FAT and SafeFAT File System User's Guide

Version 3.50

For use with FAT and SafeFAT Versions 8.1 and above

Date: 15-May-2014 13:07

Table of Contents

1 System Overview

1.1 Introduction

This guide is intended for use by embedded software engineers who have a knowledge of the C programming language and standard file APIs, and who wish to implement a fail-safe file system. This manual covers HCC'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.

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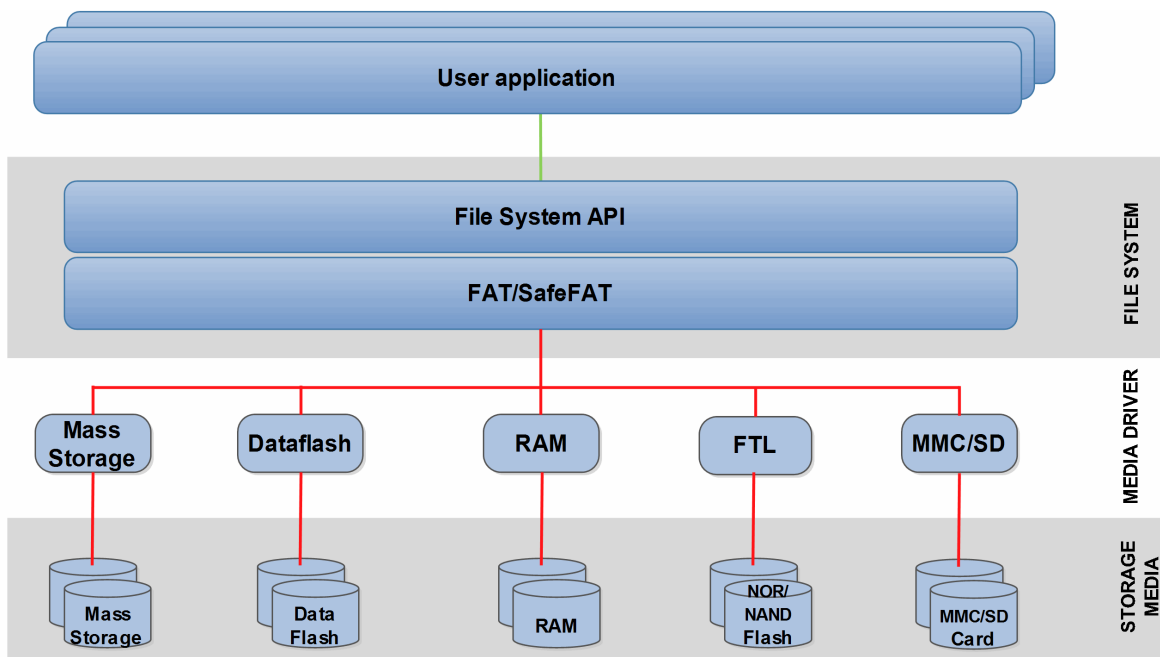
User applications use the standard file API (Application Programming Interface) to issue file system commands to the FAT file system. The FAT file system makes use of media drivers to access one or more storage media to execute the requested storage operation.

> **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:

- Code size ~24KB (FAT) ~31KB (SafeFAT).
- RAM >3KB (FAT) >6KB (SafeFAT).
- Fail safety (SafeFAT only).
- ANSI 'C'.
- Long filename support.
- Unicode 16 support.
- Multiple open files.
- Multiple open file users.
- Multiple volumes.
- Multi-sector read/write.
- Variable sector size support.
- Partition handling.
- Media error handling.
- Test suite.
- Cache options for improved performance.
- Zero copy.
- Reentrant.
- Common API (CAPI) support.
- Secure delete option (but needs special driver support).
- FAT-compatible.
- Standard drivers available for SD, SDHC, SDXC, MMC, SafeFTL, USB-MST, HDD and RAM.

1.3 Packages

The table below list the packages which need to be used with this module, and also optional modules which may interact with this module depending on your particular system's design:

Package	Description
hcc_base_doc	This contains the two guides that will help you get started.
fs_fat	The FAT file system package. This is the base system on which SafeFAT is built.
fs_fat_safe	The SafeFAT package that contains the extensions to FAT.
media_drv_base	The Media Driver base package that provides the base for all media drivers that attach to the file system.
media_drv_ram	The RAM Media Driver package, used for creating a RAM drive.

Additional packages

Other packages may also be provided to work with FAT. Examples include specific media drivers for particular targets, and Platform Specific Package (PSP) extensions for particular targets.

1.4 Documents

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's Guide

This is this document.

1.5 Disclaimer

Although every attempt has been made to make the system as simple to use as possible, developers must understand fully the requirements of the systems they are designing in order to get the best practical benefit from the system. HCC Embedded has made strenuous efforts to test and verify the correctness of its products, but it remains the sole responsibility of the user to ensure that these products meet the developer's requirements.

1.6 About HCC Embedded

HCC Embedded has been supplying professional middleware to the embedded industry for more than a decade. Our software is 'white labeled' by many of the industry's leading RTOS companies and is deployed in thousands of successful and innovative applications.

HCC Embedded is focused entirely on embedded communications and storage. In order to effectively deploy our software, we have created an advanced embedded framework that enables our software to easily drop into any environment, regardless of processor, tools or RTOS.

1.7 Getting Help

HCC Embedded has a dedicated development team with vast experience of developing embedded systems. HCC Embedded is always interested in porting or developing its systems for new environments.

Technical questions can be directed to support@hcc-embedded.com.

For custom development work or porting, please contact sales@hcc-embedded.com.

2 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 summarise:

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.

2.1 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.

2.2 PC Compatibility

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

- If the system is shut down normally, the contents of the drive are fully PC 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 in a PC, the PC 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 PC is concerned, as if it had been running a standard FAT system. That is, there may be errors on the disk, such as lost clusters. These errors can be fixed by inserting the volume in a system running SafeFAT before the PC modifies any content on the disk.

2.3 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-Embedded 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.

2.4 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.

3 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.

3.1 API Interface

The following files 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 API.

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

3.2 Configuration Files

The following files contain all the configurable parameters of the system. Configure these as required. For detailed explanations of these options, see Configuration Options.

File	Description
src/config/config_fat.c	If dynamic memory allocation is enabled on the system, this file defines dynamic memory allocation for each volume.
src/config/config_fat.h	User configuration file.

3.3 Media Drivers

All the media drivers are included under the media driver directory. The RAM drive file **src/media_drv/ram/ramdrv_f.c** is provided as standard.

Other drivers, for example for MMC/SD cards or USB mass storage, can be provided on request.

3.4 FAT File System

These files should only be modified by HCC.

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

3.5 SafeFAT File System

The following files are extensions to the FAT file system to provide the SafeFAT functionality. **These files should only be modified by HCC.**

File	Description
src/fat/safe/safefat.c	SafeFAT-specific source code.
src/fat/safe/safefat.h	SafeFAT-specific header file.

3.6 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.

4 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.

4.1 config_fat.h

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

General Options

FN_CAPI_USED

The default is zero. 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's 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 zero. To enable this option, you must uncomment the following line:

```
/* #define HCC_UNICODE */
```

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 zero.

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.

F_SUPPORT_TI64K

The default is zero. 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 zero, 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 zero, meaning these structures are statically allocated.

- Set USE_MALLOC to zero 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:** There are two versions of the file system initialization function: **f_init(1)** for when USE_MALLOC is disabled, **f_init(2)** for when it is enabled.

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 zero. 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.

File Names

F_LONGFILENAME

Enable this to use long filenames. The default is zero. 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 file path. 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 zero.

Other File Options

F_MAXFILES

The total number of files that may be open simultaneously across all volumes. The default is 7.

F_MAXSEEKPOS

The number of seek points in a file to be stored with each file descriptor. If this is set to zero, 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 zero.

> **Note:** The memory usage of the system is increased by: F_MAXSEEKPOS*F_MAXFILES*sizeof(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 zero (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 zero.

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 zero.

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.

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 zero.

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 zero.

Sector Size

F_DEF_SECTOR_SIZE

The default sector size 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 sector size of the attached media. The default is 2048. 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_SECTORx

The maximum sector size 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_READAHEADx is set to 4 and F_SZ_MAX_SECTOR_SIZEx is 2048 then, if 512-byte per sector media are connected, the allocated FAT cache block will be 4*2048=8192 bytes. The read-ahead size will therefore be 16 sectors of 512 bytes each.

Caching

FATCACHE_ENABLE

The default is 1. Refer to FAT Caching.

DIRCACHE_ENABLE

The default is 1. Refer to Directory Cache .

FATBITFIELD_ENABLE

The default is zero. 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.

F_N_FATCACHE_BLOCKSx

The "x" represents the volume number for which the cache is allocated. The default is 4. Refer to FAT Caching .

F_N_FATCACHE_READAHEADx

The "x" represents the volume number for which the cache is allocated. The default is 8. Refer to FAT Caching.

F_N_DIRCACHE_SECTORSx

The "x" represents the volume number for which the cache is allocated. The default is 8. Refer to FAT Caching .

WR_DATACACHE_SIZE

The default is 8. Refer to Write Caching .

4.2 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:

- 1024u
- 16384u
- 1048576u
- 0xffffffffu

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.

5 Other File System Information

5.1 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.

5.2 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 will use <2KB of stack. However, if long filenames are used, increase the stack size to 4KB; see Directory Cache.

5.3 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.

5.4 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:, etc.)

> **Note**: The API functions **f_getdrive()**, **f_chdrive**(), and **f_getdcwd()** 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.

5.5 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.

5.6 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. 32 (<=max. 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 setting for the first volume above requires 16K 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 32
```

This requires 768 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.

5.7 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()**.

6 API

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

6.1 Module Management

The file system follows these HCC standard module management rules:

- An **init** function allocates module resources.
- A **delete** function releases allocated resources.
- A **start** function puts the module into the running state.
- A **stop** function puts the module into the stopped state.

The following sections describe how to use these module management functions.

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 zero, 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

Argument
None.

Return values

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

Example

```
void main()
{
    fs_init(); /* initialize filesystem */
        .
        .
        .
}
```

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

Argument
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 any new call to **fs_start()** is successfully completed.

Format

```
int fs_stop ( void )
```

Arguments

Argument
None.

Return values

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

fs_delete

Use this function 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

Argument
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.

6.2 File System API

This section describes all the Application Programmer 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

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 **f_init()**.

Format

```
int f_enterFS ( void )
```

Arguments

Argument
None.

Return values

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

Example

```
void main()
{
    f_init(); /* initialize filesystem */
    f_enterFS(); /* allow current (only) task access filesystem */
        .
        .
}
```

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. It must be called if a given task is released or no longer exists.

Format

```
void f_releaseFS ( void )
```

Arguments

Argument
None.

Return values

Return value
None.

Example

```
void task_destructor()
{
    f_releaseFS(); /* release 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

Argument
None.

Return values

Return value	Description
Error code	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".

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	Drive to be initialized (0='A', 1='B', and so on).	int
driver_init	Initialization function for 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 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;
    f_init();
    f_enterFS();

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

    /*Make a Compact Flash Volume on Drive B */
    f_initvolume(1, cfc_initfunc, 0);

    /*Make an MMC Volume on Drive C */
    f_initvolume(2, mmc_initfunc, 0);


            .
            .
            .

}
```

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	Drive to be initialized (0='A', 1='B', and so on).	int
driver_init	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;
    f_init();
    f_enterFS();

    /* Make a RAM volume on Drive A */ f_initvolume_nonsafe(0, ram_initfunc, F_AUTO_ASSIGN);
    /*Make a Compact Flash Volume on Drive B */

    f_initvolume_nonsafe(1, cfc_initfunc, F_AUTO_ASSIGN);

    /*Make an MMC Volume on Drive C */

    f_initvolume_nonsafe(2, mmc_initfunc, F_AUTO_ASSIGN);
                .
                .
                .
}
```

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 to a previously opened file handle return with 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	Drive to be deleted (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	Drive to be checked (0='A', 1='B', and so on).	int

Return values

Return value	Description
F_NO_ERROR	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_setvolname

Use this function to set the name of a volume.

Specify the volume to assign the name to, and the name to be assigned to it. The name must be at least two characters long and null-terminated.

> **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	Volume number.	int
p_name	Name to set for the volume.	char *

Return values

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

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	Volume number.	int
p_buffer	Where to store name of the volume.	char *
maxlen	Buffer length (possible size is F_MAXPATHNAME).	int

Return values

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

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	Drive number.	int
str	Where to copy the label to. This should be able to hold an eight character string.	char *
len	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

Argument
None.

Return values

Return value	Description
num	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. Storage should be of size F_MAXVOLUME.	int *

Return values

Return value	Description
number	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
drivenum	Drive to be initialized (0='A', 1='B', and so on).	int
driver	Pointer to 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;

    f_init()
    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	Drive to be initialized (0='A', 1='B', and so on).	int
driver	Pointer to 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;

    f_init()

    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_createpartition

Use this function to create one or more partitions on a drive. Call it with a pointer to the function that must be called to retrieve drive configuration information.

This function may also be used to remove partitions by overwriting the current partition table.

> **Note:** If only a single volume is required, it is simpler not to use a partition table but to use **f_initvolume()** to format it.

Calling this function logically destroys all data on the drive.

The number of sectors on the target drive can be found by calling the driver->getphy(driver,&phy). This information can be used to build the F_PARTITION structure before **f_createpartition()** is called.

Format

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

Arguments

Argument	Description	Type
driver	Initialized driver; get this by calling **f_createdriver()**.	F_DRIVER *
parnum	Number of partition in partition descriptor.	int
par	Pointer to 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_DOSFAT16UPTO32MB,0},
    {2000, F_SYSIND_DOSFAT16UPTO32MB,0}
};

F_DRIVER *hdd;

int mypartitiondrive()
{
    int ret;

    f_init();
    f_enterFS();

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

    ret=f_createpartition(driver,2,par2);
    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	Initialized driver; get this by calling **f_createdriver()**.	F_DRIVER *
parnum	Number of entry in the par parameter.	int
par	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, **f_releasedriver()** must be called to release the driver.

Format

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

Arguments

Argument	Description	Type
driver	Pointer to F_DRIVER structure of the required driver.	F_DRIVER * *
driver_init	Pointer to 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 will be initialized. For more details about this, 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;

    f_init();
    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()** will release 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()** will be called automatically for each volume on this drive.

Format

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

Arguments

Argument	Description	Type
driver	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;

    f_init();
    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	Drive number 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

Argument
None.

Return values

Return value	Description
Current Drive	0='A', 1='B', and so on.
Else	See Error Codes.

Example

```
{
    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	Drive number (0='A', 1='B', and so on).	int
pspace	Pointer to 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 %d bytes total, %d bytes free, \
        %d bytes used, %d bytes bad.",
            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	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	Length of 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,12);

    if (result)
        printf("Error on Drive");
    else
        printf("Drive is %s",label);
}
```

f_setlabel

Use this function to set a volume label.

The volume label should be an ASCII string with a maximum length of 11 characters. Non-printable characters are padded out as space characters.

Format

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

Arguments

Argument	Description	Type
drivenum	Drive number (0='A', 1='B', and so on).	int
pLabel	Pointer to the null-terminated string to use.	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");
}
```

Directory Management

f_mkdir

Use this function to create a new directory.

Format

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

Arguments

Argument	Description	Type
dirname	New directory name to create.	char *

Return values

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

Example

```
void myfunc(void)
{
        .
        .
    f_mkdir("subfolder"); /*creating directory */
    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	Null-terminated string containing name of 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");      /* goto into sub2 dir */
        .
        .
}
```

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	Name of 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"); /*creating directories */
    f_mkdir("subfolder/sub1");
        .
        . /* doing some work */
        .
    f_rmdir("subfolder/sub1");
    f_rmdir("subfolder"); /*removes directory */
        .
        .
}
```

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 current working directory string.	char *
maxlen	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 current working directory string.	char *
maxlen	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

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	File to be opened.	char *
mode	The opening mode (see above).	char *

Return values

Return value	Description
F_FILE *	Pointer to the associated opened file handle, or zero if it 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 1 byte */
        printf ("'%c' 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	File to be opened.	char *
mode	The opening mode (see above).	char *

Return values

Return value	Description
F_FILE *	Pointer to the associated opened file handle, or zero if it 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 1 byte */
        printf ("'%c' 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	Handle of target 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 fn_abortclose (
    F_MULTI *   fm,
    FN_FILE *   filehandle )
```

Arguments

Argument	Description	Type
fm	Multi-structure pointer.	F_MULTI *
filehandle	The file which needs to be closed.	FN_FILE *

Return values

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

Example

```c
#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 )
    {
      /* removing cf file, rc is already signal error */
      cf_remove( vi, f->s_name );
    }
    else
    {
      /* removing cf file*/
      rc = cf_remove( vi, f->s_name );
    }
  }
#ifdef USE_MALLOC
  if ( f->WrDataCache.pos )
  {
    psp_free( f->WrDataCache.pos );
    f->WrDataCache.pos = NULL;
  }
#endif
  /* removing 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_read

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

The file must be opened with "r", "r+", "w+" or "a+".

Format

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

Arguments

Argument	Description	Type
buf	Buffer to store data in.	void *
size	Size of items to be read.	long
size_st	Number of items to be read.	long
filehandle	Handle of target file.	F_FILE *

Return values

Return value	Description
number	Number of items read.
Else	See Error Codes.

Example

```
{
    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 ("not all items read!!");
    }
    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 with "r+", "w", "w+", "a+" or "a". 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	Pointer to data to be written.	void *
size	Size of items to be written.	long
size_st	Number of items to be written.	long
filehandle	Handle of target file.	F_FILE *

Return values

Return value	Description
number	Number of items written.
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;
    }

    /* write 3 bytes */
    if(f_write(string,1,3,file)!=3)
    {
        printf ("Error: not all items written");
    }
    f_close(file);
}
```

f_getc

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

Format

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

Arguments

Argument	Description	Type
filehandle	Handle of open target file.	F_FILE *

Return values

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

Example

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

    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	Character to be written.	char
filehandle	Handle of open target file.	F_FILE *

Return values

Return value	Description
-1	Write failed.
value	Successfully written character.

Example

```
void myfunc (char *filename, long num)
{
    F_FILE *file=f_open(filename,"w");
    while (num--)
    {
        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 open target file is the end of file (EOF).

Format

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

Arguments

Argument	Description	Type
filehandle	Handle of open target file.	F_FILE *

Return values

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

Example

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

    while (!f_eof())
    {
        if (!buffsize) break;
        buffsize--;
        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	Handle of open target file.	F_FILE *

Return values

Return value	Description
0	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 obtain the current read-write position in the open target file.

Format

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

Arguments

Argument	Description	Type
filehandle	Handle of open target file.	F_FILE *

Return values

Return value	Description
filepos	Current read or write file position.

Example

```
int myreadfunc(char *filename, char *buffer, long buffsize)
{
    F_FILE *file=f_open(filename,"r");
    printf ("Current position %d",f_tell(file));
    /* position 0 */

    f_read(buffer,1,1,file); /* read 1 byte
    printf ("Current position %d",f_tell(file));
    /* positin 1 */

    f_read(buffer,1,1,file); /* read 1 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 target file. The file must be open.

The **whence** parameter is one of the following:

- F_SEEK_CUR – current position of file pointer.
- F_SEEK_END – end of file.
- F_SEEK_SET – start of 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	Handle of open target file.	F_FILE *
offset	Byte position relative to **whence**.	long
whence	Where to calculate offset from.	long

Return values

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

Example

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

    f_read(buffer,1,1,file); /* read 1st 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 last byte */
    f_close(file);
    return 0;
}
```

f_rewind

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

Format

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

Arguments

Argument	Description	Type
filehandle	Handle of open target 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 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	File to be opened.	char *
length	New length of file.	unsigned long

Return values

Return value	Description
F_FILE *	Pointer to the associated opened file handle, or zero if it 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 the length is greater than the length of the existing file, the file is padded with zeroes to the truncated 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

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	Null-terminated string with name of file to be deleted, with or without 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; that is, to set all the content to 0xFF.

This function is available only 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	File name to be deleted, with or without 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()**. Files with the system attribute set will be ignored.

> **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	Name of file 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/*.*",&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 will be ignored.

> **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/*.*",&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	File or directory name, with or without path.	char *
newname	New name of file or directory, with or without 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.

If a file or directory is read-only it cannot be renamed. If a file is open it cannot be renamed.

Format

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

Arguments

Argument	Description	Type
filename	File or directory name, with or without path.	char *
newname	New name of 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 target 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 if 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 target file.	char *
attr	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. If FAT compatibility is required, these must use the standard type definitions for time and date given below. If FAT compatibility is not required, you can use these fields as you require. See PSP Porting for information on porting.

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 & 001fH)
Month	1-12	((d & 01e0H) >> 5)
Years since 1980	0-119	((d & fe00H) >> 9)

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

Argument	Valid values	Format
2 second increments	0-30	(t & 001fH)
Minute	0-59	((t & 07e0H) >> 5)
Hour	0-23	((t & f800H) >> 11)

Format

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

Arguments

Argument	Description	Type
filename	Target file.	char *
pctime	Where to store the creation time.	unsigned short *
pcdate	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 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. If FAT compatibility is required, these must use the standard type definitions for time and date given below. If FAT compatibility is not required, you can use these fields as you require. See PSP Porting for information on porting.

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 & 001fH)
Month	1-12	((d & 01e0H) >> 5)
Years since 1980	0-119	((d & fe00H) >> 9)

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

Argument	Valid values	Format
2 second increments	0-30	(t & 001fH)
Minute	0-59	((t & 07e0H) >> 5)
Hour	0-23	((t & f800H) >> 11)

Format

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

Arguments

Argument	Description	Type
filename	File.	char *
ctime	Creation time of file or directory.	unsigned short
cdate	Creation date of 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"); /* creating directory */
    f_settimedate("subfolder", ctime, cdate);
}
```

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	Handle of target file.	F_FILE *

Return values

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

Example

```
{
    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_format

Use this function to format the specified drive.

If the media is not present, this function fails. If successful, 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	Drive to be formatted (0='A', 1='B', and so on).	int
fattype	Type of format: • 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 no. 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;

    f_init();
    f_enterFS();

    f_initvolume(0,cfc_initfunc, F_AUTO_ASSIGN);

    ret=f_format(0, F_FAT16_MEDIA);

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

f_stat

Use this function to get information about a file.

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.

> **Note:** This function can also return the opened file's current size when **f_findopensize()** is allowed to search through all open file descriptors for its modified size. If this feature is disabled then **f_findopensize()** always returns zero.

Format

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

Arguments

Argument	Description	Type
filename	Name of the file.	char *
stat	Pointer to 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	File handle.	F_FILE *
p_stat	Pointer to 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_findopensize()** is allowed to search for it. If **f_findopensize()** always returns with zero, then this feature is disabled.

Format

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

Arguments

Argument	Description	Type
filename	File name, with or without path.	char *

Return values

Return value	Description
filelength	Length of file.
-1	The requested file does not exist or has an error.

Example

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

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

    if (size>buffsize)
    {
        printf ("Not enough memory!");
        return 2;
    }

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

    return 0;
}
```

6.3 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.

Unicode Directory Management

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	Name of Unicode 16 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("subfolder");    /* creating directory */
    f_wmkdir("subfolder/sub1");
    f_wmkdir("subfolder/sub2");
    f_wmkdir("a:/subfolder/sub3"
        .
        .
}
```

f_wchdir

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

Format

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

Arguments

Argument	Description	Type
dirname	Unicode 16 directory name 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("subfolder");
    f_wchdir("subfolder");        /* change directory */
    f_wmkdir("sub2");
    f_wchdir("..");                /* go to upward */
    f_wchdir("subfolder/sub2");    /* goto into sub2 dir */
        .
        .
}
```

f_wrmdir

Use this function to remove a directory with a Unicode 16 name.

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	Unicode 16 directory name to remove.	W_CHAR *

Return values

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

Example

```
void myfunc(void)
{
        .
        .
    f_wmkdir("subfolder"); /* creating directories */
    f_wmkdir("subfolder/sub1");
        .
        . doing some work
        .
    f_wrmdir("subfolder/sub1");
    f_wmdir("subfolder"); /* removes directory */
        .
        .
}
```

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	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_wgetdcwd(drivenum,buffer, F_MAXPATH))
    {
        wprintf ("current directory is %s",buffer);
        wprintf ("on drive %c",drivenum+'A');
    }
    else
    {
        wprintf ("Drive Error")
    }
}
```

f_wgetdcwd

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

Format

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

Arguments

Argument	Description	Type
drivenum	Drive (0='A', 1='B', and so on.).	int
buffer	Where to store the current working directory string.	W_CHAR *
maxlen	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_wgetdcwd(drivenum,buffer, F_MAXPATH))
    {
        wprintf ("current directory is %s",buffer);
        wprintf ("on drive %c",drivenum+'A');
    }
    else
    {
        wprintf ("Drive Error")
    }
}
```

Unicode File Access

f_wopen

Use this function to open a file with 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 all files to be accessed in binary mode only.

Format

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

Arguments

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

Return values

Return value	Description
F_FILE *	Pointer to the associated opened file, or zero if it could not be opened.

Example

```
void myfunc(void)
{
    F_FILE *file;
    char c;
    file=f_wopen("myfile.bin","r");
    if (!file)
    {
        wprintf ("File cannot be opened!");
        return;
    }
    f_read(&c,1,1,file); /* read 1byte */
    wprintf ("'%c' 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	File to be opened.	wchar *
mode	The opening mode (see above).	wchar *

Return values

Return value	Description
F_FILE *	Pointer to the associated opened file, or zero if it could not be opened.

Example

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

    file=f_wopen_nonsafe("myfile.bin","r");
    if (!file)
    {
        printf ("File cannot be opened!");
        return;
    }
    f_read(&c,1,1,file); /*read 1 byte */
    printf ("'%c' 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	File to be opened.	W_CHAR *
length	New length of the file.	unsigned long

Return values

Return value	Description
F_FILE *	Pointer to the associated opened file handle, or NULL if it could not be opened.

Example

```
int mywtruncatefunc(W_CHAR *filename, unsigned long length)
{
    F_FILE *file=f_wtruncate(filename,length);
    if(!file)
        printf("File not found");
    else
    {
        printf("File %s truncated to %d bytes,
        filename, length);
        f_close(file);
    }
    return 0;
}
```

Unicode File Management

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	Name of file to be deleted, with or without its path.	W_CHAR *

Return values

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

Example

```
void myfunc(void)
{
        .
        .
    f_wdelete ("oldfile.txt");
    f_wdelete ("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	Name of file to be deleted, with or without its path.	wchar *

Return values

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

Example

```
void myfunc(void)
{
        .
        .
    f_wdeletecontent ("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	Unicode 16 name of file 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("A:/subdir/*.*",&find))
    {
        do
        {
            printf ("filename:%s",find.filename);
            if (find.attr&F_ATTR_DIR)
            {
                printf (" directory\n");
            }
            else
            {
                printf (" 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	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("A:/subdir/*.*",&find))
    {
        do
        {
            printf ("filename:%s",find.filename);
         if (find.attr&F_ATTR_DIR)
            {
                printf (" directory\n");
            }
            else
            {
                printf (" 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	Unicode 16 file or directory name, with or without path.	W_CHAR *
newname	New Unicode 16 name of 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 ("oldfile.txt","newfile.txt");
    f_wmove ("A:/subdir/oldfile.txt","A:/newdir/oldfile.txt");
        .
        .
}
```

w_rename

Use this function to rename a file or directory with a Unicode 16 name.

Format

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

Arguments

Argument	Description	Type
filename	Unicode 16 file or directory name, with or without path.	W_CHAR *
newname	New Unicode 16 name of 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 ("oldfile.txt","newfile.txt");
    f_wrename ("A:/dir/oldfile.txt","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	Name of target 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 if myfile.txt is read only */
    if(!f_wgetattr("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_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	Name of target file.	wchar *
attr	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("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. If FAT compatibility is required, these must use the standard type definitions for time and date given below. If FAT compatibility is not required, you can use these fields as you require. See PSP Porting for information on porting.

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 & 001fH)
Month	1-12	((d & 01e0H) >> 5)
Years since 1980	0-119	((d & fe00H) >> 9)

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

Argument	Valid values	Format
2 second increments	0-30	(t & 001fH)
Minute	0-59	((t & 07e0H) >> 5)
Hour	0-23	((t & f800H) >> 11)

Format

```
int f_wgettimedate (
    const W_CHAR *     filename,
    unsigned short *   pctime,
    unsigned short *   pcdate )
```

Arguments

Argument	Description	Type
filename	Unicode 16 name of target file or directory.	W_CHAR *
pctime	Where to store the time.	unsigned short *
pcdate	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("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 & 0x01e0) >> 5);
        unsigned short year=1980+((d & 0xfe00) >> 9);
        wprintf ("Time: %d:%d:%d",hour,minute,sec);
        wprintf ("Date: %d.%d.%d",year,month,day);
    }
    else
    {
        wprintf ("File time cannot 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. If FAT compatibility is required, these must use the standard type definitions for time and date given below. If FAT compatibility is not required, you can use these fields as you require. See PSP Porting for information on porting.

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 & 001fH)
Month	1-12	((d & 01e0H) >> 5)
Years since 1980	0-119	((d & fe00H) >> 9)

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

Argument	Valid values	Format
2 second increments	0-30	(t & 001fH)
Minute	0-59	((t & 07e0H) >> 5)
Hour	0-23	((t & f800H) >> 11)

Format

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

Arguments

Argument	Description	Type
filename	Unicode 16 name of the file.	W_CHAR *
ctime	Creation time of file or directory.	unsigned short
cdate	Creation date of 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("subfolder");    /* creating directory */
    f_wsettimedate("subfolder",ctime,cdate);
}
```

f_wstat

Use this function to get information about a Unicode 16 file.

The function retrieves information by filling the F_STAT structure passed to it. It inserts the file size, creation time/date, last access date, modified time/date, and the drive number where the file is located.

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

Format

```
int f_wstat (
    const wchar *   filename,
    F_STAT *        stat )
```

Arguments

Argument	Description	Type
filename	The file name.	wchar *
stat	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 ( "myfile.txt", &stat ) )
    {
        printf ("error");
        return;
    }
    printf ( "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	Unicode 16 file name, with or without path.	W_CHAR *

Return values

Return value	Description
filelength	Length of file. If this is zero then the file may not exist – check the last error.

Example

```
int myreadfunc(W_CHAR *filename, char *buffer, long buffsize)
{
    F_FILE *file=f_wopen(filename,"r");
    long size=f_wfilelength(filename);
    if (!file)
    {
        printf ("%s Cannot be opened!",filename);
        return 1;
    }
    if (size>buffsize)
    {
        printf ("Not enough memory!");
        return 2;
    }
    f_read(buffer,size,1,file);
    f_close(file);
    return 0;
}
```

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**.

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
0	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
0	Successful execution.
Else	Conversion failed.

6.4 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	Success.
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.

Error	Value	Meaning
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.
F_ERR_MEDIATOOSMALL	25	Media is too small for the format type requested.
F_ERR_MEDIATOOLARGE	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 as a result of 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.

Error	Value	Meaning

6.5 Types and Definitions

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 closed.

F_FIND Structure

The structure takes this form:

Parameter Name	Description	Type
filename	Long file name.	char[F_MAXPATHNAME]
name	Short file name.	char[F_MAXSNAME]
ext	Short file name extension.	char[F_MAXSEXT]
attr	Attribute setting of the file.	unsigned char
ctime	Creation time.	unsigned short
cdate	Creation date.	unsigned short
filesize	Length of file.	unsigned long
cluster	File system internal use only.	unsigned long
findfsname	File system internal use only.	F_NAME
pos	File system internal use only.	F_POS

F_WFIND Structure

The structure takes this form:

Parameter Name	Description	Type
filename[F_MAXPATHNAME]	File name+ext.	W_CHAR
name[F_MAXSNAME]	File extension.	char
ext[F_MAXSEXT]	File name.	char
attr	Attribute of the file.	unsigned char
ctime	Creation time.	unsigned short
cdate	Creation date.	unsigned short
filesize	Length of file.	unsigned long
cluster	File system internal use only.	unsigned long
findfsname	File system internal use only.	F_NAME
pos	File system internal use only.	F_POS

F_PARTITION Structure

The structure takes this form:

Parameter Name	Description	Type
secnum	Number of sectors in this partition.	unsigned long
system_indicator	Use F_SYSIND_XXX values.	unsigned char
bootable	If not 0, bootable (active) bit of partition will be set.	unsigned char

F_SPACE Structure

The F_SPACE structure takes this form:

Parameter Name	Description	Type
total	Total size in bytes of the disk.	unsigned long
free	Free bytes on the disk.	unsigned long
used	Used bytes on the disk.	unsigned long
bad	Bad bytes on the disk.	unsigned long
total_high	High part of total if greater than 4GB.	unsigned long
free_high	High part of free if greater than 4GB.	unsigned long
used_high	High part of used if greater than 4GB.	unsigned long
bad_high	High part of bad if greater than 4GB.	unsigned long

F_STAT Structure

The structure takes this form:

Parameter Name	Description	Type
filesize	Size of the file.	unsigned long
createdate	Creation date.	unsigned short
createtime	Creation time.	unsigned short
modifieddate	Last modified date.	unsigned short
modifiedtime	Last modified time.	unsigned short
lastaccessdate	Last accessed date.	unsigned short
attr	00ADVSHR	unsigned char
drivenum	Number of volume.	int

ST_FILE_CHANGED Structure

The structure takes this form:

Parameter Name	Description	Type
action	Change made to the file.	unsigned char
flags	Flag to indicate changed object type.	unsigned char
attr	File attributes.	unsigned char
ctime	Creation time of file.	unsigned short
cdate	Creation date of file.	unsigned short
filesize	Size of modified file.	unsigned long
filename	Name of modified file.	W_CHAR[F_MAXPATHNAME]

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	File name.
FFLAGS_DIR_NAME	Directory.
FFLAGS_NAME	Name.
FFLAGS_ATTRIBUTES	Attributes of object.
FFLAGS_SIZE	Size of file.
FFLAGS_LAST_WRITE	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.

Date and Time Definitions

Date Definitions

The following flags are used to interpret date settings, consistent with the standard definitions of these for FAT file systems:

Definition	Description	Value
F_CDATE_DAY_SHIFT	Shift for days value.	0
F_CDATE_DAY_MASK	Day of the month: 1-31.	0x001f
F_CDATE_MONTH_SHIFT	Shift for months value.	5
F_CDATE_MONTH_MASK	Month of the year: 1-12.	0x01e0
F_CDATE_YEAR_SHIFT	Shift for years value.	9
F_CDATE_YEAR_MASK	Year: 0-119 (1980+value).	0xfe00

Time Definitions

The following flags are used to interpret time settings, consistent with the standard definitions of these for FAT file systems:

Definition	Description	Value
F_CTIME_SEC_SHIFT	Shift for seconds value.	0
F_CTIME_SEC_MASK	Mask for seconds - value is 0-29 in two second units.	0x1f
F_CTIME_MIN_SHIFT	Shift for minutes value.	5
F_CTIME_MIN_MASK	Mask for minutes - value is 0-59.	0x07e0
F_CTIME_HOUR_SHIFT	Shift for hours value.	11
F_CTIME_HOUR_MASK	Mask for hours - value is 0-23.	0xf800

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	Is an archived file or directory.
F_ATTR_DIR	Is a directory.
F_ATTR_VOLUME	Is a volume.
F_ATTR_SYSTEM	Is a system file or directory.
F_ATTR_HIDDEN	Is a hidden file or directory.
F_ATTR_READONLY	Is 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	Standard FAT12 partition.
F_SYSIND_DOSFAT16UPTO32MB	A FAT16 partition that is less than or equal to 32MB in size.
F_SYSIND_DOSFAT16OVER32MB	A FAT16 partition that is over 32MB in size.
F_SYSIND_DOSFAT32	A standard FAT32 partition.

7 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.

7.1 OS Abstraction Layer (OAL)

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 + F_MAXVOLUMES
Events	0	0

Configuring the OAL

Configure the OAL by using the **config_oal_os.h** file. 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 **cwd**s 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.

7.2 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.

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.