零死角玩转STM32

与野火同行 乐意惬无边

1 原创教程，完全开源。

2 通俗易懂，详尽解读。

3 由浅入深，结合实操。

4 配套板子，全面玩转。

5 强强联合，不断更新。

野火团队 Wild Fire Team

0、 友情提示

《零死角玩转 STM32》系列教程由初级篇、中级篇、高级篇、系统篇、四个部分组成，根据野火 STM32 开发板旧版教程升级而来，且经过重新深入编写，重新排版，更适合初学者，步步为营，从入门到精通，从裸奔到系统，让您零死角玩转 STM32。M3 的世界，与野火同行，乐意惬无边。

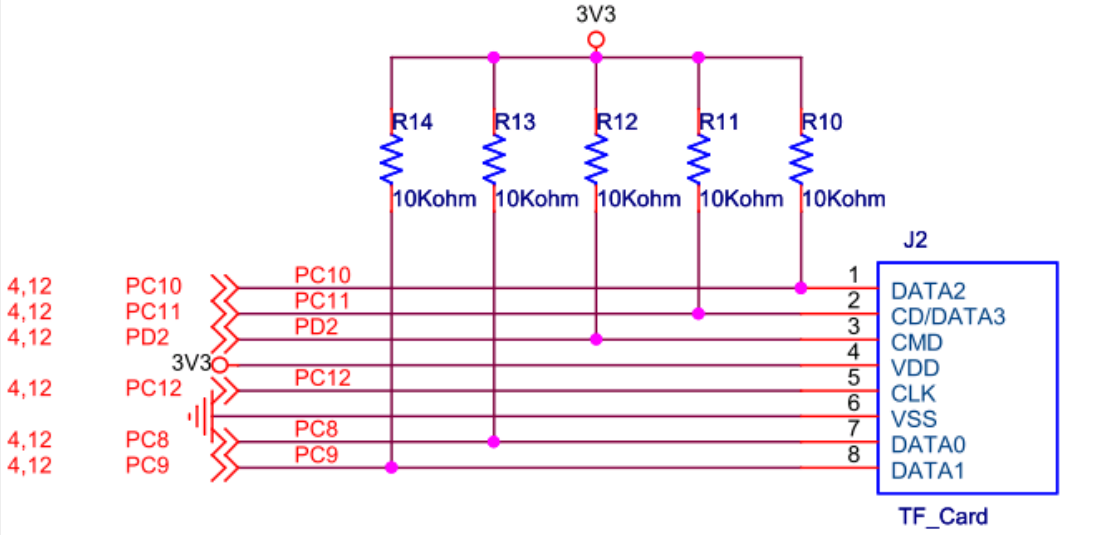
另外，野火团队历时一年精心打造的《STM32 库开发实战指南》将于今年 10 月份由机械工业出版社出版，该书的排版更适于纸质书本阅读以及更有利于查阅资料。内容上会给你带来更多的惊喜。是一本学习 STM32 必备的工具书。敬请期待！

1、SDIO（4bit + DMA、支持 SDHC）

1.1 实验描述及工程文件清单

实验描述	MicroSD 卡(SDIO 模式)测试实验，采用 4bit 数据线模式。没有跑文件系统，只是单纯地读 block 并将测试信息通过串口 1 在电脑的超级终端上 打印出来。
硬件连接	PC12-SDIO-CLK：CLK PC10-SDIO-D2 ：DATA2 PC11-SDIO-D3：CD/DATA3 PD2-SDIO-CMD ：CMD PC8-SDIO-D0：DATA0 PC9-SDIO-D1：DATA1
用到的库文件	startup/start_stm32f10x_hd.c CMSIS/core_cm3.c CMSIS/system_stm32f10x.c FWlib/stm32f10x_gpio.c FWlib/stm32f10x_rcc.c FWlib/stm32f10x_usart.c FWlib/ stm32f10x_sdio.c FWlib/ stm32f10x_dma.c FWlib/ misc.c
用户编写的文件	USER/main.c USER/stm32f10x_it.c USER/usart1.c USER/ sdio_sdcard.c

野火 STM32 开发板 MicroSD 卡硬件原理图：



1.2 SDIO 简介

野火 STM32 开发板的 CPU (STM32F103VET6)具有一个 SDIO 接口。SD/SDIO/MMC 主机接口可以支持 MMC 卡系统规范 4.2 版中的 3 个不同的数据总线模式：1 位(默认)、4 位和 8 位。在 8 位模式下，该接口可以使数据传输速率达到 48MHz，该接口兼容 SD 存储卡规范 2.0 版。SDIO 存储卡规范 2.0 版支持两种数据总线模式：1 位(默认)和 4 位。

目前的芯片版本只能一次支持一个 SD/SDIO/MMC 4.2 版的卡，但可以同时支持多个 MMC 4.1 版或之前版本的卡。除了 SD/SDIO/MMC，这个接口完全与 CE-ATA 数字协议版本 1.1 兼容。

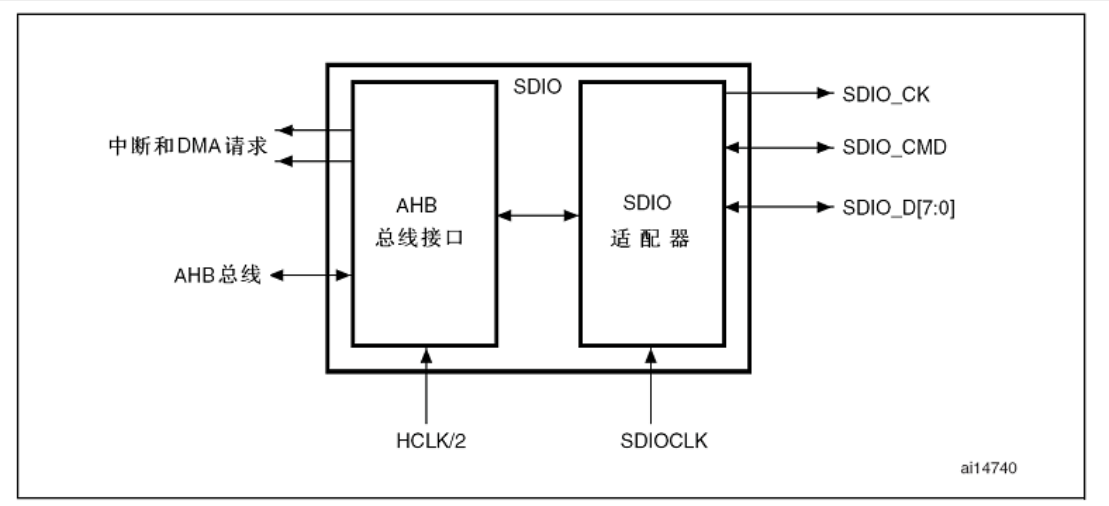
1.3 SD 协议

大多数人原来没有了解过 SD 协议，又看到 SDIO 的驱动有 2000 多行，感觉无从下手。所以野火重新写了这个文档进行详细的解释，帮助大家更快地跨过这道槛。

附资料：《Simplified_Physical_Layer_Spec.pdf》，这个资料包含了 SDIO 协议中 SD 存储卡的部分。

下面野火结合 STM32 的 SDIO，分析 SD 协议，让大家对它先有个大概了解，更具体的说明在代码中展开。

SDIO 接口图



一．从 SDIO 的时钟说起。

SDIO_CK 时钟是通过 PC12 引脚连接到 SD 卡的，是 SDIO 接口与 SD 卡用于同步的时钟。

SDIO 选配器挂载到 AHB 总线上，通过 HCLK 二分频输入到适配器得到 SDIO_CK 的时钟，这时 SDIO_CK = HCLK/（2+CLKDIV）。其中 CLKDIV 是 SDIO_CLK(寄存器)中的 CLKDIV 位。

另外，SDIO_CK 也可以由 SDIOCLK 通过设置 bypass 模式直接得到，这时 SDIO_CK = SDIOCLK=HCLK。

通过下面的库函数来配置时钟：

```
1.  SDIO_Init(&SDIO_InitStructure);
```

对 SD 卡的操作一般是大吞吐量的数据传输，所以采用 DMA 来提高效率，SDIO 采用的是 DMA2 中的通道 4。在数据传输的时候 SDIO 可向 DMA 发出请求。

二．讲解 SDIO 的命令、数据传输方式。

SDIO 的所有命令及命令响应，都是通过 SDIO-CMD 引脚来传输的。

命令只能由 host 即 STM32 的 SDIO 控制器发出。SDIO 协议把命令分成了 11 种，包括基本命令，读写命令还有 ACMD 系列命令等。其中，在发送 ACMD 命令前，要先向卡发送编号为 CMD55 的命令。

参照下面的命令格式图，其中的 start bit，transmission bit ，crc7，endbit，都是由 STM32 中的 SDIO 硬件完成，我们在软件上配置的时候只需要设置 command index 和命令参数 argument。Command index 就是命令索引（编号），如 CMD0，CMD1...被编号成 0，1...。有的命令会包含参数，读命令的地址参数等，这个参数被存放在 argument 段。

SD 卡命令格式

Bit position	47	46	[45:40]	[39:8]	[7:1]	0
Width (bits)	1	1	6	32	7	1
Value	'0'	'1'	x	x	x	'1'
Description	start bit	transmission bit	command index	argument	CRC7	end bit

Table 4-16: Command Format

可以通过下面的函数来配置、发送命令：

```
1.  SDIO_SendCommand(&SDIO_CmdInitStructure);    //发送命令
```

SD 卡对 host 的各种命令的回复称为响应，除了 CMD0 命令外，SD 卡在接收到命令都会返回一个响应。对于不同的命令，会有不同的响应格式，共 7 种，分为长响应型（136bit）和短响应型（48bit）。以下图，响应 6（R6）为例：

SD 卡命令响应格式（R6）

Bit position	47	46	[45:40]	[39:8] Argument field		[7:1]	0
Width (bits)	1	1	6	16	16	7	1
Value	'0'	'0'	x	x	x	x	'1'
Description	start bit	transmission bit	command index ('000011')	New published RCA [31:16] of the card	[15:0] card status bits: 23,22,19,12:0 (see Table 4-35)	CRC7	end bit

Table 4-32: Response R6

SDIO 通过 CMD 接收到响应后，硬件去除头尾的信息，把 command index 保存到 SDIO_RESPCMD 寄存器，把 argument field 内容保存存储到 SDIO_RESPx 寄存器中。这两个值可以分别通过下面的库函数得到。

```
1.  SDIO_GetCommandResponse();  //卡返回接收到的命令
2.  SDIO_GetResponse(SDIO_RESP1); //卡返回的 argument field 内容
```

数据写入，读取。请看下面的写数据时序图，在软件上，我们要处理的只是读忙。另外，我们的实验中用的是 Micro SD 卡，有 4 条数据线，默认的时候 SDIO 采用 1 条数据线的传输方式，更改为 4 条数据线模式要通过向卡发送命令来更改。

SD 卡的多块写入时序图

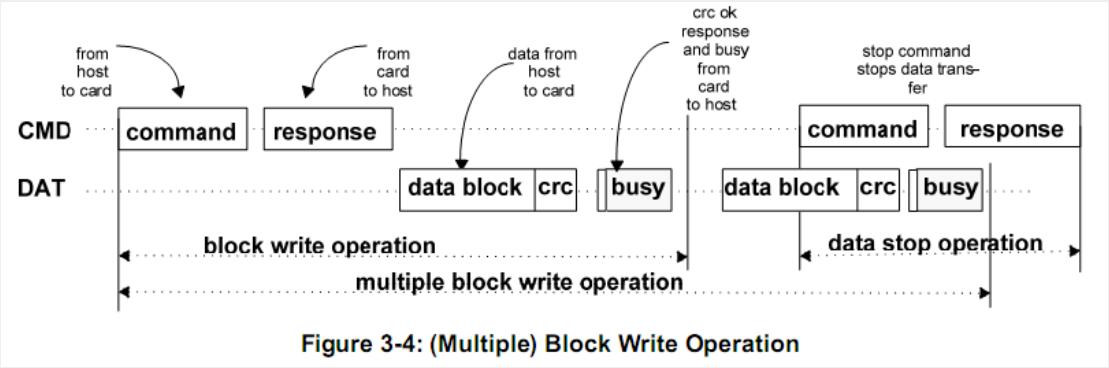


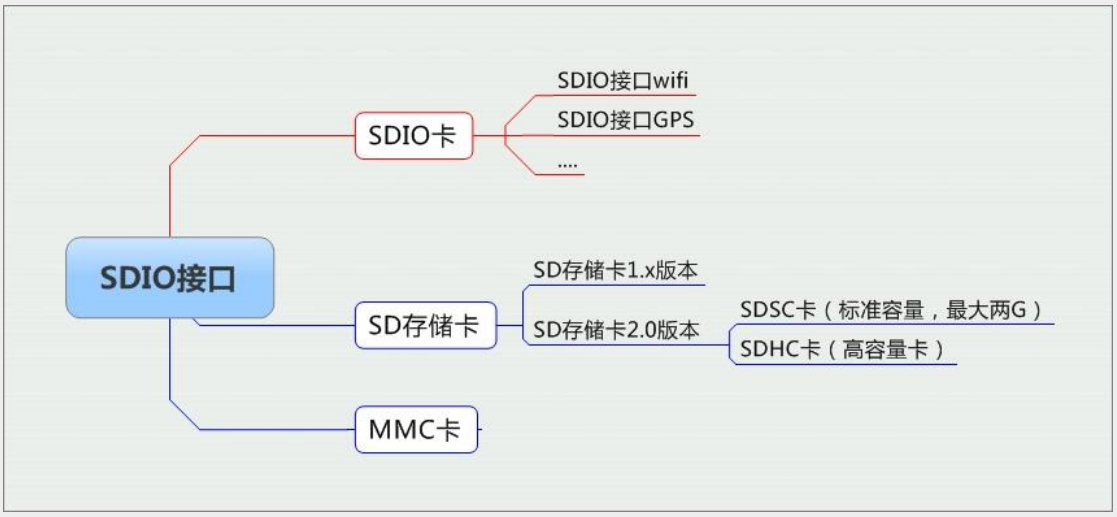
Figure 3-4: (Multiple) Block Write Operation

三．卡的种类。

STM32 的 SDIO 支持 SD 存储卡，SD I/O 卡 ，MMC 卡。

其中 SDI/O 卡与 SD 存储卡是有区别的，SDI/O 卡实际上就是利用 SDIO 接口的一些模块，插入 SD 的插槽中，扩展设备的功能，如：SDI/O wifi, SDI/O cmos 相机等。而 SD 存储卡就是我们平时常见的单纯用于存储数据的卡。

可使用 SDIO 接口类型的卡



本实验中使用的 Micro SD 卡属于 SDSC(标准容量，最大两 G)卡。介绍卡的种类是因为 SD 协议中的命令也支持这三种类型的卡，因此对 STM32 中的 SDIO 接口进行初始化后，上电后就要对接入的卡进行检测、分类，这个过程是通过向卡发送一系列不同的命令，根据卡不同的响应来进行分类。

下面进入代码展开具体讲解。

1.4 代码分析

首先要添加用的库文件，在工程文件夹下 Fwlib 下我们需添加以下库文件：

```
FWlib/stm32f10x_gpio.c

FWlib/stm32f10x_rcc.c

FWlib/stm32f10x_usart.c

FWlib/stm32f10x_sdio.c

FWlib/stm32f10x_dma.c

FWlib/misc.c
```

还要在 stm32f10x_conf.h 中把相应的头文件添加进来：

```
#include "stm32f10x_dma.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_sdio.h"
#include "stm32f10x_usart.h"
#include "misc.h"
```

保持良好的习惯，从 main 函数开始分析：

```
int main(void)
{

    /*进入到 main 函数前，启动文件 startup（startup_stm32f10x_xx.s）已经调用了在
         system_stm32f10x.c 中的 SystemInit()，配置好了系统时钟，在外部晶振 8M 的条件下，
         设置 HCLK = 72M  */

       /* Interrupt Config */
       NVIC_Configuration();

       /* USART1 config */
         USART1_Config();

    /*------------------------------ SD Init ---------------------------
--------- */
    Status = SD_Init();

    printf( "\r\n 这是一个 MicroSD 卡实验(没有跑文件系统).....\r\n " );


  if(Status == SD_OK) //检测初始化是否成功
    {
        printf( " \r\n SD_Init 初始化成功 \r\n " );
     }
  else
    {
        printf("\r\n SD_Init 初始化失败 \r\n" );
        printf("\r\n 返回的 Status 的值为:  %d \r\n",Status );
    }

    printf( " \r\n CardType is : %d ", SDCardInfo.CardType );
    printf( " \r\n CardCapacity is : %d ", SDCardInfo.CardCapacity );

    printf( " \r\n CardBlockSize is : %d ", SDCardInfo.CardBlockSize );
    printf( " \r\n RCA is : %d ", SDCardInfo.RCA);
    printf( " \r\n ManufacturerID is : %d \r\n", SDCardInfo.SD_cid.ManufacturerID );

    SD_EraseTest();     //擦除测试

    SD_SingleBlockTest();  //单块读写测试

    SD_MultiBlockTest();  //多块读写测试

  while (1)
  {}
}
```

main 函数的流程简单明了：

1. 用 NVIC_Configuration()初始化好 SDIO 的中断；
2. 用 USART1_Config()配置好用于返回调试信息的串口，SD_Init()开始进行 SDIO 的初始化；
3. 最后分别用 SD_EraseTest()、SD_SingleBlockTest()、SD_MultiBlockTest()进行擦除，单数据块读写，多数据块读写测试。

下面我们先进入 SDIO 驱动函数的大头——SD_Init()进行分析：

```
/*
 * 函数名：SD_Init
 * 描述  ：初始化 SD 卡，使卡处于就绪状态(准备传输数据)
 * 输入  ：无
 * 输出  ：-SD_Error SD 卡错误代码
 *         成功时则为 SD_OK
 * 调用  ：外部调用
 */
SD_Error SD_Init(void)
{
    /*重置 SD_Error 状态*/
  SD_Error errorstatus = SD_OK;

  /* SDIO 外设底层引脚初始化 */
  GPIO_Configuration();

  /*对 SDIO 的所有寄存器进行复位*/
  SDIO_DeInit();

  /*上电并进行卡识别流程，确认卡的操作电压  */
  errorstatus = SD_PowerON();

  /*如果上电，识别不成功，返回"响应超时"错误 */
  if (errorstatus != SD_OK)
  {
    /*!< CMD Response TimeOut (wait for CMDSENT flag) */
    return(errorstatus);
  }

  /*卡识别成功，进行卡初始化    */
  errorstatus = SD_InitializeCards();

  if (errorstatus != SD_OK)    //失败返回
  {
    /*!< CMD Response TimeOut (wait for CMDSENT flag) */
    return(errorstatus);
  }

  /*!< Configure the SDIO peripheral
  上电识别，卡初始化都完成后，进入数据传输模式，提高读写速度
  速度若超过 24M 要进入 bypass 模式
  !< on STM32F2xx devices, SDIOCLK is fixed to 48MHz
  !< SDIOCLK = HCLK, SDIO_CK = HCLK/(2 + SDIO_TRANSFER_CLK_DIV) */

  SDIO_InitStructure.SDIO_ClockDiv = SDIO_TRANSFER_CLK_DIV;
  SDIO_InitStructure.SDIO_ClockEdge = SDIO_ClockEdge_Rising;          //上
升沿采集数据
```

```
46.   SDIO_InitStructure.SDIO_ClockBypass = SDIO_ClockBypass_Disable;  //时钟
    频率若超过 24M,要开启此模式
47.   SDIO_InitStructure.SDIO_ClockPowerSave = SDIO_ClockPowerSave_Disable;
      //若开启此功能，在总线空闲时关闭 sd_clk 时钟
48.   SDIO_InitStructure.SDIO_BusWide = SDIO_BusWide_1b;
       //1 位模式
49.   SDIO_InitStructure.SDIO_HardwareFlowControl = SDIO_HardwareFlowControl_
    Disable; //硬件流，若开启，在 FIFO 不能进行发送和接收数据时，数据传输暂停
50.   SDIO_Init(&SDIO_InitStructure);
51.
52.   if (errorstatus == SD_OK)
53.   {
54.     /*----------------- Read CSD/CID MSD registers ------------------
    */
55.     errorstatus = SD_GetCardInfo(&SDCardInfo);    //用来读取 csd/cid 寄存
    器
56.   }
57.
58.   if (errorstatus == SD_OK)
59.   {
60.     /*----------------- Select Card --------------------------------
    */
61.     errorstatus = SD_SelectDeselect((uint32_t) (SDCardInfo.RCA << 16));
     //通过 cmd7   ,rca 选择要操作的卡
62.   }
63.
64.   if (errorstatus == SD_OK)
65.   {
66.     errorstatus = SD_EnableWideBusOperation(SDIO_BusWide_4b);    //开启
    4bits 模式
67.   }
68.
69.   return(errorstatus);
70. }
```

先从整体上了解这个 SD_Init()函数：

1.用 GPIO_Configuration()进行 SDIO 的端口底层配置

2.分别调用了 SD_PowerON()和 SD_InitializeCards()函数，这两个函数共同实现了上面提到的卡检测、识别流程。

3.调用 SDIO_Init(&SDIO_InitStructure)库函数配置 SDIO 的时钟，数据线宽度，硬件流（在读写数据的时候，开启硬件流是和很必要的，可以减少出错）

4. 调用 SD_GetCardInfo(&SDCardInfo)获取 sd 卡的 CSD 寄存器中的内容，在 main 函数里输出到串口的数据就是这个时候从卡读取得到的。

5. 调用 SD_SelectDeselect()选定后面即将要操作的卡。

6.调用 SD_EnableWideBusOperation(SDIO_BusWide_4b)开启 4bit 数据线模式

如果 SD_Init()函数能够执行完整个流程，并且返回值是 SD_OK 的话则说明初始化成功，就可以开始进行擦除、读写的操作了。

下面进入 SD_PowerON()函数，分析完这个函数大家就能了解 SDIO 如何接收、发送命令了。

```
1.  /*
2.   * 函数名：SD_PowerON
3.   * 描述  ：确保 SD 卡的工作电压和配置控制时钟
4.   * 输入  ：无
5.   * 输出  ：-SD_Error SD 卡错误代码
6.   *          成功时则为 SD_OK
7.   * 调用  ：在 SD_Init() 调用
8.   */
9.  SD_Error SD_PowerON(void)
10. {
11.   SD_Error errorstatus = SD_OK;
12.   uint32_t response = 0, count = 0, validvoltage = 0;
13.   uint32_t SDType = SD_STD_CAPACITY;
14.
15.   /*!< Power ON Sequence -----------------------------------------------
    -----*/
16.   /*!< Configure the SDIO peripheral */
17.   /*!< SDIOCLK = HCLK, SDIO_CK = HCLK/(2 + SDIO_INIT_CLK_DIV) */
18.   /*!< on STM32F2xx devices, SDIOCLK is fixed to 48MHz */
19.   /*!< SDIO_CK for initialization should not exceed 400 KHz */
20.   /*初始化时的时钟不能大于 400KHz*/
21.   SDIO_InitStructure.SDIO_ClockDiv = SDIO_INIT_CLK_DIV; /* HCLK = 72MHz,
    SDIOCLK = 72MHz, SDIO_CK = HCLK/(178 + 2) = 400 KHz */
22.   SDIO_InitStructure.SDIO_ClockEdge = SDIO_ClockEdge_Rising;
23.   SDIO_InitStructure.SDIO_ClockBypass = SDIO_ClockBypass_Disable;  //不使
    用 bypass 模式，直接用 HCLK 进行分频得到 SDIO_CK
24.   SDIO_InitStructure.SDIO_ClockPowerSave = SDIO_ClockPowerSave_Disable; /
    / 空闲时不关闭时钟电源
25.   SDIO_InitStructure.SDIO_BusWide = SDIO_BusWide_1b;                    /
    /1 位数据线
26.   SDIO_InitStructure.SDIO_HardwareFlowControl = SDIO_HardwareFlowControl_
    Disable;//硬件流
27.   SDIO_Init(&SDIO_InitStructure);
28.
29.   /*!< Set Power State to ON */
30.   SDIO_SetPowerState(SDIO_PowerState_ON);
31.
32.   /*!< Enable SDIO Clock */
33.   SDIO_ClockCmd(ENABLE);
34.
35.    /*下面发送一系列命令,开始卡识别流程*/
36.   /*!< CMD0: GO_IDLE_STATE ---------------------------------------------
    -----*/
37.   /*!< No CMD response required */
38.   SDIO_CmdInitStructure.SDIO_Argument = 0x0;
39.   SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_GO_IDLE_STATE; //cmd0
40.   SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_No;  //无响应
41.   SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
42.   SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;  //则 CPSM 在开始发送
    命令之前等待数据传输结束。
43.   SDIO_SendCommand(&SDIO_CmdInitStructure);          //写命令进命令寄存器
44.
45.   errorstatus = CmdError();//检测是否正确接收到 cmd0
46.
47.   if (errorstatus != SD_OK) //命令发送出错，返回
48.   {
49.     /*!< CMD Response TimeOut (wait for CMDSENT flag) */
50.     return(errorstatus);
51.   }
52.
53.   /*!< CMD8: SEND_IF_COND ----------------------------------------------
    -----*/
54.   /*!< Send CMD8 to verify SD card interface operating condition */
55.   /*!< Argument: - [31:12]: Reserved (shall be set to '0')
56.               - [11:8]: Supply Voltage (VHS) 0x1 (Range: 2.7-3.6 V)
```

```
57.                 - [7:0]: Check Pattern (recommended 0xAA) */
58.   /*!< CMD Response: R7 */
59.   SDIO_CmdInitStructure.SDIO_Argument = SD_CHECK_PATTERN;    //接收到命令 sd
    会返回这个参数
60.   SDIO_CmdInitStructure.SDIO_CmdIndex = SDIO_SEND_IF_COND;  //cmd8
61.   SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;      //r7
62.   SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;             //关闭等待中
    断
63.   SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
64.   SDIO_SendCommand(&SDIO_CmdInitStructure);
65.
66.   /*检查是否接收到命令*/
67.   errorstatus = CmdResp7Error();
68.
69.   if (errorstatus == SD_OK)      //有响应则 card 遵循 sd 协议 2.0 版本
70.   {
71.     CardType = SDIO_STD_CAPACITY_SD_CARD_V2_0; /*!< SD Card 2.0 ，先把它定
    义会 sdsc 类型的卡*/
72.     SDType = SD_HIGH_CAPACITY;  //这个变量用作 acmd41 的参数，用来询问是 sdsc 卡
    还是 sdhc 卡
73.   }
74.   else  //无响应，说明是 1.x 的或 mmc 的卡
75.   {
76.     /*!< CMD55 */
77.     SDIO_CmdInitStructure.SDIO_Argument = 0x00;
78.     SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_APP_CMD;
79.     SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
80.     SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
81.     SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
82.     SDIO_SendCommand(&SDIO_CmdInitStructure);
83.     errorstatus = CmdResp1Error(SD_CMD_APP_CMD);
84.   }
85.   /*!< CMD55 */      //为什么在 else 里和 else 外面都要发送 CMD55?
86.   //发送 cmd55，用于检测是 sd 卡还是 mmc 卡，或是不支持的卡
87.   SDIO_CmdInitStructure.SDIO_Argument = 0x00;
88.   SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_APP_CMD;
89.   SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short; //r1
90.   SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
91.   SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
92.   SDIO_SendCommand(&SDIO_CmdInitStructure);
93.   errorstatus = CmdResp1Error(SD_CMD_APP_CMD);  //是否响应，没响应的是 mmc 或
    不支持的卡
94.
95.   /*!< If errorstatus is Command TimeOut, it is a MMC card */
96.   /*!< If errorstatus is SD_OK it is a SD card: SD card 2.0 (voltage rang
    e mismatch)
97.      or SD card 1.x */
98.   if (errorstatus == SD_OK) //响应了 cmd55，是 sd 卡，可能为 1.x,可能为 2.0
99.   {
100.       /*下面开始循环地发送 sdio 支持的电压范围，循环一定次数*/
101.
102.       /*!< SD CARD */
103.       /*!< Send ACMD41 SD_APP_OP_COND with Argument 0x80100000 */
104.       while ((!validvoltage) && (count < SD_MAX_VOLT_TRIAL))
105.       {
106.       //因为下面要用到 ACMD41，是 ACMD 命令，在发送 ACMD 命令前都要先向卡发送
    CMD55
107.         /*!< SEND CMD55 APP_CMD with RCA as 0 */
108.         SDIO_CmdInitStructure.SDIO_Argument = 0x00;
109.         SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_APP_CMD;    //CMD55

110.         SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
111.         SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
112.         SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
113.         SDIO_SendCommand(&SDIO_CmdInitStructure);
```

```
114.
115.          errorstatus = CmdResp1Error(SD_CMD_APP_CMD); //检测响应
116.
117.          if (errorstatus != SD_OK)
118.          {
119.            return(errorstatus);//没响应 CMD55，返回
120.          }
121.          //acmd41，命令参数由支持的电压范围及 HCS 位组成，HCS 位置一来区分卡是 SDSc
    还是 sdhc
122.          SDIO_CmdInitStructure.SDIO_Argument = SD_VOLTAGE_WINDOW_SD | SDT
    ype;     //参数为主机可供电压范围及 hcs 位
123.          SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SD_APP_OP_COND;
124.          SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;  //r3

125.          SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
126.          SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
127.          SDIO_SendCommand(&SDIO_CmdInitStructure);
128.
129.          errorstatus = CmdResp3Error();     //检测是否正确接收到数据
130.          if (errorstatus != SD_OK)
131.          {
132.            return(errorstatus);  //没正确接收到 acmd41，出错，返回
133.          }
134.           /*若卡需求电压在 SDIO 的供电电压范围内，会自动上电并标志 pwr_up 位*/
135.          response = SDIO_GetResponse(SDIO_RESP1);    //读取卡寄存器，卡状
    态
136.          validvoltage = (((response >> 31) == 1) ? 1 : 0); //读取卡的 ocr
    寄存器的 pwr_up 位，看是否已工作在正常电压
137.          count++;              //计算循环次数
138.        }
139.        if (count >= SD_MAX_VOLT_TRIAL) //循环检测超过一定次数还没上电
140.        {
141.          errorstatus = SD_INVALID_VOLTRANGE;       //SDIO 不支持 card 的供电电
    压
142.          return(errorstatus);
143.        }
144.                  /*检查卡返回信息中的 HCS 位*/
145.        if (response &= SD_HIGH_CAPACITY)   //判断 ocr 中的 ccs 位 ，如果是 sdsc
    卡则不执行下面的语句
146.        {
147.          CardType = SDIO_HIGH_CAPACITY_SD_CARD;  //把卡类型从初始化的 sdsc 型
    改为 sdhc 型
148.        }
149.
150.      }/*!< else MMC Card */
151.
152.      return(errorstatus);
153.    }
```

这个函数的流程就是卡的上电、识别操作，如下图：

卡的上电，识别流程：

截图来自《Simplified_Physical_Layer_Spec.pdf》 page27

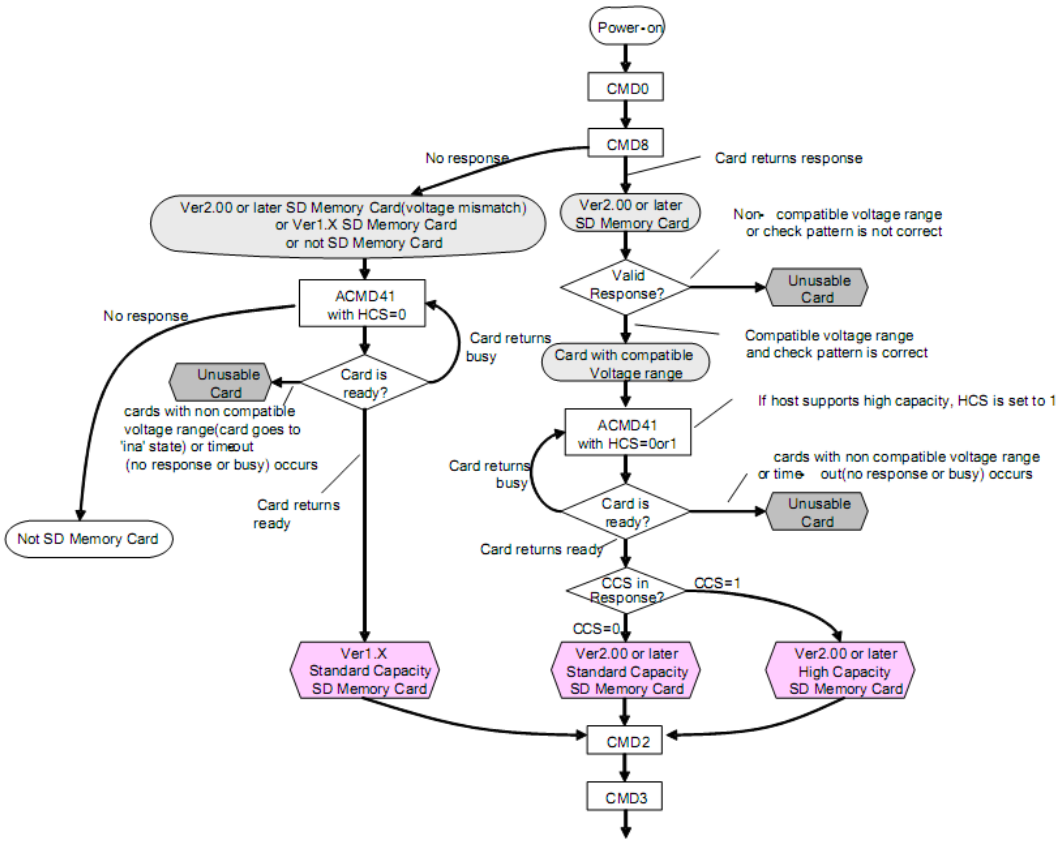


Figure 4-2: Card Initialization and Identification Flow (SD mode)

代码中所有的判断语句都是根据这个图的各个识别走向展开的，最终把卡分为 1.0 版的 SD 存储卡，2.0 版的 SDSC 卡和 2.0 版的 SDHC 卡。

在这个代码流程中有两点要注意一下：

1.初始化的时钟。SDIO_CK 的时钟分为两个阶段，在初始化阶段 SDIO_CK 的频率要小于 400KHz，初始化完成后可把 SDIO_CK 调整成高速模式，高速模式时超过 24M 要开启 bypass 模式，对于 SD 存储卡即使开启 bypass，最高频率不能超过 25MHz。

2.CMD8 命令。

CMD8 命令格式。

Table 4-15 shows the format of CMD8.

Bit position	47	46	[45:40]	[39:20]	[19:16]	[15:8]	[7:1]	0
Width (bits)	1	1	6	20	4	8	7	1
Value	'0'	'1'	'001000'	'00000h'	x	x	x	'1'
Description	start bit	transmission bit	command index	reserved bits	voltage supplied (VHS)	check pattern	CRC7	end bit

CMD8 命令中的 VHS 是用来确认主机 SDIO 是否支持卡的工作电压的。Check pattern 部分可以是任何数值，若 SDIO 支持卡的工作电压，卡会把接收到的 check pattern 数值原样返回给主机。

CMD8 命令的响应格式 R7：

Bit position	47	46	[45:40]	[39:20]	[19:16]	[15:8]	[7:1]	0
Width (bits)	1	1	6	20	4	8	7	1
Value	'0'	'0'	'001000'	'00000h'	x	x	x	'1'
Description	start bit	transmission bit	command index	reserved bits	voltage accepted	echo-back of check pattern	CRC7	end bit

Table 4-33: Response R7

在驱动程序中调用了 CmdResp7Error()来检验卡接收命令后的响应。

3.ACMD41 命令。

这个命令也是用来进一步检查 SDIO 是否支持卡的工作电压的，协议要它在调用它之前必须先调用 CMD8，另外还可以通过它命令参数中的 HCS 位来区分卡是 SDHC 卡还是 SDSC 卡。

确认工作电压时循环地发送 ACMD41，发送后检查在 SD 卡上的 OCR 寄存器中的 pwr_up 位，若 pwr_up 位置为 1，表明 SDIO 支持卡的工作电压，卡开始正常工作。

同时把 ACMD41 中的命令参数 HCS 位置 1，卡正常工作的时候检测 OCR 寄存器中的 CCS 位，若 CCS 位为 1 则说明该卡为 SDHC 卡，为零则为 SDSC 卡。

因为 ACMD41 命令属于 ACMD 命令，在发送 ACMD 命令前都要先发送 CMD55.

ACMD41 命令格式

ACMD INDEX	type	argument	resp	abbreviation	command description
ACMD41	bcr	[31]reserved bit [30]HCS(OCR[30]) [29:24]reserved bits [23:0] V_{DD} Voltage Window(OCR[23:0])	R3	SD_SEND_OP_COND	Sends host capacity support information (HCS) and asks the accessed card to send its operating condition register (OCR) content in the response on the CMD line. HCS is effective when card receives SEND_IF_COND command. Reserved bit shall be set to '0'. CCS bit is assigned to OCR[30].

ACMD41 命令的响应（R3），返回的是 OCR 寄存器的值

4.9.4 R3 (OCR register)

Code length is 48 bits. The contents of the OCR register are sent as a response to ACMD41.

Bit position	47	46	[45:40]	[39:8]	[7:1]	0
Width (bits)	1	1	6	32	7	1
Value	'0'	'0'	'111111'	x	'1111111'	'1'
Description	start bit	transmission bit	reserved	OCR register	reserved	end bit

Table 4-31: Response R3

OCR 寄存器的内容

OCR bit position	OCR Fields Definition
0-3	reserved
4	reserved
5	reserved
6	reserved
7	Reserved for Low Voltage Range
8	reserved
9	reserved
10	reserved
11	reserved
12	reserved
13	reserved
14	reserved
15	2.7-2.8
16	2.8-2.9
17	2.9-3.0
18	3.0-3.1
19	3.1-3.2
20	3.2-3.3
21	3.3-3.4
22	3.4-3.5
23	3.5-3.6
24-29	reserved
30	Card Capacity Status (CCS)[1]
31	Card power up status bit (busy)[2]

VDD Voltage Window

1) This bit is valid only when the card power up status bit is set.
2) This bit is set to LOW if the card has not finished the power up routine.

SD 卡上电确认成功后，进入 SD_InitializeCards()函数：

```
/*
 * 函数名：SD_InitializeCards
 * 描述  ：初始化所有的卡或者单个卡进入就绪状态
 * 输入  ：无
 * 输出  ：-SD_Error SD 卡错误代码
 *         成功时则为 SD_OK
 * 调用  ：在 SD_Init() 调用，在调用 power_on（）上电卡识别完毕后，调用此函数进行卡初始化
 */
SD_Error SD_InitializeCards(void)
{
  SD_Error errorstatus = SD_OK;
  uint16_t rca = 0x01;

  if (SDIO_GetPowerState() == SDIO_PowerState_OFF)
  {
    errorstatus = SD_REQUEST_NOT_APPLICABLE;
    return(errorstatus);
  }

  if (SDIO_SECURE_DIGITAL_IO_CARD != CardType)//判断卡的类型
  {
    /*!< Send CMD2 ALL_SEND_CID */
    SDIO_CmdInitStructure.SDIO_Argument = 0x0;
    SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_ALL_SEND_CID;
    SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Long;
    SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
    SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
    SDIO_SendCommand(&SDIO_CmdInitStructure);

    errorstatus = CmdResp2Error();

    if (SD_OK != errorstatus)
    {
      return(errorstatus);
    }

    CID_Tab[0] = SDIO_GetResponse(SDIO_RESP1);
    CID_Tab[1] = SDIO_GetResponse(SDIO_RESP2);
    CID_Tab[2] = SDIO_GetResponse(SDIO_RESP3);
    CID_Tab[3] = SDIO_GetResponse(SDIO_RESP4);
  }

     /*下面开始 SD 卡初始化流程*/
  if ((SDIO_STD_CAPACITY_SD_CARD_V1_1 == CardType) ||  (SDIO_STD_CAPACITY_SD_CARD_V2_0 == CardType) ||  (SDIO_SECURE_DIGITAL_IO_COMBO_CARD == CardType)
      ||  (SDIO_HIGH_CAPACITY_SD_CARD == CardType))  //使用的是 2.0 的卡
  {
    /*!< Send CMD3 SET_REL_ADDR with argument 0 */
    /*!< SD Card publishes its RCA. */
    SDIO_CmdInitStructure.SDIO_Argument = 0x00;
    SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_REL_ADDR;  //cmd3

    SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short; //r6
    SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
    SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
    SDIO_SendCommand(&SDIO_CmdInitStructure);

    errorstatus = CmdResp6Error(SD_CMD_SET_REL_ADDR, &rca); //把接收到的卡相对地址存起来。

    if (SD_OK != errorstatus)
    {
      return(errorstatus);
```

```
61.     }
62.   }
63.
64.   if (SDIO_SECURE_DIGITAL_IO_CARD != CardType)
65.   {
66.     RCA = rca;
67.
68.     /*!< Send CMD9 SEND_CSD with argument as card's RCA */
69.     SDIO_CmdInitStructure.SDIO_Argument = (uint32_t)(rca << 16);
70.     SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SEND_CSD;
71.     SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Long;
72.     SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
73.     SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
74.     SDIO_SendCommand(&SDIO_CmdInitStructure);
75.
76.     errorstatus = CmdResp2Error();
77.
78.     if (SD_OK != errorstatus)
79.     {
80.       return(errorstatus);
81.     }
82.
83.     CSD_Tab[0] = SDIO_GetResponse(SDIO_RESP1);
84.     CSD_Tab[1] = SDIO_GetResponse(SDIO_RESP2);
85.     CSD_Tab[2] = SDIO_GetResponse(SDIO_RESP3);
86.     CSD_Tab[3] = SDIO_GetResponse(SDIO_RESP4);
87.   }
88.
89.   errorstatus = SD_OK; /*!< All cards get intialized */
90.
91.   return(errorstatus);
92. }
```

这个函数向卡发送了 CMD2 和 CMD3 命令

1.CMD2

CMD2 命令是要求卡返回它的 CID 寄存器的内容。

命令的响应格式（R2）。

Bit position	135	134	[133:128]	[127:1]	0
Width (bits)	1	1	6	127	1
Value	'0'	'0'	'111111'	x	'1'
Description	start bit	transmission bit	reserved	CID or CSD register incl. internal CRC7	end bit

Table 4-30: Response R2

因为命令格式是 136 位的，属于长响应。软件接收的信息有 128 位。在长响应的时候通过 SDIO_GetResponse(SDIO_RESP4);中的不同参数来获取 CID 中的不同数

据段的数据。

表151 响应类型和SDIO_RESPx寄存器

寄存器	短响应	长响应
SDIO_RESP1	卡状态[31:0]	卡状态[127:96]
SDIO_RESP2	不用	卡状态[95:64]
SDIO_RESP3	不用	卡状态[63:32]
SDIO_RESP4	不用	卡状态[31:1]

总是先收到卡状态的最高位，SDIO_RESP3寄存器的最低位始终为0。

2.CMD3

CMD3 命令是要求卡向主机发送卡的相对地址。在接有多个卡的时候，主机要求接口上的卡重新发一个相对地址，这个地址跟卡的实际 ID 不一样。比如接口上接了 5 个卡，这 5 个卡的相对地址就分别为 1，2，3，4，5.以后主机 SDIO 对这几个卡寻址就直接使用相对地址。这个地址的作用就是为了寻址更加简单。

接下来我们回到 SD_Init()函数。分析到这里大家应该对 SDIO 的命令发送和响应比较清楚了。在 SD_InitializeCards()之后的 SD_GetCardInfo(&SDCardInfo)、SD_SelectDeselect()和 SD_EnableWideBusOperation(SDIO_BusWide_4b)的具体实现就不再详细分析了，实际就是发送相应的命令，对卡进行相应的操作。

接下来分析 main 函数中的 SD_MultiBlockTest()多块数据读写函数，让大家了解 SDIO 是怎样传输数据的。

```
1.  /*
2.   * 函数名: SD_MultiBlockTest
3.   * 描述  :      多数据块读写测试
4.   * 输入  : 无
5.   * 输出  : 无
6.   */
7.  void SD_MultiBlockTest(void)
8.  {
9.    /*--------------- Multiple Block Read/Write --------------------
    */
10.   /* Fill the buffer to send */
11.   Fill_Buffer(Buffer_MultiBlock_Tx, MULTI_BUFFER_SIZE, 0x0);
12.
13.   if (Status == SD_OK)
```

```
  {
    /* Write multiple block of many bytes on address 0 */
    Status = SD_WriteMultiBlocks(Buffer_MultiBlock_Tx, 0x00, BLOCK_SIZE, NUMBER_OF_BLOCKS);
    /* Check if the Transfer is finished */
    Status = SD_WaitWriteOperation();
    while(SD_GetStatus() != SD_TRANSFER_OK);
  }

  if (Status == SD_OK)
  {
    /* Read block of many bytes from address 0 */
    Status = SD_ReadMultiBlocks(Buffer_MultiBlock_Rx, 0x00, BLOCK_SIZE, NUMBER_OF_BLOCKS);
    /* Check if the Transfer is finished */
    Status = SD_WaitReadOperation();
    while(SD_GetStatus() != SD_TRANSFER_OK);
  }

  /* Check the correctness of written data */
  if (Status == SD_OK)
  {
    TransferStatus2 = Buffercmp(Buffer_MultiBlock_Tx, Buffer_MultiBlock_Rx, MULTI_BUFFER_SIZE);
  }

  if(TransferStatus2 == PASSED)
    printf("\r\n 多块读写测试成功！ " );

  else
    printf("\r\n 多块读写测试失败！ " );

}
```

把这个函数拿出来分析最重要的一点就是让大家注意在调用了 SD_WriteMultiBlocks()这一类读写操作的函数后，一定要调用 SD_WaitWriteOperation()[在读数据时调用 SD_WaitReadOperation]和 SD_GetStatus()来确保数据传输已经结束再进行其它操作。其中的 SD_WaitWriteOperation()是用来等待 DMA 把缓冲的数据传输到 SDIO 的 FIFO 的；而 SD_GetStatus()是用来等待卡与 SDIO 之间传输数据完毕的。

最后进入 SD_WriteMultiBlocks（）函数分析：

```
/*
 * 函数名：SD_WriteMultiBlocks
 * 描述  ：从输入的起始地址开始，向卡写入多个数据块，
 *         只能在 DMA 模式下使用这个函数
 *     注意：调用这个函数后一定要调用
 *           SD_WaitWriteOperation（）来等待 DMA 传输结束
 *           和   SD_GetStatus() 检测卡与 SDIO 的 FIFO 间是否已经完成传输
 * 输入  ：
 *         * @param  WriteAddr: Address from where data are to be read.
 *         * @param  writebuff: pointer to the buffer that contain the data to be transferred.
 *         * @param  BlockSize: the SD card Data block size. The Block size should be 512.
```

```
12.           * @param  NumberOfBlocks: number of blocks to be written.
13.  * 输出    : SD 错误类型
14.  */
15. SD_Error SD_WriteMultiBlocks(uint8_t *writebuff, uint32_t WriteAddr, u
    int16_t BlockSize, uint32_t NumberOfBlocks)
16. {
17.   SD_Error errorstatus = SD_OK;
18.   __IO uint32_t count = 0;
19.
20.   TransferError = SD_OK;
21.   TransferEnd = 0;
22.   StopCondition = 1;
23.
24.   SDIO->DCTRL = 0x0;
25.
26.   if (CardType == SDIO_HIGH_CAPACITY_SD_CARD)
27.   {
28.     BlockSize = 512;
29.     WriteAddr /= 512;
30.   }
31.
32.     /*******************add，没有这一段容易卡死在 DMA 检测中
    ************************************/
33.     /*!< Set Block Size for Card，cmd16,若是 sdsc 卡，可以用来设置块大小，若
    是 sdhc 卡，块大小为 512 字节，不受 cmd16 影响 */
34.   SDIO_CmdInitStructure.SDIO_Argument = (uint32_t) BlockSize;
35.   SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_BLOCKLEN;
36.   SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;    //r1
37.   SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
38.   SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
39.   SDIO_SendCommand(&SDIO_CmdInitStructure);
40.
41.   errorstatus = CmdResp1Error(SD_CMD_SET_BLOCKLEN);
42.
43.   if (SD_OK != errorstatus)
44.   {
45.     return(errorstatus);
46.   }
47.  /*************************************************************************
    *************/
48.
49.   /*!< To improve performance  */
50.   SDIO_CmdInitStructure.SDIO_Argument = (uint32_t) (RCA << 16);
51.   SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_APP_CMD; // cmd55
52.   SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
53.   SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
54.   SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
55.   SDIO_SendCommand(&SDIO_CmdInitStructure);
56.
57.
58.   errorstatus = CmdResp1Error(SD_CMD_APP_CMD);
59.
60.   if (errorstatus != SD_OK)
61.   {
62.     return(errorstatus);
63.   }
64.   /*!< To improve performance *///  pre-erased，在多块写入时可发送此命令进
    行预擦除
65.   SDIO_CmdInitStructure.SDIO_Argument = (uint32_t)NumberOfBlocks;  //
    参数为将要写入的块数目
66.   SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_BLOCK_COUNT;  //cmd
    23
67.   SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
68.   SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
69.   SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
70.   SDIO_SendCommand(&SDIO_CmdInitStructure);
71.
```

```
72.   errorstatus = CmdResp1Error(SD_CMD_SET_BLOCK_COUNT);
73.
74.   if (errorstatus != SD_OK)
75.   {
76.     return(errorstatus);
77.   }
78.
79.
80.   /*!< Send CMD25 WRITE_MULT_BLOCK with argument data address */
81.   SDIO_CmdInitStructure.SDIO_Argument = (uint32_t)WriteAddr;
82.   SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_WRITE_MULT_BLOCK;
83.   SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
84.   SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
85.   SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
86.   SDIO_SendCommand(&SDIO_CmdInitStructure);
87.
88.   errorstatus = CmdResp1Error(SD_CMD_WRITE_MULT_BLOCK);
89.
90.   if (SD_OK != errorstatus)
91.   {
92.     return(errorstatus);
93.   }
94.
95.   SDIO_DataInitStructure.SDIO_DataTimeOut = SD_DATATIMEOUT;
96.   SDIO_DataInitStructure.SDIO_DataLength = NumberOfBlocks * BlockSize;

97.   SDIO_DataInitStructure.SDIO_DataBlockSize = (uint32_t) 9 << 4;
98.   SDIO_DataInitStructure.SDIO_TransferDir = SDIO_TransferDir_ToCard;

99.   SDIO_DataInitStructure.SDIO_TransferMode = SDIO_TransferMode_Block;

100.        SDIO_DataInitStructure.SDIO_DPSM = SDIO_DPSM_Enable;
101.        SDIO_DataConfig(&SDIO_DataInitStructure);
102.
103.        SDIO_ITConfig(SDIO_IT_DATAEND, ENABLE);
104.        SDIO_DMACmd(ENABLE);
105.        SD_DMA_TxConfig((uint32_t *)writebuff, (NumberOfBlocks * Bloc
    kSize));
106.
107.        return(errorstatus);
108.      }
```

写操作在发送正式的多块写入命令 CMD25 前调用了 CMD23 进行预写，这样有利于提高写入的速度。在代码的最后调用了 SDIO_ITConfig()，SDIO 的数据传输结束中断就是这个时候开启的，数据传输结束时，就进入到 stm32f10x_it.c 文件中的中断服务函数 SDIO_IRQHandler()中处理了，中断服务函数主要就是负责清中断。

最后讲一下官方原版的驱动中的一个 bug。

在官方原版的 SDIO 驱动的 SD_ReadBlock()、SD_ReadMultiBlocks()、SD_WriteBlock()和 SD_WriteMultiBlocks()这几个函数中，发送读写命令前，漏掉了发送一个 CMD16 命令，这个命令用于设置读写 SD 卡的块大小。缺少这个命令很容易导致程序运行时卡死在循环检测 DMA 传输结束的代码中，网上很多

人直接移植 ST 官方例程时，用 3.5 版库函数和这个 4.5 版的 SDIO 驱动移植失败，就是缺少了这段用 CMD16 设置块大小的代码。

到这里，终于讲解完毕啦！这个讲解如果能让你从对 SDIO 一无所知到大概了解的话，我的目标就达到啦，想要更深入了解还是要好好地配合这个例程中我在代码中的注释和附带资料 SD2.0 协议《Simplified_Physical_Layer_Spec.pdf》好好研究一番! ^_^

注意：这个例程是没有跑文件系统的，而是直接就去读卡的 block，这样的话就会破坏卡的分区，在实验完成之后，你再把卡插到电脑上时，电脑会提示你要重新初始化卡，这是正常想象，并不是本实验把你的卡弄坏了，如果卡原来有资料的请先把数据备份了再进行测试。但跑文件系统时就不会出现这种问题，有关文件系统的操作将在下一讲的教程中讲解。

1.5 实验现象

将野火 STM32 开发板供电(DC5V)，插上 JLINK，插上串口线(两头都是母的交叉线)，插上 MicroSD 卡(我用的是 1G ，经测试，本驱动也适用于 2G 以上的卡（sdhc 卡）)，打开超级终端，配置超级终端为 115200 8-N-1，将编译好的程序下载到开发板，即可看到超级终端打印出如下信息：