HCC-Embedded

Embedded Flash File System
FAT12/16/32

Implementation Guide

Version 2.62

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

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.

0 Contents

0 Contents **2**
1 System Overview **5**
TARGET AUDIENCE 5
SYSTEM STRUCTURE/SOURCE CODE 6
SOURCE FILE LIST 6
SOURCE FILE LIST 7
GETTING STARTED 8
TESTING 9
2 Porting **11**
SYSTEM REQUIREMENTS 11
STACK REQUIREMENTS 11
REAL TIME REQUIREMENTS 11
REENTRANCY 11
MAXIMUM NUMBER OF VOLUMES 12
MAXIMUM OPEN FILES 12
LONG FILENAMES 12
GET TIME 14
GET DATE 14
RANDOM NUMBER 14
MEMCPY AND MEMSET 15
CACHE SETUP AND OPTIONS 16
FAT Caching 16
Write Caching 16
3 Drive Format **17**
COMPLETELY UNFORMATTED 17
MASTER BOOT RECORD 18
Master Boot Record 18
Partition Entry Description 18
BOOT SECTOR INFORMATION 19
Boot Sector Information Table 19
First 36 Bytes 19
4 File API **21**
FILE SYSTEM FUNCTIONS 21
FUNCTION ERROR CODES 22
F_GETVERSION 23
F_INITVOLUME 24
F_DELVOLUME 26
F_GET_VOLUME_COUNT 27
F_GET_VOLUME_LIST 28
F_FORMAT 29
F_HARDFORMAT 31
F_GETFREESPACE 33
F_SETLABEL 34

F_GETLABEL 35
F_MKDIR 36
F_CHDIR 37
F_RMDIR 38
F_GETDRIVE 39
F_CHDRIVE 40
F_GETCWD 41
F_GETDCWD 42
F_RENAME 43
F_DELETE 44
F_FILELENGTH 45
F_FINDFIRST 46
F_FINDNEXT 47
F_SETTIMEDATE 48
F_GETTIMEDATE 49
F_SETATTR 50
F_GETATTR 51
F_OPEN 52
F_CLOSE 54
F_WRITE 55
F_READ 56
F_SEEK 57
F_TELL 58
F_EOF 59
F_REWIND 60
F_PUTC 61
F_GETC 62
5 Driver Interface 63
DRIVER INTERFACE FUNCTIONS 63
XXX_INITFUNC 65
XXX_GETPHY 67
XXX_READSECTOR 68
XXX_READMULTIPLESECTOR 69
XXX_WRITESECTOR 70
XXX_WRITEMULTIPLESECTOR 71
XXX_GETSTATUS 72
XXX_DELFUNC 73
6 Compact Flash Card 74
OVERVIEW 74
PORTING TRUE IDE MODE 74
Files 74
Hardware Porting 74
Setting IDE Mode 75
FURTHER INFORMATION 76
7 MultiMediaCard/Secure Digital Card Driver 77
OVERVIEW 77

PORTING HARDWARE SPI DRIVER.....78
Files *78*
Hardware Porting.....*78*
MACROS.....*78*
Functions *79*
Waiting and Real-time Behavior *80*
PORTING SOFTWARE SPI DRIVER 81
Files *81*
Waiting and Realtime Behaviour.....*81*
Hardware Porting.....*83*
Bit Rates *83*
PORTING MULTIMEDIACARD DRIVER 84
PORTING SD CARD DRIVER 84
FURTHER INFORMATION 84
8 Hard Disk Drive 85
OVERVIEW.....85
Files *85*
Hardware Porting.....*85*
9 RAM Driver 87
10 Using CheckDisk.....88
FILES 88
BUILD OPTIONS 88
F_CHECKDISK 89
MEMORY REQUIREMENTS.....90
LOG FILE ENTRIES 91

1 System Overview

Target Audience

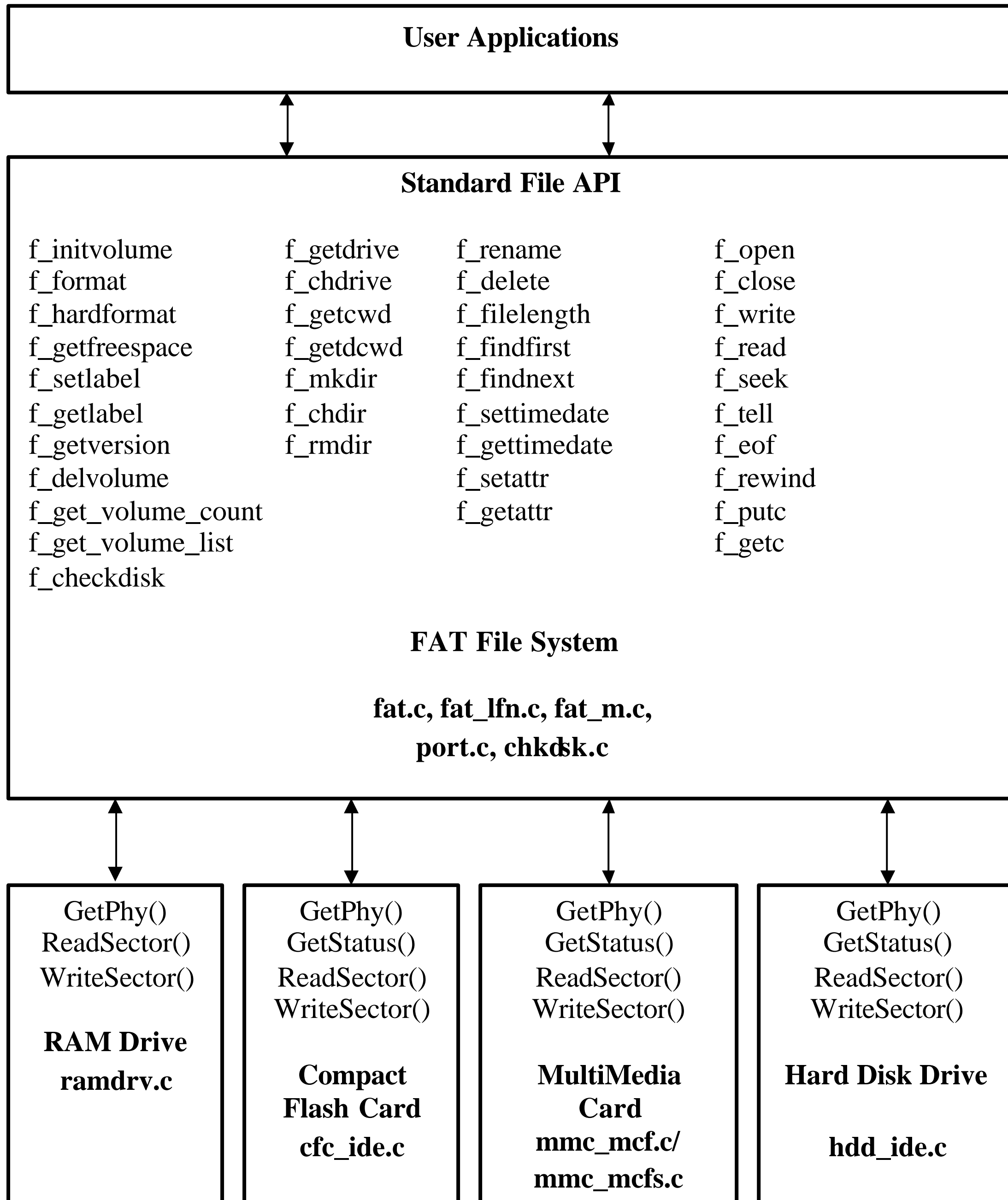
This guide is intended for use by embedded software engineers who have should have a knowledge of the C programming language, standard file API's who wish to implement a FAT12, FAT16 or FAT32 file system in any combination of RAM, Compact Flash Card, MultiMediaCard, Hard Disk Drive or other device type.

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.

System Structure/Source Code

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



Source File List

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

/src/	
fat.c	- fat file system
fat.h	- fat file header
fat_lfn.c	- alternative source file to fat.c for long filenames
fat_m.c	- fat file system reentrancy wrapper
fat_m.h	- fat file header reentrancy header
port.c	- routines that require OS specific modifications
port.h	- header for port routines.
/src/chkdsk/	
chkdsk.c	- check disk utility C source code
chkdsk.h	- header file for checkdisk utility
/src/ram/	
ramdrv.c	- RAM driver implementation
ramdrv.h	- RAM driver header file
/src/cfc/	
cfc_ide.c	- Compact Flash Card True IDE Driver
cfc_ide.h	- Compact Flash Card True IDE Header
/src/mmc/	
mmc_mcf.c	- MultiMediaCard SPI driver (based on Motorola Coldfire)
mmc_mcfs.h	- MultiMediaCard driver with software driven SPI
mmc.h	- MultiMediaCard header
/src/hdd/	
hdd_ide.c	- Hard Disk Drive IDE driver
hdd_ide.h	- Hard Disk Driver header file
/src/test/	
test.c	- Test source code for exercising the file system
test.h	- Header file for test source code

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.c** and **port.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 5. The sample drivers are documented in Sections 6, 7, 8 and 9.

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

Getting Started

To get your development started as efficiently as possible we recommend that the developer follow the instructions in Section 9 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.

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.c** and **test.h** in your test project.

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

The test program requires the following four functions to be implemented by the developer - they are host system dependent - sample code below demonstrates the required functionality:

```
 int _f_poweron(void)
/*
 * Ths function which should call f_initvolume for the drive to be
 * tested - which must be drive 0 ("A"). If the RAM drive is being
 * tested then the volume must be both initialized and formatted.
 * _f_poweron is called by the test code during the test operation.
 * This routine should return non-zero if any error is detected.
 */
int _f_poweron(void)
{
#if RAM_TEST              /* testing RAM drive */
int ret;
      ret=f_initvolume(0,f_ramdrvinit, F_AUTO_ASSIGN);
      if (ret) return ret;

      return f_format(0,F_FAT12_MEDIA);

#else  /* if testing compact flash drive */

      return f_initvolume(0,f_cfcdrvinit, F_AUTO_ASSIGN);
#endif
}

 int _f_poweroff(void)
/*
 * This function should call f_delvolume for the drive being tested.
 * _f_poweroff is called by the test code during the test operation.
 * This routine should return non-zero if any error is detected.
 * The routine may also be used to free allocated resources
 */

int _f_poweroff(void)
{
      return f_delvolume(0);
}
```

```
/* _f_dump() displays text output from the tests */

void _f_dump (char *s)
{
      printf("%s\n",s);
}

/* _f_result() displays errors detected during the test */

long _f_result(long testnum, long error)
{
      printf("test number %d failed with error %d\n", testnum, error);
      return(testnum)
}
```

2 Porting

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.

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.

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.

Reentrancy

If more than one user is going to access the file system at one time then reentrancy must be considered.

A reentrancy wrapper is included in **fat_m.c**. To enable reentrancy you must first set the define **F_REENTRANCY** to a non-zero value. This causes all the API functions to be called via the reentrancy wrapper functions in **fat_m.c**.

The reentrancy wrapper routines call semaphore routines contained in **port.c**. These are general functions and should be replaced by the routines provided by your operating system.

Nb. The semaphore 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.

It is only necessary to protect a volume from certain accesses simultaneously. Therefore it is practical to provide a separate semaphore for each volume in use. It is up to the developer to provide the management or wrapper functions to handle this.

Maximum Number of Volumes

The maximum number of volumes allowed by your system should be set in the F_MAXVOLUME definition in **fat.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.

Maximum Open Files

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

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 **fat.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 **fat.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).

NB. On December 3rd 2003 Microsoft announced that it would exercise its patent rights relating to certain elements of how long filenames are implemented in FAT file systems. As a consequence it is up to the user to contact Microsoft to get the required licenses should they use the long filename option.

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 & 001fH)
minute	(0-59 valid)	((t & 07e0H) >> 5)
hour	(0-23 valid)	((t & 0f800H) >> 11)

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 & 001fH)
month	(1-12 valid)	((d & 01e0H) >> 5)
years since 1980	(0-119 valid)	((d & fe00H) >> 9)

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.

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.

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 fat.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
#define _memcpy(d,s,l) memcpy(d,s,l)
#define _memset(d,c,l) memset(d,c,l)
#endif
```

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.

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 **fat.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:

FATCAHCE_BLOCKS*FATCACHE_READAHEAD*512

This default setting requires 16K of additional RAM.

Write Caching

The amount of data that can be written ahead depends on the depth of the write cache. 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 **fat.h** is:

```
#define WR_DATACACHE_SIZE 32
```

This will require 768 additional bytes of RAM.

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.

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.

When the ***f_format*** function is called the drive will be formatted with Boot Sector Information. This is exactly the same as if the ***f_hardformat*** function had been called. Please see Boot Sector Information section below for further information.

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.

Refer to the ***f_hardformat*** and ***f_format*** commands for description of how to choose the format type (FAT12/16/32).

Master Boot Record

If a card contains a Master Boot Record it is formatted as in the tables below.
As standard the file system does not hard format a card with an MBR but with Boot Sector Information as described in the next section. A hard format will remove the MBR information.

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.

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

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

Partition Entry Description

Nb. Should a developer require to use multiple partitions on a single card please contact support@hcc-embedded.com

Boot Sector information

This is the system used as standard by the file system. If a hard format command is issued the card is always formatted with this table in the first sector. 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
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 0x16	1 2	Media Descriptor Sectors/FAT16	0xf8 (hard disk) 0xf0 (removable media) 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

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 hard 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

Boot Sector Information Table
FAT12/16 After byte 36

Nb. 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 0x32	2 2	File System Info Backup Boot Sector	Sector number of FSINFO structure in the reserved area of the FAT32. Usually 1. 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 hard format.
0x47	11	Volume Label	"NO LABEL" is put here by a format
0x52	8	File System Type	Always set to string "FAT32 ".

Boot Sector Information Table
FAT32 After byte 36

4 File API

File System Functions

Volume functions

- ***f_getversion***
- ***f_initvolume***
- ***f_delvolume***
- ***f_get_volume_count***
- ***f_get_volume_list***
- ***f_format***
- ***f_hardformat***
- ***f_getfreespace***
- ***f_setlabel***
- ***f_getlabel***

Drive\Directory handler functions

- ***f_getdrive***
- ***f_chdrive***
- ***f_getcwd***
- ***f_getdcwd***
- ***f_mkdir***
- ***f_chdir***
- ***f_rmdir***

File functions

- ***f_rename***
- ***f_delete***
- ***f_filelength***
- ***f_findfirst***
- ***f_findnext***
- ***f_settimedate***
- ***f_gettimedate***
- ***f_getattr***
- ***f_setattr***

Read/Write functions

- ***f_open***
- ***f_close***
- ***f_write***
- ***f_read***
- ***f_seek***
- ***f_tell***
- ***f_eof***
- ***f_rewind***
- ***f_putc***
- ***f_getc***

Function Error Codes

Error Code	Literal	Meaning
F_NO_ERROR	0	No Error - function was successful
F_ERR_INVALIDDRIVE F_ERR_NOTFORMATTED	1 2	The specified drive does not exist 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	A function to access a file has been called which 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 F_ERR_INVALIDSECTOR	17 18	Non-recoverable error on drive 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_DELFUNC	28	The delete drive driver function failed
F_ERR_MOUNTED	29	The drive is already mounted

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());
}
```

f_initvolume

This function is used to initialize a volume. The function is called with a pointer to the 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.

Format

```
int f_initvolume(int drivenum,F_INITFUNC *pfunc,
            void *user_ptr)
```

Arguments

Argument	Description
drivenum	drive to be initialized (0:A, 1:B...)
pfunc	pointer to initialization function for drive
user_ptr	pointer to user information (see below)

Return values

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

Note: The **user_ptr** may 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 5.

Example:

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

  /* 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_hardformat

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_delfunc*** call will be made to the driver and afterwards the **user_ptr** will be cleared. 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.

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
```

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_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

f_format

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).

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 set by a hard format and 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 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_hardformat`

f_hardformat

Re-formats a drive ignoring current format information. All open files will be closed. This command will destroy any existing Master Boot Record or Boot Sector information. The new drive will be formatted without a Master Boot Record. The new drive will start with Boot Sector Information created from the information retrieved from the f_getphy() routine and use the whole available physical space for the volume. All data will be destroyed on the drive. (see Section 3 for further information)

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_hardformat(int drivenum, long fattype)
```

Arguments

Argument	Description
drivenum	which drive need to be hard formatted
fattype	type of format: FAT12, FAT16 or FAT32

Return values

Return value	Description
F_NO_ERROR	drive successfully formatted
else	(see error codes)

Note: The number of sectors per cluster on a FAT32 drive is set by a hard format and 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 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_hardformat(0, F_FAT16_MEDIA);
  if(ret)
       printf("Format CFC Error: %d", ret);
  else
       printf("CFC formatted");

  .
  .
  .
  .
}
```

See also `f_initvolume, f_format`

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.

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);
}
```

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");
}
```

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);

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

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`

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

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
```

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

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
```

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
```

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

f_rename

Renames a file or directory.

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
```

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

f_filelength

Get the length of a file. If the requested file does not exist this function returns with zero.

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

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

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

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

f_settimedate

Set the time and date of a file or directory. (See Section 2 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_gettimedate

Get time and date information from a file or directory. (See Section 2 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 & 001fH) << 1;
       unsigned short minute=((t & 07e0H) >> 5);
       unsigned short hour=((t & 0f800H) >> 11);
       unsigned short day= (d & 001fH);
       unsigned short month= ((d & 01e0H) >> 5);
       unsigned short year=1980+ ((d & f800H) >> 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_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);
  }
```

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");
  }
```

f_open

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

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

Note: 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,

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
```

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 bytes 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,3,1,file)!=3)
  {
       printf ("Error: write incomplete");
  }

  f_close(file);
}
```

See also

f_read, f_open, f_close

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 read bytes

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,size,1,file)!=size) {
       printf ("different number of bytes are
       read");
  }
  f_close(file);
  return 0;
}
```

See also

f_seek, f_tell, f_open, f_close, f_write

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

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
```

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

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

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
```

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

5 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.

Driver Interface Functions

- ***xxx_initfunc***
- ***xxx_getphy***
- ***xxx_readsector***
- ***xxx_readmultiplesector***
- ***xxx_writesector***
- ***xxx_writemultiplesector***
- ***xxx_getstatus***
- ***xxx_delfunc***

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

xxx_initfunc

Passed to the ***f_initvolume*** routine to create a volume. The routine passes to the file system a set of function pointers to access the volume and an optional **user_ptr**. These function pointers are to the other functions documented in this section.

Format

```
int xxx_initfunc(F_FUNC *pfunc, void *user_ptr)
```

Arguments

Argument	Description
pfunc	function structure used by file system
user_ptr	pointer to drive init information

Return values

Return value	Description
0	Always successful

The F_FUNC structure is defined as:

```
typedef struct {
        F_WRITESECTOR writesector;
        F_WRITEMULTIPLESECTOR writemultiplesector;
        F_READSECTOR readsector;
        F_READMULTIPLESECTOR readmultiplesector;
        F_GETPHY getphy;
        F_GETSTATUS getstatus;
        F_DELFUNC delfunc;
        void *user_ptr;
} F_FUNC;
```

Except for the last field these are all function pointers to inform the file system which functions to call.

The **user_ptr** is assigned by the driver. The value stored in the **user_ptr** is included in all driver function calls for that volume. The usage of this field is 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_delfunc*** with the assigned **user_ptr** which will then be removed when the driver function returns.

Note: The **user_ptr** value passed to the ***xxx_initfunc*** is determined by the ***f_initvolume*** call. The driver may use this value in the **user_ptr** 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.

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_PHY *pPhy, void *user_ptr)
```

Arguments

Argument	Description
pPhy	pointer to physical control structure
user_ptr	pointer to drive information (see ***xxx_initfunc***)

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 */
} 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.

xxx_readsector

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

Format

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

Arguments

Argument	Description
data	pointer to write data to from specified sector
sector	number of sector to be written
user_ptr	pointer to driver information (see ***xxx_initfunc***)

Return values

Return value	Description
0	Success
else	Sector out of range

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(void *data,
        unsigned long sector, int cnt, void
        *user_ptr)
```

Arguments

Argument	Description
data	pointer to write data to from specified sector
sector	number of first sector to be written
cnt	number of sectors to write
user_ptr	pointer to driver information (see ***xxx_initfunc***)

Return values

Return value	Description
0	Success
else	Sector out of range

xxx_writesector

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

Nb. This function maybe omitted if a read-only drive is required.

Format

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

Arguments

Argument	Description
data	pointer to data to write to specified sector
sector	number of sector to be written
user_ptr	pointer to drive information (see ***xxx_initfunc***)

Return values

Return value	Description
0	Success
else	Sector out of range

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(void *data,
     unsigned long sector, int count, void
     *user_ptr)
```

Arguments

Argument	Description
data	pointer to data to write to specified sector
sector	number of first sector to be written
cnt	number of sectors to write
user_ptr	pointer to drive information (see ***xxx_initfunc***)

Return values

Return value	Description
0	Success
else	Sector out of range

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.

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

Format

```
int xxx_getstatus(void *user_ptr)
```

Arguments

Argument	Description
user_ptr	pointer to drive information (see ***xxx_initfunc***)

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)

xxx_delfunc

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. After this is completed the file system removes all record of this volume including the current value of the user_ptr.

Format

```
int xxx_delfunc(void *user_ptr)
```

Arguments

Argument	Description
user_ptr	pointer to drive information (see ***xxx_initfunc***)

Return values

Return value	Description
0	Successful
Else	Error Code

6 Compact Flash Card

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 True IDE mode. For developers wishing to use other modes they should contact HCC-Embedded for further information.

Porting True IDE Mode

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

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

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.

Setting IDE Mode

A special sequence needs to be done to force the compact flash card into IDE mode. This is done in the function ***fnCFCtrueide***. This is achieved in HCC hardware by a sequence of commands to the CPLD which:

1. switches off power to the card
2. forces IDE mode
3. switches power on

This sequence may also be achieved by CPLD logic or other hardware.

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

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

7 MultiMediaCard/Secure Digital Card Driver

Overview

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 4 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).

The system currently supports the SPI driver interface. This is provided in two forms;

- Hardware SPI - where the host CPU has an SPI capability
- Software SPI - where the SPI is simulated by software using 4 GPIO pins.

How to port these is described in the sections below.

Porting Hardware SPI Driver

The hardware SPI driver is for use by systems where the host CPU has dedicated logic for handling SPI communication and in particular automatically handles SPI clock generation and bit transmission and reception such that the programmer should only receive and transmit bytes.

Unfortunately from system to system the SPI implementation varies. In particular, the handling of the SPI chip select pin may be different between systems - some automatically generate it where as other systems require it to be controlled entirely by software.

Files

The developer should include the following files to support this driver:

mmc_mcf.c	Source Code file
mmc_mcf.h	Header file

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.

MACROS

The following macros in the target specific section must be modified for the developers target platform:

SPI_CS_LO
This macro sets the SPI chip select to low. Some chips handle this automatically in which case this macro can be a NULL macro.

SPI_CS_HI
This macro sets the SPI chip select to high. Some chips handle this automatically in which case this macro can be a NULL macro.

SPI_CD_IN
This macro gets the current state of the card detect pin. Nb. If the card is not connected the pin is high.

SPI_WP_IN
This macro gets the current state of the write protect pin on the connector.

SPI_WAIT_TR
This macro waits for the transmitter to be ready. The implementation of this is UART dependant and may not be necessary.

SER_FIFOCHAR
This macro writes an 8 bit value to the transmit FIFO.

SER_FIFOWORD
This macro writes a 16 bit value to the transmit FIFO.

SER_FIFO
This macro writes a 32 bit value to the transmit FIFO.

Functions

The following functions must be modified for the developers target platform:

spiSetBR()

The setting of a baud rate is a target specific function. This routine is called with the desired baud rate divided by 10 (e.g. if 100kbit is required 10000 is passed to the function).

spiRx()

The receive handler is dependant upon the behavior of the hosts UART. This function must be modified to receive data from the SPI port of the target system.

spiInit()

Initializing the SPI interface is a target specific function so the main body of this function must be replaced. This section should do the set-up and initialization of the SPI port.

Waiting and Real-time Behavior

The following routines have wait loops inserted where they are waiting for a particular external condition to occur:

spiWaitBusy()

This can be a long wait (>10mseconds) as the data is being written into the card.

spiWaitStartBit()

This wait is dependent on the bit rate but is usually relatively short.

spiCmd()

This can be a long wait as it is waiting for a complete response from the card.

It is recommended that in initial porting these loops are left as they are until the system is stable. Then the developer should assess these loops in terms of their whole system and find an appropriate scheduling mechanism or timer mechanism. All these conditions can be delayed as long as the developer requires - i.e. there is no maximum time before the condition must be re-checked.

Porting Software SPI Driver

The software SPI driver is used to drive the SPI interface through 4 I/O pins controlled by software. Additionally two further pins are required for Card Detect and Write Protect. This driver is also useful when getting a system running even where hardware SPI is available.

Generally, if no hardware SPI driver available on the host system it is preferable to use the MultiMediaCard driver than software SPI. The main reason for this is that the performance of the two communication methods is roughly equal but the MMC Driver mode requires fewer IO pins to be connected.

Files

The developer should include the following files to support this driver:

mmc_mcfs.c	Source Code file
mmc_mcf.h	Header file

Waiting and Realtime Behaviour

The following routines have wait loops inserted where they are waiting for a particular external condition to occur:

spiWaitBusy()

This can be a long wait (>10mseconds) as the data is being written into the card and the delay is completely dependant on the card type and what it is doing.

spiWaitStartBit()

This wait is dependent on the bit rate but is usually relatively short.

spiWaitTR()

This wait is hardware dependent where there is a request to transmit but the UART requires that a transmit ready acknowledgement is given first.

spiCmd()

This can be a long wait as it is waiting for a complete response from the card.

It is recommended that in initial porting these loops are left as they are until the system is stable. Then the developer should assess these loops in terms of their whole system and

find an appropriate scheduling mechanism or timer mechanism. All these conditions can be delayed as long as the developer requires - i.e. there is no maximum time before the condition must be re-checked.

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.

Hardware porting requires the assignment of pins to each of the used pins in the driver. These pins are:

DAT_I	SPI Data input
DAT_O	SPI Data output
CLK	SPI Clock
CS	SPI Chip Select
CD	Card Detect
WP	Write Protect

For the software the following MACROS have to be written according to this assignment:

SPI_CS_LO	Set the SPI Chip select to low.
SPI_CS_HI	Set the SPI chip select to high.
SPI_DATA_LO	Set the SPI Data output to low.
SPI_DATA_HI	Set the SPI Data output to high.
SPI_CLK_LO	Set the SPI clock to low.
SPI_CLK_HI	Set the SPI clock to high.
SPI_DATA_IN	Read the SPI data input.
SPI_CD_IN	Read the Card detect.
SPI_WP_IN	Read the write protect.

Bit Rates

There is no way to generally define a bit rate for a software implementation. The developer must rely on the CPU and calculate from this. The following should be noted - the maximum guaranteed speed which all MMC/SD cards will operate at is 100Kbits/second. Generally cards operate much faster than this so it is normally not a problem if the software is much quicker. Because the interface is SPI it can be driven as slowly as required.

Porting MultiMediaCard Driver

To be inserted.

Porting SD Card Driver

To be inserted.

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.

8 Hard Disk Drive

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.

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

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

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_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.

Hard Disk Drive Registers:

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

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)

9 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_hardformat*** to format the drive.

```
void main(void){

        /* mount RAM drive as drive A: */

       f_initvolume(0, f_ramdrvinit, F_AUTO_ASSIGN);


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

       f_hardformat(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.

10 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 EFFS-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.

Files

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

/chkdsk/chkdsk.c
/chkdsk/chkdsk.h

Build Options

`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.

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");


  .
  .
}
```

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.

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.