AN1302 STM32 SDIO 的使用

本应用文档(AN1302,对应 ALIENTEK 战舰 STM32 扩展实验 2)将教大家如何使用 STM32 的 SDIO 接口来驱动 SD 卡，并结合 FATFS，实现对 SD 卡的读写操作。

本文档分为如下几部分：

1， SDIO 简介
2， 硬件连接
3， 软件实现
4， 验证

1、SDIO 接口简介

STM32F103 大容量系列(FLASH≥256K)产品都含有 SDIO 控制器，ALIENTEK 战舰 STM32 开发板所使用 STM32F103ZET6 就属于大容量产品，带有 SDIO 接口。本节，我们将简单介绍 STM32 的 SDIO 接口，包括：主要功能及框图、时钟、命令与响应和相关寄存器简介等，最后，我们将介绍 SD 卡的初始化流程。

1.1 SDIO 主要功能及框图

STM32 的 SDIO 控制器支持多媒体卡(MMC 卡)、SD 存储卡、SD I/O 卡和 CE-ATA 设备等。SDIO 的主要功能如下：

- 与多媒体卡系统规格书版本 4.2 全兼容。支持三种不同的数据总线模式：1 位(默认)、4 位和 8 位。
- 与较早的多媒体卡系统规格版本全兼容(向前兼容)。
- 与 SD 存储卡规格版本 2.0 全兼容。
- 与 SD I/O 卡规格版本 2.0 全兼容：支持良种不同的数据总线模式：1 位(默认)和 4 位。
- 完全支持 CE-ATA 功能(与 CE-ATA 数字协议版本 1.1 全兼容)。 8 位总线模式下数据传输速率可达 48MHz。
- 数据和命令输出使能信号，用于控制外部双向驱动器。

STM32 的 SDIO 控制器包含 2 个部分：SDIO 适配器模块和 AHB 总线接口，其功能框图如图 1.1.1 所示：

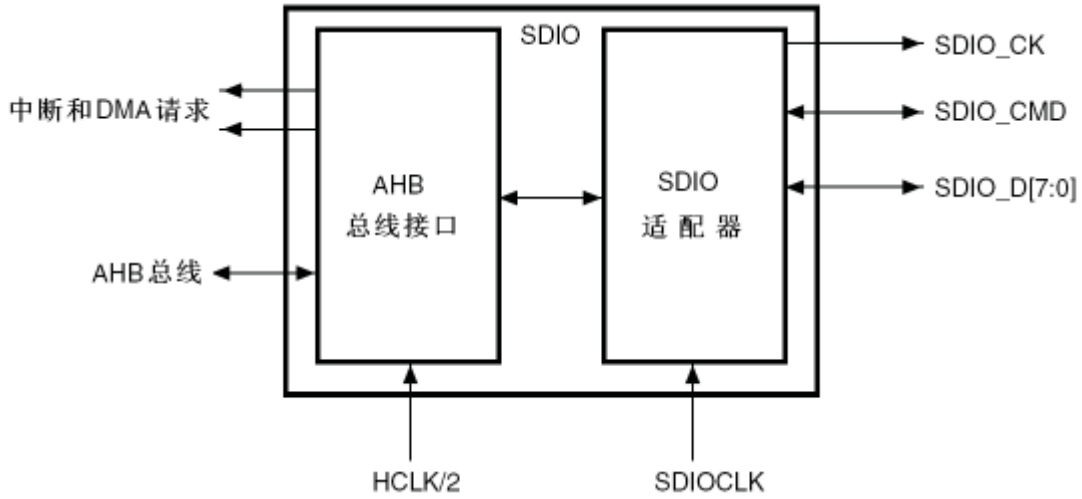


图 1.1.1 STM32 的 SDIO 控制器功能框图

复位后默认情况下 SDIO_D0 用于数据传输。初始化后主机可以改变数据总线的宽度（通过 ACMD6 命令设置）。

如果一个多媒体卡接到了总线上，则 SDIO_D0、SDIO_D[3:0]或 SDIO_D[7:0]可以用于数据传输。MMC 版本 V3.31 和之前版本的协议只支持 1 位数据线，所以只能用 SDIO_D0（为了通用性考虑，在程序里面我们只要检测到是 MMC 卡就设置为 1 位总线数据）。

如果一个 SD 或 SD I/O 卡接到了总线上，可以通过主机配置数据传输使用 SDIO_D0 或 SDIO_D[3:0]。所有的数据线都工作在推挽模式。

SDIO_CMD 有两种操作模式：

① 用于初始化时的开路模式(仅用于 MMC 版本 V3.31 或之前版本)

② 用于命令传输的推挽模式(SD/SD I/O 卡和 MMC V4.2 在初始化时也使用推挽驱动)

1.2 SDIO 的时钟

从图 1.1.1 我们可以看到 SDIO 总共有 3 个时钟，分别是：

卡时钟（SDIO_CK）：每个时钟周期在命令和数据线上传输 1 位命令或数据。对于多媒体卡 V3.31 协议，时钟频率可以在 0MHz 至 20MHz 间变化；对于多媒体卡 V4.0/4.2 协议，时钟频率可以在 0MHz 至 48MHz 间变化；对于 SD 或 SD I/O 卡，时钟频率可以在 0MHz 至 25MHz 间变化。

SDIO 适配器时钟（SDIOCLK）：该时钟用于驱动 SDIO 适配器，其频率等于 AHB 总线频率（HCLK），并用于产生 SDIO_CK 时钟。

AHB 总线接口时钟(HCLK/2)：该时钟用于驱动 SDIO 的 AHB 总线接口，其频率为 HCLK/2。

前面提到，我们的 SD 卡时钟（SDIO_CK），根据卡的不同，可能有好几个区间，这就涉及到时钟频率的设置，SDIO_CK 与 SDIOCLK 的关系为：

SDIO_CK=SDIOCLK/(2+CLKDIV)

其中，SDIOCLK 为 HCLK，一般是 72Mhz，而 CLKDIV 则是分配系数，可以通过 SDIO 的 SDIO_CLKCR 寄存器进行设置（确保 SDIO_CK 不超过卡的最大操作频率）。

这里要提醒大家，在 SD 卡刚刚初始化的时候，其时钟频率（SDIO_CK）是不能超过 400Khz 的，否则可能无法完成初始化。在初始化以后，就可以设置时钟频率到最大了（但不可超过 SD 卡的最大操作时钟频率）。

1.3 SDIO 的命令与响应

SDIO 的命令分为应用相关命令（ACMD）和通用命令（CMD）两部分，应用相关命令（ACMD）的发送，必须先发送通用命令（CMD55），然后才能发送应用相关命令（ACMD）。SDIO 的所有命令和响应都是通过 SDIO_CMD 引脚传输的，任何命令的长度都是固定为 48 位，SDIO 的命令格式如表 1.3.1 所示：

位	宽度	数值	说明
47	1	0	开始位
46	1	1	传输位
[45:40]	6	-	命令索引
[39:8]	32	-	参数
[7:1]	7	-	CRC7
0	1	1	结束位

表 1.3.1 SDIO 命令格式

所有的命令都是由 STM32 发出，其中开始位、传输位、CRC7 和结束位由 SDIO 硬件控制，我们需要设置的就只有命令索引和参数部分。其中命令索引（如 CMD0，CMD1 之类的）在 SDIO_CMD 寄存器里面设置，命令参数则由寄存器 SDIO_ARG 设置。

一般情况下，选中的 SD 卡在接收到命令之后，都会回复一个应答（注意 CMD0 是无应答的），这个应答我们称之为响应，响应也是在 CMD 线上串行传输的。STM32 的 SDIO 控制器支持 2 种响应类型，即：短响应（48 位）和长响应（136 位），这两种响应类型都带 CRC 错误检测（注意不带 CRC 的响应应该忽略 CRC 错误标志，如 CMD1 的响应）。

短响应的格式如表 1.3.2 所示：

位	宽度	数值	说明
47	1	0	开始位
46	1	0	传输位
[45:40]	6	-	命令索引
[39:8]	32	-	参数
[7:1]	7	-	CRC7(或1111111)
0	1	1	结束位

表 1.3.2 SDIO 命令格式

长响应的格式如表 1.3.3 所示：

位	宽度	数值	说明
135	1	0	开始位
134	1	0	传输位
[133:128]	6	111111	保留
[127:1]	127	-	CID或CSD(包含内部CRC7)
0	1	1	结束位

表 1.3.3 SDIO 命令格式

同样，硬件为我们滤除了开始位、传输位、CRC7 以及结束位等信息，对于短响应，命令索引存放在 SDIO_RESPCMD 寄存器，参数则存放在 SDIO_RESP1 寄存器里面。对于长响应，则仅留 CID/CSD 位域，存放在 SDIO_RESP1~SDIO_RESP4 等 4 个寄存器。

SD 存储卡总共有 5 类响应（R1、R2、R3、R6、R7），我们这里以 R1 为例简单介绍一下。R1（普通响应命令）响应输入短响应，其长度为 48 位，R1 响应的格式如表 1.3.4 所示：

位	域宽度	数值	说明
47	1	0	开始位
46	1	0	传输位
[45:40]	6	X	命令索引
[39:8]	32	X	卡状态
[7:1]	7	X	CRC7
0	1	1	结束位

表 1.3.4 R1 响应格式

在收到 R1 响应后，我们可以从 SDIO_RESPCMD 寄存器和 SDIO_RESP1 寄存器分别读出命令索引和卡状态信息。关于其他响应的介绍，请大家参考光盘:《SD 卡 2.0 协议.pdf》或《STM32 中文参考手册_V10.pdf》。

最后，我们看看数据在 SDIO 控制器与 SD 卡之间的传输。对于 SDI/SDIO 存储器，数据是以数据块的形式传输的，而对于 MMC 卡，数据是以数据块或者数据流的形式传输。本节我们只考虑数据块形式的数据传输。

SDIO（多）数据块读操作，如图 1.3.1 所示：

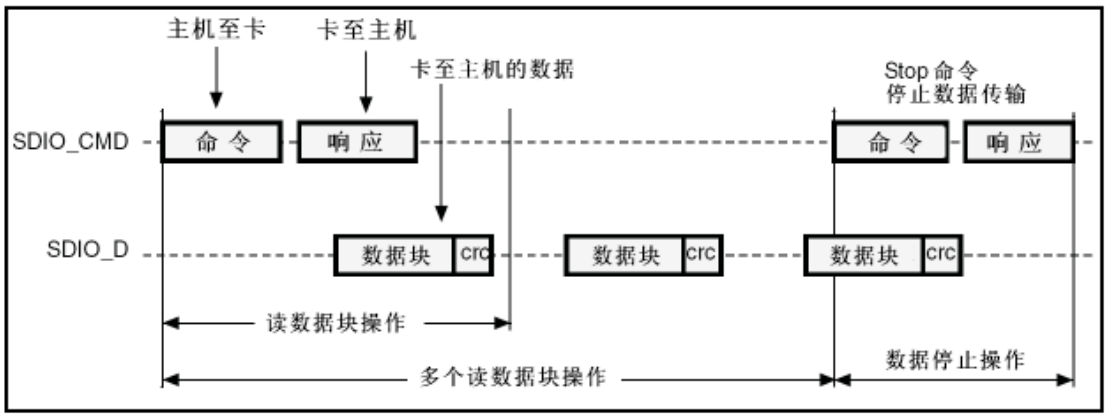


图 1.3.1 SDIO（多）数据块读操作

从上图，我们可以看出，从机在收到主机相关命令后，开始发送数据块给主机，所有数据块都带有 CRC 校验值（CRC 由 SDIO 硬件自动处理），单个数据块读的时候，在收到 1 个数据块以后即可以停止了，不需要发送停止命令（CMD12）。但是多块数据读的时候，SD 卡将一直发送数据给主机，直到接到主机发送的 STOP 命令（CMD12）。

SDIO（多）数据块写操作，如图 1.3.2 所示：

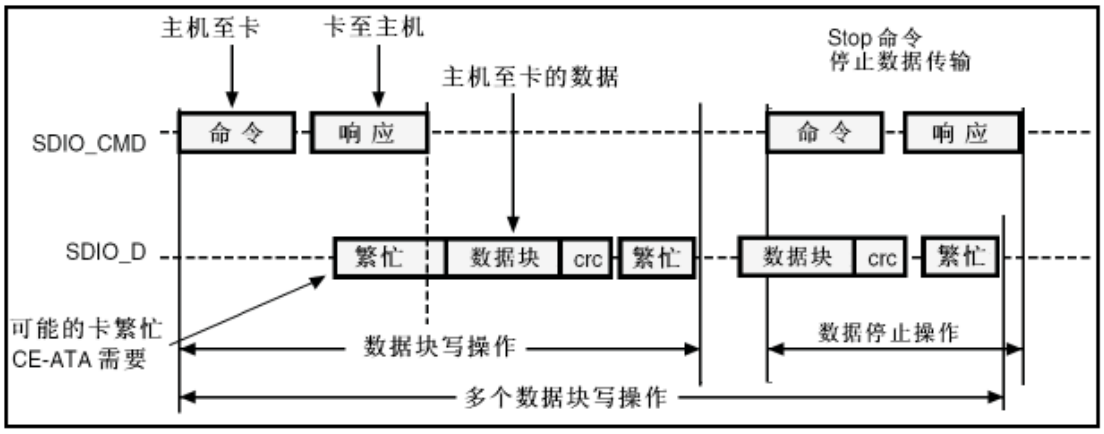


图 1.3.2 SDIO（多）数据块写操作

数据块写操作同数据块读操作基本类似，只是数据块写的时候，多了一个繁忙判断，新的数据块必须在 SD 卡非繁忙的时候发送。这里的繁忙信号由 SD 卡拉低 SDIO_D0，以表示繁忙，SDIO 硬件自动控制，不需要我们软件处理。

SDIO 的命令与响应就为大家介绍到这里。

1.4 SDIO 相关寄存器介绍

第一个，我们来看 SDIO 电源控制寄存器（SDIO_POWER），该寄存器定义如图 1.4.1 所示：

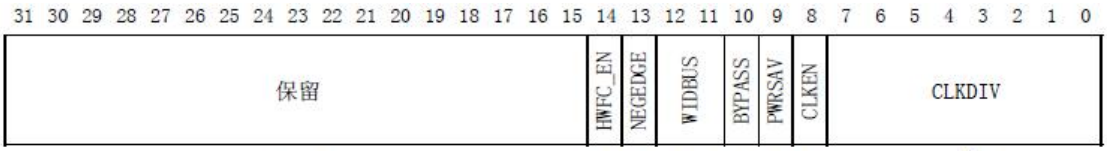
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留																														PWRCTRL	
res																														rw	

位31:2	保留，始终读为0。
位1:0	**PWRCTRL**：电源控制位 (Power supply control bits) 这些位用于定义卡时钟的当前功能状态： 00：电源关闭，卡的时钟停止。 01：保留。 10：保留的上电状态。 11：上电状态，卡的时钟开启。

图 1.4.1 SDIO_POWER 寄存器位定义

该寄存器复位值为 0，所以 SDIO 的电源是关闭的，我们要启用 SDIO，第一步就是要设置该寄存器最低 2 个位均为 1，让 SDIO 上电，开启卡时钟。

第二个，我们看 SDIO 时钟控制寄存器（SDIO_CLKCR），该寄存器主要用于设置 SDIO_CK 的分配系数，开关等，并可以设置 SDIO 的数据位宽，该寄存器的定义如图 1.4.2 所示：



位31:15	保留，始终读为0。
位14	**HWFC_EN**：硬件流控制使能 (HW Flow Control enable) 0：关闭硬件流控制 1：使能硬件流控制 当使能硬件流控制后，关于TXFIFOE和RXFIFOF中断信号的意义请参考20.9.11节的SDIO状态寄存器的定义。
位13	**NEGEDGE**：SDIO_CK相位选择位 (SDIO_CK dephasing selection bit) 0：在主时钟SDIOCLK的上升沿产生SDIO_CK。 1：在主时钟SDIOCLK的下降沿产生SDIO_CK。
位12:11	**WIDBUS**：宽总线模式使能位 (Wide bus mode enable bit) 00：默认总线模式，使用SDIO_D0。 01：4位总线模式，使用SDIO_D[3:0]。 10：8位总线模式，使用SDIO_D[7:0]。
位10	**BYPASS**：旁路时钟分频器 (Clock divider bypass enable bit) 0：关闭旁路：驱动SDIO_CK输出信号之前，依据CLKDIV数值对SDIOCLK分频。 1：使能旁路：SDIOCLK直接驱动SDIO_CK输出信号。
位9	**PWRSAV**：省电配置位 (Power saving configuration bit) 为了省电，当总线为空闲时，设置PWRSAV位可以关闭SDIO_CK时钟输出。 0：始终输出SDIO_CK。 1：仅在有总线活动时才输出SDIO_CK。
位8	**CLKEN**：时钟使能位 (Clock enable bit) 0：SDIO_CK关闭。 1：SDIO_CK使能。
位7:0	**CLKDIV**：时钟分频系数 (Clock divide factor) 这个域定义了输入时钟(SDIOCLK)与输出时钟(SDIO_CK)间的分频系数： SDIO_CK频率 = SDIOCLK/[CLKDIV + 2]。

图 1.4.2 SDIO_CLKCR 寄存器位定义

从上图可以看出时钟控制寄存器的最低 8 位，用于控制 SDIO_CK 的分配，位 8 用于控制 SDIO_CK 的开启和关闭。我们还可以通过位[12:11]，控制 SDIO 的数据位宽。

第三个，我们要介绍的是 SDIO 参数制寄存器（SDIO_ARG），该寄存器比较简单，就是一个 32 位寄存器，用于存储命令参数，不过需要注意的是，必须在写命令之前先写这个参数寄存器！

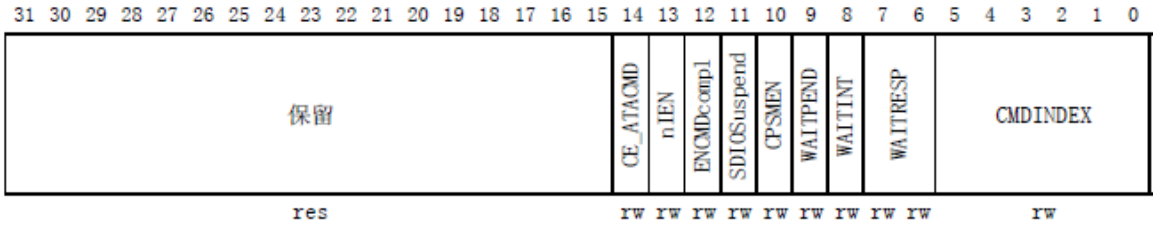
第四个，我们要介绍的是 SDIO 命令响应寄存器（SDIO_RESPCMD），该寄存器为 32 位，比较简单，用于存储最后收到的命令响应中的命令索引。如果传输的命令响应不包含命令索引，则该寄存器的内容不可预知。

第五个，我们要介绍的是 SDIO 响应寄存器组（SDIO_RESP1~SDIO_RESP4），该寄存器组总共由 4 个 32 位寄存器组成，用于存放接收到的卡响应部分信息。如果收到短响应，则数据存放在 SDIO_RESP1 寄存器里面，其他三个寄存器没有用到。而如果收到长响应，则依次存放在 SDIO_RESP1~ SDIO_RESP4 里面，如表 1.4.1 所示：

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]

表 1.4.1 响应类型和 SDIO_RESPx 寄存器

第七个，我们介绍 SDIO 命令寄存器（SDIO_CMD），该寄存器各位定义如图 1.4.3 所示：



位31:15	保留，始终读为0
位14	**ATACMD**：CE-ATA命令 (CE-ATA command) 如果设置该位，CPSM转至CMD61。
位13	**nIEN**：不使能中断 (not interrupt enable) 如果未设置该位，则使能CE-ATA设备的中断。
位12	**ENCMDcompl**：使能CMD完成 (Enable CMD completion) 如果设置该位，则使能命令完成信号。
位11	**SDIOSuspend**：SD I/O暂停命令 (SD I/O suspend command) 如果设置该位，则将要发送的命令是一个暂停命令(只能用于SDIO卡)。
位10	**CPSMEN**：命令通道状态机(CPSM)使能位 (Command path state machine (CPSM) Enable bit) 如果设置该位，则使能CPSM。
位9	**WAITPEND**：CPSM等待数据传输结束(CmdPend内部信号) (CPSM Waits for ends of data transfer (CmdPend internal signal)) 如果设置该位，则CPSM在开始发送一个命令之前等待数据传输结束。
位8	**WAITINT**：CPSM等待中断请求 (CPSM waits for interrupt request) 如果设置该位，则CPSM关闭命令超时控制并等待中断请求。
位7:6	**WAITRESP**：等待响应位 (Wait for response bits) 这2位指示CPSM是否需要等待响应，如果需要等待响应，则指示响应类型。 00：无响应，期待CMDSENT标志 01：短响应，期待CMDREND或CCRCFAIL标志 10：无响应，期待CMDSENT标志 11：长响应，期待CMDREND或CCRCFAIL标志
位5:0	**CMDINDEX**：命令索引 (Command index) 命令索引是作为命令的一部分发送到卡中。

图 1.4.3 SDIO_CMD 寄存器位定义

图中，我们这里只关心最低 8 位，其中最低 6 位为命令索引，也就是我们要发送的命令索引号（比如发送 CMD1，其值为 1，索引就设置为 1）。位[7:6]，用于设置等待响应位，用于指示 CPSM 是否需要等待，以及等待类型等。这里的 CPSM，即命令通道状态机，我们就不详细介绍了，请参阅《STM32 中文参考手册》V10 的第 368 页，有详细介绍。命令通道状态机我们一般都是开启的，所以位 10 要设置为 1。

第八个，我们要介绍的是 SDIO 数据定时器寄存器（SDIO_DTIMER），该寄存器用于存储以卡总线时钟（SDIO_CK）为周期的数据超时时间，一个计数器将从 SDIO_DTIMER 寄存器加载数值，并在数据通道状态机(DPSM)进入 Wait_R 或繁忙状态时进行递减计数，当 DPSM 处在这些状态时，如果计数器减为 0，则设置超时标志。这里的 DPSM，即数据通道状态机，类似 CPSM，详细请参考《STM32 中文参考手册》V10 的第 372 页。注意：在写入数据控制寄存器，进行数据传输之前，必须先写入该寄存器（SDIO_DTIMER）和数据长度寄存器（SDIO_DLEN）!

第九个，我们要介绍的是 SDIO 数据长度寄存器（SDIO_DLEN），该寄存器低 24 位有效，用于设置需要传输的数据字节长度。对于块数据传输，该寄存器的数值，必须是数据块长度

（通过 SDIO_DCTRL 设置）的倍数。

第十个，我们要介绍的是 SDIO 数据控制寄存器（SDIO_DCTRL），该寄存器各位定义如图 1.4.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留																				SDIOEN	RWMOD	RWSTOP	RWSTART	DBLOCKSIZE				DMAEN	DTMODE	DTDIR	DTEN
res																				rw	rw	rw	rw	rw				rw	rw	rw	rw

位31:12	保留，始终读为0。
位11	**SDIOEN**：SD I/O使能功能 (SD I/O enable functions) 如果设置了该位，则DPSM执行SD I/O卡特定的操作。
位10	**RWMOD**：读等待模式 (Read wait mode) 0：停止SDIO_CK控制读等待； 1：使用SDIO_D2控制读等待。
位9	**RWSTOP**：读等待停止 (Read wait stop) 0：如果设置了RWSTART，执行读等待； 1：如果设置了RWSTART，停止读等待。
位8	**RWSTART**：读等待开始 (Read wait start) 设置该位开始读等待操作。
位7:4	**DBLOCKSIZE**：数据块长度 (Data block size) 当选择了块数据传输模式，该域定义数据块长度： 0000：块长度 = 2^0 = 1字节；　1000：块长度 = 2^8 = 256字节； 0001：块长度 = 2^1 = 2字节；　1001：块长度 = 2^9 = 512字节； 0010：块长度 = 2^2 = 4字节；　1010：块长度 = 2^{10} = 1024字节； 0011：块长度 = 2^3 = 8字节；　1011：块长度 = 2^{11} = 2048字节； 0100：(十进制4)块长度 = 2^4 = 16字节；　1100：块长度 = 2^{12} = 4096字节； 0101：(十进制5)块长度 = 2^5 = 32字节；　1101：块长度 = 2^{13} = 8192字节； 0110：(十进制6)块长度 = 2^6 = 64字节；　1110：块长度 = 2^{14} = 16384字节； 0111：块长度 = 2^7 = 128字节；　1111：保留。
位3	**DMAEN**：DMA使能位 (DMA enable bit) 0：关闭DMA； 1：使能DMA。
位2	**DTMODE**：数据传输模式 (Data transfer mode selection) 0：块数据传输； 1：流数据传输。
位1	**DTDIR**：数据传输方向 (Data transfer direction selection) 0：控制器至卡； 1：卡至控制器。
位0	**DTEN**：数据传输使能位 (Data transfer enabled bit) 如果设置该位为1，则开始数据传输。根据DTSIR方向位，DPSM进入Wait_S或Wait_R状态，如果在传输的一开始就设置了RWSTART位，则DPSM进入读等待状态。不需要在数据传输结束后清除使能位，但必须更改SDIO_DCTRL以允许新的数据传输。

图 1.4.4 SDIO_DCTRL 寄存器位定义

该寄存器，用于控制数据通道状态机（DPSM），包括数据传输使能、传输方向、传输模式、DMA 使能、数据块长度等信息，都是通过该寄存器设置。我们需要根据自己的实际情况，来配置该寄存器，才可正常实现数据收发。

接下来，我们介绍几个位定义十分类似的寄存器，他们是：状态寄存器（SDIO_STA）、清除中断寄存器（SDIO_ICR）和中断屏蔽寄存器（SDIO_MASK），这三个寄存器每个位的定义都相同，只是功能各有不同。所以可以一起介绍，以状态寄存器（SDIO_STA）为例，该寄存器各位定义如图 1.4.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留								CEATAEND	SDIOIT	RXDAVL	TXDAVL	RXFIFOE	TXFIFOE	RXFIFOF	TXFIFOF	RXFIFOHF	TXFIFOHE	RXACT	TXACT	CMDACT	DBCKEND	STBITERR	DATAEND	CMDSENT	CMDREND	RXOVERR	TXUNDERR	DTIMEOUT	CTIMEOUT	DCRCFAIL	CCRCFAIL
res								r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位31:24	保留，始终读为0。
位23	**CEATAEND**：在CMD61接收到CE-ATA命令完成信号 (CE-ATA command completion signal received for CMD61)
位22	**SDIOIT**：收到SDIO中断 (SDIO interrupt received)
位21	**RXDVAL**：在接收FIFO中的数据可用 (Data available in receive FIFO)
位20	**TXDVAL**：在发送FIFO中的数据可用 (Data available in transmit FIFO)
位19	**RXFIFOE**：接收FIFO空 (Receive FIFO empty)
位18	**TXFIFOE**：发送FIFO空 (Transmit FIFO empty) 若使用了硬件流控制，当FIFO包含2个字时，TXFIFOE信号变为有效。
位17	**RXFIFOF**：接收FIFO满 (Receive FIFO full) 若使用了硬件流控制，当FIFO还差2个字满时，RXFIFOF信号变为有效。
位16	**TXFIFOF**：发送FIFO满 (Transmit FIFO full)
位15	**RXFIFOHF**：接收FIFO半满 (Receive FIFO half full)：FIFO中至少还有8个字。
位14	**TXFIFOHE**：发送FIFO半空 (Transmit FIFO half empty)：FIFO中至少还可以写入8个字。
位13	**RXACT**：正在接收数据 (Data receive in progress)
位12	**TXACT**：正在发送数据 (Data transmit in progress)
位11	**CMDACT**：正在传输命令 (Command transfer in progress)
位10	**DBCKEND**：已发送/接收数据块(CRC检测成功) (Data block sent/received (CRC check passed))
位9	**STBITERR**：在宽总线模式，没有在所有数据信号上检测到起始位 (Start bit not detected on all data signals in wide bus mode)
位8	**DATAEND**：数据结束(数据计数器，SDIO_DCOUNT = 0) (Data end (data counter, SDIDCOUNT, is zero))
位7	**CMDSENT**：命令已发送(不需要响应) (Command sent (no response required))
位6	**CMDREND**：已接收到响应(CRC检测成功) (Command response)
位5	**RXOVERR**：接收FIFO上溢错误 (Received FIFO overrun error)
位4	**TXUNDERR**：发送FIFO下溢错误 (Transmit FIFO underrun error)
位3	**DTIMEOUT**：数据超时 (Data timeout)
位2	**CTIMEOUT**：命令响应超时 (Command response timeout) 命令超时时间是一个固定的值，为64个SDIO_CK时钟周期。
位1	**DCRCFAIL**：已发送/接收数据块(CRC检测失败) (Data block sent/received)
位0	**CCRCFAIL**：已收到命令响应(CRC检测失败) (Command response received)

图 1.4.5 SDIO_STA 寄存器位定义

状态寄存器可以用来查询 SDIO 控制器的当前状态，以便处理各种事务。比如 SDIO_STA 的位 2 表示命令响应超时，说明 SDIO 的命令响应出了问题。我们通过设置 SDIO_ICR 的位 2 则可以清除这个超时标志，而设置 SDIO_MASK 的位 2，则可以开启命令响应超时中断，设置为 0 关闭。其他位我们就不一一介绍了，请大家自行学习。

最后，我们向大家介绍 SDIO 的数据 FIFO 寄存器（SDIO_FIFO），数据 FIFO 寄存器包括接收和发送 FIFO，他们由一组连续的 32 个地址上的 32 个寄存器组成，CPU 可以使用 FIFO 读写多个操作数。例如我们要从 SD 卡读数据，就必须读 SDIO_FIFO 寄存器，要写数据到 SD 卡，则要写 SDIO_FIFO 寄存器。SDIO 将这 32 个地址分为 16 个一组，发送接收各占一半。而我们每次读写的时候，最多就是读取发送 FIFO 或写入接收 FIFO 的一半大小的数据，也就是 8 个字（32 个字节），**这里特别提醒，我们操作 SDIO_FIFO（不论读出还是写入）必须是以 4 字节对齐的内存进行操作，否则将导致出错！**

至此，SDIO 的相关寄存器介绍，我们就介绍完了。还有几个不常用的寄存器，我们没有介绍到，请大家参考《STM32 中文参考手册》的 SDIO 寄存器介绍部分。

1.5 SD 卡初始化流程

最后，我们来看看 SD 卡的初始化流程，要实现 SDIO 驱动 SD 卡，最重要的步骤就是 SD 卡的初始化，只要 SD 卡初始化完成了，那么剩下的（读写操作）就简单了，所以我们这里重点介绍 SD 卡的初始化。从 SD 卡 2.0 协议（见光盘资料）文档，我们得到 SD 卡初始化流程图如图 1.5.1 所示：

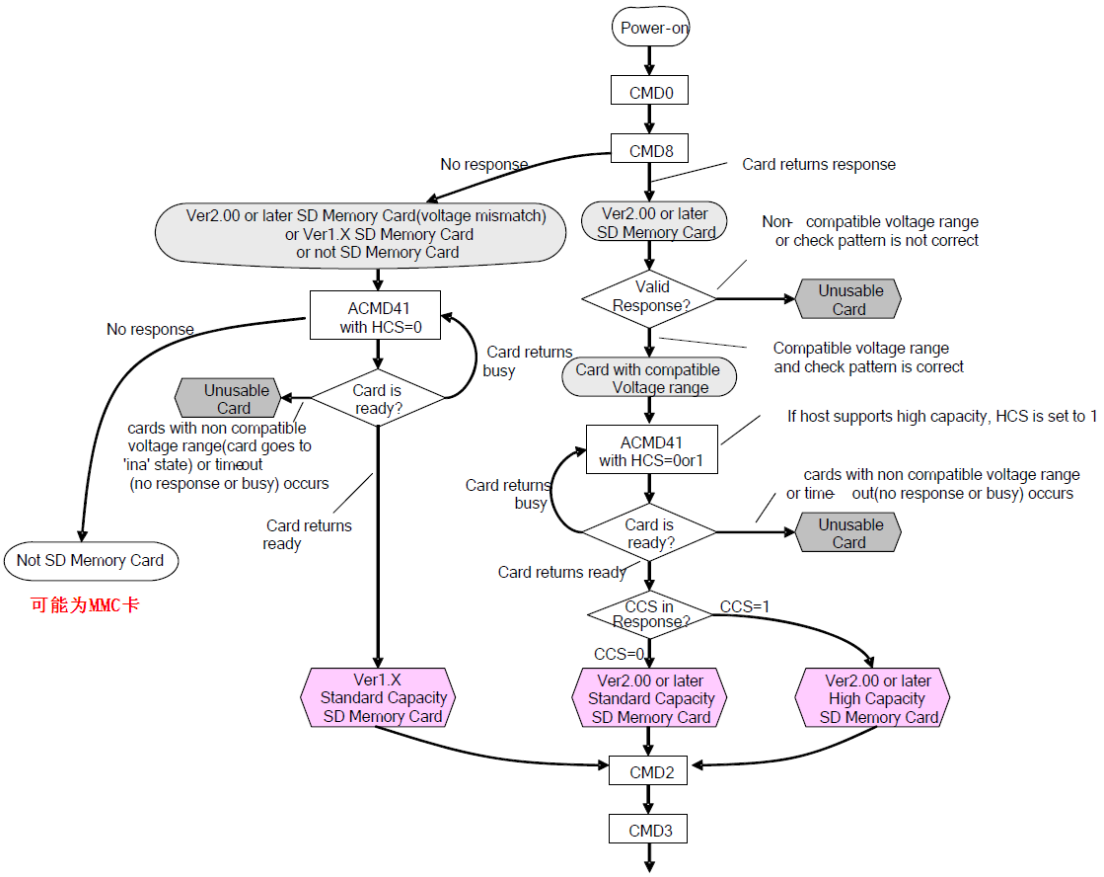


图 1.5.1 SD 卡初始化流程

从图中，我们看到，不管什么卡（这里我们将卡分为 4 类：SD2.0 高容量卡（SDHC，最大 32G），SD2.0 标准容量卡（SDSC，最大 2G），SD1.x 卡和 MMC 卡），首先我们要执行的是卡上电（需要设置 SDIO_POWER[1:0]=11），上电后发送 CMD0，对卡进行软复位，之后发送 CMD8 命令，用于区分 SD 卡 2.0，只有 2.0 及以后的卡才支持 CMD8 命令，MMC 卡和 V1.x 的卡，是不支持该命令的。CMD8 的格式如表 1.5.1 所示：

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

表 1.5.1 CMD8 命令格式

这里，我们需要在发送 CMD8 的时候，通过其带的参数我们可以设置 VHS 位，以告诉 SD 卡，主机的供电情况，VHS 位定义如表 1.5.2 所示：

Voltage Supplied	Value Definition
0000b	Not Defined
0001b	2.7-3.6V
0010b	Reserved for Low Voltage Range
0100b	Reserved
1000b	Reserved
Others	Not Defined

表 1.5.2 VHS 位定义

这里我们使用参数 0X1AA，即告诉 SD 卡，主机供电为 2.7~3.6V 之间，如果 SD 卡支持 CMD8，且支持该电压范围，则会通过 CMD8 的响应（R7）将参数部分原本返回给主机，如果不支持 CMD8，或者不支持这个电压范围，则不响应。

在发送 CMD8 后，发送 ACMD41（注意发送 ACMD41 之前要先发送 CMD55），来进一步确认卡的操作电压范围，并通过 HCS 位来告诉 SD 卡，主机是不是支持高容量卡（SDHC）。ACMD41 的命令格式如表 1.5.3 所示：

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].

表 1.5.3 ACMD41 命令格式

ACMD41 得到的响应（R3）包含 SD 卡 OCR 寄存器内容，OCR 寄存器内容定义如表 1.5.4 所示：

OCR bit position	OCR Fields Definition
0-6	reserved
7	Reserved for Low Voltage Range
8-14	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.

表 1.5.4 OCR 寄存器定义

对于支持 CMD8 指令的卡，主机通过 ACMD41 的参数设置 HCS 位为 1，来告诉 SD 卡主机支 SDHC 卡，如果设置为 0，则表示主机不支持 SDHC 卡，SDHC 卡如果接收到 HCS 为 0，则永远不会反回卡就绪状态。对于不支持 CMD8 的卡，HCS 位设置为 0 即可。

SD 卡在接收到 ACMD41 后，返回 OCR 寄存器内容，如果是 2.0 的卡，主机可以通过判断 OCR 的 CCS 位来判断是 SDHC 还是 SDSC；如果是 1.x 的卡，则忽略该位。OCR 寄存器的最后一个位用于告诉主机 SD 卡是否上电完成，如果上电完成，该位将会被置 1。

对于 MMC 卡，则不支持 ACMD41，不响应 CMD55，对 MMC 卡，我们只需要在发送 CMD0 后，在发送 CMD1（作用同 ACMD41），检查 MMC 卡的 OCR 寄存器，实现 MMC 卡的初始化。

至此，我们便实现了对 SD 卡的类型区分，图 1.5.1 中，最后发送了 CMD2 和 CMD3 命令，用于获得卡 CID 寄存器数据和卡相对地址（RCA）。

CMD2，用于获得 CID 寄存器的数据，CID 寄存器数据各位定义如表 1.5.5 所示：

Name	Field	Width	CID-slice
Manufacturer ID	MID	8	[127:120]
OEM/Application ID	OID	16	[119:104]
Product name	PNM	40	[103:64]
Product revision	PRV	8	[63:56]
Product serial number	PSN	32	[55:24]
reserved	--	4	[23:20]
Manufacturing date	MDT	12	[19:8]
CRC7 checksum	CRC	7	[7:1]
not used, always 1	-	1	[0:0]

表 1.5.5 卡 CID 寄存器位定义

SD 卡在收到 CMD2 后，将返回 R2 长响应（136 位），其中包含 128 位有效数据（CID 寄存器内容），存放在 SDIO_RESP1~4 等 4 个寄存器里面。通过读取这四个寄存器，就可以获得 SD 卡的 CID 信息。

CMD3，用于设置卡相对地址（RCA，必须为非 0），对于 SD 卡（非 MMC 卡），在收到 CMD3 后，将返回一个新的 RCA 给主机，方便主机寻址。RCA 的存在允许一个 SDIO 接口挂多个 SD 卡，通过 RCA 来区分主机要操作的是哪个卡。而对于 MMC 卡，则不是由 SD 卡自动返回 RCA，而是主机主动设置 MMC 卡的 RCA，即通过 CMD3 带参数（高 16 位用于 RCA 设置），实现 RCA 设置。同样 MMC 卡也支持一个 SDIO 接口挂多个 MMC 卡，不同于 SD 卡的是所有的 RCA 都是由主机主动设置的，而 SD 卡的 RCA 则是 SD 卡发给主机的。

在获得卡 RCA 之后，我们便可以发送 CMD9（带 RCA 参数），获得 SD 卡的 CSD 寄存器内容，从 CSD 寄存器，我们可以得到 SD 卡的容量和扇区大小等十分重要的信息。CSD 寄存器我们在这里就不详细介绍了，关于 CSD 寄存器的详细介绍，请大家参考《SD 卡 2.0 协议.pdf》。

至此，我们的 SD 卡初始化基本就结束了，最后通过 CMD7 命令，选中我们要操作的 SD 卡，即可开始对 SD 卡的读写操作了，SD 卡的其他命令和参数，我们这里就不再介绍了，请大家参考《SD 卡 2.0 协议.pdf》，里面有非常详细的介绍。

2、硬件连接

本文档源码改自标准例程的图片显示实验，标准例程用的是 SPI 驱动 SD 卡，本文档用的是 SDIO，并新增了 bmp 截屏保存和 sd 卡写测试功能。

功能简介：开机的时候先检测字库，然后检测 SD 卡是否存在，如果 SD 卡存在，则初始化 SD 卡，并通过串口 1 打印 SD 卡相关信息，然后开始查找 SD 卡根目录下的 PICTURE 文件夹，如果找到则显示该文件夹下面的图片文件（支持 bmp、jpg、jpeg 或 gif 格式），循环显示，通过按 KEY0 和 KEY2 可以快速浏览下一张和上一张，WK_UP 按键用于暂停/继续播放，DS1 用于指示当前是否处于暂停状态。如果未找到 PICTURE 文件夹/任何图片文件，则提示错误。DS0 来指示程序正在运行。在正常显示图片的时候，我们可以通过串口，利用 usmart 调用 bmp_encode 或 testwrite 函数，测试 SDIO 的写功能。其中 bmp_encode 用于截取屏幕区域部分内容保存为 16 位 bmp 图片，可以测试 SDIO 的单块写功能。而 testwrite 函数，则可以实现在 SD 卡创建一个文件（方便测试，一般是.txt）实现 SD 卡单块/多块写功能的测试。

所要用到的硬件资源如下：

1） 指示灯 DS0 、DS1

2） KEY0/KEY1 两个按键

3） 串口

4） SDIO(外加一个 SD 卡)

5） TFTLCD 模块

战舰 STM32 开发板默认将 SD 卡连接在 SPI 接口，所以，我们需要将开发板上的 P10 和 P11 排针用跳线帽连接起来，从而将 SD 卡连接到 SDIO 接口上，如图 2.1 所示：

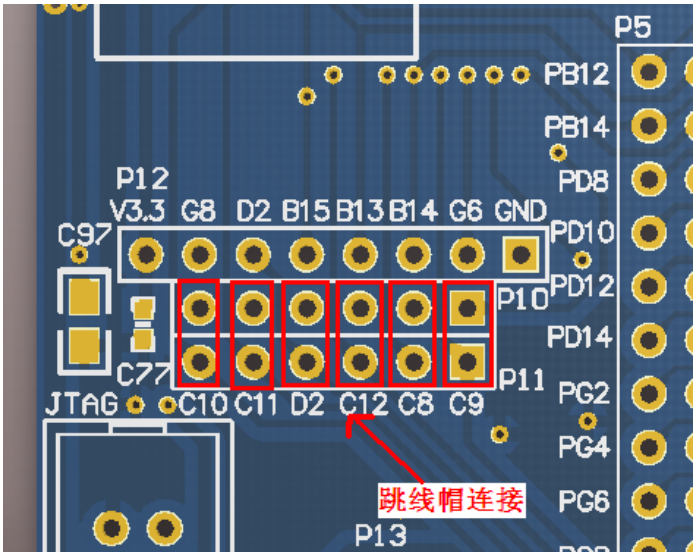


图 2.1 SDIO 连接 SD 卡示意图

将原来连接 P10 和 P12 的跳线帽，都设置到下方，即实现 SD 卡与 SDIO 的连接。这也是我们在硬件上唯一需要改动的地方，其他不需要做任何修改。

3、软件实现

本实验，我们在战舰 STM32 开发板标准实验 42 的基础上修改，在 HARDWARE 组下新建 SDIO 文件夹，并新建 sdio_sdcard.c 和 sdio_sdcard.h 两个文件存放 SDIO 驱动部分代码，这里我们重点分析 sdio_sdcard.c 里面的部分代码，其他一些修改细节就不细说了，详见战舰 STM32 开发板扩展实验 2 的源码。

在 sdio_sdcard.c 里面，代码比较多，在这里仅挑其中重要的几个函数进行讲解。

首先是 SD 卡初始化函数：SD_Init，该函数代码如下：

```
//初始化 SD 卡
```

```
//返回值:错误代码;(0,无错误)
SD_Error SD_Init(void)
{
    SD_Error errorstatus=SD_OK;
    //SDIO IO 口初始化
    RCC->APB2ENR|=1<<4;          //使能 PORTC 时钟
    RCC->APB2ENR|=1<<5;          //使能 PORTD 时钟
    RCC->AHBENR|=1<<10;          //使能 SDIO 时钟
    RCC->AHBENR|=1<<1;           //使能 DMA2 时钟
    GPIOC->CRH&=0XFFF00000;
    GPIOC->CRH|=0X000BBBBB;   //PC.8~12  复用输出
    GPIOD->CRL&=0XFFFFF0FF;
    GPIOD->CRL|=0X00000B00;     //PD2 复用输出,PD7  上拉输入
    //SDIO 外设寄存器设置为默认值
    SDIO->POWER=0x00000000;
    SDIO->CLKCR=0x00000000;
    SDIO->ARG=0x00000000;
    SDIO->CMD=0x00000000;
    SDIO->DTIMER=0x00000000;
    SDIO->DLEN=0x00000000;
    SDIO->DCTRL=0x00000000;
    SDIO->ICR=0x00C007FF;
    SDIO->MASK=0x00000000;
    MY_NVIC_Init(0,0,SDIO_IRQChannel,2);//SDIO 中断配置
    errorstatus=SD_PowerON();              //SD 卡上电
    if(errorstatus==SD_OK)errorstatus=SD_InitializeCards();//初始化 SD 卡
    if(errorstatus==SD_OK)errorstatus=SD_GetCardInfo(&SDCardInfo);//获得卡信息
    if(errorstatus==SD_OK)errorstatus=SD_SelectDeselect((u32)(SDCardInfo.RCA<<16));
    //选中 SD 卡
    if(errorstatus==SD_OK)errorstatus=SD_EnableWideBusOperation(1);
    //4 位宽度,如果是 MMC 卡,则不能用 4 位模式
    if((errorstatus==SD_OK)||(SDIO_MULTIMEDIA_CARD==CardType))
    {
        SDIO_Clock_Set(SDIO_TRANSFER_CLK_DIV);//设置时钟频率
        errorstatus=SD_SetDeviceMode(SD_DMA_MODE);       //设置为 DMA 模式
        //errorstatus=SD_SetDeviceMode(SD_POLLING_MODE);//设置为查询模式
    }
    return errorstatus;
}
```

该函数先实现 SDIO 时钟及相关 IO 口的初始化，然后对 SDIO 部分寄存器进行了清零操作，然后开始 SD 卡的初始化流程，这个我们在 1.5 节有详细介绍了，首先 SD 卡的上电操作，由 SD_PowerOn 函数实现，该函数代码如下：

```
//卡上电
//查询所有 SDIO 接口上的卡设备,并查询其电压和配置时钟
```

```
//返回值:错误代码;(0,无错误)
SD_Error SD_PowerON(void)
{
    u8 i=0;
    SD_Error errorstatus=SD_OK;
    u32 response=0,count=0,validvoltage=0;
    u32 SDType=SD_STD_CAPACITY;
    //配置 CLKCR 寄存器
    SDIO->CLKCR=0;                  //清空 CLKCR 之前的设置
    SDIO->CLKCR|=0<<9;              //非省电模式
    SDIO->CLKCR|=0<<10;             //关闭旁路,CK 根据分频设置输出
    SDIO->CLKCR|=0<<11;             //1 位数据宽度
    SDIO->CLKCR|=0<<13;             //SDIOCLK 上升沿产生 SDIOCK
    SDIO->CLKCR|=0<<14;             //关闭硬件流控制
    SDIO_Clock_Set(SDIO_INIT_CLK_DIV);//设置时钟频率(不能超过 400Khz)
    SDIO->POWER=0X03;               //上电状态,开启卡时钟
    SDIO->CLKCR|=1<<8;              //SDIOCK 使能
    for(i=0;i<74;i++)
    {
        SDIO_Send_Cmd(SD_CMD_GO_IDLE_STATE,0,0);//发送 CMD0
        errorstatus=CmdError();
        if(errorstatus==SD_OK)break;
    }
    if(errorstatus)return errorstatus;//返回错误状态
    SDIO_Send_Cmd(SDIO_SEND_IF_COND,1,SD_CHECK_PATTERN);
    //发送 CMD8,短响应,检查 SD 卡接口特性.
    //arg[11:8]:01,支持电压范围,2.7~3.6V
    //arg[7:0]:默认 0XAA
    errorstatus=CmdResp7Error();                        //等待 R7 响应
    if(errorstatus==SD_OK)                              //R7 响应正常
    {
        CardType=SDIO_STD_CAPACITY_SD_CARD_V2_0;    //SD 2.0 卡
        SDType=SD_HIGH_CAPACITY;                        //高容量卡
    }
    SDIO_Send_Cmd(SD_CMD_APP_CMD,1,0);                  //发送 CMD55,短响应
    errorstatus=CmdResp1Error(SD_CMD_APP_CMD);          //等待 R1 响应
    if(errorstatus==SD_OK)//SD2.0/SD 1.1,否则为 MMC 卡
    {
        //SD 卡,发送 ACMD41 SD_APP_OP_COND,参数为:0x80100000
        while((!validvoltage)&&(count<SD_MAX_VOLT_TRIAL))
        {
            SDIO_Send_Cmd(SD_CMD_APP_CMD,1,0);//发送 CMD55,短响应
            errorstatus=CmdResp1Error(SD_CMD_APP_CMD);  //等待 R1 响应
            if(errorstatus!=SD_OK)return errorstatus;           //响应错误
```

```
                SDIO_Send_Cmd(SD_CMD_SD_APP_OP_COND,1,
                SD_VOLTAGE_WINDOW_SD|SDType);   //发送 ACMD41,短响应
                errorstatus=CmdResp3Error();                //等待 R3 响应
                if(errorstatus!=SD_OK)return errorstatus;   //响应错误
                response=SDIO->RESP1;;                      //得到响应
                validvoltage=(((response>>31)==1)?1:0);     //判断 SD 卡上电是否完成
                count++;
            }
            if(count>=SD_MAX_VOLT_TRIAL)
            {
                errorstatus=SD_INVALID_VOLTRANGE;
                return errorstatus;
            }
            if(response&=SD_HIGH_CAPACITY)
            {
                CardType=SDIO_HIGH_CAPACITY_SD_CARD;
            }
        }else//MMC 卡
        {
            CardType=SDIO_MULTIMEDIA_CARD;
            //MMC 卡,发送 CMD1 SDIO_SEND_OP_COND,参数为:0x80FF8000
            while((!validvoltage)&&(count<SD_MAX_VOLT_TRIAL))
            {
                SDIO_Send_Cmd(SD_CMD_SEND_OP_COND,1,
                SD_VOLTAGE_WINDOW_MMC);//发送 CMD1,短响应
                errorstatus=CmdResp3Error();                //等待 R3 响应
                if(errorstatus!=SD_OK)return errorstatus;   //响应错误
                response=SDIO->RESP1;;                      //得到响应
                validvoltage=(((response>>31)==1)?1:0);
                count++;
            }
            if(count>=SD_MAX_VOLT_TRIAL)
            {
                errorstatus=SD_INVALID_VOLTRANGE;
                return errorstatus;
            }
        }
        return(errorstatus);
    }
```

通过 SD_PowerON 函数，我们将完成 SD 卡的上电，并获得 SD 卡的类型（SDHC/SDSC /SDV1.x/MMC），随后哦，在 SD_Init 函数调用 SD_InitializeCards 函数，完成 SD 卡的初始化，该函数代码如下：

```
    //初始化所有的卡,并让卡进入就绪状态
    //返回值:错误代码
```

```
SD_Error SD_InitializeCards(void)
{
    SD_Error errorstatus=SD_OK;
    u16 rca = 0x01;
    if((SDIO->POWER&0X03)==0)return SD_REQUEST_NOT_APPLICABLE;
    //检查电源状态,确保为上电状态
    if(SDIO_SECURE_DIGITAL_IO_CARD!=CardType)
    {
        SDIO_Send_Cmd(SD_CMD_ALL_SEND_CID,3,0);  /CMD2,取得 CID,长响应
        errorstatus=CmdResp2Error();                    //等待 R2 响应
        if(errorstatus!=SD_OK)return errorstatus;       //响应错误
        CID_Tab[0]=SDIO->RESP1;
        CID_Tab[1]=SDIO->RESP2;
        CID_Tab[2]=SDIO->RESP3;
        CID_Tab[3]=SDIO->RESP4;
    }
    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))//判断卡类型
    {
        SDIO_Send_Cmd(SD_CMD_SET_REL_ADDR,1,0); //发送 CMD3,短响应
        errorstatus=CmdResp6Error(SD_CMD_SET_REL_ADDR,&rca);//等待 R6 响应
        if(errorstatus!=SD_OK)return errorstatus;       //响应错误
    }
    if (SDIO_MULTIMEDIA_CARD==CardType)//MMC 卡
    {
        SDIO_Send_Cmd(SD_CMD_SET_REL_ADDR,1,(u32)(rca<<16));//发送 CMD3
        errorstatus=CmdResp2Error();                    //等待 R2 响应
        if(errorstatus!=SD_OK)return errorstatus;       //响应错误
    }
    if (SDIO_SECURE_DIGITAL_IO_CARD!=CardType)
    {
        RCA = rca;
        SDIO_Send_Cmd(SD_CMD_SEND_CSD,3,(u32)(rca<<16));
        //发送 CMD9+卡 RCA,取得 CSD,长响应
        errorstatus=CmdResp2Error();                    //等待 R2 响应
        if(errorstatus!=SD_OK)return errorstatus;       //响应错误
        CSD_Tab[0]=SDIO->RESP1;
        CSD_Tab[1]=SDIO->RESP2;
        CSD_Tab[2]=SDIO->RESP3;
        CSD_Tab[3]=SDIO->RESP4;
    }
    return SD_OK;//卡初始化成功
}
```

SD_InitializeCards 函数主要发送 CMD2 和 CMD3，获得 CID 寄存器内容和 SD 卡的相对地址（RCA），并通过 CMD9，获取 CSD 寄存器内容。到这里，实际上 SD 卡的初始化就已经完成了。

SD_Init 函数又通过调用 SD_GetCardInfo 函数，获取 SD 卡相关信息，之后调用 SD_SelectDeselect 函数，选择要操作的卡（CMD7+RCA），通过 SD_EnableWideBusOperation 函数设置 SDIO 的数据位宽为 4 位（但 MMC 卡只能支持 1 位模式！）。最后设置 SDIO_CK 时钟的频率，并设置工作模式（DMA/轮询）。

这里 SDIO_CK 时钟的设置我们要特地说一下，在 DMA 模式，如果只读，可以设置 SDIO_CK 的时钟为 24Mhz，而如果要读写的话，这个时钟设置为 18M 左右比较合适，否则出错的几率比较大，虽然我们的代码加入了防出错处理，不过这样会降低写入效率。而轮询模式，则必须设置 SDIO_CK 时钟频率不大于 18Mhz，否则不能正常读写。

接下来，我们看看 SD 卡读块函数：SD_ReadBlock，该函数用于从 SD 卡指定地址读出一个块（扇区）数据，该函数代码如下：

```
//SD 卡读取一个块
//buf:读数据缓存区(必须 4 字节对齐!!)
//addr:读取地址
//blksize:块大小
SD_Error SD_ReadBlock(u8 *buf,u32 addr,u16 blksize)
{
    SD_Error errorstatus=SD_OK;
    u8 power;
    u32 count=0,*tempbuff=(u32*)buf;//转换为 u32 指针
    u32 timeout=0;
    if(NULL==buf)return SD_INVALID_PARAMETER;
    SDIO->DCTRL=0x0;    //数据控制寄存器清零(关 DMA)
    if(CardType==SDIO_HIGH_CAPACITY_SD_CARD)//大容量卡
    {
        blksize=512;
        addr>>=9;
    }
    SDIO_Send_Data_Cfg(SD_DATATIMEOUT,0,0,0);   //清除 DPSM 状态机配置
    if(SDIO->RESP1&SD_CARD_LOCKED)return SD_LOCK_UNLOCK_FAILED;
    //卡锁了
    if((blksize>0)&&(blksize<=2048)&&((blksize&(blksize-1))==0))
    {
        power=convert_from_bytes_to_power_of_two(blksize);
        SDIO_Send_Cmd(SD_CMD_SET_BLOCKLEN,1,blksize);
        //发送 CMD16+设置数据长度为 blksize,短响应
        errorstatus=CmdResp1Error(SD_CMD_SET_BLOCKLEN);//等待 R1 响应
        if(errorstatus!=SD_OK)return errorstatus;           //响应错误
    }else return SD_INVALID_PARAMETER;
    SDIO_Send_Data_Cfg(SD_DATATIMEOUT,blksize,power,1);  //blksize,卡到控制器
    SDIO_Send_Cmd(SD_CMD_READ_SINGLE_BLOCK,1,addr);
    //发送 CMD17+从 addr 地址出读取数据,短响应
```

```
errorstatus=CmdResp1Error(SD_CMD_READ_SINGLE_BLOCK);//等待 R1 响应
if(errorstatus!=SD_OK)return errorstatus;                    //响应错误
if(DeviceMode==SD_POLLING_MODE)                              //查询模式,轮询数据
{
    while(!(SDIO->STA&((1<<5)|(1<<1)|(1<<3)|(1<<10)|(1<<9))))
    //无上溢/CRC/超时/完成(标志)/起始位错误
    {
        if(SDIO->STA&(1<<15))                                //接收区半满,表示至少存了 8 个字
        {
            for(count=0;count<8;count++)                     //循环读取数据
            {
                *(tempbuff+count)=SDIO->FIFO;
            }
            tempbuff+=8;
        }
    }
    if(SDIO->STA&(1<<3))        //数据超时错误
    {
        SDIO->ICR|=1<<3;                                     //清错误标志
        return SD_DATA_TIMEOUT;
    }else if(SDIO->STA&(1<<1))  //数据块 CRC 错误
    {
        SDIO->ICR|=1<<1;                                     //清错误标志
        return SD_DATA_CRC_FAIL;
    }else if(SDIO->STA&(1<<5))  //接收 fifo 上溢错误
    {
        SDIO->ICR|=1<<5;                                     //清错误标志
        return SD_RX_OVERRUN;
    }else if(SDIO->STA&(1<<9))  //接收起始位错误
    {
        SDIO->ICR|=1<<9;                                     //清错误标志
        return SD_START_BIT_ERR;
    }
    while(SDIO->STA&(1<<21))//FIFO 里面,还存在可用数据
    {
        *tempbuff=SDIO->FIFO;   //循环读取数据
        tempbuff++;
    }
    SDIO->ICR=0X5FF;                                         //清除所有标记
}else if(DeviceMode==SD_DMA_MODE)
{
    TransferError=SD_OK;
    StopCondition=0;                //单块读,不需要发送停止传输指令
    TransferEnd=0;                                           //传输结束标置位，在中断服务置 1
```

```
            SDIO->MASK|=(1<<1)|(1<<3)|(1<<8)|(1<<5)|(1<<9);//配置需要的中断
            SDIO->DCTRL|=1<<3;              //SDIO DMA 使能
            SD_DMA_Config((u32*)buf,blksize,0);
            timeout=SDIO_DATATIMEOUT;
            while(((DMA2->ISR&0X2000)==RESET)&&(TransferEnd==0)&&(
            TransferError==SD_OK)&&timeout)timeout--;  //等待传输完成
            if(timeout==0)return SD_DATA_TIMEOUT;      //超时
            if(TransferError!=SD_OK)errorstatus=TransferError;
        }
        return errorstatus;
    }
```

该函数先发送 CMD16，用于设置块大小，然后配置 SDIO 控制器读数据的长度，这里我们用到函数 convert_from_bytes_to_power_of_two 求出 blksize 以 2 为底的指数，用于 SDIO 读数据长度设置。然后发送 CMD17（带地址参数 addr），从指定地址读取一块数据。最后，根据我们所设置的模式（查询模式/DMA 模式），从 SDIO_FIFO 读出数据。

SD_ReadBlock 函数，我们就介绍到这里，另外，还有三个底层读写函数：SD_ReadMultiBlocks，用于多块读；SD_WriteBlock，用于单块写；SD_WriteMultiBlocks，用于多块写；**再次提醒：无论哪个函数，其数据 buf 的地址都必须是 4 字节对齐的！**限于篇幅，余下 3 个函数我们就不一一介绍了，大家可以参本实验考源代码。关于控制命令，如果有不了解的，请参考《SD 卡 2.0 协议.pdf》里面都有非常详细的介绍。

最后，我们来看看 SDIO 与文件系统的两个接口函数：SD_ReadDisk 和 SD_WriteDisk，这两个函数的代码如下：

```
//读 SD 卡
//buf:读数据缓存区
//sector:扇区地址
//cnt:扇区个数
//返回值:错误状态;0,正常;其他,错误代码;
u8 SD_ReadDisk (u8*buf,u32 sector,u8 cnt)
{
    u8 sta=SD_OK; u8 n;
    if(CardType!=SDIO_STD_CAPACITY_SD_CARD_V1_1)sector<<=9;
    if((u32)buf%4!=0)
    {
        for(n=0;n<cnt;n++)
        {
            sta=SD_ReadBlock(SDIO_DATA_BUFFER,sector, +512*n512);
            //单个 sector 读操作
            memcpy(buf,SDIO_DATA_BUFFER,512);
            buf+=512;
        }
    }else
    {
        if(cnt==1)sta=SD_ReadBlock(buf,sector,512);      //单个 sector 的读操作
        else sta=SD_ReadMultiBlocks(buf,sector,512,cnt); //多个 sector
```

```
        }
        return sta;
    }
    //写 SD 卡
    //buf:写数据缓存区
    //sector:扇区地址
    //cnt:扇区个数
    //返回值:错误状态;0,正常;其他,错误代码;
    u8 SD_WriteDisk(u8*buf,u32 sector,u8 cnt)
    {
        u8 sta=SD_OK; u8 n;
        if(CardType!=SDIO_STD_CAPACITY_SD_CARD_V1_1)sector<<=9;
        if((u32)buf%4!=0)
        {
            for(n=0;n<cnt;n++)
            {
                memcpy(SDIO_DATA_BUFFER,buf,512);
                sta=SD_WriteBlock(SDIO_DATA_BUFFER,sector+512*n,512);
                //单个 sector 写操作
                buf+=512;
            }
        }else
        {
            if(cnt==1)sta=SD_WriteBlock(buf,sector,512);          //单个 sector 的写操作
            else sta=SD_WriteMultiBlocks(buf,sector,512,cnt);     //多个 sector
        }
        return sta;
    }
```

这两个函数在 fatfs 的 diskio.c 里面被调用，其中 SD_ReadDisk 用于读数据，通过调用 SD_ReadBlock 和 SD_ReadMultiBlocks 实现。SD_WriteDisk 用于写数据，通过调用 SD_WriteBlock 和 SD_WriteMultiBlocks 实现。注意，因为 fatfs 提供给 SD_ReadDisk 或者 SD_WriteDisk 的数据缓存区地址不一定是 4 字节对齐的，所以我们在这两个函数里面做了 4 字节对齐判断，如果不是 4 字节对齐的，则通过一个 4 字节对齐缓存（SDIO_DATA_BUFFER）作为数据过度，以确保传递给底层读写函数的 buf 是 4 字节对齐的。

sdio_sdcard.c 的内容，我们就介绍到这里，接下来，我们看看 test.c 里面的内容，因为本实验移植自战舰 STM32 开发板标准例程：实验 42 图片显示实验，所以 main 函数内容基本一样，不过我们在 test.c 里面添加了 2 个函数，如下：

```
    //fname:文件名
    //strx:要写入的字符串
    //repeat:写入的次数（文件大小=strx 长度*repeat 次数）
    void testwrite(u8 *fname,u8* strx,u32 repeat)
    {
        FIL* f_test;
        u8 *databuf;                                //数组
```

```
    u8 strxlen=strlen((const char*)strx);     //字符串长度
    u32 bufrpt=2048/strxlen;                 //databuf 每次能存储 strx 的条数
    u32 rpcnt=0;
    u8 *ptr; u8 res,i;
    f_test=(FIL*)mymalloc(SRAMIN,sizeof(FIL));
    databuf=mymalloc(SRAMIN,2048);
    if(f_test==NULL||databuf==NULL)
    {
        myfree(SRAMIN,f_test);
        myfree(SRAMIN,databuf);
        return ;
    }
    res=f_open(f_test,(const TCHAR*)fname,FA_CREATE_NEW|FA_WRITE);//创建文件
    if(res==FR_OK)//创建成功
    {
        ptr=databuf;
        for(rpcnt=0;rpcnt<repeat;)
        {
            for(i=0;i<bufrpt;)//将 databuf 尽量填满
            {
                strncpy((char*)ptr,(char*)strx,strxlen);
                ptr+=strxlen;
                rpcnt++;i++;
                if(rpcnt>=repeat)break;
            }
            printf("\r\nreptx:%02d   strxlen:%d",rpcnt,strxlen);    //打印当前已写数目
            res=f_write(f_test,databuf,(u32)i*strxlen,&bw);//一次写入 i*strxlen 个字节
            if(res||bw!=bufrpt*strxlen)break;                    //写入出错/结束
            ptr=databuf;
        }
        f_close(f_test);
        printf("\r\nfile size:%d\r\n",rpcnt*strxlen);//总文件大小
    }
    myfree(SRAMIN,f_test); myfree(SRAMIN,databuf);
}
//通过串口打印 SD 卡相关信息
void show_sdcard_info(void)
{
    switch(SDCardInfo.CardType)
    {
        case SDIO_STD_CAPACITY_SD_CARD_V1_1:
            printf("Card Type:SDSC V1.1\r\n");break;
        case SDIO_STD_CAPACITY_SD_CARD_V2_0:
            printf("Card Type:SDSC V2.0\r\n");break;
```

```
            case SDIO_HIGH_CAPACITY_SD_CARD:
                printf("Card Type:SDHC V2.0\r\n");break;
            case SDIO_MULTIMEDIA_CARD:
                printf("Card Type:MMC Card\r\n");break;
        }
        printf("Card ManufacturerID:%d\r\n",SDCardInfo.SD_cid.ManufacturerID);
        //制造商 ID
        printf("Card RCA:%d\r\n",SDCardInfo.RCA);              //卡相对地址
        printf("Card Capacity:%d Mb\r\n",(u32)(SDCardInfo.CardCapacity>>20))//显示容量
        printf("Card BlockSize:%d\r\n\r\n",SDCardInfo.CardBlockSize);//显示块大小
    }
```

其中，testwrite 函数用于测试 SDIO 的写数据是否正常，通过 usmart 控制，我们通过串口调用该函数，即可实现通过 SDIO 写 SD 卡，该函数有 3 个参数，fname：为要创建的文件名，例如"0:\test001.txt"，将在 SD 卡根目录创建一个 test001.txt 的文件。strx，为要写入的字符串，repeat，为要写入的次数（即将字符串 strx 重复写入的次数），这样创建后的文件大小为：repeat*（strx 的长度）。当文件大小，小于 1024 字节的时候，可以测试 SDIO 的单块写函数，当文件大小大于等于 1024 字节的时候，可以测试 SDIO 的多块写函数。

show_sdcard_info 函数，则比较简单，只是将 SD 卡的类型、制造商 ID、RCA 地址、容量和块大小等信息通过串口打印出来，方便大家观察。

其他代码，我们就不作介绍了，请大家参考本实验源码，接下来我们来验证代码。

4、验证

在代码编译成功之后，我们通过下载代码到战舰 STM32 开发板上（SDIO 跳线已经设置好，并且插上了 SD 卡），启动界面如图 4.1 所示：

图 4.1 SDIO 实验启动界面

此时，我们可以在串口调试助手看到 SD 卡的相关信息，如图 4.2 所示：

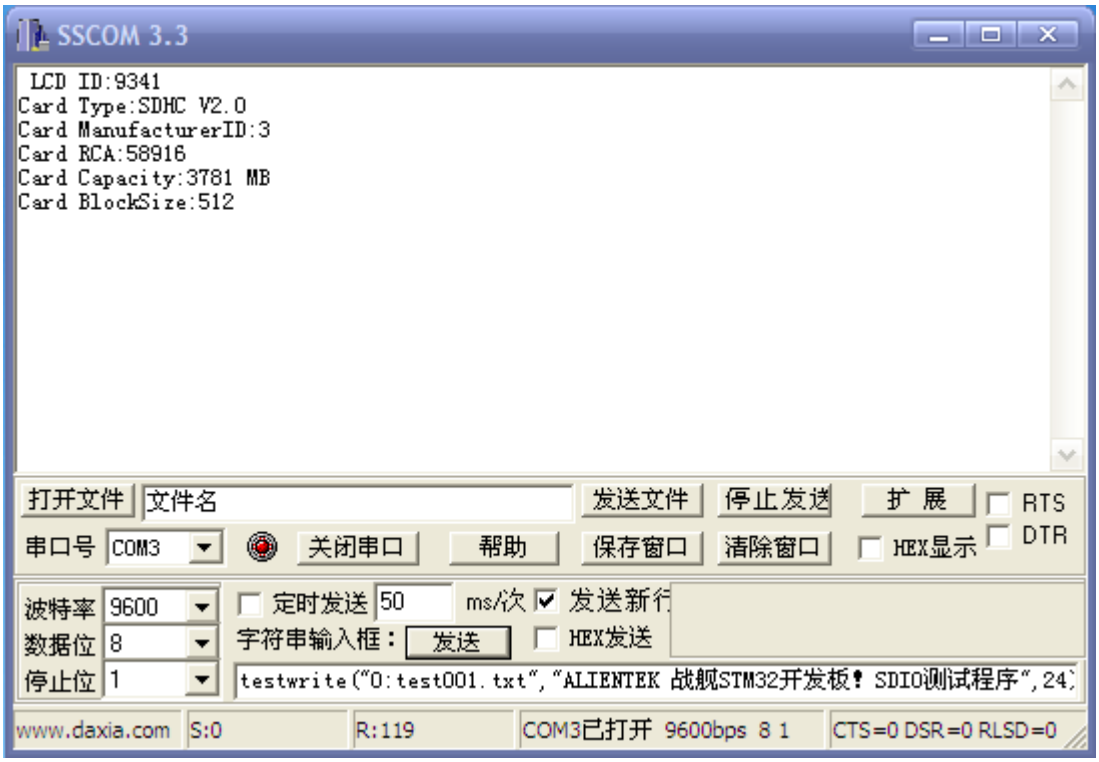


图 4.2 串口显示 SD 卡相关信息

从图中可以看到，当前 SD 卡是 SDHC 2.0 卡，制造厂商 ID 为 3，RCA 地址为 58916，容量为 3781MB，块大小为 512 字节。

然后我们通过串口，调用 testwrite 函数，实现 SD 卡的写数据测试，如图 4.3 所示：

图 4.2 串口调用 testwrite 函数，实现 SDIO 写测试

如图 4.2 所示，我们创建了两个文件：test001.txt 和 test002.txt。然后我们在电脑上打开 SD 卡，看看是不是有这两个文件，并且看看文件大小和里面的内容是否和我们设计的一样？

正点原子@ALIENTEK
2013-3-10
开源电子网：www.openedv.com
星翼电子官网：www.alientek.com