Freescale MSD FATFS
API Reference Manual

Document Number:MSDFATFSAPIRM
Rev. 0
02/2011

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

freescale™
semiconductor

Document Number: MSDFATFSAPIRM
Rev. 0
02/2011

Revision History

To provide the most up-to-date information, the revision of Freescale documents on the World Wide Web are the most current. Your printed copy may be an earlier revision. To verify you have the latest information available, refer to:

http://www.freescale.com

The following revision history table summarizes changes contained in this document.

Revision number	Revision date	Description of changes
Rev. 0	02/2011	Initial release

Chapter 1
Before Beginning

Chapter 2
FATFS API Overview

Chapter 3
FATFS API

Chapter 1
Before Beginning

1.1 About this book

This book describes the MSD FATFSAPI functions. It describes in detail the API functions that can be used to develop FATFS applications. Table 1-1 shows the summary of chapters included in this book.

Table 1-1. MSDFATFS_APIRM Summary

Chapter Title	Description
Before Beginning	This chapter provides the prerequisites for reading this book.
FATFS API Overview	This chapter gives an overview of the API functions and how to use them for developing new applications.
FATFS API	This chapter discusses the FATFS API functions.
Data Structures	This chapter discusses the various data structures used in the FATFS API functions.
Path Name and Unicode API Information	This chapter provides information about path name format and Unicode file name

1.2 Reference Material

Use this book in conjunction with:

- *Freescale MSD FATFS User Guide* (document MSDFATFSUG, Rev. 0)
- MSD FATFS source code

For better understanding, refer to the following documents:

- USB Specification Revision 1.1
- USB Specification Revision 2.0
- USB Common Class Specification Revision 1.0
- FATFS Generic FAT File System Module document from the following website http://elm-chan.org/fsw/ff/00index_e.html

1.3 Acronyms and abbreviations

API	Application Programming Interface.
DBCS	Double-Byte Character Set
FAT	File Allocation Table
FATFS	File Allocation Table File System
MBR	Master Boot Record
MSD	Mass Storage Device.
OEM	Original Equipment Manufacturer
PC	Personal computer.
SCSI	Small Computer Systems Interface
USB	Universal Serial Bus.

1.4 Function listing format

This is the general format of an entry for a function, compiler intrinsic, or macro.

function_name()

A short description of what function **function_name()** does.

Synopsis

Provides a prototype for function **function_name()**.

```
<return_type> function_name(
        <type_1> parameter_1,
        <type_2> parameter_2,
        ...
        <type_n> parameter_n)
```

Parameters

parameter_1 [in] — Pointer to x
parameter_2 [out] — Handle for y
parameter_n [in/out] — Pointer to z

Parameter passing is categorized as follows:

- *in* — Means the function uses one or more values in the parameter you give it without storing any changes.
- *out* — Means the function saves one or more values in the parameter you give it. You can examine the saved values to find out useful information about your application.
- *in/out* — Means the function changes one or more values in the parameter you give it and saves the result. You can examine the saved values to find out useful information about your application.

Description

Describes the function **function_name()**. This section also describes any special characteristics or restrictions that might apply:

- function blocks or might block under certain conditions
- function must be started as a task
- function creates a task
- function has pre-conditions that might not be obvious
- function has restrictions or special behavior

Return Value

Specifies any value or values returned by function **function_name()**.

See Also

Lists other functions or data types related to function **function_name().**

Example

Provides an example (or a reference to an example) that illustrates the use of function **function_name()**.

Chapter 2 FATFS API Overview

2.1 Introduction

The FATFS API consists of the functions that can be used at the application level. These enable you to implement file system application.

2.2 API overview

This section describes the list of API functions and their use.

Table 2-1 summarizes the FATFS API functions.

Table 2-1. Summary of Host Layer API Functions

No.	API function	Description
1	f_mount	Register/Unregister a work area
2	f_open	Open/Create a file
3	f_close	Closes a file
4	f_read	Read data from file
5	f_write	Write data to file
6	f_lseek	Move read/write file pointer, Expand file size
7	f_truncate	Truncate file
8	f_sync	Flush cached data of a write file
9	f_opendir	Open a directory
10	f_readdir	Read a directory item
11	f_getfree	Get free cluster
12	f_stat	Get status of a file or a directory
13	f_mkdir	Create a directory
14	f_unlink	Remove a file or directory
15	f_chmod	Change attribute of a file or directory
16	f_utime	Change timestamp of a file or directory
17	f_rename	Rename/Move a file or directory
18	f_mkfs	Create a file system on the drive
19	f_forward	Forward file data to the stream directly

No.	API function	Description
20	f_chdir	Change current directory
21	f_chdrive	Change current drive
22	f_getcwd	Retrieve the current directory
23	f_gets	Read a data string from a file
24	f_putc	Write a character to file
25	f_puts	Write a data string to file
26	f_printf	Write a formatted string to file
27	f_eof	Check whether file pointer is the end of a file
28	f_error	Check whether file has error
29	f_tell	Return the current position of file pointer
30	f_size	Return the size of file

NOTE

- f_eof, f_error, f_tell, f_size are implemented as macros instead of functions.
- FATFS module is very flexible. It provides many module configuration options. User can select options that are best suitable for his device. For the further information, refer to **Section 4.2 Configuration Options** of MSDFATFS User Guide document.

2.3 Using API

Steps to use FATFS APIs similar to the second method to use the Host Layer API of Freescale USB Stack with PHDC Host API Reference Manual. The only thing needs change that is in Step 8. After the INTF event is notified in the callback function, issue FATFS API instead of class-specific API.

Chapter 3 FATFS API

3.1 FATFS API Function Listing

3.1.1 f_mount()

The function registers/unregisters a work area to the FATFS module.

Synopsis

```
FRESULT f_mount(
        BYTE Drive,
        FATFS* FileSystemObject)
```

Parameters

Driver [IN] — Interface Logical drive number (0-9) to register/unregister the work area

FileSystemObject [IN] — Points to the work area (file system object) to be registered

Description

The **f_mount()** function registers/unregisters a work area to the FATFS module. The work area must be given to the each volume with this function prior to use any other file function. To unregister a work area, specify a NULL to the *FileSystemObject*, and then the work area can be discarded.

This function always succeeds regardless of the drive status. No media access is occurred in this function and it only initializes the given work area and registers its address to the internal table. The volume mount process is performed on first file access after **f_mount**() function or media change.

Return Value

- **FR_OK:** The function succeeded
- **FR_INVALID_DRIV:** The drive number is invalid

See also

FATFS

3.1.2 f_open()

The function creates a file object to be used to access the file.

Synopsis

```
FRESULT f_open (
        FIL* FileObject,
        const TCHAR* FileName,
        BYTE ModeFlags)
```

Parameters

FileObject [OUT] — Pointer to the file object structure to be created
FileName [IN] — Pointer to a null-terminated string that specifies the file name to create or open
ModeFlags [IN] — Specifies the type of access and open method for the file. It is specified by a combination of the flags in Table 2-1.

Table 3-1. File Access Types

Value	Description
FA_READ	Specifies read access to the object. Data can be read from the file. For read - write access, combine with FA_WRITE.
FA_WRITE	Specifies write access to the object. Data can be written to the file. For read - write access, combine with FA_READ.
FA_OPEN_EXISTING	Open an existing file. The function fails if the file does not exist.
FA_OPEN_ALWAYS	Open the file if it exists. If not, a new file is created. To append data to the file, use f_lseek function after file open in this method.
FA_CREATE_NEW	Create a new file. The function fails with FR_EXIST if the file has already existed.
FA_CREATE_ALWAYS	Create a new file. If the file has already existed, it is truncated and overwritten.

Description

A file object is created when the function succeeded. The file object is used for subsequent read/write functions to refer to the file. When close an open file object, use f_close() function. If the modified file is not closed, the file data can be collapsed.
Before using any file function, a work area (file system object) must be given to the logical drive with f_mount() function. All file functions can work after this procedure.

Return Value

- **FR_OK:** The function succeeded and the file object is valid
- **FR_NO_FILE:** Could not find the file
- **FR_NO_PATH:** Could not find the path
- **FR_INVALID_NAME**: The file name is invalid
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_EXIST:** The file has already existed
- **FR_DENIED:** The required access was denied due to one of the following reasons:
  – Write mode open against a read-only file

  - File cannot be created due to a directory or read-only file is existing
  - File cannot be created due to the directory table is full
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM:** There is no valid FAT volume on the drive
- **FR_LOCKED:** The function was rejected due to file sharing policy

See also

f_read(), f_write(), f_close(), FIL, FATFS

3.1.3 f_close()

The function closes an opening file.

Synopsis

```
FRESULT f_close (
        FIL* FileObject)
```

Parameters

FileObject [IN] — Points to the open file objects structure to be closed.

Description

The **f_close()** function closes an open file object. If any data has been written to the file, the cached information of the file is written back to the disk. After the function succeeded, the file object is no longer valid and it can be discarded.

Return Value

- **FR_OK:** The file object has been closed successfully
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_INVALID_OBJECT:** The file object is invalid

See also

f_open(), f_read(), f_write(), FATFS.

3.1.4 f_read()

This function reads data from a file.

Synopsis

```
FRESULT f_read(
      FIL* FileObject,
      void* Buffer,
      UINT ByteToRead,
      UINT* ByteRead)
```

Parameters

FileObject [IN] — Pointer to the open file object

Buffer [OUT] — Pointer to the buffer to store read data

ByteToRead [IN] — Number of bytes to read in range of integer

ByteRead [OUT] — Pointer to the UINT variable to return number of bytes read. The value is always valid after the function call regardless of the result.

Description

The file pointer of the file object increases in number of bytes read. After the function succeeded, **ByteRead* should be checked to detect the end of file. In case of **ByteRead* < *ByteToRead*, it means the read pointer reached end of the file during read operation.

Return Value

- **FR_OK:** The function succeeded
- **FR_DENIED:** The function denied due to the file has been opened in non-read mode
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_INVALID_OBJECT:** The file object is invalid

See also

f_open(), f_gets(), f_write(), f_close(), FIL

3.1.5 f_write()

The function writes data to a file.

Synopsis

```
FRESULT f_write(
        FIL* FileObject,
        const void* Buffer,
        UINT ByteToWrite,
        UINT* ByteWritten)
```

Parameters

FileObject [IN] — Pointer to the open file object structure
Buffer [IN] — Pointer to the data to be written
ByteToWrite [IN] — Specifies number of bytes to write in range of UINT
ByteWritten [OUT] — Pointer to the UINT variable to return the number of bytes written. The value is always valid after the function call regardless of the result

Description

The write pointer in the file object is increased in number of bytes written. After the function succeeded, **ByteWritten* should be checked to detect the disk full. In case of **ByteWritten* < *ByteToWrite*, it means the volume got full during the writing operation. The function can take a time when the volume is full or close to full.

Return

- **FR_OK:** The function succeeded
- **FR_DENIED:** The function denied due to the file has been opened in non-write mode
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_INVALID_OBJECT:** The file object is invalid

See also

f_open(), f_read(), f_putc(), f_puts(), f_printf(), f_close(), FIL.

3.1.6 f_lseek()

The function moves the file read/write pointer of an open file object.

Synopsis

```
FRESULT f_lseek(
      FIL* FileObject,
      DWORD Offset)
```

Parameters

FileObject [IN] — Pointer to the open file object

Offest [IN] — Number of bytes from the start of file

Description

The **f_lseek()** function moves the file read/write pointer of an open file. The offset can be specified in only origin from top of the file. When an offset above the file size is specified in write mode, the file size is increased and the data in the expanded area is undefined. This is suitable to create a large file quickly, for fast writing operation. After the **f_lseek()** function succeeded, member fptr in the file object should be checked in order to make sure the read/write pointer has been moved correctly. In case of *fptr* is not the expected value, either of followings has been occurred.

- End of file. The specified Offset was clipped at the file size because the file has been opened in read-only mode.
- Disk full. There is insufficient free space on the volume to expand the file size.

When **_USE_FASTSEEK** is set to 1 and **cltbl** member in the file object is not NULL, the fast seek feature is enabled. This feature enables fast backward/long seek operations without FAT access by cluster link information stored on the user defined table. The cluster link information must be created prior to do the fast seek. The required size of the table is (number of fragments + 1) * 2 items. For example, when the file is fragmented in 5, 12 items will be required to store the cluster link information. The file size cannot be expanded when the fast seek feature is enabled.

Return Value

- **FR_OK:** The function succeeded
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_INVALID_OBJECT:** The file object is invalid
- **FR_NOT_ENOUGH_CORE:** Insufficient size of link map table for the file

See also

f_open(), f_truncate(), FIL.

3.1.7 f_truncate()

The function trancates the file size

Synopsis

```
FRESULT f_truncate(
        FIL* FileObject)
```

Parameters

FileObject [IN] — Pointer to the open file object

Description

The f_truncate() function truncates the file size to the current file read/write point. This function has no effect if the file read/write pointer is already pointing end of the file.

Return Value

- **FR_OK:** The function succeeded
- **FR_DENIED:** The function denied due to the file has been opened in non-write mode
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_INVALID_OBJECT:** The file object is invalid

See also

f_open(), f_lseek(), FIL.

3.1.8 f_sync()

The function flushes cached data of a written file.

Synopsis

```
FRESULT f_sync(
        FIL* FileObject)
```

Parameters

FileObject [IN] — Pointer to the open file objects to be flushed.

Description

The **f_sync()** function performs the same process as f_close() function but the file is left opened and can continue read/write/seek operations to the file. This is suitable for the applications that open files for a long time in write mode, such as data logger. Performing **f_sync()** of periodic or immediately after f_write() can minimize the risk of data loss due to a sudden blackout or an unintentional disk removal. However, **f_sync()** immediately before f_close() has no advantage because f_close() performs **f_sync()** in it. In other words, the difference between those functions is that the file object is invalidated or not

Return Value

- **FR_OK:** The function succeeded
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_INVALID_OBJECT:** The file object is invalid

See also

f_close()

3.1.9 f_opendir()

The function opens a directory.

Synopsis

```
FRESULT f_opendir(
        DIR* DirObject,
        const TCHAR* DirName)
```

Parameters

DirObject [OUT] — Pointer to the blank directory objects to be created

DirName [IN] — Pointer to the null-terminated string that specifies the directory name to be opened

Description

The f_opendir() function opens an existing directory and creates the directory object for subsequent calls. The directory object structure can be discarded at any time without any procedure.

Return Value

- **FR_OK:** The function succeeded and the directory object is created. It is used for subsequent calls to read the directory entries
- **FR_NO_PATH:** Could not find the path
- **FR_INVALID_NAME:** The path name is invalid
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM:** There is no valid FAT volume on the drive

See also

f_readdir(), DIR

3.1.10 f_readdir()

The function reads a directory item.

Synopsis

```
FRESULT f_readdir(
        DIR* DirObject,
        FILINFO* FileInfo)
```

Parameters

DirObject [IN] — Pointer to the open directory object

FileInfo [OUT] — Pointer to the file information structure to store the read item

Description

The function reads directory entries in sequence. All items in the directory can be read by calling this function repeatedly. When all directory entries have been read and no item to read, the function returns a null string into *f_name[]* member of *FiIenfo* without any error. When a null pointer is given to the *FileInfo*, the read index of the directory object will be rewinded.

If **LFN** feature is enabled, *lfname* and *lfsize* fields of *FileInfo* must be initialized with valid value prior to use the f_readdir function. The *lfname* is a pointer to the string buffer to return the long file name. The *lfsize* is the size of the string buffer in unit of character. If either the size of read buffer or **LFN** working buffer is insufficient for the **LFN** or the object has no **LFN**, a null string will be returned to the **LFN** read buffer. If the **LFN** contains any character that cannot be converted to OEM code, a null string will be returned but this is not the case on Unicode API configuration. When *lfname* is a NULL, nothing of the LFN is returned. When the object has no LFN, any small capitals can be contained in the SFN.

When relative path feature is enabled (**_FS_RPATH == 1**), "**.**" and "**..**" entries are not filtered out and it will appear in the read entries

Return Value

- **FR_OK:** The function succeeded
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_INVALID_OBJECT:** The directory object is invalid

See also

f_opendir(), f_stat(), FILINFO, DIR.

3.1.11 f_getfree()

This function gets number of free clusters of logical volume.

Synopsis

```
FRESULT f_getfree(
        const TCHAR* Path,
        DWORD* Clusters,
        FATFS** FileSystemObject)
```

Parameters

Path [IN] — Pointer to the null-terminated string that specifies the logical drive

Clusters [OUT] — Pointer to the DWORD variable to store number of free clusters

FileSystemObject [OUT] — Pointer to pointer that to store a pointer to the corresponding file system object

Description

The function gets number of free clusters on the drive. The member *FileSystemObject->csize* reflects number of sectors per cluster, so that the free space in unit of sector can be calculated with this. When *FSInfo* structure on FAT32 volume is not in sync, this function can return an incorrect free cluster count.

Return Value

- **FR_OK:** The function succeeded. The **Clusters* has number of free clusters and **FileSystemObject* points the file system object
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM:** There is no valid FAT partition on the drive

See also

FATFS

3.1.12 f_stat()

The function get information of a file or directory.

Synopsis

```
FRESULT f_stat(
        const TCHAR* FileName,
        FILINFO* FileInfo)
```

Parameters

FileName [IN] — Pointer to the null-terminated string that specifies the file or directory to get its information

FileInfo [OUT] — Pointer to the blank FILINFO structure to store the information

Description

The function gets the information of a file or directory. For details of the information, refer to the FILINFO structure and f_readdir() function. This function is not supported in minimization level of >= 1.

Return Value

- **FR_OK:** The function succeeded
- **FR_NO_FILE:** Could not find the file or directory
- **FR_NO_PATH:** Could not find the path
- **FR_INVALID_NAME:** The file name is invalid
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM;** There is no valid FAT volume on the drive

See also

f_opendir(), f_readdir(), FILINFO.

3.1.13 f_mkdir()

The function creates a new driectory.

Synopsis

```
FRESULT f_mkdir(
        const TCHAR* DirName)
```

Parameters

DirName [IN] — Pointer to the null-terminated string that specifies the directory name to create

Description

The function creates a new directory.

Return Value

- **FR_OK:** The function succeeded
- **FR_NO_PATH:** Could not find the path
- **FR_INVALID_NAME:** The path name is invalid
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_DENIED:** The directory cannot be created due to directory table or disk is full
- **FR_EXIST:** A file or directory that has same name is already existing
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM:** There is no valid FAT volume on the drive

3.1.14 f_unlink()

The function removes a file or directory.

Synopsis

```
FRESULT f_unlink(
        const TCHAR* FileName)
```

Parameters

FileName [IN] — Pointer to the null-terminated string that specifies an object to be removed

Description

The function removes a file or directory object. It can not remove opened objects.

Return Value

- **FR_OK:** The function succeeded
- **FR_NO_FILE:** Could not find the file or directory
- **FR_NO_PATH:** Could not find the path
- **FR_INVALID_NAME:** The path name is invalid
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_DENIED:** The function was denied due to either of following reasons:
  - — The object has read-only attribute
  - — Not empty directory
  - — Current directory
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_WRITE_PROTECTED:** The medium is write-protected
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM:** There is no valid FAT volume on the drive

3.1.15 f_chmod()

The function changes the attribute of file or directory.

Synopsis

```
FRESULT f_chmod(
        const TCHAR* FileName,
        BYTE Attribute,
        BYTE AttributeMask)
```

Parameters

FileName [IN] — Pointer to the null-terminated string that specifies a file or directory to be changed

Attribute[IN] — Attribute flags to be set in one or more combination of the following flags. The specified flags are set and others are cleared.

Table 3-2. File and Directory Attribute Flags

Attribute	Description
AM_RDO	Read Only
AM_ARC	Archive
AM_SYS	System
AM_HID	Hidden

AttributeMask [IN] — Attribute mask that specifies which attribute is changed. The specified attributes are set or cleared

Description

The f_chmod() function changes the attribute of a file or directory

Return Value

- **FR_OK:** The function succeeded
- **FR_NO_FILE:** Could not find the file
- **FR_NO_PATH:** Could not find the path
- **FR_INVALID_NAME:** The file name is invalid
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM:** There is no valid FAT volume on the drive

3.1.16 f_utime()

The function changes the timestamp of file and directory.

Synopsis

```
FRESULT f_utime(
        const TCHAR* FileName,
        const FILINFO* TimeDate)
```

Parameters

FileName [IN] — Pointer to the null-terminated string that specifies a file or directory to be changed

TimeDate [OUT] — Pointer to the file information structure that has a timestamp to be set in TimeDate -> fdate and TimeDate -> ftime. Do not care any other members

Description

The f_utime() function changes the timestamp of a file or directory.

Return Value

- **FR_OK:** The function succeeded
- **FR_NO_FILE:** Could not find the file
- **FR_NO_PATH:** Could not find the path
- **FR_INVALID_NAME:** The file name is invalid
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **R_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM:** There is no valid FAT volume on the drive

See also

f_stat(), FILINFO.

3.1.17 f_rename()

The function renames/moves a file or directory.

Synopsis

```
FRESULT f_rename(
const TCHAR* OldName,
const TCHAR* NewName)
```

Parameters

OldName [IN] — Pointer to a null-terminated string specifies the old object name to be renamed
NewName [IN] — Pointer to a null-terminated string specifies the new object name without drive number

Description

The function renames a object (file or directory). The logical drive number is determined by old name; new name must not contain a logical drive number. It can also move object to other directory, in this case, new name contain a logical drive number. ***Do not rename an opened object***.

Return Value

- **FR_OK:** The function succeeded
- **FR_NO_FILE:** Could not find the old name
- **FR_NO_PATH:** Could not find the path
- **FR_INVALID_NAME:** The file name is invalid
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_EXIST:** The new name is colliding with an existing name
- **FR_DENIED:** The new name could not be created due to any reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM:** There is no valid FAT volume on the drive

3.1.18 f_mkfs()

The function creates a file system on the drive.

Synopsis

```
FRESULT f_mkfs(
        BYTE   Drive,
        BYTE   PartitioningRule,
UINT   AllocSize)
```

Parameters

Drive [IN] — Logical drive number (0-9) to be formatted.

PartitioningRule [IN] — When 0 is given, a partition table is created into the master boot record and a primary DOS partition is created and then an FAT volume is created on the partition. This is called FDISK format, used for hard disk and memory cards. When 1 is given, the FAT volume starts from the first sector on the drive without partition table. This is called SFD format, used for floppy disk and most optical disk.

AllocSize [IN] — Force the allocation unit (cluster) size in unit of byte. The value must be power of 2 and between the sector size and 128 times sector size. When invalid value is specified, the cluster size is determined depends on the volume size

Description

The function creates an FAT volume on the drive. There are two partitioning rules, FDISK and SFD, for removable media. The FDISK format is recommended for the most case. ***This function currently does not support multiple partition***, so that existing partitions on the physical drive will be deleted and re-created a new partition occupies entire disk space.

The FAT sub-type, FAT12/FAT16/FAT32, is determined by number of clusters on the volume and nothing else, according to the FAT specification issued by Microsoft. Thus which FAT sub-type is selected, is depends on the volume size and the specified cluster size. The cluster size affects performance of the file system and large cluster increases the performance.

When the number of clusters gets near the FAT sub-type boundaries, the function can fail with FR_MKFS_ABORTED

Return Value

- **FR_OK:** The function succeeded
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_NOT_READY:** The drive cannot work due to any reason
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_MKFS_ABORTED;** The function aborted before start in format due to one of following reasons:
  - — The disk size is too small.
  - — Invalid parameter was given to any parameter.

— Not allowable cluster size for this drive. This can occur when number of clusters gets near the 0xFF7 and 0xFFF7.

3.1.19 f_forward()

The function forwards file data to the stream directly.

Synopsis

```
FRESULT f_forward (
    FIL* FileObject,
    UINT (*Func)(const BYTE*,UINT),
    UINT ByteToFwd,
    UINT* ByteFwd)
```

Parameters

FileObject [IN] — Pointer to the open file object
Func [IN] — Pointer to the user-defined data streaming function
ByteToFwd [IN] — Number of bytes to forward in range of integer
ByteFwd [OUT] — Pointer to the integer variable to return number of bytes forwarded

Description

The function reads the data from the file and forwards it to the outgoing stream without data buffer. This is suitable for small memory system because it does not require any data buffer at application module. The file pointer of the file object increases in number of bytes forwarded. In case of **ByteFwd* < *ByteToFwd* without error, it means the requested bytes could not be transferred due to end of file or stream goes busy during data transfer.

Return Value

- **FR_OK:** The function succeeded
- **FR_DENIED:** The function denied due to the file has been opened in non-read mode
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_INVALID_OBJECT:** The file object is invalid

See also

f_open(), f_gets(), f_write(), f_close(), FIL.

3.1.20 f_chdir()

The function changes current directory of a drive.

Synopsis

```
FRESULT f_chdir(
        const TCHAR* Path)
```

Parameters

Path [IN] — Pointer to the null-terminated string that specifies a directory to go

Description

The function changes the current directory of the logical drive. The current directory of a drive is initialized to the root directory when the drive is auto-mounted. Note that the current directory is retained in the each file system object so that it also affects other tasks that using the drive.

Return Value

- **FR_OK:** The function succeeded
- **FR_NO_PATH:** Could not find the path
- **FR_INVALID_NAME:** The path name is invalid
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM:** There is no valid FAT volume on the drive

See also

f_chdrive(), f_getcwd().

3.1.21 f_chdrive()

The function changes the current drive.

Synopsis

```
FRESULT f_chdrive(
        BYTE Drive)
```

Parameters

Drive [IN] — Specifies the logical drive number to be set as the current drive

Description

The function changes the current drive. The initial value of the current drive number is 0. Note that the current drive is retained in a static variable so that it also affects other tasks that using the file functions.

Return Value

- **FR_OK:** The function succeeded
- **FR_INVALID_DRIVE:** The drive number is invalid

See also

f_chdir(), f_getcwd().

3.1.22 f_getcwd()

The function retrieves the current directory.

Synopsis

```
FRESULT f_getcwd (
        TCHAR* Buffer,
        UINT BufferLen)
```

Parameters

Buffer [OUT] — Pointer to the buffer to receive the current directory string.

BufferLen [IN] — Size of the buffer in unit of TCHAR

Description

The function retrieves the current directory of the current drive in full path string including drive number.

Return Value

- **FR_OK:** The function succeeded
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM:** There is no valid FAT volume on the drive
- **FR_NOT_ENOUGH_CORE:** Insufficient size of Buffer

See also

f_chdrive(), f_chdir()

3.1.23 f_gets()

The function reads a string from the file.

Synopsis

```
TCHAR* f_gets(
       TCHAR* Str,
       int Size,
       FIL* )
```

Parameters

Str [OUT] — Pointer to read buffer to store the read string
Size [IN] — Size of the read buffer in unit of character
FileObject [IN] — Pointer to the open file object structure

Description

f_gets() is a wrapper function of f_read(). The read operation continues until a '\n' is stored, reached end of the file or the buffer is filled with Size - 1 (characters). The read string is terminated with a '\0'. When no character to read or any error occurred during read operation, **f_gets()** returns a null pointer. The end of file and error status can be examined with **f_eof()** and **f_error()** macros.
When the FATFS is configured to Unicode API (**_LFN_UNICODE == 1**), the file is read in UTF-8 encoding and stored it to the buffer in UCS-2. If not the case, the file will be read in one byte per character without any code conversion.

Return Value

When the function succeeded, Str will be returned

See also

f_open(), f_read(), f_putc(), f_puts(), f_printf(), f_close(), FIL.

3.1.24 f_putc()

The function puts a character to the file.

Synopsis

```
    int f_putc(
        TCHAR Chr,
        FIL* FileObject)
```

Parameters

Chr [IN] — A character to be put.

FileObject [IN] — Pointer to the open file objects structure

Description

The **f_putc()** is a wrapper function of f_write() .

Return Value

When the character was written successfully, the function returns 1. When the function failed due to disk full or any error, an EOF (-1) will be returned.

When the FATFS is configured to Unicode API (**_LFN_UNICODE = 1**), the UCS-2 character is written to the file in UTF-8 encoding. If not this case, the byte will be written directly.

See also

f_open(), f_puts(), f_printf(), f_gets(), f_close(), FIL.

3.1.25 f_puts()

The function writes a string to the file.

Synopsis

```
int f_puts(
        const TCHAR* Str,
        FIL* FileObject)
```

Parameters

Str [IN] — Pointer to the null terminated string to be written. The null character will not be written.

FileObject [IN] — Pointer to the open file objects structure

Description

The **f_puts()** is a wrapper function of f_putc().

Return Value

When the function succeeded, number of characters written that is not minus value is returned. When the function failed due to disk full or any error, an EOF (-1) will be returned.When the FATFS is configured to Unicode API (**_LFN_UNICODE = 1**), the UCS-2 string is written to the file in UTF-8 encoding. If not the case, the byte stream will be written directly.

See also

f_open(), f_putc(), f_printf(), f_gets(), f_close(), FIL.

3.1.26 f_printf()

The function writes formatted string to the file.

Synopsis

```
    int f_printf (
        FIL* FileObject,
        const TCHAR* Format,
        ...)
```

Parameters

FileObject [IN] — Pointers to the open file object structure

Format [IN] — Pointer to the null terminated format string

Description

The function is a wrapper function of f_putc() and f_puts(). The format tags follow this prototype: ***%[flags][width][.precision][length]*** specifier

The specifier is a sub-set of standard library shown as following:

Table 3-3. Specifier in format string

Specifier	Description	Example
c	Character	‘a’
s	String of characters	“sample”
d	Signed decimal integer	392
u	Unsigned decimal integer	7235
x	Unsigned hexadecimal integer	7fa
b	Binary number	111

The tag can also contain flags, width, .precision and modifiers sub-specifiers, which are optional and follow these specifications:

Table 3-4. Flags in format string

Flags	Description
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).

Table 3-5. Width in format string

Width	Description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.

Table 3-6. Precision in format string

.precision	Description
.number	For integer specifiers (d, u, x): precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For s: this is the maximum number of characters to be printed. By default, all characters are printed until the ending null character is encountered. For c type: it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed.

Table 3-7. Length in format string

length	Description
1	The argument is interpreted as a *long int* or *unsigned long in*t for integer specifiers (d, u, x), and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a *long double*.

Return Value

When the function succeeded, number of characters written is returned. When the function failed due to disk full or any error, an EOF (-1) will be returned.

See also

f_open(), f_putc(), f_puts(), f_gets(), f_close(), FIL.

Chapter 4
Data Structures

4.1 Data Structure Listings

4.1.1 FATFS

This structure keeps information of a drive's file system.

Synopsis

```
typedef struct {
      uint_8 fs_type;
      uint_8 drv;
      uint_8 csize;
      uint_8 n_fats;
      uint_8 wflag;
      uint_8 fsi_flag;
      uint_16 id;
      uint_16 n_rootdir;
#if _MAX_SS != 512
      uint_16 ssize;
#endif
#if !_FS_READONLY
      uint_32 last_clust;
      uint_32 free_clust;
      uint_32 fsi_sector;
#endif
#if _FS_RPATH
      uint_32 cdir;
#endif
      uint_32 n_fatent;
      uint_32 fsize;
      uint_32 fatbase;
      uint_32 dirbase;
      uint_32 database;
      uint_32 winsect;
      uint_8 win[_MAX_SS];
      } FATFS;
```

Fields

fs_type — FAT sub-type (0: Not mounted)

drive — Physical drive number

csize — Sectors per cluster (1, 2, 4... 128)

n_fats — Number of FAT copies (1, 2)

wflag — win[] dirty flag (1:must be written back)

fsi_flag — file system information dirty flag (1: must be written back)
id — File system mount ID
n_rootdir — Number of root directory entries (FAT12/16)
ssize — Bytes per sector (512, 1024, 2048, 4096)
last_clust — Last allocated cluster
free_clust — Number of free clusters
fsi_sector — fsinfo sector (FAT32)
cdir — Current directory start cluster (0:root)
n_fatent — Number of FAT entries (= number of clusters + 2)
fsize — Sectors per FAT
fatbase — FAT start sector
dirbase — Root directory start sector (FAT32:Cluster#)
database — Data start sector
winsect — Current sector appearing in the win[]
win[_MAX_SS] — Disk access window for Directory, FAT (and Data on tiny configuration)

4.1.2 FIL

This structure keeps information of data file

Synopsis

```
typedef struct {
      FATFS*   fs;
      uint_16 id;
      uint_8 flag;
      uint_8 pad1;
      uint_32 fptr;
      uint_32 fsize;
      uint_32 org_clust;
      uint_32 curr_clust;
      uint_32 dsect;
#if !_FS_READONLY
      uint_32 dir_sect;
      uint_8*   dir_ptr;
#endif
#if _USE_FASTSEEK
      uint_32*  cltbl;
#endif
#if _FS_SHARE
      uint_32 lockid;
#endif
#if !_FS_TINY
      uint_8 buf[_MAX_SS];
#endif
} FIL;
```

Fields

fs — Pointer to the owner file system object

id — Owner file system mount ID

flag — File status flags

pad1 — Pad

fptr — File read/write pointer (0 on file open)

fsize — File size

org_clust — File start cluster (0 when fsize==0)

curr_clust — Current cluster

dsect — Current data sector

dir_sect — Sector containing the directory entry

dir_ptr — Points to the directory entry in the window

cltbl — Pointer to the cluster link map table (null on file open)

lockid — File lock ID (index of file semaphore table)

buf[_MAX_SS] — File data read/write buffer

4.1.3 DIR

This structure keeps information of a directory.

Synopsis

```
typedef struct {
      FATFS*   fs;
      uint_16 id;
      uint_16 index;
      uint_32 sclust;
      uint_32 clust;
      uint_32 sect;
      uint_8*   dir;
      uint_8*   fn;
#if _USE_LFN
      uint_8*  lfn;
      uint_16 lfn_idx;
#endif
} DIR;
```

Fields

fs — Pointer to the owner file system object
id — Owner file system mount ID
index — Current read/write index number
sclust — Table start cluster (0:Root dir)
clust — Current cluster
sect — Current sector
dir — Pointer to the current SFN (sort file name) entry in the win[]
fn — Pointer to the SFN (in/out) {file[8], ext[3], status[1]}
lfn — Pointer to the LFN working buffer
lfn_idx — Last matched LFN index number (0xFFFF: No LFN)

4.1.4 FILINFO

This structure contains information of file and directory.

Synopsis

```
typedef struct {
      uint_32 fsize;
      DATE fdate;
      TIME ftime;
      uint_8 fattrib;
      TCHAR fname[13];
#if _USE_LFN
      TCHAR*lfname;
      uint_32 lfsize;
#endif
} FILINFO;
```

Fields

fsize — File size

fdate — Last modified date

ftime — Last modified time

fattrib — Attribute

fname[13] — Short file name (8.3 format)

lfname — Pointer to the LFN (long file name) buffer

lfsize — Size of LFN buffer in CHAR

4.1.5 DATE

This structure contains date information

Synopsis

```
typedef union{
      uint_16 Word;
      struct{
      uint_16 day:5;     /* Day (1..31) */
      uint_16 month:4;   /* Month (1..12) */
      uint_16 year:7;    /* Year origin from 1980 (0..127) */
      }Bits;
} DATE;
```

Fields

Word — 16-bits value contains date information

day — 5-bits value specifies last modified date

month — 4-bits value specifies last modified date

year — 7-bits value specifies last modified date

4.1.6 TIME

This structure contains time information.

Synopsis

```
typedef union{
      uint_16 Word;
      struct{
      uint_16 second:5;   /* Second / 2 (0..29) */
      uint_16 minute:6;   /* Minute (0..59) */
      uint_16 hour:5;     /* Hour (0..23) */
      }Bits;
}TIME;
```

Fields

Word — 16-bits value contains time information

second — 5-bits value specifies last modified time

minute — 6-bits value specifies last modified time

hour — 5-bits value specifies last modified time

Appendix A
Path Name and Unicode API Information

A.1 Path Name Format

The path name format on the FATFS module is similar to the filename specs of DOS/Windows as follows:

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

The FATFS module supports long file name (LFN) and 8.3 format file name (SFN). The LFN can be used when LFN feature is enabled (_USE_LFN > 0). The sub directories are separated with a \ or / in the same way as DOS/Windows API. Only a difference is that the logical drive is specified in a numeral with a colon. When the drive number is omitted, it is assumed as default drive (0 or current drive).

Control characters (\0 to \x1F) are recognized as end of the path name. Leading/embedded spaces in the path name are valid as a part of the name on LFN configuration but they are recognized as end of the path name on non-LFN configuration. Trailing spaces and dots are ignored.

In default configuration (_FS_RPATH == 0), it does not have a concept of current directory like OS oriented file system. All objects on the volume are always specified in full path name that follows from the root directory. Dot directory names are not allowed. Heading separator is ignored and it can be exist or omitted. The default drive number is fixed to 0.

When relative path feature is enabled (_FS_RPATH == 1), specified path is followed from the root directory if a heading separator is exist. If not, it is followed from the current directory set with f_chdir function. Dot names are also allowed for the path name. The default drive is the current drive set with f_chdrive function. The following table lists set of invalid path names:

Table 4-1. Invalid path names

Path Name	RPATH = 0	RPATH = 1
file.txt	A file in the root directory of the drive 0	A file in the current directory of the current drive
/file.txt	A file in the root directory of the drive 0	A file in the current directory of the current drive
	The root directory of the drive 0	The current directory of the current drive
/	The root directory of the drive 0	The root directory of the current drive
2:	The root directory of the drive 2	The current directory of the drive 2
2:/	The root directory of the drive 2	The root directory of the drive 2
2:file.txt	A file in the root directory of the drive 2	A file in the current directory of the drive 2
../file/txt	Invalid name	A file in the parent directory
.	Invalid name	This directory

Table 4-1. Invalid path names

Path Name	RPATH = 0	RPATH = 1
..	Invalid name	Parent directory of the current directory
dir1/..	Invalid name	The current directory
../	Invalid name	The root directory (sticks the top level)

A.2 Correspondence between logical and physical drive

The FATFS module has work areas that called file system object for each volume (logical drive). In default, the logical drive is bound to the physical drive that has same drive number, and the first partition is mounted. When _MULTI_PARTITION == 1 is specified in configuration option, each individual logical drive can be bound to any physical drive/partition. In this case, a drive number resolution table must be defined as follows:

Example: Logical drive 0-2 are assigned to three primary partitions on the physical drive 0 (fixed disk)

Logical drive 3 is assigned to physical drive 1 (removable disk)

```
const PARTITION Drives[] = {
{0, 0},     /* Logical drive 0 ==> Physical drive 0, 1st partition */
{0, 1},     /* Logical drive 1 ==> Physical drive 0, 2nd partition */
{0, 2},     /* Logical drive 2 ==> Physical drive 0, 3rd partition */
{1, 0}      /* Logical drive 3 ==> Physical drive 1 */
};
```

There are some considerations when use _MULTI_PARTITION configuration.

- Only primary partition (0-3) can be mounted.
- When the physical drive has no partition table (SFD format), the partition number is ignored

A.3 Unicode API

FATFS supports ANSI/OEM code set on the API in default but FATFS can also switch the code set to Unicode.

The path names are input/output in either ANSI/OEM code (SBCS/DBCS) or Unicode depends on the configuration options. The type of arguments that specifies the file names are defined as TCHAR which is an alias of char in default. The code set of the file name string is the ANSI/OEM code set specified by _CODE_PAGE. When _LFN_UNICODE is set to 1 under LFN configuration, the type of the TCHAR is switched to unsigned short (UCS-2 character) to support Unicode. In this case, the LFN feature is fully supported and the Unicode specific characters, can also be used for the path name. It also affects data types and encoding of the string I/O functions. To define literal strings, _T(s) and _TEXT(s) macro are available to select either ANSI/OEM or Unicode automatically. The code shown below is an example to define the literal strings.

```
f_open(fp, "filename.txt", FA_READ);      /* ANSI/OEM only */
f_open(fp, L"filename.txt", FA_READ);     /* Unicode only */
f_open(fp, _T("filename.txt"), FA_READ);  /* Changed automatically */
```