



FAT and SafeFAT File System User Guide

Version 5.80

For use with FAT and SafeFAT versions 9.8 and above

Table of Contents

1. System Overview	7
1.1. Introduction	8
1.2. Feature Check	10
1.3. Packages and Documents	11
1.4. Change History	12
2. File Encryption	14
3. About SafeFAT	15
3.1. File Synchronization	15
3.2. Operation and FAT Compatibility	15
3.3. Lower Layer Requirements	15
3.4. Improving Performance	16
4. Source File List	17
5. Configuration Options	19
5.1. config_fat.h	19
General Options	19
File Names	21
Other File Options	22
Volume Definitions	24
Sector Size	25
Caching	26
Encryption	27
5.2. config_fat.c	27
6. Other File System Information	28
6.1. System Requirements	28
6.2. Stack Requirements	28
6.3. Real-Time Requirements	28
6.4. Drives, Partitions and Volumes	29
6.5. Drive Format	29
6.6. Cache Setup and Options	29
6.7. Use of Wildcards	31
7. Application Programming Interface	32
7.1. Module Management	32
fs_init	33

fs_start	34
fs_stop	35
fs_delete	36
7.2. File System API	37
General Management	38
f_enterFS	39
f_releaseFS	40
f_getlasterror	41
Volume Management	42
f_initvolume	44
f_initvolume_nonsafe	46
f_delvolume	48
f_checkvolume	49
f_repair	50
f_setvolname	51
f_getvolname	53
f_get_oem	54
f_get_volume_count	55
f_get_volume_list	56
f_initvolumepartition	57
f_initvolumepartition_nonsafe	59
f_format	61
f_createpartition	63
f_createpartition_align	65
f_getpartition	67
f_createdriver	69
f_releasedriver	71
f_chdrive	73
f_getdrive	74
f_getfreespace	75
f_getlabel	77
f_setlabel	78
f_get_cluster_size	79
f_volume_clean	80
Directory Management	83
f_mkdir	84
f_chdir	85

f_rmdir	86
f_getcwd	87
f_getdcwd	88
File Access	89
f_open	90
f_open_enc	92
f_open_nonsafe	94
f_close	96
f_abortclose	96
f_flush	99
f_read	100
f_write	102
f_getc	104
f_putc	105
f_eof	106
f_seteof	107
f_tell	108
f_seek	109
f_rewind	111
f_truncate	112
f_ftruncate	113
File Management	114
f_delete	115
f_deletecontent	116
f_findfirst	117
f_findnext	119
f_move	121
f_rename	122
f_getattr	123
f_setattr	125
f_gettimedate	126
f_settimedate	128
f_stat	130
f_fstat	131
f_filelength	133
f_dir_walk	134
f_disk_usage	136

7.3. File System Unicode API	138
Unicode Directory Management	139
f_wmkdir	140
f_wchdir	141
f_wrmdir	142
f_wgetcwd	143
f_wgetdcwd	144
Unicode File Access	145
f_wopen	146
f_wopen_enc	148
f_wopen_nonsafe	150
f_wtruncate	152
Unicode File Management	153
f_wdelete	154
f_wdeletecontent	155
f_wfindfirst	156
f_wfindnext	157
f_wmove	158
f_wrename	159
f_wgetattr	160
f_wsetattr	161
f_wgettimedate	162
f_wsettimedate	164
f_wstat	166
f_wfilelength	167
f_wdir_walk	168
Unicode Translation	171
f_set_ascii_to_unicode	172
f_set_unicode_to_ascii	173
7.4. Error Codes	174
7.5. Types and Definitions	176
W_CHAR: Character and Wide Character Definition	176
F_FILE: File Handle	176
F_FIND	176
F_WFIND	177
F_SPACE	177
F_PARTITION	178

F_STAT Structure	178
ST_FILE_CHANGED	178
Change Object Flags	179
Change Object Actions	179
Directory Entry Attributes	180
Format Type	180
System Indicator	180
cdate Definitions	181
ctime Definitions	181
t_dir_walk_cb_fn	182
t_wdir_walk_cb_fn	183
t_volume_clean_stage_type	184
t_volume_clean_cb_fn	184
8. Integration	185
8.1. OS Abstraction Layer	185
Configuring the OAL	186
Multiple Tasks, Mutexes and Reentrancy	186
8.2. PSP Porting	187
Get Time and Date	188
Random Number	188
9. Version	189

1. System Overview

This chapter contains the fundamental information for this module.

The component sections are as follows:

- [Introduction](#) - describes the main elements of the module.
- [Feature Check](#) - summarizes the main features of the module as bullet points.
- [Packages and Documents](#) - the *Packages* section lists the packages that you need in order to use this module. The *Documents* section lists the relevant user guides.
- [Change History](#) - lists the earlier versions of this manual, giving the software version that each manual describes.

All rights reserved. This document and the associated software are the sole property of HCC Embedded. Reproduction or duplication by any means of any portion of this document without the prior written consent of HCC Embedded is expressly forbidden.

HCC Embedded reserves the right to make changes to this document and to the related software at any time and without notice. The information in this document has been carefully checked for its accuracy; however, HCC Embedded makes no warranty relating to the correctness of this document.

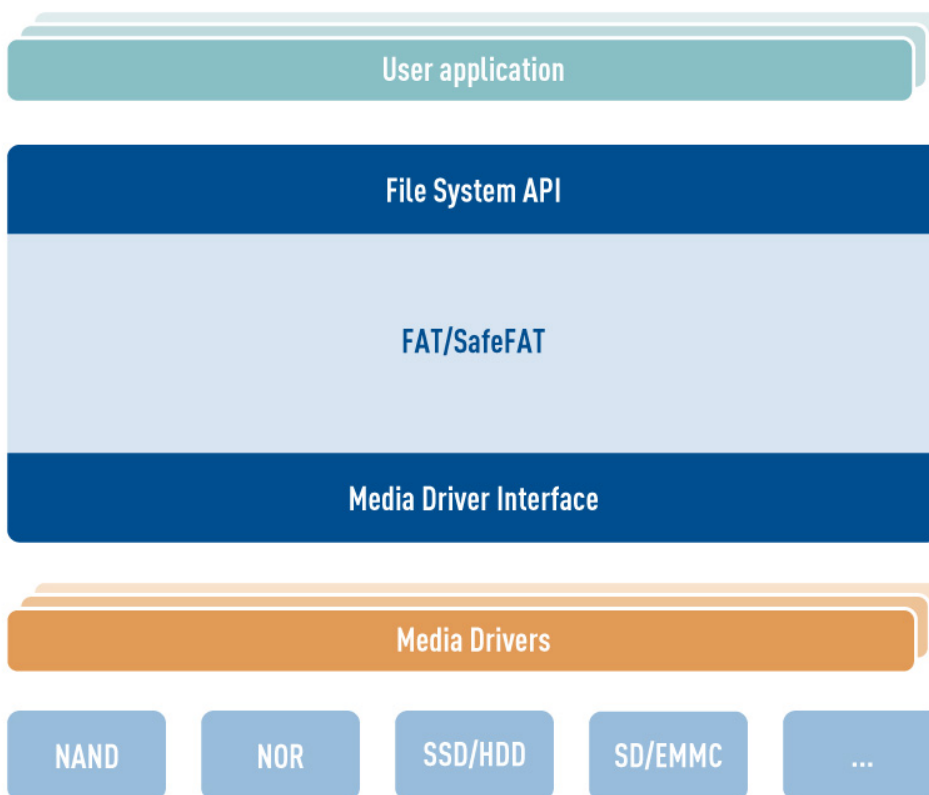
1.1. Introduction

This guide is for those who wish to implement a fail-safe file system. It covers HCC Embedded's FAT and SafeFAT file system products.

SafeFAT is a comprehensive FAT file system for FAT12, FAT16, and FAT32, designed to be truly fail-safe. SafeFAT protects against unexpected reset or power loss.

FAT and SafeFAT can access any combination of storage device types that conform to the [HCC Media Driver Interface Specification](#). HCC provides proven drives over many platforms for RAM, SD card, Compact Flash card, MultiMediaCard, HDD, Flash Translation Layer (FTL) and others.

This diagram summarizes the system architecture:



User applications use the standard file Application Programming Interface (API) to issue file system commands to the FAT file system. The FAT file system makes use of media drivers to access one or more storage media to execute the requested storage operation.

Note: Because SafeFAT is closely related to HCC's FAT file system, this manual uses the term SafeFAT only where it refers to functionality that is not available in the FAT system, but is specifically provided by SafeFAT.

Note:

- HCC offers hardware and firmware development consultancy to assist developers with the implementation of various types of file system.
- Although every attempt has been made to simplify the system's use, developers must have a good understanding of the requirements of the systems they are designing in order to obtain the maximum practical benefits.

1.2. Feature Check

The main features of the system are the following:

- Conforms to the HCC Advanced Embedded Framework.
- Designed for integration with both RTOS and non-RTOS based systems.
- Code size is ~24 KB (FAT) ~31 KB (SafeFAT).
- RAM usage is >3 KB (FAT) and >6 KB (SafeFAT).
- Provides fail safety (SafeFAT only).
- ANSI 'C'.
- Supports long filenames.
- Supports Unicode 16.
- Supports multiple open files.
- Supports multiple users of open files.
- Supports multiple volumes.
- Supports multi-sector read/write.
- File Multiple Write (FMW) package allows opening of the same file for writing multiple times.
- Supports variable sector sizes.
- Supports partition handling.
- Handles media errors.
- Test suite is provided.
- Cache options give improved performance.
- Supports zero copy.
- Reentrant.
- Common API (CAPI) support.
- Secure delete option (but this needs special driver support).
- AES 128 file encryption option.
- FAT-compatible.
- Standard drivers are available for SD, SDHC, SDXC, MMC, SafeFTL, USB-MST, HDD and RAM.

1.3. Packages and Documents

Packages

This table lists the packages that need to be used with this module, and also optional modules that may interact with this module, depending on your system's design:

Package	Description
<code>hcc_base_doc</code>	This contains the two guides that will help you get started.
<code>fs_fat</code>	The FAT file system package. This is the base system on which SafeFAT is built.
<code>fs_fat_safe</code>	The SafeFAT package that contains the extensions to FAT.
<code>fs_fat_fmws</code>	The FAT File Multiple Write (FMW) package that contains the extensions to FAT.
<code>fs_fat_test</code>	The FAT File System Test Suite.
<code>fs_fat_fmws_test</code>	The FAT FMW File System Test Suite.
<code>psp_template_base</code>	The Platform Support Package (PSP) base package.
<code>oal_base</code>	The OS Abstraction Layer (OAL) base package.
<code>media_drv_base</code>	The Media Driver base package that provides the base for all media drivers that attach to the file system.
<code>fs_capi</code>	The File System Common API (required if used with the SafeFLASH file system).

Additional packages

Other packages may also be provided, for example media drivers and PSP extensions specific to a target.

Documents

For an overview of HCC file systems and guidance on choosing a file system, see [Product Information](#) on the main HCC website.

Readers should note the points in the [HCC Documentation Guidelines](#) on the HCC documentation website.

HCC Firmware Quick Start Guide

This document describes how to install packages provided by HCC in the target development environment. Also follow the *Quick Start Guide* when HCC provides package updates.

HCC Source Tree Guide

This document describes the HCC source tree. It gives an overview of the system to make clear the logic behind its organization.

HCC FAT and SafeFAT File System User Guide

This is this document.

1.4. Change History

This section describes past changes to this manual.

- To download this manual or a PDF describing an [earlier software version, see File System PDFs](#).
- For the history of changes made to the package code itself, see [History: fs_fat](#).

The current version of this manual is 5.80. The full list of versions is as follows:

Manual version	Date	Software version	Reason for change
5.80	2020-06-12	9.8	Added f_wopen_enc() .
5.70	2020-06-09	9.7	Added refs to fs_fat_checkdisk() . Other small changes including inserting "u" in Unicode examples.
5.60	2020-05-14	9.6	Added FAT_FMW_ENABLE configuration option. F_MAXFILES default value increased to 4.
5.50	2020-04-20	9.5	F_MAXFILES default value reduced to 3.
5.40	2020-04-20	9.3	Added F_SAFE_LOG_CLUSTER_SHIFT_BYTES configuration option.
5.30	2020-03-05	9.1	Added volume cleaning: <ul style="list-style-type: none"> • configuration options: F_VOLUME_CLEAN_SUPPORT, F_VOLUME_CLEAN_FILL_CHAR and F_SZ_VOLUME_CLEAN_BUFn. • f_volume_clean() function, t_volume_clean_cb_fn() callback and <i>t_volume_clean_stage_type</i> typedef. f_disk_usage() now uses two 32 bit variables instead of one 64 bit variable.
5.20	2019-11-08	8.51	Added BAD_BLOCK_HANDLING configuration option.
5.10	2019-08-28	8.46	Added F_SAFE_AUTO_REPAIR configuration option and f_repair() function.
5.00	2019-08-22	8.45	Added information to f_setvolname() , f_setvolname() , and f_setlabel() .
4.90	2019-06-27	8.45	Changed default of configuration option F_MAX_SECTOR_SIZE. Simplified Media Drivers section in <i>Source Files</i> . Improved f_createpartition() text. Note: the following should have been added to this manual at s/w version 8.35. Added configuration options F_DIR_WALK_SUPPORT, F_DIR_WALK_MAX_DIR_DEPTH, and F_DISK_USAGE_SUPPORT. Added f_get_cluster_size() , f_disk_usage() , f_dir_walk() , f_wdir_walk() and callbacks used by the last two.
4.80	2019-01-10	8.41	Added directory to f_stat() and fwstat() pages. Also changed the note on those pages.
4.70	2018-07-23	8.40	Changed text on F_FILE_CHANGED_EVENT and FN_MAXPATHNAME configuration options.

Manual version	Date	Software version	Reason for change
4.60	2018-07-13	8.37	Changed text on date and time formats in functions f_gettimedate() and f_settimedate() ; added two pages describing their definitions.
4.50	2017-09-19	8.24	Changed encryption options section and f_open_enc() text.
4.40	2017-08-10	8.23	Added <i>File Encryption</i> section. Added note to <i>Encryption</i> . Fixed f_open_enc() example and moved this call to after f_open() .
4.30	2017-07-21	8.23	Packages list extended. <i>Unsigned long</i> replaced by <i>uint32_t</i> in structures: F_FIND, F_WFIND, F_PARTITION, F_SPACE, F_STAT, ST_FILE_CHANGED.
4.20	2017-06-23	8.21	New <i>Change History</i> format.
4.10	2016-04-21	8.15	Added function group descriptions to API.
4.00	2016-01-06	8.12	Changes to <i>Source Files</i> .
3.90	2015-05-11	8.07	Various small changes.
3.80	2015-04-28	8.06	Various small changes.
3.70	2015-03-24	8.06	Added <i>Change History</i> section.
3.60	2014-10-24	8.06	Reorganized <i>System Overview</i> .
3.50	2014-05-15	8.01	First online version.

2. File Encryption

You can optionally enable the file system to encrypt and decrypt files. To do this, you must set the [FAT_ENCRYPTION](#) option and also include [HCC's Embedded Encryption Manager](#) (EEM) in your project. The system uses AES encryption with 128 bit keys.

Note: A file must always be used consistently - if you start using it encrypted by using `f_open_enc()`, it must always be used encrypted. If you start using it unencrypted by using `f_open()`, then you must never encrypt it.

To access a file with encryption, use the [f_open_enc\(\)](#) function. When you call this, you pass to the function your secret 128 bit (16 byte) key for this data and also a 16 byte initialization vector that should be unique for each file. Files opened with encryption enabled can be accessed in r, a, and a+ modes.

Disk Level Encryption

HCC can also support disk level encryption for your file systems. This is done at driver level. Please contact HCC for further information.

3. About SafeFAT

The standard FAT file system was not designed to operate in systems that could be reset unexpectedly. This resulted in an extremely efficient file system, but one that could be damaged if a system using it did not complete its operations. There are numerous situations in a FAT file system where, for the system to be consistent, two or more areas of the disk must be updated atomically. This is clearly not possible.

To address these issues, HCC Embedded has developed SafeFAT. For critical operations, SafeFAT makes a log of operations. To summarize:

1. It records what it is going to do.
2. It does it.
3. It cancels the original record of what it was going to do.

Therefore, with some careful implementation, it is always possible for the system at boot time to check whether a critical operation was in progress. If so, the system either completes the operation that was in progress, or rewinds the system to the previous consistent state.

An additional factor is that, in the standard FAT file system, if a file pointer is moved to the middle of a file the existing data is directly overwritten. In the SafeFAT system this cannot be allowed to happen. Therefore, new sectors must be allocated (and chains modified) to ensure that the original file state can be restored if the new operation is not completed.

3.1. File Synchronization

All files are maintained in a consistent state: a file is switched to a new consistent state atomically when you decide it should be switched. When a file is modified, these modifications are not directly added to the file; the system lets you choose when a file is in its new state, and the new state is forced when a flush or close is called. Until a flush or close is called, the file remains in its previously saved state.

3.2. Operation and FAT Compatibility

SafeFAT uses a standard FAT file system format. It creates a special directory ("\$\$SAFE\$\$") to use for journaling purposes. This does not affect the normal operation of the drive. The system operates as follows:

- If the system is shut down normally, the contents of the drive are fully FAT-compatible.
- If there is an unexpected reset in a system running SafeFAT and the system is then restarted, any incomplete operation is correctly fixed using the information stored in the \$\$SAFE\$\$ directory.
- If there is an unexpected reset and the volume is inserted into a system running a standard FAT file system (for example on a PC), the FAT system will not be able to understand the special information provided by HCC in the \$\$SAFE\$\$ directory. The volume will be in an identical state, as far as the FAT system is concerned, to that produced if it had been running a standard FAT system that was unexpectedly interrupted. That is, there may be errors on the disk, such as lost clusters. These errors can be fixed by inserting the volume into a system running SafeFAT before a standard FAT file system modifies any content on the disk.

3.3. Lower Layer Requirements

In order for a file system that claims fail-safety to be able to ensure correct operation, it has to specify the minimum requirements that must be satisfied by the media interface below it. For example, suppose that a low level HDD driver has a large cache that can be written to the disk. If, when an unexpected reset occurs there's no guarantee that all data are written, it is unlikely that any system will be able to ensure a consistent state of that disk.

For SafeFAT the requirements are:

- Any sectors written to the disk are committed to the disk before the next write is started.
- Any sector written to the disk is updated atomically. That is, in all cases either the original contents of the sector are present or the new data are present; there are no intermediate states.
- If an unexpected reset condition is reached, the file system is restarted. No attempt is made to continue to use the system after a serious condition is detected.

If these conditions are not met, the system cannot be guaranteed fail-safe. However, even if they are not met the system is much safer than a standard unprotected FAT file system.

Guaranteeing that these conditions are fulfilled is not always easy. The vast majority of flash card vendors do not provide detailed information about how their cards work. This makes it very difficult to define how a system will behave when used with media whose behavior is undefined.

HCC Embedded works closely with a number of card manufacturers to provide solutions in which target devices have been designed to meet the above criteria. HCC has a test system in place to verify whether flash cards meet the required standards. Although HCC's tests cannot prove conclusively that a card is reliable, as defined above, they give a very good indication of the level of reliability that can be expected.

3.4. Improving Performance

The functions **f_open_nonsafe()** and **f_wopen_nonsafe()** allow the use of faster, non-safe, file access where appropriate. These may be used to improve performance when a file whose contents are less sensitive is being written.

Note: If you use these and the system is reset unexpectedly, the open file may be left in an uncertain state. Typically, the length may not be consistent with the amount of data written.

4. Source File List

This section lists and describes all the source code files included in the system. These files follow HCC Embedded's standard source tree system, described in the [HCC Source Tree Guide](#). All references to file pathnames refer to locations within this standard source tree, not within the package you initially receive.

Note: Do not modify any files except the configuration files.

API Header Files

The following files in the directory **src/api** must be included by any application using the system. They include all that is required to access the system. The use of these API functions is defined in [Application Programming Interface](#).

File	Description
api_fat.h	API for the module.
api_fs_err.h	Error code definitions.

Configuration Files

The following files in the directory **src/config** contain all the configurable parameters of the system. Configure these as required. For details of these options, see [Configuration Options](#).

File	Description
config_fat.c	If dynamic memory allocation is enabled on the system, this file defines dynamic memory allocation for each volume.
config_fat.h	Configuration options. (The fs_fat_safe package also has a copy of this file.)

Media Drivers

FAT works with any media driver that conforms to the [HCC Media Driver Interface Specification](#) and is included under the media driver directory.

Version File

The file **src/fat/version/ver_fat.h** contains the version number of this module. This version number is checked by all modules that use this module to ensure system consistency over upgrades.

FAT File System

These files are in the directory **src/fat/common**. **These files should only be modified by HCC.**

File	Description
fat.c	FAT short filename functions.
fat.h	FAT file system header.
fat_common.c	Common functions.
fat_common.h	Common functions header.
fat_init.h	Initialization header file.
fat_lfn.c	Alternative to fat.c for use with long filenames .
fat_m.c	FAT file system reentrancy wrapper.
fat_m.h	FAT file header reentrancy header.
fat_shjis.c	Code for Shift JIS character encoding in file and directory names.
fat_shjis.h	Header file for Shift JIS character encoding.

SafeFAT File System

The following files in the **fs_fat_safe** package's directory **src/fat/safe** are extensions to the FAT file system to provide the SafeFAT functionality. **These files should only be modified by HCC.**

File	Description
safefat.c	SafeFAT-specific source code.
safefat.h	SafeFAT-specific header file.

5. Configuration Options

Use the following files to configure your system:

- **src/config/config_fat.h** – set all the configuration options in this file.
- **src/config/config_fat.c** – set the memory allocation settings for each volume here.

5.1. config_fat.h

This section lists the available configuration options and their default values.

General Options

FN_CAPI_USED

The default is 0. Enable this only if other HCC file systems are being used in the target system and a single common API is required to cover all accesses to the volumes of both systems. Common API (CAPI) provides a wrapper for these file systems and therefore covers the majority of file system configuration options. If CAPI is used, refer to the [HCC File System Common API User Guide](#) as the starting point for system configuration.

OEM_NAME

This field is used only when a format is performed. It can be defined to identify the drive if required. The default is "MSDOS5.0" Change this with care since some operating systems may not accept all values.

HCC_UNICODE

This option enables support for Unicode 16 formatted characters. (Unicode 7/8 formats are supported as standard.) The default is 0. To enable this option, you must change the value to 1.

This forces any build to include the Unicode 16 API, making the Unicode 16 API calls documented in [File System Unicode API](#) available. This build also forces long filename support (see the [F_LONGFILENAME](#) parameter), which is necessary for Unicode 16 support.

Use of Unicode 16:

- Implies that the host system has wchar (“wide character”) support or an equivalent definition.
- Creates additional resource requirements because all string and path accesses effectively use twice the space. Therefore, use of this option is recommended only if Unicode 16 is genuinely required.

Note: To allow the file system to generate consistent short file names from the Unicode file name, you must provide conversion tables in the code. For details, see the [Unicode Translation](#) section.

F_FILE_CHANGED_EVENT

This notifies any change in the file or directory structure of the file system. The default is 0.

Enabling this is useful when the system is used in conjunction with other file system interfaces such as MTP or NFS, where the other system needs notifying of any changes to the directory or file structures in

the system. When this is enabled, a path can store the drive letter and a colon (for example, "A:") plus the string terminator.

F_SUPPORT_TI64K

The default is 0. Enabling this option ensures that read and write operations do not cross over 64K boundaries. The system automatically breaks the results of these operations into units which do not cross these boundaries.

This option is provided because certain devices, in particular TI C2000 and C5000 series DSPs, do not handle pointer increments over 64K boundaries.

USE_TASK_SEPARATED_CWD

If this is set to 1, every task has its own current working directory (cwd). This is the default setting and it is consistent with older versions of the system.

If this is set to 0, there is one cwd per volume so if any task changes it, it is changed for all tasks accessing that volume.

USE_MALLOC

Enable this if you want cache and other data structures to be allocated from the heap. The default is 0, meaning these structures are statically allocated.

- Set `USE_MALLOC` to 0 when building FAT from sources and linking to the application. All variables are allocated statically during compile time.
- Set `USE_MALLOC` to 1 to allow FAT to use `psp_malloc()` to allocate memory for variables and cache.

`USE_MALLOC` also allows setting of `max_volumes`, `max_files` and `max_tasks` at run-time, making it possible to build FAT as a separate library. In this case the application must be compiled with the same CONFIG files.

Note: The file system initialization function `fs_init()` allocates data areas differently depending whether `USE_MALLOC` is enabled or disabled.

If dynamic memory allocation is enabled, options for dynamic allocation of memory per volume are set in the `src/config/config_fat.c` configuration file.

SAFEFAT

The default is 0. Enable this to enable the creation of SafeFAT drives. SAFEFAT and NONSAFEFAT (see below) can both be set to allow a mix of drives to be used.

Note: To allow the selection of drives to be safe or non-safe, the functions `f_initvolume_nonsafe()` and `f_initvolumepartition_nonsafe()` have been added to the system to complement their safe counterparts `f_initvolume()` and `f_initvolumepartition()`.

NONSAFEFAT

The default is 1. This must be set in order to be able to create standard (non-safe) FAT drives. For details, see SAFEFAT above.

F_SAFE_AUTO_REPAIR

If SafeFAT is enabled and this option is set to the default of 1, after power-on the file system is automatically returned to a consistent state.

If this is set to 0 and repair is needed **f_initvolume()** initializes the file system as read-only. In this case, after power-on **f_repair()** must be used to return the file system to a consistent state. Otherwise, read-only access is all that will be available.

BAD_BLOCK_HANDLING

Set this to 1 to enable bad block handling. The default is 0.

There should not be bad blocks on flash USB sticks, memory cards (MMC/SD) and other media that use a Flash Translation Layer (FTL), like HCC's SafeFTL.

File Names

F_LONGFILENAME

Enable this to use long filenames. The default is 0. The system includes two main source files:

File	Description
fat.c	The file system without long filename support. If long filenames exist on the media, the system will ignore the long name part and use only the short name.
fat_lfn.c	The file system with complete long filename support.

Because more system resources are required to handle long filenames, use these only when necessary. This avoids increasing the stack sizes of applications that call the file system, and also increasing the amount of checking that is required.

Note: The maximum long filename space required by the standard is 260 bytes. As a consequence, each time a long filename is processed, large areas of memory must be available. You can, depending on the application, reduce the size of F_MAXPATHNAME and F_MAXLNAME to cut resource use.

The most critical functions for long filenames are **f_rename()** and **f_move()**, which must keep two long filenames on the stack, with additional structures for handling them. If either function is not required for your application, it is sensible to comment it out. This can reduce the stack requirements significantly.

F_MAXSNAME

The length of the name part of a short filename. The default is 8. Do not change this.

F_MAXSEXT

The length of the extension part of a short filename. The default is 3. Do not change this.

FN_MAXLNAME

The maximum length of a long filename. The default is 255. Do not increase this because this will make the system incompatible with other systems. You can decrease it to reduce the resource requirements, in particular the stack.

FN_MAXPATHNAME

The maximum length of a short/long filename with the full path (excluding the drive letter). The default is 256.

You can decrease this to reduce the resource requirements, in particular the stack. Note that this is redefined internally to F_MAXPATHNAME, which is referred to elsewhere in this manual.

F_SHIFT_JIS_SUPPORT

Set this to 1 to enable SHIFT_JIS character encoding in file and directory names. The default is 0.

Other File Options

F_MAXFILES

The total number of files that may be open simultaneously across all volumes. The default is 4.

F_MAXSEEKPOS

The number of seek points in a file to be stored with each file descriptor. If this is set to 0, seeking always works from the current position or the beginning of the file. The default is 8. F_MAXSEEKPOS should be set equal to a power of 2 or to 0.

Note: The memory usage of the system is increased by: $F_MAXSEEKPOS * F_MAXFILES * \text{sizeof}(\text{long})$

Seeking to a new position in a large file can be slow with a FAT file system because there are no backward pointers on the cluster chains. To improve seek performance, this option is provided to store key points in the file. If it is enabled then, as the file is processed, points are recorded at intervals in the file and when the file system is required to seek, these points can be used as a shortcut to get to an offset in the target file.

Example: If there is a 1GB file and there are 8 seek points, these points would be every 128MB. Consequently, the maximum amount of storage space you would have to work through to find a location in the file is 128MB of clusters instead of 1GB; this is a considerable improvement.

The seek points are inserted at regular intervals when the file is opened, on the basis of the known length of the file. These points are not dynamically updated after this, so if you need to refresh the seek points, you must perform a close and open on the file.

F_UPDATELASTACCESSDATE

If you enable this option, whenever you open a file for read ("r"), a sector write is performed on the directory entry. This updates the last accessed date (the date is checked before updating, to ensure it needs updating).

To avoid this overhead, keep 0 (the default), in which case only other file manipulations ("r+", "w", "w+", "a" and "a+") change the date entry.

F_FINDOPENFILESIZE

In standard file systems, if a file is open for writing or append its length is not updated until the file is closed. Any function that uses the length of the file, for example **f_filelength()**, gets the pre-open value.

Enabling this option allows the dynamic file length of the file to be used when these functions are called. The default is 0.

F_DELETE_CONTENT

If this is set to 1, after a file is deleted or content is truncated, all data is destroyed (overwritten to 0xFF). The default is 0.

If IOCTL is available, FAT uses those functions to perform the deletions. Otherwise the deletes are performed manually by the file system.

When deletion is enabled, the **f_deletecontent()** function may be called instead of the standard **f_delete()**. This call removes the file and destroys its contents.

This system has been designed to interoperate with HCC's SafeFTL flash translation layer so that the original data is overwritten.

On an HDD this will work normally, since logical sectors are directly mapped onto the physical disk.

On a flash card this option will not help because, when new data are written, a new block of flash is allocated and the original data may still exist on the disk, although it will not be accessible through normal methods.

Note:

- There is a significant overhead involved in erasing all data that has been written. Therefore, use of this function is recommended only when it is important to ensure that deleted data are no longer accessible.
- This can guarantee the erasure of data only if the underlying media erases the original physical sectors. Therefore, systems that have a physical-logical mapping of the data need special handling.
- If there is an interruption, such as switching off the system during a **f_deletecontent()**, it cannot be guaranteed that the deletion of data will be completed. If data deletion is important, use of HCC's SafeFAT file system is recommended.

F_DIR_WALK_SUPPORT

Set this to 1 to enable the **f_dir_walk()** function. The default is 0.

F_DIR_WALK_MAX_DIR_DEPTH

The maximum directory depth to walk. The default is 128.

F_DISK_USAGE_SUPPORT

If **F_DIR_WALK_SUPPORT** is set, **F_DISK_USAGE_SUPPORT** is set to 1 as well to enable the **f_disk_usage()** and **f_volume_clean()** functions. (**f_disk_usage()** uses the **f_dir_walk()** function.)

F_SAFE_LOG_CLUSTER_SHIFT_BYTES

A cluster of safe logs is allocated from a different place than the regular files/directories. This option sets the distance between the clusters of regular files and log files in bytes. The default is (8 * 1024 * 1024).

FAT_FMW_ENABLE

Enable File Multiple Write (FMW). This allows opening of the same file for writing multiple times.

Volume Definitions

FAT_MAXVOLUME

The maximum number of volumes allowed on the system. The default is 2. Volumes are given drive letters as specified by **f_initvolume()**.

The system is designed so that access to a specific volume is entirely independent of any other volumes. That is, if an operation is being performed on a volume it does not block access to other volumes.

FAT_MAXTASK

The number of tasks that are allowed to access the file system simultaneously.

If this is set to 1 (the default), it implies that no OS is used, or that all accesses are controlled through a single task.

FN_CURRDRIVE

This determines which drive of the system is used at system startup. If -1 is set there is no default current drive. The default is 0.

F_PATH_SEPARATOR

The default is '/'. Set this to '\\\' for FAT to use backslash as the pathname separator character.

F_DRIVE_SEPARATOR

The drive name separator character in full pathnames. The default is ':\'.

F_VOLNAME_SUPPORT

Set **F_VOLNAME_SUPPORT** to enable the **f_setvolname()** and **f_getvolname()** calls and the use of named volumes. The default is 0.

F_VOLUME_CLEAN_SUPPORT

Keep the default of 1 to enable the **f_volume_clean()** function.

F_VOLUME_CLEAN_FILL_CHAR

The character to use for filling the unused space and directory entries. The default is 0xFF.

Note: The following only apply if F_DIR_WALK_SUPPORT and F_VOLUME_CLEAN_SUPPORT are both set.

F_SZ_VOLUME_CLEAN_BUF1, F_SZ_VOLUME_CLEAN_BUF2, F_SZ_VOLUME_CLEAN_BUF3, F_SZ_VOLUME_CLEAN_BUF4

A larger buffer size may make volume clean faster. The minimum size is F_SZ_MAX_SECTOR_n (1 to 4) and the maximum is 64K (FAT32 max. cluster size).

The default for all four options is (1u * 1024).

Sector Size

F_DEF_SECTOR_SIZE

The default sector size in bytes used when formatting media. The default is 512. On removable media it may be dangerous to change this value because many systems accept only FAT file systems with 512 byte sectors.

F_MAX_SECTOR_SIZE

The maximum supported sector size of the attached media in bytes. The default is 512. Traditionally most FAT-based devices have used a sector size of 512 bytes. However, for devices whose native sector size is not 512 bytes (for example, 2K page NAND flash-based devices), it can be more efficient to use other values.

F_SZ_MAX_SECTOR_x

The maximum sector size in bytes for each volume. The default is 512. The "x" represents the volume number for which the cache is allocated.

A variable sector size is normally required only in systems that have removable media. If a larger than necessary maximum sector size is set, the system uses more RAM. Therefore, in resource-constrained systems, it may be necessary to restrict the allowed sector size.

Note: Do not attach a device with an unsupported sector size. If you do, the error F_ERR_NOTSUPPSECTORSIZE is returned.

The cache options (FATCACHE and DIRCACHE) always allocate buffers of size F_SZ_MAX_SECTOR_SIZE. If the attached media has a smaller sector size, it fills the buffer anyway.

For example, if FATCACHE_READAHEAD_x is set to 4 and F_SZ_MAX_SECTOR_SIZE_x is 2048 then, if media with 512 byte sectors are connected, the allocated FAT cache block will be 4 * 2048 bytes (that is, 8192 bytes). The read-ahead size will therefore be 16 sectors of 512 bytes each.

Caching

FATCACHE_ENABLE

Keep this at the default of 1 to enable [FAT caching](#).

DIRCACHE_ENABLE

Keep this at the default of 1 to enable [directory caching](#).

FATBITFIELD_ENABLE

The default is 0. If this is enabled the system attempts to **psp_malloc()** a block to contain a bit table of free clusters. This table is maintained by the file system and is used to accelerate searches for free clusters. The table of free clusters improves writing performance substantially when writing to a large and full disk.

This option is available only if [USE_MALLOC](#) is defined. **psp_malloc()** is used because the size of this area cannot be fixed since it depends on the size and format of the attached media. The implementation of **psp_malloc()** is performed in the PSP, so you can decide how this is implemented.

WR_DATACACHE_SIZE

The number of write cache entries. The default is 8. Refer to [Write Caching](#).

F_CLUSTERBUFFER

Set this to 1 to allocate a dedicated cluster buffer for speeding up overwriting of large files in SAFEFAT mode. The default is zero.

When [USE_MALLOC](#) is not set, the actual size of this buffer for each volume is determined by `F_N_CLUSTERBUFFERx` (see below).

Volume-dependent Configuration Templates

The following options create volume-dependent configuration templates for up to four volumes, based on the value of `FAT_MAXVOLUME`. In each case the "x" represents the relevant volume number. If you need more volumes, you can add more volume-dependent settings using the existing templates.

F_N_FATCACHE_BLOCKSx

The number of FAT cache blocks on the volume. The default is 4. Refer to [FAT caching](#).

F_N_FATCACHE_READAHEADx

The number of FAT cache readahead blocks on the volume. The maximum is 256, depending on `F_MAX_SECTOR_SIZE`. The default is 8. Refer to [FAT caching](#).

F_N_DIRCACHE_SECTORSx

The number of sectors to read ahead on the volume. The maximum is 32 (\leq maximum cluster size). The default is 8. Refer to [Directory Cache](#).

F_N_CLUSTERBUFFERx

The number of sectors buffered when copying clusters on the volume. Set this greater than 1 to speed up overwriting of large files in SAFEFAT mode.

Encryption

FAT_ENCRYPTION

Set this to 1 to enable AES encryption for files. The default is 0.

Note: Do not change the following two options unless you want to replace AES128 with another type of AES encryption.

FAT_ENC_KEY_SIZE

The encryption key size. The default is AES_128_KEY_LEN.

FAT_ENC_BLOCK_SIZE

The encryption block size. The actual sector size must be a multiple of FAT_ENC_BLOCK_SIZE. The default is AES_BLOCK_SIZE.

5.2. config_fat.c

The memory allocated dynamically at initialization for the file system to use is controlled by the *fat_cache_sectors[]* array in the file **src/config/config_fat.c**. This array is only used when [USE_MALLOC](#) is set to 1; otherwise static memory buffers are used, with their sizes configured in **config_fat.h**.

The *fat_cache_sectors[]* array defines the FAT/directory cache used, based on the number of sectors present on the media.

Set up the array to configure FATCACHE and DIRCACHE for the following volume sizes:

- 1024
- 16384
- 1048576
- 0xFFFFFFFF

For FATCACHE, specify the F_N_FATCACHE_BLOCKS and F_N_FATCACHE_READAHEAD parameters. For DIRCACHE, specify the F_N_DIRCACHE_SECTORS.

The array contains volume sizes in increasing order and FAT takes the best matching entry when allocating space for CACHE. The array may contain only a single non-zero element, but the trailing element must be all-zero.

6. Other File System Information

This section:

- describes the system, stack, and real time requirements.
- describes the functions FAT provides for creating and managing multiple drives, partitions and volumes.
- gives information about FAT formats that may be useful.

6.1. System Requirements

The FAT system is designed to be as open and portable as possible. No assumptions are made about the functionality or behavior of the underlying operating system.

For the system to work at its best, perform the porting work outlined in the following sections. This is a straightforward task for an experienced engineer.

6.2. Stack Requirements

File system functions are always called in the context of the calling thread or task. Naturally, the functions require stack space and you should allow for this in applications that call file system functions. Typically, calls to the file system use <2KB of stack. However if [long filenames](#) are used increase the stack size to 4KB; see [Directory Cache](#).

6.3. Real-Time Requirements

The bulk of the file system is code that executes without delay. There are exceptions at the driver level, where delays in reading and writing from/to the physical media, and in the communication itself, cause the system to wait on external events. The points at which delays occur are documented in the relevant driver documents.

Modify drivers to meet the FAT system's requirements, either by implementing interrupt control of the relevant events, or scheduling other parts of the system that can proceed without completion of the events. Refer to the relevant driver documents for details.

6.4. Drives, Partitions and Volumes

FAT provides functions for creating and managing multiple drives, partitions and volumes. First, note the following definitions:

- A drive consists of a physical medium that is controlled by a single driver. Examples are an HDD and a Compact Flash card.
- All drives contain zero or more partitions. If a drive is not partitioned, there is just a single volume on that drive.
- A single volume may be added to each partition. A volume can exist on a drive without partitions.

FAT operates on volumes. You can have one volume or a set of volumes. Additional functions are provided to work with multivolume sets (A:, B:, C:, and so on).

Note: The API functions **f_getdrive()**, **f_chdrive()**, and **f_getcwd()** refer to drives by name because this is the convention, but the names are really references to volumes.

If using multiple partitions, use the following functions to create drivers for partitioned drives, and to create partitions on those drives:

- **f_initvolumepartition()**
- **f_createdriver()**
- **f_releasedriver()**
- **f_createpartition()**

Partitions are created on a single volume such as an HDD, so a single driver is used to access the volume even though there are multiple partitions on it. These volumes need to be controlled by a single lock.

Note: Some operating systems do not recognize multiple partitions on removable media. It is therefore "normal" to restrict the use of multiple partitions to fixed drives. FAT-created partitions are compatible with Windows XP.

6.5. Drive Format

This document does not describe a FAT file system in detail as HCC's FAT system handles the majority of the features of a FAT file system with no need for you to understand further. However, the following information about FAT formats may be useful.

There are three different ways in which your removable media may be formatted:

- Unformatted.
- Formatted without partition table.
- Formatted with partition table.

An unformatted drive is not useable until it has been formatted. Most flash cards are pre-formatted, whereas hard disk drives tend to be unformatted when delivered.

The use of the **f_createpartition()**, **f_initvolumepartition()** and **f_format()** functions is defined in [File Management](#).

6.6. Cache Setup and Options

The file system includes two caching mechanisms to enhance performance: FAT caching and Write caching.

FAT Caching

FAT caching enables the file system to read several sectors from the FAT in one access, so that it's not necessary to read new FAT sectors so frequently.

FAT caching is arranged in blocks so that each block can cover different areas of the FAT. The number of sectors that each block contains and the number of blocks are configurable.

FAT caching requires 512 additional bytes of RAM per sector.

The following definitions are provided for the first volume in **config_fat.h**. Each volume configured needs its own set of definitions.

```
#define F_SZ_MAX_SECTOR1 512
#if FATCACHE_ENABLE
    #define F_N_FATCACHE_BLOCKS1    4
    #define F_N_FATCACHE_READAHEAD1 8 /* Max 256, depending on F_MAX_SECTOR_SIZE */
#endif
#if DIRCACHE_ENABLE
    #define F_N_DIRCACHE_SECTORS1   8 /* Max of 32 (<= maximum cluster size) */
#endif
```

Note:

- The additional RAM required for FAT caching for each configured volume is:
FATCACHE_BLOCKSx * FATCACHE_READAHEADx * F_SZ_MAX_SECTORx
- The default settings shown above for the first volume require 16KB of additional RAM.

Write Caching

The write cache defines the maximum number of sectors that can be written in one operation from the caller's data buffer. This also depends on the availability of contiguous space on the target drive.

The write cache requires an F_POS structure (24 bytes) for each entry it has. The purpose of these structures is to be able to wind back a multi-sector write in the event of an error in writing.

The default setting for write caching in **config_fat.h** is:

```
#define WR_DATACACHE_SIZE 8
```

This requires 192 additional bytes of RAM.

Directory Cache

For the directory cache to be enabled on each volume, [F_LONGFILENAME](#) must be defined.

This can be done by defining [DIRCACHE_ENABLE](#) in **config_fat.h**, at which time you must specify the number of sectors to read ahead on each volume with `F_N_DIRCACHE_SECTORSx`. This allocates the specified number of sectors of memory for directory caching (for example, if set to 32, 16KB of memory will be allocated, assuming a maximum sector size of 512 is configured for that volume.).

Note: The system never reads more than the size of a cluster into this cache. Therefore, there is no value in having a [F_N_DIRCACHE_SECTORSx](#) greater than the number of sectors per cluster on volume x.

6.7. Use of Wildcards

Wildcard characters can be used to find files or directories. They can be used only as parameters for the **f_findfirst()** function; they are then re-used when **f_findnext()** is called again.

The valid wildcard characters are:

Wildcard	Action
*	Matches any string.
?	Matches any single character.,
""	Matches a string up to the end of file or the first "." or from the first "." to the end of file. So ".*" is required to access all files or directories in the target directory.

Note: If you want to perform a logical operation such as `fdelete(".')`, you need to call **f_findfirst()** or **f_findnext()** repeatedly. When each name is returned in the `F_FIND` structure, you must use that as a parameter to **f_delete()**.

7. Application Programming Interface

This section describes all the Application Programming Interface (API) functions. It includes all the functions that are available to an application program.

7.1. Module Management

The functions are the following:

Function	Description
fs_init()	Initializes the file system and allocates the required resources.
fs_start()	Starts the file system.
fs_stop()	Stops the file system.
fs_delete()	Releases resources allocated during the initialization of the file system.

fs_init

Use this function to initialize the file system. Call it once at start-up.

Data areas for the file system to use are allocated at compile time, as follows:

- If [USE_MALLOC](#) is set to 0, allocation is based on the settings for each volume in the `src/config/config_fat.h` file.
- If [USE_MALLOC](#) is set to 1, allocation is controlled by the `fat_cache_sectors[]` array in the file [config_fat.c](#).

Format

```
int fs_init ( void )
```

Arguments

None.

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void main()  
{  
    fs_init(); /* Initialize the file system */  
    fs_start(); /* Start the file system */  
    .  
    .  
}
```

fs_start

Use this function to start the file system.

This function must complete successfully before the file system can be used.

Note: Call `fs_init()` before this to initialize the file system.

Format

```
int fs_start ( void )
```

Arguments

None.

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

fs_stop

Use this function to stop the file system.

After this, the file system cannot be used until a new call to **fs_start()** is successfully completed.

Format

```
int fs_stop ( void )
```

Arguments

None.

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

fs_delete

Use this function to to release resources allocated during the initialization of the file system.

Note: All volumes must be deleted before this function is called.

Format

```
int fs_delete ( void )
```

Arguments

None.

Return values

Return value	Description
F_NO_ERROR	Successful execution.
F_ERR_BUSY	A volume has not been deleted and this prevented the successful completion of this function.

7.2. File System API

This section describes all the Application Programming Interface (API) functions available, apart from Unicode functions. It is split into functions for general, volume, and directory management, file access and file management.

General Management

The functions are the following:

Function	Description
f_enterFS()	Creates resources for the calling task in the file system and allocates a current working directory for that task.
f_releaseFS()	Releases a previously assigned unique task ID.
f_getlasterror()	Returns the last error code.

f_enterFS

Use this function to create resources for the calling task in the file system and allocate a current working directory for that task.

Note:

- If the target system allows multiple tasks to use the file system, this function must be called by a task before it uses any other file management API functions.
- For the correct operation of this function, **oal_get_task_id()** in the [OS Abstraction Layer](#) must have been ported to give a unique identifier for each task.

f_releaseFS() must be called to release the task from the file system and free the allocated resource. If the system is a single task-based system, also call this function after calling **fs_init()**.

Format

```
int f_enterFS ( void )
```

Arguments

None.

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void main()
{
    fs_init();    /* Initialize the file system */
    fs_start();  /* Start the file system */
    f_enterFS(); /* Allow the current (only) task to access the file system */
    .
    .
}
```

f_releaseFS

Use this function to release the file system from the calling task.

This function releases the entry so another slot is available for tasks to be able to use the file system. You must call it if a given task is released or no longer exists.

Format

```
void f_releaseFS ( void )
```

Arguments

None.

Return values

None.

Example

```
void task_destructor()  
{  
    f_releaseFS(); /* Release the current task ID */  
    .  
    .  
    .  
}
```

f_getlasterror

Use this function to return the last error code.

The last error code is cleared/changed when any API function is called.

Format

```
int f_getlasterror ( )
```

Arguments

None.

Return values

Return value	Description
Error code	The last error code.

Example

```
int myopen()
{
    F_FILE *file;
    file = f_open( "nofile.tst", "rb" );
    if (!file)
    {
        int rc = f_getlasterror();
        printf( "f_open failed, errorcode:%d\n", rc );
        return rc;
    }

    return F_NO_ERROR;
}
```

Volume Management

Note: The API functions **f_getdrive()**, **f_chdrive()** and **f_getdcwd()** use the term "drive" because this is the convention. This is equivalent to the term "volume".

The functions are the following:

Function	Description
f_initvolume()	Initializes a volume.
f_initvolume_nonsafe()	Initializes a volume. (The drive will be a standard FAT drive.)
f_delvolume()	Deletes an existing volume.
f_checkvolume()	Checks the status of a drive that has been initialized.
f_repair()	Repairs a SafeFAT file system.
f_setvolname()	Sets the name of a volume.
f_getvolname()	Gets the name of a volume.
f_get_oem()	Returns the OEM name in the disk boot record.
f_get_volume_count()	Gets the number of volumes currently available to the user.
f_get_volume_list()	Gets a list of volumes currently available to the user.
f_initvolumepartition()	Initializes a volume on an existing partition.
f_initvolumepartition_nonsafe()	Initializes a volume on an existing partition. (The drive will be a standard FAT drive.)
f_format()	Formats the specified drive.
f_createpartition()	Creates one or more partitions on a drive, or removes partitions by overwriting the current partition table.
f_createpartition_align()	Creates one or more partitions on a drive, drive, aligned to given sector boundaries, or removes partitions by overwriting the current partition table.
f_getpartition()	Gets the used sectors and system indication byte from a partitioned medium.
f_createdriver()	Initializes a driver.
f_releasedriver()	Releases a driver when it is no longer required.
f_chdrive()	Changes to a new current drive.
f_getdrive()	Gets the current drive number.
f_getfreespace()	Fills a structure with information about the drive space usage: total space, free space, used space, and bad (damaged) size.
f_getlabel()	Returns the label as a function value.

Function	Description
f_setlabel()	Sets a volume label.
f_get_cluster_size()	Gets the cluster size used on a volume.
f_volume_clean()	Runs a volume clean.

f_initvolume

Use this function to initialize a volume. Call it with a pointer to the driver function that must be called to retrieve drive configuration information from the relevant driver.

This function works independently of the status of the hardware; that is, it does not matter whether a card is inserted or not.

In non-multitask systems **f_initvolume()** must be followed by an **f_chdrive()** function to select the current drive for relative file path accessing. In a multitask system every **f_enterFS()** function needs to be followed by an **f_chdrive()** function if the task is using drive relative accessing.

f_initvolume() always initiates the first partition on the media. To use multiple partitions, use **f_initvolumepartition()**.

Format

```
int f_initvolume (
    int          drvnumber,
    F_DRIVERINIT driver_init,
    unsigned long driver_param )
```

Arguments

Argument	Description	Type
drvnumber	The drive to initialize (0='A', 1='B', and so on).	int
driver_init	The initialization function for the driver.	F_DRIVERINIT
driver_param	This can optionally be used to pass information to the low level driver. Its use is driver-dependent. When the xxx_initfunc() of the driver is called, this parameter is passed to the driver. One use for this is to specify which device associated with the specified driver will be initialized. For more information, refer to the Media Driver manuals .	unsigned long

Return values

Return value	Description
F_NO_ERROR	Successful execution.
F_ERR_CARDREMOVED	The volume has been successfully created; f_initvolume() does not need to be called again. When a card is inserted, the volume will be fully functional.
Else	See Error Codes .

Example

```
void myinitfs( void )
{
    int ret;
    fs_init(); /* Initialize the file system */
    fs_start(); /* Start the file system */
    f_enterFS();

    /* Create a RAM volume on Drive A */
    f_initvolume( 0, ram_initfunc, 0 );

    /* Create an MMC Volume on Drive B */
    f_initvolume( 1, mmc_initfunc, 0 );

    /* Create a Mass Storage Volume on Drive C */
    f_initvolume( 2, mst_initfunc, 0 );

    /* Create a second Mass Storage Volume on Drive D */
    f_initvolume( 3, mst_initfunc, 1 );
        .
        .
        .
}
```


f_initvolume_nonsafe

Use this function to initialize a volume.

Call the function with a pointer to the driver function that must be called to retrieve drive configuration information from the relevant driver. It works independently of the status of the hardware; that is, it does not matter whether a card is inserted or not.

Note: If this function is used instead of **f_initvolume()**, the drive will be a standard FAT drive. It will not be protected by the SafeFAT journaling mechanisms.

This function can be used to obtain a mix of drive types, perhaps to allow a faster non-secure drive for less critical data.

In non-multitask systems **f_initvolume_nonsafe()** must be followed by an **f_chdrive()** function to select the current drive for relative file path accessing. In a multitask system every **f_enterFS()** must be followed by **f_chdrive()** if the task is using drive relative accessing.

This function always initiates the first partition on the media. To use multiple partitions, use **f_initvolumepartition_nonsafe()**.

Format

```
int f_initvolume_nonsafe (
    int          drvnumber,
    F_DRIVERINIT driver_init,
    unsigned long driver_param)
```

Arguments

Argument	Description	Type
drvnumber	The drive to initialize (0='A', 1='B', and so on).	int
driver_init	The initialization function for the driver.	F_DRIVERINIT
driver_param	This can optionally be used to pass information to the low level driver. Its use is driver-dependent. When the xxx_initfunc of the driver is called, this parameter is passed to the driver. One use for this is to specify which device associated with the specified driver will be initialized. For more information, refer to the Media Driver manuals .	unsigned long

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myinitfs( void )
{
    int ret;
    fs_init(); /* Initialize the file system */
    fs_start(); /* Start the file system */
    f_enterFS();

    /* Create a RAM volume on Drive A */
    f_initvolume_nonsafe( 0, ram_initfunc, 0 );

    /* Create an MMC Volume on Drive B */
    f_initvolume_nonsafe( 1, mmc_initfunc, 0 );

    /* Create a Mass Storage Volume on Drive C */
    f_initvolume_nonsafe( 2, mst_initfunc, 0 );

    /* Create a second Mass Storage Volume on Drive D */
    f_initvolume_nonsafe( 3, mst_initfunc, 1 );
    :
    :
}
```

f_delvolume

Use this function to delete an existing volume.

Note that:

- The link between the file system and the driver is broken; that is, an **xxx_release()** call is made to the driver.
- Any open files on the media are marked as closed, so that subsequent API accesses of a previously opened file handle return an error.
- If the volume's driver was created independently by using **f_createdriver()**, this function deletes only the volume. Call **f_releasedriver()** to call **xxx_release()** driver functions.

This function works independently of the status of the hardware; that is, it does not matter whether a card is inserted or not.

Format

```
int f_delvolume ( int drivenum )
```

Arguments

Argument	Description	Type
drivenum	The drive to delete (0='A', 1='B', and so on).	int

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void mydelfs( int num )
{
    int ret;

    /* Delete volume */
    if (f_delvolume( num ))
        printf( "Unable to delete volume %c", 'A' + num );
        .
        .
}
```

f_checkvolume

Use this function to check the status of an initialized drive.

Format

```
int f_checkvolume ( int drivenum )
```

Arguments

Argument	Description	Type
drivenum	The drive to check (0='A', 1='B', and so on).	int

Return values

Return value	Description
F_NO_ERROR	The drive is working.
Else	See Error Codes .

Example

```
void mychkfs( int num )
{
    int ret;

    /* Checking volume */
    if (f_checkvolume( num ))
    {
        printf( "Volume %d is not usable, Error %d", num, ret );
    }
    else
    {
        printf( "Volume %d is working, no error", num );
    }
    .
    .
}
```

f_repair

Use this function to repair a SafeFAT file system.

If the volume was not unmounted while a file was opened for writing, or a file system operation was interrupted by removing the media or a power failure, it may be necessary to call this function.

Note: If F_SAFE_AUTO_REPAIR is set, this function is not available as the system is repaired automatically at power-on.

Format

```
int f_repair (int drivenum )
```

Arguments

Argument	Description	Type
drivenum	The volume number to repair (0='A', 1='B', and so on).	int

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
ret = f_initvolume( drivenum, mmc_sd_initfunc, driver_param );  
  
if ( ret == F_ERR_REPAIRNEEDED )  
{  
    ret = f_repair( drivenum );  
}
```

f_setvolname

Use this function to set the name of a volume.

The volume name is the human-readable name given to a volume. This name is stored in the microcontroller's RAM only; it is not written to the media itself.

For example, suppose you have two volumes: volume 0 is SPI NAND and volume 1 is an SD card. You could use **f_setvolname (0, "spi")** and **f_setvolname(1, "sd")**.

You could access files on the SPI with **f_open("spi:my_file")** and files on the SD card with **f_open("sd:/other_file")**.

Note: [F_VOLNAME_SUPPORT](#) must be set to enable this function.

Format

```
int f_setvolname (
    int      drivenum,
    const char * p_name )
```

Arguments

Argument	Description	Type
drivenum	The volume number (0='A', 1='B', and so on).	int
p_name	The name to give the volume. This must be at least two characters long and null-terminated.	char *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void setvolname( void )
{
    int ret;

    ret = f_setvolname( f_getdrive(), "sdcard" );
    if ( ret == F_NO_ERROR )
    {
        ret = f_mkdir( "sdcard:/testdir" );
    }
    else
    {
        printf( "Error during setting volname!\r\n" );
    }
    if ( ret == F_NO_ERROR )
    {
        ret = f_mkdir( "sdcard:/testdir/testdir2" );
    }
}
```

f_getvolname

Use this function to get the name of a volume.

Note: [F_VOLNAME_SUPPORT](#) must be set to enable this function.

Format

```
int f_getvolname (
    int    drivenum,
    char *  p_buffer,
    int    maxlen )
```

Arguments

Argument	Description	Type
drivenum	The volume number (0='A', 1='B', and so on).	int
p_buffer	Where to store the name of the volume.	char *
maxlen	The buffer length (the maximum size is F_MAXPATHNAME).	int

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void getvolname( void )
{
    int ret;
    char volname[8];

    ret = f_getvolname( f_getdrive(), volname, sizeof( volname ) );
    if ( ret == F_NO_ERROR )
    {
        printf( "Volume name is '%s'\r\n", volname );
    }
    else
    {
        printf( "Error on drive!\r\n" );
    }
}
```


f_get_oem

Use this function to return the OEM name in the disk boot record.

Format

```
int f_get_oem (
    int     drivenum,
    char *   str,
    long    len )
```

Arguments

Argument	Description	Type
drivenum	The drive number (0='A', 1='B', and so on).	int
str	Where to copy the label to. This should be able to hold an eight character string.	char *
len	The length of the storage area.	long

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void get_disk_oem( void )
{
    char oem_name[9];
    int result;

    oem_name[8] = 0; /* Zero-terminate string */
    result = f_get_oem( f_getdrive(), oem_name, 8 );

    if (result)
    {
        printf( "Error on drive!" );
    }
    else
    {
        printf( "Drive OEM is %s", oem_name );
    }
}
```

f_get_volume_count

Use this function to obtain the number of active volumes.

Format

```
int f_get_volume_count ( void )
```

Arguments

None.

Return values

Return value	Description
num	The number of active volumes.
Else	See Error Codes .

Example

```
void mygetvols( void )
{
    printf( "There are %d active volumes\n", f_get_volume_count() );
    :
    :
}
```

f_get_volume_list

Use this function to obtain a list of all the active volumes.

Format

```
int f_get_volume_list ( int * buffer )
```

Arguments

Argument	Description	Type
buffer	Where to store the volume list. The storage should be of size FAT_MAXVOLUME .	int *

Return values

Return value	Description
number	The number of active volumes.
Else	See Error Codes .

Example

```
void mygetvols( void )
{
    int i, j;
    int buffer[F_MAXVOLUME];

    i = f_get_volume_list( buffer );

    if (!i) printf( "No active volume found\n" );

    for (j = 0; j<i; j++)
    {
        printf( "Volume %d is active\n", buffer[j] );
    }
}
```

f_initvolumepartition

Use this function to initialize a volume on an existing partition.

Call this function with a common driver structure pointer that can be retrieved by calling **f_createdriver()**. This driver pointer can be used for initializing all existing partitions on the media.

If only the first partition is used, use **f_initvolume()** instead.

Format

```
int f_initvolumepartition (  
    int          drvnumber,  
    F_DRIVER *   driver,  
    int          partition )
```

Arguments

Argument	Description	Type
drvnumber	The drive to initialize (0='A', 1='B', and so on).	int
driver	A pointer to the initialized driver; get this by calling f_createdriver() .	F_DRIVER *
partition	The partition to build.	int

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
F_DRIVER *hdd;

int myinitfs( void )
{
    int ret;

    fs_init(); /* Initialize the file system */
    fs_start(); /* Start the file system */
    f_enterFS();

    ret = f_createdriver( &hdd, hdd_initfunc, 0 );
    if (ret) return ret;

    ret = f_initvolumepartition( 0, hdd, 0 );
    if (ret) return ret;

    ret = f_initvolumepartition( 1, hdd, 1 );

    return ret;
}
```

f_initvolumepartition_nonsafe

Use this function to initialize a volume on an existing partition.

Call this function with a common driver structure pointer that can be retrieved by calling **f_createdriver()**. This driver pointer can be used for initializing all existing partitions on the media.

Note:

- If this call is used instead of **f_initvolumepartition()**, the drive will be a standard FAT drive; it will not be protected by the SafeFAT journaling mechanisms.
- If only the first partition is used, use **f_initvolume_nonsafe()** instead.

This function can be used to obtain a mix of drive types, perhaps to allow a faster non-secure drive for less critical data.

Format

```
int fm_initvolumepartition_nonsafe (  
    int          drvnumber,  
    F_DRIVER *   driver,  
    int          partition)
```

Arguments

Argument	Description	Type
drvnumber	The drive to initialize (0='A', 1='B', and so on).	int
driver	A pointer to the initialized driver; get this by calling f_createdriver() .	F_DRIVER *
partition	The partition to be built.	int

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
F_DRIVER *hdd;

int myinitfs( void )
{
    int ret;

    fs_init(); /* Initialize the file system */
    fs_start(); /* Start the file system */
    f_enterFS();

    ret = f_createdriver( &hdd, hdd_initfunc, 0 );

    if (ret) return ret;
    ret = f_initvolumepartition_nonsafe( 0, hdd, 0 );

    if (ret) return ret;
    ret = f_initvolumepartition_nonsafe( 1, hdd, 1 );

    return ret;
}
```

f_format

Use this function to format the specified drive.

If the media is not present, this function fails. If it succeeds, all data on the specified volume are destroyed and any open files are closed.

Any existing master boot record is unaffected by this command. The boot sector information is re-created from the information provided by **f_getphy()**.

Note: The format operation fails if the specified format type is incompatible with the size of the physical media.

Format

```
int f_format (
    int    drivenum,
    long   fattype )
```

Arguments

Argument	Description	Type
drivenum	The drive to format (0='A', 1='B', and so on).	int
fattype	The type of format: <ul style="list-style-type: none">• F_FAT12_MEDIA for FAT12• F_FAT16_MEDIA for FAT16• F_FAT32_MEDIA for FAT32	long

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Sectors per cluster

The number of sectors per cluster on a FAT32 drive is determined by the table below, which is included in the **fat.c** and **fat_lfn.c** files. The table specifies the number of sectors (in hex) on the target device; alongside these, the second number gives the number of sectors per cluster. This table may be modified if required.

```
static const t_FAT32_CS FAT32_CS[] =
{
/* {Up to this number of sectors, this is sectors/cluster} */

  { 0x00020000, 1 }, /* ->64MB */
  { 0x00040000, 2 }, /* ->128MB */
  { 0x00080000, 4 }, /* ->256MB */
  { 0x01000000, 8 }, /* ->8GB */
  { 0x02000000, 16 }, /* ->16GB */
  { 0x0FFFFFF0, 32 } /* -> ... */
};
```

Example

```
void myinitfs( void )
{
  int ret;

  fs_init(); /* Initialize the file system */
  fs_start(); /* Start the file system */
  f_enterFS();

  f_initvolume( 0, cfc_initfunc, 0 );

  ret=f_format( 0, F_FAT16_MEDIA );

  if (ret)
  {
    printf( "Unable to format CFC: Error %d", ret );
  }
  else
  {
    printf( "CFC formatted" );
  }

  .
  .
}
```

f_createpartition

Use this function to create one or more partitions on a drive, or to remove partitions by overwriting the current partition table.

The partition table is placed in the master boot sector (sector 0) and the first partition is placed at sector 16 (offset by 15 from the boot record). The partitions then follow contiguously as defined in the partition table.

If n = sector count of the media - 16, then n sectors can be divided between the partitions. If a partition's size is not aligned to 16 sectors, the remaining 1 .. 15 sectors cannot be used by the next partition.

To find the number of sectors on the target drive, call the `driver->getphy(driver,&phy)`. You can use this information to build the [F_PARTITION](#) structure before you call **f_createpartition()**.

Note:

- **Calling this function logically destroys all data on the drive.**
- If partition alignment is required, use **f_createpartition_align()** instead of this call.
- If only a single volume is required, it is simpler not to use a partition table but to use **f_initvolume()** to format it.

Format

```
int f_createpartition (
    F_DRIVER *    driver,
    int          parnum,
    F_PARTITION * par )
```

Arguments

Argument	Description	Type
driver	The initialized driver; get this by calling f_createdriver() .	F_DRIVER *
parnum	The number of partitions in the partitions table.	int
par	A pointer to the partition descriptor.	F_PARTITION *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
static F_PARTITION par2[2] =
{
    {1000, F_SYSIND_DOSFAT16UPT032MB, 0},
    {2000, F_SYSIND_DOSFAT16UPT032MB, 0}
};

F_DRIVER *hdd;

int mypartitiondrive()
{
    int ret;

    fs_init(); /* Initialize the file system */
    fs_start(); /* Start the file system */
    f_enterFS();

    ret = f_createdriver( &hdd, hdd_initfunc, 0 );
    if (ret) return ret;

    ret = f_createpartition( hdd, 2, par2 );
    if (ret) return ret;

    return ret;
}
```

f_createpartition_align

Use this function to create one or more partitions on a drive, aligned to given sector boundaries, or to remove partitions by overwriting the current partition table.

Note:

- **Calling this function logically destroys all data on the drive.**
- If partition alignment is not required, you can use **f_createpartition()** instead of this call.
- If only a single volume is required, it is simpler not to use a partition table but to use **f_initvolume()** to format it.

To find the number of sectors on the target drive, call the *driver->getphy(driver,&phy)*. You can use this information to build the [F_PARTITION](#) structure before you call **f_createpartition_align()**.

Format

```
int f_createpartition_align (
    F_DRIVER *    driver,
    int          parnum,
    F_PARTITION * par,
    int          sec_align )
```

Arguments

Argument	Description	Type
driver	The initialized driver; get this by calling f_createdriver() .	F_DRIVER *
parnum	The number of partitions in the partitions table.	int
par	A pointer to the partition descriptor.	F_PARTITION *
sec_align	The number of sectors to align each partition to.	int

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
static F_PARTITION par2[2] =
{
    {1000, F_SYSIND_DOSFAT16UPT032MB, 0},
    {2000, F_SYSIND_DOSFAT16UPT032MB, 0}
};

F_DRIVER *hdd;

int mypartitiondrive()
{
    int ret;

    fs_init(); /* Initialize the file system */
    fs_start(); /* Start the file system */
    f_enterFS();

    ret = f_createdriver( &hdd, hdd_initfunc, 0 );
    if (ret) return ret;

    ret = f_createpartition_align( hdd, 2, par2, 8 );
    if (ret) return ret;

    return ret;
}
```

f_getpartition

Use this function to get the used sectors and system indication byte from a partitioned medium.

For drives that do not contain a partition table, this function returns with the number of sectors and 0 in the system indication byte. If there is a partition table, the function collects information from the partition table entries.

Format

```
int f_getpartition (
    F_DRIVER *    driver,
    int          parnum,
    F_PARTITION * par )
```

Arguments

Argument	Description	Type
driver	The initialized driver; get this by calling f_createdriver() .	F_DRIVER *
parnum	The number of the entry in the <i>par</i> parameter.	int
par	The partition pointer to retrieve information from.	F_PARTITION *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
F_ERR_MEDIATOOLARGE	The space in the F_PARTITION table is insufficient. The medium has more partition table entries than the number passed by the table structure, so increase the number of entries in this table.
Else	See Error Codes .

F_PARTITION

The F_PARTITION structure is defined as:

```
typedef struct
{
    unsigned long secnum;           /* Number of sectors in this partition */
    unsigned char system_indicator; /* Use F_SYSIND_XX values */
} F_PARTITION;
```

Example

```
static F_PARTITION par10[10];

int mypartitionlist( F_DRIVER *driver )
{
    int par;
    int ret = f_getpartition( driver, 10, par10 );
    if (ret) return ret; /* Error */

    for (par=0; par<10; par++)
    {
        printf( "%d par - %d sys_ind %d sectors\n", par, par[10].secnum,
                par10[par].system_indicator );
    }
    return 0;
}
```

f_createdriver

Use this function to initialize a driver.

This function is necessary only if multiple partitions are used. It works independently of the status of the hardware; it does not matter whether a card is inserted or not.

Note: If **f_initvolume()** is used to initiate a volume, this function is not required as it is called automatically.

On a drive that was created directly with this function, you must call **f_releasedriver()** to release the driver.

Format

```
int f_createdriver (
    F_DRIVER * * driver,
    F_DRIVERINIT driver_init,
    unsigned long driver_param )
```

Arguments

Argument	Description	Type
driver	A pointer to the F_DRIVER structure of the required driver.	F_DRIVER **
driver_init	A pointer to the initialization function for the driver. (This must be called to retrieve drive configuration information from the relevant driver.)	F_DRIVERINIT
driver_param	This can optionally be used to pass information to the low level driver. Its use is driver-dependent. When the xxx_initfunc() of the driver is called, this parameter is passed to the driver. One use is to specify which device associated with the specified driver is to be initialized. For more details, see the HCC Media Driver Interface Guide .	unsigned long

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
F_DRIVER *hdd;
int myinitfs( void )
{
    int ret;

    fs_init(); /* Initialize the file system */
    fs_start(); /* Start the file system */
    f_enterFS();

    ret = f_createdriver( &hdd, hdd_initfunc, 0 );
    if (ret) return ret;

    ret = f_initvolumepartition( 0, hdd, 0 );
    if (ret) return ret;

    ret = f_initvolumepartition( 1, hdd, 1 );

    return ret;
}
```

f_releasedriver

Use this function to release a driver when it is no longer required. After this **f_initvolume()** or **f_createdriver()** can be called again.

Use of the function depends on how the driver was created:

- If the driver was created by **f_initvolume()**, do not call this function; **f_delvolume()** releases the driver automatically.
- If the driver was created by **f_createdriver()** then, after **f_delvolume()** has been called for each volume on this drive, call **f_releasedriver()** to release the driver.
- If the driver was created by **f_createdriver()** and **f_releasedriver()** is called, **f_delvolume()** is called automatically for each volume on this drive.

Format

```
int f_releasedriver (  
    F_DRIVER * driver,  
    int      partition )
```

Arguments

Argument	Description	Type
driver	The initialized driver; get this by calling f_createdriver() .	F_DRIVER *
partition	The partition number.	int

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
F_DRIVER *hdd;

int myinitfs( void )
{
    int ret;

    fs_init(); /* Initialize the file system */
    fs_start(); /* Start the file system */
    f_enterFS();

    ret = f_createdriver( &hdd, hdd_initfunc, 0 );
    if (ret)
        return ret;

    ret = f_initvolumepartition( 0, hdd, 0 );
    if (ret)
        return ret;

    ret = f_initvolumepartition( 1, hdd, 1 );

    return ret;
}

int myclose( void )
{
    return f_releasedriver( hdd );
}
```

f_chdrive

Use this function to change to a new current drive.

In non-multitasking and multitasking systems, call **f_chdrive()** if you need relative path access. In a multitasking system, and in a non-multitasking system after **f_initvolume()**, every **f_enterFS()** must be followed by an **f_chdrive()** function call. In a multitasking system every task has its own current drive.

Format

```
int f_chdrive ( int drivenum )
```

Arguments

Argument	Description	Type
drivenum	The number of the drive to change to (0='A', 1='B', and so on).	int

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    .
    .
    f_chdrive( 0 );    /* Select drive A */
    .
    .
}
```

f_getdrive

Use this function to get the current drive number.

Format

```
int f_getdrive ( void )
```

Arguments

None.

Return values

Return value	Description
Current drive	The drive number (0='A', 1='B', and so on).
Else	See Error Codes .

Example

```
void myfunc( void )  
{  
    int currentdrive;  
    .  
    currentdrive = f_getdrive();  
    .  
    .  
}
```

f_getfreespace

Use this function to fill a structure with information about the drive space usage: total space, free space, used space, and bad (damaged) size.

Note:

- If a drive is greater than 4GB, also read the high elements of the returned structure (for example, *pspace.total_high*) to get the upper 32 bits of each number.
- The first call to this function after a drive is mounted may take some time, depending on the size and format of the medium being used. After the initial call, changes to the volume are counted; the function then returns immediately with the data.

Format

```
int f_getfreespace (  
    int      drivenum,  
    F_SPACE * pspace )
```

Arguments

Argument	Description	Type
drivenum	The drive number (0='A', 1='B', and so on).	int
pspace	A pointer to the F_SPACE structure.	F_SPACE *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void info( void )
{
    F_SPACE space;
    int ret;

    /* Get free space on current drive */
    int ret = f_getfreespace( f_getdrive(), &space );

    if (!ret)
    {
        printf( "There are:\n
        %d bytes total,\n
        %d bytes free,\n
        %d bytes used,\n
        %d bytes bad.",\n
        space.total, space.free, space.used, space.bad );
    }
    else
    {
        printf( "\nError %d reading drive\n", ret );
    }
}
```

f_getlabel

Use this function to return the label as a function value.

Format

```
int f_getlabel (
    int     drivenum,
    char *  pLabel,
    long    len )
```

Arguments

Argument	Description	Type
drivenum	The drive number (0='A', 1='B', and so on).	int
pLabel	Where to copy the label to. This should be able to hold an 11 character string.	char *
len	The length of the storage area.	long

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void getlabel( void )
{
    char label[12];
    int result;

    result = f_getlabel( f_getdrive(), label, sizeof( label ) );

    if (result)
    {
        printf( "Error on drive!" );
    }
    else
    {
        printf( "Label of drive is %s", label );
    }
}
```


f_setlabel

Use this function to set a volume label.

This changes the label of the volume and this label is written to the media. If the media is inserted in a PC, the host OS can display it.

Format

```
int f_setlabel (
    int          drivenum,
    const char * pLabel )
```

Arguments

Argument	Description	Type
drivenum	The drive number (0='A', 1='B', and so on).	int
pLabel	A pointer to the null-terminated string to use. This must be an ASCII string with a maximum length of 11 characters. Non-printable characters are padded out as space characters.	char *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void setlabel( void )
{
    int result = f_setlabel( f_getdrive(), "DRIVE 1" );

    if (result)
        printf( "Error on drive!" );
}
```

f_get_cluster_size

Use this function to get the cluster size used on a volume.

Format

```
int f_get_cluster_size (
    F_MULTI *    fm,
    int         drivenum,
    uint32_t *   p_cluster_size_bytes )
```

Arguments

Argument	Description	Type
fm	A multi-structure pointer.	F_MULTI *
drivenum	The volume number (0='A', 1='B', and so on).	int
p_cluster_size_bytes	On return, a pointer to the size of the allocation unit in bytes.	uint32_t *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void cluster_size( void )
{
    int    rc;
    uint32_t cluster_size_bytes;

    rc = f_get_cluster_size( 0, &cluster_size_bytes );
    if ( rc == F_NO_ERROR )
    {
        printf( "Cluster size: %u bytes\r\n", cluster_size_bytes );
    }
    else
    {
        printf( "Error: %i\r\n", rc );
    }
}
```

f_volume_clean

Use this function to run a volume clean. This operation:

1. erases all free clusters.
2. walks the directories and overwrites all deleted directory entries.
3. finds all files and erases all slack space.

Note:

- No files on the volume can be open.
- This is a non-recursive implementation.

Format

```
int f_volume_clean (
    int                drvnumber,
    FN_FIND *         p_find,
    t_volume_clean_cb_fn * p_callback,
    void *            p_user_data )
```

Arguments

Argument	Description	Type
drvnumber	The drive to clean (0='A', 1='B', and so on).	int
p_find	The structure used to find files.	FN_FIND *
p_callback	A pointer to callback function, which is called for every item (directory, file, and cluster) to forward progress.	t_volume_clean_cb_fn *
p_user_data	A pointer to user-defined data, this is passed to the callback function.	void *

Return values

Return value	Description
0	Successful execution.
F_ERR_BUSY	Failed because there are opened files on the volume.
Else	See Error Codes .

Example

```
void volume_clean_example( int vol_id )
{
    int      ret;
    F_FIND  find;
    uint32_t my_data = 42;

    ret = f_volume_clean( vol_id, &find, progress_callback, &my_data );
    if ( ret != F_NO_ERROR )
    {
        printf("Error: %i\r\n", ret);
    }
}
```

This example code shows the callback used:

```

int progress_callback ( void * p_user_data
                        , t_volume_clean_stage_type stage
                        , uint32_t total
                        , uint32_t cmplt )
{
    float percent = 100.f;
    static int prev_percent10 = -1; /* percent multiplied by 10 */
    const char * stage_str[] = {
        "Metadata"
        , "Free space"
        , "Backend"
        , "Complete"
    };
    uint32_t * p_my_data = (uint32_t*) p_user_data;

    if ( *p_my_data != 42 )
    {
        /* Internal error! */
        return 1;
    }

    if ( total > 0 )
    {
        percent = 100.0f * (float)cmplt / (float)total;
    }

    if ( stage < VOLUME_CLEAN_STAGE_BACKEND )
    {
        if ( prev_percent10 != (int)( percent * 10.0f ) )
        {
            printf( "\r%.1f%% completed, items processed %i/%i, stage: %s\t", percent,
                cmplt, total, stage_str[stage] );
            prev_percent10 = (int)( percent * 10.0f );
        }
    }
    else if ( stage == VOLUME_CLEAN_STAGE_BACKEND )
    {
        printf( "\n" );
        printf( "Flushing buffers...\n" );
    }
    else
    {
        printf( "Completed.\n" );
    }

    return 0;
}
    
```

Directory Management

The functions are the following:

Function	Description
f_mkdir()	Creates a new directory.
f_chdir()	Changes the current working directory.
f_rmdir()	Removes a directory.
f_getcwd()	Gets the current working directory.
f_getdcwd()	Gets the current working directory on the selected drive.

f_mkdir

Use this function to create a new directory.

Format

```
int f_mkdir ( const char * dirname )
```

Arguments

Argument	Description	Type
dirname	The name of the directory to create.	char *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    .
    .
    f_mkdir( "subfolder" ); /* Create directories */
    f_mkdir( "subfolder/sub1" );
    f_mkdir( "subfolder/sub2" );
    f_mkdir( "a:/subfolder/sub3" );
    .
    .
}
```

f_chdir

Use this function to change the current working directory.

Every relative path starts from this directory. In a multitasking system every task has its own current working directory.

Format

```
int f_chdir ( const char * dirname )
```

Arguments

Argument	Description	Type
dirname	A null-terminated string containing the name of the directory to change to.	char *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    .
    .
    f_mkdir( "subfolder" );
    f_chdir( "subfolder" );      /* Change directory */
    f_mkdir( "sub2" );
    f_chdir( ".." );           /* Go up one directory level */
    f_chdir( "subfolder/sub2" ); /* Go into directory sub2 */
    .
    .
}
```


f_rmdir

Use this function to remove a directory.

The function returns an error code if:

- The target directory is not empty.
- The directory is read-only.

Format

```
int f_rmdir ( const char * dirname )
```

Arguments

Argument	Description	Type
dirname	The name of the directory to remove.	char *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    .
    .
    f_mkdir( "subfolder" ); /* Create directories */
    f_mkdir( "subfolder/sub1" );
    .
    . /* Do some work */
    .
    f_rmdir( "subfolder/sub1" ); /* Remove directories */
    f_rmdir( "subfolder" );
    .
    .
}
```

f_getcwd

Use this function to get the current working directory on the current drive.

Format

```
int f_getcwd (
    char *  buffer,
    int    maxlen )
```

Arguments

Argument	Description	Type
buffer	Where to store the current working directory string.	char *
maxlen	The length of the buffer.	int

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
#define BUFFLEN F_MAXPATH + F_MAXNAME

void myfunc( void )
{
    char buffer[BUFFLEN];

    if (!f_getcwd( buffer, BUFFLEN ))
    {
        printf( "Current directory is %s", buffer );
    }
    else
    {
        printf( "Drive error!" )
    }
}
```

f_getdcwd

Use this function to get the current working directory on the selected drive.

Format

```
int f_getdcwd (
    int    drivenum,
    char *  buffer,
    int    maxlen )
```

Arguments

Argument	Description	Type
drivenum	The drive number (0='A', 1='B', and so on).	int
buffer	Where to store the current working directory string.	char *
maxlen	The length of the buffer.	int

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
#define BUFFLEN F_MAXPATH + F_MAXNAME

void myfunc( long drivenum )
{
    char buffer[BUFFLEN];

    if (!f_getdcwd( drivenum, buffer, BUFFLEN ))
    {
        printf( "Current directory is %s", buffer );
        printf( "on drive %c", drivenum+'A' );
    }
    else
    {
        printf( "Drive error!" );
    }
}
```

File Access

The functions are the following:

Function	Description
f_open()	Opens a file.
f_open_enc()	Opens an encrypted file.
f_open_nonsafe()	Opens a file without the journaling enabled.
f_close()	Closes a file.
f_abortclose()	Closes a previously opened file, aborting all operations.
f_flush()	Flushes an opened file to a storage medium.
f_read()	Reads bytes from a file at the current file position.
f_write()	Writes data into a file at the current file position.
f_getc()	Reads a character from the current position in an open file.
f_putc()	Writes a character to an open file at the current file position.
f_eof()	Checks whether the current position in an open file is the end of file (EOF).
f_seteof()	Moves the end of file (EOF) to the current file pointer.
f_tell()	Obtains the current read/write position in an open file.
f_seek()	Moves the stream position in a file.
f_rewind()	Sets the file position in an open file to the start of the file.
f_truncate()	Opens a file for writing and truncates it to the specified length.
f_ftruncate()	Truncates a file that is open for writing to a specified length.

f_open

Use this function to open a file. The following opening modes are allowed:

Mode	Description
"r"	Open existing file for reading. The stream is positioned at the beginning of the file.
"r+"	Open existing file for reading and writing. The stream is positioned at the beginning of the file.
"w"	Truncate file to zero length or create file for writing. The stream is positioned at the beginning of the file.
"w+"	Open a file for reading and writing. The file is created if it does not exist; otherwise it is truncated. The stream is positioned at the beginning of the file.
"a"	Open for appending (writing to end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
"a+"	Open for reading and appending (writing to end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

Note the following:

- The same file can be opened multiple times in "r" mode.
- A file can only be opened once at a time in a mode which gives write access (that is, in "r+", "w", "w+", "a" or "a+" mode).
- The same file can be opened multiple times in "r" mode and at the same time once in one of the "r+", "a" or "a+" modes which give write access.
- If a file is opened in "w" or "w+" mode, a lock mechanism prevents it being opened in any other mode. This prevents opening of the file for reading and writing at the same time.

Note: There is no text mode. The system assumes that all files are in binary mode only.

Format

```
F_FILE * f_open (
    const char * filename,
    const char * mode )
```

Arguments

Argument	Description	Type
filename	The file to open.	char *
mode	The opening mode (see above).	char *

Return values

Return value	Description
F_FILE *	A pointer to the handle of the opened file.
NULL	If the pointer is null, the file could not be opened.

Example

```
void myfunc( void )
{
    F_FILE *file;
    char c;

    file = f_open( "myfile.bin", "r" );
    if (!file)
    {
        printf( "File cannot be opened!" );
        return;
    }

    f_read( &c, 1, 1, file ); /* Read one byte */
    printf( "'%' is read from file", c );
    f_close( file );
}
```

f_open_enc

Use this function to open an encrypted file. When a file is opened this way:

- all writes to the file are AES encrypted, using the cipher data provided when it was opened, before being committed to the media.
- all reads from the file are AES decrypted, using the cipher data provided when it was opened, before being passed to the user.

The following opening modes are allowed:

Mode	Description
"r"	Open existing file for reading. The stream is positioned at the beginning of the file.
"a"	Open for appending (writing to end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
"a+"	Open for reading and appending (writing to end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

Note the following:

- This function only applies if [FAT_ENCRYPTION](#) is set.
- The same file can be opened multiple times in "r" mode.
- A file can only be opened once at a time in a mode that gives write access (that is, in "a" or "a+" mode).
- The same file can be opened multiple times in "r" mode and at the same time once in either "a" or "a+" mode.

Note: There is no text mode. The system assumes that all files are in binary mode only.

Format

```
FN_FILE * f_open_enc (
    const char *    p_filename,
    const char *    p_mode,
    const uint8_t * p_key,
    const uint8_t * p_init_vect )
```

Arguments

Argument	Description	Type
p_filename	A pointer to the file to open.	char *
p_mode	A pointer to the opening mode: r, a, or a+ (see above).	char *
p_key	A pointer to an array of FAT_ENC_BLOCK_SIZE bytes holding the encryption key.	uint8_t *
p_init_vect	A pointer to an array of FAT_ENC_BLOCK_SIZE bytes holding the initialization vector.	uint8_t *

Return values

Return value	Description
FN_FILE *	A pointer to the handle of the opened file.
NULL	If the pointer is null, the file could not be opened.

Example

```

/* Encryption key - should be secret to whoever has access to this media */
uint8_t my_key[] = { 0x32, 0xaa, 0xf3, 0x82, 0x5b, 0x80, 0x07, 0x6a, 0x8e, 0xdb,
0x9b, 0xd6, 0x36, 0x9c, 0x47, 0xfb };

/* Initialization vector - should be unique for each file but does not need to be
secret */

uint8_t test1_iv[] = { 0x71, 0x34, 0x1f, 0xd0, 0x19, 0x3b, 0xe8, 0x51, 0x4c, 0x38,
0x9c, 0x13, 0x77, 0x8c, 0x95, 0x50 };

void myfunc( void )
{
    F_FILE *file;
    char c;
    file = f_open_enc( "test1.bin", /* File name */
                      "r", /* Open mode: only r, a, a+ supported */
                      &my_key[0], /* Encryption key */
                      &test1_iv[0]); /* Initialization vector for this file */

    if (!file)
    {
        printf( "File cannot be opened!" );
        return;
    }

    f_read( &c, 1, 1, file ); /* Read one byte */
    printf( "'%' is read from file", c );
    f_close( file );
}
    
```


f_open_nonsafe

Use this function to open a file.

The following opening modes are allowed:

Mode	Description
"r"	Open existing file for reading. The stream is positioned at the beginning of the file.
"r+"	Open existing file for reading and writing. The stream is positioned at the beginning of the file.
"w"	Truncate file to zero length or create file for writing. The stream is positioned at the beginning of the file.
"w+"	Open a file for reading and writing. The file is created if it does not exist; otherwise it is truncated. The stream is positioned at the beginning of the file.
"a"	Open for appending (writing to end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
"a+"	Open for reading and appending (writing to end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

Note the following:

- The same file can be opened multiple times in "r" mode.
- A file can only be opened once at a time in a mode which gives write access (that is, in "r+", "w", "w+", "a" or "a+" mode).
- The same file can be opened multiple times in "r" mode and at the same time once in one of the "r+", "a" or "a+" modes which give write access.
- If a file is opened in "w" or "w+" mode, a lock mechanism prevents it being opened in any other mode. This prevents opening of the file for reading and writing at the same time.

Note: There is no text mode. The system assumes that all files are in binary mode only.

Format

```
F_FILE * f_open_nonsafe (
    const char * filename,
    const char * mode )
```

Arguments

Argument	Description	Type
filename	The file to open.	char *
mode	The opening mode (see above).	char *

Return values

Return value	Description
F_FILE *	A pointer to the handle of the opened file.
NULL	If the pointer is null, the file could not be opened.

Example

```
void myfunc( void )
{
    F_FILE *file;
    char c;

    file = f_open_nonsafe( "myfile.bin", "r" );
    if (!file)
    {
        printf( "File cannot be opened!" );
        return;
    }

    f_read( &c, 1, 1, file ); /* Read one byte */
    printf( "'%' is read from file", c );
    f_close( file );
}
```

f_close

Use this function to close a previously opened file.

Format

```
int f_close ( F_FILE * filehandle )
```

Arguments

Argument	Description	Type
filehandle	The handle of the file.	F_FILE *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    F_FILE *file;
    char *string = "ABC";

    file = f_open( "myfile.bin", "w" );

    if (!file)
    {
        printf( "File cannot be opened!" );
        return;
    }

    f_write( string, 3, 1, file ); /* Write 3 bytes */
    if (!f_close( file ))
    {
        printf( "File stored" );
    }
    else
    {
        printf( "File close error!" );
    }
}
```

f_abortclose

Use this function to close a previously opened file, aborting all operations.

This restores a file's last valid state (flushed or closed state).

Format

```
int f_abortclose (
    F_MULTI *   fm,
    FN_FILE *   filehandle )
```

Arguments

Argument	Description	Type
fm	A multi-structure pointer.	F_MULTI *
filehandle	The handle of the file.	FN_FILE *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```

#if SAFEFAT
int fn_abortclose ( F_MULTI * fm, FN_FILE * filehandle )
{
    FN_FILEINT * f = _f_check_handle( filehandle );
    int rc = F_NO_ERROR;
    F_VOLUME * vi;
    if ( !f )
    {
        return fn_setlasterror( fm, F_ERR_NOTOPEN );
    }
    rc = _f_getvolume( fm, f->drivenum, &vi );
    if ( !rc )
    {
        /* Restore file's clusters */
        rc = _s_restoreclusters( vi, f->s_name );
        if ( rc )
        {
            /* Remove cf file, rc is already signal error */
            cf_remove( vi, f->s_name );
        }
        else
        {
            /* Remove cf file*/
            rc = cf_remove( vi, f->s_name );
        }
    }
    #ifndef USE_MALLOC
    if ( f->WrDataCache.pos )
    {
        psp_free( f->WrDataCache.pos );
        f->WrDataCache.pos = NULL;
    }
    #endif
    /* Remove sync afile connections */
    _f_removesyncafile( f, rc );

    /* Release file */
    f->mode = FN_FILE_CLOSE;

    return fn_setlasterror( fm, rc );
} /* fn_abortclose */

#endif /* if SAFEFAT */
    
```

f_flush

Use this function to flush an opened file to a storage medium. This is logically equivalent to performing a close and open on a file to ensure the data changed before the flush is committed to the medium.

Format

```
int f_flush ( F_FILE * filehandle )
```

Arguments

Argument	Description	Type
filehandle	The handle of the file.	F_FILE *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    F_FILE *file;
    char *string = "ABC";

    file = f_open( "myfile.bin", "w" );
    if (!file)
    {
        printf( "File cannot be opened!" );
        return;
    }
    f_write( string, 3, 1, file ); /* Write 3 bytes */

    f_flush( file ); /* Commit data written */
    :
    .
}
```

f_read

Use this function to read bytes from the current position in the specified file.

The file must be opened in "r", "r+", "w+", or "a+" mode. (See [f_open\(\)](#) for details of modes).

Format

```
long f_read (
    void *    buf,
    long      size,
    long      size_st,
    F_FILE *  filehandle )
```

Arguments

Argument	Description	Type
buf	The buffer to store data in.	void *
size	The size of the items to read.	long
size_st	The number of items to read.	long
filehandle	The handle of the file.	F_FILE *

Return values

Return value	Description
The number of items successfully read.	If this does not equal the number of items requested, call f_getlasterror() to determine the cause.

Example

```
int myreadfunc( char *filename, char *buffer, long bufsize )
{
    F_FILE *file = f_open( filename, "r" );
    long size = f_filelength( filename );

    if (!file)
    {
        printf( "%s cannot be opened!", filename );
        return 1;
    }

    if (f_read( buffer, 1, size, file )!= size)
    {
        printf( "Some items not read! Error:%d", f_getlasterror() );
    }
    f_close( file );
    return 0;
}
```


f_write

Use this function to write data into a file at the current position.

The file must be opened in "r+", "w", "w+", "a+", or "a" mode (see [f_open\(\)](#) for details of modes). The file pointer is moved forward by the number of bytes successfully written.

Note: Data is NOT permanently stored to the media until either an **f_flush()** or **f_close()** has been executed on the file.

Format

```
long f_write (
    const void *  buf,
    long         size,
    long         size_st,
    F_FILE *     filehandle )
```

Arguments

Argument	Description	Type
buf	A pointer to the data to write.	void *
size	The size of the items to write.	long
size_st	The number of items to write.	long
filehandle	The handle of the file.	F_FILE *

Return values

Return value	Description
The number of items successfully written.	If this does not equal the number of items requested, call f_getlasterror() to determine the cause.

Example

```
void myfunc( void )
{
    F_FILE *file;
    char *string = "ABC";

    file = f_open( "myfile.bin", "w" );
    if (!file)
    {
        printf( "File cannot be opened!" );
        return;
    }

    /* Write 3 bytes */
    if (f_write( string, 1, 3, file ) != 3)
    {
        printf( "Some items not written! Error:%d", f_getlasterror() );
    }
    f_close( file );
}
```

f_getc

Use this function to read a character from the current position in the specified open file.

Format

```
int f_getc ( F_FILE * filehandle )
```

Arguments

Argument	Description	Type
filehandle	The handle of the open file.	F_FILE *

Return values

Return value	Description
-1	Read failed.
value	The character read from the file.

Example

```
int myreadfunc( char *filename, char *buffer, long bufsize )
{
    F_FILE *file = f_open( filename, "r" );
    while (bufsize-->0)
    {
        int ch;
        if ((ch = f_getc( file )) == -1)
            break;
        *buffer++ = ch;
        bufsize--;
    }

    f_close( file );
    return 0;
}
```

f_putc

Use this function to write a character to the specified open file at the current file position. The current file position is incremented.

Format

```
int f_putc (
    char      ch,
    F_FILE *  filehandle )
```

Arguments

Argument	Description	Type
ch	The character to write.	char
filehandle	The handle of the open file.	F_FILE *

Return values

Return value	Description
-1	Write failed.
value	The successfully written character.

Example

```
void myfunc (char *filename, long num)
{
    F_FILE *file = f_open( filename, "w" );
    while (num-- > 0)
    {
        int ch = 'A';
        if (ch != (f_putc( ch )))
        {
            printf( "f_putc error!" );
            break;
        }
    }
    f_close( file );
    return 0;
}
```

f_eof

Use this function to check whether the current position in the specified open file is the end of file (EOF).

Format

```
int f_eof ( F_FILE * filehandle )
```

Arguments

Argument	Description	Type
filehandle	The handle of the open file.	F_FILE *

Return values

Return value	Description
F_NO_ERROR	Not at the end of the file.
Else	End of file or an error. See Error Codes .

Example

```
int myreadfunc( char *filename, char *buffer, long bufsize )
{
    F_FILE *file = f_open( filename, "r" );

    while (!f_eof())
    {
        if (!bufsize) break;
        bufsize--;
        f_read( buffer++, 1, 1, file );
    }
    f_close( file );

    return 0;
}
```

f_seteof

Use this function to move the end of file (EOF) to the current file pointer.

All data after the new EOF position are lost.

Format

```
int f_seteof ( F_FILE * filehandle )
```

Arguments

Argument	Description	Type
filehandle	The handle of the open file.	F_FILE *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
int mytruncatefunc( char *filename, int position )
{
    F_FILE *file = f_open( filename, "r+" );

    f_seek( file, position, SEEK_SET );

    if ( f_seteof( file ) )
        printf( "Truncate failed!\n" );

    f_close( file );
    return 0;
}
```

f_tell

Use this function to get the current read/write position in the specified open file.

Format

```
long f_tell ( F_FILE * filehandle )
```

Arguments

Argument	Description	Type
filehandle	The handle of the open file.	F_FILE *

Return values

Return value	Description
filepos	The current read or write file position.

Example

```
int myreadfunc( char *filename, char *buffer, long bufsize )
{
    F_FILE *file = f_open( filename, "r" );
    printf( "Current position %d", f_tell( file ) ); /* Position 0 */
    f_read( buffer, 1, 1, file ); /* Read one byte */
    printf( "Current position %d", f_tell( file ) ); /* Position 1 */
    f_read( buffer, 1, 1, file ); /* Read one byte */
    printf( "Current position %d", f_tell( file ) ); /* Position 2 */

    f_close( file );
    return 0;
}
```

f_seek

Use this function to move the stream position in the specified open file.

An optional additional non-standard flag is provided: `F_SEEK_NOWRITE`. This can be set if you want to seek past the end of file and do not want to fill the file with zeroes. This is useful for creating large files quickly, without the normal overhead of having to write to every sector.

Note: If `F_SEEK_NOWRITE` is used the contents of the extended area are undefined.

The offset position is relative to *whence*.

Format

```
long f_seek (
    F_FILE *   filehandle,
    long       offset,
    long       whence )
```

Arguments

Argument	Description	Type
filehandle	The handle of the open file.	F_FILE *
offset	The byte position relative to <i>whence</i> .	long
whence	Where to calculate the offset from, one of the following: <ul style="list-style-type: none"> • <code>F_SEEK_CUR</code> - current position of the file pointer. • <code>F_SEEK_END</code> - end of file. • <code>F_SEEK_SET</code> - start of file. 	long

Return values

Return value	Description
<code>F_NO_ERROR</code>	Successful execution.
Else	See Error Codes .

Example

```
int myreadfunc( char *filename, char *buffer, long bufsize )
{
    F_FILE *file = f_open( filename, "r" );

    f_read( buffer, 1, 1, file ); /* Read the first byte */
    f_seek( file, 0, SEEK_SET );
    f_read( buffer, 1, 1, file ); /* Read the same byte */
    f_seek( file, -1, SEEK_END );
    f_read( buffer, 1, 1, file ); /* Read the last byte */
    f_close( file );
    return 0;
}
```

f_rewind

Use this function to set the file position in the specified open file to the start of the file.

Format

```
int f_rewind ( F_FILE * filehandle )
```

Arguments

Argument	Description	Type
filehandle	The handle of the open file.	F_FILE *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    char buffer[4];
    char buffer2[4];

    F_FILE *file = f_open( "myfile.bin", "r" );

    if (file)
    {
        f_read( buffer, 4, 1, file );

        /* Rewind file pointer */
        f_rewind( file );

        /* Read from the beginning */
        f_read( buffer2, 4, 1, file );

        f_close( file );
    }
    return 0;
}
```

f_truncate

Use this function to open a file for writing and truncate it to the specified length.

If the length is greater than the length of the existing file, the file is padded with zeroes to the truncated length.

Format

```
F_FILE * f_truncate (
    const char * filename,
    unsigned long length )
```

Arguments

Argument	Description	Type
filename	The file to open.	char *
length	The new length of the file.	unsigned long

Return values

Return value	Description
F_FILE *	A pointer to the handle of the opened file.
NULL	If the pointer is null, the file could not be opened.

Example

```
int mytruncatefunc( char *filename, unsigned long length )
{
    F_FILE *file = f_truncate( filename, length );

    if (!file)
    {
        printf( "File opening error!" );
    }
    else
    {
        printf( "File %s truncated to %d bytes", filename, length );
        f_close( file );
    }
    return 0;
}
```

f_ftruncate

Use this function to truncate a file which is open for writing to a specified length.

If *length* is greater than the length of the existing file, the file is padded with zeroes to the new length.

Format

```
int f_ftruncate (
    F_FILE *      filehandle,
    unsigned long length )
```

Arguments

Argument	Description	Type
filehandle	The file handle of the open file.	F_FILE *
length	The new length of the file.	unsigned long

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
int mytruncatefunc( F_FILE *file, unsigned long length )
{
    int ret = f_ftruncate( filename, length );

    if (ret)
    {
        printf( "Error:%d\n", ret );
    }
    else
    {
        printf( "File is truncated to %d bytes", length );
    }

    return ret;
}
```

File Management

The functions are the following:

Function	Description
f_delete()	Deletes a file.
f_deletecontent()	Deletes a file and also its contents. That is, all the content is set to 0xFF.
f_findfirst()	Finds the first file or subdirectory in a specified directory.
f_findnext()	Finds the next file or subdirectory in a specified directory after a previous call to f_findfirst() or f_findnext() .
f_move()	Moves a file or directory. The original file or directory is lost.
f_rename()	Renames a file or directory.
f_getattr()	Gets the attributes of a file.
f_setattr()	Sets the attributes of a file.
f_gettimedate()	Gets time and date information from a file or directory.
f_settimedate()	Sets time and date information for a file or directory.
f_fstat()	Gets information about a file by using the file handle.
f_stat()	Gets information about a file or directory.
f_filelength()	Gets the length of a file.
f_dir_walk()	Finds all files and sub-directories.
f_disk_usage()	Gets disk usage information for a file or directory.

f_delete

Use this function to delete a file.

Note: A read-only or open file cannot be deleted.

Format

```
int f_delete ( const char * filename )
```

Arguments

Argument	Description	Type
filename	A null-terminated string with the name of the file to delete, with or without its path.	char *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    .
    .
    f_delete( "oldfile.txt" );
    f_delete( "A:/subdir/oldfile.txt" );
    .
    .
}
```

f_deletecontent

Use this function to delete a file and also delete its contents. This sets all the content to 0xFF.

This function is only available if [F_DELETE_CONTENT](#) is defined in **config_fat.h**.

Note: A read-only or open file cannot be deleted.

Format

```
int f_deletecontent ( const char * filename )
```

Arguments

Argument	Description	Type
filename	The name of the file to delete, with or without its path.	char *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )  
{  
    .  
    .  
    f_deletecontent( "oldfile.txt" );  
    .  
    .  
}
```

f_findfirst

Use this function to find the first file or subdirectory in a specified directory.

First call **f_findfirst()** and then, if the file is found, get the next file with **f_findnext()**.

The safe log directory (\$\$SAFE\$\$) is ignored if a volume is mounted in safe mode. If a volume was mounted previously in safe mode, but is later in non-safe mode, the safe log directory will be visible to the user.

Note: If this is called with "*" and it is not the root directory, then:

- the first entry found is ".", the current directory.
- the second entry found is "..", the parent directory.

Format

```
int f_findfirst (
    const char * filename,
    F_FIND * find )
```

Arguments

Argument	Description	Type
filename	The name of the file or subdirectory to find.	char *
find	Where to store the file information.	F_FIND *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void mydir( void )
{
    F_FIND find;
    if (!f_findfirst ( "A:/subdir/file*.*", &find ) )
    {
        do
        {
            printf( "filename:%s", find.filename );
            if (find.attr&F_ATTR_DIR)
            {
                printf( " directory\n" );
            }
            else
            {
                printf( " size %d\n", find.filesize );
            }
        } while (!f_findnext( &find ));
    }
}
```

f_findnext

Use this function to find the next file or subdirectory in a specified directory after a previous call to **f_findfirst()** or **f_findnext()**.

First call **f_findfirst()** and then, if a file is found, get the rest of the matching files by repeated calls to **f_findnext()**. Files with the system attribute set are ignored.

The safe log directory (\$\$SAFE\$\$) is ignored if a volume is mounted in safe mode. If a volume was mounted previously in safe mode, but is later in non-safe mode, the safe log directory will be visible to the user.

Note: If this is called with "*" and it is not the root directory, then:

- the first file found is ".", the current directory.
- the second file found is "..", the parent directory.

Format

```
int f_findnext ( F_FIND * find )
```

Arguments

Argument	Description	Type
find	File information, created by calling f_findfirst() .	F_FIND *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void mydir( void )
{
    F_FIND find;
    if (!f_findfirst( "A:/subdir/file*.*", &find ))
    {
        do
        {
            printf( "filename:%s", find.filename );
            if (find.attr&F_ATTR_DIR)
            {
                printf( " directory\n" );
            }
            else
            {
                printf( " size %d\n", find.filesize );
            }
        } while (!f_findnext( &find ));
    }
}
```

f_move

Use this function to move a file or directory. The original file or directory is lost.

The source and target must be in the same volume. A file can be moved only if it is not open. A directory can be moved only if there are no open files in it.

A file or directory can be moved, irrespective of its attribute settings. The attribute settings are moved with it.

Format

```
int f_move (
    const char * filename,
    const char * newname )
```

Arguments

Argument	Description	Type
filename	The file or directory name, with or without its path.	char *
newname	The new name of the file or directory, with or without the path.	char *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    .
    .
    f_move( "oldfile.txt", "newfile.txt" );
    f_move( "A:/subdir/oldfile.txt", "A:/newdir/oldfile.txt" );
    .
    .
}
```

f_rename

Use this function to rename a file or directory.

Note: The file or directory must not be read-only. If it is a file, it must not be open.

Format

```
int f_rename (
    const char * filename,
    const char * newname )
```

Arguments

Argument	Description	Type
filename	The file or directory name, with or without its path.	char *
newname	The new name of the file or directory.	char *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    .
    .
    f_rename( "oldfile.txt", "newfile.txt" );
    f_rename( "A:/subdir/oldfile.txt", "newfile.txt" );
    .
    .
}
```

f_getattr

Use this function to get the attributes of a specified file.

Possible [file attribute settings](#) (F_ATTR_XXX) are defined by the FAT file system.

Format

```
int f_getattr (
    const char * filename,
    unsigned char * attr )
```

Arguments

Argument	Description	Type
filename	The name of the file.	char *
attr	Where to write the attributes.	unsigned char *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    unsigned char attr;

    /* Find whether myfile.txt is read-only */
    if (!f_getattr( "myfile.txt", &attr )
    {
        if (attr & F_ATTR_READONLY)
        {
            printf( "myfile.txt is read-only" );
        }
        else
        {
            printf( "myfile.txt is writable" );
        }
    }
    else
    {
        printf( "File not found!" );
    }
}
```

f_setattr

Use this function to set the attributes of a file.

Possible [file attribute settings](#) (F_ATTR_XXX) are defined by the FAT file system.

Note: The directory and volume attributes cannot be set by using this function.

Format

```
int f_setattr (
    const char * filename,
    unsigned char attr )
```

Arguments

Argument	Description	Type
filename	The name of the file.	char *
attr	The new attribute setting.	unsigned char

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    /* Make myfile.txt read-only and hidden */
    f_setattr( "myfile.txt", F_ATTR_READONLY | F_ATTR_HIDDEN );
}
```


f_gettimedate

Use this function to get time and date information from a file or directory.

This field is automatically set by the system when a file or directory is created, and when a file is closed.

Date and Time Formats

The date and time fields are two 16 bit fields associated with each file/directory.

The [required format for the date](#) for PC compatibility is a short integer 'd' (16 bit), such that:

Argument	Valid values	Format
Day	0-31	(d & 0x001F)
Month	1-12	((d & 0x01E0) >> 5)
Years since 1980	0-119	((d & 0xFE00) >> 9)

The [required format for the time](#) for PC compatibility is a short integer 't' (16 bit), such that:

Argument	Valid values	Format
Two second increments	0-30	(t & 0x001F)
Minute	0-59	((t & 0x07E0) >> 5)
Hour	0-23	((t & 0xF800) >> 11)

Format

```
int f_gettimedate (
    const char *    filename,
    unsigned short * pctime,
    unsigned short * pdate )
```

Arguments

Argument	Description	Type
filename	The name of the file or directory.	char *
pctime	Where to store the creation time.	unsigned short *
pdate	Where to store the creation date.	unsigned short *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    unsigned short t, d, sec, min, hour;
    unsigned short day, month, year;

    if (!f_gettimedate( "subfolder", &t, &d ))
    {
        sec = (t & F_CTIME_SEC_MASK);
        min = ((t & F_CTIME_MIN_MASK) >> F_CTIME_MIN_SHIFT);
        hour = ((t & F_CTIME_HOUR_MASK) >> F_CTIME_HOUR_SHIFT);
        day = (d & F_CDATE_DAY_MASK);
        month = ((d & F_CDATE_MONTH_MASK) >> F_CDATE_MONTH_SHIFT);
        year = 1980 + ((d & F_CDATE_YEAR_MASK) >> F_CDATE_YEAR_SHIFT);

        printf( "Time: %d:%d:%d", hour, min, sec );
        printf( "Date: %d.%d.%d", year, month, day );
    }
    else
    {
        printf( "File time/date cannot be retrieved!" );
    }
}
```

f_settimedate

Use this function to set the time and date on a file or on a directory.

Date and Time Formats

The date and time fields are two 16 bit fields associated with each file/directory.

The [required format for the date](#) for PC compatibility is a short integer 'd' (16 bit), such that:

Argument	Valid values	Format
Day	0-31	(d & 0x001F)
Month	1-12	((d & 0x01E0) >> 5)
Years since 1980	0-119	((d & 0xFE00) >> 9)

The [required format for the time](#) for PC compatibility is a short integer 't' (16 bit), such that:

Argument	Valid values	Format
Two second increments	0-30	(t & 0x001F)
Minute	0-59	((t & 0x07E0) >> 5)
Hour	0-23	((t & 0xF800) >> 11)

Format

```
int f_settimedate (
    const char *   filename,
    unsigned short ctime,
    unsigned short cdate )
```

Arguments

Argument	Description	Type
filename	The name of the file or directory.	char *
ctime	The creation time of the file or directory.	unsigned short
cdate	The creation date of the file or directory.	unsigned short

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    unsigned short ctime, cdate;

    ctime = (15 << 11) + (30 << 5) + (22 >> 1); /* 15:30:22 */

    cdate = ((2002 - 1980) << 9) + (11 << 5) + (3); /* 2002.11.03. */

    f_mkdir( "subfolder" ); /* Create directory */
    f_settimedate( "subfolder", ctime, cdate );
}
```

f_stat

Use this function to get information about a file or directory.

This function retrieves information by filling the [F_STAT](#) structure passed to it. It sets the file/directory size, creation time/date, last access date, modified time/date, and the drive number where the file or directory is located.

Note: For files, this function can also return the current size of the opened file when the configuration option [F_FINDOPENFILESIZE](#) is set to 1, allowing it to search through all open file descriptors for its modified size. If this feature is disabled then file size is always 0 for opened files.

Format

```
int f_stat (
    const char * filename,
    F_STAT * stat )
```

Arguments

Argument	Description	Type
filename	The name of the file or directory.	char *
stat	A pointer to the F_STAT structure to be filled.	F_STAT *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    F_STAT stat;
    if ( f_stat( "myfile.txt", &stat ) )
    {
        printf( "Error!" );
        return;
    }
    printf( "filesize:%d", stat.filesize );
}
```

f_fstat

Use this function to get information about a file by using its file handle.

This function retrieves information by filling the [F_STAT](#) structure passed to it. It sets the file size, creation time/date, last access date, modified time/date, and the drive number where the file is located.

Format

```
int f_fstat (  
    F_FILE *    p_filehandle,  
    F_STAT *    p_stat )
```

Arguments

Argument	Description	Type
p_filehandle	The file handle.	F_FILE *
p_stat	A pointer to the F_STAT structure to be filled.	F_STAT *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc ( void )
{
    F_FILE *file;
    F_STAT stat;
    int ret;

    file = f_open( filename, "r" );

    if ( file != NULL )
    {
        ret = f_fstat( file, &stat );

        if ( ret == F_NO_ERROR )
        {
            printf( "filesize:%d\r\n", stat.filesize );
        }
        else
        {
            printf( "f_fstat error: %d.\r\n", ret );
        }
        f_close( file );
    }
    else
    {
        printf( "%s Cannot be opened!\r\n", filename );
    }
}
```

f_filelength

Use this function to get the length of a file.

Note: This function can also return with the opened file's size when **f_findopenseize()** is allowed to search for it. If **f_findopenseize()** always returns with zero, then this feature is disabled.

Format

```
long f_filelength ( const char * filename )
```

Arguments

Argument	Description	Type
filename	The file name, with or without the path.	char *

Return values

Return value	Description
filelength	The length of the file.
-1	The requested file does not exist or has an error; check the last error.

Example

```
int myreadfunc( char *filename, char *buffer, long bufsize )
{
    F_FILE *file = f_open( filename, "r" );
    long size = f_filelength( filename );

    if (!file)
    {
        printf( "%s Cannot be opened!", filename );
        return 1;
    }
    if (size > bufsize)
    {
        printf( "Not enough memory!" );
        return 2;
    }

    f_read( buffer, size, 1, file );
    f_close( file );

    return 0;
}
```


f_dir_walk

Use this function to find all files and sub-directories.

Note: This is only available if configuration option [F_DIR_WALK_SUPPORT](#) is set.

Format

```
int f_dir_walk (
    const char *      filename,
    F_FIND *         p_find,
    t_dir_walk_cb_fn * p_callback,
    void *           p_user_data )
```

Arguments

Argument	Description	Type
filename	The filename or search mask. Examples include "*.*", "foo.bin", "a:\\foo*.txt", or "*".	char *
p_find	A pointer to the structure used by f_findfirst()/f_findnext() .	F_FIND *
p_callback	A pointer to the callback function which is called for every directory and file.	t_dir_walk_cb_fn *
p_user_data	A pointer to the returned list of files and sub-directories.	void *

Return values

Return value	Description
0	Successful execution.
Else	See Error Codes .

Example

The output from this is similar to the Windows command prompt's **tree /f** command.

```

/* Callback function for f_dir_walk */
int tree_dir_walk_fn ( F_FIND * p_find, uint16_t depth, void * p_user_data )
{
    uint8_t idx;
    uint8_t b_is_only_dots; /* Will be TRUE if filename is "." or ".." */
    b_is_only_dots = FALSE;

    if ( p_find->filename[0] == '.' )
    {
        if ( p_find->filename[1] == 0 )
        {
            b_is_only_dots = TRUE;
        }
        else if ( (p_find->filename[1] == '.' ) && (p_find->filename[2] == 0 ) )
        {
            b_is_only_dots = TRUE;
        }
    }

    if ( b_is_only_dots == FALSE )
    {
        for ( idx = 1 ; idx < depth ; idx++ )
        {
            printf( "    " );
        }
        if ( depth > 0 )
        {
            printf( " +- " );
        }

        if ( p_find->attr & F_ATTR_DIR )
        {
            printf( "[%s]\r\n", p_find->filename );
        }
        else
        {
            printf( "%s %u\r\n", p_find->filename, (int)p_find->filesize );
        }
    }
    return 0;
}

void print_tree ( void )
{
    F_FIND find;
    f_dir_walk( " *.*", &find, &tree_dir_walk_fn, NULL );
}
    
```

f_disk_usage

Use this function to get disk usage information for a file or directory. Disk usage is always a multiple of the volume's cluster size.

Note: This is only available if configuration option [F_DISK_USAGE_SUPPORT](#) is set.

Format

```
int f_disk_usage (
    const char * filename,
    F_FIND * p_find,
    uint32_t * p_du_bytes_high,
    uint32_t * p_du_bytes_low )
```

Arguments

Argument	Description	Type
filename	The file or directory name or a search mask. Examples include "*.*", "foo.bin", "a:\\foo*.txt", and "*".	char *
p_find	A pointer to the structure used by f_findfirst()/f_findnext() .	F_FIND *
p_du_bytes_high	A pointer to the number of used bytes (high 32 bits).	uint32_t *
p_du_bytes_low	A pointer to the number of used bytes (low 32 bits).	uint32_t *

Return values

Return value	Description
0	Successful execution.
Else	See Error Codes .

Example

```
void disk_usage( void )
{
    int      rc;
    uint64_t disk_usage_bytes64;
    uint32_t disk_usage_bytes_high;
    uint32_t disk_usage_bytes;
    F_FIND   find;

    rc = f_disk_usage( ".*.*", &find, &disk_usage_bytes_high, &disk_usage_bytes );
    if ( rc == F_NO_ERROR )
    {
        disk_usage_bytes64 = (uint64_t)disk_usage_bytes_high << 32;
        disk_usage_bytes64 |= disk_usage_bytes;
        printf( "Disk usage: %llu bytes\r\n", disk_usage_bytes64 );
        printf( "          %llu KiB\r\n", disk_usage_bytes64 / 1024 );
    }
    else
    {
        printf("Error: %i\r\n", rc);
    }
}
```

7.3. File System Unicode API

This section describes all the API Unicode functions available with the file system. It is split into functions for directory management, file access, file management, and Unicode translation.

Unicode-Specific File System Functions

To enable Unicode API calls in the file system, set the **#define HCC_UNICODE** definition in the `/src/config/config_fat.h` file. This makes the functions in this section, as well as their standard API equivalents, available for use.

All functions are exactly the same as their standard API counterparts, except that all character string parameters are changed to “wide character” (wchar) strings.

Character and wide character definition with W_CHAR

W_CHAR is defined as **char** if Unicode is disabled and as **wchar** if it is enabled. Therefore W_CHAR is used in structures where the element could be used in either type of system.

Examples

Unicode string literals are written as `u"Unicode string"` in the examples but, depending on the compiler version and applied switches, it might be possible to create Unicode string literals like this: `L"Unicode string"`.

Please refer to the compiler’s manual for details of how to create UTF-16 string literals.

Unicode Directory Management

The functions are the following:

Function	Description
f_wmkdir()	Creates a new directory with a Unicode 16 name.
f_wchdir()	Changes the current working directory.
f_wrmdir()	Removes a directory that has a Unicode 16 name.
f_wgetcwd()	Gets the current working directory.
f_wgetdcwd()	Gets the current working directory on the selected drive.

f_wmkdir

Use this function to create a new directory with a Unicode 16 name.

Format

```
int f_wmkdir ( const W_CHAR * dirname )
```

Arguments

Argument	Description	Type
dirname	The Unicode 16 name of the directory to create.	W_CHAR *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    .
    .
    f_wmkdir( u"subfolder" );    /* Create directories */
    f_wmkdir( u"subfolder/sub1" );
    f_wmkdir( u"subfolder/sub2" );
    f_wmkdir( u"a:/subfolder/sub3" );
    .
    .
}
```

f_wchdir

Use this function to change the current working directory (that has a Unicode 16 name).

Format

```
int f_wchdir ( const W_CHAR * dirname )
```

Arguments

Argument	Description	Type
dirname	The Unicode 16 name of the directory to change to.	W_CHAR *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    .
    .
    f_wmkdir( u"subfolder" );
    f_wchdir( u"subfolder" );      /* Change directory */
    f_wmkdir( u"sub2" );
    f_wchdir( u".." );             /* Go upward */
    f_wchdir( u"subfolder/sub2" ); /* Go into directory sub2 */
    .
    .
}
```


f_wrmdir

Use this function to remove a directory that has a Unicode 16 name.

Note: The directory must be empty. Otherwise, an error code is returned and it is not removed.

Format

```
int f_wrmdir ( const W_CHAR * dirname )
```

Arguments

Argument	Description	Type
dirname	The Unicode 16 name of the directory to remove.	W_CHAR *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    .
    .
    f_wmkdir( u"subfolder" ); /* Create directories */
    f_wmkdir( u"subfolder/sub1" );
    .
    . /* Do some work */
    .
    f_wrmdir( u"subfolder/sub1" ); /* Remove directories */
    f_wrmdir( u"subfolder" );
    .
    .
}
```

f_wgetcwd

Use this function to get the current working directory on the current drive.

Format

```
int f_wgetcwd (
    W_CHAR *   buffer,
    int        maxlen )
```

Arguments

Argument	Description	Type
buffer	Where to store the current working directory string.	W_CHAR *
maxlen	The length of the buffer.	int

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( int drivenum )
{
    W_CHAR buffer[F_MAXPATH];
    if (!f_wgetcwd( drivenum, buffer, F_MAXPATH ))
    {
        wprintf( u"Current directory is %s", buffer );
        wprintf( u"on drive %c", drivenum + 'A' );
    }
    else
    {
        wprintf( u"Drive error!" )
    }
}
```

f_wgetcwd

Use this function to get the current working directory on the selected drive.

Format

```
int f_wgetcwd (
    int    drivenum,
    W_CHAR * buffer,
    int    maxlen )
```

Arguments

Argument	Description	Type
drivenum	The drive number (0='A', 1='B', and so on.).	int
buffer	Where to store the current working directory string.	W_CHAR *
maxlen	The length of the buffer.	int

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( int drivenum )
{
    W_CHAR buffer[F_MAXPATH];
    if (!f_wgetcwd( drivenum, buffer, F_MAXPATH ))
    {
        wprintf( u"Current directory is %s", buffer );
        wprintf( u"on drive %c", drivenum + 'A' );
    }
    else
    {
        wprintf( u"Drive error!" )
    }
}
```

Unicode File Access

The functions are the following:

Function	Description
f_wopen()	Opens a file that has a Unicode 16 file name.
f_wopen_enc()	Opens an encrypted file that has a Unicode 16 file name.
f_wopen_nonsafe()	Opens a file with a Unicode file name without the journaling enabled.
f_wtruncate()	Opens a file with a Unicode file name for writing and truncates it to the specified length.

f_wopen

Use this function to open a file with a Unicode 16 filename. The following opening modes are allowed:

Modes	Description
"r"	Open an existing file for reading. The stream is positioned to the beginning of the file.
"r+"	Open an existing file for reading and writing. The stream is positioned to the beginning of the file.
"w"	Truncate file to zero length or create file for writing. The stream is positioned to the beginning of the file.
"w+"	Open for reading and writing. The file is created if it does not exist; otherwise it is truncated. The stream is positioned to the beginning of the file.
"a"	Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned to the end of the file.
"a+"	Open for reading and appending (writing at end of file). The file is created if it does not exist. The stream is positioned to the end of the file.

Note the following:

- The same file can be opened multiple times in "r" mode.
- A file can only be opened once at a time in a mode which gives write access (that is, in "r+", "w", "w+", "a" or "a+" mode).
- The same file can be opened multiple times in "r" mode and at the same time once in one of the "r+", "a" or "a+" modes which give write access.
- If a file is opened in "w" or "w+" mode, a lock mechanism prevents it being opened in any other mode. This prevents opening of the file for reading and writing at the same time.

Note: There is no text mode. The system assumes that all files are in binary mode only.

Format

```
F_FILE * f_wopen (
    const W_CHAR * filename,
    const char * mode )
```

Arguments

Argument	Description	Type
filename	The Unicode 16 name of the file.	W_CHAR *
mode	The opening mode (see above).	char *

Return values

Return value	Description
F_FILE *	A pointer to the handle of the opened file.
NULL	If the pointer is null, the file could not be opened.

Example

```
void myfunc( void )
{
    F_FILE *file;
    char c;

    file = f_wopen( u"myfile.bin", u"r" );

    if (!file)
    {
        wprintf( u"File cannot be opened!" );
        return;
    }
    f_read( &c, 1, 1, file ); /* Read one byte */
    wprintf( u"'%c' is read from file", c );
    f_close( file );
}
```

f_wopen_enc

Use this function to open an encrypted file that has a Unicode file name. When a file is opened this way:

- all writes to the file are AES-encrypted, using the cipher data provided when it was opened, before being committed to the media.
- all reads from the file are AES-decrypted, using the cipher data provided when it was opened, before being passed to the user.

The following opening modes are allowed:

Mode	Description
"r"	Open existing file for reading. The stream is positioned at the beginning of the file.
"a"	Open for appending (writing to end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
"a+"	Open for reading and appending (writing to end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

Note the following:

- This function only applies if [FAT_ENCRYPTION](#) is set.
- The same file can be opened multiple times in "r" mode.
- A file can only be opened once at a time in a mode that gives write access (that is, in "a" or "a+" mode).
- The same file can be opened multiple times in "r" mode and at the same time once in either "a" or "a+" mode.

Note: There is no text mode. The system assumes that all files are in binary mode only.

Format

```
FN_FILE * f_wopen_enc (
    const wchar *    p_filename,
    const wchar *    p_mode,
    const uint8_t *  p_key,
    const uint8_t *  p_init_vect )
```

Arguments

Argument	Description	Type
p_filename	A pointer to the file to open.	char *
p_mode	A pointer to the opening mode: r, a, or a+ (see above).	char *
p_key	A pointer to an array of FAT_ENC_BLOCK_SIZE bytes holding the encryption key.	uint8_t *
p_init_vect	A pointer to an array of FAT_ENC_BLOCK_SIZE bytes holding the initialization vector.	uint8_t *

Return values

Return value	Description
FN_FILE *	A pointer to the handle of the opened file.
NULL	If the pointer is null, the file could not be opened.

Example

```

/* Encryption key - should be secret to whoever has access to this media */
uint8_t my_key[] = { 0x32, 0xaa, 0xf3, 0x82, 0x5b, 0x80, 0x07, 0x6a, 0x8e, 0xdb,
0x9b, 0xd6, 0x36, 0x9c, 0x47, 0xfb };

/* Initialization vector - should be unique for each file but does not need to be
secret */

uint8_t test1_iv[] = { 0x71, 0x34, 0x1f, 0xd0, 0x19, 0x3b, 0xe8, 0x51, 0x4c, 0x38,
0x9c, 0x13, 0x77, 0x8c, 0x95, 0x50 };

void myfunc( void )
{
    F_FILE *file;
    char c;
    file = f_wopen_enc( u"test1.bin", /* File name */
                       u"r", /* Open mode: only r, a, a+ supported */
                       &my_key[0], /* Encryption key */
                       &test1_iv[0]); /* Initialization vector for this file */

    if (!file)
    {
        wprintf( u"File cannot be opened!" );
        return;
    }

    f_read( &c, 1, 1, file ); /* Read one byte */
    wprintf( u"'%' is read from file", c );
    f_close( file );
}
    
```


f_wopen_nonsafe

Use this function to open a file without the journaling enabled. This means that if the system is reset unexpectedly the open file could be left in an uncertain state. Typically the length may not be consistent with the amount of data written.

This function may be used to improve performance when a file with less sensitive data is being written.

The following opening modes are allowed:

Modes	Description
"r"	Open an existing file for reading. The stream is positioned at the beginning of the file.
"r+"	Open an existing file for reading and writing. The stream is positioned at the beginning of the file.
"w"	Truncate file to zero length or create file for writing. The stream is positioned at the beginning of the file.
"w+"	Open for reading and writing. The file is created if it does not exist; otherwise it is truncated. The stream is positioned at the beginning of the file.
"a"	Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
"a+"	Open for reading and appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

Note the following:

- The same file can be opened multiple times in "r" mode.
- A file can only be opened once at a time in a mode which gives write access (that is, in "r+", "w", "w+", "a" or "a+" mode).
- The same file can be opened multiple times in "r" mode and at the same time once in one of the "r+", "a" or "a+" modes which give write access.
- If a file is opened in "w" or "w+" mode, a lock mechanism prevents it being opened in any other mode. This prevents opening of the file for reading and writing at the same time.

Note: There is no text mode. The system assumes that all files are accessed in binary mode only.

Format

```
F_FILE * f_wopen_nonsafe (
    const wchar * filename,
    const wchar * mode )
```

Arguments

Argument	Description	Type
filename	The Unicode 16 name of the file.	wchar *
mode	The opening mode (see above).	wchar *

Return values

Return value	Description
F_FILE *	A pointer to the handle of the opened file.
NULL	If the pointer is null, the file could not be opened.

Example

```
void myfunc( void )
{
    F_FILE *file;
    char c;

    file = f_wopen_nonsafe( u"myfile.bin", u"r" );

    if (!file)
    {
        wprintf( u"File cannot be opened!" );
        return;
    }
    f_read( &c, 1, 1, file ); /* Read one byte */
    wprintf( u"'%' is read from file", c );

    f_close( file );
}
```

f_wtruncate

Use this function to open an existing file for writing and truncate it to the specified length.

If the length is greater than the length of the existing file, the file is padded with zeroes to the truncated length.

Format

```
F_FILE * f_wtruncate (
    const W_CHAR * filename,
    unsigned long length )
```

Arguments

Argument	Description	Type
filename	The name of the file.	W_CHAR *
length	The new length of the file.	unsigned long

Return values

Return value	Description
F_FILE *	A pointer to the handle of the opened file.
NULL	If the pointer is null, the file could not be opened.

Example

```
int mywtruncatefunc( W_CHAR *filename, unsigned long length )
{
    F_FILE *file = f_wtruncate( filename, length );

    if (!file)
    {
        wprintf( u"File not found!" );
    }
    else
    {
        wprintf( u"File %s truncated to %d bytes", filename, length );
        f_close( file );
    }
    return 0;
}
```

Unicode File Management

The functions are the following:

Function	Description
f_wdelete()	Deletes a file that has a Unicode 16 name.
f_wdeletecontent()	Deletes a Unicode 16 file and also its contents. This sets all the content to 0xFF.
f_wfindfirst()	Finds the first Unicode 16 file or subdirectory in a specified directory.
f_wfindnext()	Finds the next Unicode 16 file or subdirectory in a specified directory after a previous call to f_wfindfirst() or f_wfindnext() .
f_wmove()	Moves a Unicode 16 file or directory. The original file or directory is lost.
f_wrename()	Renames a Unicode 16 file or directory.
f_wgetattr()	Gets the attributes of a specified Unicode 16 file.
f_wsetattr()	Sets the attributes of a specified Unicode 16 file.
f_wgettimedate()	Gets time and date information from a Unicode 16 file or directory.
f_wsettimedate()	Sets time and date information for a Unicode 16 file or directory.
f_wstat()	Gets information about a Unicode 16 file or directory.
f_wfilelength()	Gets the length of a Unicode 16 file.
f_wdir_walk()	Finds all files and sub-directories.

f_wdelete

Use this function to delete a file with a Unicode 16 name.

Format

```
int f_wdelete ( const W_CHAR * filename )
```

Arguments

Argument	Description	Type
filename	The Unicode 16 name of the file, with or without the path.	W_CHAR *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )  
{  
    .  
    .  
    f_wdelete( u"oldfile.txt" );  
    f_wdelete( u"A:/subdir/oldfile.txt" );  
    .  
    .  
}
```

f_wdeletecontent

Use this function to delete a file with a Unicode 16 name and also its contents. This sets all the content to 0xFF.

Note: This function is available only if [F_DELETE_CONTENT](#) is defined in **config_fat.h**. A read-only or open file cannot be deleted.

Format

```
int f_wdeletecontent ( const wchar * filename )
```

Arguments

Argument	Description	Type
filename	The Unicode 16 name of the file, with or without the path.	wchar *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )  
{  
    .  
    .  
    f_wdeletecontent( u"oldfile.txt" );  
    .  
    .  
}
```

f_wfindfirst

Use this function to find the first Unicode 16 file or subdirectory in the specified directory.

First call **f_wfindfirst()** then, if a file is found, get the next file with **f_wfindnext()**.

Format

```
int f_wfindfirst (
    const W_CHAR *   filename,
    F_WFIND *        find )
```

Arguments

Argument	Description	Type
filename	The Unicode 16 name of the file or subdirectory to find.	W_CHAR *
find	Where to store the file information.	F_WFIND *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void mydir( void )
{
    F_WFIND find;
    if (!f_wfindfirst( u"A:/subdir/file*.*", &find ))
    {
        do
        {
            wprintf( u"filename:%s", find.filename );
            if (find.attr&F_ATTR_DIR)
            {
                wprintf( u" directory\n" );
            }
            else
            {
                wprintf( u" size %d\n", find.len );
            }
        } while (!f_wfindnext( &find ));
    }
}
```

f_wfindnext

Use this function to find the next Unicode 16 file or subdirectory in a specified directory after a previous call to **f_wfindfirst()** or **f_wfindnext()**.

First call **f_wfindfirst()** then, if a file is found, get the rest of the matching files by repeated calls to **f_wfindnext()**.

Format

```
int f_wfindnext ( F_WFIND * find )
```

Arguments

Argument	Description	Type
find	A Find structure (created by calling f_wfindfirst()).	F_WFIND*

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void mydir( void )
{
    F_WFIND find;
    if (!f_wfindfirst( u"A:/subdir/file*.*", &find ))
    {
        do
        {
            wprintf( u"filename:%s", find.filename );
            if (find.attr&F_ATTR_DIR)
            {
                wprintf( u" directory\n" );
            }
            else
            {
                wprintf( u" size %d\n", find.len );
            }
        } while (!f_wfindnext( &find ));
    }
}
```


f_wmove

Use this function to move a file or directory with a Unicode 16 name.

The source and target must be in the same volume. The original file or directory is lost.

Format

```
int f_wmove (
    const W_CHAR * filename,
    const W_CHAR * newname )
```

Arguments

Argument	Description	Type
filename	The Unicode 16 file or directory name, with or without the path.	W_CHAR *
newname	The new Unicode 16 name of the file or directory.	W_CHAR *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    .
    .
    f_wmove( u"oldfile.txt", u"newfile.txt" );
    f_wmove( u"A:/subdir/oldfile.txt", u"A:/newdir/oldfile.txt" );
    .
    .
}
```

f_wrename

Use this function to rename a file or directory with a Unicode 16 name.

Format

```
int f_wrename (
    const W_CHAR * filename,
    const W_CHAR * newname )
```

Arguments

Argument	Description	Type
filename	The Unicode 16 file or directory name, with or without the path.	W_CHAR *
newname	The new Unicode 16 name of the file or directory.	W_CHAR *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    .
    .
    f_wrename( u"oldfile.txt", u"newfile.txt" );
    f_wrename( u"A:/dir/oldfile.txt", u"newfile.txt" );
    .
    .
}
```

f_wgetattr

Use this function to get the attributes of a specified file with a Unicode 16 name. Possible file attribute settings are listed in the [F_ATTR_XXX](#) table.

Format

```
int f_wgetattr (
    const wchar *   filename,
    unsigned char * attr )
```

Arguments

Argument	Description	Type
filename	The Unicode 16 name of the file.	wchar *
attr	Where to write the attribute value.	unsigned char *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    unsigned char attr;
    /* Find whether myfile.txt is read-only */
    if (!f_wgetattr( u"myfile.txt", &attr )
        {
            if (attr & F_ATTR_READONLY)
            {
                wprintf( u"myfile.txt is read-only" );
            }
            else
            {
                wprintf( u"myfile.txt is writable" );
            }
        }
    else
    {
        wprintf( u"File not found!" );
    }
}
```

f_wsetattr

Use this function to set the attributes of a file with a Unicode 16 name.

Possible file attribute settings are listed in the [F_ATTR_XXX](#) table.

Note: The directory and volume attributes cannot be set by this function.

Format

```
int f_wsetattr (
    const wchar * filename,
    unsigned char attr )
```

Arguments

Argument	Description	Type
filename	The Unicode 16 name of the file.	wchar *
attr	The new attribute setting for that file.	unsigned char

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc ( void )
{
    /* Make myfile.txt read-only and hidden */
    f_wsetattr( u"myfile.txt", F_ATTR_READONLY | F_ATTR_HIDDEN );
}
```

f_wgettimedate

Use this function to get time and date information for a file or directory with a Unicode 16 name.

This field is automatically set by the system when a file or directory is created, and when a file is closed.

Date and Time Formats

The date and time fields are two 16 bit fields associated with each file/directory.

The [required format for the date](#) for PC compatibility is a short integer 'd' (16 bit), such that:

Argument	Valid values	Format
Day	0-31	(d & 0x001F)
Month	1-12	((d & 0x01E0) >> 5)
Years since 1980	0-119	((d & 0xFE00) >> 9)

The [required format for the time](#) for PC compatibility is a short integer 't' (16 bit), such that:

Argument	Valid values	Format
Two second increments	0-30	(t & 0x001F)
Minute	0-59	((t & 0x07E0) >> 5)
Hour	0-23	((t & 0xF800) >> 11)

Format

```
int f_wgettimedate (
    const W_CHAR *   filename,
    unsigned short * pctime,
    unsigned short * pdate )
```

Arguments

Argument	Description	Type
filename	The Unicode 16 name of the file or directory.	W_CHAR *
pctime	Where to store the time.	unsigned short *
pdate	Where to store the date.	unsigned short *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    unsigned short t,d;
    if (!f_wgettimedate( u"subfolder",&t,&d))
    {
        unsigned short sec = (t & 0x001F) << 1;
        unsigned short minute = ((t & 0x07E0) >> 5);
        unsigned short hour = ((t & 0x0F800) >> 11);
        unsigned short day = (d & 0x001F);
        unsigned short month = ((d & 0x01F0) >> 5);
        unsigned short year = 1980 + ((d & 0xFE00) >> 9);

        wprintf( u"Time: %d:%d:%d", hour, minute, sec );
        wprintf( u>Date: %d.%d.%d", year, month, day );
    }
    else
    {
        wprintf( u"File time cannot be retrieved!" );
    }
}
```

f_wsettimedate

Use this function to set the time and date on a file or on a directory with a Unicode 16 name.

Date and Time Formats

The date and time fields are two 16 bit fields associated with each file/directory.

The [required format for the date](#) for PC compatibility is a short integer 'd' (16 bit), such that:

Argument	Valid values	Format
Day	0-31	(d & 0x001F)
Month	1-12	((d & 0x01E0) >> 5)
Years since 1980	0-119	((d & 0xFE00) >> 9)

The [required format for the time](#) for PC compatibility is a short integer 't' (16 bit), such that:

Argument	Valid values	Format
Two second increments	0-30	(t & 0x001F)
Minute	0-59	((t & 0x07E0) >> 5)
Hour	0-23	((t & 0xF800) >> 11)

Format

```
int f_settimedate (
    const W_CHAR *   filename,
    unsigned short   ctime,
    unsigned short   cdate )
```

Arguments

Argument	Description	Type
filename	The Unicode 16 name of the file.	W_CHAR *
ctime	The creation time of the file or directory.	unsigned short
cdate	The creation date of the file or directory.	unsigned short

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    unsigned short ctime, cdate;

    ctime = (15 << 11) + (30 << 5) + (23 >> 1);    /* 15:30:22 */
    cdate = ((2002 - 1980) << 9) + (11 << 5) + (3); /* 2002.11.03. */

    f_wmkdir( u"subfolder" ); /* Create directory */
    f_wsettimedate( u"subfolder", ctime, cdate );
}
```


f_wstat

Use this function to get information about a Unicode 16 file or directory.

The function retrieves information by filling the [F_STAT](#) structure passed to it. It sets the file/directory size, creation time/date, last access date, modified time/date, and the drive number where the file or directory is located.

Note: For files, this function can also return the current size of the opened file when the configuration option [F_FINDOPENFILESIZE](#) is set to 1, allowing it to search through all open file descriptors for its modified size. If this feature is disabled then file size is always 0 for opened files.

Format

```
int f_wstat (
    const wchar *   filename,
    F_STAT *       stat )
```

Arguments

Argument	Description	Type
filename	The Unicode 16 file or directory name.	wchar *
stat	A pointer to the F_STAT structure to be filled.	F_STAT *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	See Error Codes .

Example

```
void myfunc( void )
{
    F_STAT stat;
    if ( f_wstat( u"myfile.txt", &stat ) )
    {
        wprintf( u"Error!" );
        return;
    }
    wprintf( u"filesize:%d", stat.filesize );
}
```

f_wfilelength

Use this function to obtain the length of a file with a Unicode 16 name.

Format

```
long f_wfilelength ( W_CHAR * filename )
```

Arguments

Argument	Description	Type
filename	The Unicode 16 file name, with or without the path.	W_CHAR *

Return values

Return value	Description
filelength	The length of the file.
-1	The requested file does not exist or has an error; check the last error.

Example

```
int myreadfunc( W_CHAR *filename, char *buffer, long bufsize )
{
    F_FILE *file = f_wopen( filename, u"r" );
    long size = f_wfilelength( filename );

    if (!file)
    {
        wprintf( u"%s Cannot be opened!", filename );
        return 1;
    }
    if (size > bufsize)
    {
        wprintf( u"Not enough memory!" );
        return 2;
    }

    f_read( buffer, size, 1, file );
    f_close( file );
    return 0;
}
```

f_wdir_walk

Use this function to find all files and sub-directories.

Note: This is only available if configuration option `F_DIR_WALK_SUPPORT` is set.

Format

```
int f_wdir_walk (
    const char *      filename,
    F_WFIND *        p_find,
    t_wdir_walk_cb_fn * p_callback,
    void *           p_user_data )
```

Arguments

Argument	Description	Type
filename	The filename or search mask. Examples include "*.*", "foo.bin", "a:\\foo*.txt", or "*".	char *
p_find	A pointer to the structure used by <code>wfindfirst()/wfindnext()</code> .	F_WFIND *
p_callback	A pointer to the callback function which is called for every directory and file.	t_wdir_walk_cb_fn *
p_user_data	A pointer to the returned list of files and sub-directories.	void *

Return values

Return value	Description
0	Successful execution.
Else	See Error Codes .

Example

The output from this is similar to the Windows command prompt's **tree /f** command.

```

/* Callback function for f_wdir_walk */

int tree_wdir_walk_fn ( F_WFIND * p_find, uint16_t depth, void * p_user_data )
{
    uint8_t idx;
    uint8_t b_is_only_dots; /* Will be TRUE if filename is "." or ".." */

    b_is_only_dots = FALSE;

    if ( p_find->filename[0] == '.' )
    {
        if ( p_find->filename[1] == 0 )
        {
            b_is_only_dots = TRUE;
        }
        else if ( ( p_find->filename[1] == '.' ) && ( p_find->filename[2] == 0 ) )
        {
            b_is_only_dots = TRUE;
        }
    }

    if ( b_is_only_dots == FALSE )
    {
        for ( idx = 1 ; idx < depth ; idx++ )
        {
            printf( "    " );
        }
        if ( depth > 0 )
        {
            printf( " +- " );
        }

        if ( p_find->attr & F_ATTR_DIR )
        {
            printf( "[%ls]\r\n", p_find->filename );
        }
        else
        {
            printf( "[%ls %u\r\n", p_find->filename, (int)p_find->filesize );
        }
    }

    return 0;
}
    
```

```
void print_treew ( void )
{
    F_WFIND find;
    f_wdir_walk( u"*.*", &find, &tree_wdir_walk_fn, NULL );
}
```

Unicode Translation

To enable full Unicode support, you must provide functions to translate characters from Unicode to ASCII and to convert characters from ASCII to Unicode.

This is performed using the two user-provided functions described below. The package contains sample implementations of these functions for the [Shift JIS](#) Japanese character set in the **src/fat/common/fat_shjis.c** file.

The functions are the following:

Function	Description
f_set_ascii_to_unicode()	Converts one or two single byte ASCII characters to a single UNICODE wide-byte character.
f_set_unicode_to_ascii()	Converts a single UNICODE wide-byte character to one or two single byte ASCII characters.

f_set_ascii_to_unicode

Use this function to convert one or two single byte ASCII characters to a single UNICODE wide-byte character.

Format

```
uint32_t f_set_ascii_to_unicode(  
    wchar *      p_dst,  
    const char * p_src,  
    uint32_t *   p_len_src )
```

Arguments

Argument	Description	Type
p_dst	Where to place the single wide-byte character.	wchar *
p_src	The ASCII character(s) to convert.	char *
p_len_src	The available space in the location. On return this holds the number of input characters written to * p_dst.	uint32_t *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	Conversion failed.

f_set_unicode_to_ascii

Use this function to convert a single UNICODE wide-byte character to one or two single byte ASCII characters.

Format

```
uint32_t f_set_unicode_to_ascii (
    char *      p_dst,
    const wchar src,
    uint32_t *  p_len_dst )
```

Arguments

Argument	Description	Type
p_dst	Where to place the ASCII character(s).	char *
src	The single wide-byte Unicode character to convert.	wchar
p_len_dst	The available space in the location. On return this holds the number of characters written to * p_dst.	uint32_t *

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	Conversion failed.

7.4. Error Codes

The table below lists all the error codes that may be generated by API calls to HCC's file systems. Please note that some error codes are not used by every file system.

The header file to include for this list is: `/src/api/api_fs_err.h`

Error	Value	Meaning
F_NO_ERROR	0	Successful execution.
F_ERR_INVALIDDRIVE	1	The specified drive does not exist.
F_ERR_NOTFORMATTED	2	The specified volume has not been formatted.
F_ERR_INVALIDDIR	3	The specified directory is invalid.
F_ERR_INVALIDNAME	4	The specified file name is invalid.
F_ERR_NOTFOUND	5	The file or directory could not be found.
F_ERR_DUPLICATED	6	The file or directory already exists.
F_ERR_NOMOREENTRY	7	The volume is full.
F_ERR_NOTOPEN	8	The file access function requires the file to be open.
F_ERR_EOF	9	End of file.
F_ERR_RESERVED	10	Not used.
F_ERR_NOTUSEABLE	11	Invalid parameters for f_seek() .
F_ERR_LOCKED	12	The file has already been opened for writing/appending.
F_ERR_ACCESSDENIED	13	The necessary physical read and/or write functions are not present for this volume.
F_ERR_NOTEMPTY	14	The directory to be moved or deleted is not empty.
F_ERR_INITFUNC	15	No init function is available for a driver, or the function generates an error.
F_ERR_CARDREMOVED	16	The card has been removed.
F_ERR_ONDRIVE	17	Non-recoverable error on drive.
F_ERR_INVALIDSECTOR	18	A sector has developed an error.
F_ERR_READ	19	Error reading the volume.
F_ERR_WRITE	20	Error writing file to volume.
F_ERR_INVALIDMEDIA	21	Media not recognized.
F_ERR_BUSY	22	The caller could not obtain the semaphore within the expiry time.
F_ERR_WRITEPROTECT	23	The physical medium is write protected.
F_ERR_INVFATTYPE	24	The type of FAT is not recognized.

Error	Value	Meaning
F_ERR_MEDIATOOSMALL	25	Media is too small for the format type requested.
F_ERR_MEDIATOO LARGE	26	Media is too large for the format type requested.
F_ERR_NOTSUPPSECTORSIZE	27	The sector size is not supported. The only supported sector size is 512 bytes.
F_ERR_UNKNOWN	28	Unspecified error has occurred.
F_ERR_DRVALREADYMNT	29	The drive is already mounted.
F_ERR_TOOLONGNAME	30	The name is too long.
F_ERR_NOTFORREAD	31	Not for read.
F_ERR_DELFUNC	32	The delete drive driver function failed.
F_ERR_ALLOCATION	33	psp_malloc() failed to allocate the required memory.
F_ERR_INVALIDPOS	34	An invalid position is selected.
F_ERR_NOMORETASK	35	All task entries are exhausted.
F_ERR_NOTAVAILABLE	36	The called function is not supported by the target volume.
F_ERR_TASKNOTFOUND	37	The caller's task identifier was not registered - normally because f_enterFS() has not been called.
F_ERR_UNUSABLE	38	The file system has become unusable, normally due to excessive error rates on the underlying media.
F_ERR_CRCERROR	39	A CRC error has been detected on the file.
F_ERR_CARDCHANGED	40	The card that was being accessed has been replaced with a different card.
F_ERR_REPAIRNEEDED	41	f_repair() must be used to return the system to a consistent state.
F_ERR_CHECKNEEDED	42	Endless FAT chain detected; call fs_fat_checkdisk() .

7.5. Types and Definitions

This section describes the main elements that are defined in the API Header file.

W_CHAR: Character and Wide Character Definition

W_CHAR is defined to char if Unicode is disabled and to wchar if it is enabled. Therefore W_CHAR is used in structures where the element could be used in either type of system.

F_FILE: File Handle

The file handle, used as a reference for accessing files.

The handle is obtained when a file is opened and released when it is closed.

F_FIND

The *F_FIND* structure takes this form:

Element	Type	Description
filename[F_MAXPATHNAME]	char	Long file name.
name[F_MAXNAME]	char	Short file name.
ext[F_MAXSEXT]	char	Short file name extension.
attr	unsigned char	Attribute setting of the file.
ctime	unsigned short	Creation time.
cdate	unsigned short	Creation date.
filesize	uint32_t	Length of file.
cluster	uint32_t	For internal use only.
findfsname	F_NAME	For internal use only.
pos	F_POS	For internal use only.

F_WFIND

The *F_WFIND* structure takes this form:

Element	Type	Description
filename[F_MAXPATHNAME]	W_CHAR	File name + extension.
name[F_MAXNAME]	char	File extension.
ext[F_MAXSEXT]	char	File name.
attr	unsigned char	Attribute of the file.
ctime	unsigned short	Creation time.
cdate	unsigned short	Creation date.
filesize	uint32_t	Length of file.
cluster	uint32_t	File system internal use only.
findfsname	F_NAME	File system internal use only.
pos	F_POS	File system internal use only.

F_SPACE

The *F_SPACE* structure takes this form:

Element	Type	Description
total	uint32_t	The total size in bytes of the disk.
free	uint32_t	The number of free bytes on the disk.
used	uint32_t	The number of used bytes on the disk.
bad	uint32_t	The number of bad bytes on the disk.
total_high	uint32_t	The high part of total if greater than 4GB.
free_high	uint32_t	The high part of free if greater than 4GB.
used_high	uint32_t	The high part of used if greater than 4GB.
bad_high	uint32_t	The high part of bad if greater than 4GB.

F_PARTITION

The *F_PARTITION* structure takes this form:

Element	Type	Description
secnum	uint32_t	The number of sectors in this partition.
system_indicator	unsigned char	Use F_SYSIND_XXX values.
bootable	unsigned char	If this is not 0, the bootable (active) bit of the partition will be set.

F_STAT Structure

The *F_STAT* structure takes this form:

Element	Type	Description
filesize	uint32_t	The size of the file.
createdate	unsigned short	The creation date.
createtime	unsigned short	The creation time.
modifieddate	unsigned short	The last modified date.
modifiedtime	unsigned short	The last modified time.
lastaccessdate	unsigned short	The last accessed date.
attr	unsigned char	00ADVSHR
drivenum	int	The number of the volume.

ST_FILE_CHANGED

The *ST_FILE_CHANGED* structure takes this form:

Element	Type	Description
action	unsigned char	Change made to the file.
flags	unsigned char	Flag to indicate changed object type.
attr	unsigned char	File attributes.
ctime	unsigned short	Creation time of file.
cdate	unsigned short	Creation date of file.
filesize	uint32_t	Size of modified file.
filename[F_MAXPATHNAME]	W_CHAR	Name of modified file.

Change Object Flags

These flags are used to indicate the type of property that has changed.

Definition	Description
FFLAGS_NONE	No object specified.
FFLAGS_FILE_NAME	The file name.
FFLAGS_DIR_NAME	The directory name.
FFLAGS_NAME	The name.
FFLAGS_ATTRIBUTES	The attributes of the object.
FFLAGS_SIZE	The file size.
FFLAGS_LAST_WRITE	The modification time.

Change Object Actions

These flags are used to indicate the action that has been applied to the object that has changed:

Definition	Description
FACTION_ADDED	File was added to the system.
FACTION_REMOVED	File was removed from the system.
FACTION_MODIFIED	File was modified.
FACTION_RENAMED_OLD_NAME	Old name of a file which was renamed.
FACTION_RENAMED_NEW_NAME	New name of a file which was renamed.

Directory Entry Attributes

Directory entries, meta-description elements for files and directories, can have attributes assigned to them. These are detailed in the table below.

Attribute	Description
F_ATTR_ARC	An archived file or directory.
F_ATTR_DIR	A directory.
F_ATTR_VOLUME	A volume.
F_ATTR_SYSTEM	A system file or directory.
F_ATTR_HIDDEN	A hidden file or directory.
F_ATTR_READONLY	A read-only file or directory.

Format Type

These definitions are used to specify how a drive should be formatted:

Definition	Description
F_FAT12_MEDIA	Format as FAT12.
F_FAT16_MEDIA	Format as FAT16.
F_FAT32_MEDIA	Format as FAT32.

System Indicator

These definitions indicate the type of the partition.

Definition	Description
F_SYSIND_DOSFAT12	A standard FAT12 partition.
F_SYSIND_DOSFAT16UPTO32MB	A FAT16 partition of less than or equal to 32MB.
F_SYSIND_DOSFAT16OVER32MB	A FAT16 partition that is over 32MB in size.
F_SYSIND_DOSFAT32	A standard FAT32 partition.

cdate Definitions

The *cdate* definitions are as follows:

Element	Value	Description
F_CDATE_DAY_SHIFT	0	The day shift.
F_CDATE_DAY_MASK	0x001F	0-31.
F_CDATE_MONTH_SHIFT	5	The month shift.
F_CDATE_MONTH_MASK	0x01E0	1-12.
F_CDATE_YEAR_SHIFT	9	The year shift.
F_CDATE_YEAR_MASK	0xFE00	0-119 (1980+value).

ctime Definitions

The *ctime* definitions are as follows:

Element	Value	Description
F_CTIME_SEC_SHIFT	0	The second shift.
F_CTIME_SEC_MASK	0x001F	0-30 in 2 second intervals.
F_CTIME_MIN_SHIFT	5	The minute shift.
F_CTIME_MIN_MASK	0x07E0	0-59.
F_CTIME_HOUR_SHIFT	11	The hour shift.
F_CTIME_HOUR_MASK	0xF800	0-23.

t_dir_walk_cb_fn

This *typedef* defines the callback function used by **f_dir_walk()**.

For a code example, see [f_dir_walk\(\)](#).

Format

```
typedef int t_dir_walk_cb_fn (  
    F_FIND * p_find,  
    uint16_t depth,  
    void * p_user_data )
```

Arguments

Argument	Description	Type
p_find	A pointer to the structure used by findfirst()/findnext() .	F_FIND *
depth	The directory depth, counted from the base directory.	uint16_t
p_user_data	A pointer to the user's data.	void *

Return values

Return value	Description
0	Successful execution.
Else	See Error Codes .

t_wdir_walk_cb_fn

This *typedef* defines the callback function used by **f_wdir_walk()**.

For a code example, see [f_wdir_walk\(\)](#).

Format

```
typedef int t_wdir_walk_cb_fn (
    F_WFIND * p_find,
    uint16_t depth,
    void * p_user_data )
```

Arguments

Argument	Description	Type
p_find	A pointer to the structure used by f_wfindfirst()/f_wfindnext() .	F_WFIND *
depth	The directory depth, counted from the base directory.	uint16_t
p_user_data	A pointer to the user's data.	void *

Return values

Return value	Description
0	Successful execution.
Else	See Error Codes .

t_volume_clean_stage_type

The `t_volume_clean_stage_type` typedef is used by the callback [t_volume_clean_cb_fn\(\)](#) to report progress from `f_volume_clean()`. See the description of the callback for a description of the *total* and *cmplt* parameters.

There are four stages:

Stage	Description
VOLUME_CLEAN_STAGE_METADATA = 0	<i>total</i> and <i>cmplt</i> will be valid. Directory entries and files are cleaned in this stage.
VOLUME_CLEAN_STAGE_FREE_SPACE	<i>total</i> and <i>cmplt</i> will be valid. Unused clusters are cleaned in this stage.
VOLUME_CLEAN_STAGE_BACKEND	Progress will not be known so <i>total</i> and <i>cmplt</i> will be 0. IOCTL flush is used in this stage.
VOLUME_CLEAN_STAGE_COMPLETE	<i>total</i> and <i>cmplt</i> will be 0.

t_volume_clean_cb_fn

The callback function `t_volume_clean_cb_fn()` is used by `f_volume_clean()`.

Format

```
typedef int t_volume_clean_cb_fn (
    void *      p_user_data,
    t_volume_clean_stage_type stage,
    uint32_t    total,
    uint32_t    cmplt )
```

Arguments

Argument	Description	Type
<code>p_user_data</code>	A pointer to the user's data.	<code>void *</code>
<code>stage</code>	The current stage of the clean.	
<code>total</code>	The total number of items that will be processed during the current stage.	
<code>cmplt</code>	The number of items processed so far during current stage.	

Return values

Return value	Description
0	Successful execution..
Else	See Error Codes .

8. Integration

This section describes all aspects of the file system that require integration with your target project.

This includes porting and configuration of external resources.

8.1. OS Abstraction Layer

The module uses the OS Abstraction Layer (OAL) that allows it to run seamlessly with a wide variety of RTOSes, or without an RTOS.

The file system uses the following OAL components:

OAL Resource	Number required if FN_MAXTASK is 1	Number required if FN_MAXTASK > 1
Tasks	0	0
Mutexes	0	1 + FAT_MAXVOLUME
Events	0	0

Configuring the OAL

Configure the OAL by using the file **config_oal_os.h**. Do the following:

1. Define `OAL_TASK_GET_ID_SUPPORTED` and `OAL_MUTEX_SUPPORTED`.
2. Set the `OAL_MUTEX_COUNT` in **config_oal_os.h** to `FN_MAXVOLUMES+1`.

Multiple Tasks, Mutexes and Reentrancy

Note: If your system has multiple tasks that access the file system, you must implement this section.

Each volume should be protected by a mutex mechanism to ensure that file access is safe. A reentrancy wrapper is included in **fat_m.c**. The reentrancy wrapper routines call mutex routines contained in the OAL.

If reentrancy is required, the following functions from the OAL are used:

- **oal_mutex_create()** – called on volume init/delete and also on file system init/delete.
- **oal_mutex_delete()** – called on volume init/delete and also on file system init/delete.
- **oal_mutex_get()** – called when a mutex is required.
- **oal_mutex_put()** – called when the mutex is released.

Within the standard API there is no support for the current working directory (**cwd**) to be maintained on a per-caller basis. By default the system provides a single **cwd** that can be changed by any user. The **cwd** is maintained on a per-volume basis, or on a per-task basis if reentrancy is implemented.

For a multitasking system, you must do the following:

1. Set `FN_MAXTASK` to the maximum number of tasks that can simultaneously maintain access to the file system. This effectively creates a table of **cwds** for each task.
2. Modify the function **oal_task_get_id()** to get a unique identifier for the calling task.
3. Ensure that any task using the file system calls **f_enterFS()** before using any other API calls; this ensures that the calling task is registered and the current working directory can be maintained for it.
4. Ensure that any application using the file system calls **f_releaseFS()** with its unique identifier to free that table entry for use by other applications.

Once this is done, each caller is logged as it acquires the mutex, and a current working directory is associated with it. The caller must release this when it has finished using the file system; that is, when the calling task is terminated. This frees the entry for use by other tasks.

8.2. PSP Porting

The Platform Support Package (PSP) is designed to hold all platform-specific functionality, either because it relies on specific features of a target system, or because this provides the most efficient or flexible solution for the developer.

The module makes use of the following standard PSP functions:

Function	Package	Element	Description
psp_getcurrenttimedate()	psp_base	psp_rtc	Returns the current time and date. This is used for date and time-stamping files.
psp_getrand()	psp_base	psp_rand	Generates a random number. This is used for the volume serial number.
psp_memcpy()	psp_base	psp_string	Copies a block of memory. The result is a binary copy of the data.
psp_memset()	psp_base	psp_string	Sets the specified area of memory to the defined value.

If [USE_MALLOC](#) is defined, the module also makes use of the following functions:

Function	Package	Element	Description
psp_free()	psp_base	psp_alloc	Deallocates a block of memory allocated by psp_malloc() , making it available for further allocation.
psp_malloc()	psp_base	psp_alloc	Allocates a block of memory, returning a pointer to the beginning of the block.

The module makes use of the following standard PSP macros:

Macro	Package	Element	Description
PSP_RD_LE16	psp_base	psp_endianness	Reads a 16 bit value stored as little-endian from a memory location.
PSP_RD_LE32	psp_base	psp_endianness	Reads a 32 bit value stored as little-endian from a memory location.
PSP_WR_LE16	psp_base	psp_endianness	Writes a 16 bit value to be stored as little-endian to a memory location.
PSP_WR_LE32	psp_base	psp_endianness	Writes a 32 bit value to be stored as little-endian to a memory location.

Get Time and Date

For compatibility with other systems, you must provide a real-time function so that files can be time-stamped and date-stamped.

A pseudo time/date function, **psp_getcurrenttimedate()**, is provided in **psp_rtc.c**. Modify this to provide the time in standard format from a Real-Time Clock source (RTC).

Random Number

The **psp_rand.c** file contains a function **psp_getrand()** that the file system uses to obtain a pseudo-random number to use as the volume serial number. This function is required only if a hard-format of a device is required.

It is recommended that you replace this routine with a random function from the base system, or alternatively generate a random number based on a combination of the system time and date and a system constant such as a MAC address.

9. Version

Version 5.80

For use with FAT and SafeFAT versions 9.8 and above