目　录

第1章 FatFs 使用说明

1.1 简介

随着信息技术的发展，当今社会的信息量越来越大，以往由单片机构成的系统简单地对存储媒介按地址、按字节的读 / 写已经不能满足人们实际应用的需要，于是利用文件系统对存储媒介进行管理成了今后单片机系统的一个发展方向。目前常用的文件系统主要有微软的 FATl2、FATl6、FAT32、NTFS 以及 Linux 系统下的 EXT2 和 EXT3 等。由于微软 Windows 的广泛应用，在当前的消费类电子产品中，用得最多的还是 FAT 文件系统，如 U 盘、MP3、MP4 和数码相机等，所以找到一款容易移植和使用、占用硬件资源相对较小而功能又强大的 FAT 开源文件系统，对于单片机系统设计者来说是很重要的。

FatFs Module 是一种完全免费开源的 FAT 文件系统模块，专门为小型的嵌入式系统而设计。它完全用标准 C 语言编写，且完全独立于 I/O 层，可以移植到 8051、PIC、AVR、SH、Z80、H8 和 ARM 等系列单片机上且只需做简单的修改。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。

FatFs Module 有个简化版本 Tiny-FatFs，它跟完全版 FatFs 不同之处主要有两点：

(1) 占用内存更少，只要 1 KB RAM；

(2) 1 次仅支持 1 个存储介质。

完全版 FatFs 和 Tiny-FatFs 的用法一样，仅仅是包含不同的头文件，本文主要以完全版讲解 FatFs 的使用。

1.2 特性

- Windows 兼容的 FAT 文件系统；
- 平台无关，容易移植；
- 代码量小；
- 多种配置选项：
  - ✧ 支持多卷（物理驱动器或分区）；
  - ✧ 多个 ANSI/OEM 代码页包括 DBCS；
  - ✧ 支持长文件名，ANSI/OEM 或 Unicode；
  - ✧ 支持 RTOS；
  - ✧ 支持多种扇区大小；
  - ✧ 只读、最小化的 API 和 I/O 缓冲区等。

1.3 应用

FatFs Module 一开始就是为了能在不同的单片机上使用而设计的，所以具有良好的层次结构，如图 1.1 所示。

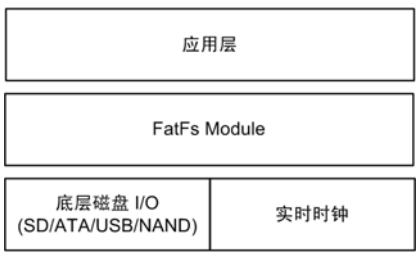

图 1.1 FatFs 模块层次结构图

最顶层是应用层，使用者无需理会 FatFs Module 的内部结构和复杂的 FAT 协议，只需要调用 FatFs Module 提供给用户的一系列应用接口函数，如 f_open，f_read，f_write 和 f_close 等，就可以像在 PC 上读／写文件那样简单。

中间层 FatFs Module 实现了 FAT 文件读／写协议。FatFs Module 的完全版提供的是 ff.c、ff.h，简化版 Tiny-FatFs 提供的是 tff.c、tff.h。除非有必要，使用者一般不用修改，使用时将需要版本的头文件直接包含进去即可。

需要使用者编写移植代码的是 FatFs Module 提供的底层接口，它包括存储媒介读／写接口 Disk I/O 和供给文件创建修改时间的实时时钟。

本文讲解时移植硬件平台为 ZLG 公司的 SmartCortexM3-1700 和普通 U 盘。LPC1768 是一款 32 位 Cortex-M3 内核的单片机，具有多达 64 KB 的 SRAM、512 KB 的内部 Flash 和丰富的外设。软件平台是 Keil 集成开发环境。

1.3.1 FatFs 软件包中相关文件

1. 平台无关

ffconf.h	FatFs 模块配置文件
ff.h	FatFs 和应用模块公用的包含文件
ff.c	FatFs 模块
diskio.h	FatFs and disk I/O 模块公用的包含文件
integer.h	数据类型定义
option	可选的外部功能

2. 平台相关（不属于 FatFs 需要由用户提供）

diskio.c	FatFs 与 disk I/O 模块接口层文件

1.3.2 FatFs 应用范围

- 支持 FAT12、FAT16 和 FAT32；
- 可打开的文件：无限制，依赖于有效的存储器；
- 支持最多 10 个卷；
- 文件大小：与 FAT 类型有关(upto 4G-1 bytes)；
- 卷大小：与 FAT 类型有关(upto 2T bytes on 512 bytes/sector)；
- 簇大小：与 FAT 类型有关(upto 64K bytes on 512 bytes/sector)；
- 扇区大小：与 FAT 类型有关(upto 4K bytes)。

FatFs 模块在移植时需先注意以下两点：

- ANSI C

FatFs 模块是用 ANSI C 编写的中间件，只要编译器遵循 ANSI C，它都是平台无关的。

- 整型大小

FatFs 假定 char/short/long 的长度为 8/16/32 位，而 int 为 16 位或 32 位，这些相应的定义位于 integer.h 文件。这在大多数的编译器上都不会是问题，但是当与预定义的内容发生冲突时，需要用户注意。

1.3.3 FatFs 配置

文件系统的配置项都在 ffconf.h 文件之中。

(1) _FS_TINY ：这个选项在 R0.07 版本之中开始出现，在之前的版本都是以独立的 C

文件出现，现在通过一个宏来修改使用起来更方便；

(2) _FS_MINIMIZE、_FS_READONLY、_USE_STRFUNC、_USE_MKFS、_USE_FORWARD 这些宏是用来对文件系统进行裁剪的，下面的 1.3.4 小节中有详细介绍；

(3) _CODE_PAGE ：本选项用于设置语言码的类型，对应的字库可以在网上下载，其选项如下：

- 932 - Japanese Shift-JIS (DBCS, OEM, Windows)
- 936 - Simplified Chinese GBK (DBCS, OEM, Windows)
- 949 - Korean (DBCS, OEM, Windows)
- 950 - Traditional Chinese Big5 (DBCS, OEM, Windows)
- 1250 - Central Europe (Windows)
- 1251 - Cyrillic (Windows)
- 1252 - Latin 1 (Windows)
- 1253 - Greek (Windows)
- 1254 - Turkish (Windows)
- 1255 - Hebrew (Windows)
- 1256 - Arabic (Windows)
- 1257 - Baltic (Windows)
- 1258 - Vietnam (OEM, Windows)
- 437 - U.S. (OEM)
- 720 - Arabic (OEM)
- 737 - Greek (OEM)
- 775 - Baltic (OEM)
- 850 - Multilingual Latin 1 (OEM)
- 858 - Multilingual Latin 1 + Euro (OEM)
- 852 - Latin 2 (OEM)
- 855 - Cyrillic (OEM)
- 866 - Russian (OEM)
- 857 - Turkish (OEM)
- 862 - Hebrew (OEM)
- 874 - Thai (OEM, Windows)
- ASCII only (Valid for non LFN cfg.)

(4) _USE_LFN ： 取值为 0~3，主要用于长文件名的支持及缓冲区的动态分配：

- 0：不支持长文件名；
- 1：支持长文件名存储的静态分配，一般是存储在 BSS 段；
- 2：支持长文件名存储的动态分配，存储在栈上；
- 3：支持长文件名存储的动态分配，存储在堆上。

(5) _MAX_LFN ： 可存储长文件的最大长度，其值一般为（12~255），但是缓冲区一般占(_MAX_LFN + 1) * 2 bytes；

(6) _LFN_UNICODE ： 为 1 时才支持 unicode 码；

(7) _FS_RPATH ： R0.08a 版本改动配置项，取值范围 0~2：

- 0：去除相对路径支持和函数；
- 1：开启相对路径并且开启 f_chdrive()和 f_chdir()两个函数；
- 2：在 1 的基础上添加 f_getcwd()函数。

(8) _VOLUMES ：支持的逻辑设备数目；

(9) _MAX_SS ：扇区缓冲的最大值，其值一般为 512；

(10) _MULTI_PARTITION：定义为 1 时，支持磁盘多个分区；

(11) _USE_ERASE ：R0.08a 新加入的配置项，设置为 1 时，支持扇区擦除；

(12) _WORD_ACCESS ：如果定义为 1，则可以使用 word 访问；

(13) _FS_REENTRANT ：定义为 1 时，文件系统支持重入，但是需要加上跟操作系统信号量相关的几个函数，函数在 syscall.c 文件中；

(14) _FS_SHARE ：文件支持的共享数目。

FatFs 只要求提供 FatFs 模块所必需的底层磁盘 I/O 函数，如果存在一个可工作的目标磁盘模块，你仅需将编写的新函数附加到 FatFs 模块上，如果没有，则需要提供其它磁盘模块或者从头编写底层驱动。FatFs 中所有定义的函数并不总是必需的，例如，在只读配置模式下，磁盘写函数是不需要的。表 1.1 显示了 FatFs 的函数需要依赖于配置选项。

表 1.1 函数依赖配置选项

函数	要求	注意事项
disk_initialize	Always	磁盘 I/O 函数
disk_status	Always	
disk_read	Always	
disk_write	_FS_READONLY == 0	
disk_ioctl (CTRL_SYNC)	_FS_READONLY == 0	
disk_ioctl (GET_SECTOR_COUNT)	_USE_MKFS == 1	
disk_ioctl (GET_SECTOR_SIZE)	_MAX_SS >= 1024	
disk_ioctl (GET_BLOCK_SIZE)	_USE_MKFS == 1	
disk_ioctl (CTRL_ERASE_SECTOR)	_USE_ERASE == 1	
get_fattime	_FS_READONLY == 0	
ff_convert	_USE_LFN >= 1	Unicode 支持函数 有效文件位于 option/cc*.c.
ff_wtoupper	_USE_LFN >= 1	
ff_cre_syncobj	_FS_REENTRANT == 1	操作系统依赖函数 有效文件位于 option/sysca ll.c.
ff_del_syncobj	_FS_REENTRANT == 1	
ff_req_grant	_FS_REENTRANT == 1	
ff_rel_grant	_FS_REENTRANT == 1	
ff_mem_alloc	_USE_LFN == 3	
ff_mem_free	_USE_LFN == 3	

1.3.4 FatFs API 函数选择

表 1.2 显示了通过配置选项对 API 函数进行选择（剪裁）以减小模块代码大小。

表 1.2　API 函数选择

功能	_FS_MINIMIZE				_FS_READONLY		_USE_STRFUNC		_FS_RPATH			_USE_MKFS		_USE_FORWARD	
	0	1	2	3	0	1	0	1	0	1	2	0	1	0	1
f_mount															
f_open															
f_close															
f_read															
f_write						x									
f_sync						x									
f_lseek				x											
f_opendir			x	x											
f_readdir			x	x											
f_stat		x	x	x											
f_getfree		x	x	x		x									
f_truncate		x	x	x		x									
f_unlink		x	x	x		x									
f_mkdir		x	x	x		x									
f_chmod		x	x	x		x									
f_utime		x	x	x		x									
f_rename		x	x	x		x									
f_chdir									x						
f_chdrive									x						
f_getcwd									x	x					
f_mkfs						x						x			
f_forward														x	
f_putc						x	x								
f_puts						x	x								
f_printf						x	x								
f_gets							x								

1.3.5 路径名格式

FatFs 模块上使用的路径名格式类似于 DOS/Windows 下的文件名，格式如下：

"[drive#:][/]directory/file"

FatFs 模块支持长文件名（LFN）和 DOS 8.3 文件名（SFN）。当使能 LFN 特性时（_USE_LFN > 0），可以使用长文件名，子目录用‘\’或者‘/’隔开，这与 DOS/Windows API 函数接口是相同的方式，不同的是逻辑驱动器用数字带一个冒号来指定的。当一个驱动器号被忽略时，默认为驱动器 0 或者当前驱动器。控制字符（\0~\x1F）被认为是路径名的末尾。首位或中间嵌入的空格是合法的，当配置为 LFN 时被看作是名字的一部分，配置为非 LFN 时被视为路径名的末尾。空格和点号被忽略。

默认配置下（_FS_RPATH == 0），FatFs 没有一个像操作系统面向文件系统的当前目录的概念。卷上的所有对象总是以根目录下的完整路径名来指定，不允许点目录名。标题分隔符’/’被忽略，它可以存在或省略，默然的驱动器号为 0。

当使能相对路径时（_FS_RPATH == 1），如果存在标题分隔符，指定的路径是指相对于根目录，如果没有标题分隔符，指定目录是指相对于用 f_chdir 函数设置的当前目录，路径名中允许使用点名称。默认的驱动器是用 f_chdrive 函数设定的当前驱动器。

表 1.3　路径名格式

路径名	_FS_RPATH == 0	_FS_RPATH == 1
file.txt	驱动器 0 根目录下的一个文件	当前驱动器当前目录下的一个文件
/file.txt	驱动器 0 根目录下的一个文件	当前驱动器根目录下的一个文件
	驱动器 0 根目录	当前驱动器当前目录
/	驱动器 0 根目录	当前驱动器根目录
2:	驱动器 2 根目录	驱动器 2 当前目录
2:/	驱动器 2 根目录	驱动器 2 根目录
2:file.txt	驱动器 2 根目录下的一个文件	驱动器 2 当前目录下的一个文件
../file.txt	文件名非法	上级目录下的一个文件
.	文件名非法	此目录
..	文件名非法	当前目录的上级目录
dir1/..	文件名非法	当前目录
/..	文件名非法	根目录（）

1.3.6 关于长文件名

FatFs 从 0.07 版本开始支持长文件名（LFN）。在调用文件函数时，一个文件的两个文件名（SFN 与 LFN）是通用的，除了 f_readdir 函数。支持长文件特性将需要一个额外的工作缓冲区，此缓冲区的大小可以通过设置_MAX_LFN 来以可用的内存大小相符。因为长文件名可长达 255 个字符，因此_MAX_LFN 应该设置为 255 来支持全特性的 LFN 选项。当工作缓冲区的大小容不下给出的文件名时文件函数就会因为 FR_INVALID_NAME 而调用失败。

表 1.4　ARM7 上的 LFN 配置

编码页	程序大小
SBCS（单字节字符集）	+3.7K
932（日文 Shift-JIS）	+62K
936（简体中文 GBK）	+177K
949（韩文 Korean）	+139K
950（繁体中文 Big5）	+111K

当使能 LFN，模块增加的大小由编码页（Code Page）类型决定，表 1.4 显示了 LFN 禁用与使用某些编码来使能时模块的不同大小。日语、中文与韩国语拥有成千上万的字词，因需要一个巨大的 OEM-Unicode 双向转换表，模块的大小将大大的增大，如表 1.4 所示。

注：FAT 文件系统的 LFN 特性是微软公司的专利。当在商用产品上使用时，根据最终目的的不同可能需要获得微软的许可证。

1.3.7 重入

对不同卷的文件操作总是可以同时地工作，而与重入设置无关。而对于同一个卷的重入访问可以通过使能_FS_REENTRANT 选项。此时，在 ff.c 中的与平台相关的锁定函数必须为每个 RTOS 重新编写。如果一个文件函数调用时其访问的卷正被另一个线程使用，则此访问将阻塞直到该卷解锁。如果等待时间超过了_TIMEOUT 毫秒，则函数将因 FR_TIMEOUT 而终止。某些 RTOS 可能不支持超时操作。

f_mount 与 f_mkfs 函数是个例外，这些函数对于同一个卷不会重入。当使用这些函数时，其它线程必须关闭此卷中相应的文件，避免对此卷的访问。

注意此部分描述的是 FatFs 自身的重入，与底层磁盘 I/O 的重入无关。

1.3.8 执行有效的文件访问

为了在小型的嵌入式系统中得到优秀的读写效率，应用程序程序员需要可考虑 FatFs 究竟做了什么。磁盘中的数据是通过下面的方式来被 f_read 函数传送。

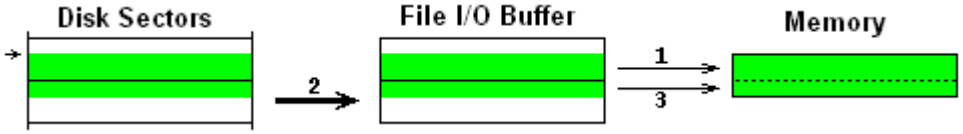

图 1.2　读非对齐扇区（短型）

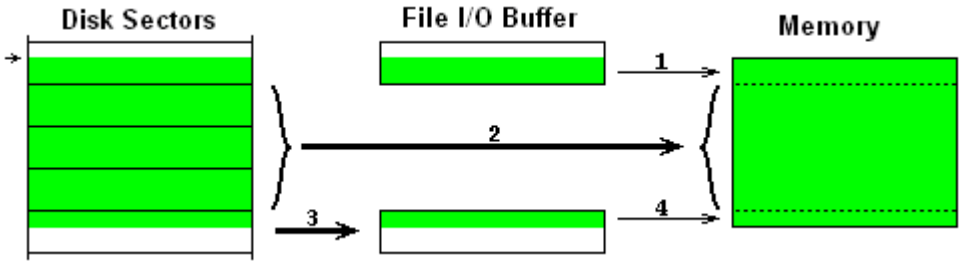

图 1.3　读非对齐扇区（长型）

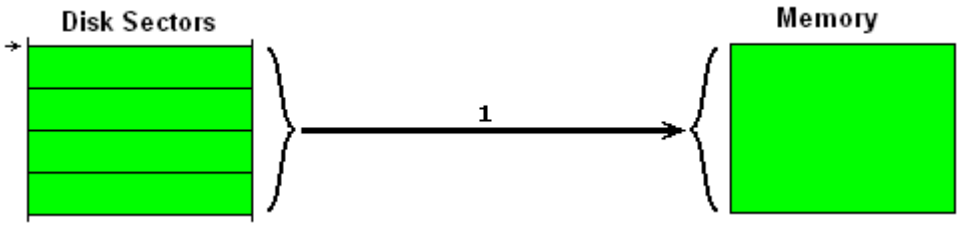

图 1.4　读对齐扇区

文件 I/O 缓冲区（file I/O buffer）表示一个将被读/写数据的扇区的缓冲区。扇区缓冲区可以是每个文件对象私有的，或者是文件系统共享的，缓冲区配置选项_FS_TINY 决定在文件数据传输中使用哪种扇区缓冲区。当选择小缓冲区（1），数据内存的使用量将降低到每个文件对象 512 字节。在这种情况下，FatFs 只使用一个扇区缓冲区来进行文件数据传输以及 FAT/目录访问。配置为小缓冲区的缺点是：每次文件数据传输时 FAT 数据缓冲都会丢失而必须从一个簇的边界开始重新载入数据。不过从其体面的表现与少内存消耗这方面来考虑，这对于多数的应用也是合适的。

图 1.2 显示扇区局部的数据通过文件 I/O 缓冲区来传输；图 1.3 显示的长数据传输，中间的一或多个扇区的数据直接传输到应用程序缓冲区；图 1.4 显示的是在以扇区对齐的整块数据传输的情况下，不使用文件 I/O 缓冲区。在直接传输时 disk_read 函数一次会最大程度的读取更多的扇区，但是多扇区传输不会跨越簇边界，即使它们是相邻的。

因此使用扇区对齐的方式来进行读写访问可以避免缓冲区数据传输，并且读写效率将被提升。除了效率以外，在 tiny 配置的情况下 FAT 快速缓存数据在文件数据传输时不会刷新，所以可以用小内存消耗来达到非 tiny 配置相同的性能。

1.3.9 临界区

当对 FAT 文件系统的写操作由于意外而中断，如突然断电，不正确的磁盘移除或不可恢复的磁盘错误，FAT 结构可能被毁坏。图 1.5 显示了 FatFs 的临界段。

```
f_mount(...);
f_open(...);          //Create  file

// any procedure
do {
    t = get_adc(...);

    // any procedure

    f_write(...);     // write file

    delay_second(1);
} while (...);
// any procedure
f_close(...);         // close file
```

```
f_mount(...);
f_open(...);          //Create  file
f_sync(...);
// any procedure
do {
    t = get_adc(...);

    // any procedure

    f_write(...);     // write file
    f_sync(...);
    delay_second(1);
} while (...);
// any procedure
f_close(...);         // close file
```

```
f_mkdir(...);
f_rename(...);
f_unlink(...);
```

```
f_mkdir(...);
f_rename(...);
f_unlink(...);
```

图 1.5　长临界区（左），最小化的临界区（右）

红色区域的中断会导致一个交叉链接，这可能导致正在修改的文件/目录丢失。而黄色区域中断可能导致的效果在下面列出：

- 正在重写的文件数据被毁坏；
- 正在添加内容的文件回到初始状态；
- 丢失新建的文件；
- 一个新建或覆盖的文件保持长度为 0；
- 因为丢失关联，磁盘的使用效率变坏。

若文件不是用写模式打开时，这些情况不会发生。为了最小化磁盘数据的丢失，临界区可以像图 1.5（右侧）显示的那样最小化，通过最小化文件处于写模式打开的时间或者适当的使用 f_sync 函数。

1.3.10　FatFs 的其它特性和新进展

(1)　支持并发操作；

在多任务操作系统中，各个任务是并发的。当它们要同时访问文件系统时，先要获得同步对象。比如在 uCOS 中，可以采用互斥信号量来同步。在 f_mount()时，创建同步对象，在 check_mount()和 validate()函数调用时，先申请同步对象，若是其它任务在使用文件系统，则在同步对象上等待。任务完成后，再释放同步对象。这个功能与操作系统的任务同步特性相关，以后如果要使用这个特性的话在详细分析。

(2) 支持文件的共享打开（主要是多次以读的方式打开）；

这是 FatFs 作者在 3 月份新上传的 R008 Rev1 版本里新增加的功能：在 ffconf.h 中增加了_FS_SHARE 共享数目定义。在 ff.h 增加了：文件共享信息结构体定义，并在文件系统结构体中使用，在 ff.c 中增加了 5 个函数。

(3) 支持文件的快速定位；

内存中划一块缓冲区用于存储 文件的簇链映射图，以方便查找簇链。稍微阅读了以下源代码，其实现方法是这样的：当给出参数 ofs == CREATE_LINKMAP 的时候，在 tbl 所指向的内存区域建立新的簇链映射。

表的第一项是整个映射表的长度，以后每两项为一对：前者存储相邻簇的数目，后者存储相邻簇的起始簇号。举个例子，某个文件占据 5、6、7、8、15、16、17 共 7 个簇，则需要两对表项。分别是（4，5）和（3，15）两项。以后就可以利用该簇链映射图实现文件的快速定位了，不用沿着簇链一项一项找下去，最后实现定位。

(4) 支持长文件名缓冲区的动态分配；

(5) 添加函数 f_getcwd()；

(6) 添加_USE_ERASE；

将以前的 auto_mount()改成了现在的 chk_mounted()添加部分功能使文件系统更适合多逻辑设备数目。

1.3.11 程序移植

移植 FatFs 主要分为三步：

(1) 数据类型：在 integer.h 里面去定义好数据的类型。这里需要了解你用的编译器的数据类型，并根据编译器定义好数据类型。

(2) 配置：打开 ffconf.h（我用的 FatFs，不是 Tiny，可以在此头文件中进行定义），文件系统的配置裁剪等均在此头文件中进行定义配置。

(3) 函数编写：打开 diskio.c，进行底层驱动编写，实际上需要编写 6 个接口函数，如图 1.6 所示。

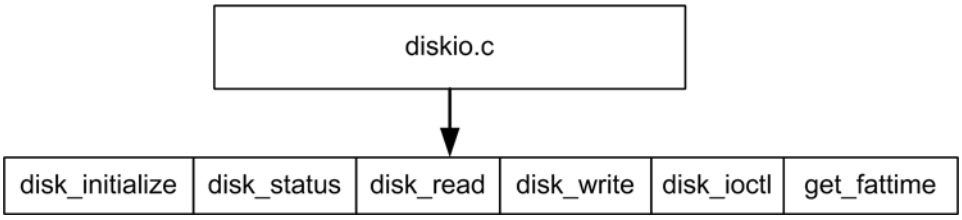

图 1.6 Disk I/O 函数结构图

1. 数据类型

LPC1768 是以 Cortex-M3 为内核的 32 位微处理器，支持的开发环境有 Keil 和 IAR 等，程序清单 1.1 为我们在 KEIL 下定义的数据类型。

程序清单 1.1　数据类型定义

```
/* 下面的类型必须为 16 位、32 位或更长的类型，Keil 下 32 位*/
typedef int                 INT;
typedef unsigned  int       UINT;

/* 下面的类型必须为 8 位 */
typedef char                CHAR;
typedef unsigned char       UCHAR;
typedef unsigned char       BYTE;

/* 下面的类型必须为 16 位 */
typedef short               SHORT;
typedef unsigned short      USHORT;
typedef unsigned short      WORD;
typedef unsigned short      WCHAR;

/* 下面的类型必须为 32 位 */
typedef long                LONG;
typedef unsigned long       ULONG;
typedef unsigned long       DWORD;
```

2. 配置

给出了 FatFs 移植到 uCOS-II 上支持 LPC1768 USB Host 读/写 U 盘，该配置支持长文件名。

```
/*---------------------------------------------------------------------------/
/  使能字符串功能
/---------------------------------------------------------------------------*/
#define     _USE_STRFUNC      1                                 /* 0:Disable or 1/2:Enable                  */
/*---------------------------------------------------------------------------/
/  code pages 设置，支持不同语言，当设置为 936 时可以支持中文，但是代码将增加 160KB 左右
/---------------------------------------------------------------------------*/
#define _CODE_PAGE              850                             /* 850   - Multilingual Latin 1 (OEM)     */
/*---------------------------------------------------------------------------/
/  选择是否支持长文件名
/---------------------------------------------------------------------------*/
#define     _USE_LFN            2                               /* 使用长文件名并以栈为工作区          */
/*---------------------------------------------------------------------------/
/  定义支持的最长文件名，_USE_LFN！=0 时有效
/---------------------------------------------------------------------------*/
#define     _MAX_LFN            255                             /* 最大的文件名长度(12 to 255)          */
/*---------------------------------------------------------------------------/
/  定义支持的卷数，最大 10 个
/---------------------------------------------------------------------------*/
#define _VOLUMES   1
```

```
/*-----------------------------------------------------------------------------------------------*
**  定义扇区大小，支持 512, 1024, 2048 或 4096
*-----------------------------------------------------------------------------------------------*/
#define     _MAX_SS             512
/*-----------------------------------------------------------------------------------------------*
**  定义是否支持重入，使用 uCOS-II 时需要定义
*-----------------------------------------------------------------------------------------------*/
#define _FS_REENTRANT       1
/*-----------------------------------------------------------------------------------------------*
**  定义超时周期，使用 uCOS-II 时需要定义
*-----------------------------------------------------------------------------------------------*/
#define _FS_TIMEOUT         1000
/*-----------------------------------------------------------------------------------------------*
**  定义 sync object.的类型，使用 uCOS-II 时需要定义
*-----------------------------------------------------------------------------------------------*/
#define     _SYNC_t             OS_EVENT*
```

3. 函数编写

因为 FatFs 模块完全与磁盘 I/O 层分开，因此需要下面的函数来实现底层物理磁盘的读写与获取当前时间。底层磁盘 I/O 模块并不是 FatFs 的一部分，并且必须由用户提供。下面的函数位于 diskio.c 中。

表 1.5　disk_initialize

函数名称	disk_initialize
函数原型	DSTATUS disk_initialize(BYTE Drive)
功能描述	初始化磁盘驱动器
函数参数	Drive：指定要初始化的逻辑驱动器号，即盘符，应当取值 0~9
返回值	函数返回一个磁盘状态作为结果，对于磁盘状态的细节信息，请参考 disk_status 函数
所在文件	ff.c
示例	disk_initialize(0);　　/* 初始化驱动器 0 */
注意事项	disk_initialize 函数初始化一个逻辑驱动器为读/写做准备，函数成功时，返回值的 STA_NOINIT 标志被清零； 应用程序不应调用此函数，否则卷上的 FAT 结构可能会损坏； 如果需要重新初始化文件系统，可使用 f_mount 函数； 在 FatFs 模块上卷注册处理时调用该函数可控制设备的改变； 此函数在 FatFs 挂在卷时调用，应用程序不应该在 FatFs 活动时使用此函数

程序清单 1.2 为 LPC1768 上实现 FatFs 应用于 USB 读写 U 盘而编写的 disk_initialize 函数。

程序清单 1.2　disk_initialize 函数实现

```
DSTATUS USBStat = STA_NOINIT;                   /* 磁盘状态                    */
DSTATUS disk_initialize (
    BYTE drv                                    /* 物理驱动器号                */
)
{
```

```
    DSTATUS   stat;
    INT8U     ucRet;
    USBStat   |= STA_NOINIT;

    switch (drv) {
    case ATA :
        return STA_NOINIT;
    case MMC :
        return STA_NOINIT;
    case USB :
        ucRet = msHostInit(0);                          /*  大容量类设备初始化                  */
        if (ucRet != 0x00) {
            return STA_NOINIT;
        } else {
            USBStat &= ~STA_NOINIT;
            return USBStat;
        }
    }
    return STA_NOINIT;
}
```

表 1.6　disk_status

函数名称	disk_status
函数原型	DSTATUS disk_status(BYTE Drive)
功能描述	返回当前磁盘驱动器的状态
函数参数	Drive：指定要确认的逻辑驱动器号，即盘符，应当取值 0~9
返回值	磁盘状态返回下列标志的组合，FatFs 只使用 STA_NOINIT 和 STA_PROTECTED STA_NOINIT：　表明磁盘驱动未初始化，下面列出了产生该标志置位或清零的原因： 置位：系统复位，磁盘被移除和磁盘初始化函数失败； 清零：磁盘初始化函数成功 STA_NODISK：　表明驱动器中没有设备，安装磁盘驱动器后总为 0 STA_PROTECTED：表明设备被写保护，不支持写保护的设备总为 0，当 STA_NODISK 置位时非法
所在文件	ff.c
示例	disk_status(0);　　/* 获取驱动器 0 的状态　　*/

程序清单 1.3 为 LPC1768 上实现 FatFs 应用于 USB 读写 U 盘而编写的 disk_status 函数。

程序清单 1.3　disk_status 函数实现

```
DSTATUS disk_status (
    BYTE drv                                        /*物理驱动器号                        */
)
{
    DSTATUS stat;
    switch (drv) {
    case ATA :
```

```
        return STA_NOINIT;
    case MMC :
        return STA_NOINIT;
    case USB :
        return USBStat;
    }
    return STA_NOINIT;
}
```

表 1.7　disk_read

函数名称	disk_read
函数原型	DRESULT disk_read(BYTE Drive, BYTE* Buffer, DWORD SectorNumber, BYTE SectorCount)
功能描述	从磁盘驱动器上读取扇区
函数参数	Drive：　指定逻辑驱动器号，即盘符，应当取值 0~9 Buffer：　指向存储读取数据字节数组的指针，需要为所读取字节数的大小，扇区统计的扇区大小是需要的 注：FatFs 指定的内存地址并不总是字对齐的，如果硬件不支持不对齐的数据传输，函数里需要进行处理 SectorNumber：指定起始扇区的逻辑块（LBA）上的地址 SectorCount：　指定要读取的扇区数，取值 1~128
返回值	RES_OK(0)：　函数成功 RES_ERROR：　读操作期间产生了任何错误且不能恢复它 RES_PARERR：　非法参数 RES_NOTRDY：　磁盘驱动器没有初始化
所在文件	ff.c

程序清单 1.4 为 LPC1768 上实现 FatFs 应用于 USB 读写 U 盘而编写的 disk_read 函数。

程序清单 1.4　disk_read 函数实现

```
DRESULT disk_read (
    BYTE drv,                   /* 物理驱动器号                    */
    BYTE *buff,                 /* 存储读取数据的数据缓冲区        */
    DWORD sector,               /* 起始扇区号                      */
    BYTE count                  /* 要读取的扇区数 (1..255)         */
)
{
    DRESULT  res;
    INT8U    ucRet;

    if (USBStat & STA_NOINIT) {
        return RES_NOTRDY;
    } else if (!count) {
        return RES_PARERR;
    }
```

```
    switch (drv) {
    case ATA :
        return RES_NOTRDY;
    case MMC :
        return RES_NOTRDY;
    case USB :
        ucRet = rbcRead10(0, 0, sector, count, buff);
        if (ucRet != 0x00) {
            return RES_ERROR;
        } else {
            return RES_OK;
        }
    }
    return RES_PARERR;
}
```

表 1.8　disk_write

函数名称	disk_write
函数原型	DRESULT disk_write(BYTE Drive, const BYTE* Buffer, DWORD SectorNumber, BYTE SectorCount)
功能描述	向磁盘写入一个或多个扇区
函数参数	Drive：　指定逻辑驱动器号，即盘符，应当取值 0~9 Buffer：指向要写入字节数组的指针， 注：FatFs 指定的内存地址并不总是字对齐的，如果硬件不支持不对齐的数据传输，函数里需要进行处理 SectorNumber：　指定起始扇区的逻辑块（LBA）上的地址 SectorNumber：　指定要写入的扇区数，取值 1~128
返回值	RES_OK(0)：　函数成功 RES_ERROR：　读操作期间产生了任何错误且不能恢复它 RES_WRPRT：　媒体被写保护 RES_PARERR：　非法参数 RES_NOTRDY：磁盘驱动器没有初始化
所在文件	ff.c
注意事项	只读配置中不需要此函数

程序清单 1.5 为 LPC1768 上实现 FatFs 应用于 USB 读写 U 盘而编写的 disk_write 函数。

程序清单 1.5　disk_write 函数实现

```
DRESULT disk_write (
    BYTE drv,                   /* Physical drive number                */
    const BYTE *buff,           /* Data to be written                   */
    DWORD sector,               /* Sector address (LBA)                 */
    BYTE count                  /* Number of sectors to write           */
)
{
```

```
    DRESULT  res;
    INT8U       ucRet;

    if (USBStat & STA_NOINIT) {
        return RES_NOTRDY;
    } else if (!count) {
        return RES_PARERR;
    }

    switch (drv) {
    case ATA :
        return RES_NOTRDY;
    case MMC :
        return RES_NOTRDY;
    case USB :
        ucRet =    rbcWrite10(0, 0, sector, count, (unsigned char *)buff);
        if (ucRet != 0x00) {
            return RES_ERROR;
        } else {
            return RES_OK;
        }
    }
    return RES_PARERR;
}
```

表 1.9　disk_ioctl

函数名称	disk_ioctl
函数原型	DRESULT disk_ioctl(BYTE Drive, BYTE Command, void* Buffer)
功能描述	控制设备指定特性和除了读/写外的杂项功能
函数参数	Drive：　指定逻辑驱动器号，即盘符，应当取值 0~9 Command：指定命令代码 Buffer：　指向参数缓冲区的指针，取决于命令代码，不使用时，指定一个 NULL 指针
返回值	RES_OK(0)：　函数成功 RES_ERROR：　读操作期间产生了任何错误且不能恢复它 RES_PARERR：非法参数 RES_NOTRDY：磁盘驱动器没有初始化
所在文件	ff.c
注意事项	CTRL_SYNC：　确保磁盘驱动器已经完成了写处理，当磁盘 I/O 有一个写回缓存，立即刷新原扇区，只读配置下不适用此命令 GET_SECTOR_SIZE：　返回磁盘的扇区大小，只用于 f_mkfs() GET_SECTOR_COUNT：返回可利用的扇区数, _MAX_SS >= 1024 时可用 GET_BLOCK_SIZE：　获取擦除块大小，只用于 f_mkfs() CTRL_ERASE_SECTOR：强制擦除一块的扇区，_USE_ERASE >0 时可用

程序清单 1.6 为 LPC1768 上实现 FatFs 应用于 USB 读写 U 盘而编写的 disk_ioctl 函数。

程序清单 1.6　disk_ioctl 函数实现

```
DRESULT disk_ioctl (
    BYTE drv,                                   /* Physical drive number                */
    BYTE ctrl,                                  /* Control code                         */
    void *buff                                  /* Buffer to send/receive control data  */
)
{
    DRESULT res;
    if (USBStat & STA_NOINIT) {
        return RES_NOTRDY;
    }
    switch (drv) {
    case ATA :
        return RES_NOTRDY;
    case MMC :
        return RES_NOTRDY;
    case USB :
        switch (ctrl) {
            case CTRL_SYNC:
                return RES_OK;
            default:
                return RES_PARERR;
        }
    }
    return RES_PARERR;
}
```

表 1.10　get_fattime

函数名称	get_fattime
函数原型	DWORD get_fattime()
功能描述	获取当前时间
函数参数	无
返回值	当前时间以双字值封装返回，位域如下： bit31:25　年　（0~127）（从 1980 开始） bit24:21　月　（1~12） bit20:16　日　（1~31） bit15:11　小时（0~23） bit10:5　分钟（0~59） bit4:0　秒　（0~29）
所在文件	ff.c
注意事项	get_fattime 函数必须返回一个合法的时间即使系统不支持实时时钟，如果返回 0，文件没有一个合法的时间； 只读配置下无需此函数

程序清单 1.7 为 LPC1768 上实现 FatFs 应用于 USB 读写 U 盘而编写的 get_fattime 函数，用户可以增添实时时钟功能来获取实时时间。

程序清单 1.7　get_fattime 函数实现

```
DWORD get_fattime (void)
{
    return      ((2010UL-1980) << 25)                   /* Year = 2010                */
                | (11UL << 21)                          /* Month = 11                 */
                | (2UL << 16)                           /* Day = 2                    */
                | (15U << 11)                           /* Hour = 15                  */
                | (0U << 5)                             /* Min = 0                    */
                | (0U >> 1)                             /* Sec = 0                    */
                ;
}
```

1.3.12　FatFs 软件包提供的 API 函数

FatFs 模块提供了以下功能的应用函数。

表 1.11　f_mount — 注册/注销一个工作区

函数名称	f_mount
函数原型	FRESULT f_mount(BYTE Drive, FATFS* FileSystemObject)
功能描述	为 FatFs 模块注册/注销一个工作区
函数参数	Drive：　　注册/注销工作区的逻辑驱动器号，即盘符，应当取值 0~9 FileSystemObject：指向注册的工作区（文件系统对象）的指针，为 NULL 时为注销工作区
返回值	FR_OK(0)　　函数成功 FR_INVALID_DRIVE　驱动器号无效
所在文件	ff.c
示例	FATFS fs;　　// 用户定义的文件系统结构体 f_mount(0, &fs);　　// 注册工作区，驱动器号 0，初始化后其他函数可使用里面的参数
注意事项	此函数的作用在磁盘里注册一个缓冲区域,用来存储 FAT32 文件系统的一些相关信息。 对磁盘进行操作之前,这个函数是不可少的。 无论驱动器状态如何，该函数总是成功的。函数只是初始化给定的工作区和注册内部表的地址，并没有进行媒体访问
快速信息	始终可用

表 1.12　f_open — 打开/创建一个文件

函数名称	f_open
函数原型	FRESULT f_ open(FIL* FileObject, const TCHAR* FileName, BYTE ModeFlags)

续上表

功能描述	为读写文件创建一个文件对象
函数参数	FileObject： 指向创建的文件对象结构体的指针 FileName： 指向空结尾字符串文件名的指针 ModeFlags：指定读写类型和打开文件的方式，可以是下列值一种或几种的组合： FA_READ // 读模式，从文件中读取数据（读写模式可同时生效） FA_WRITE // 写模式，往文件里写入数据（读写模式可同时生效） FA_OPEN_EXISTING // 打开文件（默认方式），如果文件不存在则函数失败 FA_OPEN_ALWAYS // 打开文件，如果文件不存在，则创建一个新文件。此种方式可使用 f_lseek 函数对打开的文件追加数据。 FA_CREATE_NEW // 创建一个新文件，如果文件存在则函数失败，返回值 FR_EXIST FA_CREATE_ALWAYS // 创建一个新文件，如果文件存在则覆盖旧文件
返回值	FR_OK(0)： 函数成功，文件对象合法 FR_NO_FILE： 无法找到文件 FR_NO_PATH： 无法找到路径 FR_INVALID_NAME： 文件名非法 FR_INVALID_DRIVE： 驱动器号非法 FR_EXIST： 文件已存在 FR_DENIED： 拒绝请求，要求的访问由于以下原因被拒绝： 对一个只读文件进行些操作； 文件不能创建由于一个目录或只读文件是存在的； 文件不能创建由于目录表已满。 FR_NOT_READY： 磁盘驱动器无法工作，由于驱动器中没有媒体或其他原因 FR_WRITE_PROTECTED：媒体写保护 FR_DISK_ERR： 函数失败由于磁盘运行的一个错误 FR_INT_ERR： 函数失败由于一个错误的 FAT 结构或内部错误 FR_NOT_ENABLED： 逻辑驱动器没有工作区 FR_NO_FILESYSTEM： 驱动器上没有合法的 FAT 卷 FR_LOCKED： 函数被拒由于文件共享机制（_FS_SHARE）
所在文件	ff.c
示例	FIL file; /* 文件对象 */ FRESULT res; /* FatFs 函数通用结果代码 */ …… /* 打开一个存在的文件，只读模式（不可追加数据） */ res = f_open(&file, "srcfile.txt", FA_OPEN_EXISTING | FA_READ); /* 打开一个存在的文件，读/写模式（可追加数据） */

续上表

	res = f_open(&file, "srcfile.txt", FA_OPEN_EXISTING | FA_READ | FA_WRITE);
	/* 打开一个文件（不存在则创建），读/写模式（可追加数据） */
	res = f_open(&file, "srcfile.txt", FA_OPEN_ALWAYS | FA_READ | FA_WRITE);

注意事项	函数成功时会创建一个文件对象，以供随后的读/写函数调用文件时使用； 使用 f_close 函数来关闭一个打开的文件对象，如果修改的文件没被关闭，文件数据将会丢失； 在使用任何文件函数之前，必须对相应的逻辑驱动器使用 f_mount 函数注册一个工作区（文件系统对象），所有的文件函数在注册工作区后才能正常工作
快速信息	始终可用。 当_FS_READONLY == 1 时模式标志 FA_WRITE,FA_CREATE_ALWAYS,FA_CREATE_NEW 和 FA_OPEN_ALWAYS 不可用

表 1.13 f_close — 关闭一个文件

函数名称	f_close
函数原型	FRESULT f_ close(FIL* FileObject)
功能描述	关闭一个打开的文件
函数参数	FileObject：指向要关闭的文件对象结构体的指针
返回值	FR_OK(0)： 文件对象应经成功关闭 FR_DISK_ERR： 函数失败由于磁盘运行的一个错误 FR_INT_ERR： 函数失败由于一个错误的 FAT 结构或内部错误 FR_NOT_READY： 磁盘驱动器无法工作，由于驱动器中没有媒体或其他原因 FR_INVALID_OBJECT：文件对象非法
所在文件	ff.c
示例	FIL file; /* 文件对象 */ …… f_close(&file); /* 关闭文件对象 */
注意事项	f_close 函数来关闭一个打开的文件对象，如果有任何数据写入文件，文件缓存信息将被写回到磁盘； 函数成功后，之前的文件对象不再合法而被丢弃
快速信息	始终可用

表 1.14 f_read — 读文件

函数名称	f_read
函数原型	FRESULT f_ read(FIL* FileObject, void* Buffer, UINT ByteToRead, UINT* ByteRead)

续上表

功能描述	从文件中读取数据
函数参数	FileObject： 指向打开文件对象结构体的指针 Buffer： 指向存储读取数据缓冲区的指针 ByteToRead：UINT 范围内要读取的字节数 ByteRead： 指向返回的已经读取的字节数 UINT 变量的指针，不管此函数掉调用后返回的结果如何，这个值始终有效
返回值	FR_OK(0)： 函数成功 FR_DENIED： 函数被拒由于文件已经开启了不可读模式 FR_DISK_ERR： 函数失败由于磁盘运行的一个错误 FR_INT_ERR： 函数失败由于一个错误的 FAT 结构或内部错误 FR_NOT_READY： 磁盘驱动器无法工作，由于驱动器中没有媒体或其他原因 FR_INVALID_OBJECT：文件对象非法
所在文件	ff.c
示例	FIL file; /* 文件对象 */ FRESULT res; /* FatFs 函数通用结果代码 */ BYTE buffer[4096] /* 读取数据缓冲区 */ UINT br /* 读取字节数 */ …… res = f_read(&file,buffer,10,&br); /* 读取文件的 10 个字节存到 buffer 缓冲区 */
注意事项	每次 f_read 函数执行完后，*ByteRead 值等于本次读取的字节数，如果*ByteRead < ByteToRead，则表示读/写指针在读操作期间已经到达文件末尾
快速信息	始终可用

表 1.15 f_write — 写文件

函数名称	f_write
函数原型	FRESULT f_write(FIL* FileObject, const void* Buffer, UINT ByteToWrite, UINT* ByteWritten)
功能描述	往文件中写数据
函数参数	FileObject： 指向打开文件对象结构体的指针 Buffer： 指向要写入的数据缓冲区的指针 ByteToWrite：UINT 范围内要写入的字节数 ByteWritten： 指向返回的已经写入的字节数 UINT 变量的指针，不管此函数掉调用后返回的结果如何，这个值始终有效
返回值	FR_OK(0)： 函数成功 FR_DENIED： 函数被拒由于文件已经开启了不可写模式

续上表

	FR_DISK_ERR： 函数失败由于磁盘运行的一个错误 FR_INT_ERR： 函数失败由于一个错误的 FAT 结构或内部错误 FR_NOT_READY： 磁盘驱动器无法工作，由于驱动器中没有媒体或其他原因 FR_INVALID_OBJECT：文件对象非法
所在文件	ff.c
示例	FIL file; /* 文件对象 */ FRESULT res; /* FatFs 函数通用结果代码 */ BYTE buffer[4096] /* 要写入的数据缓冲区 */ UINT bw /* 写入的字节数 */ …… res = f_write(&file, buffer, 10, &bw); /* 往文件里写 10 个字节 */
注意事项	每次 f_writ 函数执行完后，*ByteWritte 值等于本次写入的字节数，如果* ByteWritten < ByteToWrite，则表示在写操作期间卷已满
快速信息	当_FS_READONLY == 0 时可用

表 1.16 f_lseek — 移动读/写指针，扩展文件大小

函数名称	f_lseek
函数原型	FRESULT f_lseek(FIL* FileObject, DWORD offset)
功能描述	移动一个打开文件对象的文件读写指针，也可以用来增加文件大小
函数参数	FileObject：指向打开文件对象结构体的指针 offset： 偏移文件起始位置的字节数
返回值	FR_OK(0)： 函数成功 FR_DISK_ERR： 函数失败由于磁盘运行的一个错误 FR_INT_ERR： 函数失败由于一个错误的 FAT 结构或内部错误 FR_NOT_READY： 磁盘驱动器无法工作，由于驱动器中没有媒体或其他原因 FR_INVALID_OBJECT： 文件对象非法 FR_NOT_ENOUGH_CORE：文件的链接映射表大小不足
所在文件	ff.c
示例	FIL file; /* 文件对象 */ FRESULT res; /* FatFs 函数通用结果代码 */ …… res = f_lseek(&file, 5000); /* 指针指向文件的第 5000 个字节 */ res = f_lseek(&file, file ->fsize); /* 指针移动到文件末尾以附加数据 */ res = f_lseek(&file, file ->fptr + 3000); /* 当前指针向前移动 3000 个字节 */

续上表

	res = f_lseek(&file, file ->fptr -2000);　　/* 当前指针向后移动 2000 个字节（防止溢出）　　*/
注意事项	偏移值可以在文件起始和结尾的范围内指定，当在写模式下偏移值超出文件大小后，文件大小会增加，扩充区域的数据是未定义的。这适合快速创建大文件，尤其是快速的写操作； f_lseek 函数成功后，应该对文件读写指针 fptr 检查以确保读/写指针被正确移动，如果 fptr 不是期望值，可能是发生了以下情况： 文件结束，指定的偏移值被文件大小截断因为开启了只读模式； 磁盘已满，卷上没有足够的空间扩展文件大小。 当_USE_FASTSEEK 设为 1，文件对象成员 cltbl 非空时，快速寻找特性被使能，此时文件大小不能被扩大
快速信息	当_FS_MINIMIZE <= 2 时可用

表 1.17　f_truncate — 截断文件大小

函数名称	f_truncate
函数原型	FRESULT f_truncate(FIL* FileObject)
功能描述	截断文件大小
函数参数	FileObject：指向要被截断的文件对象结构体的指针
返回值	FR_OK(0)：　　函数成功 FR_DENIED：　　函数已经在只读模式下打开 FR_DISK_ERR：　　函数失败由于磁盘运行的一个错误 FR_INT_ERR：　　函数失败由于一个错误的 FAT 结构或内部错误 FR_NOT_READY：　　磁盘驱动器无法工作，由于驱动器中没有媒体或其他原因 FR_INVALID_OBJECT：　文件对象非法 FR_NOT_ENOUGH_CORE：文件的链接映射表大小不足
所在文件	ff.c
示例	FIL file;　/* 文件对象 */ FRESULT res;　　/* FatFs 函数通用结果代码　　*/ …… res = f_lseek(&file, 60);　　/* 读/写指针指向文件的第 60 个字节　　*/ res = f_truncate(&file);　　/* 将文件在当前指针处截断　　*/
注意事项	f_truncate 函数将文件在当前读/写指针处截断，如果文件读/写指针已经指向文件末尾，执行该函数将无效果
快速信息	当_FS_READONLY == 0 和 _FS_MINIMIZE == 0 时可用

表 1.18　f_sync — 刷新缓冲区

函数名称	f_sync
函数原型	FRESULT f_sync(FIL* FileObject)
功能描述	刷新写文件的缓存信息
函数参数	FileObject：指向要刷新的文件对象结构体的指针
返回值	FR_OK(0)：　　函数成功 FR_DISK_ERR：　　函数失败由于磁盘运行的一个错误 FR_INT_ERR：　　函数失败由于一个错误的 FAT 结构或内部错误 FR_NOT_READY：　　磁盘驱动器无法工作，由于驱动器中没有媒体或其他原因 FR_INVALID_OBJECT：文件对象非法
所在文件	ff.c
示例	FIL file;　　/* 文件对象　　*/ …… res = f_sync(&file);　　/* 刷新文件缓存，及时保存　　*/
注意事项	f_sync 函数执行和 f_close 函数相同的处理，不同在于执行后文件仍保持打开，文件对象依然有效，可以继续对文件进行读/写/移动操作； 当文件处于长时间的写模式,如数据记录时，定期调用此函数，或写入数据后立即调用此函数，可以减少因断电等意外情况带来的损失
快速信息	当_FS_READONLY == 0 时可用

表 1.19　f_opendir — 打开一个目录

函数名称	f_opendir
函数原型	FRESULT f_opendir(DIR* DirObject, const TCHAR* DirName)
功能描述	打开一个目录
函数参数	DirObject：指向空目录结构体的指针，用来存储要打开的目录信息 DirName：指向空结尾字符串目录名的指针
返回值	FR_OK(0)：　　函数成功，目录结构体被创建，以供随后的读目录调用 FR_NO_PATH：　　无法找到路径 FR_INVALID_NAME：路径名非法 FR_INVALID_DRIVE：驱动器号非法 FR_NOT_READY：　　磁盘驱动器无法工作，由于驱动器中没有媒体或其他原因 FR_DISK_ERR：　　函数失败由于磁盘运行的一个错误 FR_INT_ERR：　　函数失败由于一个错误的 FAT 结构或内部错误 FR_NOT_ENABLED：　逻辑驱动器没有工作区

续上表

	FR_NO_FILESYSTEM：驱动器上没有合法的 FAT 卷
所在文件	ff.c
示例	参见 f_readdir()示例
注意事项	f_opendir 打开一个存在的目录并创建该目录对象以供后面调用，目录对象结构体无需任何步骤可在任何时间被丢弃
快速信息	当_FS_MINIMIZE <= 1 时可用

表 1.20 f_readdir — 读取目录

函数名称	f_readdir
函数原型	FRESULT f_readdir(DIR* DirObject, FILINFO* FileInfo)
功能描述	读取目录
函数参数	DirObject：指向打开的目录结构体的指针 FileInfo： 指向文件信息结构体的指针，用来存储读取到的文件信息
返回值	FR_OK(0)： 函数成功 FR_NOT_READY： 磁盘驱动器无法工作，由于驱动器中没有媒体或其他原因 FR_DISK_ERR： 函数失败由于磁盘运行的一个错误 FR_INT_ERR： 函数失败由于一个错误的 FAT 结构或内部错误 FR_INVALID_OBJECT：文件对象非法
所在文件	ff.c
示例	/***** ** Function name: FatReadDirTest ** Descriptions: Fatfs 读取目录测试 ** input parameters: 无 ** output parameters: 无 ** Returned value: 无 *****/ void FatReadDirTest() { FATFS fs; /* 文件系统对象结构体 */ DIR dir; /* 声明目录结构体，保存要打开的 */ /* 文件夹信息 */ FILINFO fno; /* 文件状态结构体 */ FRESULT res; /* FatFs 函数通用结果代码 */

续上表

	f_mount(0,&fs); /* 注册工作区 */
	
	if(f_opendir(&dir,"folder/zlgmcu") == FR_OK) { /* 打开\folder\zlgmcu 文件夹成功 */
	while(f_readdir(&dir, &fno) == FR_OK) { /* 读目录信息到文件状态结构体中 */
	if(!fno.fname[0]) break; /* 如果文件名为 0，结束 */
	if(fno.fattrib == AM_ARC) { /* 如果读取的文件只有存档属性 */
	xprintf("文件名：%s\r\n", fno.fname);
	}
	}
	}
	}

注意事项	f_readdir 按顺序读取目录内文件，重复调用此函数可读取目录内所有文件； 当所有的目录入口被读完而没有条目可读时，函数返回一个空字符串到 f_name[]，据此可判断目录内所有文件是否读完； 如果一个空指针赋给 FileInfo，将返回从第一个文件开始读取； 当 LFN 功能启用时，文件信息结构体中的 lfname 和 lfsize 必须在使用 f_readdir 函数前初始化为有效值，lfname 是一个指向字符串缓冲区返回长文件名的指针，lfsize 是字符串缓冲区的大小，如果读缓冲区或 LFN 工作缓冲区的大小对于 LFN 不够或者对象是短文件名，一个空字符串将返回到 LFN 读缓冲区； 如果 LFN 包含任何不能被 OEM 代码转换的字符，一个空字符串将被返回，但这不是针对 Unicode API 配置的情况； 当 lfname 为 NULL 时，没有什么返回，当对象中没有 LFN 时，大小写字母都被包含在 SFN 中； 当相对路径功能使能时（_FS_RPATH == 1），将会影响到 f_readdir()的读取，具体表现为”.”和”..”条目不会被过滤，他们将出现在读取条目中
快速信息	当_FS_MINIMIZE <= 1 时可用

表 1.21 f_getfree — 获取空闲簇

函数名称	f_getfree
函数原型	FRESULT f_getfree(const TCHAR* Path, DWORD* Clusters, FATFS** FileSystemObject)
功能描述	获取空闲簇
函数参数	Path： 指向空结尾字符串逻辑驱动器号的指针 Clusters： 指向双字变量的指针，用来存储空闲簇 FileSystemObject：指向存储相应文件系统对象指针的指针
返回值	FR_OK(0)： 函数成功，* Clusters 有空闲簇，* FileSystemObject 指向文件系统对象 FR_INVALID_DRIVE：驱动器号非法

续上表

	FR_NOT_READY： 磁盘驱动器无法工作，由于驱动器中没有媒体或其他原因 FR_DISK_ERR： 函数失败由于磁盘运行的一个错误 FR_INT_ERR： 函数失败由于一个错误的 FAT 结构或内部错误 FR_NOT_ENABLED： 逻辑驱动器没有工作区 FR_NO_FILESYSTEM：驱动器上没有合法的 FAT 分区
所在文件	ff.c

示例	FATFS *fs; /* 声明文件系统对象结构体的指针 */
	DWORD fre_clust, fre_sect, tot_sect; /* 定义空闲簇数，空闲扇区数，全部扇区数 */
	FRESULT res; /* FatFs 函数通用结果代码 */
	……
	/* 获取驱动器 1 上的卷信息和空闲簇数 */
	res = f_getfree(“1:”, &fre_clust, &fs);
	/* 获取全部扇区数和空闲扇区数 */
	tot_sect =(fs->n_fatent -2) * fs->csize;
	fre_sect = fre_clust * fs->csize;
	/* 输出显示剩余空间（单位：KB 假定 512 字节/每扇区） */
	printf(“%lu KB total drive space.\n” “%lu KB available.\n”, tot_sect / 2, fre_sect / 2);

注意事项	f_getfree 函数获取驱动器上的空闲簇，文件系统对象的成员 csize 代表每簇的扇区数，所以剩余空间可由单位扇区计算出来，当 FSInfo 结构在 FAT32 卷上不同步时，函数会返回一个错误的空闲簇
快速信息	当_FS_READONLY == 0 和_FS_MINIMIZE == 0 时可用

表 1.22 f_stat — 获取文件状态

函数名称	f_stat
函数原型	FRESULT f_ stat(const TCHAR* FileName, FILINFO* FileInfo)
功能描述	获取文件状态
函数参数	FileName：指向空结尾字符串文件名或目录的指针 FileInfo： 指向空的 FILINFO 结构体，用来保存文件信息
返回值	FR_OK(0)： 函数成功 FR_NO_FILE： 无法找到文件或目录 FR_INVALID_NAME：文件名非法 FR_INVALID_DRIVE：驱动器号非法 FR_NOT_READY： 磁盘驱动器无法工作，由于驱动器中没有媒体或其他原因 FR_DISK_ERR： 函数失败由于磁盘运行的一个错误 FR_INT_ERR： 函数失败由于一个错误的 FAT 结构或内部错误

续上表

	FR_NOT_ENABLED： 逻辑驱动器没有工作区 FR_NO_FILESYSTEM：驱动器上没有合法的 FAT 卷
所在文件	ff.c

示例	FILINFO finfo; /* 文件信息结构体 */
	FRESULT res; /* FatFs 函数通用结果代码 */
	……
	/* 读取 folder 文件夹下 zlgmcu 目录的信息到 finfo 结构体中 */
	res = f_stat("folder/zlgmcu", &finfo);
	/* 读取根目录下 zlgmcu.txt 文件的信息 */
	res = f_stat("0: zlgmcu.txt ", &finfo);

注意事项	f_stat 函数获取一个文件或目录的信息，如最近修改时间，属性等，获取的信息存在 FileInfo 结构体中； 如果目标是文件夹，获取的大小为 0； 此函数对根目录无效； 对于信息的细节，参考 FILINFO 结构体和 f_readdir 函数，该功能不支持最小等级>=1。
快速信息	无

表 1.23 f_mkdir — 创建一个新目录

函数名称	f_mkdir
函数原型	FRESULT f_mkdir(const TCHAR* DirName)
功能描述	创建一个新目录
函数参数	DirName：指向要创建的空结尾字符串目录名的指针
返回值	FR_OK(0)： 函数成功 FR_NO_PATH： 无法找到路径 FR_INVALID_NAME： 路径名非法 FR_INVALID_DRIVE： 驱动器号非法 FR_DENIED： 不能创建目录由于目录表或磁盘已满 FR_EXIST： 存在同名文件或目录 FR_NOT_READY： 磁盘驱动器无法工作，由于驱动器中没有媒体或其他原因 FR_WRITE_PROTECTED：媒体被写保护 FR_DISK_ERR： 函数失败由于磁盘运行的一个错误 FR_INT_ERR： 函数失败由于一个错误的 FAT 结构或内部错误 FR_NOT_ENABLED： 逻辑驱动器没有工作区 FR_NO_FILESYSTEM： 驱动器上没有合法的 FAT 卷

续上表

所在文件	ff.c
示例	FRESULT res; /* FatFs 函数通用结果代码 */ …… res = f_mkdir(”folder”); /* 创建 folder 文件夹 */ res = f_mkdir(”folder/zlgmcu”); /* 在 folder 目录下创建 zlgmcu 文件夹 */ res = f_mkdir(”folder/zlgmcu.txt”); /* 在 folder 目录下创建 zlgmcu.txt 文件夹 */
注意事项	f_mkdir 函数创建一个新目录，目录名应符合 FatFs 标准，不能包含非法字符； 不能用来创建文件，不能在不存在的目录下创建新目录； 若不支持长文件名，文件名长度不能大于 8，否则创建不成功
快速信息	当_FS_READONLY == 0 和_FS_MINIMIZE == 0 时可用

表 1.24 f_unlink — 删除一个文件

函数名称	f_unlink
函数原型	FRESULT f_unlink(const TCHAR* FileName)
功能描述	移除一个对象
函数参数	FileName：指向要移除的空结尾字符串对象的指针
返回值	FR_OK(0)： 函数成功 FR_NO_FILE： 无法找到文件或目录 FR_NO_PATH： 无法找到路径 FR_INVALID_NAME： 文件名非法 FR_INVALID_DRIVE： 驱动器号非法 FR_DENIED： 函数由于以下原因被拒绝： 对象属性为只读； 目录下非空； 当前目录。 FR_NOT_READY： 磁盘驱动器无法工作，由于驱动器中没有媒体或其他原因 FR_WRITE_PROTECTED：媒体写保护 FR_DISK_ERR： 函数失败由于磁盘运行的一个错误 FR_INT_ERR： 函数失败由于一个错误的 FAT 结构或内部错误 FR_NOT_ENABLED： 逻辑驱动器没有工作区 FR_NO_FILESYSTEM： 驱动器上没有合法的 FAT 卷 FR_LOCKED： 函数被拒由于文件共享机制（_FS_SHARE）
所在文件	ff.c

续上表

示例	FRESULT res; /* FatFs 函数通用结果代码 */ …… res = f_unlink("folder/zlgmcu"); /* 移除 folder 目录下的 zlgmcu 文件夹 */ res = f_unlink("/zlgmcu.txt"); /* 移除根目录下的 zlgmcu.txt 文件 */
注意事项	f_unlink 函数用来移除一个对象，但不能移除已经打开的对象。
快速信息	当_FS_READONLY == 0 和_FS_MINIMIZE == 0 时可用

表 1.25 f_chmod — 改变一个文件或目录的属性

函数名称	f_chmod
函数原型	FRESULT f_chmod(const TCHAR* FileName, BYTE Attribute, BYTE AttributeMask)
功能描述	改变一个文件或目录的属性
函数参数	FileName： 指向要改变的空结尾字符串文件名或目录名的指针 Attribute： 要置位的属性，可以是以下一种或几种的组合，指定属性被置位而其它被清除 AM_RDO /* 只读 */ AM_ARC /* 存档 */ AM_SYS /* 系统 */ AM_HID /* 隐藏 */ AttributeMask：需要改变的属性，包括要置位和要清除的属性
返回值	FR_OK(0)： 函数成功 FR_NO_FILE： 无法找到文件 FR_NO_PATH： 无法找到路径 FR_INVALID_NAME： 文件名非法 FR_INVALID_DRIVE： 驱动器号非法 FR_NOT_READY： 磁盘驱动器无法工作，由于驱动器中没有媒体或其他原因 FR_WRITE_PROTECTED：媒体被写保护 FR_DISK_ERR： 函数失败由于磁盘运行的一个错误 FR_INT_ERR： 函数失败由于一个错误的 FAT 结构或内部错误 FR_NOT_ENABLED： 逻辑驱动器没有工作区 FR_NO_FILESYSTEM： 驱动器上没有合法的 FAT 卷
所在文件	ff.c
示例	FRESULT res; /* FatFs 函数通用结果代码 */ …… /* 设定为只读属性，清除存档属性，其他属性保持 */ res = f_chmod(”zlgmcu.txt”, AM_RDO, AM_RDO | AM_ARC);

续上表

注意事项	f_chmod 函数改变一个文件或目录的属性，Attribute 须为 AttributeMask 的子集，若 Attribute 包含在 AttributeMask 中则置位，否则清除
快速信息	当_FS_READONLY == 0 和_FS_MINIMIZE == 0 时可用

表 1.26 f_utime —改变一个文件或目录的时间戳

函数名称	f_utime
函数原型	FRESULT f_utime(const TCHAR* FileName, const FILINFO* TimeDate)
功能描述	改变一个文件或目录的时间戳
函数参数	FileName：指向要改变的空结尾字符串文件名或目录名的指针 TimeDate：指向文件信息结构体的指针，其时间戳可被内部成员 fdate 和 ftime 设定，可忽略其他内部成员
返回值	FR_OK(0)：函数成功 FR_NO_FILE：无法找到文件 FR_NO_PATH：无法找到路径 FR_INVALID_NAME：文件名非法 FR_INVALID_DRIVE：驱动器号非法 FR_NOT_READY：磁盘驱动器无法工作，由于驱动器中没有媒体或其他原因 FR_WRITE_PROTECTED：媒体被写保护 FR_DISK_ERR：函数失败由于磁盘运行的一个错误 FR_INT_ERR：函数失败由于一个错误的 FAT 结构或内部错误 FR_NOT_ENABLED：逻辑驱动器没有工作区 FR_NO_FILESYSTEM：驱动器上没有合法的 FAT 卷
所在文件	ff.c

示例	FRESULT res; /* FatFs 函数通用结果代码 */
	FRESULT set_timestamp(char* obj, int year, int month, int mday, int hour, int min, int sec) {
	FILINFO fno;
	fno.fdate =(WORD)(((year - 1980) * 512U) | month * 32U | mday);
	fno.ftime =(WORD)((hour * 2048U) | min * 32U | sec / 2U);
	return f_utime(obj,&fno);
	}
	……
	res = set_timestamp(“zlgmcu.txt”,2010,10,15,02,03,46); /* 改变 zlgmcu.txt 文件的修改时间 */

注意事项	f_utime 函数可改变一个文件或目录的最近修改时间
快速信息	当_FS_READONLY == 0 和_FS_MINIMIZE == 0 时可用

表 1.27　f_rename —重命名一个对象

函数名称	f_rename
函数原型	FRESULT f_rename(const TCHAR* OldName, const TCHAR* NewName,)
功能描述	重命名一个对象
函数参数	OldName：指向要重命名的空结尾字符串旧对象名的指针 NewName：指向空结尾字符串新对象名的指针，无需驱动器号
返回值	FR_OK(0)：　　函数成功 FR_NO_FILE：　　无法找到旧文件名 FR_NO_PATH：　　无法找到路径 FR_INVALID_NAME：　文件名非法 FR_INVALID_DRIVE：　驱动器号非法 FR_NOT_READY：　磁盘驱动器无法工作，由于驱动器中没有媒体或其他原因 FR_EXIST：　　新名称和现有名称冲突 FR_DENIED：　　由于某些原因新名称不能被创建 FR_WRITE_PROTECTED：媒体写保护 FR_DISK_ERR：　　函数失败由于磁盘运行的一个错误 FR_INT_ERR：　　函数失败由于一个错误的 FAT 结构或内部错误 FR_NOT_ENABLED：　逻辑驱动器没有工作区 FR_NO_FILESYSTEM：　驱动器上没有合法的 FAT 卷 FR_LOCKED：　　函数被拒由于文件共享机制（_FS_SHARE）
所在文件	ff.c
示例	FRESULT res;　　/* FatFs 函数通用结果代码　　*/ …… res = f_rename("oldname.txt", “newname.txt”);　　/* 重命名根目录下的一个文件　　*/ res = f_rename("oldfolder/oldname.txt", "newfolder/newname.txt"); 　　/* 移动文件到其他目录并重命名　　*/ res = f_rename("oldfolder", "otherfolder/newfolder"); /* 移动文件夹到其他目录并重命名　　*/
注意事项	f_rename 函数用来重命名一个对象（文件或目录），也可以移动对象到其他目录； 逻辑驱动器号由旧名称确定，新名称不能包含逻辑驱动器号； 不能重命名已经打开的对象
快速信息	当_FS_READONLY == 0 和_FS_MINIMIZE == 0 时可用

表 1.28　f_mkfs — 格式化

函数名称	f_mkfs
函数原型	FRESULT f_mkfs(BYTE Drive, BYTE PartitioningRule, UINT AllocSize)
功能描述	在驱动器上创建文件系统
函数参数	Drive：　逻辑驱动器号，取值 0~9 PartitioningRule：　取值 0，分区表被创建到主引导记录，主 DOS 分区被创建，然后分区上 FAT 卷被创建，这就是 FDISK 格式化，用作硬盘和记忆卡； 取值 1，FAT 卷起始于第一个扇区，驱动器上没有分区表，这属于 SFD 格式化，用作软盘和光盘 AllocSize：　强制分配单位（簇）大小（以字节为单位），取值应在扇区大小和 128 倍的扇区大小之前，如果指定了非法值，簇大小由卷大小决定
返回值	FR_OK(0)：　函数成功 FR_INVALID_DRIVE：　驱动器号非法 FR_NOT_READY：　磁盘驱动器无法工作，由于驱动器中没有媒体或其他原因 FR_WRITE_PROTECTED：驱动器被写保护 FR_NOT_ENABLED：　逻辑驱动器没有工作区 FR_DISK_ERR：　函数失败由于磁盘运行的一个错误 FR_MKFS_ABORTED：　由于以下原因函数在开始格式化之前被取消： 磁盘容量太小； 函数参数非法； 驱动器上没有正当的扇区大小，当获取的簇数接近 0xFF7 和 0xFFF7 时会发生
所在文件	ff.c
示例	FRESULT res;　/* FatFs 函数通用结果代码　*/ …… /* SFD 格式化，不需要引导扇区，分配每簇为 4096 个字节 */ /* 以每扇区 512 个字节计，即每簇 8 个扇区 */ Res = f_mkfs(0, 1, 4096);
注意事项	f_mkfs 函数在驱动器上创建一个 FAT 卷，对于可移动设备有两个分区规则，FDISK 和 SFD，对于大多数场合，推荐使用 FDISK 格式化； 函数不支持多分区，故物理驱动器上存在的分区将会被删除，重建一个占用整个磁盘空间的新分区； FAT 子类型，FAT12/FAT16/FAT32 是由卷上簇数决定的，遵循微软规定的 FAT 规范，因此选择哪个 FAT 子类型，取决于卷大小和指定的簇大小，簇大小影响文件系统的性能，更大的簇可提高性能； 当簇数到达 FAT 子类型临界区时，函数会失败，返回 FR_MKFS_ABORTED
快速信息	当_FS_READONLY == 0 和_USE_MKFS == 1 时可用

表 1.29　f_forward —读取文件数据转移到数据流设备

函数名称	f_forward
函数原型	FRESULT f_forward(FIL* FileObject, UINT(*Func)(const BYTE*,UINT), UINT ByteToFwd, UINT* ByteFwd)
功能描述	读取文件数据转移到数据流设备
函数参数	FileObject：　指向打开的文件对象的指针 Func：　　指向用户定义的数据流函数的指针，具体细节请参考示例代码 ByteToFwd：指向 UINT 范围内的转移字节数的指针 ByteFwd：　指向返回转移字节数的 UINT 变量的指针
返回值	FR_OK(0)：　　函数成功 FR_DENIED：　　函数被拒由于文件已在只读模式下打开 FR_DISK_ERR：　　函数失败由于磁盘运行的一个错误 FR_INT_ERR：　　函数失败由于一个错误的 FAT 结构或内部错误 FR_NOT_READY：　磁盘驱动器无法工作，由于驱动器中没有媒体或其他原因 FR_INVALID_OBJECT：文件对象非法
所在文件	ff.c

示例	/* 音频播放--被 f_forward 调用的数据传送函数示例　*/
	UINT out_stream(　　/* 返回发送的字节数或流状态　　*/
	　const BYTE *p,　　/* 指向要发送的数据块的指针　　*/
	　UINT btf　　/* >0: Transfer call(Number of bytes to be sent). 0: Sense call */
	) {
	　UINT cnt = 0;
	　if(btf == 0) {　　/* Sense call */
	　　/* 返回流状态(0: Busy, 1: Ready)　*/
	　　/* When once it returned ready to sense call, it must accept a byte at least */
	　　/* at subsequent transfer call, or f_forward will fail with FR_INT_ERROR. */
	　　if(FIFO_READY) cnt = 1;
	　}
	　else {　　/* Transfer call */
	　　do {　　/* 当有数据要发送、且流已经准备好则重复　　*/
	　　FIFO_PORT = *p++;
	　　cnt++;
	　　} while(cnt < btf && FIFO_READY);
	　}
	　return cnt;

续上表

	}
	/* 使用 f_forward 函数示例 */
	FRESULT play_file (char *fn) /* 指向要播放的音频文件名的指针 */
	{
	FRESULT rc;
	FIL fil;
	UINT dmy;
	/* 只读模式下打开音频文件 */
	rc = f_open(&fil, fn, FA_READ);
	if(rc) return rc;
	/* 重复直到文件读指针到达文件末尾 */
	while(rc == FR_OK && fil.fptr < fil.fsize) {
	/* 其他程序代码... */
	/* Fill output stream periodicaly or on-demand */
	rc = f_forward(&fil, out_stream, 1000, &dmy);
	}
	/* 关闭文件并返回 */
	f_close(&fil);
	return rc;
	}

注意事项	f_forward 函数从文件中读取数据转移到无需数据缓冲区的输出流，对于在应用模块不需要数据缓冲区的小内存系统是非常适合的，文件对象的文件指针随转移的字节数增加，如果* ByteFwd < ByteToFwd 且没有错误，则表示到了文件末尾，请求的的字节没有转移，或者在数据转移期间数据流处于忙状态
快速信息	当_USE_FORWARD == 1 和_FS_TINY == 1 时可用

表 1.30 f_chdir —改变驱动器的当前目录

函数名称	f_chdir
函数原型	FRESULT f_chdir(const TCHAR* Path)
功能描述	改变驱动器的当前目录
函数参数	Path：指向要跳转到的空结尾字符串目录的指针
返回值	FR_OK(0)： 函数成功 FR_NO_PATH： 无法找到路径 FR_INVALID_NAME：路径名非法 FR_INVALID_DRIVE：驱动器号非法

续上表

	FR_NOT_READY： 磁盘驱动器无法工作，由于驱动器中没有媒体或其他原因 FR_DISK_ERR： 函数失败由于磁盘运行的一个错误 FR_INT_ERR： 函数失败由于一个错误的 FAT 结构或内部错误 FR_NOT_ENABLED： 逻辑驱动器没有工作区 FR_NO_FILESYSTEM：驱动器上没有合法的 FAT 卷
所在文件	ff.c
示例	f_chdir(“/folder”); /* 改变当前驱动器的当前目录（根目录下的 folder 文件夹） */ f_chdir(“2:”); /* 变驱动器 2 的当前目录（总目录） */
注意事项	f_chdir 函数改变驱动器的当前目录，当驱动器自动注册的时候，当前目录初始化为根目录；在每个文件系统对象中，当前目录是保持的，所以这会影响到使用这个驱动器的其他任务
快速信息	当_FS_RPATH >= 1 时可用

表 1.31 f_chdrive —改变当前驱动器

函数名称	f_chdrive
函数原型	FRESULT f_chdrive(BYTE Drive)
功能描述	改变当前驱动器
函数参数	Drive：指定驱动器号作为当前驱动器
返回值	FR_OK(0)：函数成功 FR_INVALID_DRIVE：驱动器号非法
所在文件	ff.c
示例	f_chdrive(2); /* 设定驱动器 2 为当前驱动器 */
注意事项	f_chdrive 函数改变当前驱动器，当前驱动器号初始值为 0； 当前驱动器保持在一个静态变量中，所以这会影响到使用文件函数的其他任务
快速信息	当_FS_RPATH >= 1 时可用

表 1.32 f_getcwd —检索当前目录

函数名称	f_getcwd
函数原型	FRESULT f_ getcwd(TCHAR* Buffer, UINT BufferLen)
功能描述	检索当前目录
函数参数	Buffer： 指向接受当前目录字符串缓冲区的指针 BufferLen：缓冲区大小(单位 TCHAR)
返回值	FR_OK(0)： 函数成功 FR_NOT_READY： 磁盘驱动器无法工作，由于驱动器中没有媒体或其他原因

续上表

	FR_DISK_ERR：　函数失败由于磁盘运行的一个错误 FR_INT_ERR：　函数失败由于一个错误的 FAT 结构或内部错误 FR_NOT_ENABLED：　逻辑驱动器没有工作区 FR_NO_FILESYSTEM：　驱动器上没有合法的 FAT 卷 FR_NOT_ENOUGH_CORE：缓冲区太小
所在文件	ff.c
示例	TCHAR Buffer[MAXPATH];　/* 定义缓冲区 */ …… res = f_getcwd(Buffer, MAXPATH);　/* 复制当前目录到 Buffer 缓冲区 */
注意事项	f_getcwd 函数会将当前目录的绝对路径复制到缓冲区
快速信息	当_FS_RPATH == 2 时可用

表 1.33　f_gets —从文件中读取字符串

函数名称	f_gets
函数原型	TCHAR* f_ gets(TCHAR* Str, int Size, FIL* FileObject)
功能描述	从文件中读取字符串
函数参数	Str：　指向存储读字符串读缓冲区的指针 Size：　读缓冲区的字节大小 FileObject：指向打开文件对象结构体的指针
返回值	函数成功时 返回缓冲区指针 Str
所在文件	ff.c
示例	FIL file;　/* 文件对象 */ TCHAR buffer[512]　/* 定义读缓冲区 */ …… f_gets(buffer, sizeof(buffer), &file);　/* 获取文件内容到 buffer 缓冲区 */
注意事项	f_gets()是 f_read()的封装函数，读操作一直持续直到存储一个'\n'字符，到达文件末尾，或者缓冲区已经填充了 Size – 1 个字符； 读字符串以'\0'结尾，当没有字符可读或在读操作期间出现任何错误，f_gets()会返回一个空指针。文件结束和错误状态可以通过调用 f_eof()和 f_error()得知； 当 FatFs 配置成 Unicode API(_LFN_UNICODE == 1)时，文件将以 UTF-8 编码读取，以 UCS-2 存储到缓冲区，如果不是这种情况，文件将每字符一个字节的方式读取而不进行任何代码转换
快速信息	当_USE_STRFUNC 等于 1 或 2 时可用，若设为 2，文件中包含'\r'的会换行

表 1.34　f_putc —往文件中写一个字符

函数名称	f_putc
函数原型	int f_putc(TCHAR Chr, FIL* FileObject)
功能描述	往文件中写一个字符
函数参数	Chr：　　要写入的字符 FileObject：指向打开文件对象结构体的指针
返回值	字符成功写入时，函数将返回 1，若由于磁盘已满或任何错误导致函数失败，则返回 EOF(-1)； 当 FatFs 配置成 Unicode API(_LFN_UNICODE == 1)时，UCS-2 字符以 UTF-8 编码的方式写入文件，如果不是这种情况，字节将被直接写入
所在文件	ff.c
示例	FIL file;　　/* 文件对象　　*/ TCHAR CHARACTER;　　/* 声明要写入的字符　　*/ …… f_putc(CHARACTER, &file);　　/* 写一个字符到文件中　　*/
注意事项	f_putc()是 f_write()的封装函数
快速信息	当_FS_READONLY == 0 和_USE_STRFUNC 等于 1 或 2 时可用，若设为 2，’\n’会被转换为”\r\n”

表 1.35　f_puts —往文件中写一个字符串

函数名称	f_puts
函数原型	int f_puts(const TCHAR* Str, FIL* FileObject)
功能描述	往文件中写一个字符串
函数参数	Str：　　指向要写入的空结尾字符串的指针，空字符不能被写入 FileObject：指向打开文件对象结构体的指针
返回值	函数成功时，返回写入的字节数，若由于磁盘已满或任何错误导致函数失败，则返回 EOF(-1)； 当 FatFs 配置成 Unicode API(_LFN_UNICODE == 1)时，UCS-2 字符串以 UTF-8 编码的方式写入文件，如果不是这种情况，字节流将被直接写入
所在文件	ff.c
示例	FIL file;　　/* 文件对象　　*/ TCHAR* Str;　　/* 声明要写入的字符　　*/ f_puts(Str, &file);　　/* 写一个字符串到文件中　　*/
注意事项	f_puts()是 f_putc()的封装函数
快速信息	当_FS_READONLY == 0 和_USE_STRFUNC 等于 1 或 2 时可用，若设为 2，’\n’会被转换为”\r\n”

表 1.36　f_printf —往文件中写入格式化字符串

函数名称	f_printf
函数原型	TCHAR f_puts(FIL* FileObject, const TCHAR* Formatt, …)
功能描述	往文件中写入格式化字符串
函数参数	FileObject：指向打开文件对象结构体的指针 Format：　指向空结尾格式化字符串的指针 …：　　可选参数
返回值	函数成功时，返回写入的字节数，若由于磁盘已满或任何错误导致函数失败，则返回 EOF(-1)
所在文件	ff.c
示例	FIL file;　　/* 文件对象　　*/ …… f_printf(&file, “%6d”, -200);　　/* 写入“-200”　　*/ f_printf(&file, “%02u”, 5);　　/* 写入”05”　　*/ f_printf(&file, “%ld”, 12345678L);　　/* 写入”12345678”　　*/ f_printf(&file, “%081X”, 1194684UL);　　/* 写入”00123ABC”　　*/ f_printf(&file, “%08b”, 0x55);　　/* 写入”01010101”　　*/ f_printf(&file, “%s”, “ZLGMCU”);　　/* 写入”ZLGMCU”　　*/ f_printf(&file, “%c”, ‘a’);　　/* 写入”a”　　*/
注意事项	f_printf()是 f_putc()和 f_puts()的封装函数，格式控制指令是标准库的一个子集，如下所示： Type:c C s S d D u U x X b B Size:l L Flag:0
快速信息	当_FS_READONLY == 0 和_USE_STRFUNC 等于 1 或 2 时可用，若设为 2，’\n’会被转换为”\r\n”； 当 FatFs 配置成 Unicode API(_LFN_UNICODE == 1)时，UCS-2 字符串以 UTF-8 编码的方式写入文件，如果不是这种情况，字符将被直接写入