

# 零死角玩转STM32

与野火同行 乐意惬意无边



原创教程，完全开源。



由浅入深，结合实操。



通俗易懂，详尽解读。



配套板子，全面玩转。



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，乐意惬意无边。

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！



### 3、MP3（支持中英文、长短文件名）

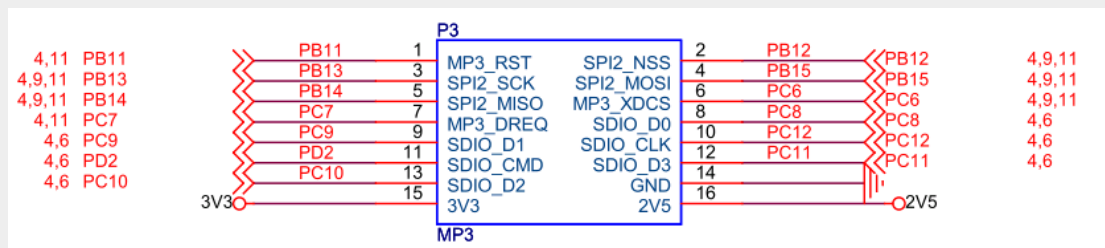
#### 3.1 实验描述及工程文件清单

实验描述	<p>将 MicroSD 卡(以文件系统 FATFS 访问)里面的 mp3 文件通过 VS1003B 解码，然后将解码后的数据送到功放 TDA1308 后通过耳机播放出来。注意：野火 M3-V1 的 MP3 模块是加了功放，野火 M3-V3 里面去掉了功放 TDA1308，因为从 VS1003 出来的模拟信号足够驱动耳机。</p> <p>（这个文档是更新版本的，配套的例程已经可以支持长中文文件名、4G 的 sd 卡，可以播放 mp3，wma，mid 和部分的 wav 格式的音频文件）</p>
硬件连接	<p>PB13-SPI2_SCK : VS1003B-SCLK</p> <p>PB14-SPI2_MISO : VS1003B-SO</p> <p>PB15-SPI2_MOSI : VS1003B-SI</p> <p>PB12-SPI2_NSS : VS1003B-XCS</p> <p>PB11 : VS1003B-XRET</p> <p>PC6 : VS1003B-XDCS</p> <p>PC7 : VS1003B-DREQ</p>
用到的库文件	<p>startup/start_stm32f10x_hd.c</p> <p>CMSIS/core_cm3.c</p> <p>CMSIS/system_stm32f10x.c</p> <p>FWlib/stm32f10x_gpio.c</p> <p>FWlib/stm32f10x_rcc.c</p> <p>FWlib/stm32f10x_usart.c</p> <p>FWlib/stm32f10x_sdio.c</p> <p>FWlib/stm32f10x_dma.c</p>

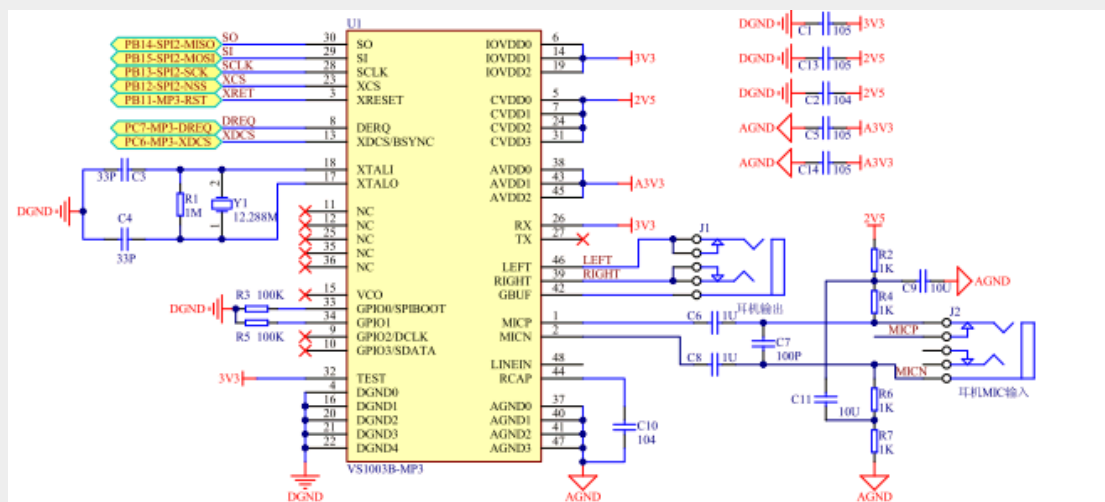


	FWlib/stm32f10x_spi.c FWlib/misc.c
用户编写的文件	USER/main.c USER/stm32f10x_it.c USER/sdio_sdcard.c USER/ff.c USER/usart1.c USER/mp3play.c USER/vs1003.c USER/SysTick.c
文件系统文件	ff9/diskio.c ff9/ff.c ff9/cc936.c

野火 STM32 开发板中 MP3 硬件接口图



MP3 模块原理图（野火 M3-V3 没有板载 MP3，需要另外选购）



解码部分采用 VS1003-MP3/WMA 音频解码器，然后将解码后的数据送

TDA1308 放大后由音频接口外播出来。注意：野火 M3-V1 的 MP3 模块是加了功放，野火 M3-V3 里面去掉了功放 TDA1308，因为从 VS1003 出来的模拟信号足够驱动耳机。

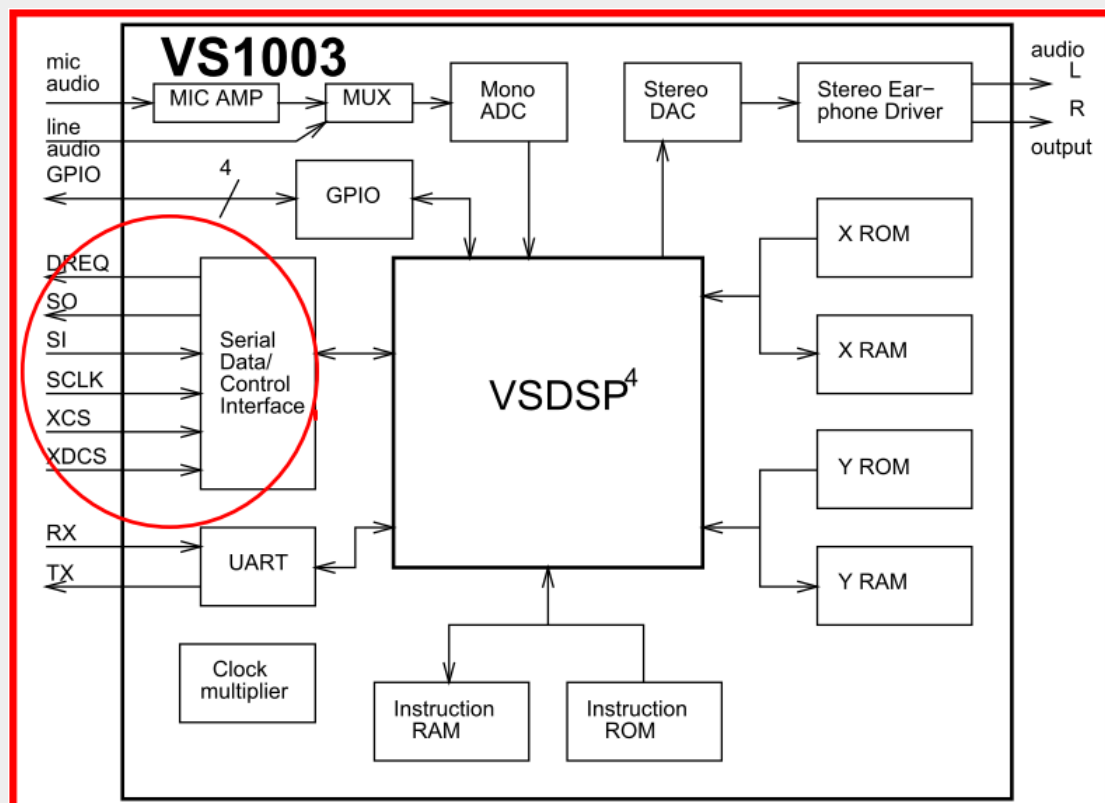
## 3.2 VS1003 & TDA1308 简介

### 3.2.1 VS1003

VS1003 是一个单片 MP3/WMA/MIDI 音频解码器和 ADPCM 编码器。它包含一个高性能，自主产权的低功耗 DSP 处理器核 VS\_DSP 4，工作数据存储器，为用户应用提供 5KB 的指令 RAM 和 0.5KB 的数据 RAM。串行的控制和数据接口，4 个常规用途的 I/O 口，一个 UART，也有一个高品质可变采样率的 ADC 和立体声 DAC，还有一个耳机放大器和地线缓冲器。

VS1003 通过一个串行接口来接收输入的比特流，它可以作为一个系统的从机。输入的比特流被解码，然后通过一个数字音量控制器到达一个 18 位过采样多位  $\epsilon$ - $\Delta$  DAC。通过串行总线控制解码器。除了基本的解码，在用户 RAM 中它还可以做其他特殊应用，例如 DSP 音效处理。

VS1003 原理框图：

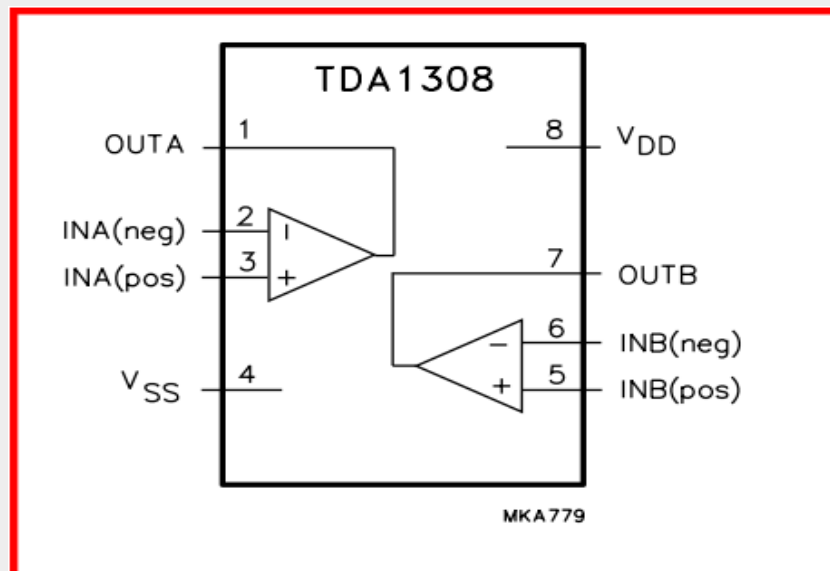


本实验中我们只用了红色圆圈中的那几个数据口，这些数据口是串行模式的，我们用到了开发板中的 **SPI2** 来控制。其中数据经 **SI** 接口进去，经解码后由 **L、R** 这两个左右声道引脚出来，因为 **VS1003** 内部集成了一个 **DA**，所以出来的数据是模拟的，可直接驱动耳机，一般不需要另外加耳机功放。



## 3.2.2 TDA1308

TDA1308 是一款双通道的立体耳机驱动器，是一款专门用于耳机驱动的功能。其原理框图如右：



有关 VS1003B 和 TDA1308 的详细应用，大家可参考官方的 datasheet，野火就不在这里罗嗦。野火只是在这里介绍 TDA1308 下，让大家知道有这回事。野火 M3-V3 里面的 MP3 模块中采用的是 VS1003 的官方应用电路，没有加耳机功放，而是直接驱动耳机。

## 3.3 实验讲解

本实验是在《2、FatFS(Rev-R0.09)》这个实验基础上进行的。没做过这个实验的话可参考前面的教程，否则有些代码会让您犯糊涂。

首先需要将需要用到的库文件添加进来，有关库的配置可参考前面的教程，这里不再详述。在配置好库的环境之后我们从 main 函数开始分析：

```
1. int main(void)
2. {
3.     SysTick_Init();           /* 配置 SysTick 为 10us 中断一次 */
4.     USART1_Config();         /* 配置串口 1 115200 8-N-1 */
5.
6.     /* Interrupt Config,配置 sdio 的中断优先级, */
7.     NVIC_Configuration();
8.
9.     printf(" \r\n 这是一个 MP3 测试例程 !\r\n ");
10.
11.     VS1003_SPI_Init();       /* MP3 硬件 I/O 初始化 */
```



```
12.
13.      MP3_Start();          /* MP3 就绪，准备播放，在 vs1003.c 实
   现 */
14.
15.      MP3_Play();           /* 播放 SD 卡 (FATFS) 里面的音频文
   件 */
16.
17.  /* Infinite loop */
18.  while (1)
19.  {
20.  }
21. }
```

这里没有调用库函数 `SystemInit()`；是因为在 3.5 的固件库中，在 **3.5 版本**的库中 `SystemInit()` 函数在启动文件 `startup_stm32f10x_hd.d` 中已用汇编语句调用了，设置的时钟为默认的 **72M**。所以在 `main` 函数就不需要再调用啦，当然，再调用一次也是没问题的。

如果你使用的是其它版本的库，在所有工作之前首先要做的就是先设置系统时钟，这可千万别忘了。在 **ST3.0.0** 版本之后的库中，这部分工作都放在了启动文件中了，由汇编实现，只要用户代码一进入 `main` 函数就表示已经初始化好系统时钟了，完全不用用户考虑，用户不知道这点的话还以为不需要初始化系统时钟呢。至于 **ST3.0.0** 和之后高版本的库有什么区别，我想说的是没什么大的区别，代码的目录结构基本没有改变，只是在代码的功能增多了，支持更完善的外设。

`SysTick` 为 10us 中断一次用于 `SysTick` 为 10us 中断一次，用于后面的延时函数。

`USART1_Config()`；配置串口 1 波特率为 115200，8 个数据位，1 个停止位，无硬件流控制。

`NVIC_Configuration()`；用于配置 MicroSD 卡的中断优先级。

`VS1003_SPI_Init()`；用于初始化 MP3 解码芯片 VS1003B 需要用到的 I/O 口，包括数据口(SPI2)和控制 I/O。`VS1003_SPI_Init()`；由用户在 `vs1003.c` 中实现：

```
1.  /*
2.   * 函数名: VS1003_SPI_Init
3.   * 描述   : VS1003 所用 I/O 初始化
4.   * 输入   : 无
5.   * 输出   : 无
6.   * 调用   : 外部调用
7.   */
8.  void VS1003_SPI_Init(void)
9.  {
10.     SPI_InitTypeDef  SPI_InitStructure;
11.     GPIO_InitTypeDef GPIO_InitStructure;
12.
13.     /* 使能 VS1003B 所用 I/O 的时钟 */
```



```
14.     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB | RCC_APB2Periph_GPIOC, ENABLE);
15.     /* 使能 SPI2 时钟 */
16.     RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE);
17.
18.     /* 配置 SPI2 引脚: PB13-SCK, PB14-MISO 和 PB15-MOSI */
19.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;
20.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
21.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
22.     GPIO_Init(GPIOB, &GPIO_InitStructure);
23.
24.     /* PB12-XCS(片选) */
25.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
26.     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
27.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
28.     GPIO_Init(GPIOB, &GPIO_InitStructure);
29.
30.     /* PB11-XRST(复位) */
31.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;
32.     GPIO_Init(GPIOB, &GPIO_InitStructure);
33.
34.
35.     /* PC6-XDCS(数据命令选择) */
36.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
37.     GPIO_Init(GPIOC, &GPIO_InitStructure);
38.
39.     /* PC7-DREQ(数据中断) */
40.     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPD;
41.     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;
42.     GPIO_Init(GPIOC, &GPIO_InitStructure);
43.
44.     /* SPI2 configuration */
45.     SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
46.     SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
47.     SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
48.     SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;
49.     SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;
50.     SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
51.     SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_32;
52.     SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
53.     SPI_InitStructure.SPI_CRCPolynomial = 7;
54.     SPI_Init(SPI2, &SPI_InitStructure);
55.
56.     /* Enable SPI2 */
57.     SPI_Cmd(SPI2, ENABLE);
58. }
```

假如我们要将数据口换成 SPI1 或者改变其他控制 I/O，只需改变这个函数即可，移植性非常强。关于 STM32 的 SPI 接口详细使用教程，请参照前面的 FLASH 或 E<sup>2</sup>PROM 的文档。

`MP3_Start();` 使 MP3 进入就绪模式(standby)，随时播放音乐。`MP3_Start();` 在 vs1003.c 中实现：

```
1.  * 函数名: MP3_Start
2.  * 描述   : 使 MP3 进入就绪模式，随时准备播放音乐。
3.  * 输入   : 无
4.  * 输出   : 无
5.  * 调用   : 外部调用
6.  */
7.  void MP3_Start(void)
8.  {
9.      u8 BassEnhanceValue = 0x00;      // 低音值先初始化为 0
10.     u8 TrebleEnhanceValue = 0x00;     // 高音值先初始化为 0
11.     TRST_SET(0);
12.     Delay_us( 1000 );                 // 1000*10us = 10ms
```

```
13.
14.     VS1003_WriteByte(0xff);           // 发送一个字节的无效数据, 启动 SPI 传输
15.     TXDCS_SET(1);
16.     TCS_SET(1);
17.     TRST_SET(1);
18.     Delay_us( 1000 );
19.
20.     Mp3WriteRegister( SPI_MODE,0x08,0x00);    // 进入 VS1003 的播放模式
21.     Mp3WriteRegister(3, 0x98, 0x00);         // 设置 vs1003 的时钟,3 倍频
22.     Mp3WriteRegister(5, 0xBB, 0x81);         // 采样率 48k, 立体声
23.     // 设置重低音
24.     Mp3WriteRegister(SPI_BASS, TrebleEnhanceValue, BassEnhanceValue);
25.     Mp3WriteRegister(0x0b,0x00,0x00);    // VS1003 音量
26.     Delay_us( 1000 );
27.
28.     while( DREQ == 0 );                   // 等待 DREQ 为高 表示能够接受音乐数据输入
29. }
```

函数中涉及到的宏定义都在 `vs1003.h` 这个头文件中实现。关于函数中为什么要这样操作寄存器, 或者为什么要按照这个顺序来操作寄存器, 请大家查阅 `vs1003` 的 pdf, 里面讲得很详细, 有 e 文跟中文资料。

`MP3_Play()`; 这个函数逐个扫描我们卡里面的音频文件, 把根目录下的所有音频文件播放一次, 若音频文件放在其它目录, 可以通过修改代码中的文件路径来实现。以下是 `MP3_Play()` 在 `vs1003.c` 中实现:

```
1. /*
2.  * 函数名: MP3_Play
3.  * 描述   : 读取 SD 卡里面的音频文件, 并通过耳机播放出来
4.  *         支持的格式: mp3,mid,wma, 部分的 wav
5.  * 输入   : 无
6.  * 输出   : 无
7.  * 说明   : 已添加支持长中文文件名
8.  */
9. void MP3_Play(void)
10. {
11.
12.     FATFS fs;           // Work area (file system object) for logical drive
13.     FRESULT res;
14.     UINT br;            /*读取出的字节数, 用于判断是否到达文件尾*/
15.     FIL fsrc;           // file objects
16.     FILINFO finfo;      /*文件信息*/
17.     DIR dirs;
18.     uint16_t count = 0;
19.
20.     char lfn[70];        /*为支持长文件的数组, []最大支持 255*/
21.     char j = 0;
22.     char path[100] = {" "}; /* MicroSD 卡根目录 */
23.     char *result1, *result2, *result3, *result4;
24.
25.     BYTE buffer[512];    /* 存放读取出的文件数据 */
26.
27.     finfo.lfname = lfn;  /*为长文件名分配空间*/
28.     finfo.lfsize = sizeof(lfn); /*空间大小*/
29.
30.     f_mount(0, &fs);    /* 挂载文件系统到 0
31.     */
```



```
32.     if (f_opendir(&dirs,path) == FR_OK)                /* 打开根目
录 */
33.     {
34.         while (f_readdir(&dirs, &finfo) == FR_OK)      /* 依次读取文件
名 */
35.         {
36.
37.             if ( finfo.fattrib & AM_ARC )                /* 判断是否为存档型文档 */
38.             {
39.                 if(finfo.lfname[0] == NULL && finfo.fname !=NULL) /*当长文
件名称为空，短文件名非空时转换*/
40.                     finfo.lfname =finfo.fname;
41.
42.
43.                 if( !finfo.lfname[0] )                  /* 文件名为空即到达了目录的末尾，退
出 */
44.                     break;
45.
46.                     printf( " \r\n 文件名为: %s \r\n",finfo.lfname );
47.
48.                     result1 = strstr( finfo.lfname, ".mp3" ); /* 判断是否
为音频文件 */
49.                     result2 = strstr( finfo.lfname, ".mid" );
50.                     result3 = strstr( finfo.lfname, ".wav" );
51.                     result4 = strstr( finfo.lfname, ".wma" );
52.
53.                     if ( result1!=NULL || result2!=NULL || result3!=NULL |
| result4!=NULL )
54.                     {
55.
56.                         if(result1 != NULL)/*若是 mp3 文件则读取 mp3 的信息*/
57.                         {
58.                             res = f_open( &fsrc, finfo.lfname, FA_OPEN_EXI
STING | FA_READ ); /* 以只读方式打开 */
59.
60.
61.                             /* 获取歌曲信息
(ID3V1 tag / ID3V2 tag) */
62.                             if ( Read_ID3V1(&fsrc, &id3v1) == TRUE )
63.                             {
64.                                 printf( "\r\n 曲
目      : %s \r\n", id3v1.title );
65.                                 printf( "\r\n 艺术
家      : %s \r\n", id3v1.artist );
66.
67.                                 printf( "\r\n 专
辑      : %s \r\n", id3v1.album );
68.                             }
69.                             else
70.                             {
71.                                 /* 有些 MP3 文件没有 ID3V1 tag,只有
ID3V2 tag
72.
73.                                 res = f_lseek(&fsrc, 0);
74.                                 Read_ID3V2(&fsrc, &id3v2);
75.
76.                                 printf( "\r\n 曲
目      : %s \r\n", id3v2.title );
77.                                 printf( "\r\n 艺术
家      : %s \r\n", id3v2.artist );
78.                             }
79.                         }
80.                     }
81.                     /* 使文件指针 fsrc 重新指向文件头，因为在调用
Read_ID3V1/Read_ID3V2 时，
```





```
78.             fsrc 的位置改变了 */
79.             res = f_open( &fsrc, finfo.lfname, FA_OPEN_EXI
    STING | FA_READ );
80.             res = f_lseek(&fsrc, 0);
81.
82.
83.             br = 1;                               /* br 为全局变
    量 */
84.             TXDCS_SET( 0 );                         /* 选择 VS1003 的数据接
    口 */
85. /* ----- 一曲开始 ----- */
86.             printf( " \r\n 开始播放 \r\n" );
87.             for (;;)
88.             {
89.                 res = f_read( &fsrc, buffer, sizeof(buffer), &
    br );
90.                 if ( res == 0 )
91.                 {
92.                     count = 0;
93.                     /* 512 字节完重新计数 */
94.                     Delay_us( 1000 );               /* 10ms 延
    时 */
95.                     while ( count < 512)           /* SD 卡
    读取一个 sector, 一个 sector 为 512 字节 */
96.                     {
97.                         if ( DREQ != 0 )           /* 等待 DREQ 为高, 请求
    数据输入 */
98.                         {
99.                             for (j=0; j<32; j++ ) /* VS100
    3 的 FIFO 只有 32 个字节的缓冲 */
100.                            {
101.                                VS1003_WriteByte( b
    uffer[count] );
102.                                count++;
103.                            }
104.                        }
105.                    }
106.                }
107.                if (res || br == 0) break;          /* 出错或者到
    了 MP3 文件尾 */
108.            }
109.             printf( " \r\n 播放结束 \r\n" );
110.             /* ----- 一曲结束 ----- */
111.             count = 0;
112.             /* 根据 VS1003 的要求, 在一曲结束后需发送 2048 个
    0 来确保下一首的正常播放 */
113.             while ( count < 2048 )
114.             {
115.                 if ( DREQ != 0 )
116.                 {
117.                     for ( j=0; j<32; j++ )
118.                     {
119.                         VS1003_WriteByte( 0 );
120.                         count++;
121.                     }
122.                 }
123.             }
124.             count = 0;
125.             TXDCS_SET( 1 );                         /* 关闭 VS1003 数据端
    口 */
```



```
126.             f_close(&fsrc);    /* 关闭打开的文件 */
127.         }
128.     }
129.     } /* while (f_readdir(&dirs, &finfo) == FR_OK) */
130.     } /* if (f_opendir(&dirs, path) == FR_OK) */
131. } /* end of MP3_Play */
```

由于代码比较长，在格式编排上不是很好，野火建议大家还是配合源代码一起阅读^\_^。

现在我们来大概分析下 `MP3_Play()`；这个函数，这里边涉及到一些文件系统操作的函数，关于这部分函数的操作大家可参考前面的教程或者阅读 **FATFS** 的官方文档，其实我的教程也不完全正确，阅读官方的文档才是最可靠的。

首先说一下为支持中文长文件名的文件系统配置。

要在 `ffconf.h` 文件中的 **Namespace configuration** 宏配置中设定如下：

```
1.  /*-----
2.  /-----/
3.  / Locale and Namespace Configurations
4.  /-----*/
5.  #define _CODE_PAGE 936
6.  /* The _CODE_PAGE specifies the OEM code page to be used on the target
   system.
7.  / Incorrect setting of the code page can cause a file open failure.
8.  /
9.  / 932 - Japanese Shift-JIS (DBCS, OEM, Windows)
10. / 936 - Simplified Chinese GBK (DBCS, OEM, Windows)
11. / 949 - Korean (DBCS, OEM, Windows)
12. / 950 - Traditional Chinese Big5 (DBCS, OEM, Windows)
13. / 1250 - Central Europe (Windows)
14. / 1251 - Cyrillic (Windows)
15. / 1252 - Latin 1 (Windows)
16. / 1253 - Greek (Windows)
17. / 1254 - Turkish (Windows)
18. / 1255 - Hebrew (Windows)
19. / 1256 - Arabic (Windows)
20. / 1257 - Baltic (Windows)
21. / 1258 - Vietnam (OEM, Windows)
22. / 437 - U.S. (OEM)
23. / 720 - Arabic (OEM)
24. / 737 - Greek (OEM)
25. / 775 - Baltic (OEM)
26. / 850 - Multilingual Latin 1 (OEM)
27. / 858 - Multilingual Latin 1 + Euro (OEM)
28. / 852 - Latin 2 (OEM)
29. / 855 - Cyrillic (OEM)
30. / 866 - Russian (OEM)
31. / 857 - Turkish (OEM)
32. / 862 - Hebrew (OEM)
33. / 874 - Thai (OEM, Windows)
34. / 1 - ASCII only (Valid for non LFN cfg.)
35. */
36.
37.
38. #define _USE_LFN 2 /* 0 to 3 */
```

```
39. #define _MAX_LFN      255      /* Maximum LFN length to handle (12 to 255) */
40. /* The _USE_LFN option switches the LFN support.
41. /
42. /    0: Disable LFN feature. _MAX_LFN and _LFN_UNICODE have no effect.

43. /    1: Enable LFN with static working buffer on the BSS. Always NOT reentrant.
44. /    2: Enable LFN with dynamic working buffer on the STACK.
45. /    3: Enable LFN with dynamic working buffer on the HEAP.
46. /
47. /    The LFN working buffer occupies (_MAX_LFN + 1) * 2 bytes. To enable LFN,
48. /    Unicode handling functions ff_convert() and ff_wtoupper() must be added
49. /    to the project. When enable to use heap, memory control functions
50. /    ff_memalloc() and ff_memfree() must be added to the project. */
51.
52.
53. #define _LFN_UNICODE    0      /* 0:ANSI/OEM or 1:Unicode */
54. /* To switch the character code set on FatFs API to Unicode,
55. /    enable LFN feature and set _LFN_UNICODE to 1. */
56.
57.
58. #define _FS_RPATH        0      /* 0 to 2 */
59. /* The _FS_RPATH option configures relative path feature.
60. /
61. /    0: Disable relative path feature and remove related functions.
62. /    1: Enable relative path. f_chdrive() and f_chdir() are available.

63. /    2: f_getcwd() is available in addition to 1.
64. /
65. /    Note that output of the f_readdir function is affected by this option. */
```

修改的第一个宏配置是 `_CODE_PAGE` 改成简体中文的 **936**。

Code page 是什么？我们知道 ASCII 码的前 7 位定义的是我们常用的标准字符集，于是 128 位以下的用处达成了共识，而 ASCII 码中的第 8 位没有被使用，对于 128 位以上的可能有不同的解释，这些不同的解释就叫做 code\_page，我们使用 936 这个宏就是调用了简体中文的 code\_page。所以要支持中文，还要添加 fatfs 源文件中 option 目录下的 cc936.c 文件到工程中。

接下来还要修改 `_USE_LFN` 和 `MAX_LFN` 的宏，这两个是长文件名支持的配置。

`MAX_LFN` 定义了最大文件名长度，单位为 Byte。

`_USE_LFN >= 1` 则开启长文件支持。

=1 表示长文件名的存储在 静态存储区。

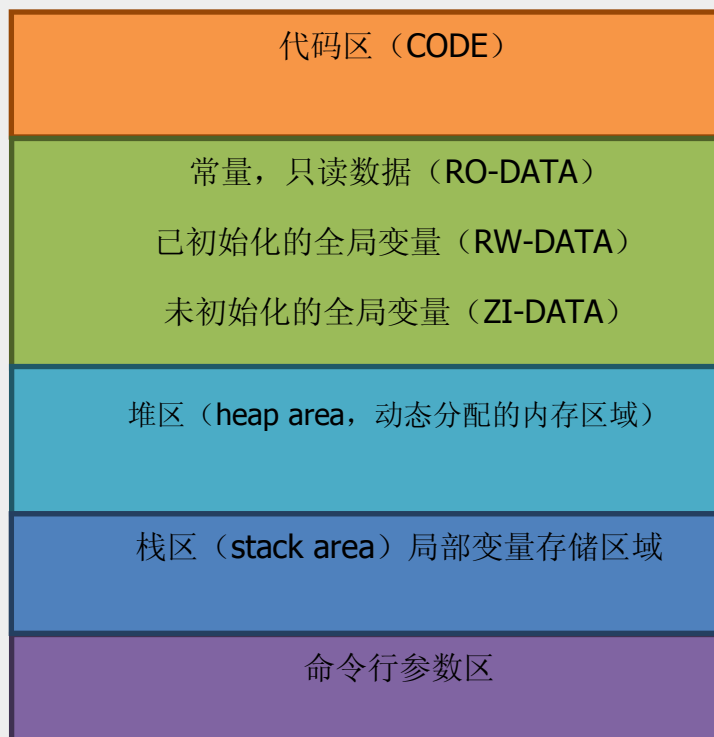
=2 表示长文件名的存储在 栈区。

=3 表示长文件名的存储在 堆区。



这里涉及到变量的存储分布问题。

sram 内存变量分布图:



存储在全局变量的内存空间是不会被回收的, 栈区是用来存放局部变量的, 在子函数调用运行完成之后会释放内存, 而且少用全局变量会让代码的移植性更好。所以这里的长文件名变量我们把它设置为 2, 把它放在栈区。

另外, 因为我们的 `mp3_play()` 函数中定义了很多局部变量, 占用的栈空间很大, 所以我们要修改启动文件 `startup_stm32f10x_hd.s` 中的栈空间大小:

```
1. Stack_Size      EQU      0x00000f00      ;Stack_Size, 标号. EQU 定义
2.                AREA      STACK, NOINIT, READWRITE, ALIGN=3      ;定义名称
   为 stack 的栈, nointStack_Mem      SPACE      Stack_Size      ;定义名称为
   Stack_Mem 大小为 stack_size 大小
3. __initial_sp
```

在这个文件中把原来的

```
Stack_size EQU 0x00000400
```

改成了

```
Stack_size EQU 0x0000f00 。
```

有时我们调试程序的时候会发现代码莫名奇妙地卡在 `harddefault` 的硬中断里，这时可以检查一下是不是在启动文件中把栈大小设置得太保守了，可以根据实际需要把这个设置得大一点。

文件系统中的文件信息结构体：

```
1. /* File status structure (FILINFO) */
2.
3. typedef struct {
4.     DWORD    fsize;           /* File size */
5.     WORD     fdate;           /* Last modified date */
6.     WORD     ftime;           /* Last modified time */
7.     BYTE     fattrib;         /* Attribute */
8.     TCHAR    fname[13];       /* Short file name (8.3 format) */
9. #if _USE_LFN
10.    TCHAR*    lfname;          /* Pointer to the LFN buffer */
11.    UINT      lfsz;            /* Size of LFN buffer in TCHAR */
12. #endif
13. } FILINFO;
```

关于长文件名（包管中英文）的支持，最后还要注意一点，在使用文件名信息时，不要再使用 `FILINFO->fname`（短文件名数组）。而应该使用 `FILINFO->lfname`（长文件名指针）。而且长文件名在结构体中定义的是一个指针，在使用前我们要为这个指针分配内存空间，注意不要使用野指针。具体的使用方法可以参照 `mp3_play()` 函数中开头的[变量定义和赋初值部分](#)。

还要注意一下如果读取的文件名长度不超过 `FILINFO->fname`（短文件名）的空间时，文件名的信息只会保存在短文件名数组中，而 `FILINFO->lfname`（长文件名指针）的值将会是空的，所以我在代码中加了一个[判断语句](#)才可以进行正常的使用。

函数 `f_mount(0, &fs);` 为我们在文件系统中注册一个工作区，并初始化盘符的名为 0。这个函数还调用了底层的 `disk_initialize()`，进行 `sdio` 的初始化，所以在文件操作之前必须调用这个函数。不建议在 `main` 函数直接调用 `disk_initialize()` 来对 `sdio` 进行初始化，要尽量使用封装好的脱离硬件层的函数，这样会令代码移植性更好呀。

函数 `f_opendir(&dirs, path)` 用于打开卡的根目录，并将这个根目录关联到 `dirs` 这个结构指针，然后我们就可以通过这个结构指针来操作这个目录了，其实这个结构指针就类似 `LINUX` 下系统编程中的文件描述符，不论是操作还是目录都得通过文件描述符才能操作。





`f_readdir(&dirs, &finfo)` 函数通过刚刚的 `dirs` 结构指针来读取目录里面的信息，并将目录的信息储存在 `finfo` 这个结构体变量中。这个结构体中包括了文件名，文件大小，文件类型，修改时间等信息。

紧接着判断文件的属性，如果是存档型文件的话就将文件名打印出来，然后比较文件的后缀名，查看是否为音频文件，支持的音频格式有 `mp3`、`mid`、`wav`、`wma`。

如果是音频文件的话则调用 `f_open(&fsrc, finfo.fname, FA_OPEN_EXISTING | FA_READ)`；打开这个音频文件。

如果是 `mp3` 类型的文件，我们还可以调用 `Read_ID3V1()` 和 `Read_ID3V2()` 来读取 `mp3` 的文件信息，这些文件信息是属于 `mp3` 文件的内部数据，可以参照《`mp3` 文件的存储格式》这个文档来理解这两个函数，实质就是把文件记录的数据，按格式把相应的信息整合到结构体里便于使用而已。

我们把读取到的音频数据直接通过 `SPI` 接口送入到 `vs1003` 就可以进行各种音频数据的解码了。

`TXDCS_SET(0)`；用于选择 `vs1003` 的数据端口，准备往 `vs1003` 中输入数据。其中 `TXDCS_SET(0)`；是在 `vs1003.h` 中实现的一个宏：

```
1. #define XDCS      (1<<6)    // PC6-XDCS
2.
3. #define TXDCS_SET(x)  GPIOC->ODR=(GPIOC->ODR&~XDCS)|(x ? XDCS:0)
```

紧接着进入一个大循环中播放我们的 `mp3` 文件。

函数 `f_read(&fsrc, buffer, sizeof(buffer), &br)`；从文件中读取 512 个字节的数据到缓冲区中，至于为什么是 512 个字节，这是因为卡的一个 `sector` 是 512 个字节，一次只能读取一个 `sector`，实际上也可以一次读取 `n` 个 `sector`，但在这里没必要。

函数 `VS1003_WriteByte(buffer[count])`；将缓冲区中的数据写入 `vs1003` 的数据缓冲区。注意，这里一次只能写入 32 个字节，这是因为 `vs1003` 的 `FIFO` 的大小为 32 个字节，写多了无效。



当文件出错或者一曲播放完毕时就跳出 `for` 循环，并打印出“播放结束”的调试信息。

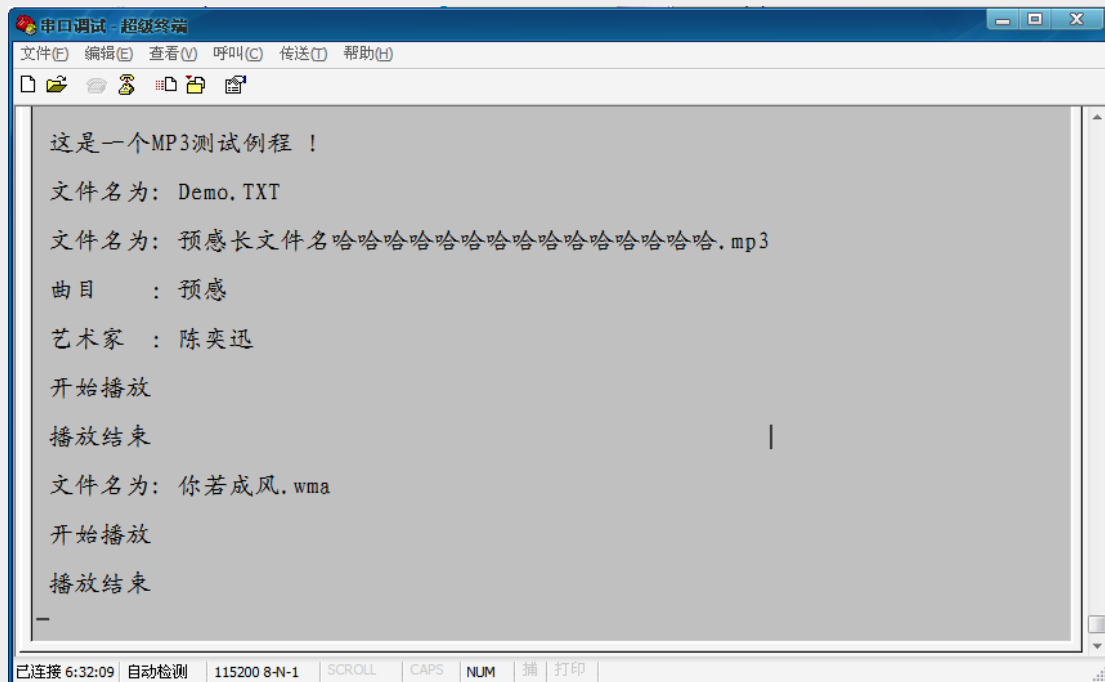
根据 VS1003 的要求，在一曲结束后需发送 2048 个 0 来确保下一首的正常播放。

一曲播放完毕我们关闭 `vs1003` 的数据端，关闭打开的文件，等待下一曲的播放，直到目录下的音频文件播放完为止。

这里面涉及到了 `vs1003` 操作的一些特性，需大家参考 `vs1003` 的 `datasheet` 来帮助理解。

### 3.4 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(野火用的是 1G，也可支持 2G、4G、8G)，在卡的根目录下要有 `mp3` 文件，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：



```
串口调试-超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
这是一个MP3测试例程！
文件名为: Demo.TXT
文件名为: 预感长文件名哈哈哈哈哈哈哈哈哈哈哈哈哈哈哈.mp3
曲目 : 预感
艺术家 : 陈奕迅
开始播放
播放结束
文件名为: 你若成风.wma
开始播放
播放结束
已连接 6:32:09 | 自动检测 | 115200 8-N-1 | SCROLL | CAPS | NUM | 插 | 打印
```

野火的卡的根目录下放了 1 个 `mp3` 文件，1 个 `wma` 文件。可以看到，这个代码支持了超长的中文文件名；也支持了 `wma` 的格式，根据 `vs1003` 的



datasheet 说明，还可以支持 mid 和部分的 wav 音频，大家可以尝试一下音量可通过耳机来调，前提是你的耳机要能调节音量才行。

