THIN File System

FAT12/16/32
For Systems with Limited Resources

Developer's Guide

Version 1.92

0 Contents

1 System Overview

Target Audience

This guide is intended for use by embedded software engineers who have a knowledge of the C programming language, and standard file API's, and who wish to implement a DOS-compatible FAT file system in any combination of RAM, Compact Flash Card, MultiMediaCard/Secure Digital Card, Atmel DataFlash, or another device type.

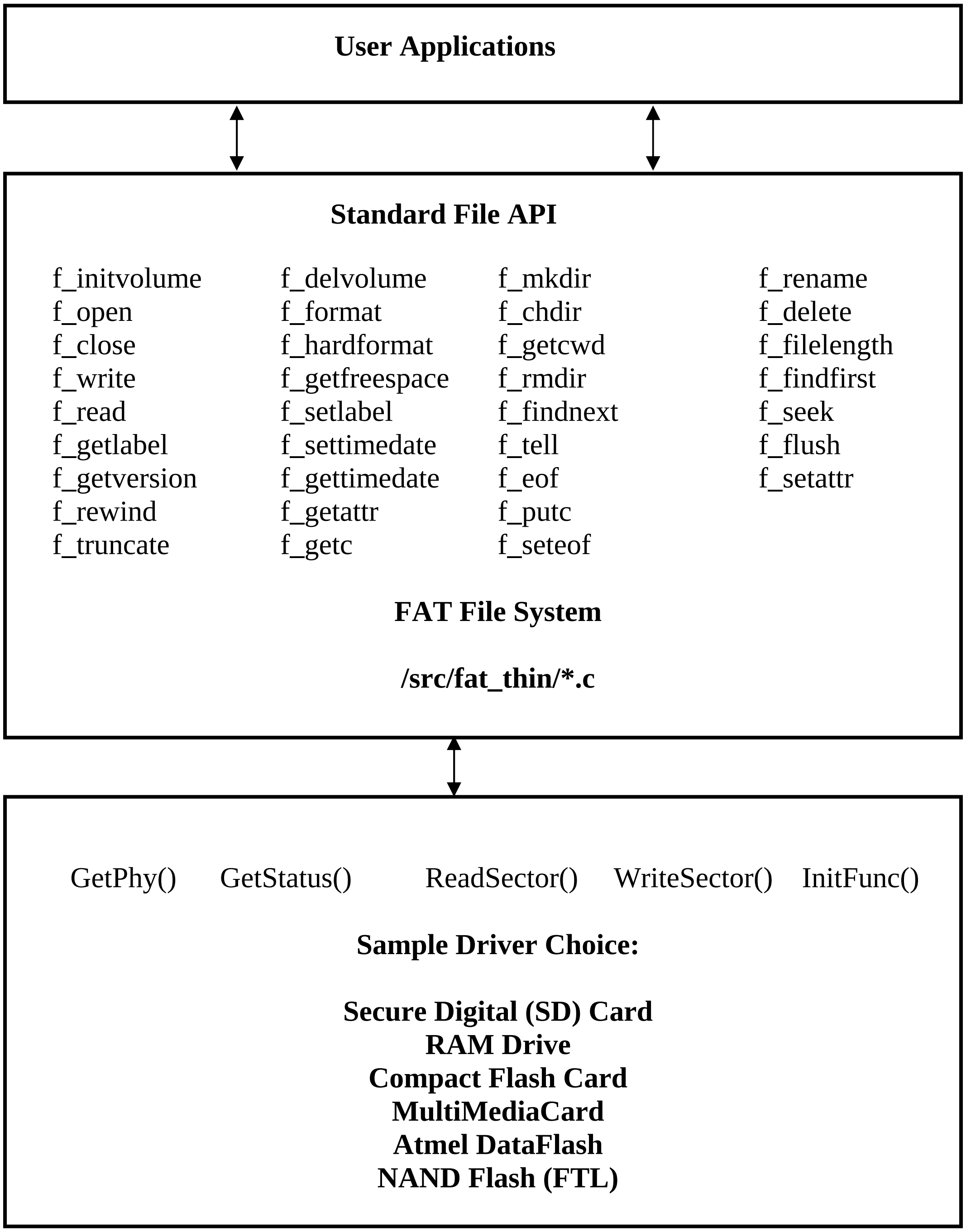
THIN is particularly targeted at systems with limited resources. There may be restrictions on the available code space or available RAM. For developers who do not have these concerns, and have >20K for code and >5K for RAM, HCC-Embedded's FAT product is recommended.

Although every attempt has been made to simplify the system's use, developers must have a good understanding of the requirements of the systems they are designing in order to obtain the maximum practical benefits.

HCC-Embedded offers hardware and firmware development consultancy to assist developers with 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 THIN file system.

/src	- root directory of source code
thin_usr.h	- global configuration file for project
/src/fat_thin	- source code for standard THIN build
dir.c	- directory handling functions without long filenames
dir.h	- header for directory functions
dir_lfn.c	- directory handling functions with long filenames
dir_lfn.h	- header for directory functions
drv.c	- low-level driver interface functions
drv.h	- header file for low-level driver interface functions
fat.c	- fat file system general functions
fat.h	- fat file general functions header
fat_thin.h	- header file for whole system
file.c	- file manipulation functions
file.h	- file functions header file
main_fth.c	- main include file for build
port.c	- routines that require OS specific modifications
port.h	- header for port routines.
util.c	- general utility functions
util.h	- general utilities header file
util_lfn.c	- general utility functions for long filenames
util_lfn.h	- general utility header file
util_sfn.c	- general utility functions for short filenames
util_sfn.h	- general utility header file
volume.c	- volume manipulation functions
volume.h	- volume functions header file
/src/fat_thin/test	- test code for standard THIN build
test.c	- Test source code for exercising the system
test.h	- Test code header file
/src/fat_sthin	- source code for super-thin reduced RAM build
dir.c	- directory handling functions without long filenames
dir.h	- header for directory functions
drv.c	- low-level driver interface functions
drv.h	- header file for low-level driver interface functions
fat.c	- fat file system general functions
fat.h	- fat file general functions header

fat_sthin.h	- header file for whole system
file.c	- file manipulation functions
file.h	- file functions header file
main_fth.c	- main include file for build
port.c	- routines that require OS-specific modifications
port.h	- header for port routines
util.c	- general utility functions
util.h	- general utilities header file
util_sfn.c	- general utility functions for short filenames
util_sfn.h	- general utility header file
volume.c	- volume manipulation functions
volume.h	- volume functions header file

/src/fat_sthin/test - test code for super-thin THIN build

test.c	- Test source code for exercising the system
test.h	- Test code header file

/src/ram

ramdrv.c	- RAM driver implementation
ramdrv.h	- RAM driver header file

/src/cfc

cfc.c	- Compact Flash Card Memory mode Driver
cfc.h	- Compact Flash Card Header
cfc_ide.c	- Compact Flash Card True IDE Driver
cfc_ide.h	- Compact Flash Card True IDE Header

/src/mmc

mmc_opt.h	- MMC Header enabling options
mmc.h	- MultiMediaCard/SD card driver header
mmc.c	- MultiMediaCard/SD card driver
mmc_dsc.h	- MMC configuration descriptor header

/src/mmc/drv430

mmc_drv.c	- Source file for MSP430 SPI driver
mmc_drv.h	- Header for MSP430 SPI driver

/src/mmc/drv8051

mmc_drv.c	- Source file for 8051 SPI driver
mmc_drv.h	- Header for generic SPI driver
mmc_drvs.c	- Source file for 8051 software SPI driver
mmc_drvs.h	- Header for generic software SPI driver

/src/mmc/sam7s256

mmc_drv.c	- Source file for AT91SAM7 SPI driver
mmc_drv.h	- Header for generic SPI driver

The developer should not normally modify the THIN 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 developer's target system. The tasks required are straightforward and ensure easy integration with any operating environment. Full guidance for this is given in the porting section below.

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 hardware. Driver interface functions are documented in Section 5. Sample drivers are documented in Sections 6, 7 and 8.

Driver implementation is straightforward. Any new driver should be based on the RAM driver, which is the simplest possible starting point.

Getting Started

To get your development started as efficiently as possible, follow the instructions in Section 8 to set up a RAM drive on the target (on systems with limited RAM resources this may not be possible). This enables the developer to become familiar with the system and to develop test code without the need to worry about a new hardware interface.

2 Porting

System Requirements

THIN is designed to be as open and as portable as possible. No assumptions are made about the functionality or behavior, or even the existence, of the underlying operating system. For the system to work at its best, the porting outlined below is required. This is a straightforward task for an experienced engineer.

SuperThin Build

There are two basic build options controlled by the **FAT_SUPERTHIN** define in **thin_src.h**. The **FAT_SUPERTHIN** build option minimizes the RAM requirements of the system but also restricts the available functionality.

The following restrictions apply to the SuperThin build:

1. No Long filenames
2. Only one file is allowed to be open at one time.
3. Performance is reduced approximately 35%. This depends on many configuration factors.
4. Get label and Set label commands are not available.
5. Get attribute and Set attribute commands are not available.
6. Directory commands are possible only if there is no open file.
7. Get time and Set time and data functions are not available

The benefit of SuperThin is that an otherwise fully functional file system can be built using an absolute minimum of RAM. If compiled for an MSP430 this has been measured below 700 bytes with no directories and at about 800 bytes if directories are included. This is the complete RAM allocation including stack and data.

Note: This figure depends on both the target and the compiler. On an 8051, for example, the RAM required for stack/overlay will increase these numbers significantly.

Stack Requirements

THIN functions are always called in the context of the calling thread or task. Naturally, the functions require stack space, which must be allocated in order to use file system functions. Typically calls to the file system will use <0.5Kbytes of stack. However, if long filenames are used, then the stack size should be increased to 1K. See the 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 occurs are documented in the applicable driver sections. The delays should be modified to meet the system requirements, either by implementing interrupt control of events or by scheduling other parts of the system. Read the relevant driver section for details.

Get Time

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

An empty function ***f_gettime*** is provided in **port.c.** It should be modified 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 & f800H) >> 11)

Get Date

For THIN to be compatible with other systems, it is normally necessary to use a function that provides the date 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 the function ***f_getrand*** which THIN uses to get a pseudo-random number to use as the volume serial number.

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

Long Filenames

There are two sets of source files from which to choose:

dir_sfn.c, util_sfn.c - contains the THIN 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.

dir_lfn.c, util_lfn.c - contains the THIN file system with complete long filename support.

Using long filename support is optional because of the increase in system resources required to process long filenames. Among other things, the stack sizes of applications that call the file system must be increased and the amount of checking required is increased.

To choose between using the long and short filenames, use the F_LONGFILENAME definition in **fat_thin.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. Depending on your application, you can reduce the size of F_MAXPATH and F_MAXLNAME (in **fat_thin.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 ***fn_rename,*** 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. This can significantly reduce the stack requirements (by approximately 1K).

Memcpy and Memset

The THIN package includes ***memcpy*** and ***memset*** functions.

It is recommended that you re-define these functions so that they 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 **util.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
```

3 Drive Format

This document does not seek to describe a FAT file system in detail; there are many reference works to choose from for this purpose. THIN handles a majority of the features of a FAT file system with no need for further understanding of the underlying issues. However, there are some areas where an understanding is required. This section describes these features and provides some additional useful information.

Removable media may be formatted in three different forms:

- Completely unformatted media
- Master boot record
- Boot sector Information only

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

Completely unformatted

If the drive is completely unformatted then it is not usable until it has been formatted.

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

The format of the file medium is determined by the number of clusters on it. Information about the connected device is given to the system from the ***f_getphy*** call, from which the number of available clusters on the device is calculated.

Refer to ***f_hardformat*** and ***f_format*** for a 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 has just 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

Boot Sector information

This is how boot sector information is organized 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(1-128)
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

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

Note: The serial number field is generated by the random number function. See the 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 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_format***
- ***f_hardformat***
- ***f_getfreespace***
- ***f_setlabel****
- ***f_getlabel****

Drive\Directory handler functions

- ***f_getcwd***
- ***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****
- ***f_truncate***

Read/Write functions

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

* These functions are not available if the FAT_SUPERTHIN build is defined.

Function Error Codes

Error Code	Literal	Meaning
F_NO_ERROR	0	No Error - function was successful
F_ERR_RESERVED_1	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	A function that requires the file to be open to access a file has been called
F_ERR_EOF	9	End of file
F_ERR_RESERVED_2	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 is available for a driver or the function generates an error
F_ERR_CARDREMOVED	16	The card has been removed
F_ERR_ONDRIVE	17	Non-recoverable error on drive
F_ERR_INVALIDSECTOR	18	A sector has developed an error
F_ERR_READ	19	Error reading the volume
F_ERR_WRITE	20	Error writing file to volume
F_ERR_INVALIDMEDIA	21	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_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 the volume. It works independently of the status of the hardware; i.e., it does not matter if a card is inserted or not.

Format

```
unsigned char f_initvolume(void)
```

Arguments

Argument	Description

Return values

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

Example:

```
void myinitfs(void) {
  unsigned char ret;

  /* Initialize Drive */

  ret=f_initvolume();

  if(ret)
       printf("Drive init error %d\n",ret);
  .
  .
}
```

See also

`f_format, f_hardformat, f_delvolume`

f_delvolume

This function is used to free resources associated with the volume. It works independently of the status of the hardware; i.e., it does not matter if a card is inserted or not.

Format

```
unsigned char f_delvolume(void)
```

Arguments

Argument	Description

Return values

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

Example:

```
void mykillfs(void) {
  unsigned char ret;

  /* Remove the volume */

  ret=f_delvolume();

  if(ret)
       printf("Drive delete error %d\n",ret);
  .
  .
}
```

See also

f_format, f_hardformat, f_initvolume

f_format

Format the drive. If the media is not present, this function 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

```
unsigned char f_format(unsigned char fattype)
```

Arguments

Argument	Description
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

Example:

```
void myinitfs(void) {
  unsigned char ret;

  f_initvolume();

  ret=f_format(F_FAT16_MEDIA);

  if(ret)
       printf("Unable to format drive: Error
  %d",ret);
  else
       printf("Drive formatted");

  .
  .
}
```

See also `f_initvolume, f_hardformat`

f_hardformat

Format the 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

```
unsigned char f_hardformat(unsigned char
fattype)
```

Arguments

Argument	Description
fattype	type of format: FAT12, FAT16 or FAT32

Return values

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

Example

```
void myinitfs(void) {
  unsigned char ret;

  f_initvolume();

  ret=f_hardformat(F_FAT16_MEDIA);
  if(ret)
       printf("Format Error: %d", ret);
  else
       printf("Drive 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: This function supports only drives up to 4GB in size.

Format

```
unsigned char f_getfreespace(F_SPACE *pspace)
```

Arguments

Argument	Description
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;
unsigned char ret;
  /* get free space on current drive */

  ret = f_getfreespace(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 the volume label. The label should be an ASCII string with a maximum length of 11 characters. Non-printable characters will be padded out as space characters.

Format

```
unsigned char f_setlabel(const char *pLabel)
```

Arguments

Argument	Description
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) {
  unsigned char ret;

  ret=f_setlabel(f_getcurrdrive(),"DRIVE 1");

  if (ret)
       printf("Error %d\n", ret);
}
```

See also

f_getlabel

f_getlabel

This writes the volume label to a defined buffer. The pointer passed for storage should be capable of holding 12 characters.

Format

```
unsigned char f_getlabel(
   char *pLabel, unsigned char len)
```

Arguments

Argument	Description
pLabel	pointer to buffer to store label in
len	length of buffer pointed to

Return values

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

Example

```
void getlabel(void) {
  char label[12];
  unsigned char ret;

  ret =
  f_getlabel(label,12);

  if (ret)
       printf("Error %d\n",ret);
  else
       printf("Drive is %s",label);
}
```

See also

```
f_setlabel
```

f_mkdir

Make a new directory.

Format

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

Arguments

Argument	Description
dirname	name of directory to create

Return values

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

Example

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

See also

f_chdir, f_rmdir

f_chdir

Change directory

Format

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

Arguments

Argument	Description
dirname	name of target directory

Return values

Return value	Description
F_NO_ERROR	Success
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_rmdir

Remove directory. If the target directory is not empty or it is read-only then this returns with an error.

Format

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

Arguments

Argument	Description
dirname	name of target directory

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_getcwd

Get current working folder.

Format

```
unsigned char 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 256

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

  ret = f_getcwd(buffer, BUFFLEN);
  if (!ret)
       printf ("current directory is %s",buffer);
  else
       printf ("Error %d", ret)
}
```

See also

```
f_chdir
```

f_rename

Rename a file or directory.

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

Format

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

Arguments

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

Delete a file. A read-only or open file cannot be deleted.

Format

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

Arguments

Argument	Description
filename	name of target file with/without path

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 file does not exist this function returns with zero.

Format

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

Arguments

Argument	Description
filename	name of target file with/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 the first file or subdirectory in specified directory. First call ***f_findfirst*** and then if file was found get the next file with ***f_findnext***.
Files with the system attribute set will be ignored.

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

Format

```
unsigned char 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("/subdir/*.*",&find)) {
       do {
            printf (“filename:%s”,find.filename);
            if (find.attr&F_ATTR_DIR) {
                 printf (“ directory\n”);
            }
            else {
            printf (“ size %d\n”,find.filesize);
            }
       } while (!f_findnext(&find));
  }
}
```

See also

`f_findnext`

f_findnext

Find the next file or subdirectory in a specified directory after a previous call to ***f_findfirst*** or ***f_findnext***. First call ***f_findfirst*** and, if a 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

```
unsigned char f_findnext(F_FIND *find)
```

Arguments

Argument	Description
find	find information (from ***f_findfirst***)

Return values

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

Example

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

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

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

Arguments

Argument	Description
filename	target 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

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

Arguments

Argument	Description
filename	name of target file
pctime	pointer where to store creation time
pcdate	pointer 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 function 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

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

Arguments

Argument	Description
filename	name of 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 function 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

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

Arguments

Argument	Description
filename	name of 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

Open a file. A file can be opened in the following modes:

"r" - open file for reading
"w" - open file for writing and truncate to zero length
"a" - open file for appending (append data to the end only)
"w+" - open file for writing and reading and truncate to zero length
"a+" - open the file for appending and reading
"r+" - open the file for reading and writing

Format

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

Arguments

Argument	Description
filename	name of target file
mode	mode open with

Return values

Return value	Description
F_FILE *	pointer to the associated opened file or zero if 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 1byte */
  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

```
unsigned char 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_flush

Flushes an open file to disk. This is logically equivalent to closing and then opening a file to ensure that the data changed before the flush is committed to the disk.

Format

```
unsigned char 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

f_write

Write data into file at the current file position. The current file position will be increased by the number of bytes successfully written. File must be opened with “w”, “w+”, "a+", "r+" or “a”.

Format

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

Arguments

Argument	Description
buf	buffer where data is
size	size of items to be written
size_t	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;
  }
  if (f_write(string,1,3,file)!=3)
  {                        /* write 3 items */
       printf (“different number of items written”);
  }

  f_close(file);
}
```

See also

f_read, f_open, f_close

f_read

Read bytes from the current file position. The current file pointer will be increased by the number of bytes read. File must be opened with “r”, "r+", "w+" or "a+".

Format

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

Arguments

Argument	Description
buf	buffer where to store data
size	size of items to be read
size_t	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 (“different number of items read”);
  }
  f_close(file);
  return 0;
}
```

See also

f_seek, f_tell, f_open, f_close, f_write

f_seek

Move the current file position. The file must be open. The **whence** parameter may be one of:

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

The *offset* position is relative to whence.

Format

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

Arguments

Argument	Description
filehandle	handle of 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 1 byte */
  f_seek(file,0,SEEK_SET);
  f_read(buffer,1,1,file); /*read the same 1 byte*/
  f_seek(file,-1,SEEK_END);
  f_read(buffer,1,1,file); /* read last 1 byte */
  f_close(file);
  return 0;
}
```

See also

f_read, f_tell

f_tell

Tell the current read/write position in the requested file.

Format

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

Arguments

Argument	Description
filehandle	handle of 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));
  f_read(buffer,1,1,file); /* read 1 byte */
  printf (“Current position %d”,f_tell(file));
  f_read(buffer,1,1,file); /* read 1 byte */
  printf (“Current position %d”,f_tell(file));
  f_close(file);
  return 0;
}
```

See also

f_seek, f_read, f_write, f_open

f_eof

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

Format

```
unsigned char f_eof(F_FILE *filehandle)
```

Arguments

Argument	Description
filehandle	handle of 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

Set the file position in the target open file to the file beginning.

Format

```
unsigned char f_rewind(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) {
  char buffer[4];
  char buffer2[4];
  F_FILE *file=f_open("myfile.bin",”r”);
  if (file) {
     f_read(buffer,4,1,file);
     f_rewind(file); /* rewind file pointer */
     f_read(buffer2,4,1,file);
                      /* read from beginning */
     f_close(file);
  }
  return 0;
}
```

See also

f_seek, f_read, f_write, f_open

f_putc

Write a character to the target open file at the current file position. The current file position is incremented.

Format

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

Arguments

Argument	Description
ch	character to be written
filehandle	handle of target file

Return values

Return value	Description
-1	Write Failed
value	Successfully written character

Example

```
void myfunc (char *filename, long num) {
  int ch='A';

  F_FILE *file=f_open(filename,”w”);
  while (num>0) {
       num--;
       if(ch != f_putc('ch', file))
       {
            printf("Error!!!");
            break;
       }
  }
  f_close(file);
  return 0;
}
```

See also

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

f_getc

Read a character from the current position in the target open file. The current file position will be incremented.

Format

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

Arguments

Argument	Description
filehandle	handle of target file

Return values

Return value	Description
-1	Failed
value	character which is read from file

Example

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

  while ((ch=f_getc(file)) != -1)
  {
       if (!buffsize) break;
       *buffer++=ch;
       buffsize--;
  }

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

See also

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

f_seteof

Move the end of file to the current file pointer. All data after the new EOF position are 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
```

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`

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

THIN calls the following driver functions:

- ***drv_initfunc***
- ***drv_getphy***
- ***drv_readsector***
- ***drv_writesector***
- **drv_getstatus**

Within the header file of each driver module (e.g., in **mmc.h** or **cfc.h**), there must be a definition linking these names to the physical functions for that driver, as in this example.

```
#define drv_initfunc xxx_initfunc
#define drv_getphy xxx_getphy
#define drv_readsector xxx_readsector
#define drv_writesector xxx_writesector
#define drv_getstatus xxx_getstatus
```

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, or xxx=mmc for MultiMediaCard driver.

The ***drv_initfunc*** routine is compulsory and is called by ***f_initvolume*** to initialize a volume.

The ***drv_getphy*** routine is compulsory and is called by the file system to determine the physical properties (e.g., number of sectors) of the device.

The ***drv_readsector*** routine is compulsory and is used to read a sector in the file system.

The **drv_writesector** routine is optional and is required in order to write to the file system. It is also necessary if format is required.

The ***drv_getstatus*** routine is optional and is used only to discover their status of removable media (e.g., whether a card has been removed or changed).

drv_initfunc

This function is called by the file system to initialize the volume.

Format

```
unsigned char drv_initfunc(void)
```

Arguments

Argument	Description

Return values

Return value	Description
0	Always successful

drv_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.

Currently only the number of sectors is used by the upper level.

Format

```
unsigned char drv_getphy(F_PHY *pPhy)
```

Arguments

Argument	Description
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 */
} F_PHY;
```

drv_readsector

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

Format

```
unsigned char drv_readsector(void *data,
                 unsigned long sector)
```

Arguments

Argument	Description
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

drv_writesector

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

Note: This function may be omitted if the drive is read-only.

Format

```
unsigned char drv_writesector(void *data,
        unsigned long sector)
```

Arguments

Argument	Description
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

drv_getstatus

This function is called by the file system to check the status of the media. It 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 the drive contains only permanent media (e.g., hard disk or RAM drive), this function may be omitted.

Format

```
unsigned char drv_getstatus(void)
```

Arguments

None

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)

6 Compact Flash Card

Overview

The Compact Flash Card (CFC) driver is designed to operate with all standard compact flash cards of 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

Currently there is no sample driver available for the PC I/O mode. Those who wish to use the PC I/O mode should contact HCC-Embedded for further information.

Chip Select and Wait States

The drivers contain no chip select control. This must be set up for the specific hardware design. In particular the developer should check the wait state setting if the hardware generates a delay greater than 300ns for accessing the CFC.

Porting True IDE Mode

Files

There are two 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

Parts of the THIN code are hardware design-specific and must be modified to meet the hardware requirements.

The following are the header file definitions that must be modified

CFC_TOVALUE - this value is hardware-dependent and is a counter for loop expiry. It may be replaced with a host OS timeout function.

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)

Hardware I/Os Required

The HCC sample driver uses three I/O pins to control the compact flash interface:

CFC_CDPORT	- Card removed
CFC_PWPORT	- Power On
CFC_RSTPORT	- Card reset

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

Additionally a card changed state must be generated. The logic for this is shown in the cd_irq() function of the sample driver.

Setting IDE Mode

A special sequence is used to force the compact flash card into IDE mode. This is achieved in HCC hardware by a CPLD logic which:

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

This sequence must also be executed in the *cfc_getstatus* routine.

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

Porting PC Memory I/O Mode Driver

The driver consists of two files: **cfc.c** and **cfc.h**.

The following sections describe the modifications that may be necessary.

Addresses

The **cfc.h** file contains a set of CFC register definitions that are set up for a particular test system used by HCC-Embedded. These settings should be checked against the hardware design and then they must be set appropriately. The definitions that must be checked are:

CFC_BASE /* sets the base address for accessing the CF Card */

CFC_SECTORNOCOU
CFC_CYLINDERHILO
CFC_COMMANDSELC
CFC_STATESELC
CFC_DATAHILO
CFC_DRVADDRESSHI

In addition there are two ports that are hardware-specific. HCC accesses them through CPLD logic. These are:
CFC_CPLDSTATE (*(volatile unsigned long*)(CFC_BASE+0x1000))

Reading this has these meanings:

CFC_CPLDSTATE_CDCH 0x08 /* card has been changed */
CFC_CPLDSTATE_CFCD 0x04 /* card has been removed */

The default settings provided for these are for interoperability with an HCC-Embedded reference design and in particular the CPLD.

Further Information

HCC-Embedded provides design and consultancy services for developers implementing Compact Flash Cards. HCC-Embedded also offers several 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. They can be used exactly in the same manner as MMCs but have additional functionality. In particular they have an additional two interface pins.

When used in Secure Digital mode there are three methods for communicating with the card.

1. 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.

2. MultiMediaCard Mode

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

3. 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). However, 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).

Implementation

THIN provides a generic MMC/SD card driver, which can be found in the **/mmc/** directory. Normally this driver does not require modification.

The MMC driver includes certain configuration options that may be set in **mmc_opt.h**.

The SDHC_ENABLE option enables support for SDv2 cards and SDHC cards. Disabling this option reduces code size but also reduces card compatibility.

Sample SPI drivers are included in the driver sub-directories. These must be ported for a particular target.

USE_CRC

To save further space or to increase the performance of the interface, the USE_CRC define in **mmc_opt.h** may be set to zero. Even in its assembler-optimized form the CRC calculation can put a significant load on an 8 bit microprocessor. USE_CRC tells the driver to use CRCs in the communications with the flash card. Generally communication is very reliable and this will not introduce significant errors. However, if cards are badly inserted or if the socket becomes overly worn, then communication reliability may deteriorate more quickly than expected and errors may appear in files**. Disabling of CRC checking is not recommended if the data used are of a critical nature.**

CRC_ROM_TABLE

If USE_CRC is enabled then this allows the developer to choose whether the CRC is generated and put in RAM or is programmed permanently into flash.

Porting the SPI Driver

The sample drivers are included to provide 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_tx512 (unsigned char *buf)
```

Transmits 512 bytes through the SPI port. This may simply call **spi_tx1()** 512 times.

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

Receives a single byte.

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

Receives 512 bytes.

```
void spi_cs_lo (void)
```

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

```
void spi_cs_hi (void)
```

Sets 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.

Further Information

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

8 RAM Driver

It is simple to implement a RAM drive for the file system. There is no physical driver associated with the RAM drive.

The RAM driver is a good starting point for implementing a new physical driver.

Note: Building a RAM driver requires a large amount of RAM and therefore it may not be possible to do this on certain systems with limited RAM. The typical minimum size of RAM we recommend using for a FAT12 RAM drive is 32K
The minimum size of a FAT12 RAM drive is 36 sectors (18K), which allows just one sector (512 bytes) for file storage.

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.

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 block size of the RAM you wish to use for this drive. This is statically assigned. If you require it to be malloc'd this is a minor change. Also note that 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.

3. Call ***f_initvolume*** to initialize the drive.

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

```
void main(void){

      /* Initialize the RAM drive */

      f_initvolume();


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

      f_hardformat(F_FAT12_MEDIA);  /* create FAT12 in RAM */

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

The RAM drive may now be used as a standard drive.

9 Optimization

The C source code for THIN has been carefully crafted to minimize code size and to maximize execution speed. Additionally provided is a set of defines in **thin_usr.h** that can be used to select the available functionality in your file system.

API Function Selection

By defining functions to be included in, or to be excluded from the file system, the amount of space used by the file system can be controlled. This is more manageable than using libraries where adding or removing a piece of code can cause unpredictable changes in the size of your code.

The following defines are available in **thin_usr.h** to enable/disable the availability of API functions:

F_GETVERSION
F_SETLABEL
F_GETLABEL
F_GETFREESPACE
F_CHDIR
F_MKDIR
F_RMDIR
F_RENAME
F_DELETE
F_GETTIMEDATE
F_SETTIMEDATE
F_GETATTR
F_SETATTR
F_FILELENGTH
F_FINDING
F_TELL
F_GETC
F_PUTC
F_REWIND
F_EOF
F_SEEK
F_WRITE

Other Build Options

There are several other options that allow the developer to focus the file system to do only what is required. These are listed below.

F_FORMATTING

This option enables/disables the availability of the formatting commands. The code for formatting requires a significant amount of ROM and RAM to run. Generally, when using flash devices, they come pre-formatted and can also be formatted on a standard PC. Therefore it is not normally necessary to include formatting functions in a deeply embedded system for handling flash cards.

F_MAX_FORMAT_SIZE

This option specifies the maximum size in Mbytes of the target media to use when formatting. If this value is set to zero then the driver is formatted on the basis of the physical size of the media returned by the *drv_getphy()* driver call.

F_WRITING

This option enables/disables the ability to write to a flash card. If writing is disabled then the file system is read-only. If this option is used, a relatively large amounts of code and RAM space will be saved.

F_DIRECTORIES

This option enables/disables the use of directories. If directories are disabled then all file access must be in the root directory. This reduces the code space required and can also reduce the RAM required; in particular the F_MAXPATH may be reduced (see below).

F_CHECKNAME

This option disables the checking and validation of filenames. On many embedded systems, only valid file names will be used, and as a consequence the complicated and CPU-consuming checking routines can be disabled.

F_LONGFILENAME

This option enables/disables long filename support. Long filename support generates substantially more code in the file system and also requires more RAM to be used since the longer names have to be accommodated. Additionally it should be noted that using long filenames may place a significant CPU overhead on a small device because of the more complex handling required.

F_FAT12/F_FAT16/F_FAT_32

This option enables/disables the use of the different FAT formats. Generally on flash cards those less than 32MB are formatted as FAT12 and those less than 2GB are formatted as FAT16.

FAT12 cannot be used on cards greater than 32MB in size.
FAT16 cannot be used on cards greater than 2GB.
If cards greater than 2GB are used then FAT32 must be enabled.

F_MAXFILES

This defines the maximum number of files that may be open simultaneously. Additionally, if long filenames are used, this number must be one greater than the number of simultaneous files that may be open. If long filenames are enabled, F_MAXFILES has a minimum value of 2, otherwise it has a minimum value of 1.

Limiting the maximum number of files that are open reduces the RAM requirement of the system. For every additional file allowed to be open, 0.5K will be added to the RAM requirements of the system.

F_MAXPATH

This defines the maximum path length that the file system will handle if long filenames are NOT used. The default value for this is 128. The worst case value for this on a PC is 260 but in practice on embedded devices much smaller and often predictable path lengths can be relied upon. Using a smaller maximum path length reduces the RAM requirements of the system.

F_MAXLNAME

This defines the maximum path length that the file system will handle if long filenames are used. The default value is 128. The worst case value on a PC is 260 but in practice on embedded devices much smaller and often predictable path lengths can be relied upon. Using a smaller maximum path length reduces the RAM requirements of the system.

FATBITFIELD_ENABLE

This define enables an option for the system to maintain a bitmap record of the FAT clusters which do not contain any free clusters. If enabled this option uses more code and significantly more RAM. The actual amount depends on the size of the device you attach and the FAT type. But this option also greatly accelerates the search for a free cluster in the FAT, particularly on a full card. This will result in far fewer FAT accesses and hence reduced power consumption.
Note: If FATBITFIELD_ENABLE is enabled, calls to be *malloc()* will be made from *f_getvolume()*. Facilities for handling this must included.

F_GETFREESPACE_QUICK

If enabled this option causes the system to maintain a record of the changes in free space. If *f_getfreespace()* is used then at some stage the whole FAT must be scanned for the amount of free space. If this option is not enabled a complete scan must be done on each call to *f_getfreespace()*. If this option is enabled then the FAT is scanned only once and changes are recorded. This option requires more code and a little more RAM. There is also a considerable improvement in performance and efficiency and hence reduced power consumption if this option is used.

Hints and Tips

Merging files

Some compilers can do better size optimization if all the code is contained in one file. Particularly on smaller processors this is useful by finding common pieces of code and turning them into a single call. There are two approaches to this:

1. Combine all the source files in **src/fat_thin** into a single file
2. Create a master file that contains just a list of the source files to include. The compiler thus treats the files as a single source.

Power Consumption

To use the minimum power when accessing your flash device it is important to minimize the number of accesses. If it is possible to design the application so that a large file is created before use, and then you modify the file using only ***f_seek***, this ensures that there is no need to update the FAT each time a new block is appended. This can be a useful mechanism for conserving power in a data-logging application.

Safety

FAT file systems are by design not power failsafe. If power is lost at the “wrong” moment, part or all of the file system can be lost. Normally part or all of the lost data can be recovered using PC-based disk recovery software. One method to reduce the risk of losing the whole device is to put files only in sub-directories; i.e., do not use the root directory for storing files.

Note: THIN is vulnerable to corruption only when files are being written, in particular when the FAT or directory entries are being updated.

10 Test Code

Provided with the system is a set of test routines for exercising the file system and ensuring that it is behaving correctly.

Note: On some systems the test code may be difficult or impossible to run because of the lack of resources. Also note that the test code depends on what features of the file system have been enabled.

The test code is in the files **test.c** and **test.h**.

There is a table in **test.h** for defines that must be enabled for a particular test to run. The test suite is automatically built for the set of defines in **thin_usr.h**.

To run the tests simply call ***f_dotest*** with the parameter of the number of the test you want to run or zero if you want to run all the available tests.

Note that seek tests use more RAM. F_MAX_SEEK_TEST, in **test.h** limits the maximum size of the seek test to be done.

The F_FAT_TYPE must be defined in the **test.h** file to determine whether the tests will be executed on a FAT12, 16 or 32 card.