THIN File System User's Guide

Version 3.40

For use with THIN File System Versions 5.1 and above

Date: 23-May-2014 11:47

Table of Contents

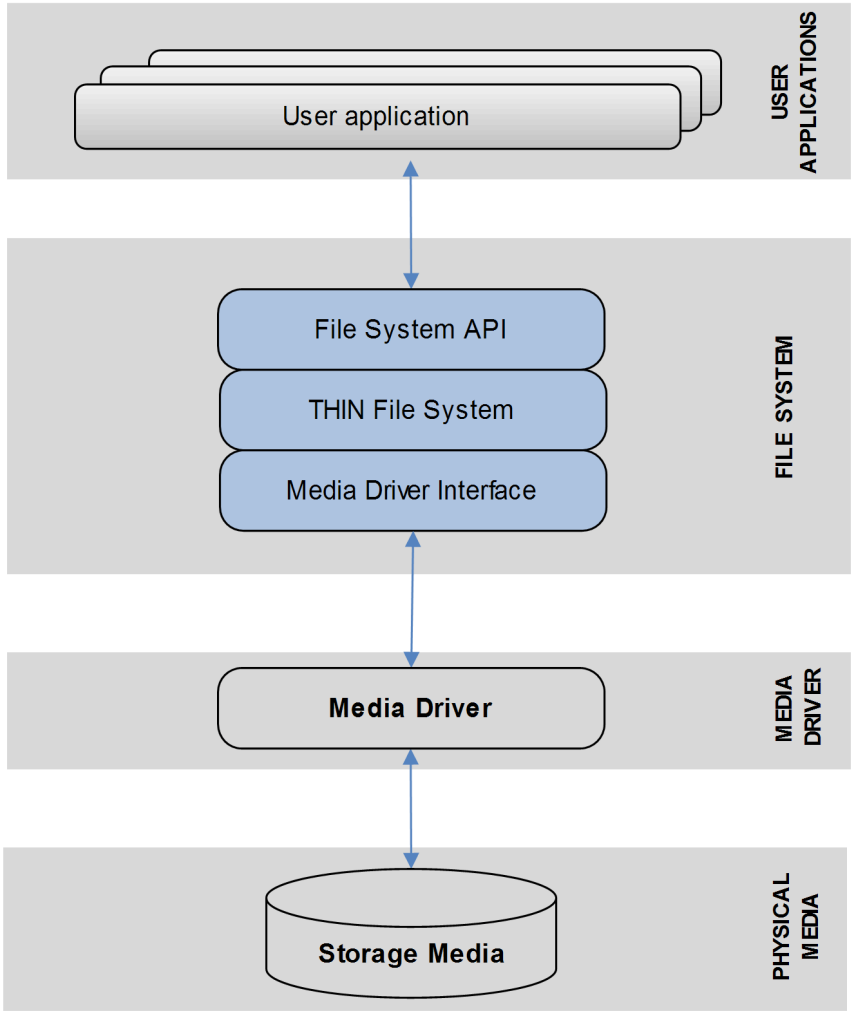
1 System Overview

1.1 Introduction

This guide is intended for use by embedded software engineers who have a knowledge of the C programming language and standard file APIs. It is for those who want to implement a full-featured FAT file system, optimized to use minimal ROM/RAM.

The THIN file system makes use of media drivers to access one or more storage media to execute the requested storage operation. THIN can access any combination of storage device types that conform to the *HCC Media Driver Interface Specification*.

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



The File System API is the interface used by the user application to access the THIN file system and the attached storage media.

The THIN file system:

- Is a FAT-compatible file system designed for embedded microcontrollers with limited system resources (restrictions on the available code space or available RAM).
- Has a code size of from 4 to 12.5KB and requires from 1.5 to 2KB of RAM.
- Provides a balance of speed against memory needed, with options that allow you to make performance trade-offs using available resources. This permits a full file system to be run on a low cost microcontroller with limited resources.
- Allows developers to attach PC-compatible media like SD cards or pen drives to their systems. It is compatible with media such as SD/MMC and Compact flash cards.
- Can use any media driver that conforms to HCC's *Media Driver Interface Specification*. The system is limited to using only a single media driver at any time; before a second media driver can be used, the first volume must be deleted.

Note:

- For developers who have even more limited resources, HCC Embedded's SuperTHIN file system is the recommended option.
- For developers who do not have such limited resources, but have **>**20KB for code and **>**5KB for RAM, HCC Embedded's FAT file system is the recommended option.
- Although every attempt has been made to simplify the system's use, you need a good understanding of the requirements of the systems you are designing in order to obtain the maximum practical benefits. HCC Embedded offers hardware and firmware development consultancy to help developers implement a flash file system.

1.2 Feature Check

The main features of the system are the following:

- Code size 4-12.5KB.
- RAM usage 1.5-2KB.
- ANSI 'C'.
- Reentrant.
- Support for long filenames.
- Support for multiple open files.
- Test suite provided.
- Zero copy.
- FAT 12/16/32-compatible.

1.3 Packages

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

Package	Description
hcc_base_doc	This contains the two guides that will help you get started.
fs_thin	The THIN file system package described in this manual.
media_drv_base	The Media Driver base package that provides the base for all media drivers that attach to the file system.
media_drv_ram	The RAM Media Driver package, used for creating a RAM drive. This is provided as a reference driver.

Additional packages

Other packages may also be provided to work with THIN. Examples include specific media drivers for particular targets, and PSP extensions for particular targets.

1.4 Documents

HCC Firmware Quick Start Guide

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

HCC Source Tree Guide

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

HCC Media Driver Interface Specification

This document describes the media driver interface.

HCC THIN File System User's Guide

This is this document.

1.5 Disclaimer

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

1.6 About HCC Embedded

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

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

1.7 Getting Help

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

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

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

2 Source File List

This section describes all the source code files included in the system. These files follow the HCC Embedded standard source tree system, described in the *HCC Source Tree Guide*. All references to file pathnames refer to locations within this standard source tree, not within the package you initially receive.

> **Note:** Do not modify any files except the configuration file.

2.1 API Header File

The file **src/api/api_thin.h** should be included by any application using the system. This is the only file that should be included by an application using this module. For details of the functions, see API.

2.2 Configuration File

The file **src/config/config_thin.h** contains all the configurable parameters of the system. Configure these as required. This is the only file in the module that you should modify. For details of these options, see Configuration Options.

2.3 Version File

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

2.4 Test Files

The test code is in files named **test.c** and **test.h** in the folder **src/fat_thin/test**.

2.5 THIN File System

These files should only be modified by HCC.

File	Description
src/fat_thin/common/dir.c	Directory handling functions without long filenames.
src/fat_thin/common/dir.h	Header file for short filename directory functions.
src/fat_thin/common/dir_lfn.c	Directory handling functions with long filenames.
src/fat_thin/common/dir_lfn.h	Header file for long filename directory functions.
src/fat_thin/common/drv.c	Low level driver interface functions.
src/fat_thin/common/drv.h	Header file for low level driver interface functions.
src/fat_thin/common/f_rtos.c	RTOS functions.
src/fat_thin/common/f_rtos.h	Header file for RTOS functions.
src/fat_thin/common/fat.c	FAT file system general functions.
src/fat_thin/common/fat.h	Header file for FAT file system general functions.
src/fat_thin/common/file.c	File manipulation functions.
src/fat_thin/common/file.h	Header file for file manipulation functions.
src/fat_thin/common/util.c	General utility functions.
src/fat_thin/common/util.h	Header file for general utility functions.
src/fat_thin/common/util_lfn.c	General utility functions for long filenames.
src/fat_thin/common/util_lfn.h	Header file for general utility functions for long filenames.
src/fat_thin/common/util_sfn.c	General utility functions for short filenames.
src/fat_thin/common/util_sfn.h	Header file for general utility functions for short filenames.
src/fat_thin/common/volume.c	Volume manipulation functions.
src/fat_thin/common/volume.h	Header file for volume manipulation functions.

3 Configuration Options

3.1 Options

Set the system configuration options in the file **src/config/config_thin.h**. This section lists the available configuration options and their default values.

The options listed below allow you to focus the file system to do only what is required.

F_SECTOR_SIZE

The sector size of the target media, for use when formatting. The default is 512u.

RTOS_SUPPORT

Set this to 1 to enable RTOS support. The default value is zero. The OS Abstraction Layer (OAL) is only used when this is enabled.

F_LONGFILENAME

This enables/disables long filename support. The default is zero.

Long filename support generates substantially more code in the file system. It also requires more RAM since the longer names have to be accommodated. Among other things, the stack sizes of applications that call the file system must be increased, and more checking is required. Additionally, note that using long filenames may place a significant CPU overhead on a small device because of the more complex handling required.

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 to reduce the resource usage of the system.

The most critical function for long filenames is **f_rename(),** which must keep two long filenames on the stack as well as additional structures for handling it.

> **Note: Do not modify the structure F_LFNINT** as this is used to process the files on the media which may be created by other systems.

Choose one of the following sets of source files:

- **dir.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.

FATBITFIELD_ENABLE

This enables/disables the system's keeping of a bitmap record of the FAT clusters which do not contain any free clusters. The default is zero.

If it is 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 results in far fewer FAT accesses and hence reduced power consumption.

> **Note:** If FATBITFIELD_ENABLE is enabled, **psp_malloc()** will be called from **f_getvolume()** to allocate space for this table.

F_MAXFILES

The maximum number of files that may be open simultaneously. The default is 1. If long filenames are used:

- F_MAXFILES must be one greater than the number of files that may be open simultaneously.
- 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, 0.5KB is added to the RAM requirement.

F_MAXPATH

The maximum path length that the file system handles if long filenames are NOT used. The default value is 64.

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

The maximum path length that the file system will handle if long filenames are used. The default is 64. 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.

F_FILE_CHANGED_EVENT

Set this to 1 if you want to make a file state change an event. The default is zero.

3.2 Hints and Tips for Optimization

This section outlines other ways to improve performance.

Merging files

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

- Combine all the source files in **src/fat_thin/common** into a single file.
- Create a master file that contains just a list of the source files to include. The compiler then 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 you can 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 fail-safe. 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; that is, 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.

4 Drive Format

THIN handles most of the features of a FAT file system with no need for explanation of the underlying issues. However, this section describes some areas which you do need to understand.

Removable media may be formatted in three different ways:

- Completely unformatted.
- Master Boot Record.
- Boot Sector Information only.

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

4.1 Completely Unformatted Media

An unformatted drive is not useable until it has been formatted. Most flash cards are pre-formatted, whereas hard disk drives tend to be unformatted when delivered. When **f_format()** is called, the drive is formatted with Boot Sector Information. This is exactly the same as if **f_hardformat()** had been issued at any time.

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

4.2 Master Boot Record (MBR)

As standard, the file system does not hard format a card with an MBR but with Boot Sector Information. A hard format will remove the MBR information.

When a device with an MBR is inserted, it is treated as if it has just one partition (the first in the partition table).

4.3 Boot Sector Information

If **f_hardformat()** is called, the card is always formatted with the Boot Sector Information table in the first sector.

5 API

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

5.1 Module Management

fs_init

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

Data areas for the file system to use are allocated at compile time, based on the settings for each volume in the **config_thin.h** file.

Format

```
unsigned char fs_init ( void )
```

Arguments

Argument
None.

Return values

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

Example

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

fs_delete

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

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

Format

```
unsigned char fs_delete ( void )
```

Arguments

Argument
None.

Return values

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

5.2 File System API

The functions are divided into four groups: volume management, directory management, file access, and file management.

Volume Management

f_initvolume

Use this function to initialize the volume.

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

Format

```
unsigned char f_initvolume (
    F_DRIVERINIT    initfunc,
    unsigned long   driver_param )
```

Arguments

Argument	Description	Type
initfunc	Initialization function.	F_DRIVERINIT
driver_param	Driver parameter.	unsigned long

Return values

Argument	Description
F_NO_ERROR	Drive successfully initialized.
Else	See Error Codes.

Example

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

    /* Initialize file system */
    f_init();

    /* Make a volume on RAM */
    ret = f_initvolume(ram_initfunc, 0));

    if (ret != F_NO_ERROR)
    {
        printf("Volume initialization Failed");
    }
    else
    {
        /* Volume Ready for Use */
        .
        .
        .
    }
}
```

f_delvolume

Use this function to free resources associated with the volume.

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

Format

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

Arguments

Argument
None.

Return values

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

Example

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

    /* Initialize Drive */

    ret=f_delvolume();

    if(ret != F_NO_ERROR)
        printf("Unable to delete volume, Error: %d\n",ret);
  .
  .
}
```

f_format

Use this function to format the specified drive.

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

Any existing Master Boot Record is unaffected by this command. The Boot Sector Information is re-created from the information provided by **f_getphy()**.

> **Note:** The format fails 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	Type
fattype	Type of format: • F_FAT12_MEDIA for FAT12 • F_FAT16_MEDIA for FAT16 • F_FAT32_MEDIA for FAT32	unsigned char

Return values

Return value	Description
F_NO_ERROR	Successful execution.
Else	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”);
  .
  .
}
```

f_hardformat

Use this function to format the drive, ignoring current format information. All open files will be closed.

This destroys any existing Master Boot Record or Boot Sector Information. The new drive is formatted without a master boot record. The new drive starts with boot sector information created from the information retrieved from the **f_getphy()** routine, and uses the whole available physical space for the volume. All data on the drive are destroyed.

> **Note:** The format fails 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	Type
fattype	Type of format: • F_FAT12_MEDIA for FAT12 • F_FAT16_MEDIA for FAT16 • F_FAT32_MEDIA for FAT32	unsigned char

Return values

Return value	Description
F_NO_ERROR	Successful execution.
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”);
  .
  .
}
```

f_getlabel

Use this function to write the volume label to a defined buffer.

Format

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

Arguments

Argument	Description	Type
label	Pointer to buffer to store label in. This should be capable of holding 12 characters.	char *
len	Length of buffer pointed to.	unsigned char

Return values

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

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);
}
```

f_setlabel

Use this function to set 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 * label )
```

Arguments

Argument	Description	Type
label	Pointer to null-terminated string to use.	char *

Return values

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

Example

```
void setlabel(void)
{
    unsigned char ret;
    ret=f_setlabel(f_getcurrdrive(),"DRIVE 1");
    if (ret)
        printf("Error %d\n", ret);
}
```

f_getfreespace

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

Note:

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

Format

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

Arguments

Argument	Description	Type
pspace	Pointer to F_SPACE structure.	F_SPACE *

Return values

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

Example

```
void info(void)
{
    F_SPACE space;
    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_getserial

Use this function to get the volume's serial number.

Format

```
unsigned char fn_getserial ( unsigned long * serial)
```

Arguments

Argument	Description	Type
serial	Where to store the serial number.	unsigned long *

Return values

Return value	
Zero	Successful execution.
Else	See Error Codes.

Directory Management

f_mkdir

Use this function to create a new directory.

Format

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

Arguments

Argument	Description	Type
dirname	Name of directory to create.	char *

Return values

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

Example

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

f_chdir

Use this function to change the current working directory.

Format

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

Arguments

Argument	Description	Type
dirname	Name of target directory.	char *

Return values

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

Example

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

f_rmdir

Use this function to remove a directory.

The function returns an error code if:

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

Format

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

Arguments

Argument	Description	Type
dirname	Name of directory to remove.	char *

Return values

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

Example

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

f_getcwd

Use this function to get the current working directory.

Format

```
unsigned char f_getcwd (
    char *          buffer,
    unsigned char   maxlen
    char            root )
```

Arguments

Argument	Description	
buffer	Where to store the current working directory string.	char *
maxlen	Length of the buffer.	unsigned char
root	The root.	char

Return values

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

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)
}
```

File Access

f_open

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

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

Note the following:

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

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

Format

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

Arguments

Argument	Description	Type
filename	File to be opened.	char *
mode	The opening mode (see above).	char *

Return values

Return value	Description
F_FILE *	Pointer to the associated opened file handle.
Zero	File could not be opened.

Example

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

    file=f_open(“myfile.bin”,”r”);
    if (!file)
    {
        printf (“File cannot be opened!”);
        return;
    }
    f_read(&c,1,1,file); /*read 1 byte */
        printf (“’%c’ is read from file”,c);
    f_close(file);
}
```

f_close

Use this function to close a previously opened file.

Format

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

Arguments

Argument	Description	Type
filehandle	Handle of target file.	F_FILE *

Return values

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

Example

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

    file=f_open(“myfile.bin”,”w”);

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

    f_write(string,3,1,file); /*write 3 bytes */
    if (!f_close(file))
    {
        printf (“file stored”);
    }
    else printf (“file close error”);
}
```

f_flush

Use this function to flush 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 * f )
```

Arguments

Argument	Description	
f	Handle of target file.	F_FILE *

Return values

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

Example

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

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

f_read

Use this function to read bytes from the current file position. The current file pointer is increased by the number of bytes read. The file must be opened in “r”, "r+", "w+" or "a+" mode.

Format

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

Arguments

Argument	Description	Type
buf	Buffer to store data in.	void *
size	Size of items to be read.	long
size_t	Number of items to be read.	long
filehandle	Handle of target file.	F_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;
}
```

f_write

Use this function to write data into a file at the current file position. The current file position is increased by the number of bytes successfully written. The 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	Type
buf	Buffer which contains the data.	void *
size	Size of items to be written.	long
size_t	Number of items to be written.	long
filehandle	Handle of target file.	F_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);
}
```

f_getc

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

Format

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

Arguments

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

Return values

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

Example

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

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

f_putc

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

Format

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

Arguments

Argument	Description	Type
ch	Character to be written.	char
filehandle	Handle of open target file.	F_FILE *

Return values

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

Example

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

f_eof

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

Format

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

Arguments

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

Return values

Return value	Description
Zero	Not at end of file.
Else	End of file or an error; see Error Codes.

Example

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

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

    return 0;
}
```

f_seteof

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

All data after the new EOF position are lost.

Format

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

Arguments

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

Return values

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

Example

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

    f_seek(file,position,SEEK_SET);

    if(f_seteof(file))
        printf(“Truncate Failed\n”);

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

f_tell

Use this function to obtain the current read-write position in the open target file.

Format

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

Arguments

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

Return values

Return value	Description
filepos	Current read or write file position.

Example

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

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

    f_read(buffer,1,1,file); /* read 1 byte
    printf (“Current position %d”,f_tell(file));
    /* position 2 */

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

f_seek

Use this function to move the stream position in the target file. The file must be open.

Format

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

Arguments

Argument	Description	Type
filehandle	Handle of target file.	F_FILE *
offset	Relative byte position according to whence.	long
whence	Where to calculate offset from: • F_SEEK_CUR – current position of file pointer. • F_SEEK_END – end of file. • F_SEEK_SET – beginning of file.	unsigned char

Return values

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

Example

```
int myreadfunc(char *filename, char *buffer, long buffsize)
{
    F_FILE *file=f_open(filename,”r”);
    f_read(buffer,1,1,file); /* read 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;
}
```

f_rewind

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

Format

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

Arguments

Argument	Description	Type
filehandle	Handle of target file.	F_FILE *

Return values

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

Example

```
void myfunc(void)
{
    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;
}
```

f_truncate

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

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

Format

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

Arguments

Argument	Description	Type
filename	File to be opened.	char *
length	New length of file.	unsigned long

Return values

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

Example

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

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

File Management

f_delete

Use this function to delete a file.

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

Format

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

Arguments

Argument	Description	Type
filename	Null-terminated string with name of file to be deleted, with or without path.	char *

Return values

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

Example

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

f_findfirst

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

First call **f_findfirst()** and then, if the file is found, get the next file with **f_findnext()**. Files with the system attribute set are ignored.

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

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

Format

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

Arguments

Argument	Description	Type
filename	Name of file to find.	char *
find	Where to store the file information.	F_FIND *

Return values

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

Example

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

f_findnext

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

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

> **Note:** If this function 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	Type
find	Find information (created by calling **f_findfirst()**).	F_FIND *

Return values

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

Example

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

f_rename

Use this function to rename a file or directory.

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

Format

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

Arguments

Argument	Description	Type
filename	Target file or directory name, with or without path.	char *
newname	New name of file or directory (without path).	char *

Return values

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

Example

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

f_getattr

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

Format

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

Arguments

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

Return values

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

Example

```
void myfunc(void)
{
    unsigned char attr;
    /* find if myfile is read only */
    if(!f_getattr("myfile.txt",&attr)
    {
        if(attr & F_ATTR_READONLY)
            printf("myfile.txt is read only");
        else
            printf("myfile.txt is writable");
    }
    else
        printf("file not found");
}
```

f_setattr

Use this function to set the file attributes (F_ATTR_XXX) of a file.

> **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	Type
filename	Name of target file.	char *
attr	The new attribute settings.	unsigned char

Return values

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

Example

```
void myfunc(void)
{
    /* make myfile read only and hidden*/

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

f_gettimedate

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

Date and Time Formats

The date and time fields are two 16 bit fields associated with each file/directory. If FAT compatibility is required, these must use the standard type definitions for time and date given below. If FAT compatibility is not required, you can use these fields as you require. See PSP Porting for information on porting.

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

Argument	Valid values	Format
Day	0-31	(d & 001fH)
Month	1-12	((d & 01e0H) >> 5)
Years since 1980	0-119	((d & fe00H) >> 9)

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

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

Format

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

Arguments

Argument	Description	Type
filename	Name of target file.	char *
pctime	Where to store the creation time.	unsigned short *
pcdate	Where to store the creation date.	unsigned short *

Return values

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

Example

```
void myfunc(void)
{
    unsigned short t,d;
    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 be retrieved!”
    }
}
```

f_settimedate

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

Date and Time Formats

The date and time fields are two 16 bit fields associated with each file/directory. If FAT compatibility is required, these must use the standard type definitions for time and date given below. If FAT compatibility is not required, you can use these fields as you require. See PSP Porting for information on porting.

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

Argument	Valid values	Format
Day	0-31	(d & 001fH)
Month	1-12	((d & 01e0H) >> 5)
Years since 1980	0-119	((d & fe00H) >> 9)

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

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

Format

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

Arguments

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

Return values

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

Example

```
void myfunc(void)
{
    f_mkdir(“subfolder”); /*creating directory*/
    f_settimedate(“subfolder”,f_gettime(),f_getdate());
}
```

5.3 Types and Definitions

F_FILE: File Handle

The file handle, used as a reference for accessing files.

The handle is obtained when a file is opened and released when closed.

F_FIND Structure

The F_FIND structure takes this form:

Name	Description	Type
filename[F_MAXPATH]	File name+extension.	Char.
name[F_MAXNAME]	File name.	Char.
ext[F_MAXEXT]	File extension.	Char.
attr	File attribute.	Unsigned char.
ctime	Creation time.	Unsigned short.
cdate	Creation date.	Unsigned short.
cluster	For internal use only.	Unsigned long.
filesize	Length of file.	Long.
findfsname	For internal use only.	F_NAME.
pos	For internal use only.	F_POS.

Note: The F_NAME and F_POS structures are for file system internal use only.

File Attribute Settings

The following possible file attribute settings are defined by the FAT file system:

Attribute Bit Definition	Description
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.

F_SPACE Structure

The structure takes this form:

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

5.4 Error Codes

The table below lists all the error codes that may be generated by the API calls. Please note that some error codes are not used by every file system.

Error Code	Value	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	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.

Error Code	Value	Meaning
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_ALLOCATION	28	Memory allocation error.
F_ERR_OS	29	Only possible if RTOS_SUPPORT is enabled.

6 Integration

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

> **Note:** THIN only uses the OS Abstraction Layer (OAL) when RTOS support is enabled.

6.1 Requirements

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 <0.5KB of stack. However, if long filenames are used, increase the stack size to 1KB; see F_LONGFILENAME.

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. Modify the delays 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.

6.2 OS Abstraction Layer (OAL)

When RTOS Support is enabled, the module uses the OS Abstraction Layer (OAL) that allows it to run seamlessly with a wide variety of RTOSes, or without an RTOS.

The system uses the following OAL components:

OAL Resource	Number Required
Tasks	0
Mutexes	1
Events	0

6.3 PSP Porting

The Platform Support Package (PSP) is designed to hold all platform-specific functionality, either because it relies on specific features of a target system, or because this provides the most efficient or flexible solution for the developer.

The THIN system makes use of the following standard PSP functions:

Function	Package	Package Element	Description
psp_free()	psp_base	psp_alloc	Deallocates a block of memory allocated by **psp_malloc()**, making it available for further allocation.
psp_getcurrenttimedate()	psp_base	psp_rtc	Returns the current time and date. This is used for date and time-stamping files.
psp_getrand()	psp_base	psp_rand	Generates a random number. This is used for the volume serial number.
psp_malloc()	psp_base	psp_alloc	Allocates a block of memory, returning a pointer to the beginning of the block. This is only used if the FATBITFIELD_ENABLE option is enabled.
psp_memcpy()	psp_base	psp_string	Copies a block of memory. The result is a binary copy of the data.
psp_memset()	psp_base	psp_string	Sets the specified area of memory to the defined value.

The system does not make use of any standard PSP macros.

7 Test Routines

A set of test routines is provided for exercising the file system and ensuring that it behaves correctly. The test code is in the folder **src/fat_thin/test** in files named **test.c** and **test.h**.

> **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 the features of the file system which you enable.

7.1 Running Tests

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

Note the following:

- In **test.h** there is a table of defines that must be enabled for a particular test to run. The test suite is automatically built for the default set of defines in **src/config/config_thin.h**.
- Seek tests use more RAM. The option F_MAX_SEEK_TEST in **test.h** limits the maximum size of the seek test to be performed. The options are: 128, 256, 512, 1024, 2048, 4096, 8192, 16384 (the default) and 32768.
- You must define the F_FAT_TYPE in the **test.h** file to specify whether the tests will be executed on a FAT12 or FAT16 card.

7.2 Test Summary

The tests are the following:

> **Note:** Only seek tests allowed by F_MAX_SEEK_TEST are executed.

Test	Function
0	Run all the tests
2	Directory test.
3	Find test.
5	seek 128
6	seek 256
7	seek 512
8	seek 1024
9	seek 2048
10	seek 4096
11	seek 8192
12	seek 16384
13	seek 32768
14	Open test.
15	Append test.
16	Write test.
17	Dots test.
18	rit test.