FAT File System

Implementation Guide

Version 3.31

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

Contents

1. System Overview

1.1. Target Audience

This guide is intended for use by embedded software engineers who have should have a knowledge of the C programming language, standard file API's who wish to implement a FAT12, FAT16 or FAT32 file system in any combination of RAM, Compact Flash Card, MultiMediaCard, Hard Disk Drive or other device type.

Although every attempt has been made to make the system as simple to use as possible the developer must understand the requirements of the system they are designing to get the best practical benefit from the system.

HCC-Embedded offers hardware and firmware development consultancy to assist developers with the implementation of a flash file system.

1.2. System Structure/Source Code

The following diagram illustrates the structure of the file system software.

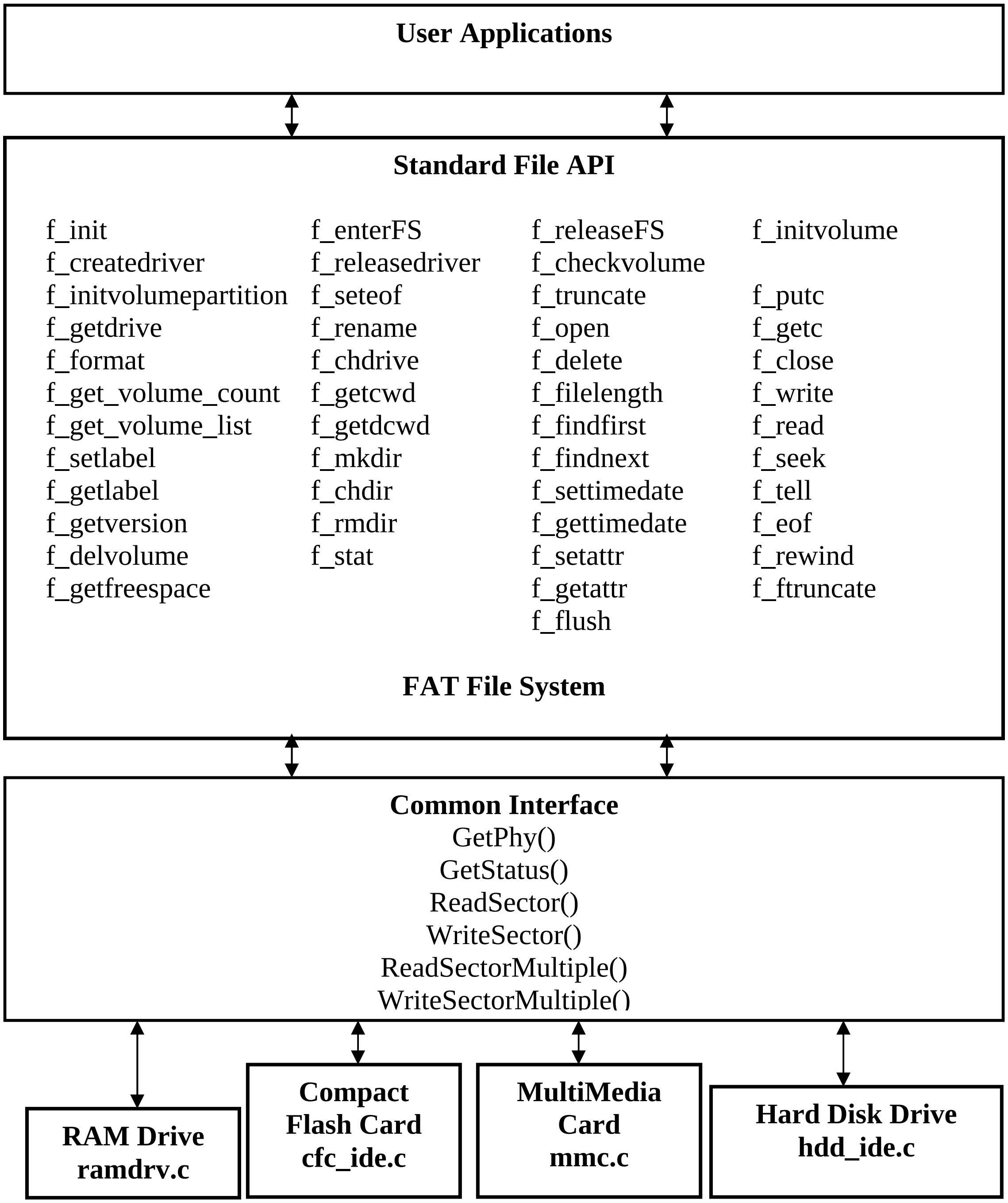

Figure 1, System Structure

1.3. Source File List

The following is a list of all the source code files included in the file system.

1.3.1. Standard Source Files

/src/common

udefs.h	- user definitions file
defs.h	- external definition file
fat.c	- fat short filename functions
fat.h	- fat file system header
fwerr.h	- error codes definitions
fat_lfn.c	- alternative source file to fat.c for long filenames
common.c	- common functions
common.h	- common functions header
fat_m.c	- fat file system reentrancy wrapper
fat_m.h	- fat file header reentrancy header
port_f.c	- routines that require OS specific modifications
port_f.h	- header for port routines.
api_f.h	- public definitions file

1.3.2. Test Code Source Files

/src/test/

test_f.c	- Test source code for exercising the file system
test_f.h	- Header file for test source code
testdrv_f.c	- test driver for testing system
testdrv_f.h	- header file for test driver
testport_f.c	- porting file for test
testport_ram_f.c	- porting fiel for RAM test

1.3.3. Checkdisk Source Files

/src/chkdsk/

chkdsk.c	- check disk utility C source code
chkdsk.h	- header file for checkdisk utility

1.3.4. Sample Driver Source Files

/src/ram/	
ramdrv_f.c	- RAM driver implementation
ramdrv_f.h	- RAM driver header file

/src/cfc/arm/	ARM7 tested Compact Flash Drivers
cfc_ide.c	- Compact Flash Card True IDE Driver
cfc_ide.h	- Compact Flash Card True IDE Header
cfc_io.c	- Compact Flash Card IO mode Driver
cfc_io.h	- Compact Flash Card IO mode Header
cfc_mem.c	- Compact Flash Card memory mode Driver
cfc_mem.h	- Compact Flash Card memory mode Header

/src/cfc/mcf/	MCF5xxx tested Compact Flash Drivers
cfc_ide.c	- Compact Flash Card True IDE Driver
cfc_ide.h	- Compact Flash Card True IDE Header

/src/mmc/multi/	MMC/SD card, multiple interface drivers
drv.h	- Header file for MMC/SD card driver
mmc.c	- Generic MultiMediaCard driver
mmc.h	- MultiMediaCard header
mmc_dsc.h	- Card specific information header

/src/mmc/multi/arm/	ARM7 tested SPI drivers
drv.c	- SPI driver for ARM7
drvs.c	- Software driven SPI driver

/src/mmc/multi/mcf/	MCF5xxx tested SPI drivers
drv.c	- SPI driver for MCF5xxx
drvs.c	- Software driven SPI driver for MCF5xxxx

/src/mmc/single/	MMC/SD single interface drivers
drv.h	- Header file for MMC/SD card driver
mmc.c	- Generic MultiMediaCard driver
mmc.h	- MultiMediaCard header
mmc_dsc.h	- Card specific information header

/src/mmc/single/arm/	ARM7 tested SPI drivers
drv.c	- SPI driver for ARM7
drvs.c	- Software driven SPI driver

/src/mmc/single/mcf/	MCF5xxx tested SPI drivers
drv.c	- SPI driver for MCF5xxx
drvs.c	- Software driven SPI driver for MCF5xxxx

/src/hdd/mcf/	MCF5xxx tested HDD driver
hdd_ide.c	- Hard Disk Drive IDE driver

hdd_ide.h	- Hard Disk Driver header file

The developer should not normally modify the fat source files. These files contain all the file system handling and maintenance including FATs, directories, formatting etc.

The **port_f.c** and **port_f.h** files need to be modified to conform to the target system the developer is working with. The tasks required of the developer are straightforward and ensure easy integration with any operating environment. Full guidance to this is given in the Section 2.

The driver files are fully tested working driver examples. For any particular implementation key parts of these must be changed to conform to the development environment. In particular address mapping and IO port mapping must be done to configure the driver to work with the developer's hardware. The driver interface functions are documented in Section 6.

The sample drivers are documented in Sections 7, 8, 9 and 10.

To implement a customized driver is straightforward. The developer should base any new driver on the RAM driver - the simplest possible starting point.

1.4. Getting Started

To get your development started as efficiently as possible we recommend that the developer follow the instructions in section RAM Driver to set up a RAM drive on their target. This enables the developer to become familiar with the system and develop test code without the need to worry about a new hardware interface.

1.5. Testing

Supplied with the system is test code for exercising the system and ensuring that the file system is working correctly. Most functionality of the file system is exercised with this program including file read/write/append/seek/file content, directories and file manipulation functions. To use the test program include **test_f.c** and **test_f.h** in your test project. **testdrv_f.c** and **testdrv_f.h** contains a test driver which is a special ram drive used for greater coverage test of the file system. **testport_f.c** contains functions which need to be modified for the target environment (e.g. if ***printf*** is called with different name). When testing a drive the F_FAT_MEDIA define in **test_f.h** should be set to that required on your target e.g. for testing the RAM drive it should be changed to:

```
#define F_FAT_MEDIA F_FAT12_MEDIA
```

There is a #define for full coverage test called TEST_FULL_COVERAGE. This define can be set to 1 if a full coverage test is needed. In this case **testdrv_f.c** must be included into the project. If the target media device is to be tested, then full coverage test should not be used because it needs some special function to simulate a variety of error conditions. In this case use normal test instead of full coverage test.

```
void f_dotest(void)  is called to execute the test code.
```

For all file system tests **testport_f.c** is needed. This file includes basic function for powering the system on and off and for displaying test results. If a target device test is requested, then ***f_initvolume()*** function call parameters must be modified in the ***_f_poweron()*** function to use the correct driver. See the comments in the file to understand how to do this.

2. Porting – Step by Step Guide

2.1. System Requirements

The 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 certain porting work should be done as outlined below. This is a straightforward task for an experienced engineer.

2.2. Stack Requirements

The file system functions are always called in the context of the calling thread or task. Naturally the functions require stack space and the developer should allow for this in applications calling file system functions. Typically calls to the file system will use <2Kbytes of stack. However, if long filenames are used then the stack size should be increased to 4K but see Long Filenames section below.

2.3. 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 writing to the physical media and in the communication cause the system to wait on external events. The points at which this occur are documented in the applicable driver sections and the developer should modify them to meet the system requirements - either by implementing interrupt control of that event or scheduling other parts of the system. Read the relevant driver section for details.

2.4. User Definitions

From release 2.70 a user definitions file has been included in the source tree to include all the main user definitions. This is done to make the product upgrade task simpler so that when a new release is provided this file should not be overwritten if you wish to retain your previous settings.

2.5. Unicode Support

From version 2.70 the support for 16-bit Unicode is provided.

> **Note:** Unicode 7/8 are supported by the file system transparently. This additional option is only required for Unicode16 support.

To support Unicode16 character sets the developer must uncomment the line:

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

This will force any build to include the Unicode 16 API. This build will also force Long Filename support (see next section) which is necessary for Unicode16 support.

With this build you may now use the Unocode16 API calls. Section 5 describes the API functions that may be used with Unicode16 strings.

Use of Unicode16 implies that the host system has wchar ("wide character") support or an equivalent definition.

Using the Unicode16 system creates additional resource usage in the system because all string and path accesses effectively use twice the space. Therefore it is recommended that this option is only used if it is a requirement of your system to use Unicode16.

Note: To allow the file system to generate consistent short filenames then the user may want to include character set conversion tables in to the code. There are two points in the code where you must insert this conversion if required – these are in the *_f_createlfn()* function in the **fat_lfn.c** module and marked with the comment:

```
/* here we can add …..
```

2.6. Drives, Partitions and Volumes

FAT provides functions for creating and managing multiple drives, partitions and volumes.

First some definitions:

- A drive consists of a physical media which is controlled by a single driver. Examples are a HDD or a Compact Flash Card
- All drives contain zero or more partitions – if the drive is not partitioned then there is just a single volume on that drive. Normally for removable media such as flash cards there are zero or one partitions on the card.
- On each partition may be added a single volume. A volume can exist on a drive without partitions

The file system operates on volumes – all additional functions are provided to make the volumes on the different drives and partitions appear as a set of volumes. i.e. A:, B: etc.

> **Note**: the API function calls *f_getdrive(), f_chdrive()* and *f_getdcwd()* refer to drive by name, because this is the convention, but are really references to volumes.

If the developer does not require partitions to be created or deleted then the *f_initvolumepartition()*, *f_createdriver()*, *f_releasedriver()* and *f_createpartition()* functions should be left out of the system.

If multiple partitions are to be used then the developer should use these four functions to create drivers for partitioned drives and to create partitions on those drives.

Partitions are created on a single volume, like on a HDD, and 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 will not recognize multiple partitions on a removable media. It is "normal" to restrict the use of multiple partitions to fixed drives. FAT created partitions are Windows XP compatible.

2.7. Long Filenames

The system includes two main source files to choose between:

fat.c - contains 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 - contains file system with complete long filename support.

The long filename is optional because of the increase in system resources required to do long filenames. In particular the stack sizes of applications which call the file system must be increased and the amount of checking required is increased.

To choose between using the long filename version and the short use the

```
    F_LONGFILENAME definition in udefs.h.
```

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. The developer may, depending on their application, reduce the size of F_MAXPATH and F_MAXLNAME (in **udefs.h**) to reduce the resource usage of the system. The structure F_LFNINT must NOT be modified as this is used to process the files on the media which may be created by other systems.

The most critical function for long filenames is the ***fn_rename*** function which must keep two long filenames on the stack and additional structures for handling it. If this function is not required for your application it is sensible to comment it out and this can significantly reduce the stack requirements (by approximately 1K).

2.8. Maximum Number of Volumes and Reentrancy

The maximum number of volumes allowed by your system should be set in the F_MAXVOLUME definition in **udefs.h**. Set this value to the maximum volumes that will be available on the target system. (E.g. if only RAM drive is used set the value to 1, if RAM drive and CF card drive then set this value to 2, etc).

Volumes are given drive letters as specified in the ***f_initvolume()*** function.

The system is designed such that access to each volume is entirely independent i.e. if an operation is being performed on a volume then it does not block access to other volumes.

If your system has only a single task which accesses the file system then no changes to **port.c** are required.

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 **port.c**. These are general functions and should be replaced by the routines provided by your operating system.

> **Note**: The mutex routines supplied with the system are vulnerable to the classic priority inversion problem which can only be resolved by the use of routines specific to the target's RTOS.

2.9. Mutex Functions

If reentrancy is required as described in the previous section then the following functions in **port.c** must be implemented – normally provided by the host RTOS:

f_mutex_create() – called at volume initialization
f_mutex_delete() – called at volume deletion
f_mutex_get() – called when a mutex is required
f_mutex_put() – called when the mutex is released

> **Note**: If the CAPI is used (i.e. F_CAPI_USED is defined in **udefs.h**) then these mutex functions will be replaced by those of the CAPI. Consult the CAPI guide for further information

2.10. Maximum Open Files

The maximum number of simultaneously open files allowed must be specified in the **udefs.h** file. This is set in the F_MAXFILES definition. This is the total number of files that may be simultaneously open across all volumes.

2.11. Maximum Tasks and CWD

If more than a single task is allowed to access the file system then reentrancy and maintenance of the current working directory must be considered.

Reentrancy is handled on a per volume basis and is documented in the sections above.

Within the standard API there is no support for the current working directory to be maintained on a per caller basis. By default the system provides a single **cwd** which can be changed by any user. This is maintained on a per volume basis.

An additional option has been provided which enables the file system to keep track of the **cwd** on a per calling task basis. To use this option the developer must take the following steps:

1. Set **`F_MAXTASK`** to the maximum number of tasks that can simultaneously maintain access to the file system. This effectively creates a table of cwds for each task.
2. Modify the function `fn_gettaskID()` in the **port.c** file 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 will be logged as it acquires the semaphore, and a current working directory will be associated with it. The caller must release this when it has finished using the file system e.g. when the calling task is terminated. This frees the entry for other tasks to use.

> **Note**: If the CAPI is used (i.e. F_CAPI_USED is defined in **udefs.h**) then the `fn_gettaskID()` function will be replaced by that in the CAPI. Consult the CAPI guide for further information

2.12. Cache Setup and Options

The system includes two caching mechanisms to enhance the performance of the system; these are FAT caching and write data caching.

2.12.1. FAT Caching

FAT caching enables the file system to read several sectors from the FAT in one access so that when accessing the files the file system does not have to read new FAT sectors so frequently. The FAT caching is arranged in blocks such that each block can cover different areas of the FAT. The number of sectors that each block contains and the number of blocks is configurable.

FAT caching requires additional RAM – 512 bytes per sector.

The following definitions are provided in **udefs.h**

```
#define FATCACHE_ENABLE

#ifdef FATCACHE_ENABLE
#define FATCACHE_BLOCKS 4 /*number of different FAT cache blocks*/
#define FATCACHE_READAHEAD 8  /* number of FAT sectors to read */
                              /* to a block */
#define FATCACHE_SIZE (FATCACHE_BLOCKS*FATCACHE_READAHEAD)
#endif
```

Note: The additional RAM required for FAT caching is:

```
FATCACHE_BLOCKS*FATCACHE_READAHEAD*512
```

This default setting requires 16K of additional RAM.

2.12.2. Write Caching

The write cache defines the maximum number of sectors which can be written in one operation from the caller's data buffer. This is also dependent on there being contiguous space available on the target drive. The write cache requires an F_POS structure (24 bytes) for each entry in the write cache. The main purpose of these structures is to be able to wind back a write in the event of an error in writing.

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

```
#define WR_DATACACHE_SIZE 32
```

This will require 768 additional bytes of RAM.

2.12.3. Directory Cache

This can only be enabled if F_LONGFILENAME is defined. This can be enabled by defining DIRCACHE_ENABLE in **udefs.h**. If this is enabled you must specify the number of sectors to read ahead with DIR_CACHESIZE. This will allocate this number of sectors of memory for directory caching (e.g. if set to 32; 16Kbytes of memory will be allocated). Note also that the system will never read more than the size of a cluster into this cache – therefore if there is no value in having a DIR_CACHESIZE greater than the sectors per cluster of the target device.

2.13. Fat Free Cluster Bit Field

In the **udefs.h** there is a FATBITFIELD_ENABLE definition. If this is enabled then the system will attempt to *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. This makes a large difference to the write performance when writing to a large and full disk.

2.14. Memcpy and Memset

Supplied with the system are ***memcpy*** and ***memset*** functions.

It is recommended to re-define these to call versions of these functions that are optimized for your target system. As with all embedded systems, these routines are used frequently and take time and having a good ***memcpy*** routine can have a large impact on the overall performance of your system.

The following has been defined in **udefs.h** and should be modified to call target optimized versions of these functions:

```
#ifdef INTERNAL_MEMFN
#define _memcpy(d,s,l) _f_memcpy(d,s,l)
#define _memset(d,c,l) _f_memset(d,c,l)
#else
#include <string.h>
#define _memcpy(d,s,l) memcpy(d,s,l)
#define _memset(d,c,l) memset(d,c,l)
#endif
```

2.15. Malloc and Free

In udefs.h, _malloc and _free functions are predefined. They only exist when USE_MALLOC is defined and in this case they are pointed to original library functions malloc and free. If the application wants to use its separated memory management routines then set _malloc and _free to point to them.

```
#define USE_MALLOC

#ifdef USE_MALLOC
#define _malloc(x) malloc(x)        /*  normally  use  malloc  from
library */
#define _free(x) free(x)          /*   normally   use   free   from
library */
#endif
```

2.16. Get Time

For the system to be compatible with other systems it is necessary to provide a real time function so that files can be time-stamped.

An empty function (***f_gettime***) is provided in **port.c** which should be modified by the developer to provide the time in standard format.

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

2-second increments	(0-30 valid)	(t & 0x001f)
minute	(0-59 valid)	((t & 0x07e0) >> 5)
hour	(0-23 valid)	((t & 0xf800) >> 11)

2.17. Get Date

For the system to be compatible with other systems it is necessary to provide a real time function so that files can be date-stamped.

An empty function (***f_getdate***) is provided in **port.c** which should be modified by the developer to provide the date in standard format.

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

day	(0-31)	(d & 0x001f)
month	(1-12 valid)	((d & 0x01e0) >> 5)
years since 1980	(0-119 valid)	((d & 0xfe00) >> 9)

2.18. Last accessed date

In udefs.h there is a #define for using auto update last accessed time field in directory entry on read file. Set F_UPDATELASTACCESSDATE to 1 if you want to allow this option, in this case whenever you open a file for read ("r"), then a sector write will happen on directory entry which updates the last accessed date (date is checked before updating to ensure it needs updating). To avoid this option (which saves unnecessary sector writes) set F_UPDATELASTACCESSDATE to 0. In this case only other file manipulations ("r+","w","w+","a","a+") change this date entry.

2.19. Random Number

The **port.c** file contains a function (***f_getrand***) which the file system uses to get a pseudo-random number to use as the volume serial number. This function is only required if a hard-format of devices is required.

It is recommended that the developer replace this routine with a random function from their base system or alternatively generate their own random number based on a combination of the system time/date and a system constant such as a MAC address.

2.20. Separator Character

The **udefs.h** file contains a definition `F_SEPARATORCHAR` which allows the developer to select whether the forward or back slash character is used as a separator in file paths.

2.21. Fast Seeking

The developer can define a number of points in a file to use as markers to allow fast seeking in a file. The F_MAXSEEKPOS definition in **udefs.h** sets a number of points to be stored with every file descriptor. Setting this to zero will mean that seeking will always work from the current position or the beginning of the file only. F_MAXSEEKPOS should only take power of 2 values or zero.

> **Note:** The memory usage of the system is increased by:
>
> `F_MAXSEEKPOS*F_MAXFILES*sizeof(long)`

3. Drive Format

This document does not describe a FAT file system in detail - there are many reference works to choose from. This file system handles the majority of the features of a FAT file system with no need for the developer to understand further. However, there are some areas where an understanding may help - this section describes these features and provides additional information about FAT formats.

There are three different forms in which your removable media maybe formatted with:

- Completely Unformatted Media
- Master Boot Record
- Boot sector Information only

The sections below describe how the system handles these three situations.

3.1. Completely unformatted

If a drive is completely unformatted then it 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 format of the card is determined by the number of sectors on it. Information about the connected device is given to the system from the ***xxx_getphy*** call to the driver from which the number of available clusters on the device is calculated.

When the ***f_format*** function is called the drive will be formatted with Boot Record Information or ***xxx_getphy***.

If more partition requested to be created then use ***f_createpartition***, ***f_initvolumepartition*** and ***f_format*** function for formatting.

3.2. Master Boot Record

If a card contains a Master Boot Record it is formatted as in the tables below. Function f_createpartition also can create MBR.

When a device is inserted with an MBR it will be treated as if it just has one partition (the first in the partition table if f_initvolume is used. Multiple partitions can be initially by f_initvolumepartition function.

Offset	Bytes	Entry Description	Value/Range
0x0	446	Consistency check routine	
0x1be	16	Partition table entry	(table below)
0x1ce	16	Partition table entry	(table below)
0x1de	16	Partition table entry	(table below)
0x1ee	16	Partition table entry	(table below)
0x1fe	1	Signature	0x55
0x1fe	1	Signature	0xaa

Table 1, Master Boot Record

Offset	Bytes	Entry Description	Value/Range
0x0	1	Boot descriptor	0x00 (non-bootable device) 0x80 (bootable device)
0x1	3	First partition sector	Address of first sector
0x4	1	File system descriptor	0 = empty 1 = FAT12 4 = FAT16 < 32MB 5 = Extended DOS 6 = FAT16 >= 32MB 0xB=FAT32 0x10-0xff free
0x5	3	Last partition sector	Address of last sector
0x8	4	First sector position relative to device start	First sector number
0xc	4	Number of sectors in partition	Between 1 and max number on device

Table 2, Partition Entry Description

3.3. Boot Sector information

This is the system used as standard by the file system. The first 36 bytes of the boot sector are the same for FAT12/16/32 as in the first table. The second table shows the format for the rest of the boot sector for FAT12/16. The third table shows the format of the boot sector for FAT32.

Offset	Bytes	Entry Description	Value/Range
0x0	3	Jump Command	0xeb 0xXX 0x90
0x3	8	OEM Name	XXX: specify in udefs.h
0xb	2	Bytes/Sector	512
0xd	1	Sectors/Cluster	XXX(1-64)
0xe	2	Reserved Sectors	1
0x10	1	Number of FATs	2
0x11	2	Number of root directory entries	512
0x13	2	Number of sectors on media	XXX (dependent on card size, if greater than 65535 then 0 and number of total sectors is used)
0x15	1	Media Descriptor	0xf8 (hard disk) 0xf0 (removable media)
0x16	2	Sectors/FAT16	XXX (normally 2). This must be zero for FAT32.
0x18	2	Sectors/Track	32 (not relevant)
0x1a	2	Number of heads	2 (not relevant)
0x1c	4	Number of hidden sectors	0 or if MBR present number relative sector offset of this sector.
0x20	4	Number of total sectors	XXX (depends on card size) or 0

Table 3, Boot Sector Information
Table First 36 bytes

Offset	Bytes	Entry Description	Value/Range
0x24	1	Drive Number	0
0x25	1	Reserved	0
0x26	1	Extended boot signature	0x29
0x27	4	Volume ID or Serial Number	Random number generated at format
0x2b	11	Volume Label	"NO LABEL" is put here by a format
0x36	8	File System type	“FAT16” or "FAT12"
0x3e	448	Load Program Code	Filled with zeroes.
0x1fe	1	Signature	0x55
0x1ff	1	Signature	0xaa

Table 4, Boot Sector Information Table
FAT12/16 after byte 36

Note: The serial number field is generated by the random number function – see porting section for information about its generation.

Offset	Bytes	Entry Description	Value/Range

0x24	4	Sectors/FAT32	The number of sectors in one FAT
0x28	2	ExtFlags	Always zero.
0x2a	2	File System Version	0 0
0x2c	4	Root Cluster	Cluster number of the first cluster of the root directory
0x30	2	File System Info	Sector number of FSINFO structure in the reserved area of the FAT32. Usually 1.
0x32	2	Backup Boot Sector	If non-zero it indicates the sector number in the reserved area of the volume of a copy of the boot record. Usually 6.
0x34	12	Reserved	All bytes always zero
0x40	1	Drive Number	0
0x41	1	Reserved	0
0x42	1	Boot Signature	0x29
0x43	4	Volume ID	Random number generated at format.
0x47	11	Volume Label	"NO LABEL" is put here by a format
0x52	8	File System Type	Always set to string "FAT32 ".

Table 5, Boot Sector Information Table
FAT32 After byte 36

4. File API

4.1. File System Functions

General File System System functions

f_init
f_getversion
f_enterFS()
f_releaseFS()

Volume functions

f_initvolume
f_initvolumepartition
f_delvolume
f_checkvolume
f_get_volume_count
f_get_volume_list
f_format
f_createdriver
f_releasedriver
f_getfreespace
f_setlabel
f_getlabel
f_get_oem
f_createpartition
f_getpartition

Drive\Directory handler functions

f_getdrive
f_chdrive
f_getcwd
f_getdcwd
f_mkdir
f_chdir
f_rmdir

File functions

f_rename
f_move
f_delete
f_filelength
f_findfirst
f_findnext
f_settimedate
f_gettimedate
f_getattr
f_setattr
f_stat

Read/Write functions

f_open
f_close
f_write
f_read
f_seek
f_tell
f_eof
f_seteof
f_rewind
f_putc
f_getc
f_truncate
f_flush
f_ftruncate

4.2. Function Error Codes

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 renamed or deleted is not empty.
F_ERR_INITFUNC	15	If no init function 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	The media is not recognized
F_ERR_BUSY	22	The caller could not obtain the semaphore within the expiry time
F_ERR_WRITEPROTECT	23	The physical media 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_RESERVED_1	31	Reserved
F_ERR_DELFUNC	32	The delete drive driver function failed
F_ERR_ALLOCATION	33	Malloc failed to allocate 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 callers task identifier was not registered – normally because the f_enterfs() function 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,

Table 6, Error Codes

4.3. f_getversion

This function is used to retrieve file system version information.

Format

```
char * f_getversion(void)
```

Arguments

None

Return values

Return value	Description
Any	pointer to null terminated ASCII string

Example

```
void display_fs_version(void)
{
   printf("File System Version: %s",f_getversion());
}
```

4.4. f_init

This function should be called once at startup to initialize the file system.

The developer can insert code into this function if there are any special requirements for a particular target system. Function initiates internal variables.

Format

```
int f_init(void)
```

Arguments

None

Return values

Return value	Description
F_NO_ERROR	drive successfully initialized
else	failed - see error codes

Example

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

4.5. f_enterFS

If the target system allows multiple tasks to use the file system then this function must be called by a task before using any other file API functions. This function creates resource for the calling task in the file system and allocates a current working directory for that task.

The *f_releaseFS()* call must be made to release the task from the file system and free the allocated resource..

The correct operation of this function also requires that the *fn_gettaskID()* in **port_f.c** has been ported to give a unique identifier for each task.

Format

```
int f_enterFS(void)
```

Arguments

Argument	Description

Return values

Return value	Description
0	Success
Non-zero	Error Code

4.6. f_releaseFS

This function is called by the user to release a previously assigned unique task ID used to track the calling task's current working directory.

The unique task identifier is that generated by *fn_gettaskID()* in **port_f.c**

Format

```
void f_releaseFS(long ID)
```

Arguments

Argument	Description
ID	unique identifier for calling task

Return values

Return value	Description
none	

4.7. f_initvolume

This function is used to initialize a volume. The function is called 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 i.e. it does not matter if a card is inserted or not.

Function *f_initvolume* always initiates the 1st partition on the media. To use multiple partitions then use the *f_initvolumepartition* function.

Format

```
int f_initvolume(int drivenum, F_DRIVERINIT *driver_init, unsigned
      long driver_param)
```

Arguments

Argument	Description
drivenum	drive to be initialized (0:A, 1:B...)
driver_init	pointer to initialization function for driver
driver_param	driver parameter (see below)

Return values

Return value	Description
F_NO_ERROR	drive successfully initialized
else	failed - see error codes

Note: The **driver_param** can be used to pass information to the low-level driver. When the ***xxx_initfunc*** of the driver is called this parameter will be passed to the driver. The usage of this parameter is optional and driver dependent. One use is to specify which device associated with the specified driver will be initialized. For convenience a definition F_AUTO_ASSIGN has been predefined to mean that the driver should assign devices as it wishes – this convention is optional and has no affect on the file system.

For more information about its usage please see Section Driver Interface.

Example

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

   f_init();

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

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

   /*Make an MMC Volume on Drive C */
   f_initvolume(2, f_mmcinit, F_AUTO_ASSIGN);

         .
         .
         .
}
```

See also

`f_format, f_initvolumepartition, f_checkvolume`

4.8. f_initvolumepartition

This function is used to initialize a volume on an existing partition. The function is called with a pointer to the function that must be called to retrieve drive configuration information. This function requires the target drive to be connected.

If only the 1st partition is used on a media then *f_initvolume()* should be used.

Format

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

Arguments

Argument	Description
drivenum	drive to be initialized (0:A, 1:B...)
driver	initialized driver (get from f_createdriver)
partition	which partition is requested to be built

Return values

Return value	Description
F_NO_ERROR	drive successfully initialized
else	failed - see error codes

Example

```
F_DRIVER *hdd;

int myinitfs(void)
{
   int ret;

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

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

   ret=f_initvolumepartition(1,hdd,1);

   return ret;
}
```

See also

```
f_format, f_initvolume, f_createdriver
```

4.9. f_createdriver

This function is used to initialize a driver. The function is called 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 i.e. it does not matter if a card is inserted or not.

This function is only necessary if multiple partitions on a drive are used.

If *f_initvolume()* is used to initiate a volume, then *f_createdriver()* is not required as it is called automatically.

On a drive which was created directly with the *f_createdriver()* function then *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
driver	driver ptr, where to set up driver pointer
driver_init	pointer to initialization function for driver
driver_param	driver parameter (see below)

Return values

Return value	Description
F_NO_ERROR	drive successfully initialized
else	failed - see error codes

Note: The **driver_param** can be used to pass information to the low-level driver. When the ***xxx_initfunc*** of the driver is called this parameter will be passed to the driver. The usage of this parameter is optional and driver dependent. One use is to specify which device associated with the specified driver will be initialized. For convenience a definition F_AUTO_ASSIGN has been predefined to mean that the driver should assign devices as it wishes – this convention is optional and has no affect on the file system.

For more information about its usage please see Section Driver Interface.

Example

```
F_DRIVER *hdd;

int myinitfs(void)
{
   int ret;

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

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

   ret=f_initvolumepartition(1,hdd,1);

   return ret;
}
```

See also

f_format, f_initvolumepartition, f_createpartition,
f_releasedriver, f_delvolume

4.10. f_releasedriver

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

If the driver was created by *f_initvolume()* then this function should not be called – *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, then *f_releasedriver()* should be called to release the driver.

If the driver was created by *f_createdriver()* and *f_releasedriver()* is called then *f_delvolume()* will be called automatically for each volume on this drive.

Format

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

Arguments

Argument	Description
driver	initialized driver (get from f_createdriver)

Return values

Return value	Description
F_NO_ERROR	drive successfully initialized
else	failed - see error codes

Example:

```
F_DRIVER *hdd;

int myinitfs(void)
{
   int ret;

   ret=f_createdriver(&hdd,f_hdddrvinit,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);
}
```

See also

f_format, f_initvolumepartition, f_createdriver

4.11. f_createpartition

This function is used to create 1 or more partitions on a drive. This function is called 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.

If only a single volume is required then it is simpler not to use a partition table and use *f_initvolume()* to format.

Calling this function will logically destroy 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
driver	initialized driver (get from f_createdriver)
parnum	number of partition is in par ptr
par	partition pointer points to partition descriptor

Return values

Return value	Description
F_NO_ERROR	drive successfully initialized
else	failed - see error codes

Note: 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;
```

In this descriptor secnum is the number of sector in the partition, system_indicator value is depending on the format it will be used on the partition. See F_SYSIND_xx values in fat.h.

This function works similarly as the MS-DOS Fdisk function.

Example:

```
static F_PARTITION par2[2]=
{
   {1000, F_SYSIND_DOSFAT16UPTO32MB},
   {2000, F_SYSIND_DOSFAT16UPTO32MB}
};

F_DRIVER *hdd;

int mypartitiondrive()
{
   int ret;

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

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

   return ret;
}
```

See also

f_format, f_initvolumepartition, f_createdriver, f_getpartition

4.12. f_getpartition

This function is used to get the used sectors and system indication byte from a portioned media.

For drives which do not contain a partition table, then this function returns with the number of sectors and 0 in the system indication byte.
If there is a partition table, then it collects information from the partition table entries. If there is not enough space in the passed F_PARTITION table, then it signals F_ERR_MEDIATOOLARGE error. In this case media has more partition table entries than number of entries passed F_PARTITION table structure, so the caller should increase the number of entries in this table.

Format

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

Arguments

Argument	Description
driver	initialized driver (get from f_createdriver)
parnum	number of entry in the par parameter
par	partition pointer to retrieve information inside

Return values

Return value	Description
F_NO_ERROR	drive successfully initialized
else	failed - see error codes

Note: 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;
}
```

See also

f_format, f_initvolumepartition, f_createdriver, f_createpartition

4.13. f_delvolume

This function is used to delete an existing volume. The link between the file system and the driver will be broken i.e. an ***xxx_release*** call will be made to the driver. Any open files on the media will be marked as closed so that subsequent API accesses to a previously opened file handle will return with an error. If the volume's driver was created independently with f_createdriver, then this function deletes only the volume and f_release function is needed to be called for calling ***xxx_release*** driver functions.

This function works independently of the status of the hardware i.e. it does not matter if a card is inserted or not.

Format

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

Arguments

Argument	Description
drivenum	drive to be deleted (0:A, 1:B...)

Return values

Return value	Description
F_NO_ERROR	drive successfully deleted
else	failed - see error codes

Example:

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

   /*Delete volume 1 */
   if(f_delvolume(num))
      printf("Unable to delete volume %d", num);
         .
         .
         .
}
```

See also

```
f_initvolume
```

4.14. f_checkvolume

This function is used to check the status of a drive which has been initialized.

Format

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

Arguments

Argument	Description
drivenum	drive to be checked (0:A, 1:B...)

Return values

Return value	Description
F_NO_ERROR	drive is working
else	there is an error on the drive e.g. card missing

Example

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

   /*Delete volume 1 */
   if(f_checkvolume(num))
   {
      printf(“Volume %d is not usable, Error %d”, num, ret);
   }
   else
   {
      printf((“Volume %d is working”, num);
   }
      .
      .
}
```

See also

```
f_initvolume, f_delvolume
```

4.15. f_get_volume_count

This function returns the number of volumes currently available to the user.

Format

```
int f_get_volume_count(void)
```

Arguments

Argument	Description
none	

Return values

Return value	Description
num	number of active volumes

Example

```
void mygetvols(void)
{
   printf(“there are %d active volumes\n”,
      f_get_volume_count());
         .
         .
}
```

See also

```
f_get_volune_list
```

4.16. f_get_volume_list

This function returns a list of volumes currently available to the user.

Format

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

Arguments

Argument	Description
none	

Return values

Return value	Description
number	number of active volumes

Example:

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

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

See also

f_get_volume_count

4.17. f_format

It formats the specified drive. If the media is not present this routine will fail. If successful all data on the specified volume will be destroyed. Any open files will be closed.

Any existing Master Boot Record will be unaffected by this command. The boot sector information will be re-created from the information provided by f_getphy() (see Section 3 Drive Format).

The caller must specify the required format:

```
F_FAT12_MEDIA  for FAT12
F_FAT16_MEDIA  for FAT16
F_FAT32_MEDIA  for FAT32
```

The format will fail 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
drivenum	drive to be formatted (0=”A”…)
fattype	type of format: FAT12, FAT16 or FAT32

Return values

Return value	Description
F_NO_ERROR	drive successfully formatted
else	format failed - see error codes

Note: 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 on the target device below which the second number gives the number of sectors per cluster. This table may be modified if required.

```
static const t_FAT32_CS FAT32_CS[]=
{
      { 0x00020000, 1 },   /* ->64MB */
      { 0x00040000, 2 },   /* ->128MB */
      { 0x00080000, 4 },   /* ->256MB */
      { 0x01000000, 8 },   /* ->8GB */
      { 0x02000000, 16 },  /* ->16GB */
      { 0x0ffffff0, 32 }   /* -> ... */
};
```

Example:

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

   f_initvolume(0,f_cfcinit, F_AUTO_ASSIGN);

   ret=f_format(0, F_FAT16_MEDIA);

   if(ret)
      printf(“Unable to format CFC: Error %d”,ret);
   else
      printf(“CFC formatted”);

         .
         .
}
```

See also

```
 f_initvolume, f_format
```

4.18. f_getfreespace

This function fills a structure with information about the drive space usage - total space, free space, used space and bad (damaged) size.

> **Note:** If a drive size of greater than 4GB is being used then the high elements of the returned structure should also be read to get the upper 32 bits of each of the numbers i.e pspace.total_high etc.
>
> **Note:** The first call to this function after a drive is mounted may take some time depending on the size and format of the disk being used. After the initial call changes to the volume are counted – the function then returns immediately with this data.

Format

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

Arguments

Argument	Description
drivenum	drive number
pspace	pointer to F_SPACE structure

Return values

Return value	Description
F_NO_ERROR	no error
else	error code

Example

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

   /* get free space on current drive */
   int ret = f_getfreespace(f_getcurrdrive(),&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);
   }
}
```

4.19. f_setlabel

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

Format

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

Arguments

Argument	Description
drivenum	drive number
pLabel	pointer to null terminated string to use

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

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

   if (result)
   printf("Error on Drive");
}
```

4.20. f_getlabel

This returns the label to a function. The pointer passed for storage should be capable of holding an 11 character string.

Format

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

Arguments

Argument	Description
drivenum	drive number
pLabel	pointer to copy label to
len	length of storage area

Return values

Return value	Description
F_NOERROR	success
else	(see error codes table)

Example

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

   result = f_getlabel(f_getcurrdrive(),label,12);

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

4.21. f_get_oem

This returns the OEM name in the disk bootrecord. The pointer passed for storage should be capable of holding an 8 character long string.

Format

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

Arguments

Argument	Description
drivenum	drive number
str	pointer to copy label to
len	length of storage area

Return values

Return value	Description
F_NOERROR	success
else	(see error codes table)

Example

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

   oem_name[8]=0; /* zero terminate string */
   result = f_get_oem(f_getcurrdrive(),oem_name,8);

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

4.22. f_mkdir

Makes a new directory.

Format

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

Arguments

Argument	Description
dirname	new directory name to create

Return values

Return value	Description
F_NO_ERROR	new directory name created successfully
else	(see error codes table)

Example

```
void myfunc(void)
{
      .
      .
   f_mkdir(“subfolder”);   /*creating directory */
   f_mkdir(“subfolder/sub1”);
   f_mkdir(“subfolder/sub2”);
   f_mkdir(“a:/subfolder/sub3”
      .
      .
}
```

See also

```
f_chdir, f_rmdir
```

4.23. f_chdir

Change directory

Format

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

Arguments

Argument	Description
dirname	directory to change to

Return values

Return value	Description
F_NO_ERROR	directory has been change successfully
else	(see error codes table)

Example

```
void myfunc(void)
{
      .
      .
   f_mkdir(“subfolder”);
   f_chdir(“subfolder”); /*change directory */   f_mkdir(“sub2”);
   f_chdir(“..”);    /*go to upward */
   f_chdir(“subfolder/sub2”); /*goto into sub2 dir */
      .
      .
}
```

See also

```
f_mkdir, f_rmdir, f_getcwd, f_getdcwd
```

4.24. f_rmdir

Remove a directory. The target directory must be empty when this is called; otherwise it returns an error code.

If a directory is read-only then this function returns an error code.

Format

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

Arguments

Argument	Description
dirname	name of directory to remove

Return values

Return value	Description
F_NO_ERROR	directory name is removed successfully
else	(see error codes table)

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 */
      .
      .
}
```

See also

```
f_mkdir, f_chdir
```

4.25. f_getdrive

Get current drive number

Format

```
int f_getdrive(void)
```

Arguments

none

Return values

Return value	Description
Current Drive	0-A, 1-B, 2-C etc

Example

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

See also

```
f_chdrive
```

4.26. f_chdrive

Change to a new current drive.

Format

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

Arguments

Argument	Description
drivenum	drive number to change to (0-A,1-B,2-C,…)

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

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

See also

```
f_getdrive
```

4.27. f_getcwd

Get current working directory on current drive.

Format

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

Arguments

Argument	Description
buffer	where to store current working directory string
maxlen	length of the buffer

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

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”)
   }
}
```

See also

```
f_chdir, f_getdcwd
```

4.28. f_getdcwd

Get current working folder on selected drive.

Format

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

Arguments

Argument	Description
drivenum	specify drive (0-A, 1-B, 2-C)
buffer	where to store current working directory string
maxlen	length of the buffer

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

```
#define BUFFLEN F_MAXPATH+F_MAXNAME

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

   if (!f_getcwd(drivenum,buffer, BUFFLEN))
   {
      printf (“current directory is %s”,buffer);
      printf (“on drive %c”,drivenum+’A’);
   }
   else
   {
      printf (“Drive Error”)
   }
}
```

See also

```
f_chdir, f_getcwd
```

4.29. f_rename

Renames a file or directory. This function has been obsoleted by ***f_move***.

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

Format

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

Arguments

Argument	Description
filename	file or directory name with/without path
newname	new name of target file or directory (without path)

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

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

See also

```
f_mkdir, f_open
```

4.30. f_move

Moves a file or directory – the original is lost. This function obsoletes ***f_rename()***. The source and target must be in the same volume.

Format

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

Arguments

Argument	Description
filename	file or directory name with/without path
newname	new name of file or directory with/without path

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

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

See also

```
f_mkdir, f_open, f_rename
```

4.31. f_delete

Deletes a file.

A read-only or open file cannot be deleted.

Format

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

Arguments

Argument	Description
filename	file name with or without path to be deleted

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

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

See also

```
f_open
```

4.32. f_filelength

Get the length of a file. If the requested file does not exist or has any error then this function returns with -1.

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

Format

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

Arguments

Argument	Description
filename	file name with or without path

Return values

Return value	Description
filelength	length of file
-1	if any 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;
}
```

See also

```
f_open
```

4.33. f_findfirst

Find first file or subdirectory in specified directory. First call ***f_findfirst*** function and if file was found get the next file with ***f_findnext*** function.
Files with the system attribute set will be ignored.

> **Note:** If this is called with "*.*" and this is not the root directory the first entry found will be "." - the current directory.

Format

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

Arguments

Argument	Description
filename	name of file to find
find	where to store find information

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

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.len);
         }
      } while (!f_findnext(&find));
   }
}
```

See also

```
f_findnext
```

4.34. f_findnext

Finds the next file or subdirectory in a specified directory after a previous call to ***f_findfirst*** or ***f_findnext***. First call ***f_findfirst*** function and if file was found get the rest of the matching files by repeated calls to the ***f_findnext*** function.
Files with the system attribute set will be ignored.

> **Note:** If this is called with "*.*" and it is not the root directory the first file found will be ".." - the parent directory.

Format

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

Arguments

Argument	Description
find	find information (created by ***f_findfirst*** call)

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

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.len);
         }
      } while (!f_findnext(&find));
   }
}
```

See also

f_findfirst

4.35. f_stat

Get information about a file. This function retrieves information by filling the F_STAT structure passed to it. It sets filesize, creation time/date, last access date, modified time/date, and the drive number where the file is located.

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

Format

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

Arguments

Argument	Description
filename	file
stat	pointer to F_STAT structure to be filled

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

```
void myfunc(void)
{
   F_STAT stat;
   if (f_stat(“myfile.txt”,&stat))
   {
      printf (“error”);
      return;
   }
   printf (“filesize:%d”,stat.filesize);
}
```

See also

```
f_gettimedate, f_settimedate
```

4.36. f_settimedate

Set the time and date of a file or directory. (See Section 2 Porting – Step by Step Guide for further information about porting).

Format

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

Arguments

Argument	Description
filename	file
ctime	creation time of file or directory
cdate	creation date of file or directory

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

```
void myfunc(void)
{
   f_mkdir(“subfolder”);   /*creating directory */

   f_settimedate(“subfolder”,f_gettime(),f_getdate());
}
```

See also

```
f_gettimedate, f_stat
```

4.37. f_gettimedate

Get time and date information from a file or directory. (See Section 2 Porting – Step by Step Guide for more information about porting).

Format

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

Arguments

Argument	Description
filename	target file
pctime	pointer to where to store creation time
pcdate	pointer to where to store creation date

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

```
void myfunc(void)
{
   unsigned short t,d;
   if (!f_gettimedate(“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 & ox01e0) >> 5);
      unsigned short year=1980+((d & 0xf800) >> 9);

      printf (“Time: %d:%d:%d”,hour,minute,sec);
      printf (“Date: %d.%d.%d”,year,month,day);
   }
   else
   {
      printf (“File time cannot retrieved!”
   }
}
```

See also

```
f_settimedate, f_stat
```

4.38. f_setattr

This routine is used to set the attributes of a file. Possible file attribute settings are defined by the FAT file system:

F_ATTR_ARC	Archive
F_ATTR_DIR	Directory
F_ATTR_VOLUME	Volume
F_ATTR_SYSTEM	System
F_ATTR_HIDDEN	Hidden
F_ATTR_READONLY	Read Only

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

Format

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

Arguments

Argument	Description
filename	target file
attr	new attribute setting

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

```
void myfunc(void)
{

   /* make myfile read only and hidden */

   f_setattr("myfile.txt", F_ATTR_READONLY | F_ATTR_HIDDEN);
}
```

4.39. f_getattr

This routine is used to get the attributes of a specified file. Possible file attribute settings are defined by the FAT file system:

F_ATTR_ARC	Archive
F_ATTR_DIR	Directory
F_ATTR_VOLUME	Volume
F_ATTR_SYSTEM	System
F_ATTR_HIDDEN	Hidden
F_ATTR_READONLY	Read Only

Format

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

Arguments

Argument	Description
filename	target file
attr	pointer to place attribute setting

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

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

   /* find if myfile 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");
   }
}
```

4.40. f_open

Opens a file. The following modes are allowed to open:

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.

Table 7, f_open modes

The same file can be opened multiple times in “r” mode.

If a file is opened in “w” or “w+” mode then there is a lock mechanism which denies opening file for in any other mode. This prevents the file to be opened for reading and writing at the same time.

If a file is open in “a” or “a+” mode, then any number of “r” mode opens are allowed 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_open(const char *filename, const char *mode);
```

Arguments

Argument	Description
filename	file to be opened
mode	mode to open file with

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);
}
```

See also

f_read, f_write, f_close,

4.41. f_close

Close a previously opened file.

Format

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

Arguments

Argument	Description
filehandle	handle of target file

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

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”);
}
```

See also

```
f_open, f_read, f_write
```

4.42. f_flush

Flushes an open file to disk. This is logically equivalent to doing a close and open on a file to ensure the data changed before the flush is committed to the disk.

Format

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

Arguments

Argument	Description
filehandle	handle of target file

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

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

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

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

See also

```
f_open, f_close
```

4.43. f_write

Write data to file at current stream position. File has to be opened with “w”, “w+”, "a+", "r+" or “a”.

Format

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

Arguments

Argument	Description
buf	pointer to data to be written
size	size of items to be written
size_st	number of items to be written
filehandle	handle of target file

Return values

Return value	Description
number	number of items written

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);
}
```

See also

```
f_read, f_open, f_close
```

4.44. f_read

Read bytes from the current position in the target file. File has to 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
buf	buffer where to store data
size	size of items to be read
size_st	number of items to be read
filehandle	handle of target file

Return values

Return value	Description
number	number of items read

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 (f_read(buffer,1,size,file)!=size)
   {
      printf ("not all items read!!");
   }
   f_close(file);
   return 0;
}
```

See also

```
f_seek, f_tell, f_open, f_close, f_write
```

4.45. f_seek

Move stream position in the target file. The file must be open.

The **Whence** parameter could be one of:

```
      F_SEEK_CUR - Current position of file pointer
      F_SEEK_END - End of file
      F_SEEK_SET - Beginning of file
```

offset position is relative to whence.

Format

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

Arguments

Argument	Description
filehandle	handle of open target file
offset	relative byte position according to whence
whence	where to calculate offset from

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

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;
}
```

See also

```
f_read, f_tell
```

4.46. f_tell

Tells the current read-write position in the open target file.

Format

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

Arguments

Argument	Description
filehandle	handle of open target 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;
}
```

See also

```
f_seek, f_read, f_write, f_open
```

4.47. f_eof

Check whether the current position in the open target file is the end of the file.

Format

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

Arguments

Argument	Description
filehandle	handle of open target file

Return values

Return value	Description
0	not at end of file
else	end of file or any error

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;
}
```

See also

```
f_seek, f_read, f_write, f_open
```

4.48. f_seteof

Move the end of file to the current file pointer. All data after the new EOF position is lost.

Format

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

Arguments

Argument	Description
filehandle	handle of open target file

Return values

Return value	Description
0	Success
else	Failed – 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;
}
```

See also

```
f_truncate, f_write, f_open
```

4.49. f_rewind

Sets the file position in the open target file to the start of the file.

Format

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

Arguments

Argument	Description
filehandle	handle of open target file

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

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;
}
```

See also

```
f_seek, f_read, f_write, f_open
```

4.50. f_putc

Writes 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
ch	character to be written
filehandle	handle of open target 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;
}
```

See also

```
f_seek, f_read, f_write, f_open
```

4.51. f_getc

Reads a character from the current position in the target open file.

Format

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

Arguments

Argument	Description
filehandle	handle of open target 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;
}
```

See also

```
f_seek, f_read, f_write, f_open, f_eof
```

4.52. f_truncate

Opens a file for writing and truncates it to the specified length. If the length is greater than the length of the existing file then the file is padded with zeroes to the truncated length.

Format

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

Arguments

Argument	Description
filename	file to be opened
length	new length of file

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 not found”);
   else
   {
      printf(“File %s truncated to %d bytes, filename, length);
      f_close(file);
   }
   return 0;
}
```

See also

```
f_open, f_ftruncate
```

4.53. f_ftruncate

If a file is opened for writing, then this function truncates it to the specified length. If the length is greater than the length of the existing file then the file is padded with zeroes to the truncated length.

Format

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

Arguments

Argument	Description
filehandle	open file handle
length	new length of file

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

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;
}
```

See also

```
f_open, f_truncate
```

4.54. f_getlasterror

It returns with the last error code. Last error code is cleared/changed when any API function is called.

Format

```
int f_getlasterror()
```

Arguments

none

Return values

Return value	Description
Error code	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;
}
```

See also

`f_open, f_filelength, f_read, f_write`

5. Unicode API

5.1. Unicode Specific File System Functions

When Unicode is enabled the following functions are available as well as their standard API equivalents. All functions are exactly as their standard API counterparts except that all character string parameters are changed to "wide character" (wchar) strings.

Drive\Directory handler functions

f_wgetcwd
f_wgetdcwd
f_wmkdir
f_wchdir
f_wrmdir

File functions

f_wrename
f_wmove
f_wdelete
f_wfilelength
f_wfindfirst
f_wfindnext
f_wsettimedate
f_wgettimedate
f_wgetattr
f_wsetattr
f_wstat

Read/Write functions

f_wopen
f_wtruncate

5.2. f_wmkdir

Makes a new directory.

Format

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

Arguments

Argument	Description
dirname	new directory name to create

Return values

Return value	Description
F_NO_ERROR	new directory name created successfully
else	(see error codes table)

Example

```
void myfunc(void)
{
      .
      .
   f_wmkdir("subfolder");  /*creating directory */
   f_wmkdir("subfolder/sub1");
   f_wmkdir("subfolder/sub2");
   f_wmkdir("a:/subfolder/sub3"
      .
      .
}
```

See also

```
f_wchdir, f_wrmdir
```

5.3. f_wchdir

Change directory

Format

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

Arguments

Argument	Description
dirname	directory to change to

Return values

Return value	Description
F_NO_ERROR	directory has been change successfully
else	(see error codes table)

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 */
      .
      .
}
```

See also

```
f_wmkdir, f_wrmdir, f_wgetcwd, f_wgetdcwd
```

5.4. f_wrmdir

Remove a directory. The target directory must be empty when this is called; otherwise it returns an error code.

If a directory is read-only then this function returns an error code.

Format

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

Arguments

Argument	Description
dirname	name of directory to remove

Return values

Return value	Description
F_NO_ERROR	directory name is removed successfully
else	(see error codes table)

Example

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

See also

```
f_wmkdir, f_wchdir
```

5.5. f_wgetcwd

Get current working directory on current drive.

Format

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

Arguments

Argument	Description
buffer	where to store current working directory string
maxlen	length of the buffer

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

```
#define BUFFLEN F_MAXPATH+F_MAXNAME

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

   if (!f_wgetcwd(buffer, BUFFLEN))
   {
      printf (“current directory is %s”,buffer);
   }
   else
   {
      printf (“Drive Error”)
   }
}
```

See also

```
f_wchdir, f_wgetdcwd
```

5.6. f_wgetdcwd

Get current working folder on selected drive.

Format

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

Arguments

Argument	Description
drivenum	specify drive (0-A, 1-B, 2-C)
buffer	where to store current working directory string
maxlen	length of the buffer

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

```
#define BUFFLEN F_MAXPATH+F_MAXNAME

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

   if (!f_wgetcwd(drivenum, buffer, BUFFLEN))
   {
      printf (“current directory is %s”,buffer);
      printf (“on drive %c”,drivenum+’A’);
   }
   else
   {
      printf (“Drive Error”)
   }
}
```

See also

```
f_wchdir, f_wgetcwd
```

5.7. f_wrename

Renames a file or directory. This function is obsoleted by ***f_wmove***.

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

Format

```
int f_wrename(const wchar *filename, const wchar *newname)
```

Arguments

Argument	Description
filename	file or directory name with/without path
newname	new name of target file or directory (without path)

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

```
void myfunc(void)
{
      .
      .
   f_wrename (“oldfile.txt”,”newfile.txt”);
   f_wrename (“A:/subdir/oldfile.txt”,”newfile.txt”);
      .
      .
}
```

See also

```
f_wmkdir, f_wopen, f_wmove
```

5.8. f_wmove

Moves a file or directory with unicode16 name. The original is lost. This function obsoletes ***f_wrename***. The source and target must be in the same volume.

Format

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

Arguments

Argument	Description
filename	file or directory name with/without path
newname	new name of file or directory with/without path

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

```
void myfunc(void)
{
      .
      .
   f_wmove (“oldfile.txt”,”newfile.txt”);
   f_wmove (“A:/subdir/oldfile.txt”,“A:/newdir/oldfile.txt”);
      .
      .
}
```

See also

```
f_wmkdir, f_wopen, f_wrename
```

5.9. f_wdelete

Deletes a file.

A read-only or open file cannot be deleted.

Format

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

Arguments

Argument	Description
filename	file name with or without path to be deleted

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

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

See also

```
f_wopen
```

5.10. f_wfilelength

Get the length of a file. If the requested file does not exist or has any error then this function returns with -1.

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

Format

```
long f_wfilelength (const wchar *filename)
```

Arguments

Argument	Description
filename	file name with or without path

Return values

Return value	Description
filelength	length of file
-1	if any error

Example

```
int myreadfunc(wchar *filename, char *buffer, long buffsize)
{
   F_FILE *file=f_wopen(wfilename,”r”);

   long size=f_wfilelength(wfilename);
   if (file==-1)
   {
      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;
}
```

See also

```
f_wopen
```

5.11. f_wfindfirst

Find first file or subdirectory in specified directory. First call ***f_wfindfirst*** function and if file was found get the next file with ***f_wfindnext*** function.
Files with the system attribute set will be ignored.

> **Note:** If this is called with "*.*" and this is not the root directory the first entry found will be "." - the current directory.

Format

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

Arguments

Argument	Description
filename	name of file to find
find	where to store find information

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

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));
   }
}
```

See also

```
f_wfindnext
```

5.12. f_wfindnext

Finds the next file or subdirectory in a specified directory after a previous call to ***f_wfindfirst*** or ***f_wfindnext***. First call ***f_wfindfirst*** function and if file was found get the rest of the matching files by repeated calls to the ***f_wfindnext*** function. Files with the system attribute set will be ignored.

> **Note:** If this is called with "*.*" and it is not the root directory the first file found will be ".." - the parent directory.

Format

```
int f_wfindnext(F_WFIND *find)
```

Arguments

Argument	Description
find	find information (created by ***f_wfindfirst*** call)

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

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));
   }
}
```

See also

```
f_wfindfirst
```

5.13. f_wstat

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

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

Format

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

Arguments

Argument	Description
filename	file
stat	pointer to F_STAT structure to be filled

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

```
void myfunc(void)
{
   F_STAT stat;
   if (f_wstat(“myfile.txt”,&stat))
   {
      printf (“error”);
      return;
   }
   printf (“filesize:%d”,stat.filesize);
}
```

See also

```
f_wgettimedate, f_wsettimedate
```

5.14. f_wsettimedate

Set the time and date of a file or directory. (See Section 2 for further information about porting).

Format

```
int  f_wsettimedate(const  wchar  *filename,  unsigned  short  ctime,
       unsigned short cdate)
```

Arguments

Argument	Description
filename	file
ctime	creation time of file or directory
cdate	creation date of file or directory

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

```
void myfunc(void)
{
   f_wmkdir(“subfolder”);  /*creating directory */

   f_wsettimedate(“subfolder”,f_wgettime(),f_wgetdate());
}
```

See also

`f_wgettimedate, f_wstat`

5.15. f_wgettimedate

Get time and date information from a file or directory. (See Section 2 for more information about porting).

Format

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

Arguments

Argument	Description
filename	target file
pctime	pointer to where to store creation time
pcdate	pointer to where to store creation date

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

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 & 0xf800) >> 9);
      printf ("Time: %d:%d:%d",hour,minute,sec);
      printf ("Date: %d.%d.%d",year,month,day);
   }
   else printf ("File time cannot retrieved!"

}
```

See also

```
f_wsettimedate, f_wstat
```

5.16. f_wsetattr

This routine is used to set the attributes of a file. Possible file attribute settings are defined by the FAT file system:

```
F_ATTR_ARC  Archive
F_ATTR_DIR  Directory
F_ATTR_VOLUME  Volume
F_ATTR_SYSTEM  System
F_ATTR_HIDDEN  Hidden
F_ATTR_READONLY   Read Only
```

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
filename	target file
attr	new attribute setting

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

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

See also

```
f_wgetattr
```

5.17. f_wgetattr

This routine is used to get the attributes of a specified file. Possible file attribute settings are defined by the FAT file system:

```
F_ATTR_ARC  Archive
F_ATTR_DIR  Directory
F_ATTR_VOLUME  Volume
F_ATTR_SYSTEM  System
F_ATTR_HIDDEN  Hidden
F_ATTR_READONLY   Read Only
```

Format

```
int f_wgetattr(const wchar *filename, unsigned char *attr)
```

Arguments

Argument	Description
filename	target file
attr	pointer to place attribute setting

Return values

Return value	Description
F_NO_ERROR	success
else	(see error codes table)

Example

```
void myfunc(void)
{
   unsigned char attr;
   /* find if myfile 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");
}
```

5.18. f_wopen

Opens a file. The following modes are allowed to open:

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.

Table 8, f_wopen modes

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

Format

```
F_FILE *f_wopen(const wchar *filename, const wchar *mode);
```

Arguments

Argument	Description
filename	file to be opened
mode	mode to open file with

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_wopen(“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);
}
```

See also

f_read, f_write, f_close, f_wtruncate

5.19. f_wtruncate

Opens a file for writing and truncates it to the specified length. If the length is greater than the length of the existing file then the file is padded with zeroes to the truncated length.

Format

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

Arguments

Argument	Description
filename	file to be opened
length	new length of file

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(wchar *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;
}
```

See also

```
f_wopen
```

6. Driver Interface

This section documents the required interface functions to provide a media driver for the file system.

Reference should also be made to the sample device drivers supplied with the code when developing a new driver. The easiest starting point is the RAM driver.

6.1. Driver Interface Functions

```
xxx_initfunc
xxx_getphy
xxx_readsector
xxx_readmultiplesector
xxx_writesector
xxx_writemultiplesector
xxx_getstatus
xxx_release
```

These are the routines that may be supplied by any driver.

The **xxx** is a reference to the particular driver being developed e.g. **xxx**=**cfc** for compact flash card driver.

The ***xxx_initfunc*** routine is mandatory and is passed to the ***f_initvolume*** routine to initialize a volume. This passes a set of pointers to the driver interface functions below to the file system.

The ***xxx_getphy*** routine is mandatory and is called by the file system to find out the physical properties of the device e.g. number of sectors.

The ***xxx_readsector*** routine is mandatory and is used to read a sector from the target device.

The ***xxx_readmultiplesector*** routine is optional and is used to read a series of sector from the target device. If not available ***xxx_readsector*** will be used.

The ***xxx_writesector*** routine is optional and is required to write a sector to the target device. It is mandatory if format is required.

The ***xxx_writemultiplesector*** routine is optional and is used to write a series of sectors to the target device. If not available ***xxx_writesector*** will be used.

The ***xxx_getstatus*** routine is optional and is only used for removable media to discover their status i.e. whether a card has been removed or changed.

The ***xxx_release*** routine is optional and can be used to release any resources associated with a drive when it is removed.

6.2. xxx_initfunc

Passed to the ***f_initvolume*** and/or ***f_createdriver*** routine to create the driver. The routine passes to the file system a set of function pointers to access the volume and also the driver pointer itself. These function pointers are to the other functions documented in this section.

Format

```
F_DRIVER *xxx_initfunc(unsigned long driver_param)
```

Arguments

	Argument	Description
`	driver_param	driver parameter

Return values

Return value	Description
F_DRIVER *	driver pointer or NULL if it is failed

All driver init function should allocate or use a static structure and it has to return with the filled F_DRIVER structure and it's pointer value. The F_DRIVER structure is defined as:

```
typedef struct F_DRIVER
{
   FN_MUTEX_TYPE mutex;    /* mutex for the driver    */
   char separated;         /* signal if the driver is separated */

   unsigned long user_data;         /* user defined data */
   void *user_ptr;         /* user define pointer */

   /* driver functions */
   F_WRITESECTOR writesector;
   F_WRITEMULTIPLESECTOR writemultiplesector;
   F_READSECTOR readsector;
   F_READMULTIPLESECTOR readmultiplesector;
   F_GETPHY getphy;
   F_GETSTATUS getstatus;
   F_RELEASE release;
} _F_DRIVER;
```

All function pointers to inform the file system which functions to call.

The **user_ptr** and/or ***user_data*** is assigned by the driver. The value stored in the **user_ptr** and/or ***user_data*** is included in F_DRIVER structure and all driver function calls for that volume. The usage of these fields are determined by the driver but is typically used to identify one of a set of attached interfaces e.g. if there are multiple Compact Flash card slots being controlled by a single driver. A call to ***f_delvolume*** will cause the file system to call the driver ***xxx_release*** with F_DRIVER structure pointer, where the assigned **user_ptr,** which will then be removed when the driver function returns.

> **Note:** The **driver_param** value passed to the ***xxx_initfunc*** is determined by the ***f_initvolume*** or ***f_createdriver*** call. The driver may use this value in the **user_ptr** or ***user_data*** field of the returned structure or assign another value as the driver requires. The file system will make all subsequent calls to driver functions with the assigned value in the F_DRIVER structure.

6.3. xxx_getphy

This function is called by the file system to discover the physical properties of the drive. The routine will set the number of cylinders, heads and tracks and the number of sectors per track.

Format

```
int xxx_getphy(F_DRIVER *driver,F_PHY *pPhy)
```

Arguments

Argument	Description
driver	driver structure
pPhy	pointer to physical control structure

Return values

Return value	Description
0	Success
else	Error codes for this device e.g. device not present

The F_PHY structure is defined as follows:

```
typedef struct
{
   unsigned short number_of_cylinders; /* number of cylinders */
   unsigned short sector_per_track;    /* sectors per track */
   unsigned short number_of_heads;     /* number of heads */
   unsigned long number_of_sectors;    /* number of sectors */
   unsigned char media_descriptor;     /* fix or removable */
                                       /* use _MEDIADESC_xxx */
} F_PHY;
```

Note: the number of cylinders is not required by the system. All other parameters must be set correctly by the ***xxx_getphy*** function.

6.4. xxx_readsector

This function is called by the file system to read a complete sector.

Format

```
int  xxx_readsector(F_DRIVER  *driver,  void  *data,  unsigned  long
       sector)
```

Arguments

Argument	Description
driver	driver structure
data	pointer to write data to from specified sector
sector	number of sector to be written

Return values

Return value	Description
0	Success
else	Sector out of range

6.5. xxx_readmultiplesector

This function is called by the file system to read a series of consecutive sectors. This function is optional – its inclusion will enhance performance on most devices and is particularly important with Hard Disk Drives.

Format

```
int xxx_readmultiplesector(F_DRIVER *driver, void *data, unsigned
      long sector, int cnt )
```

Arguments

Argument	Description
driver	driver structure
data	pointer to write data to from specified sector
sector	number of first sector to be written
cnt	number of sectors to write

Return values

Return value	Description
0	Success
else	Sector out of range

6.6. xxx_writesector

This function is called by the file system to write a complete sector.

> **Note**. This function maybe omitted if a read-only drive is required.

Format

```
int xxx_writesector(F_DRIVER *driver, void *data, unsigned long
    sector)
```

Arguments

Argument	Description
driver	driver structure
data	pointer to data to write to specified sector
sector	number of sector to be written

Return values

Return value	Description
0	Success
else	Sector out of range

6.7. xxx_writemultiplesector

This function is called by the file system to write a series of consecutive sectors. This function is optional – its inclusion will enhance performance on most devices and is particularly important with Hard Disk Drives.

Format

```
int xxx_writemultiplesector(F_DRIVER *driver, void *data, unsigned
      long sector, int count)
```

Arguments

Argument	Description
driver	driver structure
data	pointer to data to write to specified sector
sector	number of first sector to be written
cnt	number of sectors to write

Return values

Return value	Description
0	Success
else	Sector out of range

6.8. xxx_getstatus

This function is called by the file system to check the status of the media. This is used with removable media to check that a card has not been removed or swapped. The function returns a bit field of new status information.

> **Note**:. If this drive is for a permanent media (e.g. Hard disk or RAM drive), this function may be omitted.

Format

```
int xxx_getstatus(F_DRIVER *driver)
```

Arguments

Argument	Description
driver	driver structure

Return values

Return value	Description
0	All Ok
F_ST_MISSING	Card has been removed (Bit field)
F_ST_CHANGED	The card has been removed and replaced (Bit field)
F_ST_WRITEPROTECT	The card is write protected (Bit field)

6.9. xxx_release

This function is called by the file system to remove a drive. The drive can use this call to free any resources associated to that drive. Use of this routine in the driver is optional.

This function is called is an ***f_delvolume*** API call is made if volume was created by f_initvolume or this function is called when ***f_releasedriver*** is called. After this is completed the file system removes all record of this volume.

Format

```
void xxx_relese (F_DRIVER *driver)
```

Arguments

Argument	Description
driver	driver structure

Return values

none

7. Compact Flash Card

7.1. Overview

The Compact Flash Card (CFC) driver is designed to operate with all standard compact flash cards types 1 and 2.

There are three methods for interfacing with a Compact Flash Card:

- True IDE Mode
- PC Memory Mode
- PC I/O Mode

The package contains a sample driver for all three modes.

Throughout the code the areas which are target specific have been put within an HCC_HW definition e.g.

```
#ifdef HCC_HW
Target specific hardware parts
#endif
```

Within these areas the parts listed in this section must be provided for the driver to function.

7.2. Porting True IDE Mode

7.2.1. Files

There are three files for using True IDE mode:

cfc_ide.h - header file for ide source files
cfc_ide.c - source code for running IDE without interrupts

7.2.2. Hardware Porting

The following are the header file definitions which must be modified

CFC_TOVALUE - this value is hardware dependent and is a counter for loop expiry. The developer may replace this with a host OS timeout function.

CFC_CSO - this is for accessing a chip select register and is hardware dependent. The code assumes a chip select is used to access the card and is removed after access. The developer must modify this and all accesses to meet the host system design. It should also be noted that the chip select needs to be set for a relatively long access time (>300ns). Developers should check the timing in the CFC Specification.

Compact Flash Registers:

The following definitions are used to access the compact flash registers:

CFC_BASE	- Base address of the compact flash card
CFC_DATA	- Macro to access the data register
CFC_SECTORCOU	- Macro to access the sector count register
CFC_SECTORNO	- Macro to access the sector number register
CFC_CYLINDERLO	- Macro to access the cylinder low word register
CFC_CYLINDERHI	- Macro to access the cylinder high word register
CFC_SELC	- Macro to access the select card register
CFC_COMMAND	- Macro to access the command register
CFC_STATE	- Macro to access the state register (same address as command)

CPLD Logic:

HCC uses CPLD logic in most of its reference designs for CFCards. The following definitions are used to read from HCC CPLD logic state changes in the card.

CFC_CPLDSTATE	- MACRO for reading the state
CFC_CPLDSTATE_CDCH	- State bit for card has changed
CFC_CPLDSTATE_CFCD	- State bit for card removed

The developer must implement something to reflect this functionality. Contact support@hcc-embedded.com for reference design information.

7.2.3. Setting IDE Mode

A special sequence needs to be done to force the compact flash card into IDE mode. This is done in HCC hardware by a sequence executed by the CPLD which:

1. switches off power to the card
2. –OE signal is grounded
3. switches power on

Please reference the CFC specification or contact support@hcc-embedded.com for reference design information.

7.3. Further Information

HCC-Embedded provide design and consultancy services for developers implementing Compact Flash Cards. HCC-Embedded also has a range of specific drivers for different CF configurations such as with interrupts and in PC IO mode.

HCC-Embedded also have several hardware reference designs for Compact Flash interfaces.

The complete compact flash card specification may be obtained from www.compactflash.org.

8. MultiMediaCard/Secure Digital Card Driver

8.1. Overview

The sample drivers provided support MMC cards, SD cards Version 1 and 2 and SDHC cards. Various other derivative types are also supported such as mini-SD cards and Transflash.

Secure Digital cards are a super-set of MultiMediaCards i.e. they can be used exactly in the same manner as MMCs but have additional functionality available. In particular they have an additional two interface pins.

When used in Secure Digital mode there are 3 methods of communicating with the card:

SPI mode

This is available on both MMC and SD cards primarily because of its wide availability and ease of use. Because many standard CPUs support an SPI interface it reduces the load on the host system compared to other interface methods. When SPI is implemented by software control this benefit is lost.

MultiMediaCard Mode

This is a special mode for communicating with MultiMediaCards requiring very few IO pins. It has the disadvantage that generally software has to control every bit transfer and clock.

Secure Digital Mode

This is not compatible with MultiMediaCards. It has the basic advantage that it uses four data lines and thus the potential transfer speeds are higher (up to 10MBytes/sec) but unless there is specific UART hardware on the host system the load on the host is generally much higher than in SPI mode (with hardware support).

8.2. Implementation

FAT provides two generic MMC/SD card drivers – one for handling a single MMC card interface, the other for handling multiple MMC interfaces through a single driver.
These drivers can be found in the **/mmc/multi** and **mmc/single** directories. These drivers do not normally require modification.

In sub-directories from these there drivers are included sample SPI drivers – these must be ported for a particular target.

8.3. Porting the SPI Driver

The sample drivers are included to give an easy porting reference. There are no standards for SPI implementations so each target is different though generally this functionality is easy to realize.

The SPI driver must include the following functions:

```
void spi_tx1 (unsigned char data8)
```

Transmits a single byte through the SPI port.

```
void spi_tx2 (unsigned short data16)
```

Transmits two bytes through he SPI port. This may simply call spi_tx1() twice.

```
void spi_tx4 (unsigned long data32)
```

Transmits four bytes throught he SPI port. This may simply call spi_tx1() four times.

```
void spi_tx512 (unsigned char *buf)
```

Transmits two bytes throught he SPI port. This may simply call spi_tx1() twice.

```
unsigned char spi_rx1 (void)
```

Receives a single byte.

```
void spi_rx512 (unsigned char *buf)
```

Receives 512 bytes.

```
void spi_cs_lo (void)
```

Set the SPI chip select to low (active) state.

```
void spi_cs_hi (void)
```

Set the SPI chip select to high (inactive) state.

```
int spi_init (void)
```

Does any required SPI port initialization.

```
void spi_set_baudrate (unsigned long br)
```

Sets the baud rate of the SPI port.

```
unsigned long spi_get_baudrate (void)
```

Gets the current baud rate of the SPI port.

```
int get_cd (void)
```

Gets the state of the Card Detect signal.

```
int get_wp (void)
```

Gets the state of the Write Protect signal.

```
t_mmc_dsc *get_mmc_dsc (void)
```

Gets the MMC parameter structure – maybe used by higher level for information about the connected card.

The following functions are only required where the driver supports multiple MMC/SD card interfaces simultaneously.

```
F_DRIVER *spi_add_device (unsigned long driver_param)
```

This function adds a new sub-device or interface to the driver.

```
int spi_del_device (void *user_ptr)
```

This function removes a sub-device or interface from the driver.

```
int spi_check_device (void *user_ptr)
```

This function ensures that the interface pointed to by the user_ptr is the active interface.

8.3.1. Further Information

HCC-Embedded provide design and consultancy services for developers implementing MultiMediaCard Host interfaces. HCC-Embedded also have several reference designs for MultiMediaCard Host interfaces.

9. Hard Disk Drive

9.1. Overview

The Hard Disk Drive (HDD) driver is designed to operate with a standard IDE HDD. The sample driver is designed to handle two HDDs simultaneously.

The design uses some CPLD logic for controlling the interface – for details of this contact: support@hcc-embedded.com.

9.1.1. Files

There are two files for the HDD driver:

hdd_ide.h	- header file for ide source files
hdd_ide.c	- source code for running IDE

9.1.2. Hardware Porting

Throughout the code the areas which are target specific have been put within an HCC_HW definition e.g.

```
      #ifdef HCC_HW
      Target specific hardware parts
      #endif
```

Within these areas the parts listed in this section must be provided for the driver to function.

The following are the header file definitions which must be modified

#define	description
HDD_TOVALUE	this value is hardware dependent and is a counter for loop expiry. The developer may replace this with a host OS timeout function.
HDD_INIT_TO	this value is hardware dependent and is a counter for loop expiry. The developer may replace this with a host OS timeout function
HDD_BASE0	Base address of the first HDD
HDD_CSBASE0	Chip select base register for first HDD
HDD_CSOPT0	Chip select option register for first HDD
HDD_CONTROL0	Control register in CPLD control logic for HDD.

Table 9, HDD defines

Hard Disk Drive Registers:

The following definitions are used to access the hard disk drive registers:

Register name	description
HDD_DATA	Macro to access the data register
HDD_FEATURE	Macro to access the feature register
HDD_SECTORCOU	Macro to access the sector count register
HDD_SECTORNO	Macro to access the sector number register
HDD_CYLINDERLO	Macro to access the cylinder low word register
HDD_CYLINDERHI	Macro to access the cylinder high word register
HDD_SELC	Macro to access the select card register
HDD_COMMAND	Macro to access the command register
HDD_STATE	Macro to access the state register (same address as command)

Table 10, HDD registers

10. RAM Driver

The RAM driver is a good starting point for implementing a new driver. The sample RAM driver is written to support two independent drives.

The RAM driver does not include a ***ram_getstatus*** routine because there is no concept of removing and replacing the drive - it is always present once initialized.

Follow the following steps to build a RAM drive:

1. Include the **ramdrv.c** and **ramdrv.h** files in your file system build. This ensures it can be mounted.

2. Modify the RAMDRIVE_SIZE define to the size of block of RAM you wish to use for this drive. Nb. This is statically assigned - if you require it to be malloc'd this is a minor change. Also note - there are minimum sizes for FAT16 and FAT32 - to build a FAT16 file system you must assign 2.8MB of RAM and for a FAT32 32MB. Because of this, it is normal to run FAT12 in RAM. About 50K is minimum required to run a RAM drive.

3. Call ***f_initvolume*** with the number of the volume you wish it to be also a pointer to the ***f_ramdrvinit*** function.

4. Call ***f_format*** to format the drive.

```
void main(void)
{
   /* Initialize File System */
   f_init();

   /* mount RAM drive as drive A: */
   f_initvolume(0, f_ramdrvinit, F_AUTO_ASSIGN);

   /* format the drive */
   /* creates boot sector information and volume */

   f_format(0, F_FAT12_MEDIA);  create FAT12 in RAM */

   /* now free to use the drive */
            .
            .
            .
}
```

The RAM drive may now be accessed as a standard drive using the API calls.

> **Note:** When running the test suite with the RAM drive certain tests will fail because the drive is destroyed through the simulated power on/off.
>
> **Note:** Building a RAM driver requires a quantity of RAM. The typical minimum size of RAM we recommend using for a FAT12 RAM drive is 32K. The actual

minimum size of a FAT12 RAM drive is 36 sectors (18K) which allows just one sector (512 bytes) for file storage!

11. Using CheckDisk

This section describes the usage of the ***f_checkdisk*** utility.

FAT file systems were not designed to be failsafe i.e. they were not designed in such a way that if power is lost unexpectedly they will always be reconstructed in a clean state. Several types of error may occur such as loss of chains, or lost directory entries. This utility is designed to correct all errors that can occur from unexpected power loss when using FAT. Note that if the media is used in a device with a different FAT implementation then not all errors may be correctable.

This utility must be used stand-alone i.e. no other application should be accessing the file system while this program is running.

Often a check-disk operation can be performed by more powerful devices such as desktop computers and in this case it is normal to omit the check-disk files from the build. However, if there is a non-removable media then the ***f_checkdisk*** utility should be included in the build.

> **Note:** To use check disk the system must have USE_MALLOC defined. This is necessary because with removable media the size of the table required for check disk can vary a lot and this memory is only required for the duration of the check disk process

11.1. Files

To include the ***f_checkdisk*** utility in your project add the following files to your build:

> **/chkdsk/chkdsk.c**
> **/chkdsk/chkdsk.h**

> **Note:** To use check disk the system must have USE_MALLOC defined. This is necessary because with removable media the size of the table required for check disk can vary a lot and this memory is only required for the duration of the check disk process

11.2. Build Options

For checkdisk operation these settings needed to be revising:

```
CHKDSK_LOG_ENABLE
```

This option should be enabled in **chkdsk.h** if you want to generate a log file for the actions of ***f_checkdisk***. This is recommended.

```
CHKDSK_LOG_SIZE
```

This specifies the maximum size in RAM to be used for storing check disk log information.

11.3. f_checkdisk

This function checks the state of the attached media and automatically fixes errors detected and can create a log file of what it has found.

Format

```
int f_checkdisk(int drivenum, int param)
```

Arguments

Argument	Description
drivenum	Number of drive to be checked
param	see below

Return values

Return value	Description
FC_NO_ERROR	Completed Successfully
FC_WRITE_ERROR	Unable to write a sector
FC_READ_ERROR	Unable to read a sector
FC_CLUSTER_ERROR	Unable to access a cluster in the FAT
FC_ALLOCATION_ERROR	Memory allocation failed

Parameter Values:

`CHKDSK_ERASE_BAD_CHAIN`

The function will automatically erase all bad chains found. Otherwise the file with the bad chain will be terminated at the last good cluster.

`CHKDSK_ERASE_LOST_CHAIN`

The function will automatically erase all lost chains found. Otherwise a LOSTxxxx file will be created with the files contents.

`CHKDSK_ERASE_LOST_BAD_CHAIN`

The function will automatically erase all bad lost chains. Otherwise a LOSTxxxx file will be created and this file will be terminated at the last good cluster.

Example

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

   /* check drive 0 (“A”) */
   if(ret=f_checkdisk(0, 0)
   {
      printf(“Check Disk Failed: error %d\n”,ret);
   }
   else
   {
      printf(“Check Disk Finished\n”);
   }
   .
   .
   .
}
```

11.4. Memory Requirements

The ***f_checkdisk*** utility requires memory to run. This is typically 1K of static memory (0.5K if logging is disabled) and 1.5K of stack.

Additionally a two blocks must be allocated dynamically (using ***malloc***) the sizes of which are approximately:

```
(NUMBER_OF_CLUSTERS+4096) / 8

            and

   512 + CHKDSK_LOG_SIZE
```

The second of these is not required if logging is not enabled – the `CHKDSK_LOG_SIZE` is defined in **chkdsk.h**. The number of clusters on a device can be very large and depends on how the device is formatted (number of sectors per cluster) and the size of the device. The number of clusters on a device can be approximated to:

```
(SIZE_OF_MEDIA) / (512 * SECTORS_PER_CLUSTER)
```

The number of sectors per cluster is always in the range 2^n where 0 <= n < 7.

11.5. Log File Entries

Each time the ***f_checkdisk*** utility is run a log file is generated if enabled. The following messages may appear in the log file:

`Directory: <directory_path>`

Displays directory where error messages below have been found.

`Directory entry deleted: <name>`

Either a file entry or a directory entry has been deleted from this directory

`Lost entry deleted (found in a subdirectory):/ <LOSTxxxx>`

The named lost directory or file entry has been recovered.

`Entry deleted (reserved/bad cluster): <name>`

The first cluster in a directory entry is unusable or if there is a bad element in the chain and `CHKDSK_ERASE_BAD_CHAIN` is set.

`File size changed: <name> < old_size> <new_size>`

A file was found whose size is smaller than the minimum number of clusters needed to store that file or the file size is greater than that which can be stored in the cluster chain. The file size has been changed to the maximum for the clusters allocated to that file. The user should analyze this file to find the correct termination point.

`Start cluster changed: <name>` (either "." or "..")

An invalid cluster has been found in a directory entry for either "." or "..". This has been fixed.

`Entry deleted (cross linked chain): <name>`

If the start cluster of the named file is cross-linked or if any subsequent cluster is cross-linked and `CHKDSK_ERASE_BAD_CHAIN` is set then this message will give the name of the removed file.

`Lost directory chain saved: <LOSTxxxx>`

A directory chain with no references has been found. It has been recreated with the name LOSTxxxx.

`Lost file chain saved: <LOSTxxxx>`

A file chain with no references has been found. It has been recreated in the root directory with the name LOSTxxxx.

`Lost chain removed (first cluster/cnt): <cluster> <count>`

A lost chain has been discovered and removed. This will only appear if `CHKDSK_ERASE_LOST_CHAIN` or `CHKDSK_ERASE_LOST_BAD_CHAIN` enabled. If not a LOSTxxxx file will be created.

`Last cluster changed (bad next cluster value): <name>`

In checking the file chain an invalid cluster was discovered. The cluster prior to the bad cluster is changed to end of file and the file size adjusted to the maximum for the new size of cluster chain.

`Moving lost directory: /<LOSTxxxx>`

A lost directory has been recovered.

`'..' changed to root: <LOSTxxxx>`

A lost directory entry has been placed in root so its '..' entry has been changed to point to root.

`FAT2 updated according to FAT1`.

FAT1 and FAT2 were found to be different and FAT1 is used as the correct version. This can appear only once at the beginning of the log file.

`Long filename entry/entries removed. Count=`

This appears at the end of the log file and is a count of the number of long filename entries that were invalid and unrecoverable.