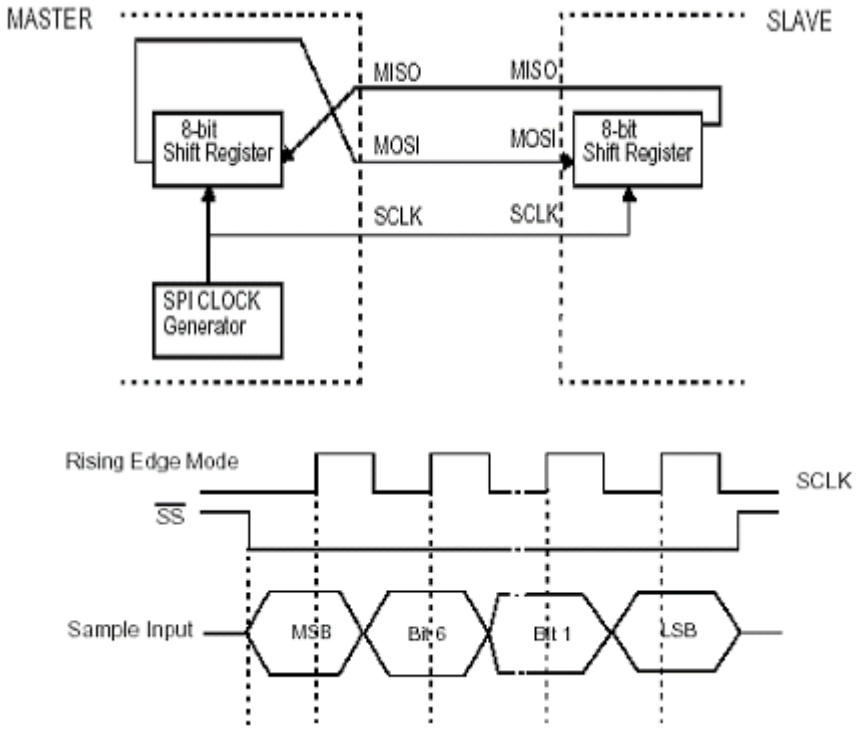
在点对点的通信中，SPI 接口不需要进行寻址操作，且为全双工通信，显得简单高效。多个从器件硬件连接示意图在多个从器件的系统中，每个从器件需要独立的使能信号，硬件上比 I2C 系统要稍微复杂一些。



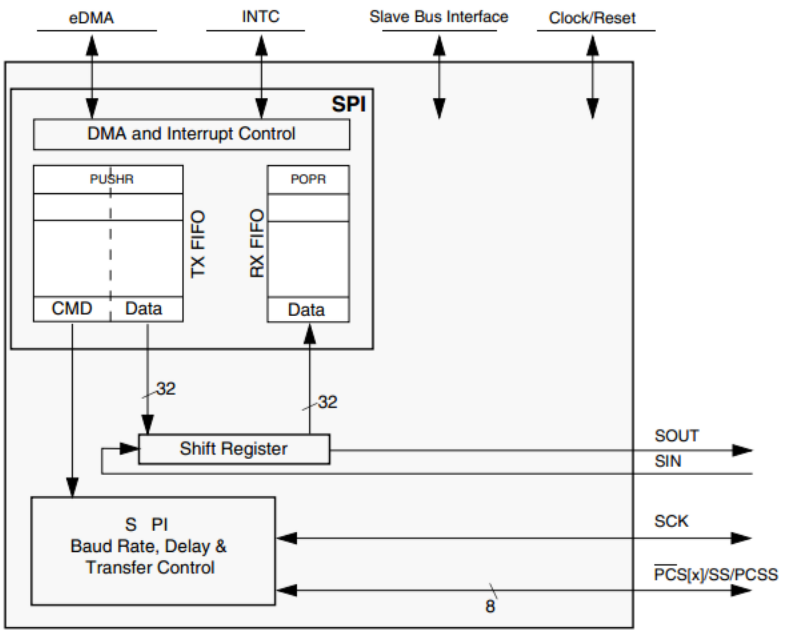


2.2 内部硬件

SPI 接口在内部硬件实际上是两个简单的移位寄存器，传输的数据为 8 位，在主器件产生的从器件使能信号和移位脉冲下，按位传输，高位在前，低位在后。如下图所示，在 SCLK 的下降沿上数据改变，同时一位数据被存入移位寄存器。



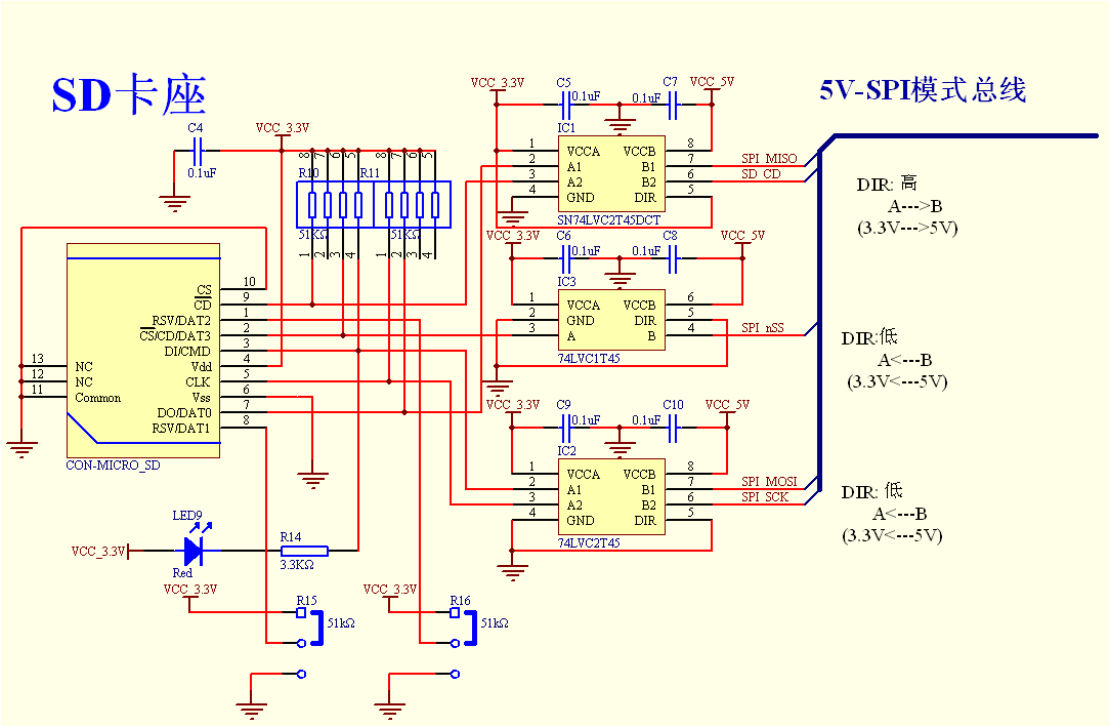
K60 的 SPI 内部结构如下图所示：



3 K60 下 SD 的开发

3.1 硬件电路实现

我们使用的 SD 卡部分电路如下所示：



因为 SD 卡使用的电压是 3.3V，而为了兼容 K60 和 XS128，我们的信号引脚

电压是 5V，这就需要中间的三个升压芯片。

在这个电路中，我们只需要关心总线中的四个信号线：

(1) SPI_SCK：SPI 总线的时钟信号

(2) SPI_nSS：SD 卡使能的片选信号

(3) SPI_MOSI：单片机（主机）的输出信号

(4) SPI_MISO：单片机（主机）的输入信号

我们在程序中也主要是对这四个信号进行操作，在测试过程中需要利用示波器测试这四个信号线的信号，根据时序来验证发送和接受的数据是否正确。

3.2 SPI 底层驱动

SPI 部分的驱动只有三个函数：SPI 初始化、SPI 发送数据、SPI 接收数据。但是这三个函数至关重要，因为涉及到芯片的具体特性，所以需要详细阅读芯片资料的相关部分。这一部分的驱动没有问题，我们的 SD 卡驱动也就完成了一大半的工作。

3.2.1 SPI 初始化

```
ER spi_init(uint8_t spino, uint8_t master)
{
    if((spino < 0) || (spino > SPI_NO_GET(SPI2)))
    {
        return (E_ID);
    }

    SPI_MemMapPtr base_addr = spi_get_base_address(spino);

    /* 使能 SPI 模块时钟，配置 SPI 引脚功能 */
    if(SPI_MOD_SET(spino) == SPI0)
    {
        SIM_SCGC6 |= SIM_SCGC6_DSPI0_MASK;
```

```
    /* PORT_PCR_MUX(0x2) ： SPI 功能
     * PORT_PCR_DSE_MASK ； Drive Strength Enable */
    gpio_init(PORT_NO_GET(SD_CS), PIN_NO_GET(SD_CS), OUT_PUT, HIGH_POWER);    /* PCS0 */
    PORTA_PCR15 = 0 | PORT_PCR_MUX(0x2) | PORT_PCR_DSE_MASK; /* SCK */
    PORTA_PCR16 = 0 | PORT_PCR_MUX(0x2) | PORT_PCR_DSE_MASK; /* SOUT */
    PORTA_PCR17 = 0 | PORT_PCR_MUX(0x2);                     /* SIN */
}
else if(SPI_MOD_SET(spino) == SPI1)
{
    SIM_SCGC6 |= SIM_SCGC6_SPI1_MASK;
    PORTE_PCR4 = 0 | PORT_PCR_MUX(0x2) | PORT_PCR_DSE_MASK; /* PCS0 */
    PORTE_PCR2 = 0 | PORT_PCR_MUX(0x2) | PORT_PCR_DSE_MASK; /* SCK */
    PORTE_PCR1 = 0 | PORT_PCR_MUX(0x2) | PORT_PCR_DSE_MASK; /* SOUT */
    PORTE_PCR3 = 0 | PORT_PCR_MUX(0x2);                     /* SIN */
}
else
{
    SIM_SCGC3 |= SIM_SCGC3_SPI2_MASK;
}

SPI_MCR_REG(base_addr) = 0
                | SPI_MCR_CLR_TXF_MASK      /* Clear the Tx FIFO counter. */
                | SPI_MCR_CLR_RXF_MASK      /* Clear the Rx FIFO counter. */
                | SPI_MCR_HALT_MASK;        /* Starts and stops DSPI transfers */

/* 根据主从机模式设置工作模式 */
if(master == MASTER)
{
    SPI_MCR_REG(base_addr) |= SPI_MCR_MSTR_MASK;    /* Master/Slave Mode Select */

    SPI_CTAR_REG(base_addr,0) = 0
                    | SPI_CTAR_DBR_MASK         /* Double Baud Rate */
                    | SPI_CTAR_FMSZ(0x07)       /* Frame Size； 8bit */
                    | SPI_CTAR_PDT_MASK         /* 延时因子为 7 */
```

```
                        | SPI_CTAR_BR(0x8);             /* Selects the scaler value for the baud rate.   */
                        //| SPI_CTAR_CPOL_MASK;        /* Clock Polarity */
                        //| SPI_CTAR_CPHA_MASK;        /* Clock Phase */
    }
    else
    {
        SPI_CTAR_SLAVE_REG(base_addr,0) = 0
                            | SPI_CTAR_SLAVE_FMSZ(0x07)
                            | SPI_CTAR_SLAVE_CPOL_MASK
                            | SPI_CTAR_SLAVE_CPHA_MASK;
   }

   SPI_SR_REG(base_addr) = SPI_SR_EOQF_MASK      /* End of Queue Flag */
                         | SPI_SR_TFUF_MASK       /* Transmit FIFO Underflow Flag */
                         | SPI_SR_TFFF_MASK       /* Transmit FIFO Fill Flag */
                         | SPI_SR_RFOF_MASK       /* Receive FIFO Overflow Flag */
                         | SPI_SR_RFDF_MASK;     /* Receive FIFO Drain Flag */

    SPI_MCR_REG(base_addr) &= ~SPI_MCR_HALT_MASK;    /* start */

    return (E_OK);
}
```

这里需要注意几个问题：

1. SPI_CTAR_FMSZ(0x07)：这里对应资料，我们使用的八位模式，是这个寄存器值+1 之后的位模式，而不是寄存器的值是多少，就是多少位模式。
2. SPI_CTAR_CPOL 和 SPI_CTAR_CPHA：这两位是设置时钟信号的极性和有效值的，根据 fatfs 官网的移植说明，最好都置 0。
3. SPI_MCR_PCSIS：这个寄存器是设置使能信号的极性的，我们需要的是低有效，所以这个寄存器要置 0。

3.2.2 SPI 发送数据

```
void spi_snd(uint8_t spino, uint8_t data[], uint32_t len)
```

```
{
    uint32_t i = 0;
    SPI_MemMapPtr base_addr = spi_get_base_address(spino);

    SPI_SR_REG(base_addr) = (SPI_SR_EOQF_MASK
          | SPI_SR_TFUF_MASK
          | SPI_SR_TFFF_MASK
          | SPI_SR_RFOF_MASK
          | SPI_SR_RFDF_MASK);

    SPI_MCR_REG(base_addr) |= SPI_MCR_CLR_TXF_MASK       /* Clear TX FIFO */
                            | SPI_MCR_CLR_RXF_MASK;    /* Clears the RX Counter */

    for (i = 0; i < len; i++)
    {
        if (i == (len - 1))
        {
            SPI_PUSHR_REG(base_addr) = 0
                    | SPI_PUSHR_CTAS(0)                 /* Clock and Transfer Attributes Select */
                    | SPI_PUSHR_EOQ_MASK                /* End Of Queue */
                    | SPI_PUSHR_TXDATA(data[i]);    /* Transmit Data */
        }
        else
        {
            SPI_PUSHR_REG(base_addr) = 0
                    | SPI_PUSHR_CONT_MASK
                    | SPI_PUSHR_CTAS(0)
                    | SPI_PUSHR_TXDATA(data[i]);
        }
    }

    /* 等待数据发送完毕 */
    while((SPI_SR_REG(base_addr) & SPI_SR_TCF_MASK)==0);
    SPI_SR_REG(base_addr) |= SPI_SR_TCF_MASK;
```

```
}
```

这里需要注意几个问题：

1. SPI_SR_TCF：这是数据发送完毕的标志位。在发送数据之后一定要确定数据已经发送完毕，否则无法读取出正确的数据。确定发送完毕之后还要对标志位置 1 清零。

3.2.3 SPI 接收数据

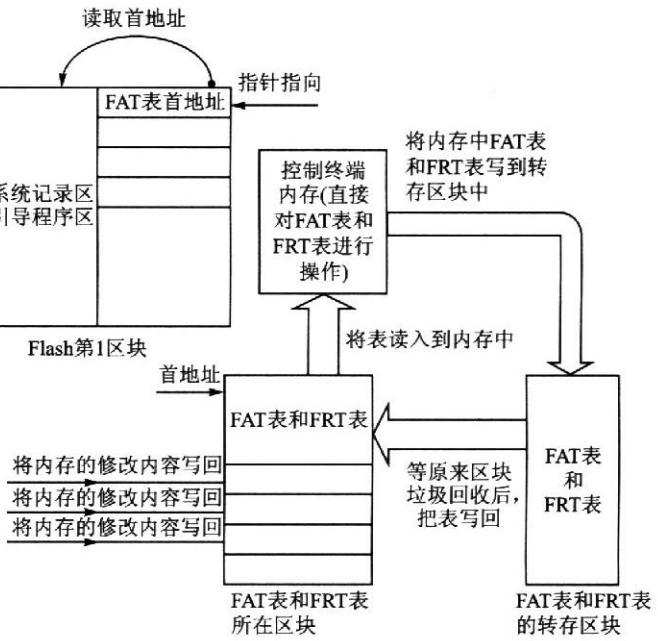
```
uint8_t spi_rcv(uint8_t spino, uint8_t data[])
{
    SPI_MemMapPtr base_addr = spi_get_base_address(spino);

    if(SPI_SR_REG(base_addr) & SPI_SR_RFDF_MASK)    /* Rx FIFO is not empty */
    {
        data[0] = (uint8_t)SPI_POPR_REG(base_addr);        /* Received Data */
        SPI_SR_REG(base_addr) |= SPI_SR_RFDF_MASK;         /* The RFDF bit can be cleared by writing 1  */
        return 1;
    }
    SPI_SR_REG(base_addr) = (SPI_SR_EOQF_MASK
        | SPI_SR_TFUF_MASK
        | SPI_SR_TFFF_MASK
        | SPI_SR_RFOF_MASK
        | SPI_SR_RFDF_MASK);

    SPI_MCR_REG(base_addr) |= SPI_MCR_CLR_TXF_MASK      /* Clear the Tx FIFO counter. */
                            | SPI_MCR_CLR_RXF_MASK;     /* Clear the Rx FIFO counter. */

    return 0;
}
```

3.3 Fatfs 文件系统移植

3.3.1 文件系统简介

文件系统是操作系统用于明确磁盘或分区上的文件的方法和数据结构；即在磁盘上组织文件的方法。也指用于存储文件的磁盘或分区，或文件系统种类。操作系统中负责管理和存储文件信息的软件机构称为文件管理系统，简称文件系统。文件系统由三部分组成：与文件管理有关软件、被管理文件以及实施文件管理所需数据结构。从系统角度来看，文件系统是对文件存储器空间进行组织和分配，负责文件存储并对存入的文件进行保护和检索的系统。具体地说，它负责为用户建立文件，存入、读出、修改、转储文件，控制文件的存取，当用户不再使用时撤销文件等。

下图是文件系统的操作原理：



3.3.2 Fatfs 文件系统

FatFs 是一个通用的文件系统模块，用于在小型嵌入式系统中实现 FAT 文件

系统。 FatFs 的编写遵循 ANSI C，因此不依赖于硬件平台。它可以嵌入到便宜的微控制器中，如 8051, PIC, AVR, SH, Z80, H8, ARM 等等，不需要做任何修改。

FatFs 提供下面的函数：

函数	功能
f_mount	注册/注销一个工作区域（Work Area）
f_open	打开/创建一个文件
f_close	关闭一个文件
f_read	读文件
f_write	写文件
f_lseek	移动文件读/写指针
f_truncate	截断文件
f_sync	冲洗缓冲数据 Flush Cached Data
f_opendir	打开一个目录
f_readdir	读取目录条目
f_getfree	获取空闲簇 Get Free Clusters
f_stat	获取文件状态
f_mkdir	创建一个目录
f_unlink	删除一个文件或目录
f_chmod	改变属性（Attribute）
f_utime	改变时间戳（Timestamp）
f_rename	重命名/移动一个文件或文件夹
f_mkfs	在驱动器上创建一个文件系统
f_forward	直接转移文件数据到一个数据流
f_gets	读一个字符串
f_putc	写一个字符
f_puts	写一个字符传
f_printf	写一个格式化的字符磁盘 I/O 接口

f_tell	获取当前读/写文件的指针
f_size	获取文件的大小

因为 FatFs 模块完全与磁盘 I/O 层分开，因此需要下面的函数来实现底层物理磁盘的读写与获取当前时间。底层磁盘 I/O 模块并不是 FatFs 的一部分，并且必须由用户提供。资源文件中也包含有范例驱动。

disk_initialize	Initialize disk drive 初始化磁盘驱动器
disk_status	Get disk status 获取磁盘状态
disk_read	Read sector(s) 读扇区
disk_write	Write sector(s) 写扇区
disk_ioctl	Control device dependent features 设备相关的控制特性
get_fattime	Get current time 获取当前时间

3.3.3 Fatfs 文件系统移植

Fatfs 的官网上有很多例程，我们使用了其中的一个例程进行移植。需要进行修改的几个文件是 sd.c、sd.h、mmc.c。将这三个文件加入到工程目录中，然后把 Fatfs 的底层文件复制到 library 文件夹下面。diskio.h、ff.h、ffconf.h、integer.h 这几个头文件需要加入到工程的的头文件里。

首先需要移植 SD 卡初始化的驱动程序。首先在 sd.c 中建立我们文件系统需要使用的全局变量：

```
FATFS       Fatfs;                  /* File system object for each logical drive */
FIL         FileObj;                /* File objects */
char        filename[5]="a.txt";    /* File name */
```

然后相应修改 sd_init 函数。其中大部分是不需要动的，我们只需要加入一些为了调试方便使用的开关 LED 灯函数。

```
void sd_init(FATFS *fs, FIL *fp, char *fln)
{
    /* SD 卡和 SPI 初始化 */
```

```
    while (disk_initialize(0))
    {
        light_open(LIGHT7);
    }
    light_open(LIGHT6);

    /* 创建文件系统 */
    if (!f_mount(0, fs))                /* mount a file system */
    {
         sd_create_file(fp, fln);    /* create a file */
    }
    light_close(LIGHT6);
    light_close(LIGHT7);
}
```

这里面最重要的就是 disk_initialize 函数，它调用底层 SPI 对 SD 卡进行初始化，如果 SPI 底层驱动配置的没有问题，这里就比较好调试。我之前就是因为底层写好，导致在这里卡了一个星期都没有初始化成功。

disk_initialize 函数在 mmc.c 中。这个文件是文件系统的底层部分，也是 SPI 的高层部分，需要将设计到硬件的部分都进行修改。

我们从 disk_initialize 函数开始看起。首先就是 power_on 函数，这个函数主要是 SPI 底层的初始化，直接调用底层即可。

```
static void power_on (void)
{
    spi_init(SPI_NO_GET(SPI0), MASTER);
}
```

然后是 FCLK_SLOW 宏定义，这是一个系列的宏定义，是用来设置 SD 卡的读写速度的。

mmc.c 中

```
#define    FCLK_SLOW()       SD_SPI_LOW_SPEED()        /* Set slow clock (100k-400k)  */
#define    FCLK_FAST()  SD_SPI_HIGH_SPEED()       /* Set fast clock (depends on the CSD) */
```

sd.h 中

```
/* 设置 SD 的 SPI 速度  */
#define SD_SPI_VERY_HIGH_SPEED()      SPI_CTAR_REG(SPI0_BASE_PTR,0)  |= 0x00
#define SD_SPI_HIGH_SPEED()           SPI_CTAR_REG(SPI0_BASE_PTR,0)  |= 0x01
#define SD_SPI_LOW_SPEED()            SPI_CTAR_REG(SPI0_BASE_PTR,0)  |= 0x08
```

这里需要注意，在 SD 卡初始化过程中速度不能太快，也不能太慢，需要保持在 100khz~400khz 之间，我们选用了 200khz 左右的频率，在初始化结束后，SD 卡的读写那就是越快越好了。

设置好初始化速度之后，就进入了 SD 卡的控制流程，可以参考 1.3.1 的 SD 卡的 SPI 工作模式一章中的流程图进行配置。

为了提高移植性，我们将调用 spi 收发数据的函数又进行了一次封装：

```
/*-----------------------------------------------------------------------*/
/* Exchange a byte between PIC and MMC via SPI    (Platform dependent)   */
/*-----------------------------------------------------------------------*/
#define xmit_spi(dat)       xchg_spi(dat)
#define rcvr_spi()          xchg_spi(0xFF)

static
BYTE xchg_spi (BYTE dat)
{
    uint8_t rcv_data = 0;
    spi_snd(SPI_NO_GET(SPI0),  &dat, 1);
    spi_rcv(SPI_NO_GET(SPI0),  &rcv_data);

    return (BYTE)rcv_data;
}

static
void rcvr_spi_m (BYTE *p)
{
    BYTE data = 0xff;
    spi_snd(SPI_NO_GET(SPI0),  &data, 1);
    spi_rcv(SPI_NO_GET(SPI0),  p);
}
```

这几个宏定义和函数就是不同形式的 SPI 收发数据封装。

另外，SD 卡的使能信号我们也进行了宏定义的封装：

mmc.c

```
#define CS_LOW()    SD_SPI_SELECT()             /* MMC CS = L */
#define CS_HIGH()  SD_SPI_DESELECT()          /* MMC CS = H */
```

sd.h

/* 设置 SD 卡片选 */

```
#define SD_SPI_SELECT()                  gpio_set(PORT_NO_GET(SD_CS),
PIN_NO_GET(SD_CS),  LOW_POWER)
#define SD_SPI_DESELECT()                gpio_set(PORT_NO_GET(SD_CS),
PIN_NO_GET(SD_CS),  HIGH_POWER)
```

因为我们使用的使能信号是利用 gpio 口实现的，所以初始化和反转操作都调用 gpio 函数即可。

根据流程，我们在 SD 卡上电之后需要至少等待 72 个周期的脉冲时间，才能让 SD 卡完全上电成功，所以利用 for (n = 10; n; n--) rcvr_spi();我们发送了 80 个脉冲（一个 bit 就是一个脉冲）。

SD 卡上电完成后，就需要让 SD 卡进入 SPI 的工作模式下。首先发送 CMD0 让 SD 卡进入 idle 状态，如果 SD 卡返回 1，说明已经进入了 idle 状态，如果返回其他，那只能继续发送 CMD0，直到有 1 返回，否则初始化失败，需要认真检查是不是 SPI 底层驱动或是电路存在问题。如果 SD 卡成功进入 idle 状态，接下来就是按照那个流程图继续向下初始化，最后会得到 SD 卡的类型，说明初始化成功。

在初始化结束后，将 SD 卡的读写速度提高，返回 SD 类型，回到 sd.c 中的 sd_init 函数，进入下一步创建文件系统。

创建文件系统调用的是 sd_create_file 函数：

```
void sd_create_file(FIL *fp, char *fln)
{
    /* if file exist create a new one */
    while (f_open(fp, fln, FA_CREATE_NEW|FA_WRITE)  == FR_EXIST)
    {
```

```
            fln[0] = fln[0] + 1;
        }
}
```

检查 SD 卡中有无和当前要创建的文件重名的文件，如果有则对需要创建的文件重命名，如果没有就直接创建文件。

文件系统创建完毕之后，sd_init 函数的使命就完成了，我们的初始化工作也就告一段落了。接下来就是向刚才创建的文件中写数据了。

向 SD 卡写数据的格式很简单，和 printf 类似：

```
f_printf(&FileObj, "Hello SD Card!!\n");
f_printf(&FileObj, "This is a test!!\n");
f_printf(&FileObj, "1 2 3 4 5 6 7 8 9\n");
f_printf(&FileObj, "! @ $ & * ( ) + -\n");
```

最后注意一定要把文件关闭，否则 SD 卡中的文件中是找不到数据的。

```
sd_close_file(&FileObj); /* 关闭文件 */
```

3.3.4 Fatfs 文件系统的使用

如果调试没问题，接下来我们就可以在程序中使用 Fatfs 了。

首先要建立几个变量：

```
FATFS   Fatfs;                  /* File system object for each logical drive */
FIL     test_data;              /* File objects */
char    test_data_name[FILE_NAME_LENGHT]="a_data.txt";   /* File name */
```

一般来说 Fatfs 是一个文件系统，在一个 SD 卡只有一个。test_data 是一个文件对象，我们可以同时在 SD 卡中建立多个文件对象，每一个对应一个文件，在不同需求的时候写不同的文件。test_data_name 是对应文件的文件名，不同文件相应有一个文件名数组。

然后要在程序一开始挂载和创建文件系统，就是利用前面介绍的初始化函数：

```
sd_init(&Fatfs, &test_data, test_data_name);          /* 初始化 SD 卡，并创建文件 */
```

此时就会把 SD 卡挂载，然后在其中创建一个文件。

接着就可以写文件了：

```
f_printf(&test_data, "Hello SD Card!!\n");
sd_close_file(&test_data); /* 关闭文件 */
```

写文件的时候一定要确保对应的文件是打开的，可以利用下面的语句：

```
if (test_data.fs)
{
    f_printf(&test_data, "Hello SD Card!!\n");
    f_printf(&test_data , "This is a test!!\n");
    f_printf(&test_data, "1 2 3 4 5 6 7 8 9\n");
    f_printf(&test_data "! @ $ & * ( ) + -\n");
}
sd_close_file(&test_data); /* 关闭文件 */
```

fs 是对应文件的指针，如果文件打开了就是非 0。最后一定要记住文件写完了或者程序执行完了，一定要关闭文件，否则数据不会在文件里。

参考文献

1. SPI 模式下 MCU 对 SD 卡的控制及操作命令：
   http://www.cnblogs.com/zyqgold/archive/2012/01/02/2310340.html
2. How to Use MMC/SDC：
   http://elm-chan.org/docs/mmc/mmc_e.html
3. About SPI：
   http://elm-chan.org/docs/spi_e.html
4. K60 用 spi 写 MicroSD 卡
   http://www.ourdev.cn/thread-5467131-1-1.html
5. FatFs Generic FAT File System Module：
   http://elm-chan.org/fsw/ff/00index_e.html