1.1 STM32 之 FATFS 文件系统

1.1.1 学习目的

1. 了解文件系统概念
2. 掌握文件系统的移植
3. 掌握通过文件系统对 SD 卡的访问

1.1.2 FATFS 介绍

（1） FATFS 概述

在上一章节中，我们已经介绍了 SD 卡的驱动。 SD_WriteDisk 函数只是将缓冲区的数据写入到 SD 卡的某个地址，在需要的时候从该地址把数据读出来。 那么问题来了，假如对 SD 卡进行更多的操作， 比如难以确定存储介质的剩余空间，或者说用什么格式来解读数据。就像是给你个大仓库， 东西都是随便放的，塞满就行。这样感觉就会很乱。

此时我们就需要用到某些方法来对仓库内的物品进行管理。而这个方法就是文件系统。 它是为了存储和管理数据，而在存储介质建立的一种组织结构，这些结构包括操作系统引导区、目录和文件。

常用的 windows 下的文件系统格式包括 FAT32、NTFS、 exFAT。在使用文件系统前，要先对存储介质进行格式化，格式化之后， 在存储进制会创建一个文件分配表和目录。此时，文件系统就可以记录数据存放的物理地址，剩余空间。具体来说，文件系统负责为用户建立文件、 存入、 读出、 修改、 存储文件、 控制文件的存取，还有针对不使用的撤销文件。

FatFs 是一种完全免费开源的 FAT 文件系统模块，用于小型的嵌入式系统中实现 FAT 文件系统。 FatFS 的编写遵循 ANSI C， 因此不依赖于硬件平台。可以移植到 8051、 PIC、 AVR、 SH、 Z80、 H8、 ARM 等微型控制器中。它支持 FATl2、 FATl6 和 FAT32，支持多个存储媒介；有独立的缓冲区，可以对多个文件进行读写，并特别对 8 位单片机和 16 位单片机进行优化。

（2） FATFS 框图

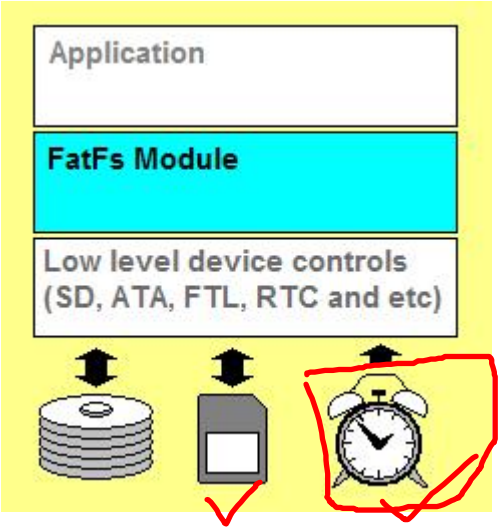


图 1.4.1 FATFS 框图

从图中可以看出：FATFS 主要由应用层、FSTFS 模块、底层存储媒介接口等组成。

最顶层为应用层，作为用户，我们是不需要考虑这一层的结构及组成。我们只需要调用 FatFs 提供给我们的一系列接口函数，如 f_open，f_read，f_write 等函数，通过函数名我们就知道是一些读写操作。

中间层是 FatFS 模块，实现了 FAT 文件读/写协议，一般不用修改，使用时直接将头文件包含即可。

需要我们移植、编写代码的是 FatFS 模块提供的底层接口，包括存储媒介读/写口和供给文件创建修改时间的实时时钟。

（3） FATFS 源码下载

下载地址：http://elm-chan.org/fsw/ff/00index_e.html。这里使用的是 R0.11a 版本。如图 1.4.2 所示：

图 1.4.2 FATFS 源码下载

FATFS 有两个版本，一个大版本，一个小版本。小版本主要用于 8 位机(内存小)使用。小版本下载如图 1.4.3 所示：

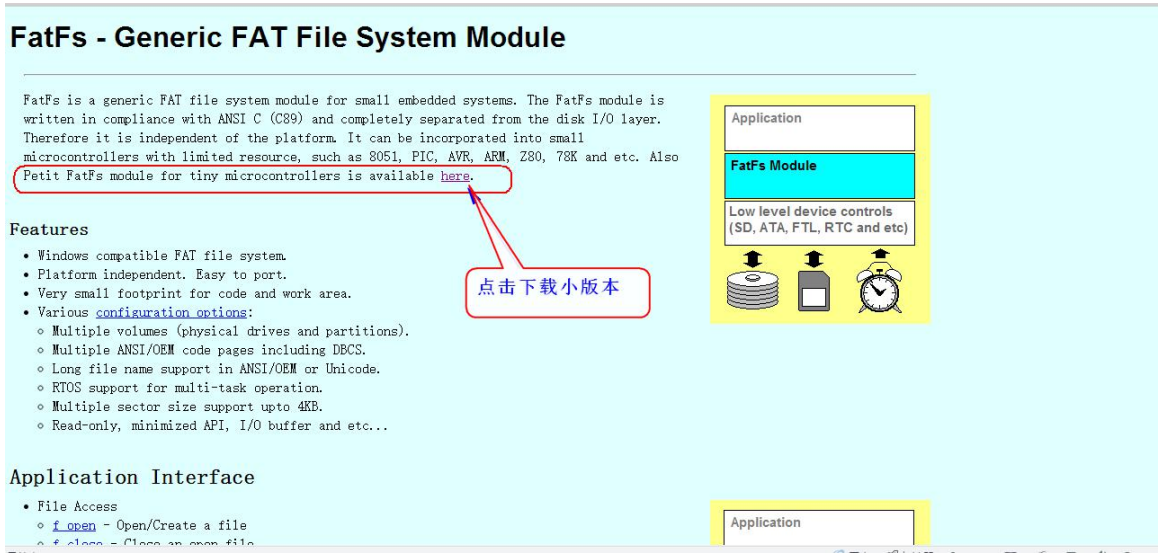


图 1.4.3 FATFS 源码小版本下载入口

（4） FATFS 源码文件介绍

将之前下载的源码解压后可以得到两个文件夹：doc 和 src。其中 doc 文件夹主要是对 FATFS 的介绍(离线文档—英文和日文)，而 src 文件夹才是 FATFS 的源码。

其中，与平台相关的代码：

- diskio.c 底层接口文件（需要用户修改）

与平台无关的是：

- ffconf.h FATFS 配置文件
- ff.h 应用层头文件
- ff.c 应用层源文件
- diskio.h 硬件层头文件
- interger.h 数据类型定义头文件
- option 可选的外部功能（比如支持中文等）

FATFS 模块在移植的时候，一般只需要修改 diskio.c 和 ffconf.h 这两个文件即可。FATFS 模块的所有配置选项都是存放在 ffconf.h 中，我们可以通过配置里面的某些选项来满足设计的需求。

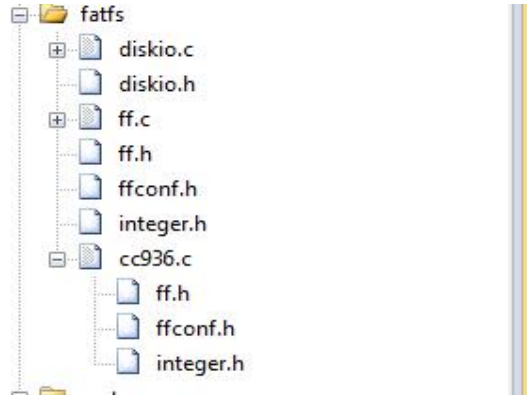
1.1.3 FATFS 的移植

（1） FATFS 移植准备

本文档移植的开发环境与芯片型号:
目标芯片：STM32F407ZGT6
编译软件：KEIL5.14
移植文件系统之前，需要有一个完整的工程，并且有完整的 SD 卡驱动。
还需要准备 SD 一张。

（2） FATFS 移植过程

1．将 FATFS 源文件添加到工程。把之前下载源文件中 src 文件夹的 ff.c 、diskio.c 和 option 目录下的 cc936.c 这几个文件添加到工程中。并把其头文件路径也添加到工程中。如图 1.4.4 所示和图 1.4.5 所示



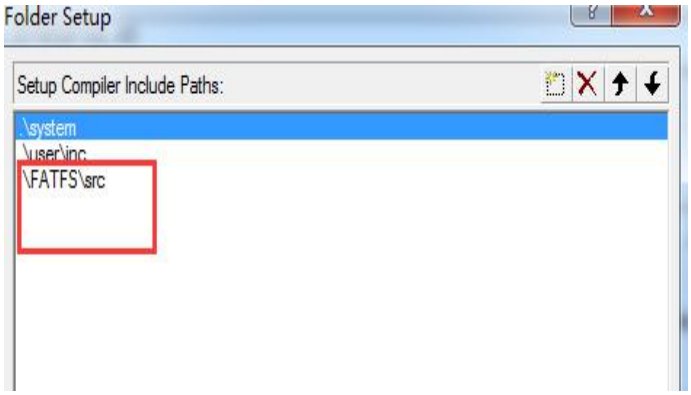


图 1.4.4 添加到工程中的 FATFS 源文件　　图 1.4.5 添加 FATFS 头文件路径

2．修改 diskio.c 文件中的相关函数。

注释掉现在不需要的用到的文件，因为我们现在用的是 SD 卡，与 USB，ATA，MMC 卡没关系。并加入一个新的宏 ：

#define SD 0

定义 SD 卡的物理驱动器号为 0。如图 9.4.6 所示：

```
#include "diskio.h"    /* FatFs lower layer API */
//#include "usbdisk.h"  /* Example: Header file of existing USB MSD control module */
//#include "atadrive.h" /* Example: Header file of existing ATA harddisk control module */
//#include "sdcard.h"   /* Example: Header file of existing MMC/SDC contorl module */
#include "sd.h"
/* Definitions of physical drive number for each drive */
//#define ATA   0 /* Example: Map ATA harddisk to physical drive 0 */
//#define MMC   1 /* Example: Map MMC/SD card to physical drive 1 */
//#define USB   2 /* Example: Map USB MSD to physical drive 2 */
#define SD    0 /* Example: SD harddisk to physical drive 0 */
```

图 1.4.7 更改 SD 卡驱动需要的宏定义

需要修改的函数有以下几个：

1) disk_status 函数。函数描述如下：

函数名	disk_status
函数原型	DSTATUS disk_status (BYTE pdrv)
函数功能	返回当前驱动器的状态
函数参数	驱动器编号
函数返回值	RES_OK：初始化成功 RES_ERROR：读写错误 RES_WRPRT：当前驱动器写保护 RES_NOTRDY：当前驱动器未初始化成功

	RES_PARERR：无效的参数

如果不需要获取驱动器的状态，可以直接把此函数的功能返回 RES_OK。

```
/*-----------------------------------------------------------------------*/
/* Get Drive Status                                                      */
/*-----------------------------------------------------------------------*/
DSTATUS disk_status (
    BYTE pdrv           /* Physical drive nmuber to identify the drive */
)
{
    switch (pdrv) {
    case SD :
        return RES_OK;
    }
    return STA_NOINIT;
}
```

2) disk_initialize 函数。函数描述如下：

函数名	disk_initialize
函数原型	DSTATUS disk_initialize (BYTE pdrv)
函数功能	初始化磁盘驱动器
函数参数	驱动器编号
函数返回值	RES_OK：初始化成功 RES_ERROR：读写错误 RES_WRPRT：当前驱动器写保护 RES_NOTRDY：当前驱动器未初始化成功 RES_PARERR：无效的参数

我们现在只使用 SD 卡，所以在这个函数中添加我们上一章节讲述的 SD 卡驱动函数。其他的卡可以不添加。

```
/*-----------------------------------------------------------------------*/
/* Inidialize a Drive                                                    */
/*-----------------------------------------------------------------------*/

DSTATUS disk_initialize (
    BYTE pdrv                   /* Physical drive nmuber to identify the drive */
)
{
    DSTATUS stat;

    switch (pdrv) {
    case SD :
        stat = SD_Init();
        return stat;
    }
    return STA_NOINIT;
}
```

3) disk_read 函数。函数描述如下：

函数名	disk_read
函数原型	DRESULT disk_read (BYTE pdrv, BYTE *buff, DWORD sector, UINT count)

函数功能	从磁盘驱动器中读取扇区
函数参数	pdrv： 驱动器编号 buff： 读取数据缓冲区 sector：扇区地址 count：需要读取的扇区数
函数返回值	RES_OK：初始化成功 RES_ERROR：读写错误 RES_WRPRT：当前驱动器写保护 RES_NOTRDY：当前驱动器未初始化成功 RES_PARERR：无效的参数

我们现在只使用 SD 卡，所以在这个函数中添加我们上一章节讲述的 SD 卡读取数据函数。其他的卡可以不添加。

```
/*-----------------------------------------------------------------------*/
/* Read Sector(s)                                                        */
/*-----------------------------------------------------------------------*/

DRESULT disk_read (
	BYTE pdrv,		/* Physical drive nmuber to identify the drive */
	BYTE *buff,		/* Data buffer to store read data */
	DWORD sector,	/* Sector address in LBA */
	UINT count		/* Number of sectors to read */
)
{
	DRESULT res;
	switch (pdrv) {
	case SD :
		res = (DRESULT)SD_ReadDisk(buff,sector,count);
		return res;
	}

	return RES_PARERR;
}
```

4) disk_write 函数。函数描述如下：

函数名	disk_write
函数原型	DRESULT disk_write (BYTE pdrv, const BYTE *buff, DWORD sector, UINT count)
函数功能	从磁盘驱动器中写入扇区
函数参数	pdrv： 驱动器编号 buff： 数据写入缓冲区 sector：扇区地址 count：需要读取的扇区数
函数返回值	RES_OK：初始化成功 RES_ERROR：读写错误 RES_WRPRT：当前驱动器写保护 RES_NOTRDY：当前驱动器未初始化成功 RES_PARERR：无效的参数

我们现在只使用 SD 卡，所以在这个函数中添加我们上一章节讲述的 SD 卡写入数据函数。其他的卡可以不添加。

```
/*-----------------------------------------------------------------------*/
/* Write Sector(s)                                                       */
```

```
/*-----------------------------------------------------------------------*/

#if _USE_WRITE
DRESULT disk_write (
	BYTE pdrv,		/* Physical drive nmuber to identify the drive */
	const BYTE *buff,	/* Data to be written */
	DWORD sector,		/* Sector address in LBA */
	UINT count		/* Number of sectors to write */
)
{
	DRESULT res;
	switch (pdrv) {
	case SD :
		res =(DRESULT)SD_WriteDisk((u8 *)buff,sector,count);
		while(res!=0)
		{
		  res =(DRESULT)SD_WriteDisk((u8 *)buff,sector,count);
		}
		return res;
	}
	return RES_PARERR;
}
#endif
```

5) disk_ioctl 函数。函数描述如下：

函数名	disk_ioctl
函数原型	DRESULT disk_ioctl (BYTE pdrv,　BYTE cmd, void *buff)
函数功能	控制磁盘驱动器指定特性和除了读写之外的其他功能
函数参数	pdrv：　驱动器编号 cmd:　 控制命令 buff：　发送/接收数据缓冲区地址
函数返回值	RES_OK：初始化成功 RES_ERROR：读写错误 RES_WRPRT：当前驱动器写保护 RES_NOTRDY：当前驱动器未初始化成功 RES_PARERR：无效的参数

此函数主要用于获取卡的数据块大小，卡的扇区总数及卡容量等信息。我们现在只使用 SD 卡，所以在这个函数中添加我们上一章节讲述的获取 SD 卡相关信息函数。其他的卡可以不添加。

```
/*-----------------------------------------------------------------------*/
/* Miscellaneous Functions                                               */
/*-----------------------------------------------------------------------*/

#if _USE_IOCTL
DRESULT disk_ioctl (
	BYTE pdrv,		/* Physical drive nmuber (0..) */
	BYTE cmd,		/* Control code */
	void *buff		/* Buffer to send/receive control data */
)
{
```

```
    DRESULT res;
    switch (pdrv) {
    case SD :
        switch(cmd)
        {
            case CTRL_SYNC:
                res = RES_OK;
                break;
            case GET_SECTOR_SIZE:
                *(DWORD*)buff = 512;
                res = RES_OK;
                break;
            case GET_BLOCK_SIZE:
                *(WORD*)buff = SDCardInfo.CardBlockSize;
                res = RES_OK;
                break;
            case GET_SECTOR_COUNT:
                *(DWORD*)buff = SDCardInfo.CardCapacity/512;
                res = RES_OK;
                break;
            default:
                res = RES_PARERR;
                break;
        }
            return res;
    }
    return RES_PARERR;
}
#endif
```

3．在 diskio.c 文件中添加以下几个功能函数

1) get_fattime 函数。函数描述如下：

函数名	get_fattime
函数原型	DWORD get_fattime (void)
函数功能	获取当前时间
函数参数	无
函数返回值	当前时间 位[31：25] 年 位[24：21] 月 位[20：16] 日 位[15：11] 时 位[10：5] 分 位[4： 0] 秒

此函数是用于获取当前时间的，如果想实现此函数功能，就必须要在工程中实现 RTC 功能。如果工程中没有 RTC 功能，可以直接返回 0 即可。

```
/*-----------------------------------------------------------------------*
*  函数名： DWORD get_fattime (void)
*  功能描述: 获取当前时间
*  输入参数: 无
```

```
*   输出参数: 无
*   返回值:   当前时间
*   其他:
*   作者:
*****************************************************************************/
DWORD get_fattime (void)
{
      u32 date;
      get_rtc_time(&time_date);
      date =
      (
      ((time_date.year+2000) << 25) |
      (time_date.month << 21 )          |
      (time_date.date << 16 )           |
      (time_date.hour << 11 )                   |
      (time_date.min << 5 )                     |
      ( time_date.sec )
      );
      return date;
}
```

2) ff_memalloc 函数，此函数是用于申请内存大小的。跟 C 语言中 malloc 函数功能一样。

```
/*****************************************************************
*   函数名：  void* ff_memalloc (UINT msize)
*   功能描述: 申请内存空间
*   输入参数: 申请内存空间的大小（字节）
*   输出参数: 无
*   返回值:   申请到空间的首地址
*   其他:
*   作者:
*****************************************************************************/
void* ff_memalloc (UINT msize)
{
      return malloc(msize);
}
```

3) ff_memfree 函数，此函数是用于申请内存大小的。跟 C 语言中 free 函数功能一样。

```
/*****************************************************************
*   函数名：  void* ff_memfree (void* mblock)
*   功能描述: 释放内存空间
*   输入参数: 释放内存空间的首地址
*   输出参数: 无
*   返回值:   无
*   其他:
*   作者:
*****************************************************************************/
void ff_memfree (void* mblock)
{
      free(mblock);
}
```

注意：ff_memalloc 和 ff_memfree 函数需要包含 stdlib.h 这个头文件。

4．修改 ffconf.h 文件

FATFS 模块的所有配置项都是存放在 ffconf.h 里面，我们可以根据自己的需求来进行更改设置。接下来我们介绍几个比较重要的配置选项。

1) _FS_READONLY。只读选项。用来配置是不是只读，0 为可读可写，1 为只读。
2) _USE_STRFUNC。用来配置是否支持字符串的操作，0 为不支持，1 为支持。
3) _USE_MKFS。用来配置是否支持格式化，0 为不支持，1 为支持。
4) _USE_FASTSEEK。用来配置是否支持快速定位，0 为不支持，1 为支持。
5) _USE_LABEL。用来配置是否支持磁盘盘符（磁盘名字）读取与设置，0 为不支持，1 为支持。
6) _CODE_PAGE。用来配置语言的种类，这里设置为 936，即使用中文简体。需要把 FATFS 源文件中的 cc936.c 文件添加到工程中才能使用。
7) _USE_LFN。用来配置是否支持长文件名。取值范围为 0~3。0 为不支持长文件名，1~3 为支持长文件名，但是存储地方不一样，这里设置为 3，可以通过 ff_memalloc 函数来动态分配长文件名的存储区域。
8) _VOLUMES。用来配置支持多少个设备，我们这里只使用 SD 卡一个设备，所以这里设置为 1。

5．修改堆栈空间

因为长文件支持需要占用堆空间。所以我们需要修改工程中的堆栈空间，堆栈空间在启动文件中修改，如图 1.4.8 所示：

图 1.4.8 修改堆栈空间的大小

1.1.4 FATFS 硬件设计

FATFS 硬件设计使用的 SD 卡相关硬件。

1.1.5 FATFS 软件设计

（1） FATFS 软件设计思想

1．SD 卡初始化。本实验中使用 SD 卡来实现文件系统的思想，所以需要使用到 SD 卡的初始化。
2．编写 SD 卡与 FATFS 文件系统桥接函数，即 FATFS 移植部分函数。
3．使用 SD 卡前挂载文件系统。

4．调用函数实现 FATFS 文件系统的测试。

（2） 完全程序分析

FATFS 文件系统的读写功能都在主函数中实现。

```
/*********************************************************************
*   函数名：  int main ()
*   功能描述: 实现 FATFS 文件系统的读写功能
*   输入参数: 无
*   输出参数: 无
*   返回值:    无
*   其他:
*   作者:
*********************************************************************/
int main ()
{
    FATFS fp;
    FIL   file;
    BYTE write_buff[]="实训改变命运,技能改变中国";
    BYTE read_buff[sizeof(write_buff)];
    u8 res;
    UINT bw;
    UINT br;
    USART1_Init(9600);//串口初始化，波特率为 9600
    Delay_Init();        //延时初始化
    SD_Init();            //SD 卡初始化
  // 注册工作区，驱动器号 0
    res=f_mount (& fp, "0", 1);
    if(res==FR_OK)
    {
        printf("成功注册工作区\r\n");
    }
    else
    {
        printf("注册失败\r\n");
    }
    //调用打开文件函数，权限，可读可写，如果文件存在则覆盖
     res = f_open(&file, "0:/xyd.txt", FA_READ|FA_WRITE|FA_OPEN_ALWAYS);
     if(res==FR_OK)
     {
            printf("文件创建成功！！ \r\n");
     }
        else
        {
            printf("文件创建失败！！ \r\n");
        }

    //写入数据
    res =f_write(&file,write_buff,strlen((const char*)write_buff),&bw);
    if(res==FR_OK)
    {
```

```
        printf("成功写入%d 个字节数据！！ \r\n",bw);
    }
    else
    {
        printf("数据写入失败！！ \r\n");
    }

    //以只读的方式打开刚才创建的文件
    res = f_open(&file, "0:/xyd.txt", FA_READ);
    if(res==FR_OK)
    {
        printf("成功打开文件！！ \r\n");
    }
    else
    {
        printf("打开文件失败\r\n");
    }
    //读取打开文件的内容,并通过串口显示读到的内容
    res=f_read(&file,read_buff,strlen((const char*)write_buff),&br);
    if(res==FR_OK)
    {
        printf("成功读取%d 个字节数据！！ \r\n",br);
        printf("读到的数据为: %s\r\n",read_buff);
    }
    else
    {
        printf("读取失败\r\n");
    }
    //关闭文件
    res=f_close(&file);
    if(res==FR_OK)
    {
        printf("成功关闭\r\n");
    }
    else
    {
        printf("关闭失败\r\n");
    }
    while(1)
    {
        ;
    }
}
```

函数说明如下：

1．f_mount 函数。函数功能注册驱动器，只有注册后才能使用。我们移植时 SD 卡的驱动号设置为 0。

2．f_open 函数。函数功能为打开文件。第一次调用此函数时，传的参数为 FA_READ、FA_WRITE、FA_OPEN_ALWAYS，设置此文件为具有可读可写功能，并且如果没有此文件则创建一个。第二次调用时以只读的方式打开文件。

3．f_write 函数。函数功能为写数据。在之前打开的文件中写入数据。

4．f_read 函数。函数功能为读数据。在调用此函数之前必须先调用 f_write 打开文件。

5．f_close 函数。函数功能为关闭文件。

1.1.6 FATFS 程序仿真，下载，测试

图 1.4.9 FATFS 读写数据