第四十三章　SD 卡实验

很多单片机系统都需要大容量存储设备，以存储数据。目前常用的有 U 盘，FLASH 芯片，SD 卡等。他们各有优点，综合比较，最适合单片机系统的莫过于 SD 卡了，它不仅容量可以做到很大（32GB 以上），支持 SPI/SDIO 驱动，而且有多种体积的尺寸可供选择（标准的 SD 卡尺寸，以及 TF 卡尺寸等），能满足不同应用的要求。

只需要少数几个 IO 口即可外扩一个高达 32GB 以上的外部存储器，容量从几十 M 到几十 G 选择尺度很大，更换也很方便，编程也简单，是单片机大容量外部存储器的首选。

ALIENTKE 探索者 STM32F4 开发板自带了标准的 SD 卡接口，使用 STM32F4 自带的 SDIO 接口驱动，4 位模式，最高通信速度可达 48Mhz（分频器旁路时），最高每秒可传输数据 24M 字节，对于一般应用足够了。在本章中，我们将向大家介绍，如何在 ALIENTEK 探索者 STM32F4 开发板上实现 SD 卡的读取。本章分为如下几个部分：

43.1 SDIO 接口简介
43.2 硬件设计
43.3 软件设计
43.4 下载验证

43.1 SDIO 简介

ALIENTEK 探索者 STM32F4 开发板自带 SDIO 接口，本节，我们将简单介绍 STM32F4 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。

43.1.1 SDIO 主要功能及框图

STM32F4 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（分频器旁路时）。
- 数据和命令输出使能信号，用于控制外部双向驱动器。

STM32F4 的 SDIO 控制器包含 2 个部分：SDIO 适配器模块和 APB2 总线接口，其功能框图如图 43.1.1.1 所示：

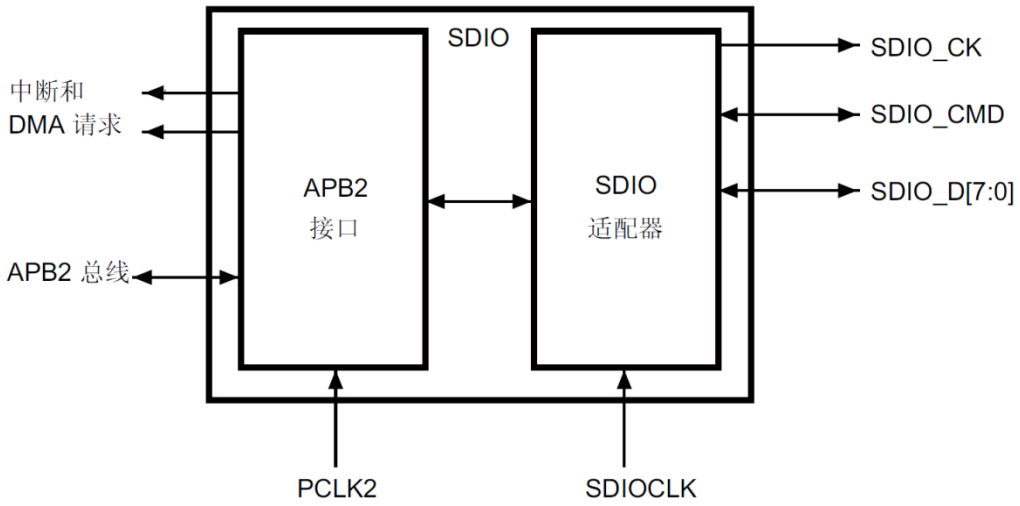


图 43.1.1.1 STM32F4 的 SDIO 控制器功能框图

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)

43.1.2 SDIO 的时钟

从图 43.1.1.1 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，来自 PLL48CK，一般为 48Mhz，并用于产生 SDIO_CK 时钟。

APB2 总线接口时钟（PCLK2）：该时钟用于驱动 SDIO 的 APB2 总线接口，其频率为 HCLK/2，一般为 84Mhz。

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（时钟分频器不旁路时）为：

SDIO_CK=SDIOCLK/(2+CLKDIV)

其中，SDIOCLK 为 PLL48CK，一般是 48Mhz，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）。注意，以上公式，是时钟分频器不旁路时的计算公式，当时钟分频器旁路时，SDIO_CK 直接等于 SDIOCLK。

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）。

43.1.3 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 43.1.3.1 所示：

位的位置	宽度	值	说明
47	1	0	起始位
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 43.1.3.1 SDIO 命令格式

所有的命令都是由 STM32F4 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。STM32F4 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 43.1.3.2 所示：

位的位置	宽度	值	说明
47	1	0	起始位
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 43.1.3.2 SDIO 命令格式

长响应的格式如表 43.1.3.3 所示：

位的位置	宽度	值	说明
135	1	0	起始位
134	1	0	传输位
[133:128]	6	111111	保留
[127:1]	127	-	CID 或 CSD（包括内部 CRC7）
0	1	1	结束位

表 43.1.3.3 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 43.1.3.4 所示：

位的位置	宽度（位）	值	说明
47	1	0	起始位
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 43.1.3.4 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《STM32F4xx 中文参考手册》第 28 章。

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 43.1.3.1 所示：

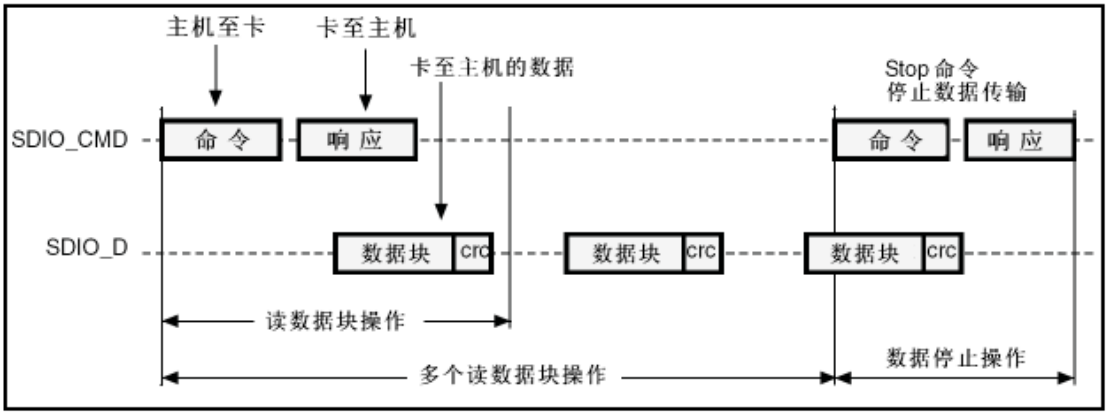


图 43.1.3.1 SDIO（多）数据块读操作

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 43.1.3.2 所示：

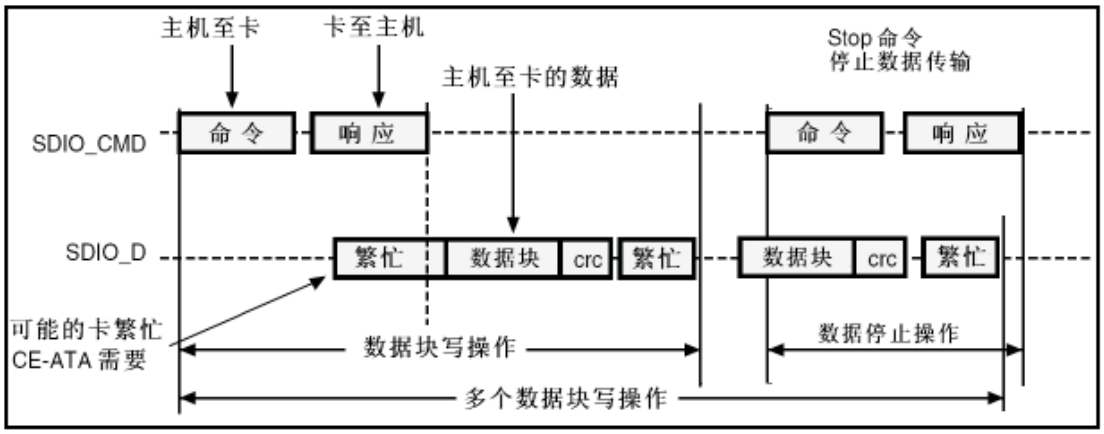


图 43.1.3.2 SDIO（多）数据块写操作

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。

43.1.4 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 43.1.4.1 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
Reserved																														PWRCTRL	
																														rw	rw

位 31:2 保留，必须保持复位值

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：掉电：停止为卡提供时钟。 10：保留，上电
01：保留 11：通电：为卡提供时钟。

图 43.1.4.1 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 43.1.4.2 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
Reserved																	HWFC_EN	NEGEDGE	WIDBUS		BYPASS	PWRSAV	CLKEN	CLKDIV							
																	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

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 位宽总线模式：使用 SDIO_D[3:0]
10：8 位宽总线模式：使用 SDIO_D[7:0]

位 10 **BYPASS**：时钟分频器旁路使能位 (Clock divider bypass enable bit)
0：禁止旁路：在驱动 SDIO_CK 输出信号前，根据 CLKDIV 值对 SDIOCLK 进行分频。
1：使能旁路：SDIOCLK 直接驱动 SDIO_CK 输出信号。

位 8 **CLKEN**：时钟使能位 (Clock enable bit)
0：禁止 SDIO_CK
1：使能 SDIO_CK

位 7:0 **CLKDIV**：时钟分频系数 (Clock divide factor)
该字段定义输入时钟 (SDIOCLK) 与输出时钟 (SDIO_CK) 之间的分频系数：
SDIO_CK 频率 = SDIOCLK / [CLKDIV + 2]。

图 43.1.4.2 SDIO_CLKCR 寄存器位定义

上图仅列出了部分我们要用到的位设置，WIDBUS 用于设置 SDIO 总线位宽，正常使用的时候，设置为 1，即 4 位宽度。BYPASS 用于设置分频器是否旁路，我们一般要使用分频器，所以这里设置为 0，禁止旁路。CLKEN 则用于设置是否使能 SDIO_CK，我们设置为 1。最后，CLKDIV，则用于控制 SDIO_CK 的分频，一般设置为 0，即可得到 24Mhz 的 SDIO_CK 频率。

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，但只有低 6 位有效，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 43.1.4.1 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	未使用	卡状态 [95:64]
SDIO_RESP3	未使用	卡状态 [63:32]
SDIO_RESP4	未使用	卡状态 [31:1]0b

表 43.1.4.1 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 43.1.4.3 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
Reserved																	CE-ATACMD	nIEN	ENCMDcompl	SDIOSuspend	CPSMEN	WAITPEND	WAITINT	WAITRESP		CMDINDEX						
																	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 10 **CPSMEN**：命令路径状态机 (CPSM) 使能位 (Command path state machine (CPSM) Enable bit)
如果此位置 1，则使能 CPSM。

位 7:6 WAITRESP：等待响应位 (Wait for response bits)
这些位用于配置 CPSM 是否等待响应，如果等待，将等待哪种类型的响应。
00：无响应，但 CMDSENT 标志除外
01：短响应，但 CMDREND 或 CCRCFAIL 标志除外
10：无响应，但 CMDSENT 标志除外
11：长响应，但 CMDREND 或 CCRCFAIL 标志除外

位 5:0 **CMDINDEX**：命令索引 (Command index)
命令索引作为命令消息的一部分发送给卡。

图 43.1.4.3 SDIO_CMD 寄存器位定义

图中只列出了部分位的描述，其中低 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《STM32F4xx 中文参考手册》第 776 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《STM32F4xx 中文参考手册》第 780 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 25 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 43.1.4.4 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
Reserved																				SDIOEN	RWMOD	RWSTOP	RWSTART	DBLOCKSIZE				DMAEN	DTMODE	DTDIR	DTEN
																				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 11 **SDIOEN**：SD I/O 使能功能 (SD I/O enable functions)
如果将该位置 1，则 DPSM 执行特定于 SD I/O 卡的操作。

位 10 **RWMOD**：读取等待模式 (Read wait mode)
0：通过停止 SDIO_D2 进行读取等待控制
1：使用 SDIO_CK 进行读取等待控制

位 9 **RWSTOP**：读取等待停止 (Read wait stop)
0：如果将 RWSTART 位置 1，则读取等待正在进行中
1：如果将 RWSTART 位置 1，则使能读取等待停止

位 8 **RWSTART**：读取等待开始 (Read wait start)
如果将该位置 1，则读取等待操作开始。

位 7:4 **DBLOCKSIZE**：数据块大小 (Data block size)
定义在选择了块数据传输模式时数据块的长度：
0000：（十进制数 0）块长度 = 2^0 = 1 字节
0001：（十进制数 1）块长度 = 2^1 = 2 字节
0010：（十进制数 2）块长度 = 2^2 = 4 字节
0011：（十进制数 3）块长度 = 2^3 = 8 字节
0100：（十进制数 4）块长度 = 2^4 = 16 字节
0101：（十进制数 5）块长度 = 2^5 = 32 字节
0110：（十进制数 6）块长度 = 2^6 = 64 字节
0111：（十进制数 7）块长度 = 2^7 = 128 字节
1000：（十进制数 8）块长度 = 2^8 = 256 字节
1001：（十进制数 9）块长度 = 2^9 = 512 字节
1010：（十进制数 10）块长度 = 2^{10} = 1024 字节
1011：（十进制数 11）块长度 = 2^{11} = 2048 字节
1100：（十进制数 12）块长度 = 2^{12} = 4096 字节
1101：（十进制数 13）块长度 = 2^{13} = 8192 字节
1110：（十进制数 14）块长度 = 2^{14} = 16384 字节
1111：（十进制数 15）保留

位 3 **DMAEN**：DMA 使能位 (DMA enable bit)
0：禁止 DMA。
1：使能 DMA。

位 2 **DTMODE**：数据传输模式选择 (Data transfer mode selection)
0：块数据传输
1：流或 SDIO 多字节数据传输

位 1 **DTDIR**：数据传输方向选择 (Data transfer direction selection)
0：从控制器到卡。
1：从卡到控制器。

位 0 **DTEN**：数据传输使能位 (Data transfer enabled bit)
如果 1 写入到 DTEN 位，则数据传输开始。根据方向位 DTDIR，如果在传输开始时立即将 RW 置 1开始，则 DPSM 变为 Wait_S 状态、Wait_R 状态或读取等待状态。在数据传输结束后不需要将使能位清零，但必须更新 SDIO_DCTRL 以使能新的数据传输

图 43.1.4.4 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 43.1.4.5 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
Reserved								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 23 **CEATAEND**：针对 CMD61 收到了 CE-ATA 命令完成信号

位 22 **SDIOIT**：收到了 SDIO 中断 (SDIO interrupt received)

位 21 **RXDAVL**：接收 FIFO 中有数据可用 (Data available in receive FIFO)

位 20 **TXDAVL**：传输 FIFO 中有数据可用 (Data available in transmit FIFO)

位 19 **RXFIFOE**：接收 FIFO 为空 (Receive FIFO empty)

位 18 **TXFIFOE**：发送 FIFO 为空 (Transmit FIFO empty)
如果使能了硬件流控制，则 TXFIFOE 信号在 FIFO 包含 2 个字时激活。

位 17 **RXFIFOF**：接收 FIFO 已满 (Receive FIFO full)
如果使能了硬件流控制，则 RXFIFOF 信号在 FIFO 差 2 个字便变满之前激活。

位 16 **TXFIFOF**：传输 FIFO 已满 (Transmit FIFO full)

位 15 **RXFIFOHF**：接收 FIFO 半满：FIFO 中至少有 8 个字

位 14 **TXFIFOHE**：传输 FIFO 半空：至少可以写入 8 个字到 FIFO

位 13 **RXACT**：数据接收正在进行中 (Data receive in progress)

位 12 **TXACT**：数据传输正在进行中 (Data transmit in progress)

位 11 **CMDACT**：命令传输正在进行中 (Command transfer in progress)

位 10 **DBCKEND**：已发送/接收数据块（CRC 校验通过）

位 9 **STBITERR**：在宽总线模式下，并非在所有数据信号上都检测到了起始位

位 8 **DATAEND**：数据结束（数据计数器 SDIDCOUNT 为零）

位 7 **CMDSENT**：命令已发送（不需要响应）(Command sent (no response required))

位 6 **CMDREND**：已接收命令响应（CRC 校验通过）

位 5 **RXOVERR**：收到了 FIFO 上溢错误 (Received FIFO overrun error)

位 4 **TXUNDERR**：传输 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)
命令超时周期为固定值 64 个 SDIO_CK 时钟周期。

位 1 **DCRCFAIL**：已发送/接收数据块（CRC 校验失败）

位 0 **CCRCFAIL**：已接收命令响应（CRC 校验失败）

图 43.1.4.5 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《STM32F4xx 中文参考手册》第 28 章相关章节。

43.1.5 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 43.1.5.1 所示：

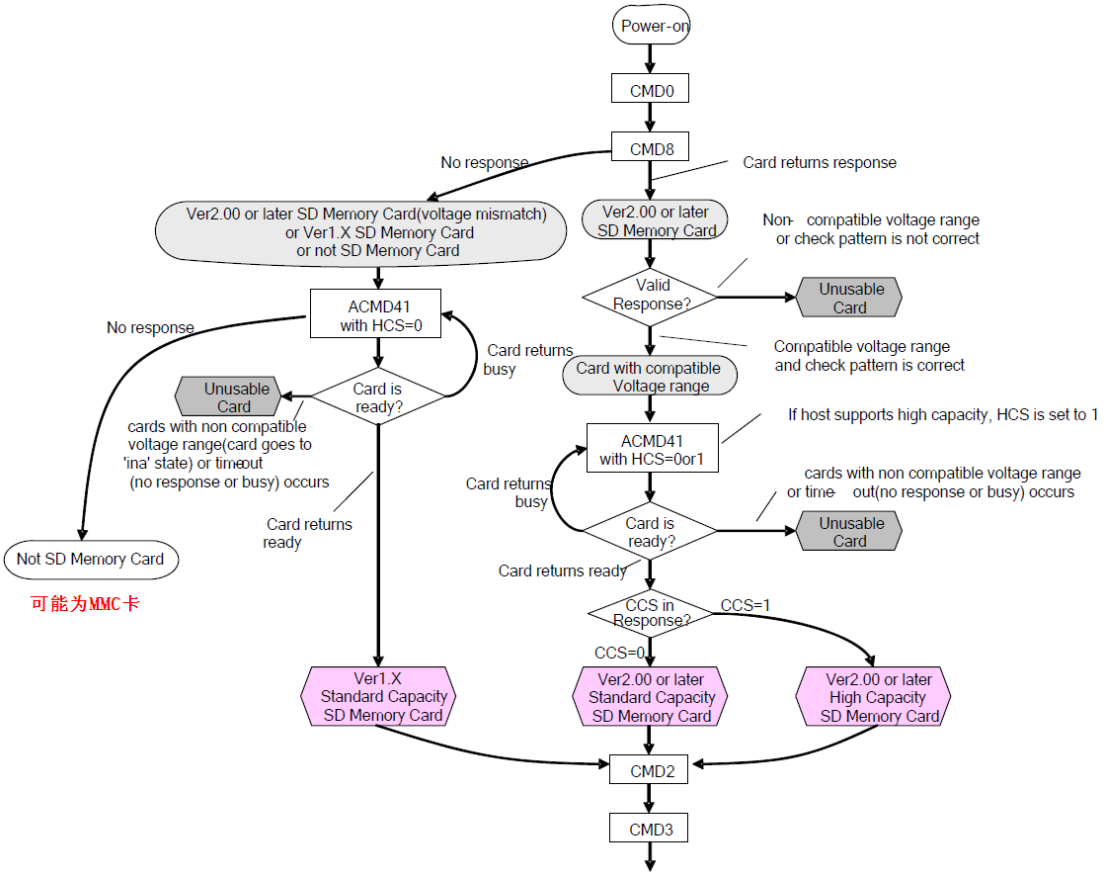


图 43.1.5.1 SD 卡初始化流程

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 43.1.5.1 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 43.1.5.1 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 43.1.5.2 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 43.1.5.2 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 43.1.5.3 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 43.1.5.3 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 43.1.5.4 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 43.1.5.4 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 43.1.5.1 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 43.1.5.5 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 43.1.5.5 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。

43.2 硬件设计

本章实验功能简介：开机的时候先初始化 SD 卡，如果 SD 卡初始化完成，则提示 LCD 初始化成功。按下 KEY0，读取 SD 卡扇区 0 的数据，然后通过串口发送到电脑。如果没初始化

通过，则在 LCD 上提示初始化失败。同样用 DS0 来指示程序正在运行。

本实验用到的硬件资源有：

1） 指示灯 DS0

2） KEY0 按键

3） 串口

4） TFTLCD 模块

5） SD 卡

前面四部分，在之前的实例已经介绍过了，这里我们介绍一下探索者 STM32F4 开发板板载的 SD 卡接口和 STM32F4 的连接关系，如图 43.2.1 所示：

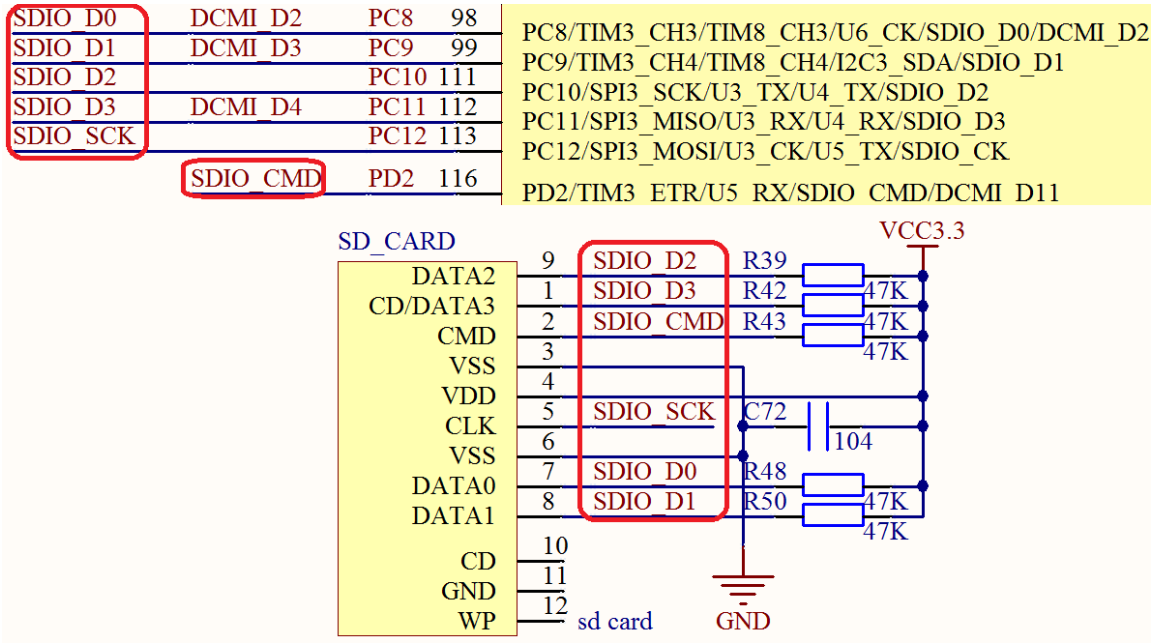


图43.2.1 SD卡接口与STM32F4连接原理图

探索者STM32F4开发板的SD卡座（SD_CARD），在PCB背面，SD卡座与STM32F4的连接在开发板上是直接连接在一起的，硬件上不需要任何改动。

43.3 软件设计

打开本章实验工程可以看到，我们不但增加了固件库 SDIO 支持文件 stm32f4xx_sdio.c 以及头文件 stm32f4xx_sdio.h，同时，我们还新增了 SD 卡的 SDIO 支持文件 sdio_sdcard.c 以及头文件 sdio_sdcard.h。由于 sdio_sdcard.c 里面代码比较多，由于篇幅限制，我们不贴出所有代码，仅介绍几个重要的函数，第一个是 SD_Init 函数，该函数源码如下：

```
//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;
    GPIO_InitTypeDef  GPIO_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC|RCC_AHB1Periph_GPIOD|
        RCC_AHB1Periph_DMA2, ENABLE);//使能 GPIOC,GPIOD DMA2 时钟
```

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SDIO, ENABLE);//SDIO 时钟使能

RCC_APB2PeriphResetCmd(RCC_APB2Periph_SDIO, ENABLE);//SDIO 复位

GPIO_InitStructure.GPIO_Pin =GPIO_Pin_8|GPIO_Pin_9|GPIO_Pin_10|
                    GPIO_Pin_11|GPIO_Pin_12//PC8,9,10,11,12 复用功能输出
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用功能
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;//100M
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;//上拉
GPIO_Init(GPIOC, &GPIO_InitStructure);// PC8,9,10,11,12 复用功能输出

GPIO_InitStructure.GPIO_Pin =GPIO_Pin_2;
GPIO_Init(GPIOD, &GPIO_InitStructure);//PD2 复用功能输出

//引脚复用映射设置
GPIO_PinAFConfig(GPIOC,GPIO_PinSource8,GPIO_AF_SDIO); //PC8,AF12
GPIO_PinAFConfig(GPIOC,GPIO_PinSource9,GPIO_AF_SDIO);
GPIO_PinAFConfig(GPIOC,GPIO_PinSource10,GPIO_AF_SDIO);
GPIO_PinAFConfig(GPIOC,GPIO_PinSource11,GPIO_AF_SDIO);
GPIO_PinAFConfig(GPIOC,GPIO_PinSource12,GPIO_AF_SDIO);
GPIO_PinAFConfig(GPIOD,GPIO_PinSource2,GPIO_AF_SDIO);

RCC_APB2PeriphResetCmd(RCC_APB2Periph_SDIO, DISABLE);//SDIO 结束复位
SDIO_Register_Deinit();//SDIO 外设寄存器设置为默认值

NVIC_InitStructure.NVIC_IRQChannel = SDIO_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=0;//抢占优先级 3
NVIC_InitStructure.NVIC_IRQChannelSubPriority =0;          //响应优先级 3
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;            //IRQ 通道使能
NVIC_Init(&NVIC_InitStructure);      //根据指定的参数初始化 VIC 寄存器、

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);     //获取卡信息
if(errorstatus==SD_OK)
     errorstatus=SD_SelectDeselect((u32)(SDCardInfo.RCA<<16));//选中 SD 卡
if(errorstatus==SD_OK)
```

```
            errorstatus=SD_EnableWideBusOperation(SDIO_BusWide_4b);
                                    //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))
         {
            //设置时钟频率,SDIO 时钟计算公式:SDIO_CK 时钟=SDIOCLK/[clkdiv+2];
            //其中,SDIOCLK 固定为 48Mhz
             SDIO_Clock_Set(SDIO_TRANSFER_CLK_DIV);
           //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，然后开始 SD 卡的初始化流程，这个过程在 43.1.5 节有详细介绍了。首先，通过 SD_PowerON 函数（该函数我们这里不做介绍，请参考本例程源码），我们将完成 SD 卡的上电，并获得 SD 卡的类型（SDHC/SDSC/SDV1.x/MMC），然后，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_GetPowerState() == SDIO_PowerState_OFF)  //检查电源状态,确保为上电状态
      {
        errorstatus = SD_REQUEST_NOT_APPLICABLE;
        return(errorstatus);
      }

        if(SDIO_SECURE_DIGITAL_IO_CARD!=CardType)//非 SECURE_DIGITAL_IO_CARD
        {
            SDIO_CmdInitStructure.SDIO_Argument = 0x0;//发送 CMD2,取得 CID,长响应
            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);//发送 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_CmdInitStructure.SDIO_Argument = 0x00;//发送 CMD3,短响应
            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);     //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)
        {

            SDIO_CmdInitStructure.SDIO_Argument = (u32)(rca<<16);//发送 CMD3,短响应
            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);     //发送 CMD3,短响应
            errorstatus=CmdResp2Error();                  //等待 R2 响应
            if(errorstatus!=SD_OK)return errorstatus;          //响应错误
        }
    if (SDIO_SECURE_DIGITAL_IO_CARD!=CardType)//非 SECURE_DIGITAL_IO_CARD
        {
            RCA = rca;

            SDIO_CmdInitStructure.SDIO_Argument = (uint32_t)(rca << 16);
                                                //发送 CMD9+卡 RCA,取得 CSD,长响应
            SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SEND_CSD;
            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();                    //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。

随后，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）。

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,long long addr,u16 blksize)
{
    SD_Error errorstatus=SD_OK;
    u8 power;
    u32 count=0,*tempbuff=(u32*)buf;//转换为 u32 指针
    u32 timeout=SDIO_DATATIMEOUT;
    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_DataInitStructure.SDIO_DataBlockSize= 0 ;//清除 DPSM 状态机配置
    SDIO_DataInitStructure.SDIO_DataLength= 0 ;
    SDIO_DataInitStructure.SDIO_DataTimeOut=SD_DATATIMEOUT ;
    SDIO_DataInitStructure.SDIO_DPSM=SDIO_DPSM_Enable;
    SDIO_DataInitStructure.SDIO_TransferDir=SDIO_TransferDir_ToCard;
```

```
 SDIO_DataInitStructure.SDIO_TransferMode=SDIO_TransferMode_Block;
SDIO_DataConfig(&SDIO_DataInitStructure);

 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_CmdInitStructure.SDIO_Argument =    blksize;
                          //发送 CMD16+设置数据长度为 blksize,短响应
      SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_SET_BLOCKLEN;
      SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
      SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
      SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
      SDIO_SendCommand(&SDIO_CmdInitStructure);

      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_DataInitStructure.SDIO_DataBlockSize= power<<4; ;//清除 DPSM 状态机配置
 SDIO_DataInitStructure.SDIO_DataLength= blksize ;
 SDIO_DataInitStructure.SDIO_DataTimeOut=SD_DATATIMEOUT ;
 SDIO_DataInitStructure.SDIO_DPSM=SDIO_DPSM_Enable;
 SDIO_DataInitStructure.SDIO_TransferDir=SDIO_TransferDir_ToSDIO;
 SDIO_DataInitStructure.SDIO_TransferMode=SDIO_TransferMode_Block;
 SDIO_DataConfig(&SDIO_DataInitStructure);

 SDIO_CmdInitStructure.SDIO_Argument =    addr;
                                  //发送 CMD17+从 addr 地址出读取数据,短响应
 SDIO_CmdInitStructure.SDIO_CmdIndex = SD_CMD_READ_SINGLE_BLOCK;
 SDIO_CmdInitStructure.SDIO_Response = SDIO_Response_Short;
 SDIO_CmdInitStructure.SDIO_Wait = SDIO_Wait_No;
 SDIO_CmdInitStructure.SDIO_CPSM = SDIO_CPSM_Enable;
 SDIO_SendCommand(&SDIO_CmdInitStructure);

 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
 {
      INTX_DISABLE();//关闭总中断(POLLING 模式,严禁中断打断 SDIO 读写操作!!!)
      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_GetFlagStatus(SDIO_FLAG_RXFIFOHF) != RESET)
                                    //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;
                timeout=0X7FFFF;
            }else       //处理超时
            {
                if(timeout==0)return SD_DATA_TIMEOUT;
                timeout--;
            }
        }
    if(SDIO_GetFlagStatus(SDIO_FLAG_DTIMEOUT) != RESET)//数据超时错误
    {           SDIO_ClearFlag(SDIO_FLAG_DTIMEOUT); //清错误标志
                return SD_DATA_TIMEOUT;
     }else if (SDIO_GetFlagStatus(SDIO_FLAG_DCRCFAIL) != RESET)//数据块 CRC 错误
    {           SDIO_ClearFlag(SDIO_FLAG_DCRCFAIL);     //清错误标志
                return SD_DATA_CRC_FAIL;
    }else if(SDIO_GetFlagStatus(SDIO_FLAG_RXOVERR) != RESET) //接收 fifo 上溢错误
    {           SDIO_ClearFlag(SDIO_FLAG_RXOVERR);          //清错误标志
                return SD_RX_OVERRUN;
    }else if(SDIO_GetFlagStatus(SDIO_FLAG_STBITERR) != RESET) //接收起始位错误
    {
                SDIO_ClearFlag(SDIO_FLAG_STBITERR);//清错误标志
                return SD_START_BIT_ERR;
        }
    while(SDIO_GetFlagStatus(SDIO_FLAG_RXDAVL) != RESET)//FIFO 还存在可用数据
        {   *tempbuff=SDIO->FIFO;     //循环读取数据
            tempbuff++;
        }
        INTX_ENABLE();//开启总中断
        SDIO_ClearFlag(SDIO_STATIC_FLAGS);//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,DMA_DIR_PeripheralToMemory);
        while((DMA_GetFlagStatus(DMA2_Stream3,DMA_FLAG_TCIF3)==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。

该函数有两个注意的地方：1，addr 参数类型为 long long，以支持大于 4G 的卡，否则操作大于 4G 的卡，可能有问题!!! 2，轮询方式，读写 FIFO 时，严禁任何中断打断，否则可能导致读写数据出错!! 所以使用了 INTX_DISABLE 函数关闭总中断，在 FIFO 读写操作结束后，才打开总中断（INTX_ENABLE 函数设置）。

SD_ReadBlock 函数，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;
    long long lsector=sector;
    u8 n;
    if(CardType!=SDIO_STD_CAPACITY_SD_CARD_V1_1)lsector<<=9;
    if((u32)buf%4!=0)
    {
        for(n=0;n<cnt;n++)
        {
            sta=SD_ReadBlock(SDIO_DATA_BUFFER,lsector+512*n,512);//单扇区读操作
            memcpy(buf,SDIO_DATA_BUFFER,512);
            buf+=512;
        }
    }else
    {
        if(cnt==1)sta=SD_ReadBlock(buf,lsector,512);        //单个 sector 的读操作
```

```
            else sta=SD_ReadMultiBlocks(buf,lsector,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;
        long long lsector=sector;
        if(CardType!=SDIO_STD_CAPACITY_SD_CARD_V1_1)lsector<<=9;
        if((u32)buf%4!=0)
        {
            for(n=0;n<cnt;n++)
            {
                memcpy(SDIO_DATA_BUFFER,buf,512);
                sta=SD_WriteBlock(SDIO_DATA_BUFFER,lsector+512*n,512);//单扇区写
                buf+=512;
            }
        }else
        {
            if(cnt==1)sta=SD_WriteBlock(buf,lsector,512);        //单个 sector 的写操作
            else sta=SD_WriteMultiBlocks(buf,lsector,512,cnt);   //多个 sector
        }
        return sta;
    }
```

这两个函数在下一章(FATFS 实验)将会用到的,这里提前给大家介绍下,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,sdio_sdcard.h 我们就不做介绍了,请大家参考本例程源码。接下来,打开 main.c 文件,代码如下:

```
    //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
}
int main(void)
{
        u8 key; u8 t=0; u8 *buf;
        u32 sd_size;
        NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
        delay_init(168);    //初始化延时函数
        uart_init(115200);          //初始化串口波特率为 115200
        LED_Init();                             //初始化 LED
        LCD_Init();                             //LCD 初始化
        KEY_Init();                             //按键初始化
        my_mem_init(SRAMIN);      //初始化内部内存池
        my_mem_init(SRAMCCM); //初始化 CCM 内存池
        POINT_COLOR=RED;//设置字体为红色
        LCD_ShowString(30,50,200,16,16,"Explorer STM32F4");
        LCD_ShowString(30,70,200,16,16,"SD CARD TEST");
        LCD_ShowString(30,90,200,16,16,"ATOM@ALIENTEK");
        LCD_ShowString(30,110,200,16,16,"2014/5/15");
        LCD_ShowString(30,130,200,16,16,"KEY0:Read Sector 0");
        while(SD_Init())//检测不到 SD 卡
        {
            LCD_ShowString(30,150,200,16,16,"SD Card Error!"); delay_ms(500);
            LCD_ShowString(30,150,200,16,16,"Please Check! "); delay_ms(500);
            LED0=!LED0;//DS0 闪烁
        }
        show_sdcard_info();      //打印 SD 卡相关信息
        POINT_COLOR=BLUE;      //设置字体为蓝色
        //检测 SD 卡成功
        LCD_ShowString(30,150,200,16,16,"SD Card OK       ");
        LCD_ShowString(30,170,200,16,16,"SD Card Size:        MB");
```

```
        LCD_ShowNum(30+13*8,170,SDCardInfo.CardCapacity>>20,5,16);//显示 SD 卡容量
        while(1)
        {
            key=KEY_Scan(0);
            if(key==KEY0_PRES)//KEY0 按下了
            {
                buf=mymalloc(0,512);            //申请内存
                if(SD_ReadDisk(buf,0,1)==0)     //读取 0 扇区的内容
                {
                    LCD_ShowString(30,190,200,16,16,"USART1 Sending Data...");
                    printf("SECTOR 0 DATA:\r\n");
                    for(sd_size=0;sd_size<512;sd_size++)printf("%x ",buf[sd_size]);//扇区数据
                    printf("\r\nDATA ENDED\r\n");
                    LCD_ShowString(30,190,200,16,16,"USART1 Send Data Over!");
                }
                myfree(0,buf);//释放内存
            }
            t++;
            delay_ms(10);
            if(t==20) { LED0=!LED0; t=0;}
        }
    }
```

这里总共 2 个函数，show_sdcard_info 函数用于从串口输出 SD 卡相关信息。而 main 函数，则先初化 SD 卡，初始化成功，则调用 show_sdcard_info 函数，输出 SD 卡相关信息，并在 LCD 上面显示 SD 卡容量。然后进入死循环，如果有按键 KEY0 按下，则通过 SD_ReadDisk 读取 SD 卡的扇区 0（物理磁盘，扇区 0），并将数据通过串口打印出来。这里，我们对上一章学过的内存管理小试牛刀，稍微用了下，以后我们会尽量使用内存管理来设计。

43.4 下载验证

在代码编译成功之后，我们通过下载代码到 ALIENTEK 探索者 STM32F4 开发板上，可以看到 LCD 显示如图 43.4.1 所示的内容（假设 SD 卡已经插上了）：

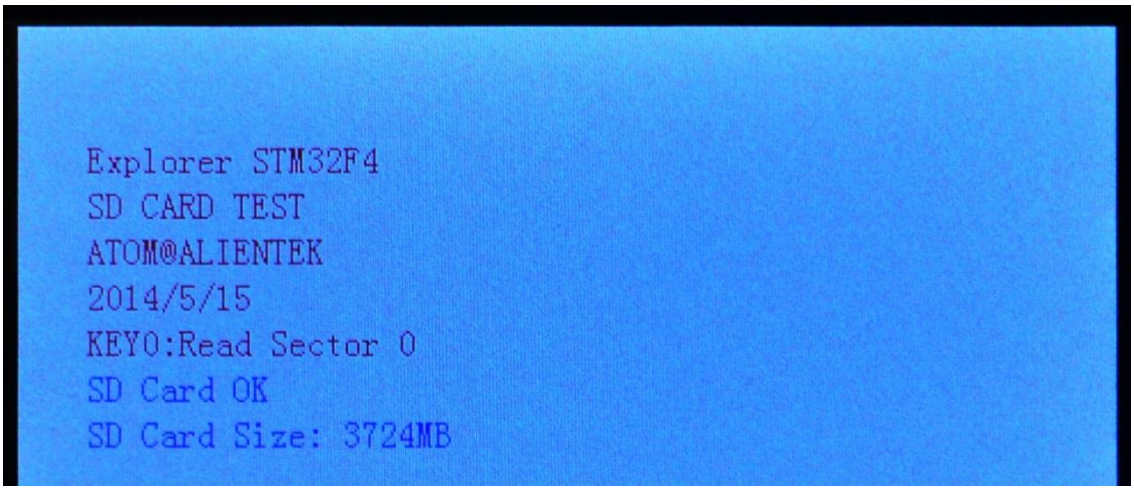
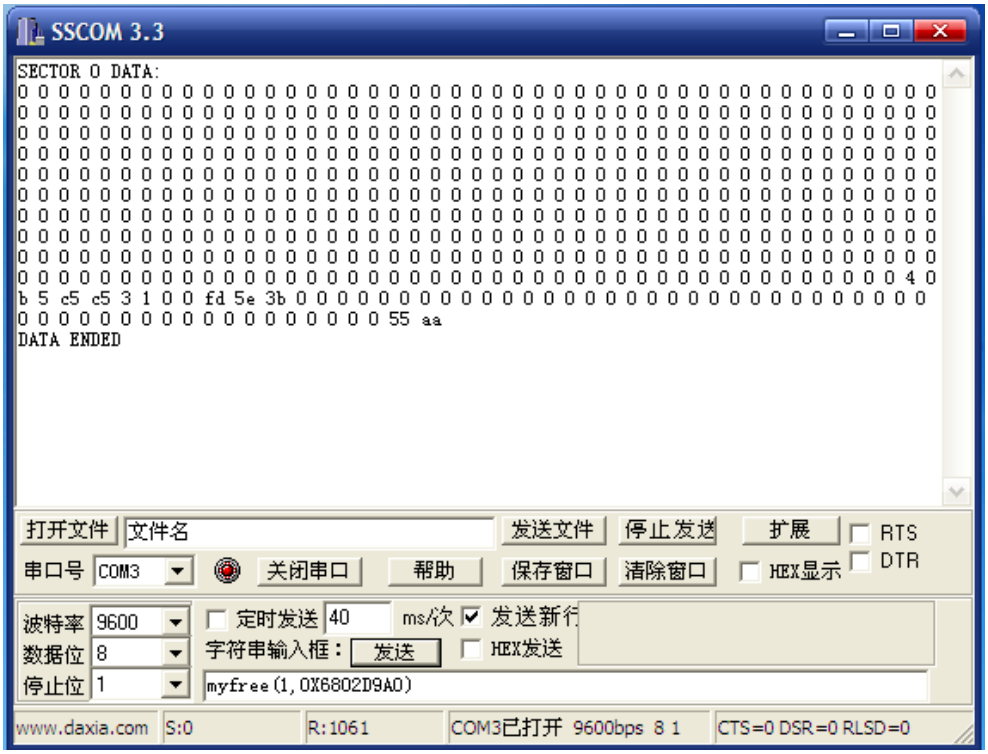


图 43.4.1 程序运行效果图

打开串口调试助手，按下 KEY0 就可以看到从开发板发回来的数据了，如图 43.4.2 所示：


SSCOM 3.3
SECTOR 0 DATA:
b 5 c5 c5 3 1 0 0 fd 5e 3b 0 0 0