

FatFs 通用 FAT 文件系统模块

中文手册

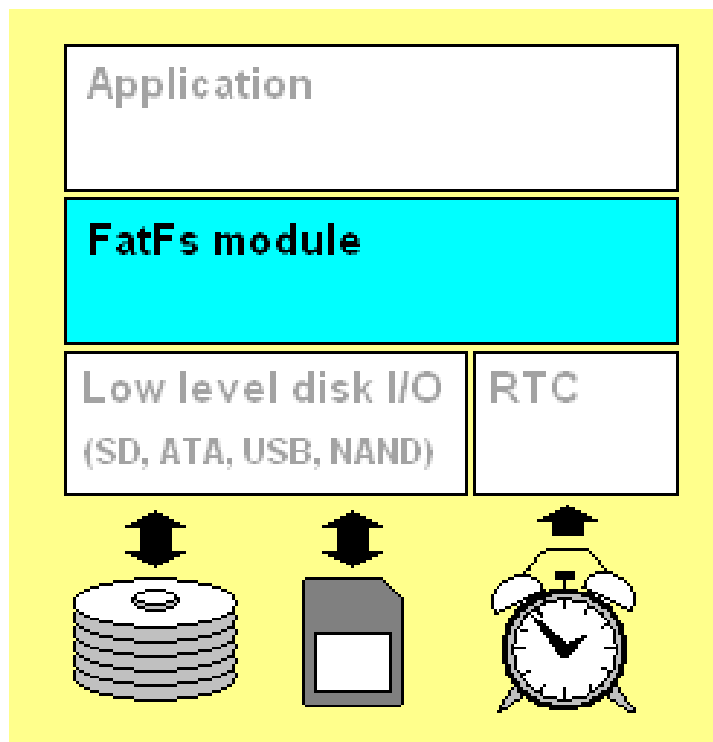
版本：R0.009a

目录

1、特点	1
2、应用程序接口	2
2.1 f_mount	3
2.2 f_open	4
2.3 f_close	6
2.4 f_read	7
2.5 f_write	8
2.6 f_lseek	9
2.7 f_truncate	10
2.8 f_sync	11
2.9 f_opendir	12
2.10 f_readdir	13
2.11 f_getfree	15
2.12 f_stat	16
2.13 f_mkdir	17
2.14 f_unlink	18
2.15 f_chmod	19
2.16 f_utime	20
2.17 f_rename	21
2.18 f_chdir	22
2.19 f_chdrive	23
2.20 f_getcwd	24
2.21 f_forward	25
2.22 f_mkfs	27
2.23 f_fdisk	28
2.24 f_gets	29
2.25 f_putc	30
2.26 f_puts	31
2.27 f_printf	32
2.28 f_tell	33
2.29 f_eof	34
2.30 f_size	35
2.31 f_error	36
3、磁盘 I/O 接口	37
3.1 disk_initialize	37
3.2 disk_status	37
3.3 disk_read	38
3.4 disk_write	38
3.5 disk_ioctl	39
3.6 get_fattime	40
4、FatFs 模块应用	41
4.1 如何移植	41

4.1.1 基本概念	41
4.1.2 系统组织	41
4.1.3 需要哪些函数?	41
4.2 限制	42
4.3 内存使用(R0.09a).....	42
4.4 模块大小裁减	42
4.5 长文件名	43
4.6 Unicode API.....	43
4.7 重入	44
4.8 重复的文件访问	44
4.9 执行有效的文件访问	44
4.10 关于闪存媒体的考虑	45
4.10.1 使用多扇区写	45
4.10.2 强制内存擦除	45
4.11 临界区 (Critical Section)	46

FatFs 是一个为小型嵌入式系统设计的通用 FAT(File Allocation Table)文件系统模块。FatFs 的编写遵循 ANSI C，并且完全与磁盘 I/O 层分开。因此，它独立(不依赖)于硬件架构。它可以被嵌入到低成本的微控制器中，如 AVR、8051、PIC、ARM、Z80、68K 等等，而不需要做任何修改。



1、特点

- ◆ Windows 兼容的 FAT 文件系统
- ◆ 不依赖于平台，易于移植
- ◆ 代码和工作区占用空间非常小
- ◆ 多种配置选项：
 - 多卷(物理驱动器和分区)
 - 多 ANSI/OEM 代码页，包括 DBCS
 - 在 ANSI/OEM 或 Unicode 中长文件名的支持
 - RTOS 的支持
 - 多扇区大小的支持
 - 只读，最少 API，I/O 缓冲区等等

2、应用程序接口

FatFs 模块为应用程序提供了下列函数，这些函数描述了 FatFs 能对 FAT 卷执行哪些操作。

函数名	描述
f_mount	注册/注销一个工作区
f_open	打开/创建一个文件
f_close	关闭一个文件
f_read	读取文件
f_write	写文件
f_lseek	移动读/写指针，扩展文件大小
f_truncate	截断文件大小
f_sync	清空缓冲数据
f_opendir	打开一个目录
f_readdir	读取一个目录项
f_getfree	获取空闲簇
f_stat	获取文件状态
f_mkdir	创建一个目录
f_unlink	删除一个文件或目录
f_chmod	修改属性
f_utime	修改时间戳
f_rename	删除/移动一个文件或目录
f_chdir	修改当前目录
f_chdrive	修改当前驱动器
f_getcwd	恢复当前目录
f_forward	直接输出文件数据流
f_mkfs	在驱动器上创建一个文件系统
f_fdisk	划分一个物理驱动器
f_gets	读取一个字符串
f_putc	写一个字符
f_puts	写一个字符串
f_printf	写一个格式化的字符串
f_tell	获取当前读/写指针
f_eof	测试一个文件是否到达文件末尾
f_size	获取一个文件的大小
f_error	测试一个文件是否出错

2.1 f_mount

在 FatFs 模块上注册/注销一个工作区(文件系统对象)。

```
FRESULT f_mount (  
    BYTE  Drive,          /* 逻辑驱动器号 */  
    FATFS* FileSystemObject /* 工作区指针 */  
);
```

参数

Drive 注册/注销工作区的逻辑驱动器号(0-9)。

FileSystemObject 工作区(文件系统对象)指针。

返回值

FR_OK (0)函数成功。

FR_INVALID_DRIVE 驱动器号无效

描述

f_mount 函数在 FatFs 模块上注册/注销一个工作区。在使用任何其他文件函数之前，必须使用该函数为每个卷注册一个工作区。要注销一个工作区，只要指定 FileSystemObject 为 NULL 即可，然后该工作区可以被丢弃。

该函数只初始化给定的工作区，以及将该工作区的地址注册到内部表中，不访问磁盘 I/O 层。卷装入过程是在 f_mount 函数后或存储介质改变后的第一次文件访问时完成的。

2.2 f_open

创建/打开一个用于访问文件的文件对象

```
FRESULT f_open (
    FIL* FileObject,          /* 空白文件对象结构指针 */
    const XCHAR* FileName,    /* 文件名指针 */
    BYTE ModeFlags            /* 模式标志 */
);
```

参数

FileObject 将被创建的文件对象结构的指针。

FileName

NULL 结尾的字符串指针，该字符串指定了将被创建或打开的文件名。

ModeFlags 指定文件的访问类型和打开方法。它是由下列标志的一个组合指定的。

模式	描述
FA_READ	指定读访问对象。可以从文件中读取数据。 与 FA_WRITE 结合可以进行读写访问。
FA_WRITE	指定写访问对象。可以向文件中写入数据。 与 FA_READ 结合可以进行读写访问。
FA_OPEN_EXISTING	打开文件。如果文件不存在，则打开失败。(默认)
FA_OPEN_ALWAYS	如果文件存在，则打开；否则，创建一个新文件。
FA_CREATE_NEW	创建一个新文件。如果文件已存在，则创建失败。
FA_CREATE_ALWAYS	创建一个新文件。如果文件已存在，则它将被截断并覆盖。

注意：当 _FS_READONLY == 1 时，模式标志 FA_WRITE, FA_CREATE_ALWAYS, FA_CREATE_NEW, FA_OPEN_ALWAYS 是无效的。

返回值

FR_OK (0) 函数成功，该文件对象有效。

FR_NO_FILE 找不到该文件。

FR_NO_PATH 找不到该路径。

FR_INVALID_NAME 文件名无效。

FR_INVALID_DRIVE 驱动器号无效。

FR_EXIST 该文件已存在。

FR_DENIED 由于下列原因，所需的访问被拒绝：

- 以写模式打开一个只读文件。
- 由于存在一个同名的只读文件或目录，而导致文件无法被创建。
- 由于目录表或磁盘已满，而导致文件无法被创建。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_WRITE_PROTECTED 在存储介质被写保护的情况下，以写模式打开或创建文件对象。

FR_DISK_ERR 由于底层磁盘 I/O 接口函数中的一个错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_ENABLED 逻辑驱动器没有工作区。

FR_NO_FILESYSTEM 磁盘上没有有效地 FAT 卷。

描述

如果函数成功，则创建一个文件对象。该文件对象被后续的读/写函数用来访问文件。如果想要关闭一个打开的文件对象，则使用 f_close 函数。如果不关闭修改后的文件，那么文件可能会崩溃。

在使用任何文件函数之前，必须使用 f_mount 函数为驱动器注册一个工作区。只有这样，其他文件函数

才能正常工作。

示例(文件拷贝)

```
void main (void)
{
    FATFS fs[2];          /* 逻辑驱动器的工作区(文件系统对象) */
    FIL fsrc, fdst;        /* 文件对象 */
    BYTE buffer[4096];     /* 文件拷贝缓冲区 */
    FRESULT res;           /* FatFs 函数公共结果代码 */
    UINT br, bw;           /* 文件读/写字节计数 */

    /* 为逻辑驱动器注册工作区 */
    f_mount(0, &fs[0]);
    f_mount(1, &fs[1]);

    /* 打开驱动器 1 上的源文件 */
    res = f_open(&fsrc, "1:srcfile.dat", FA_OPEN_EXISTING | FA_READ);
    if (res) die(res);

    /* 在驱动器 0 上创建目标文件 */
    res = f_open(&fdst, "0:dstfile.dat", FA_CREATE_ALWAYS | FA_WRITE);
    if (res) die(res);

    /* 拷贝源文件到目标文件 */
    for (;;) {
        res = f_read(&fsrc, buffer, sizeof(buffer), &br);
        if (res || br == 0) break; /* 文件结束错误 */
        res = f_write(&fdst, buffer, br, &bw);
        if (res || bw < br) break; /* 磁盘满错误 */
    }

    /* 关闭打开的文件 */
    f_close(&fsrc);
    f_close(&fdst);

    /* 注销工作区(在废弃前) */
    f_mount(0, NULL);
    f_mount(1, NULL);
}
```

2.3 f_close

关闭一个打开的文件

```
FRESULT f_close (  
    FIL* FileObject          /* 文件对象结构的指针 */  
);
```

参数

FileObject 指向将被关闭的已打开的文件对象结构的指针。

返回值

FR_OK (0) 文件对象已被成功关闭。>FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_INVALID_OBJECT 文件对象无效。

描述

f_close 函数关闭一个打开的文件对象。无论向文件写入任何数据，文件的缓存信息都将被写回到磁盘。该函数成功后，文件对象不再有效，并且可以被丢弃。如果文件对象是在只读模式下打开的，不需要使用该函数，也能被丢弃。

2.4 f_read

从一个文件读取数据

```
FRESULT f_read (
    FIL* FileObject,          /* 文件对象结构的指针 */
    void* Buffer,              /* 存储读取数据的缓冲区的指针 */
    UINT ByteToRead,          /* 要读取的字节数 */
    UINT* ByteRead             /* 返回已读取字节数变量的指针 */
);
```

参数

FileObject 指向将被读取的已打开的文件对象结构的指针。

Buffer 指向存储读取数据的缓冲区的指针。

ByteToRead 要读取的字节数，UINT 范围内。

ByteRead 指向返回已读取字节数的 UINT 变量的指针。在调用该函数后，无论结果如何，数值都是有效的。

返回值

FR_OK (0)函数成功。

FR_DENIED 由于文件是以非读模式打开的，而导致该函数被拒绝。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_INVALID_OBJECT 文件对象无效。

描述

文件对象中的读/写指针以已读取字节数增加。该函数成功后，应该检查 *ByteRead 来检测文件是否结束。在读操作过程中，一旦 *ByteRead < ByteToRead，则读/写指针到达了文件结束位置。

2.5 f_write

写入数据到一个文件

```
FRESULT f_write (  
    FIL* FileObject,          /* 文件对象结构的指针 */  
    const void* Buffer,       /* 存储写入数据的缓冲区的指针 */  
    UINT ByteToWrite,        /* 要写入的字节数 */  
    UINT* ByteWritten         /* 返回已写入字节数变量的指针 */  
);
```

参数

FileObject 指向将被写入的已打开的文件对象结构的指针。

Buffer 指向存储写入数据的缓冲区的指针。

ByteToWrite 要写入的字节数，UINT 范围内。

ByteWritten 指向返回已写入字节数的 UINT 变量的指针。在调用该函数后，无论结果如何，数值都是有效的。

返回值

FR_OK (0)函数成功。

FR_DENIED 由于文件是以非写模式打开的，而导致该函数被拒绝。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_INVALID_OBJECT 文件对象无效。

描述

文件对象中的读/写指针以已写入字节数增加。该函数成功后，应该检查 *ByteWritten 来检测磁盘是否已满。在写操作过程中，一旦 *ByteWritten < *ByteToWrite，则意味着该卷已满。

2.6 f_lseek

移动一个打开的文件对象的文件读/写指针。也可以被用来扩展文件大小(簇预分配)。

```
FRESULT f_lseek (
    FIL* FileObject,          /* 文件对象结构指针 */
    DWORD Offset              /* 文件字节偏移 */
);
```

参数

FileObject 打开的文件对象的指针

Offset 相对于文件起始处的字节数

返回值

FR_OK (0)函数成功。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_INVALID_OBJECT 文件对象无效。

描述

f_lseek 函数当 FS_MINIMIZE <= 2 时可用。

offset 只能被指定为相对于文件起始处的字节数。当在写模式下指定了一个超过文件大小的 offset 时，文件的大小将被扩展，并且该扩展的区域中的数据是未定义的。这适用于为快速写操作迅速地创建一个大的文件。f_lseek 函数成功后，为了确保读/写指针已被正确地移动，必须检查文件对象中的成员 fptr。如果 fptr 不是所期望的值，则发生了下列情况之一。

- 文件结束。指定的 offset 被钳在文件大小，因为文件已被以只读模式打开。
- 磁盘满。卷上没有足够的空闲空间去扩展文件大小。

示例

```
/* 移动文件读/写指针到相对于文件起始处偏移为 5000 字节处 */
res = f_lseek(file, 5000);
    /* 移动文件读/写指针到文件结束处，以便添加数据 */
res = f_lseek(file, file->fsize);
    /* 向前 3000 字节 */
res = f_lseek(file, file->fptr + 3000);
    /* 向后(倒带)2000 字节(注意溢出) */
res = f_lseek(file, file->fptr - 2000);
    /* 簇预分配(为了防止在流写时缓冲区上溢) */
res = f_open(file, recfile, FA_CREATE_NEW | FA_WRITE); /* 创建一个文件 */
res = f_lseek(file, PRE_SIZE); /* 预分配簇 */
if (res || file->fptr != PRE_SIZE) ... /* 检查文件大小是否已被正确扩展 */
    res = f_lseek(file, DATA_START); /* 没有簇分配延迟地记录数据流 */
...
    res = f_truncate(file); /* 截断未使用的区域 */
    res = f_lseek(file, 0); /* 移动到文件起始处 */
...
res = f_close(file);
```

2.7 f_truncate

截断文件大小

```
FRESULT f_truncate (  
    FIL* FileObject          /* 文件对象结构指针 */  
);
```

参数

FileObject 待截断的打开的文件对象的指针。

返回值

FR_OK (0 函数成功。

FR_DENIED 由于文件是以非写模式打开的，而导致该函数被拒绝。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_INVALID_OBJECT 文件对象无效。

描述

f_truncate 函数当 _FS_READONLY == 0 并且 _FS_MINIMIZE == 0 时可用。

f_truncate 函数截断文件到当前的文件读/写指针。当文件读/写指针已经指向文件结束时，该函数不起作用。

2.8 f_sync

冲洗一个写文件的缓存信息

```
FRESULT f_sync (  
    FIL* FileObject          /* 文件对象结构的指针 */  
);
```

参数

FileObject 待冲洗的打开的文件对象的指针。

返回值

FR_OK (0)函数成功。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_INVALID_OBJECT 文件对象无效。

描述

f_sync 函数当_FS_READONLY == 0 时可用。

f_sync 函数和 f_close 函数执行同样的过程，但是文件仍处于打开状态，并且可以继续对文件执行读/写/移动指针操作。这适用于以写模式长时间打开文件，比如数据记录器。定期的或 f_write 后立即执行 f_sync 可以将由于突然断电或移去磁盘而导致数据丢失的风险最小化。在 f_close 前立即执行 f_sync 没有作用，因为在 f_close 中执行了 f_sync。换句话说，这两个函数的差异就是文件对象是不是无效的。

2.9 f_opendir

打开一个目录

```
FRESULT f_opendir (  
    DIR* DirObject,          /* 空白目录对象结构的指针 */  
    const XCHAR* DirName     /* 目录名的指针 */  
);
```

参数

DirObject 待创建的空白目录对象的指针。

DirName 以 '\0' 结尾的字符串指针，该字符串指定了将被打开的目录名。

返回值

FR_OK (0) 函数成功，目录对象被创建。该目录对象被后续调用，用来读取目录项。

FR_NO_PATH 找不到路径。

FR_INVALID_NAME 路径名无效。

FR_INVALID_DRIVE 驱动器号无效。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_ENABLED 逻辑驱动器没有工作区。

FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

描述

f_opendir 函数当 _FS_MINIMIZE <= 1 时可用。

f_opendir 函数打开一个已存在的目录，并为后续的调用创建一个目录对象。该目录对象结构可以在任何时候不经任何步骤而被丢弃。

2.10 f_readdir

读取目录项

```
FRESULT f_readdir (
    DIR* DirObject,          /* 指向打开的目录对象结构的指针 */
    FILINFO* FileInfo        /* 指向文件信息结构的指针 */
);
```

参数

DirObject 打开的目录对象的指针。

FileInfo 存储已读取项的文件信息结构指针。

返回值

FR_OK (0)函数成功。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_INVALID_OBJECT 文件对象无效。

描述

f_readdir 函数当_FS_MINIMIZE <= 1 时可用。

f_readdir 函数顺序读取目录项。目录中的所有项可以通过重复调用 f_readdir 函数被读取。当所有目录项已被读取并且没有项要读取时，该函数没有任何错误地返回一个空字符串到 f_name[]成员中。当 FileInfo 给定一个空指针时，目录对象的读索引将被回绕。

当 LFN 功能被使能时，在使用 f_readdir 函数之前，文件信息结构中的 lfname 和 lfsize 必须被初始化为有效数值。lfname 是一个返回长文件名的字符串缓冲区指针。lfsize 是以字符为单位的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)不足，或者对象没有 LFN，则一个空字符串将被返回到 LFN 读缓冲区。如果 LFN 包含任何不能被转换为 OEM 代码的字符，则一个空字符串将被返回，但是这不是 Unicode API 配置的情况。当 lfname 是一个空字符串时，没有 LFN 的任何数据被返回。当对象没有 LFN 时，任何小型大写字母可以被包含在 SFN 中。

当相对路径功能被使能(_FS_RPATH == 1)时，"."和".."目录项不会被过滤掉，并且它将出现在读目录项中。

示例

```
FRESULT scan_files (
    char* path                /* Start node to be scanned (also used as work area) */
)
{
    FRESULT res;
    FILINFO fno;
    DIR dir;
    int i;
    char *fn;    /* This function is assuming non-Unicode cfg. */
    #if _USE_LFN
        static char lfn[_MAX_LFN + 1];
        fno.lfname = lfn;
        fno.lfsize = sizeof lfn;
    #endif
}
```

```
res = f_opendir(&dir, path);                /* Open the directory */
if (res == FR_OK) {
    i = strlen(path);
    for (;;) {
        res = f_readdir(&dir, &fno);        /* Read a directory item */
        if (res != FR_OK || fno.fname[0] == 0) break; /* Break on error or end of dir */
        if (fno.fname[0] == '.') continue;    /* Ignore dot entry */
#if _USE_LFN
        fn = *fno.lfname ? fno.lfname : fno.fname;
#else
        fn = fno.fname;
#endif
        if (fno.fattrib & AM_DIR) {          /* It is a directory */
            sprintf(&path[i], "%s", fn);
            res = scan_files(path);
            if (res != FR_OK) break;
            path[i] = 0;
        } else {                            /* It is a file. */
            printf("%s/%s\n", path, fn);
        }
    }
}

return res;
}
```

2.11 f_getfree

获取空闲簇的数目

```
FRESULT f_getfree (
    const XCHAR* Path,          /* 驱动器的根目录 */
    DWORD* Clusters,           /* 存储空闲簇数目变量的指针 */
    FATFS** FileSystemObject /* 文件系统对象指针的指针 */
);
```

参数

Path\0'结尾的字符串指针，该字符串指定了逻辑驱动器的目录。

Clusters 存储空闲簇数目的 DWORD 变量的指针。

FileSystemObject 相应文件系统对象指针的指针。

返回值

FR_OK (0)函数成功。*Clusters 表示空闲簇的数目，并且*FileSystemObject 指向文件系统对象。

FR_INVALID_DRIVE 驱动器号无效。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_ENABLED 逻辑驱动器没有工作区。

FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

描述

f_getfree 函数当_FS_READONLY == 0 并且_FS_MINIMIZE == 0 时有效。

f_getfree 函数获取驱动器上空闲簇的数目。文件系统对象中的成员 csize 是每簇中的扇区数，因此，以扇区为单位的空闲空间可以被计算出来。当 FAT32 卷上的 FSInfo 结构不同步时，该函数返回一个错误的空闲簇计数。

示例

```
FATFS *fs;
    DWORD fre_clust, fre_sect, tot_sect;

    /* Get drive information and free clusters */
    res = f_getfree("/", &fre_clust, &fs);
    if (res) die(res);

    /* Get total sectors and free sectors */
    tot_sect = (fs->max_clust - 2) * fs->csize;
    fre_sect = fre_clust * fs->csize;

    /* Print free space in unit of KB (assuming 512B/sector) */
    printf("%lu KB total drive space.\n"
           "%lu KB available.\n",
           fre_sect / 2, tot_sect / 2);
```

2.12 f_stat

获取文件状态

```
FRESULT f_stat (  
    const XCHAR* FileName,    /* 文件名或目录名的指针 */  
    FILINFO* FileInfo         /* FILINFO 结构的指针 */  
);
```

参数

FileName 以 '\0' 结尾的字符串指针，该字符串指定了待获取其信息的文件或目录。

FileInfo 存储信息的空白 FILINFO 结构的指针。

返回值

FR_OK (0) 函数成功。

FR_NO_FILE 找不到文件或目录。

FR_NO_PATH 找不到路径。

FR_INVALID_NAME 路径名无效。

FR_INVALID_DRIVE 驱动器号无效。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_ENABLED 逻辑驱动器没有工作区。

FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

描述

f_stat 函数当 _FS_MINIMIZE == 0 时可用。

f_stat 函数获取一个文件或目录的信息。信息的详情，请参考 FILINFO 结构和 f_readdir 函数。

2.13 f_mkdir

创建一个目录

```
FRESULT f_mkdir (
    const XCHAR* DirName /* 目录名的指针 */
);
```

参数

DirName 以 '\0' 结尾的字符串指针，该字符串指定了待创建的目录名。

返回值

FR_OK (0) 函数成功。

FR_NO_PATH 找不到路径。

FR_INVALID_NAME 路径名无效。

FR_INVALID_DRIVE 驱动器号无效。

FR_DENIED 由于目录表或磁盘满，而导致目录不能被创建。

FR_EXIST 已经存在同名的文件或目录。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_WRITE_PROTECTED 存储介质被写保护。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_ENABLED 逻辑驱动器没有工作区。

FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

描述

f_mkdir 函数当 _FS_READONLY == 0 并且 _FS_MINIMIZE == 0 时可用。

f_mkdir 函数创建一个新目录。

示例

```
res = f_mkdir("sub1");
if (res) die(res);
res = f_mkdir("sub1/sub2");
if (res) die(res);
res = f_mkdir("sub1/sub2/sub3");
if (res) die(res);
```

2.14 f_unlink

移除一个对象

```
FRESULT f_unlink (  
    const XCHAR* FileName /* 对象名的指针 */  
);
```

参数

FileName 以 '\0' 结尾的字符串指针，该字符串指定了一个待移除的对象。

返回值

FR_OK (0) 函数成功。

FR_NO_FILE 找不到文件或目录。

FR_NO_PATH 找不到路径。

FR_INVALID_NAME 路径名无效。

FR_INVALID_DRIVE 驱动器号无效。

FR_DENIED 由于下列原因之一，而导致该函数被拒绝：

对象具有只读属性

目录不是空的

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_WRITE_PROTECTED 存储介质被写保护。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_ENABLED 逻辑驱动器没有工作区。

FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

描述

f_unlink 函数当 _FS_READONLY == 0 并且 _FS_MINIMIZE == 0 时可用。

f_unlink 函数移除一个对象。不要移除打开的对象或当前目录。

2.15 f_chmod

修改一个文件或目录的属性。

```
FRESULT f_chmod (
    const XCHAR* FileName, /* 文件或目录的指针 */
    BYTE Attribute,        /* 属性标志 */
    BYTE AttributeMask     /* 属性掩码 */
);
```

参数

FileName"0"结尾的字符串指针，该字符串指定了一个待被修改属性的文件或目录。

Attribute 待被设置的属性标志，可以是下列标志的一个或任意组合。指定的标志被设置，其他的被清除。

属性	描述
AM_RDO	只读
AM_ARC	存档
AM_SYS	系统
AM_HID	隐藏

AttributeMask 属性掩码，指定修改哪个属性。指定的属性被设置或清除。

返回值

FR_OK (0)函数成功。

FR_NO_FILE 找不到文件或目录。

FR_NO_PATH 找不到路径。

FR_INVALID_NAME 路径名无效。

FR_INVALID_DRIVE 驱动器号无效。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_WRITE_PROTECTED 存储介质被写保护。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_ENABLED 逻辑驱动器没有工作区。

FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

描述

f_chmod 函数当_FS_READONLY == 0 并且_FS_MINIMIZE == 0 时可用。

f_chmod 函数修改一个文件或目录的属性。

示例

```
// 设置只读标志，清除存档标志，其他不变
f_chmod("file.txt", AR_RDO, AR_RDO | AR_ARC);
```

2.16 f_utime

f_utime 函数修改一个文件或目录的时间戳。

```
FRESULT f_utime (
    const XCHAR* FileName, /* 文件或目录路径的指针 */
    const FILINFO* TimeDate /* 待设置的时间和日期 */
);
```

参数

FileName 以 '\0' 结尾的字符串的指针，该字符串指定了一个待修改时间戳的文件或目录。

TimeDate 文件信息结构指针，其中成员 ftime 和 fdata 存储了一个待被设置的时间戳。不关心任何其他成员。

返回值

FR_OK (0) 函数成功。

FR_NO_FILE 找不到文件或目录。

FR_NO_PATH 找不到路径。

FR_INVALID_NAME 路径名无效。

FR_INVALID_DRIVE 驱动器号无效。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_WRITE_PROTECTED 存储介质被写保护。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_ENABLED 逻辑驱动器没有工作区。

FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

描述

f_utime 函数当 _FS_READONLY == 0 并且 _FS_MINIMIZE == 0 时可用。

f_utime 函数修改一个文件或目录的时间戳。

2.17 f_rename

重命名一个对象。

```
FRESULT f_rename (
    const XCHAR* OldName, /* 原对象名的指针 */
    const XCHAR* NewName  /* 新对象名的指针 */
);
```

参数

OldName\0'结尾的字符串的指针，该字符串指定了待被重命名的原对象名。

NewName\0'结尾的字符串的指针，该字符串指定了重命名后的新对象名，不能包含驱动器号。

返回值

FR_OK (0)函数成功。

FR_NO_FILE 找不到原名。

FR_NO_PATH 找不到路径。

FR_INVALID_NAME 文件名无效。

FR_INVALID_DRIVE 驱动器号无效。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_EXIST 新名和一个已存在的对象名冲突。

FR_DENIED 由于任何原因，而导致新名不能被创建。

FR_WRITE_PROTECTED 存储介质被写保护。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_ENABLED 逻辑驱动器没有工作区。

FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

描述

f_rename 函数当_FS_READONLY == 0 并且_FS_MINIMIZE == 0 时可用。

f_rename 函数重命名一个对象，并且也可以将对象移动到其他目录。逻辑驱动器号由原名决定，新名不能包含一个逻辑驱动器号。不要重命名打开的对象。

示例

```
/* 重命名一个对象 */
f_rename("oldname.txt", "newname.txt");

/* 重命名并且移动一个对象到另一个目录 */
f_rename("oldname.txt", "dir1/newname.txt");
```

2.18 f_chdir

f_chdir 函数改变一个驱动器的当前目录。

```
FRESULT f_chdir (
    const XCHAR* Path /* 路径名的指针 */
);
```

参数

Path\0'结尾的字符串的指针，该字符串指定了将要进去的目录。

返回值

FR_OK (0)函数成功。

FR_NO_PATH 找不到路径。

FR_INVALID_NAME 路径名无效。

FR_INVALID_DRIVE 驱动器号无效。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_ENABLED 逻辑驱动器没有工作区。

FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

描述

f_chdir 函数当_FS_RPATH == 1 时可用。

f_chdir 函数改变一个逻辑驱动器的当前目录。当一个逻辑驱动器被自动挂载时，它的当前目录被初始化为根目录。注意：当前目录被保存在每个文件系统对象中，因此它也影响使用同一逻辑驱动器的其它任务。

示例

```
// 改变当前驱动器的当前目录(根目录下的 dir1)
f_chdir("/dir1");
// 改变驱动器 2 的当前目录(父目录)
f_chdir("2:..");
```

2.19 f_chdrive

f_chdrive 函数改变当前驱动器。

```
FRESULT f_chdrive (  
    BYTE Drive /* 逻辑驱动器号 */  
);
```

参数

Drive 指定将被设置为当前驱动器的逻辑驱动器号。

返回值

FR_OK (0) 函数成功。

FR_INVALID_DRIVE 驱动器号无效。

描述

f_chdrive 函数当_FS_RPATH == 1 时可用。

f_chdrive 函数改变当前驱动器。当前驱动器号初始值为 0，注意：当前驱动器被保存为一个静态变量，因此它也影响使用文件函数的其它任务。

2.20 f_getcwd

恢复当前目录.

```
FRESULT f_getcwd (
    TCHAR* Buffer, /* Pointer to the buffer */
    UINT BufferLen /* The length of the buffer */
);
```

参数

Buffer——指向接收当前目录字符串的缓冲区

BufferLen——缓冲区的大小，单位为 TCHAR

返回值

FR_OK (0)函数成功。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_ENABLED 逻辑驱动器没有工作区。

FR_NO_FILESYSTEM 磁盘上没有有效的 FAT 卷。

FR_TIMEOUT 函数由于线程安全控制超时而退出。（相关选项：_TIMEOUT）

FR_NOT_ENOUGH_CORE 没有足够的内存进行操作。原因有：

- 不能为 LFN 工作缓冲区分配内存（相关选项：_USE_LFN）。
- 得到的表大小不能满足要求的大小。

描述

f_getcwd 函数用完整的路径字符串（包括驱动器号）来恢复当前驱动器上的当前目录。

提示

在_FS_RPATH == 2 时可用。

2.21 f_forward

读取文件数据并将其转发到数据流设备。

```
FRESULT f_forward (
    FIL* FileObject,          /* 文件对象 */
    UINT (*Func)(const BYTE*,UINT), /* 数据流函数 */
    UINT ByteToFwd,          /* 要转发的字节数 */
    UINT* ByteFwd            /* 已转发的字节数 */
);
```

参数

FileObject 打开的文件对象的指针。

Func 用户定义的数据流函数的指针。详情参考示例代码。

ByteToFwd 要转发的字节数，UINT 范围内。

ByteFwd 返回已转发的字节数的 UINT 变量的指针。

返回值

FR_OK (0)函数成功。

FR_DENIED 由于文件已经以非读模式打开，而导致函数失败。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INT_ERR 由于一个错误的 FAT 结构或一个内部错误，而导致该函数失败。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_INVALID_OBJECT 文件对象无效。

描述

f_forward 函数当_USE_FORWARD == 1 并且_FS_TINY == 1 时可用。

f_forward 函数从文件中读取数据并将数据转发到输出流，而不使用数据缓冲区。这适用于小存储系统，因为它在应用模块中不需要任何数据缓冲区。文件对象的文件指针以转发的字节数增加。如果*ByteFwd < ByteToFwd 并且没有错误，则意味着由于文件结束或在数据传输过程中流忙，请求的字节不能被传输。

示例(音频播放)

```
/*-----*/
/* 示例代码：数据传输函数，将被 f_forward 函数调用 */
/*-----*/

UINT out_stream ( /* 返回已发送字节数或流状态 */
    const BYTE *p, /* 将被发送的数据块的指针 */
    UINT btf       /* >0: 传输调用(将被发送的字节数)。0: 检测调用 */
)
{
    UINT cnt = 0;
    if (btf == 0) { /* 检测调用 */
        /* 返回流状态(0: 忙, 1: 就绪) */
        /* 当检测调用时，一旦它返回就绪，那么在后续的传输调用时，它必须接收至少一个字节，或者
        f_forward 将以 FR_INT_ERROR 而失败。 */
        if (FIFO_READY) cnt = 1;
    }
    else { /* 传输调用 */
        do { /* 当有数据要发送并且流就绪时重复 */
```

```

        FIFO_PORT = *p++;
cnt++;
    } while (cnt < btf && FIFO_READY);
}

return cnt;
}

/*-----*/
/* 示例代码：使用 f_forward 函数 */
/*-----*/

FRESULT play_file (
    char *fn          /* 待播放的音频文件名的指针 */
)
{
    FRESULT rc;
    FIL fil;
    UINT dmy;

    /* 以只读模式打开音频文件 */
rc = f_open(&fil, fn, FA_READ);

    /* 重复，直到文件指针到达文件结束位置 */
while (rc == FR_OK && fil.fptr < fil.fsize) {

        /* 任何其他处理... */

        /* 定期或请求式填充输出流 */
rc = f_forward(&fil, out_stream, 1000, &dmy);
    }

    /* 该只读的音频文件对象不需要关闭就可以被丢弃 */
return rc;
}

```

2.22 f_mkfs

在驱动器上创建一个文件系统

```
FRESULT f_mkfs (
    BYTE  Drive,           /* 逻辑驱动器号 */
    BYTE  PartitioningRule, /* 分区规则 */
    WORD  AllocSize        /* 分配单元大小 */
);
```

参数

Drive 待格式化的逻辑驱动器号(0-9)。

PartitioningRule 当给定 0 时, 首先在驱动器上的第一个扇区创建一个分区表, 然后文件系统被创建在分区上。这被称为 **FDISK** 格式化, 用于硬盘和存储卡。当给定 1 时, 文件系统从第一个扇区开始创建, 而没有分区表。这被称为超级软盘(**SFD**)格式化, 用于软盘和可移动磁盘。

AllocSize 指定每簇中以字节为单位的分配单元大小。数值必须是 0 或从 512 到 32K 之间 2 的幂。当指定 0 时, 簇大小取决于卷大小。

返回值

FR_OK (0)函数成功。

FR_INVALID_DRIVE 驱动器号无效。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因, 而导致磁盘驱动器无法工作。

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 由于底层磁盘 I/O 函数中的错误, 而导致该函数失败。

FR_MKFS_ABORTED 由于下列原因之一, 而导致函数在开始格式化前终止:

- 磁盘容量太小
- 参数无效
- 该驱动器不允许的簇大小。

描述

f_mkfs 函数当 `_FS_READONLY == 0` 并且 `_USE_MKFS == 1` 时可用。

f_mkfs 函数在驱动器中创建一个 **FAT** 文件系统。对于可移动媒介, 有两种分区规则: **FDISK** 和 **SFD**, 通过参数 PartitioningRule 选择。**FDISK** 格式在大多数情况下被推荐使用。该函数当前不支持多分区, 因此, 物理驱动器上已存在的分区将被删除, 并且重新创建一个占据全部磁盘空间的新分区。

根据 Microsoft 发布的 **FAT** 规范, **FAT** 分类: **FAT12/FAT16/FAT32**, 由驱动器上的簇数决定。因此, 选择哪种 **FAT** 分类, 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, 并且大簇会提高性能。

2.23 f_fdisk

划分一个物理驱动器。

```
FRESULT f_fdisk (
    BYTE Drive,           /* Physical drive number */
    const DWORD Partitions[], /* Partition size */
    void* Work            /* Work area */
);
```

参数

Drive——指定要划分的物理驱动器

Partitions[]——分区映象表，必须有四个项目。

Work——指向函数工作区的指针。其大小必须至少为_MAX_SS 字节。

返回值

FR_OK (0)函数成功。

FR_NOT_READY 由于驱动器中没有存储介质或任何其他原因，而导致磁盘驱动器无法工作。

FR_WRITE_PROTECTED 驱动器被写保护。

FR_DISK_ERR 由于底层磁盘 I/O 函数中的错误，而导致该函数失败。

FR_INVALID_PARAMETER 所给参数无效或不一致。

描述

f_fdisk 函数创建一个分区表到物理驱动器的 MBR。分区规则为通用 FDISK 格式，所以可以创建多达四个主分区。不支持扩展分区。Partitions[]指定了如何划分物理驱动器。第一个项目指定第一个主分区的大小，第四个项目指定第四个主分区。如果其值小于或等于 100，表示分区占整个磁盘空间的百分比。如果大于 100，则表示以扇区为单位的分区大小。

提示

在_FS_READOLNY == 0、_USE_MKFS == 1 并且_MULTI_PARTITION == 2 时可用。

示例

```
/* Volume management table defined by user (required when _MULTI_PARTITION != 0) */
PARTITION VolToPart[] = {
    {0, 1},    /* Logical drive 0 ==> Physical drive 0, 1st partition */
    {0, 2},    /* Logical drive 1 ==> Physical drive 0, 2nd partition */
    {1, 0}     /* Logical drive 2 ==> Physical drive 1, auto detection */
};

/* Initialize a brand-new disk drive mapped to physical drive 0 */
FATFS Fatfs;
DWORD plist[] = {50, 50, 0, 0}; /* Divide drive into two partitions */
BYTE work[_MAX_SS];
f_fdisk(0, plist, work); /* Divide physical drive 0 */
f_mount(0, &Fatfs);
f_mkfs(0, 0, 0);          /* Create an FAT volume on the logical drive 0. 2nd argument is ignored. */
f_mount(0, 0);
f_mount(1, &Fatfs);
f_mkfs(1, 0, 0);
f_mount(1, 0);
```

2.24 f_gets

f_gets 从文件中读取一个字符串。

```
char* f_gets (  
    char* Str,          /* 读缓冲区 */  
    int Size,           /* 读缓冲区大小 */  
    FIL* FileObject     /* 文件对象 */  
);
```

参数

Str 存储读取字符串的读缓冲区指针。

Size 读缓冲区大小。

FileObject 打开的文件对象结构指针。

返回值当函数成功后，Str 将被返回。

描述

f_gets 函数当_USE_STRFUNC == 1 或者_USE_STRFUNC == 2 时可用。如果_USE_STRFUNC == 2，文件中包含的'\r'则被去除。

f_gets 函数是 f_read 的一个封装函数。当读取到'\n'、文件结束或缓冲区被填满了 Size - 1 个字符时，读操作结束。读取的字符串以'\0'结束。当文件结束或读操作中发生了任何错误，f_gets()返回一个空字符串。可以使用宏 f_eof()和 f_error()检查 EOF 和错误状态。

2.25 f_putc

f_putc 函数向文件中写入一个字符。

```
int f_putc (  
    int Chr,          /* 字符 */  
    FIL* FileObject  /* 文件对象 */  
);
```

参数

Chr 待写入的字符。

FileObject 打开的文件对象结构的指针。

返回值

当字符被成功地写入后，函数返回该字符。由于磁盘满或任何错误而导致函数失败，将返回 EOF。

描述

f_putc 函数当(_FS_READONLY == 0)&&(_USE_STRFUNC == 1 || _USE_STRFUNC == 2)时可用。当 _USE_STRFUNC == 2 时，字符'\n'被转换为"\r\n"写入文件中。

f_putc 函数是 f_write 的一个封装函数。

2.26 f_puts

f_puts 函数向文件中写入一个字符串。

```
int f_puts (  
    const char* Str, /* 字符串指针 */  
    FIL* FileObject /* 文件对象指针 */  
);
```

参数

Str——待写入的'\0'结尾的字符串的指针。'\0'字符不会被写入。

FileObject——打开的文件对象结构的指针。

返回值

函数成功后，将返回写入的字符数。由于磁盘满或任何错误而导致函数失败，将返回 EOF。

描述

f_puts()当(_FS_READONLY == 0)&&(_USE_STRFUNC == 1 || _USE_STRFUNC == 2)时可用。当 _USE_STRFUNC == 2 时，字符串中的'\n'被转换为"\r\n"写入文件中。

f_puts()是 f_putc()的一个封装函数。

2.27 f_printf

f_printf 函数向文件中写入一个格式化字符串。

```
int f_printf (
    FIL* FileObject,      /* 文件对象指针 */
    const char* Format,    /* 格式化字符串指针 */
    ...
);
```

参数

FileObject——已打开的文件对象结构的指针。

Format——'\0'结尾的格式化字符串指针。

... ——可选参数

返回值

函数成功后，将返回写入的字符数。由于磁盘满或任何错误而导致函数失败，将返回 EOF。

描述

f_printf 函数当(_FS_READONLY == 0)&&(_USE_STRFUNC == 1 || _USE_STRFUNC == 2)时可用。当 _USE_STRFUNC == 2 时，包含在格式化字符串中的'\n'将被转换成"\r\n"写入文件中。

f_printf 函数是 f_putc 和 f_puts 的一个封装函数。如下所示，格式控制符是标准库的一个子集：

- 类型：c s d u X
- 大小：1
- 标志：0

示例

```
f_printf(&fil, "%d", 1234);           /* "1234" */
f_printf(&fil, "%6d,%3d%%", -200, 5); /* " -200,  5%" */
f_printf(&fil, "%-6u", 100);          /* "100   " */
f_printf(&fil, "%ld", 12345678L);      /* "12345678" */
f_printf(&fil, "%04x", 0xA3);          /* "00a3" */
f_printf(&fil, "%08LX", 0x123ABC);     /* "00123ABC" */
f_printf(&fil, "%016b", 0x550F);       /* "0101010100001111" */
f_printf(&fil, "%s", "String");       /* "String" */
f_printf(&fil, "%-4s", "abc");         /* "abc " */
f_printf(&fil, "%4s", "abc");         /* " abc" */
f_printf(&fil, "%c", 'a');             /* "a" */
f_printf(&fil, "%f", 10.0);            /* f_printf lacks floating point support */
```

2.28 f_tell

获取一个文件的当前读/写指针。

```
DWORD f_tell (  
    FIL* FileObject    /* File object */  
);
```

参数

FileObject——指向打开文件对象结构的指针。

返回值

返回文件的当前读/写指针。

描述

在这个版本里，f_tell 函数是以一个宏来实现的。

```
#define f_tell(fp) ((fp)->fptr)
```

提示

总是可用。

2.29 f_eof

测试一个文件的文件末尾。

```
int f_eof(  
    FIL* FileObject    /* File object */  
);
```

参数

FileObject——指向打开文件对象结构的指针。

返回值

如果读/写指针到达文件末尾，f_eof 函数返回一个非零值；否则返回 0。

描述

在这个版本里，f_eof 函数是以一个宏来实现的。

```
#define f_eof(fp) (((fp)->fptr) == ((fp)->fsize) ? 1 : 0)
```

提示

总是可用。

2.30 f_size

获取一个文件的大小。

```
DWORD f_size (  
    FIL* FileObject    /* File object */  
);
```

参数

FileObject——指向打开文件对象结构的指针。

返回值

返回文件的大小，单位为字节。

描述

在这个版本里，f_size 函数是以一个宏来实现的。

```
#define f_size(fp) ((fp)->fsize)
```

提示

总是可用。

2.31 f_error

测试文件是否出错。

```
int f_error (  
    FIL* FileObject    /* File object */  
);
```

参数

FileObject——指向打开文件对象结构的指针。

返回值

如果有错误返回非零值；否则返回 0。

描述

在这个版本里，f_error 函数是以一个宏来实现的。

```
#define f_error(fp) (((fp)->flag & FA__ERROR) ? 1 : 0)
```

提示

总是可用。

3、磁盘 I/O 接口

由于 FatFs 模块完全与磁盘 I/O 层分开，因此底层磁盘 I/O 需要下列函数去读/写物理磁盘以及获取当前时间。由于底层磁盘 I/O 模块并不是 FatFs 的一部分，因此它必须由用户提供。

3.1 disk_initialize

初始化磁盘驱动器

```
DSTATUS disk_initialize (
    BYTE Drive          /* 物理驱动器号 */
);
```

参数

Drive 指定待初始化的物理驱动器号。

返回值

disk_initialize 函数返回一个磁盘状态作为结果。磁盘状态的详情，参考 disk_status 函数。

描述

disk_initialize 函数初始化一个物理驱动器。函数成功后，返回值中的 STA_NOINIT 标志被清除。

disk_initialize 函数被 FatFs 模块在卷挂载过程中调用，去管理存储介质的改变。当 FatFs 模块起作用时，或卷上的 FAT 结构可以被瓦解时，应用程序不能调用该函数。可以使用 f_mount 函数去重新初始化文件系统。

3.2 disk_status

获取当前磁盘的状态

```
DSTATUS disk_status (
    BYTE Drive          /* 物理驱动器号*/
);
```

参数

Drive 指定待确认的物理驱动器号。

返回值

磁盘状态，是下列标志的组合：STA_NOINIT

指示磁盘驱动器还没有被初始化。当系统复位、磁盘移除和 disk_initialize 函数失败时，该标志被设置；

当 disk_initialize 函数成功时，该标志被清除。STA_NODISK

指示驱动器中没有存储介质。当安装了磁盘驱动器后，该标志始终被清除。

STA_PROTECTED

指示存储介质被写保护。在不支持写保护缺口的驱动器上，该标志始终被清除。当 STA_NODISK 被设置时，该标志无效。

3.3 disk_read

从磁盘驱动器中读取扇区

```
DRESULT disk_read (
    BYTE Drive,          /* 物理驱动器号 */
    BYTE* Buffer,         /* 读取数据缓冲区的指针 */
    DWORD SectorNumber,  /* 起始扇区号 */
    BYTE SectorCount     /* 要读取的扇区数 */
);
```

参数

Drive 指定物理驱动器号。

Buffer 存储读取数据的缓冲区的指针。该缓冲区大小需要满足要读取的字节数(扇区大小 * 扇区总数)。

由上层指定的存储器地址可能会也可能不会以字边界对齐。SectorNumber

指定在逻辑块地址(LBA)中的起始扇区号。

SectorCount 指定要读取的扇区数(1-255)。

返回值

RES_OK (0)函数成功

RES_ERROR 在读操作过程中发生了不能恢复的硬错误。

RES_PARERR 无效的参数。

RES_NOTRDY 磁盘驱动器还没被初始化。

3.4 disk_write

向磁盘驱动器中写入扇区

```
DRESULT disk_write (
    BYTE Drive,          /* 物理驱动器号 */
    const BYTE* Buffer,   /* 写入数据缓冲区的指针(可能未对齐) */
    DWORD SectorNumber,  /* 起始扇区号 */
    BYTE SectorCount     /* 要写入的扇区数 */
);
```

参数

Drive 指定物理驱动器号。

Buffer 存储写入数据的缓冲区的指针。由上层指定的存储器地址可能会也可能不会以字边界对齐。

SectorNumber 指定在逻辑块地址(LBA)中的起始扇区号。

SectorCount 指定要写入的扇区数(1-255)。

返回值

RES_OK (0)函数成功

RES_ERROR 在读操作过程中发生了不能恢复的硬错误。

RES_WRPRT 存储介质被写保护。

RES_PARERR 无效的参数。

RES_NOTRDY 磁盘驱动器还没被初始化。

描述

在只读配置中，不需要此函数。

3.5 disk_ioctl

控制设备特定的功能以及磁盘读写以外的其它功能。

```
DRESULT disk_ioctl (
    BYTE Drive,      /* 驱动器号 */
    BYTE Command,    /* 控制命令代码 */
    void* Buffer      /* 数据传输缓冲区 */
);
```

参数

Drive 指定驱动器号(1-9)。

Command 指定命令代码。

Buffer 取决于命令代码的参数缓冲区的指针。当不使用时，指定一个 NULL 指针。

返回值

RES_OK (0)函数成功。

RES_ERROR 发生错误。

RES_PARERR 无效的命令代码。

RES_NOTRDY 磁盘驱动器还没被初始化。

描述

FatFs 模块只使用下述与设备无关的命令，没有使用任何设备相关功能。

命令	描述
CTRL_SYNC	确保磁盘驱动器已经完成等待写过程。当磁盘 I/O 模块有一个写回高速缓存时，立即冲洗脏扇区。在只读配置中，不需要该命令。
GET_SECTOR_SIZE	返回驱动器的扇区大小赋给 Buffer 指向的 WORD 变量。在单个扇区大小配置中(_MAX_SS 为 512)，不需要该命令。
GET_SECTOR_COUNT	返回总扇区数赋给 Buffer 指向的 DWORD 变量。只在 f_mkfs 函数中，使用了该命令。
GET_BLOCK_SIZE	返回以扇区为单位的存储阵列的擦除块大小赋给 Buffer 指向的 DWORD 变量。当擦除块大小未知或是磁盘设备时，返回 1。只在 f_mkfs 函数中，使用了该命令。

3.6 get_fattime

获取当前时间

`DWORD get_fattime (void);`

参数

void

返回值

返回的当前时间被打包进一个 DWORD 数值。各位域定义如下：

bit31:25 年，从 1980 年开始算起(0..127)

bit24:21 月(1..12)

bit20:16 日(1..31)

bit15:11 时(0..23)

bit10:5 分(0..59)

bit4:0 秒/2(0..29)，由此可见 FatFs 的时间分辨率为 2 秒

描述

get_fattime 函数必须返回任何有效的的时间，即使系统不支持实时时钟。如果返回一个 0，则文件将没有一个有效的的时间。在只读配置中，不需要此函数。

4、FatFs 模块应用

4.1 如何移植

4.1.1 基本概念

FatFs 模块在移植性方面做了以下假设：

- ANSI C

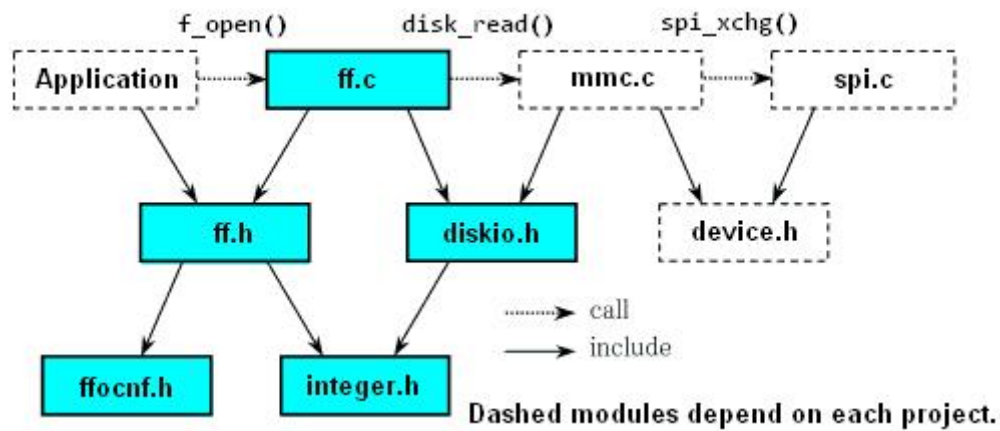
FatFs 模块是用 ANSI C (C89)编写的中间件。只要编译器与 ANSI C 兼容，则与平台无关

- 整数类型的大小

FatFs 模块假定 char/short/long 的大小为 8/16/32 位，而 int 为 16 位或 32 位。这些在 integer 中定义。这在大多数编译器上是没有总是的。如果这些定义有任何冲突，则必须仔细地解决。

4.1.2 系统组织

下面的依赖图给出了使用 FatFs 模块在嵌入式系统的典型配置。



4.1.3 需要哪些函数？

只需要提供 FatFs 模块需要的底层的磁盘 I/O 函数即可。如果已经存在一个目标系统的工作磁盘模块，则只需要写聚合函数附加到 FatFs 模块即可。如果没有，则需要移植任何其他磁盘模块或写一个。所有已经定义的函数不需要再写了。例如，磁盘写函数在只读配置方面就不需要。下表给出了根据配置选项需要哪些函数。

函数	什么时候需要	备注
disk_initialize disk_status disk_read	Always	磁盘 I/O 函数。 在 ffsample.zip 中有示例。 网站有许多实现实例。
disk_write get_fattime disk_ioctl (CTRL_SYNC)	_FS_READONLY == 0	
disk_ioctl (GET_SECTOR_COUNT) disk_ioctl (GET_BLOCK_SIZE)	_USE_MKFS == 1	
disk_ioctl (GET_SECTOR_SIZE)	_MAX_SS > 512	
disk_ioctl (CTRL_ERASE_SECTOR)	_USE_ERASE == 1	
ff_convert ff_wtoupper	_USE_LFN >= 1	Unicode 支持函数。 在 option/cc*.c 中可用。
ff_cre_syncobj	_FS_REENTRANT == 1	O/S 相关函数。

ff_del_syncobj ff_req_grant ff_rel_grant		在 option/syscall.c 有相关实例。
ff_mem_alloc ff_mem_free	_USE_LFN == 3	

4.2 限制

FAT 子类型: FAT12、FAT16 和 FAT32

打开文件数量: 无限制, 与可用内存有关。

卷 (volume) 数量: 最多 10 个。

文件大小: 与 FAT 规范有关 (最大 4G-1 字节)。

卷大小: 与 FAT 规范有关 (在 512 字节/扇区上, 最大 2T 字节)

簇 (Cluster) 大小: 与 FAT 规范有关 (在 512 字节/扇区上, 最大 64K 字节)

扇区 (Sector) 大小: 与 FAT 规范有关 (最大 4K 字节)

4.3 内存使用(R0.09a)

	ARM7 32bit	ARM7 Thumb	CMS3 Thumb-2	AVR	H8/300H	PIC24	RL78	V850ES	SH-2A	RX600	IA-32
Compiler	GCC	GCC	GCC	GCC	CH38	C30	CC78K0R	CA850	SHC	RXC	VC6
_WORD_ACCESS	0	0	0	1	0	0	0	1	0	1	1
text (Full, R/W)	10375	7019	6561	13267	10525	11205	12812	7783	8715	5782	7615
text (Min, R/W)	6487	4727	4283	8521	6967	7398	8800	5019	5635	3784	5003
text (Full, R/O)	4551	3125	2895	6219	4895	5268	6123	3629	3843	2699	3527
text (Min, R/O)	3321	2457	2197	4527	3797	3999	4663	2773	2999	2064	2736
bss	D*4 + 2	D*4 + 2	D*4 + 2	D*2 + 2	D*4 + 2	D*2 + 2	D*2 + 2	D*4 + 2	D*4 + 2	D*4 + 2	D*4 + 2
Work area (_FS_TINY == 0)	V*560 + F*550	V*560 + F*550	V*560 + F*550	V*560 + F*544	V*560 + F*550	V*560 + F*544	V*560 + F*544	V*560 + F*544	V*560 + F*550	V*560 + F*550	V*560 + F*550
Work area (_FS_TINY == 1)	V*560 + F*36	V*560 + F*36	V*560 + F*36	V*560 + F*32	V*560 + F*36	V*560 + F*32	V*560 + F*32	V*560 + F*36	V*560 + F*36	V*560 + F*36	V*560 + F*36

这是在以下条件下的内存使用。内存大小以字节为单位, V 表示挂载的卷号, F 表示打开的文件号。所有示例在代码大小上都作了优化。

_FS_READONLY	0 (R/W), 1 (R/O)
_FS_MINIMIZE	0 (Full function), 3 (Minimized function)
_USE_STRFUNC	0 (Disable string functions)
_USE_MKFS	0 (Disable f_mkfs function)
_USE_FORWARD	0 (Disable f_forward function)
_USE_FASTSEEK	0 (Disable fast seek feature)
_CODE_PAGE	932 (Japanese Shift-JIS)
_USE_LFN	0 (Disable LFN)
_MAX_SS	512 (Fixed sector size)
_FS_RPATH	0 (Disable relative path)
_VOLUMES	D (Number of logical drives to be used)
_MULTI_PARTITION	0 (Single partition per drive)
_FS_REENTRANT	0 (Disable reentrancy)
_FS_SHARE	0 (Disable shareing control)

4.4 模块大小裁减

下表给出了通过配置选项能去掉哪些 API 函数来进行模块大小裁减。

Function	_FS_MINIMIZE				_FS_READONLY		_USE_STRFUNC		_FS_RPATH			_USE_MKFS		_USE_FORWARD		_MULTI_PARTITION	
	0	1	2	3	0	1	0	1/2	0	1	2	0	1	0	1	0/1	2
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_ftime	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_fdisk						x						x				x	
f_forward														x			
f_putc						x	x										
f_puts						x	x										
f_printf						x	x										
f_gets							x										

4.5 长文件名

FatFs 模块从版本 0.07 就已经开始支持长文件名 (LFN)。文件的两种文件名: SFN 和 LFN, 对除了 f_readdir 以外的其他函数都是透明的。要使能 LFN 特性, 将 _USE_LFN 设置为 1、2 或 3, 并且添加 Unicode 代码转换函数 ff_convert() 和 ff_wtoupper() 到工程中。LFN 特性需要额外的工作缓冲区。该缓冲区的大小可以通过与可用内存大小相对应的 _MAX_LFN 来配置。长文件名的大小可达到 255 个字符, _MAX_LFN 应该设置为 255 以满足全功能 LFN 操作。如果工作缓冲区的大小不足以存放给定的文件名, 文件函数将会失败, 并返回以 FR_INVALID_NAME。在使能具有重入特性的 LFN 特性, _USE_LFN 必须设置为 2 或 3。在这种情况下, 文件函数将在栈或堆中分配工作缓冲区。工作缓冲区使用 (_MAX_LFN+1)*2 字节。

LFN 在 ARM7 上配置

Code page	Program size
SBCS	+3.7K
932(Shift-JIS)	+62K
936(GBK)	+177K
949(Korean)	+139K
950(Big5)	+111K

4.6 Unicode API

FatFs 的 API 默认支持 ANSI/OEM 代码集, 但是 FatFs 也能将代码集切换到 Unicode。更多信息参考关于文件名的描述。

4.7 重入

对不同卷的文件操作总是重入的，并且能同时操作。对相同卷的操作则不是重入的，但能使用 `_FS_REENTRANT` 选项来配置为线程安全模式。在这种情况下，操作系统相关的同步对象控制函数：`ff_cre_syncobj`、`ff_del_syncobj`、`ff_req_grant` 和 `ff_rel_grant` 也必须添加到工程中。

在卷使用期间有任何其他任务调用一个文件函数，该文件函数将被挂起直到该任务放弃文件函数。如果等待时间超过了由 `_TIMEOUT` 定义时间，文件函数将以 `FR_TIMEOUT` 中止。某些 RTOS 不支持超时特性。

函数 `f_mount` 和 `f_mkfs` 函数有一个例外。这些函数对相同的卷是不可重入的。在使用这些函数时，所有其他任务必须关闭卷上的相应文件，防止访问卷。

要注意的是，本节描述了 FatFs 模块本身的重入，而底层磁盘 I/O 层必须可重入。

4.8 重复的文件访问

FatFs 模块默认是不支持重复文件访问的共享控制。在文件的打开方式为只读模式时，是允许的。以只写模式重复打开一个文件则总是被禁止的，并且已经打开的文件不能被重命名、删除，否则簇上的 FAT 结构将会崩溃。

文件共享控制也只有在 `_FS_SHARE` 设置为大于等于 1 的值时才可用。该值指定了同时管理的文件数量。以这种情况下，如果进行任何上面描述的违反共享规则的打开、重命名或删除操作，文件函数都将以 `FR_LOCKED` 失败。如果打开的文件数量超过 `_FS_SHARE` 值，则 `f_open` 函数将以 `FR_TOO_MANY_OPEN_FILES` 失败。

4.9 执行有效的文件访问

为了在小型嵌入式系统中很好地执行读/写文件，应用程序员应该考虑在 FatFs 模块完成了哪些处理。磁盘上的文件数据由 `f_read` 函数按以下步骤进行传输。

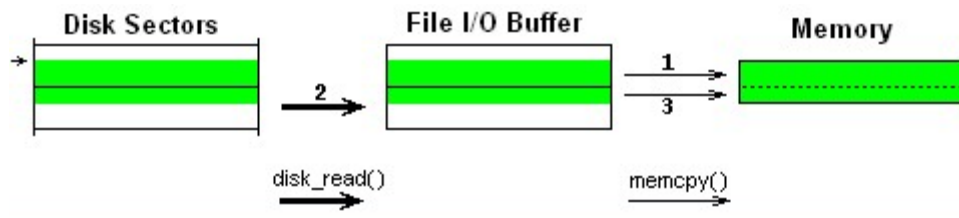


图 1 扇区非对齐 (miss-aligned) 读 (short)

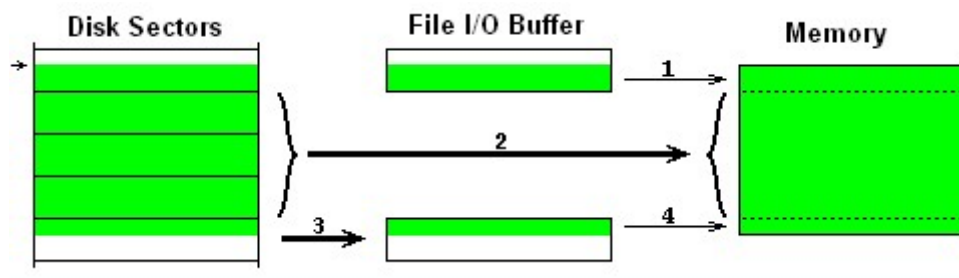


图 2 扇区非对齐 (miss-aligned) 读 (long)

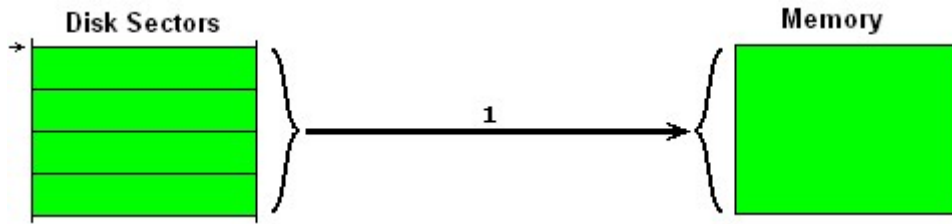


图 2 扇区对齐 (aligned) 读

文件 I/O 缓冲区 (file I/O buffer) 表示用于读/写扇区上部分数据的扇区缓冲区。扇区缓冲区既是每一个文件对象的文件私有扇区缓冲区，也是文件系统对象的共享扇区缓冲区。缓冲区配置选项 `_FS_TINY` 决定使用哪一个扇区缓冲区来传输文件数据。选择 `tiny buffer (1)` 时，每个文件对象将减少 512 个字节的数据内存。这样，FatFs 模块在文件系统对象上就只用一个扇区缓冲区来传输数据和访问 FAT 目录。Tiny buffer 配置的缺点是：扇区缓冲区上的 FAT 数据缓存会由于文件数据传输而丢失，而必须在每个簇边界时重新加载。但是从高性能和低内存消耗的大多数应用角度来看，这是最适合的。

图 1 给出了部分扇区数据通过文件 I/O 缓冲区进行传输的情况。图 2 给出了长数据传输的情况，在覆盖了一个或多个扇区的传输数据的中间，被直接传输到应用缓冲区。图 3 给出了整个传输数据对齐到扇区边界的情况。在这种情况下，文件 I/O 缓冲区就不再需要了。在直接传输时，`disk_read` 函数一次会读取最大程度的扇区，即使簇是相邻的，多扇区传输也绝不会跨越簇边界。

所以尽量采用扇区对齐读/写访问来避免缓冲数据传输，而提高读/写性能。除了下面的影响，缓存的 FAT 数据不会被 Tiny buffer 配置的文件数据传输清空，这样它能达到与非 Tiny buffer 配置使用少量内存需求同样的性能。

4.10 关于闪存媒体的考虑

为了尽量可能提高闪存媒体的写的性能，如 SDC 和 CFC，必须在其定位的考虑方面给予控制。

4.10.1 使用多扇区写

Figure 6. Comparison between Multiple/Single Sector Write

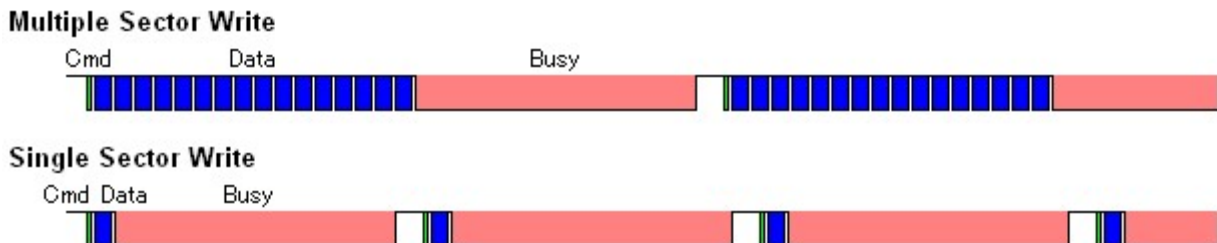


图 6 多/单扇区写的比较

闪存媒体的写能力在单扇区写时是非常差的，使得每次写操作的扇区数量成比例增加。其影响大多出现在快速总线时钟，其比率通常会超过 10。写操作的数量也影响着媒体的寿命。所以应用程序应该按尽可能大的块来写数据。理想的块大小为簇大小或 2 的幂字节，而字节偏移应该与块对齐。当然，在应用和媒体之间的所有层都必须支持多扇区写特性，然后大多数开源的磁盘驱动程序却做不到。不要将一个多扇区写请求分割成单扇区写，否则写能力会变差。注意，FatFs 模块和其示例磁盘驱动程序都支持多扇区读/写性能。

4.10.2 强制内存擦除

FAT 文件系统，用 `f_remove` 函数删除了一个文件后，由文件占用的数据簇会被标记为 “free”。但是包含文件数据的数据扇区却不再适应任何处理，所以留下来的文件数据占据着一部分闪存矩阵作为有活块。如果在删除文件时强制擦除文件数据，闪存的空闲块就会增加。在下次写时，还会跳过对数据块的内部块擦除操作。最终，写性能进一步提高。要启用该性能，将 `_USE_ERASE` 设置为 1。注意：这是对闪存媒体

内部处理的扩展。但这并不总是很有用，因为 `f_remove` 函数在删除一个大文件时会花费较长时间。

4.11 临界区（Critical Section）

如果一个对 FAT 文件系统的写操作由于任何意外失败而中断，如突然停电、错误的磁盘移除和不可恢复的磁盘错误等，FAT 结构将会崩溃。

```
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_mkdir(...);
```

```
f_rename(...);
```

```
f_unlink(...);
```

图 4 较长的临界区

```
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(...);
```

图 5 最小化的临界区

红色部分的出现的一个中断会引起一个交叉链接，结果是，被修改的对象可能丢失。当黄色部分发生一个中断，将出现下面列出的一个或多个可能性。

被改写的文件数据崩溃。

被挂起的文件返回初始状态。

新创建的文件没有了。

新创建的或写过的文件仍然没有内容。

磁盘使用的效率由于丢失簇而变差。

不是以写模式打开的文件不受上面各种情况的影响。为了最小化数据丢失的风险，临界区可以象图 5 所示的那样通过最小化文件以写模式打开的时间或适当使用 `f_sync` 函数来最小化。