TINY File System User's Guide

Version 2.20

For use with TINY Versions 3.08 and above

Date: 24-Jun-2015 16:47

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

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

Table of Contents

System Overview 4
Introduction 4
Feature Check 5
Packages and Documents 5
Packages 5
Documents 6
Change History 6
Source File List 7
API Header File 7
Configuration File 7
TINY File System 7
Driver Files 8
Version File 8
Configuration Options 9
Summary 9
Including and Excluding API Functions 10
Other Build Options 11
Application Programming Interface 13
Module Management 13
f_init 13
File System API 14
Volume Management 14
f_initvolume 14
f_format 15
f_getfreespace 16
f_get_serial 17
f_set_serial 18
f_get_size 19
Directory Management 20
f_mkdir 20
f_chdir 21
f_rmdir 22
f_getcwd 23
File Access 24
f_open 24
f_close 26
f_read 27
f_write 29
f_getc 30
f_putc 31
f_eof 32
f_tell 33

f_seek 34
f_rewind 35
f_ftruncate 36
File Management 37
f_delete 37
f_findfirst 38
f_findnext 40
f_rename 42
f_gettimedate 43
f_settimedate 45
f_filelength 47
f_getpermission 48
f_setpermission 50
Power Management 51
f_enter_low_power 51
f_exit_low_power 52
Error Codes 53
Types and Definitions 54
F_FILE: File Handle 54
F_FIND 54
F_SPACE 54
Integration 55
OS Abstraction Layer 55
PSP Porting 56

1 System Overview

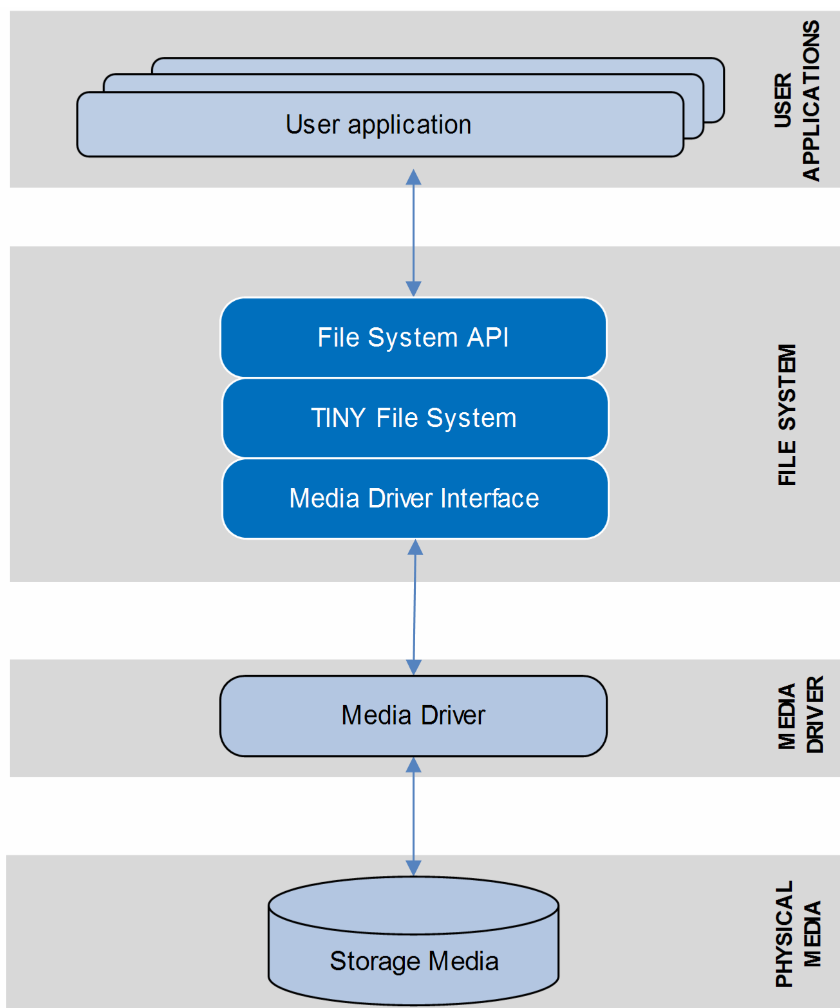
1.1 Introduction

This guide is for those who want to implement a full-featured, fail-safe flash file system for use in resource-constrained applications.

TINY is designed for use with NOR Flash with erasable sectors <4KB. This includes many serial flash devices and even the internal flash on some MCUs. Typical devices include Atmel® DataFlash AT45, MSP430 internal flash, and many serial flash devices including ST and Microchip SST Serial Flash.

Limiting the application of TINY to this subset of NOR flash devices makes TINY a compact and reliable file system. TINY eliminates many fragmentation and flash management problems and gives a compact and reliable file system that provides a full set of features, even on a low cost microcontroller.

The system structure is shown in the diagram below:



This file system is designed specifically for use with RAM. It creates pseudo-flash sectors to provide a solid logical framework for building the fail-safe system. It has been carefully crafted to ensure the reliability required by embedded systems, and to minimize code space and RAM requirements.

TINY is designed specifically for media in which the erasable sector size is relatively small, typically less than 2KB. For large NOR and NAND flash devices, the erasable sector size tends to be large (32KB or more) and, as a consequence, file systems such as HCC's SAFE must handle fragmented blocks. TINY's design assumes that the target's erasable blocks will not become fragmented and this allowed HCC to build a small footprint SAFE file system. This has given large improvements in efficiency, resulting in saving of resources (code, RAM, CPU cycles) and power.

This manual describes the TINY file API.

1.2 Feature Check

The main features of the system are the following:

- It conforms to the HCC Advanced Embedded Framework.
- It can be used with or without an RTOS.
- The code size is 8.2KB.
- The RAM usage is <256 bytes.
- It is fail safe.
- ANSI ‘C’.
- It supports long filenames.
- It supports multiple open files.
- A test suite is provided.
- It supports zero copy.
- It supports dynamic wear leveling.
- It is reentrant.
- It supports many small sector flash types.

1.3 Packages and Documents

Packages

The table below lists the packages that 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_tiny	The TINY 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.
fs_tiny_drv_ram	The Media Driver RAM package, used for creating a RAM drive.

Additional packages

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

Documents

Readers should note the points in the HCC Documentation Guidelines on the HCC documentation website.

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 TINY File System User's Guide

This is this document.

1.4 Change History

This section includes recent changes to this product. For a list of all the changes, refer to the file **src/history/tiny/tiny.txt** in the distribution package.

Version	Changes
3.08	Corrected return values for functions not returning an error code when F_LOW_POWER is enabled. (These functions could incorrectly return values returned by **f_check_low_power()**.)
3.07	Fixed problem that meant compilation failed if F_DIRECTORIES was not set. Removed warnings.
3.06	Fixed problem with **f_truncate()** when SMALL_FILE_OPT was enabled. Incorrect operation occurred when the file size shrank to a level where the cluster area was not required.

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 and your driver files.

2.1 API Header File

The file **src/api/api_tiny.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 Application Programming Interface.

2.2 Configuration File

The file **src/config/config_tiny.h** contains all the configurable parameters of the system. Configure these as required. For details of these options, see Configuration Options.

2.3 TINY File System

These files are in the directory **src/tiny/common**. **These files should only be modified by HCC.**

File	Description
f_api.c	API source code.
f_dir.c	Directory handling source code.
f_dir.h	Directory handling header file.
f_file.c	File handling source code.
f_file.h	File handling header file.
f_util.c	Utilities source code.
f_util.h	Utilities header file.
f_volume.c	Volume handling source code.
f_volume.h	Volume handling header file.
tiny.h	File system configuration definitions.
tiny_types.h	User definitions header file.

Driver Files

The file **src/tiny/driver/f_driver.h** is the flash driver header file.

The following files are for specific storage media:

File	Description
src/tiny/driver/xxx/xxx.c	Source code for target-specific driver.
src/tiny/driver/xxx/xxx.h	Header file for target-specific driver.
src/xxxDriver/flashset.h	Definition header for system.

2.4 Version File

The file **src/version/ver_tiny.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.

3 Configuration Options

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

3.1 Summary

This table summarizes the options. For more details on any option, see the sections which follow.

Option	Default	Description
F_WILDCARD	1	Enables use of wildcards.
QUICK_WILDCARD_SEARCH	0	Enables quick wildcard search.
F_CHECKNAME	1	Check for valid file name characters.
F_CHECKMEDIA	1	Check for different media at startup.
F_DIRECTORIES	1	Enables usage of directories.
F_CHDIR	1	Enables **f_chdir()**.
F_MKDIR	1	Enables **f_mkdir()**.
F_RMDIR	1	Enables **f_rmdir()**.
F_GETCWD	1	Enables **f_getcwd()**.
F_DIR_OPTIMIZE	1	Enables directory storage optimization.
F_FINDING	1	Enables **f_findfirst()** and **f_findnext()**.
F_FILELENGTH	1	Enables **f_filelength()**.
F_GETTIMEDATE	1	Enables **f_gettimedate()**.
F_SETTIMEDATE	1	Enables **f_settimedate()**.
F_GETFREESPACE	1	Enables **f_getfreespace()**.
F_DELETE	1	Enables **f_delete()**.
F_RENAME	1	Enables **f_rename()**.
F_GETPERMISSION	1	Enables **f_getpermission()**.
F_SETPERMISSION	1	Enables **f_setpermission()**.
F_SEEK_WRITE	1	Enables seeking for write.
F_TRUNCATE	1	Enables **f_truncate()**.

Copyright HCC Embedded 2015

Option	Default	Description
F_LOW_POWER	0	Enables low power support.
SMALL_FILE_OPT	1	Enables small file optimization.
QUICK_FILE_SEARCH	1	Enables quick search.
USE_ECC	0	Enables use of ECC on file management pages.
RTOS_SUPPORT	0	Enables RTOS support.
F_FILE_CHANGED_EVENT	0	Makes a file state change an event.
F_FILE_CHANGED_MAXPATH	64	Maximum path length the file system will handle if long filenames are not used.
F_MAX_OPEN_FILE	2	Maximum number of files that can be open simultaneously.
F_MAX_FILE_NAME_LENGTH	16	Maximum length of a file or directory name.
F_MAX_FILE	32	Maximum number of files in the system.
F_MAX_DIR	16	Maximum number of directories in the system.
F_ATTR_SIZE	1	Size of attribute in bytes (1/2/4)
F_COPY_BUF_SIZE	32	Size of a copy buffer.
F_ATTR_DIR	0x10	Directory attribute (this must be in the F_ATTR_SIZE range).

3.2 Including and Excluding API Functions

By defining functions to be included in, or excluded from, the file system, you can control the amount of space it uses. 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.

Every entry in the above table which has the text "Enables <function name>" in the **Description** column is a function which you can disable if you do not need it in your system. For example, setting the F_CHDIR option to 0 disables the **f_chdir()** function.

The F_FINDING option disables two functions: **f_findfirst()** and **f_findnext()**.

3.3 Other Build Options

This section gives more detail on those configuration options which do more than simply enable/disable a single function.

F_WILDCARD

This enables use of wildcards. The default is 1.

QUICK_WILDCARD_SEARCH

This enables quick wildcard search; this is useful if F_MAX_FILE is large. The default is zero.

F_CHECKNAME

This enables checking for valid name characters. If this is enabled (the default), the system:

- Accepts multiple / or \ characters.
- Accepts multiple * characters in wildcards.
- Handles a / at the end of a dirname if **f_mkdir()** is called (for example, a/b/).
- Handles a / at the end of a filename **if f_open()** is called.
- Handles upper and lower case characters.

F_CHECKMEDIA

This enables checking for different media at startup (TINY with different drive geometry). The default is 1.

F_DIRECTORIES

This enables use of directories, making the directory API functions available. The default is 1.

F_DIR_OPTIMIZE

This enables directory storage optimization. The default is 1.

F_SEEK_WRITE

This enables seeking for write. The default is 1.

F_LOW_POWER

This enables low power support. The default is zero.

SMALL_FILE_OPT

This enables small file optimization. The default is 1.

QUICK_FILE_SEARCH

This enables quick search; this is useful if F_MAX_FILE is large. The default is 1.

USE_ECC

This enables use of ECC on file management pages. The default is zero.

RTOS_SUPPORT

This enables RTOS support. The default is zero.

F_FILE_CHANGED_EVENT

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

F_FILE_CHANGED_MAXPATH

The maximum path length the file system will handle if long filenames are not used. This is only used if F_FILE_CHANGED_EVENT is enabled.

F_MAX_OPEN_FILE

The maximum number of files that can be open simultaneously. The default is 2.

F_MAX_FILE_NAME_LENGTH

The maximum length of a file or directory name. The default is 16.

F_MAX_FILE

The maximum number of files in the system. The default is 32.

F_MAX_DIR

The maximum number of directories in the system. The default is 16.

F_ATTR_SIZE

The size of the directory attribute in bytes (1/2/4). The default is 1.

F_COPY_BUF_SIZE

The size of a copy buffer. The default is 32.

F_ATTR_DIR

The directory attribute (this must be in the F_ATTR_SIZE range). The default is 0x10.

4 Application Programming Interface

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

4.1 Module Management

f_init

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

Format

```
unsigned char f_init ( void )
```

Arguments

Argument
None.

Return values

Return value	Description
F_NOERR	Successful execution.
F_ERR_OS	OS error.
Else	See Error Codes.

Example:

```
void main()
{
    f_init();  /* Initialize file system */
        .
        .
        .
}
```

4.2 File System API

Volume Management

f_initvolume

Use this function to initialize the volume.

Call this every time the file system is started.

Format

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

Arguments

Argument
None.

Return values

Argument	Description
F_NOERR	Successful execution.
Else	See Error Codes.

Example

```
void myinitfs( void )
{
    unsigned char ret;
    /* Initialize the drive */
    ret = f_initvolume();
    if (ret)
        printf( "Drive init error %d\n", ret );
      ...
}
```

f_format

Use this function to format the specified drive.

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

Format

```
unsigned char f_format ( void )
```

Arguments

Argument
None.

Return values

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

Example

```
void myinitfs()
{
    unsigned char ret;
    f_initvolume();
    ret = f_format();
    if (ret)
    {
        printf( "Unable to format drive! Error %d", ret );
    }
    else
    {
        printf( "Drive formatted correctly" );
    }
    .
    .
}
```

f_getfreespace

Use this function to fill a structure with information about the drive space usage: total space and free space.

Format

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

Arguments

Argument	Description	Type
sp	On return, a pointer to an F_SPACE structure.	F_SPACE *

Return values

Return value	Description
F_NOERR	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.",\
        space.total, space.free );
    }
    else
    {
        printf( "\nError %d reading drive!\n", ret );
    }
}
```

f_get_serial

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

Format

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

Arguments

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

Return values

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

f_set_serial

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

Format

```
unsigned char f_set_serial ( unsigned long serial)
```

Arguments

Argument	Description	Type
serial	The serial number.	unsigned long

Return values

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

f_get_size

Use this function to get the total size of the flash the file system can use.

Format

```
unsigned char f_get_size ( unsigned long * size)
```

Arguments

Argument	Description	Type
serial	Where to write the size.	unsigned long *

Return values

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

Directory Management

Note: The following functions are only available if the F_DIRECTORIES configuration option is enabled.

f_mkdir

Use this function to create a new directory.

Format

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

Arguments

Argument	Description	Type
path	Name of directory to create.	const char *

Return values

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

Example

```
void myfunc( void )
{
        .
        .
    f_mkdir( "subfolder" );   /* Creating directories */
    f_mkdir( "subfolder/sub1" );
    f_mkdir( "subfolder/sub2" );
    f_mkdir( "/subfolder/sub3" );
        .
        .
}
```

f_chdir

Use this function to change the current working directory.

Format

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

Arguments

Argument	Description	Type
path	Name of target directory.	const char *

Return values

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

Example

```
void myfunc( void )
{
        .
        .
    f_mkdir( "subfolder" );
    f_chdir( "subfolder" );    /* Change directory */
    f_mkdir( "sub2" );
    f_chdir( ".." );           /* Go upward */
    f_chdir( "subfolder/sub2" );  /* Go into directory sub2 */
        .
        .
}
```

f_rmdir

Use this function to remove a directory.

The function returns an error code if:

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

Format

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

Arguments

Argument	Description	Type
path	Name of directory to remove.	const char *

Return values

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

Example

```
void myfunc()
{
        .
    f_mkdir( "subfolder" );   /* Create directories */
    f_mkdir( "subfolder/sub1" );
        .
        .  /* Do some work */
        .
    f_rmdir( "subfolder/sub1" ); /* Remove directories */
    f_rmdir( "subfolder" );
        .
        .
}
```

f_getcwd

Use this function to get the current working directory.

Format

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

Arguments

Argument	Description	
path	Where to store the current working directory string.	char *
maxlen	The length of the buffer.	int

Return values

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

Example

```
#define BUFFLEN 256
void myfunc()
{
    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 modes are allowed for opening:

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	The file to be opened.	const char *
mode	The opening mode.	const char *

Return values

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

Example

```
void myfunc()
{
    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 one byte */
    printf( "'%c' is read from file", c );
    f_close( file );
}
```

f_close

Use this function to close a previously opened file.

Format

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

Arguments

Argument	Description	Type
filehandle	The file handle.	F_FILE *

Return values

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

Example

```
void myfunc()
{
    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_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 *      bbuf,
    long        size,
    long        size_st,
    F_FILE *    filehandle )
```

Arguments

Argument	Description	Type
bbuf	A pointer to the buffer to store data in.	void *
size	The size of the items to read.	long
size_st	The number of items to read.	long
filehandle	The handle of the file.	F_FILE *

Return values

Return value	Description
number	The number of items read successfully.
-1	Error.

Example

```
int myreadfunc( char *filename, char *buffer, long buffsize )
{
    F_FILE *file = f_open( filename, "r" );
    long size = f_filelength( filename );
    if (!file)
    {
        printf( "%s cannot be opened!", filename );
        return 1;
    }
    if (f_read( buffer, 1, size, file) != size )
    {
        printf( "Fewer bytes read than requested!" );
    }
    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 *     bbuf,
    long       size,
    long       size_st,
    F_FILE *   filehandle)
```

Arguments

Argument	Description	Type
bbuf	The buffer which contains the data.	void *
size	The size of the items to be written.	long
size_t	The number of items to write.	long
filehandle	The handle of the file.	F_FILE *

Return values

Return value	Description
number	The number of items written successfully.
-1	Operation failed.

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 bytes */
    {
        printf( "Not all bytes were 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	The handle of the open target file.	F_FILE *

Return values

Return value	Description
-1	Operation failed.
value	The character read from the 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;
}
```

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 (
    int         ch,
    F_FILE *    filehandle )
```

Arguments

Argument	Description	Type
ch	The character to write.	int
filehandle	The file handle.	F_FILE *

Return values

Return value	Description
-1	Operation failed.
value	The 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;
}
```

f_eof

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

Format

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

Arguments

Argument	Description	Type
filehandle	The file handle.	F_FILE *

Return values

Return value	Description
Zero	Not at the 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_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	The file handle.	F_FILE *

Return values

Return value	Description
filepos	The current read or write file position.
Else	See Error Codes.

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 one byte */
        printf( "Current position %d", f_tell( file ) );
    f_read( buffer, 1, 1, file );  /* Read one byte */
        printf( "Current position %d", f_tell( file ) );
    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,
    long        whence )
```

Arguments

Argument	Description	Type
filehandle	The file handle.	F_FILE *
offset	The 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.	long

Return values

Return value	Description
F_NOERR	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 one byte */
    f_seek( file, 0, SEEK_SET );
    f_read( buffer, 1, 1, file ); /* Read the same byte */
    f_seek( file,-1, SEEK_END );
    f_read( buffer, 1, 1, file ); /* Read the last byte */
    f_close( file );
    return 0;
}
```

f_rewind

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

Format

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

Arguments

Argument	Description	Type
filehandle	The file handle.	F_FILE *

Return values

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

Example

```
 void myfunc()
{
    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_ftruncate

Use this function to truncate a file which is open for writing to a specified length.

A file can only be truncated to a size less than or equal to its current size.

Format

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

Arguments

Argument	Description	Type
filehandle	The file handle.	F_FILE *
length	The new length of the file.	unsigned long

Return values

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

Example

```
int mytruncatefunc( F_FILE *file, unsigned long length )
{
    int ret = f_ftruncate( filename, length );
    if (ret)
    {
        printf( "Error:%d\n", ret );
    }
    else
    {
        printf( "File is truncated to %d bytes", length );
    }
    return ret;
}
```

File Management

f_delete

Use this function to delete a file.

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

Format

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

Arguments

Argument	Description	Type
filename	A null-terminated string with the name of the file to delete, with or without its path.	char *

Return values

Return value	Description
F_NOERR	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:

- The first entry found will be ".", the current directory.
- The second entry is "..", the parent directory.

Format

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

Arguments

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

Return values

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

Example

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

f_findnext

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

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

> **Note:** If this 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	The Find information (from **f_findfirst()**).	F_FIND *

Return values

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

Example

```
void mydir()
{
    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

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

Arguments

Argument	Description	Type
filename	The file or directory name, with or without its path.	char *
newname	The new name of the file or directory (without the path).	char *

Return values

Return value	Description
F_NOERR	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_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 & 0x001F)
Month	1-12	((d & 0x01E0) >> 5)
Years since 1980	0-119	((d & 0xFE00) >> 9)

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

Argument	Valid values	Format
Two second increments	0-30	(t & 0x001F)
Minute	0-59	((t & 0x07E0) >> 5)
Hour	0-23	((t & 0xF800) >> 11)

Format

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

Arguments

Argument	Description	Type
filename	The name of the file or directory.	char *
pctime	Where to store the creation time.	unsigned short *
pcdate	Where to store the creation date.	unsigned short *

Copyright HCC Embedded 2015

Return values

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

Example

```
void myfunc( void )
{
    unsigned short t, d;
    if (!f_gettimedate( "subfolder", &t, &d ))
    {
        unsigned short sec = (t & 0x001F) << 1;
        unsigned short minute = ((t & 0x07E0) >> 5);
        unsigned short hour = ((t & 0xF800) >> 11);
        unsigned short day = (d & 0x001F);
        unsigned short month = ((d & 0x01E0) >> 5);
        unsigned short year = 1980 + ((d & 0xF800) >> 9)
        printf( "Time: %d:%d:%d", hour, minute, sec );
        printf( "Date: %d.%d.%d", year, month, day );
    }
    else
    {
        printf( "File time cannot be retrieved!" );
    }
}
```

f_settimedate

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

Date and Time Formats

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

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

Argument	Valid values	Format
Day	0-31	(d & 0x001F)
Month	1-12	((d & 0x01E0) >> 5)
Years since 1980	0-119	((d & 0xFE00) >> 9)

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

Argument	Valid values	Format
Two second increments	0-30	(t & 0x001F)
Minute	0-59	((t & 0x07E0) >> 5)
Hour	0-23	((t & 0xF800) >> 11)

Format

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

Arguments

Argument	Description	Type
filename	The file or directory.	char *
pctime	The creation time of the file or directory.	unsigned short
pcdate	The creation date of the file or directory.	unsigned short

Return values

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

Example

```
void myfunc( void )
{
    unsigned short ctime, cdate;
    ctime = (15 << 11) + (30 << 5) + (22 >> 1);      /* 15:30:22 */
    cdate = ((2002 - 1980) << 9) + (11 << 5) + (3);  /* 2002.11.03. */
    f_mkdir( "subfolder" );  /* Create directory */
    f_settimedate( "subfolder", ctime, cdate );
}
```

f_filelength

Use this function to get the length of a file. If the file does not exist this function returns zero.

Format

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

Arguments

Argument	Description	Type
filename	The name of the file, with or without the path.	char *

Return Values

Return value	Description
filelength	The number of bytes in the file.
F_ERR_INVALID	The file does not exist or the volume is not working.

Example

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

f_getpermission

Use this function to retrieve the file or directory permission field associated with a file.

Every file and directory in the file system has an associated 32 bit field, known as the permission setting. Except for the top six bits, you can program this field as required. You could, for example, use it to create a user access system. The first six bits are reserved for use by the system, as follows:

```
#define FSSEC_ATTR_ARC (0x20UL << (31-6))
#define FSSEC_ATTR_DIR (0x10UL << (31-6))
#define FSSEC_ATTR_VOLUME (0x08UL << (31-6))
#define FSSEC_ATTR_SYSTEM (0x04UL << (31-6))
#define FSSEC_ATTR_HIDDEN (0x02UL << (31-6))
#define FSSEC_ATTR_READONLY (0x01UL << (31-6))
```

Format

```
int f_getpermission (
    const char *     filename,
    F_ATTR_TYPE *    attr)
```

Arguments

Argument	Description	Type
filename	The name of the file.	char *
attr	Where to store the attribute.	F_ATTR_TYPE *

Return values

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

Example

```
void myfunc( void )
{
    unsigned long secure;
    if (!f_getpermission( "subfolder", &secure ))
    {
        printf( "Permission is: %d", secure );
    }
    else
    {
        printf( "Permission cannot be retrieved!" );
    }
}
```

f_setpermission

Use this function to set the file or directory permission field associated with a file.

Every file/directory in the file system has an associated 32 bit field, known as the permission setting. Except for the top six bits, this field is freely programmable by the user and could, for instance, be used to create a user access system. The first six bits are reserved for use by the system, as follows:

```
#define FSSEC_ATTR_ARC (0x20UL << (31-6))
#define FSSEC_ATTR_DIR (0x10UL << (31-6))
#define FSSEC_ATTR_VOLUME (0x08UL << (31-6))
#define FSSEC_ATTR_SYSTEM (0x04UL << (31-6))
#define FSSEC_ATTR_HIDDEN (0x02UL << (31-6))
#define FSSEC_ATTR_READONLY (0x01UL << (31-6))
```

Format

```
int f_setpermission (
    const char *   filename,
    F_ATTR_TYPE    attr)
```

Arguments

Argument	Description	Type
filename	The name of the file.	char *
attr	A 32 bit number to associate with the filename.	F_ATTR_TYPE

Return values

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

Example

```
void myfunc( void )
{
    f_mkdir( "subfolder" );      /* Create directory */
    f_setpermission( "subfolder", 0x00FF0000 );
}
```

Power Management

Some flash devices can be put into ultra low power mode; these functions only have an effect if the flash driver supports this feature. Typically any internal RAM buffer is lost while in low power mode.

The power management functions allow the developer to put the flash chip into an ultra low power mode (and take it out of that mode) when it is known the system is going to be idle for a short while. Do not over-use these functions because to restore the previous RAM buffer state typically requires a page to be read.

> **Note:** These functions are only available if the F_LOW_POWER option is enabled.

f_enter_low_power

Use this function to enter low power mode.

Format

```
unsigned char f_enter_low_power ( void )
```

Arguments

Argument
None.

Return values

Return value	Description
F_NOERR	Successful execution.
F_ERR_ACCESSDENIED	The media does not support low power mode.
Else	See Error Codes.

f_exit_low_power

Use this function to leave low power mode.

Format

```
unsigned char f_exit_low_power ( void )
```

Arguments

Argument
None.

Return values

Return value	Description
F_NOERR	Successful execution.
F_ERR_ACCESSDENIED	The media does not support low power mode.
Else	See Error Codes.

4.3 Error Codes

The table below lists the error codes generated by the API functions.

Error Code	Value	Meaning
F_NOERR	0	Function was successful.
F_ERR_INVALIDVOLUME	1	No volume found.
F_ERR_INVALIDHANDLE	2	Invalid file handle.
F_ERR_INVALIDOFFSET	3	Invalid offset in file.
F_ERR_INVALIDMODE	4	Invalid open mode.
F_ERR_EOF	5	End of file.
F_ERR_NOTFOUND	6	File not found.
F_ERR_DIRFULL	7	Directory is full.
F_ERR_INVALIDNAME	8	Invalid name.
F_ERR_INVALIDDIR	9	Invalid directory.
F_ERR_OPEN	10	File is already open.
F_ERR_NOTOPEN	11	File not opened, or opened in different mode.
F_ERR_NOTFORMATTED	12	Volume not formatted.
F_ERR_DIFFMEDIA	13	Invalid volume type.
F_ERR_NOMOREENTRY	14	No more entries available.
F_ERR_DUPLICATED	15	Duplicated file name.
F_ERR_NOTEMPTY	16	Trying to remove a non-empty directory.
F_ERR_INVALIDSIZE	17	Buffer size too small (**f_getcwd()**).
F_ERR_ACCESSDENIED	18	Access is denied.
F_ERR_BUSY	19	System busy, mutex get failure.
F_ERR_CORRUPTED	20	Corrupted file.
F_ERR_LOW_POWER	21	File system in low power mode.
F_ERR_OS	22	OS error.
F_ERROR	23	General error.

4.4 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 it is closed.

F_FIND

The *F_FIND* structure takes this form:

Element	Type	Description
attr	F_ATTR	File attribute.
ctime	Unsigned short.	Creation time.
cdate	Unsigned short.	Creation date.
filesize	F_LENGTH_TYPE	Length of file.
....	....	....

The remainder of the structure is system-specific.

F_SPACE

The *F_SPACE* structure takes this form:

Element	Type	Description
total	F_LENGTH_TYPE	The total size of the volume in bytes.
free	F_LENGTH_TYPE	The number of free bytes on the volume.

Note: F_LENGTH_TYPE depends on the size of the flash.

5 Integration

TINY 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.

5.1 OS Abstraction Layer

All HCC modules use the OAL that allows the module 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 (only if RTOS_SUPPORT is enabled).
Events	0

5.2 PSP Porting

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

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

Function	Package	Element	Description
psp_getcurrenttimedate()	psp_base	psp_rtc	Returns the current time and date. This is used for date and time-stamping files.
psp_memcpy()	psp_base	psp_string	Copies a block of memory. The result is a binary copy of the data.
psp_memmove()	psp_base	psp_string	Moves a block of memory from one location to another. The two areas of memory may overlap without this causing problems as a temporary intermediate array is used.
psp_memset()	psp_base	psp_string	Sets the specified area of memory to the defined value.
psp_strncat()	psp_base	psp_string	Appends a string.
psp_strncmp()	psp_base	psp_string	Compares two strings of defined length.
psp_strncpy()	psp_base	psp_string	Copies one string of defined length to another.
psp_strnlen()	psp_base	psp_string	Gets the length of a string.

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