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！

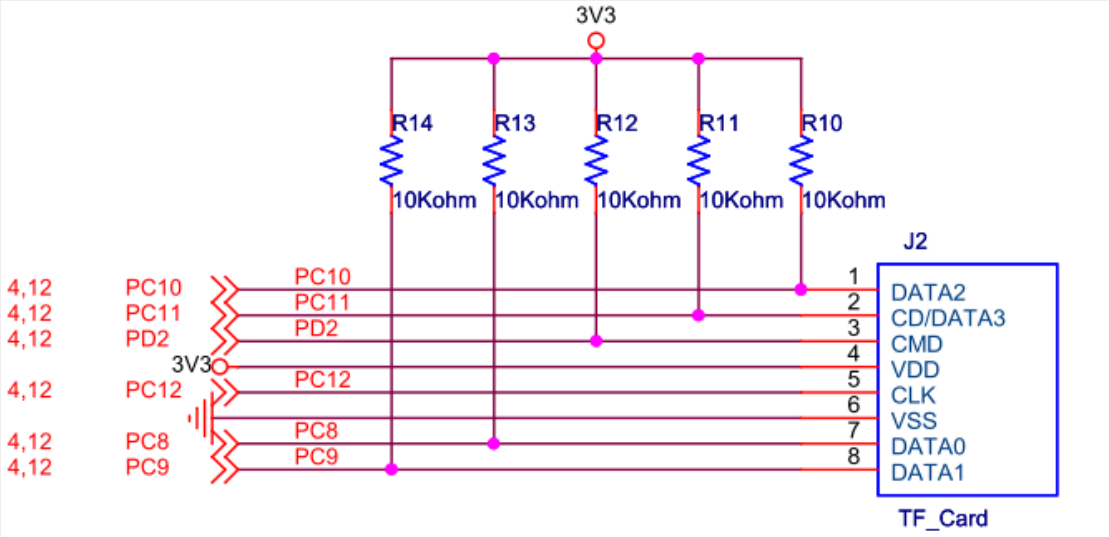
2、FatFs (Rev-R0.09)

2.1 实验描述及工程文件清单

实验描述	MicroSD 卡文件系统 FATFS R0.07C 测试实验。在 MicroSD 卡里面创建一个 DEMO.TXT 文本文件，在文件里面写入字符串“感谢您选用 野火 STM32 开发板 ！ ^_^”，然后通过串口将这些内容打印在电脑的超级终端上。这个更新版本的代码增加了简体中文和长文件名的支持，采用 SDIO 的 4bit+DMA 模式。并图解了文件系统移植的全部过程。
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
文件系统文件	ff9/diskio.c ff9/ff.c ff9/cc936.c

野火 STM32 开发板 MicroSD 卡硬件原理图：



2.2 实验简介

本实验是在上一讲《SDIO（4bit + DMA）》的基础上讲解的，只有上一讲的实验成功了，文件系统才能跑起来。有关卡的底层的初始化，这里不再详述，这里着重讲解文件系统的移植和文件系统的应用。这个文档是更新版本，采用的文件系统的版本是目前最新的 R0.09。 文件系统的源码可以从 fatfs 官网下载 http://elm-chan.org/fsw/ff/00index_e.html

2.3 FatFS 文件系统简介

FAFFS 是面向小型嵌入式系统的一种通用的 FAT 文件系统。FATFS 完全是由 AISI C 语言编写并且完全独立于底层的 I/O 介质。因此它可以很容易地不加修改地移植到其他的处理器当中，如 8051、PIC、AVR、SH、Z80、H8、ARM

等。FATFS 支持 FAT12、FAT16、FAT32 等格式，所以我们利用前面写好的 SDIO 驱动，把 FATFS 文件系统代码移植到工程之中，就可以利用文件系统的各种函数，对已格式化的 SD 卡进行读写文件了。

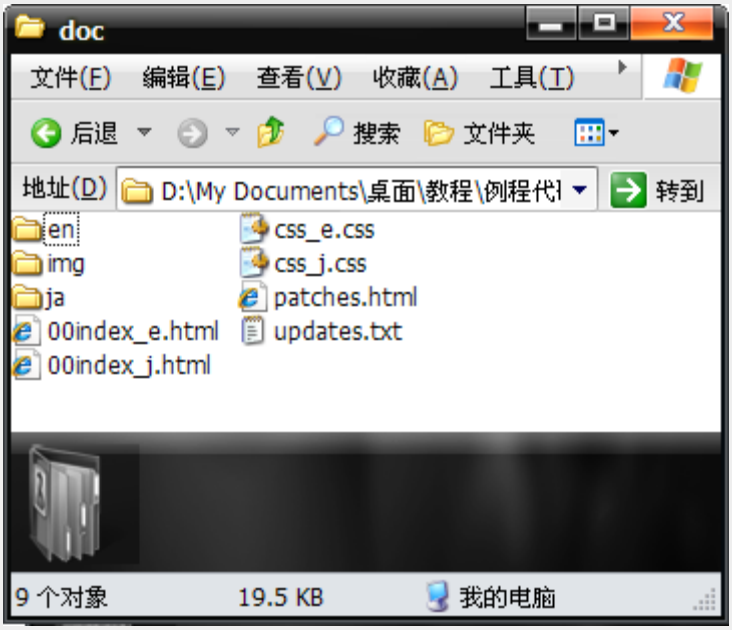
本实验是将 FATFS 移植到野火 STM32 开发板中，CPU 为 STM32F103VET6，是采用 ARM 公司最新内核 ARMV7 的一款单片机。

2.4 移植前的工作

2.4.1 分析 FATFS 的目录结构

在移植 FATFS 文件系统之前，我们先要到 FAT 的官网获取源码，版本为 R0.07C。解压之后可看到里面有 doc 和 src 这两个文件夹。doc 文件夹里面是一些使用文档， src 里面是文件系统的源码。

打开 doc 文件夹，可看到如下文件目录：



其中 en 和 ja 这两个文件夹里面是编译好的 html 文档，讲的是 FATFS 里面各个函数的使用方法，这些函数就如 LINUX 下的系统调用，是封装得非常好的函数，利用这些函数我们就可以操作我们的 MicroSD 卡了。有关具体的函数我们在用到的时候再讲解。这两个文件夹的唯一区别就是 en 文件夹下的文档是英文的，ja 文件夹下的是日文的。偏偏就是没中文的，真狗血呀。00index_e.html 是一些关于 FATFS 的英文简介，updates.txt 是 FATFS 的更新信息，至于其他几个文件可以不看。

打开 src 文件夹，可看到如下目录：

option 文件夹下是一些可选的外部 c 文件，包含了多语言支持需要用到的文件和转换函数。

00readme.txt 说明了当前目录下 diskio.c 、diskio.h、ff.c、ff.h、integer.h 的功用、涉及了 FATFS 的版权问题(是自由软件)，还讲到了 FATFS 的版本更新信息。

integer.h：是一些数值类型定义

diskio.c ：底层磁盘的操作函数，这些函数需要用户自己实现

ff.c ：独立于底层介质操作文件的函数，完全由 ANSI C 编写

cc936.c :简体中文支持所需要添加的文件，包含了简体中文的 GBK 和转换函数。

只要添加进来就行，这个文件不需要修改。

ffconf.h:这个头文件包含了对文件系统的各种配置，如需要支持简体中文要把_CODE_PAGE 的宏改成 936 并把上面的 cc936.c 文件加入到工程之中

建议阅读这些源码的顺序为：integer.h -> diskio.c -> ff.c 。

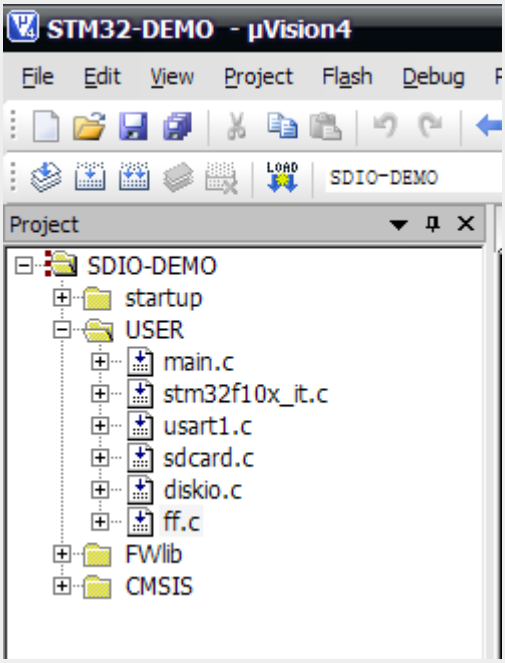
关于具体源码的分析不是我能力所及呀，大家就自己研究吧。我的主要工作是带领大家把这个文件系统移植到我们的开发板上，让这个文件系统先跑起来，这样才是硬道理呀。文件系统工作起来了的话，源码的分析那自然是大家的活啦。

2.5 开始移植

首先我们要获取一个完全没有修改过的文件系统源码，然后在 10-MicroSD 卡这个文件夹下的实验代码下移植，这个实验代码实现的是卡的底层的块操作。注意，我们在移植这个文件系统的过程中会尽量保持文件系统源码的纯净，尽量做到在修改最少量的源码的情况下移植成功。

首先将 integer.h、diskio.h、diskio.c、ff.h、ff.c 添加到工程目录下的 USER 文件夹下，如下截图：

然后并回到 MDK 界面下将 diskio.c 、ff.c 这两个文件添加到 USER 目录下，如下截图：



因为我们要用到这两个 c 文件，所以我们在 main.c 中将这两个 c 文件对应的头文件 diskio.h 、ff.h 包含进来，如下截图：

```
/********************* (C) COPYRIGHT 2011 野
 * 文件名  : main.c
 * 描述    : MicroSD卡文件系统 FATFS R0.07
 * 实验平台: 野火STM32开发板
 * 库版本  : ST3.0.0
 *
 * 作者    : fire  QQ: 313303034
 * 博客    : firestm32.blog.chinaunix.net
*****************************************/
#include "stm32f10x.h"
#include "usart1.h"
#include "sdcard.h"
#include "ff.h"
#include "diskio.h"

#include <string.h>
```

好嘞，下面我们开始编译，这时会出现如下错误：

```
Build Output
main.c(70): warning:  #870-D: invalid multibyte character sequence
main.c(70): warning:  #870-D: invalid multibyte character sequence
main.c(70): warning:  #870-D: invalid multibyte character sequence
main.c(70): warning:  #870-D: invalid multibyte character sequence
main.c(70): warning:  #870-D: invalid multibyte character sequence
main.c(70): warning:  #870-D: invalid multibyte character sequence
main.c(70): warning:  #870-D: invalid multibyte character sequence
main.c(70): warning:  #870-D: invalid multibyte character sequence
compiling diskio.c...
..\CMSIS\stm32f10x.h(237): error:  #101: "FALSE" has already been declared in the current scope
..\CMSIS\stm32f10x.h(237): error:  #101: "TRUE" has already been declared in the current scope
compiling ff.c...
Target not created
```

意思是说 FALSE 跟 TRUE 这两个变量已经定义过了，为什么会出现这个错误呢？因为在 integer.h 和我们的 M3 库头文件 stm32f10x.h 中都定义了这两个变量，所以就产生了重复定义：

integer.h

```
31  /* Boolean type */
32  typedef enum { FALSE = 0, TRUE } BOOL;
33
```

stm32f10x.h

```
0236
0237  typedef enum {FALSE = 0, TRUE = !FALSE} bool;
0238
```

怎么解决呢，很简单，只要搞掉一个即可，但要去掉哪个呢？我们考虑到外面的 M3 库很多文件都包含了 stm32f10x.h 这个头文件，假如是修改

stm32f10x.h 的话工作量会非常大，鉴于此就只能委屈 integer.h 了，修改如下，将它注释掉：

```
31 /* Boolean type */
32 //typedef enum { FALSE = 0, TRUE } BOOL; /* add by fire */
33
```

好嘞，修改之后，我们再编译下，接着又出现如下错误：

```
diskio.h(29): error:  #20: identifier "BOOL" is undefined
compiling ff.c...
diskio.h(29): error:  #20: identifier "BOOL" is undefined
ff.c(584): error:  #20: identifier "BOOL" is undefined
ff.c(861): error:  #20: identifier "FALSE" is undefined
ff.c(915): error:  #20: identifier "FALSE" is undefined
ff.c(1008): error:  #20: identifier "TRUE" is undefined
ff.c(2254): error:  #20: identifier "FALSE" is undefined
Target not created
```

意思是说在 diskio.h、ff.c 中 BOOL 、FALSE、TRUE 没定义。刚刚才把人家注释掉了，不报错才怪。

解决方法如下：

1、将 integer.h 中有关 BOOL 的那句注释掉，注释掉也没太大关系，因为注释掉的不会怎么用到：

```
28
29 //BOOL assign_drives (int argc, char *argv[]); /* add by fire */
30
```

2、在 ff.c 文件的开头重新定义一个布尔变量，取名为 bool，与 stm32f10x.h 中的名字一样：

```
0076 //#include "stm32f10x.h"  /* add by fire */
0077 typedef enum {FALSE = 0, TRUE = !FALSE} bool; /* add by fire */
0078
```

同时在 ff.c 的第 585 行做如下修改：

```
0581 static
0582 FRESULT dir_next (  /* FR_OK:Succeeded, FR_NO
0583   DIR *dj,     /* Pointer to directory object
0584   //BOOL streach  /* FALSE: Do not streach tal
0585   bool streach    /* add by fire */
0586 )
```

现在我们再编译下，发现既没警告也没错误：

```
Build Output
Build target 'SDIO-DEMO'
compiling main.c...
compiling diskio.c...
compiling ff.c...
linking...
Program Size: Code=22822 RO-data=342 RW-data=636 ZI-data=3428
FromELF: creating hex file...
"..\Output\STM32-DEMO.axf" - 0 Error(s), 0 Warning(s).
```

到这里我们算是把文件系统移植成功了，接下来的任务就是调用文件系统的函数来操作我们的卡了。其实这里的移植是非常非常简单的，要是你学过 LINUX 的话，那里面的 UBOOT 移植，系统移植，那才叫人头疼，就光是目录里面的文件夹都几千个，更别说是找到要修改的源代码了，刚接触的话绝对叫你吐血，就连下载个交叉编译器都涉及到移植。

2.6 实验代码分析

FATFS 是独立于底层介质的应用函数库，对底层介质的操作都要交给用户去实现，其仅仅是提供了一个函数接口而已，函数为空，要用户添加代码。

这几个函数的原型如下，在 diskio.c 中定义：

```
1.  /* Inidialize a Drive */
2.  DSTATUS disk_initialize (
3.      BYTE drv        /* Physical drive nmuber (0..) */
4.  )
```

```
1.  /* Return Disk Status  */
2.  DSTATUS disk_status (
3.      BYTE drv    /* Physical drive nmuber (0..) */
4.  )
```

```
1.  /* Read Sector(s) */
2.  DRESULT disk_read (
```

```
    BYTE drv,    /* Physical drive nmuber (0..) */
    BYTE *buff, /* Data buffer to store read data */
    DWORD sector,    /* Sector address (LBA) */
    BYTE count  /* Number of sectors to read (1..255) */
)
```

```
/* Write Sector(s) */
#if _READONLY == 0
DRESULT disk_write (
    BYTE drv,             /* Physical drive nmuber (0..) */
    const BYTE *buff,    /* Data to be written */
    DWORD sector,        /* Sector address (LBA) */
    BYTE count       /* Number of sectors to write (1..255) */
)
```

```
/* Miscellaneous Functions                                          */
DRESULT disk_ioctl (
    BYTE drv,    /* Physical drive nmuber (0..) */
    BYTE ctrl,  /* Control code */
    void *buff  /* Buffer to send/receive control data */
)
```

这些函数都是操作底层介质的函数，都需要用户自己实现，然后 FATFS 的应用函数就可以调用这些函数来操作我们的卡了。关于这些底层介质函数是如何实现的，请参考源码，这里就不贴出来了。

在 diskio.c 的最后我们还得提供了获取时间的函数，因为 ff.c 中调用了这个函数，而 FATFS 库又没有给出这个函数的原型，所以需要用户实现，不然会编译出错，函数体为空即可（也可以为它加载 STM32 的 RTC 驱动）：

```
/* 得到文件Calendar格式的建立日期,是DWORD get_fattime (void) 逆变换 */
/*-----------------------------------------------------------------------*/
/* User defined function to give a current time to fatfs module          */
/* 31-25: Year(0-127 org.1980), 24-21: Month(1-12), 20-16: Day(1-31) */
/* 15-11: Hour(0-23), 10-5: Minute(0-59), 4-0: Second(0-29 *2) */
DWORD get_fattime (void)
{
  return 0;
}
```

实现好底层介质的操作函数之后，我们就可以回到应用层了，下面我们从 main 函数开始看起。有关系统初始化和串口初始化这部分请参考前面的教程，这里不再详述。

首先我们调用函数 `disk_initialize( 0 );` 将我们的底层硬件初始化好，这一步非常重要，如果不成功的话，接下来什么都干不了。

`f_open( &fsrc , "0:/Demo.TXT" , FA_CREATE_NEW | FA_WRITE);` 将在刚刚开辟的工作区的盘符 0 下打开一个名为 Demo.TXT 的文件，以只写的方式打开，如果文件不存在的话则创建这个文件。并将 Demo.TXT 这个文件关联到 fsrc 这个结构指针，以后我们操作文件就是通过这个结构指针来完成的。

`f_write(&fsrc, textFileBuffer, sizeof(textFileBuffer), &br);` 将缓冲区的数据写到刚刚打开的 Demo.TXT 文件中。写完之后调用 `f_close(&fsrc);`。关闭文件，

`f_open(&fsrc, "0:/Demo.TXT", FA_OPEN_EXISTING | FA_READ);`以只读的方式打开刚刚的文件。

`f_read( &fsrc, buffer, sizeof(buffer), &br );` 将文件的内容读到缓冲区，然后调用 `printf("\r\n %s ", buffer);`将数据打印到电脑的超级终端。

最后调用 `f_close(&fsrc);`关闭文件。当被打开的文件操作完成之后都要调用 `f_close();`将它关闭，就像一块动态分配的内存在用完之后都要调用 `free()`来将它释放。

这里涉及到了 FATFS 文件系统库函数的操作，如果你学过 LINUX 系统调用的话，操作这些函数将是非常简单，没有学过的话也没太大的关系，因为 FATFS 源码目录 doc 这个文件夹中提供了每个应用函数的用法，如 `f_mount()`:

f_mount

The f_mount fucntion registers/unregisters a work area to the FatFs module.

```
FRESULT f_mount (
  BYTE   Drive,               /* Logical drive number */
  FATFS*   FileSystemObject   /* Pointer to the work area */
);
```

Parameters

Drive

Logical drive number (0-9) to register/unregister the work area.

FileSystemObject

Pointer to the work area (file system object) to be registered.

Return Values

FR_OK (0)

The function succeeded.

FR_INVALID_DRIVE

The drive number is invalid.

Description

The f_mount function registers/unregisters a work area to the FatFs modu
function. To unregister a work area, specify a NULL to the **FileSystemO**

This function only initializes the given work area and registers its addres
process is performed on first file access after f_mount or media change.

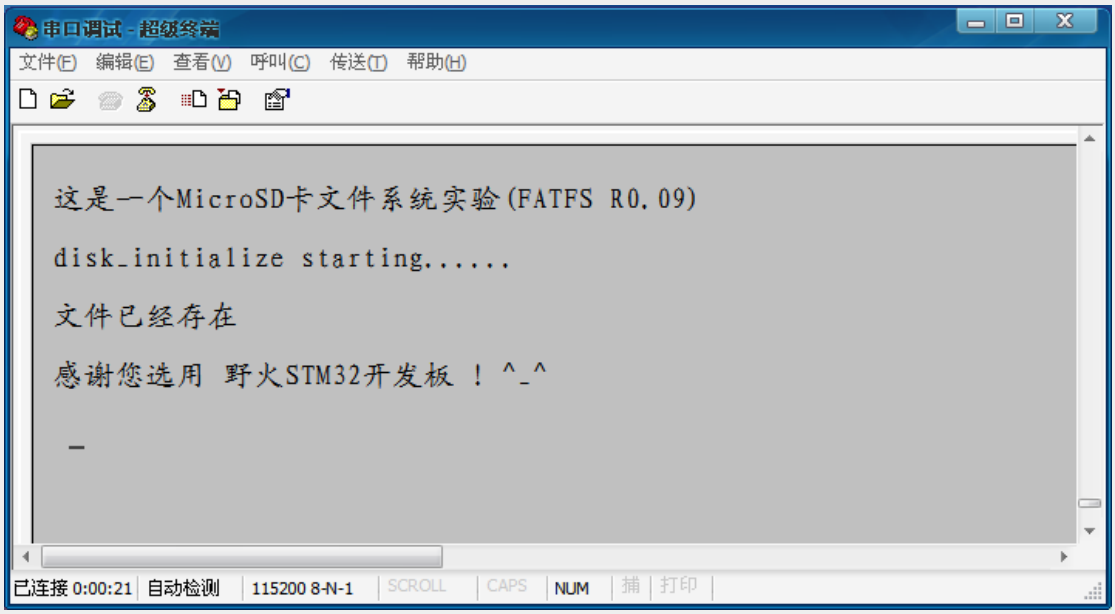
References

FATFS

Return

2.7 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，4G 的也已经测试通过)，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MicroSD卡文件系统实验(FATFS R0.09)
disk_initialize starting.....
文件已经存在
感谢您选用 野火STM32开发板 ！ ^_^
已连接 0:00:21 自动检测 115200 8-N-1 SCROLL CAPS NUM 捕 打印