

SafeFLASH File System User's Guide

Version 3.10

For use with SAFE File System Versions 4.11 and above

Date: 20-Aug-2014 12:21

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	5
Introduction	5
Feature Check	6
Packages and Documents	7
Packages	7
Documents	8
Source File List	9
API Interface	9
Configuration File	9
Version File	9
SafeFLASH System	10
Test Files	10
Configuration Options	11
System Features	14
Other Media Types	14
Power Fail Safety	14
Multiple Open Files in a Volume	14
Wildcards	14
Static Wear Leveling	15
Getting Started	17
API	18
Module Management	18
f_init	18
File System API	19
General Management	20
f_enterFS	20
f_releaseFS	21
f_getlasterror	22
f_getversion	23
fs_staticwear	24
Volume Management	25
f_mountdrive	25
f_unmountdrive	28
f_chdrive	29
f_getdrive	30
f_checkvolume	31
f_format	32
f_get_drive_count	33
f_get_drive_list	34
f_getlabel	35
f_setlabel	36
f_get_oem	37

f_getfreespace	38
Directory Management	40
f_mkdir	40
f_chdir	41
f_rmdir	42
f_getcwd	43
f_getdcwd	44
File Access	45
f_open	45
f_close	47
f_flush	48
f_read	49
f_write	51
f_getc	53
f_putc	54
f_eof	55
f_seteof	56
f_tell	57
f_seek	58
f_rewind	60
f_truncate	61
f_ftruncate	62
File Management	63
f_delete	63
f_findfirst	64
f_findnext	66
f_move	68
f_rename	69
f_getpermission	70
f_setpermission	72
f_gettimedate	73
f_settimedate	75
f_fstat	77
f_stat	79
f_filelength	80
File System Unicode API	82
Unicode Directory Management	83
f_wmkdir	83
f_wchdir	84
f_wrmdir	85
f_wgetcwd	86
f_wgetdcwd	87
Unicode File Access	88
f_wopen	88
f_wtruncate	90
Unicode File Management	91

f_wdelete	91
f_wmove	92
f_wfilelength	93
f_wfindfirst	94
f_wfindnext	96
f_wrename	98
f_wgetpermission	99
f_wsetpermission	101
f_wgettimedate	102
f_wsettimedate	104
Error Codes	106
Types and Definitions	108
W_CHAR: Character and Wide Character Definition	108
F_FILE: File Handle	108
F_FIND Structure	108
F_WFIND Structure	109
F_STAT Structure	109
F_SPACE Structure	110
Testing the System	111
File System Test	111
Flash Driver Test	111
Integration	112
Requirements	112
Stack Requirements	112
Timeouts	112
Memory Allocation	112
OS Abstraction Layer (OAL)	113
PSP Porting	114

1 System Overview

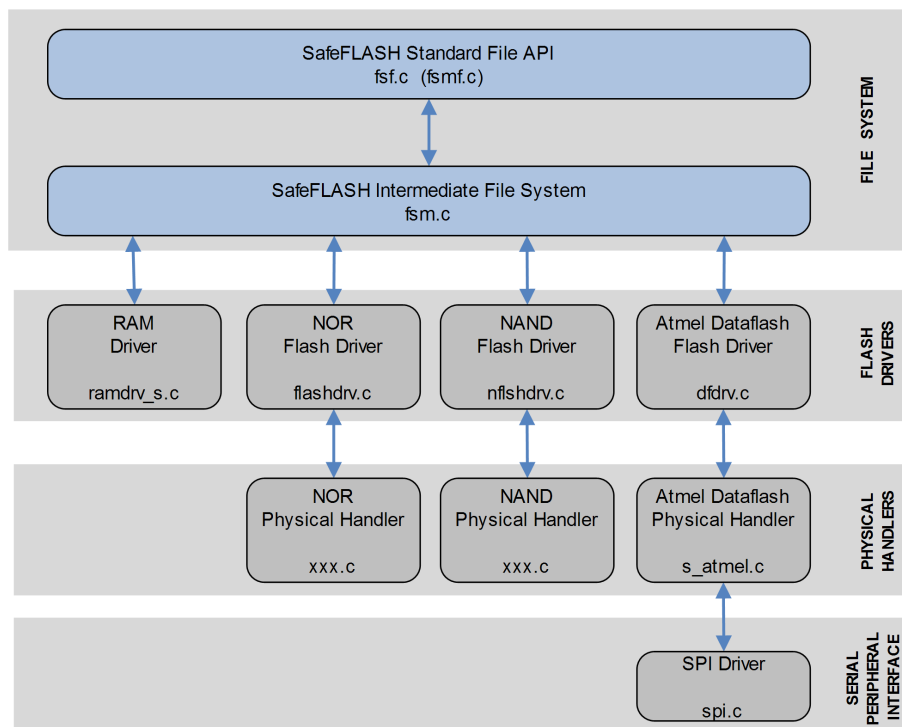
1.1 Introduction

This guide is for those who want to implement a file system in any combination of RAM, NOR flash, NAND flash, and Atmel® DataFlash.

The SafeFLASH file system driver is highly portable without compromising excellent performance. In short, SafeFLASH:

- is a package of source code designed for flash file system development in embedded systems.
- is a high performance truly fail-safe file system that can be used with all NOR and NAND flash, and any media that can simulate a block-structured array.
- supports dynamic and static wear leveling and provides a highly efficient solution for products in which data integrity is critical.

The following diagram illustrates the structure of the SafeFLASH file system:



This diagram shows:

- The Standard API and intermediate layer. The file **fsmf.c** is the Standard API multi-thread wrapper.
- The drivers – the basic device architecture includes a high level driver for each general media type. These drivers share some common properties. The driver handles issues of FAT maintenance, wear leveling, and so on.

- A physical device handler below the driver (except for the RAM driver) performs the translation between the driver and the physical flash hardware. Separate manuals detail the implementation of physical handlers for NOR flash, NAND flash, and Atmel® DataFlash.

Generally only the physical handler needs to be modified when the hardware configuration changes (for example, a different chip type, the number of devices in parallel, and so on). HCC Embedded provides a range of physical handlers to make the porting process as simple as possible.

Note:

- HCC Embedded offers hardware and firmware development consultancy to assist developers with the implementation of flash file systems.
- The SafeFLASH file system was previously known as EFFS-STD. All references to STD in the code are historical and refer to the file system's original name.

1.2 Feature Check

The main features of the system are the following:

- Code size of just 17-20KB.
- RAM usage depends on the configuration and flash type. HCC provides a tool for calculating this number.
- Fail safety.
- ANSI 'C'.
- Long filename support.
- Unicode 16 name support.
- Multiple open files.
- Multiple open file users.
- Multiple volumes.
- Media error handling.
- CRC on files (optional).
- Test suite.
- High relative performance.
- Cache option.
- Zero copy.
- Static wear leveling.
- Dynamic wear leveling.
- Re-entrant.
- Common API (CAPI) support.
- Secure delete option (NOR flash only).

NOR Flash Support

- Supports all NOR flash types.
- Easy porting for all known device types.

- Sample driver available with porting description.

Atmel® DataFlash Support

- Supports all devices.
- Manages the 10K writes/sector limitation.
- Fail-safe implementation of the DataFlash interface.

NAND Flash Support

- Supports all NAND flash types.
- Error Correction Codes (ECC) algorithm.
- Easy porting for all known device types.
- Sample driver with porting description.
- MCU/NAND controller support.

Note: SafeFLASH does not support removable media and is not recommended for arrays of flash greater than 4GB. For removable media and very large arrays, we recommend using the HCC FAT or SafeFAT system, with HCC SafeFTL where NAND flash is required.

1.3 Packages and Documents

Packages

The following table 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
<code>hcc_base_doc</code>	This contains the two guides that will help you get started.
<code>fs_safe</code>	SafeFLASH base package.
<code>media_drv_base</code>	The Media Driver base package that provides the base for all media drivers that attach to the file system.
<code>media_drv_ram</code>	The RAM Media Driver package, used for creating a RAM drive.
<code>fs_safe_ram</code>	SafeFLASH package for RAM.
<code>fs_safe_nor</code>	SafeFLASH package for NOR flash.
<code>fs_safe_nand</code>	SafeFLASH package for NAND flash.
<code>fs_safe_df</code>	SafeFLASH package for Atmel® DataFlash.

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 SafeFLASH File System User's Guide

This is this document.

Other HCC SafeFLASH Guides

These describe how to use SafeFLASH with the various drivers/physical handlers:

- *HCC SafeFLASH (RAM) User's Guide* – documents the SafeFLASH RAM driver.
- *HCC SafeFLASH (NAND) User's Guide* – documents the SafeFLASH NAND setup.
- *HCC SafeFLASH (NOR) User's Guide* – documents the SafeFLASH NOR setup.
- *HCC SafeFLASH (Atmel® DataFlash) User's Guide* – documents the SafeFLASH Atmel® DataFlash setup.

2 Source File List

This section lists all the files included in the file system. These files follow HCC Embedded's 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 Interface

The following files must be included by any application using the system. They include all that is required to access the system. For details of the API functions, see [API](#).

File	Description
<code>src/api/fsf.c</code>	API for the module.
<code>src/api/api_fs_err.h</code>	Error code definitions.

2.2 Configuration File

The file `src/config/config_safe.h` contains all the configurable parameters of the system. Configure these as required. For details of these options, see [Configuration Options](#).

2.3 Version File

The file `src/version/ver_safe.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 SafeFLASH System

These files should only be modified by HCC.

File	Description
<code>src/safe-flash/common/fsf.c</code>	SafeFLASH Standard API code.
<code>src/safe-flash/common/fsf.h</code>	SafeFLASH Standard API header.
<code>src/safe-flash/common/fsm.c</code>	SafeFLASH intermediate layer code.
<code>src/safe-flash/common/fsm.h</code>	SafeFLASH intermediate layer header.
<code>src/safe-flash/common/fsmf.c</code>	SafeFLASH Standard API multi-thread wrapper code.
<code>src/safe-flash/common/fsmf.h</code>	SafeFLASH Standard API multi-thread wrapper header.
<code>src/safe-flash/common/fstaticw.c</code>	Static wear leveling code.
<code>src/safe-flash/common/fstaticw.h</code>	Static wear leveling header.
<code>src/safe-flash/common/port_s.h</code>	Header file for port functions.

2.5 Test Files

Use the following files to exercise your file system.

File	Description
<code>src/safe-flash/test/test_s.c</code>	Source of test program used to exercise the file system.
<code>src/safe-flash/test/test_s.h</code>	Header file for test program.
<code>src/safe-flash/test/testdrv_s.c</code>	Source of test program used to exercise a flash driver.
<code>src/safe-flash/test/testdrv_s.h</code>	Header file for flash driver test program.
<code>src/safe-flash/test/testport_ram_s.c</code>	Sample port file for running test applications.

3 Configuration Options

Set the SafeFLASH configuration options in the file `src/config/config_safe.h`. This section lists the available configuration options and their default values.

FS_MAXDENAME

This is the maximum size of a name in a base directory entry. SafeFLASH supports filenames of almost unlimited length. A filename is built from a chain of small fragments taken from the descriptor block, consisting of one FS_MAXDENAME entry that may have FS_MAXLFN size entries chained to it. The maximum file name length is limited by the FS+MAXLNAME definition:

- FS_MAXLFN – if a filename is longer than FS_MAXDENAME (the default is 13), an additional FS_MAXLFN (the default is 11) byte block is allocated to store the longer name. These additional blocks are added by the file system automatically.
- FS_MAXLNAME – sets the maximum allowed name length. By default this is set to FS_MAXDENAME+4*FS_MAXLFN (57 bytes). You may increase/decrease this by multiples of FS_MAXLFN bytes; just change the FS_MAXLFN multiplier in the FS_MAXLNAME definition. This sets the number of these structures that may be used for a single name.

Long filenames use memory from the descriptor blocks in the file system. The system uses an efficient algorithm for allocating additional blocks in units of FS_MAXLFN. The use of long filenames reduces the number of file and directory entries that can be stored.

FS_CAPI_USED

If you are using FAT in the same system as SafeFLASH, you can use the Common API (CAPI) to provide a common API for accessing both systems. To do this, set FS_CAPI_USED to 1. If you are using SafeFLASH on its own, do not change this setting from the default zero.

FS_SAFE_CASE_SENSITIVE

By default SafeFLASH uses case insensitive names. To enable case sensitive names, set this to 1.

FS_MAXVOLUME

The maximum number of volumes. The default is 2. Set this value to the maximum volume number used. If only a RAM drive is used, set the value to 1; if you use a RAM drive and NOR flash, set it to 2, and so on. Volume letters are assigned by passing a parameter in the `f_mountdrive()` function.

SafeFLASH supports multiple volumes. Each volume must have its own driver routine, which normally has its own physical handler (except for the RAM drive).

FS_MAXTASK

The maximum number of tasks. The default is 1.

FS_MAXPATHNAME

The maximum length of a path. The default is 256.

FS_CURRDRIVE

This sets the current drive at startup. The default is zero. A value of -1 means there's no default current drive.

HCC_16BIT_CHAR, TI_COMPRESS

Some TI DSP devices (for example, C2000 and C5000) require special handling by the file system because of their unique architecture. For these devices, modify these two parameters as follows:

- HCC_16BIT_CHAR – enable this if the target controller has a char type that is 16 bits wide.
- TI_COMPRESS – this option allows more highly optimized storage of data in the file system. If this is enabled and the file is opened with the special mode for this, only the lower half (8 bits) is stored for all data written by the file system, and all data read out of the file system is stored in the lower 8 bits of the chars in the buffer.

To use the TI_COMPRESS option, add a "c" to the open mode after the "r", "w" or "a". For example:

```
f_open("test", rc+);  
f_open("test", wc);
```

If TI_COMPRESS is set and the "c" is not included in the open mode, the file data is handled normally.

Note: When using devices in which the pointer wraps at 64KB word boundaries, special effort is needed to allocate memory for the system in a way that this can work. Please contact support@hcc-embedded.com to discuss this further.

CRCONFILES

To handle all files with a CRC, enable this (by default it is disabled). When it is enabled, each time a file is stored the CRC is stored, and each time a file is opened its CRC is verified.

Note: Enabling this option has a major effect on system performance.

F_FILE_CHANGED_EVENT

Set this to 1 enable Change Event Notification when a file state changes. By default it is zero.

USE_TASK_SEPARATED_CWD

If this is set to 1, every task has its own current working directory. This is the default and is consistent with older versions of the system.

If it is set to zero, there is one current working directory per volume. If any task changes it, it is changed for all tasks accessing that volume.

HCC_UNICODE

To enable the use of the [Unicode 16 API functions](#), set this to 1. These functions are prefixed with "f_w", for example **f_wopen()** instead of **f_open()**.

FS_SEPARATORCHAR

This defines the file separator character. By default this is a slash ("/"). Set this to '\\' to use backslash as the pathname separator character.

FSF_MOST_FREE_ALLOC

Set this to 1, the default, to use Free Block Allocation. This allocates the block that has the most free sectors.

The alternative algorithm for allocating file system blocks just finds a block with a single available sector.

4 System Features

4.1 Other Media Types

The SafeFLASH system design is based on the concept of a storage device with a logical block arrangement. Because of this, any device that can emulate a logical block arrangement can be used as a storage medium. However, note that:

- SafeFLASH does not support removable media.
- SafeFLASH is not recommended for arrays of flash greater than 4GB.
- For removable media and very large arrays, we recommend using the HCC FAT or SafeFAT system.

4.2 Power Fail Safety

The SafeFLASH file system is entirely safe against power failure. The system may be stopped at any point, then restarted, without data being lost; the previously completed state of the file system is restored.

When a file is closed, its data are automatically flushed from the file system. Until this closure takes place, the file is preserved. You may also use the **f_flush()** function to write the current state of the file to the medium, thus updating its fail-safe state.

4.3 Multiple Open Files in a Volume

SafeFLASH allows multiple files to be opened simultaneously on a volume, or on different volumes. Within each driver (**ramdrv_s.c**, **flashdrv.c**, **nflashdrv.c** and **dfdrv.c**) there is a MAXFILE definition that determines the number of files that can be opened simultaneously on that volume at any particular time.

For each opened file, an array must be allocated that contains a sector size buffer. Therefore, increasing MAXFILE for a particular volume increases the RAM required by the system.

4.4 Wildcards

Wildcard characters can be used to find files or directories. Wildcard characters can be used only as parameters for the **f_findfirst()** function; these are then re-used when **f_findnext()** is called. The valid wildcard characters are:

Wildcard	Description
*	Matches any string.
?	Matches any single character.
"**"	Matches a string up to the end of file or the first ".", or from the first "." to the end of file. So "**.*" is required to access all files or directories in the target directory.

Note: If you want to perform a logical operation such as `f_delete(".");`:

1. Call `f_findfirst()/f_findnext()` repeatedly.
2. When each name is returned in the `F_FIND` structure, use that as a parameter for `f_delete()`.

4.5 Static Wear Leveling

Flash devices are usually manufactured to a specification that includes a guaranteed number of write-erase cycles that can be performed on each block before it may develop a fault. Because of this, it is important to use the blocks in a device "evenly" to maximize the device lifetime.

SafeFLASH uses a process called **dynamic wear leveling** to allocate the least-used blocks from those available. However, in systems where there are large areas of static data (for example, the executable binary for the system), the areas may be written only once. This leaves a relatively small section of the device to handle the much more heavily used files.

To counter this, a process called **static wear leveling** is used. When the `fs_staticwear()` function is called, it searches for blocks that have been used much less than the most used blocks in the system. If the difference between their usage rates is greater than a defined threshold (`FS_STATIC_DISTANCE`), the two blocks are exchanged.

To use static wear leveling, you must include the files `fstaticw.c` and `fstaticw.h` in your project. The header file should include the following two defines:

Define	Description
<code>FS_STATIC_DISTANCE</code>	This specifies the minimum difference between a heavily used block and a lightly used block before a static swap is allowed. Do not set this number so small that it causes unnecessary swapping. A reasonable figure is between 1% and 10% of the guaranteed erase/write cycles of the target chip.
<code>FS_STATIC_PERIOD</code>	This specifies how often this function will actually attempt to perform a swap. To reduce unnecessary checking of the system, you may use this to reduce the number of times that <code>fs_staticwear()</code> is executed. If you always know that the system will be idle when <code>fs_staticwear()</code> is called, you may set this to 1 so that it is always executed; for example, if you make just a few calls to <code>fs_staticwear()</code> at start-up. If <code>fs_staticwear()</code> is called at every available opportunity, you may want to execute it less frequently.

While the static wear leveling function executes, the file system is not accessible. The length of time it takes depends on the specification of the target chips being used, in particular the time required to erase a block and the time required to copy one block to another.

BlockCopy

For static wear leveling to function, an additional **BlockCopy** driver function must also be provided. See the appropriate driver documents (for NOR flash or NAND flash) for information on implementing this function for your target media. It is important to provide a highly optimized version of **BlockCopy**, preferably by using special copy functions that are specific to the target chip, in order to achieve the best system performance and least system disruption.

Do I need static wear-leveling?

In many cases it is an unnecessary overhead. To assess its importance, look at how your product is to be used and consider the specifications of your target devices. Many devices have up to 1 million guaranteed erase/write cycles per block and in many applications this number will not be reached in the lifetime of the product.

When should I perform static wear-leveling?

Because wear leveling involves swapping blocks in the file system, all access is excluded for the duration of the process. Thus, if there are time-critical features in your device, it is preferable to perform static wear leveling during idle moments. For effective management of the system, call the function regularly during idle time.

5 Getting Started

To start your development as efficiently as possible, take the following steps:

1. Build the file system using the API (**fsf.c**, **fsmf.c**), the intermediate file system (**fsm.c**), and the RAM driver (**ramdrv_s.c** from the **fs_safe_ram** package), including the relevant header files. In this way you can build a file system that runs in RAM with little or no dependency on your hardware platform.
2. Build a test program to exercise this file system and check how it works in RAM. All build and integration issues can thus be addressed before worrying about specific flash devices.
3. Now add the next volume to the system, depending on your requirements.

For a NOR drive:	For a DataFlash drive:	For a NAND drive:
Add flashdrv.c from the fs_safe_nor package to the build.	Add dfdrv.c from the fs_safe_df package to the build.	Add nflashdrv.c from the fs_safe_nand package to the build.

4. Now add a physical device driver to the build.

For NOR chips:	For DataFlash chips:	For NAND chips:
Read the <i>HCC SafeFLASH File System (NOR) User's Guide</i> carefully. Using the available sample drivers as a basis, create a driver that meets your specific needs.	Read the <i>HCC SafeFLASH File System (Atmel® DataFlash) User's Guide</i> carefully. Using the available sample drivers as a basis, create a driver that meets your specific needs.	Read the <i>HCC SafeFLASH File System (NAND) User's Guide</i> carefully. Using the available sample drivers as a basis, create a driver that meets your specific needs.

5. Add new volumes by repeating steps 3 and 4.

6 API

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

6.1 Module Management

f_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 configuration file **src/config/config_safe.h**.

Format

```
int f_init ( void )
```

Arguments

Argument

None.

Return values

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

Example:

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

6.2 File System API

This section describes all the Application Programmer Interface (API) functions available, apart from [Unicode functions](#). It is split into functions for general, volume, directory, and file management, also file access.

General Management

f_enterFS

Use this function to create resources for the calling task in the file system and allocate a current working directory for that task.

Note:

- If the target system allows multiple tasks to use the file system, this function must be called by a task before it uses any other file API functions.
- Correct operation of this function also requires that **oal_get_task_id()** in the [OS Abstraction Layer \(OAL\)](#) has been ported to give a unique identifier for each task.

f_releaseFS() must be called to release the task from the file system and free the allocated resource. If the system is a single task-based system, this function must also be called after **f_init()** is called.

Format

```
int f_enterFS ( void )
```

Arguments

Argument

None.

Return values

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

Example

```
void main()
{
    f_init();    /* initialize filesystem */
    f_enterFS(); /* allow current (only) task access filesystem */
    .
    .
}
```

f_releaseFS

Use this function to release a previously assigned unique task ID.

This function must be called if a given task is to be killed.

Format

```
void f_releaseFS ( void )
```

Arguments

Argument
None.

Return values

Return value
None.

Example

```
void task_destructor()  
{  
    f_releaseFS(); /* release current task ID */  
    .  
    .  
    .  
}
```

f_getlasterror

Use this function to return the last error code.

The last error code is cleared/changed when any API function is called.

Format

```
int f_getlasterror ( )
```

Arguments

Argument
None.

Return values

Return value	Description
Error code	Last error code.

Example

```
int myopen()
{
    F_FILE *file;
    file=f_open("nofile.tst","rb");
    if (!file)
    {
        int rc=f_getlasterror();
        printf ("f_open failed, errorcode:%d\n",rc);
        return rc;
    }

    return F_NO_ERROR;
}
```

f_getversion

Use this function to retrieve file system version information.

Format

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

Arguments

Argument
None.

Return values

Return value	Description
char *	Pointer to null-terminated ASCII string.

Example

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

fs_staticwear

Use this function to even the wear of blocks that are rarely used.

See [Static Wear Levelling](#) for information about when and how to use this function.

Format

```
int fs_staticwear ( int drvnum )
```

Arguments

Argument	Description	Type
drvnum	Number of target drive.	int

Return values

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

Example

```
void idle(void)
{
    int ret;
    /* try static wear on Drive A */
    ret = fs_staticwear(0);
    if(!ret)
    {
        printf("Static Wear Done\n");
    }
    Else
    {
        printf("Error in static wear!!\n",ret);
    }
}
```


Volume Management

Note: The API functions `f_chdrive()`, `f_getdrive()`, `f_get_drive_count()`, `f_get_drive_list()`, `f_mountdrive()` and `f_unmountdrive()` refer to drives by name because this is the convention, but the names are really references to volumes.

`f_mountdrive`

Use this function to mount and map a new drive. Call it with the following parameters:

`drivenum`

The number of the drive to be mounted, where 0 is drive 'A', 1 is drive 'B', and so on. The maximum value of `drivenum` is set in `FS_MAXVOLUME-1` in `fsm.h`.

`buffer`

A pointer for a buffer area to be used by the generic driver. Its size depends on the specific devices and configuration used.

- For a RAM drive allocate a buffer of the size required for the whole RAM file system, as shown in the example below.
- For a NOR drive call the generic NOR flash function `fs_getmem_flashdrive()` with a pointer to the `get-physical()` function of the specific physical chip driver to be mounted (for example, `fs_phy_nor_29lvxxx()`). This function calculates and returns the amount of memory that must be allocated for the physical driver. The caller must then allocate the memory and pass its pointer and size to `f_mountdrive()`. See the example code below.
- For a NAND drive call the generic NAND flash function `fs_getmem_nandflashdrive()` with a pointer to the `get-physical()` function of the specific physical chip driver to be mounted (for example, `fs_phy_nand_K9F2816X0C()`). This function calculates and returns the amount of memory that must be allocated for the physical driver. The caller must then allocate this amount of memory and pass its pointer and size to `f_mountdrive()`. See the example code below.

`buffsize`

The size of the allocated buffer that is passed to the mount function.

`mountfunc`

A pointer to the generic mount function for the specific media type. `mountfunc()` is a driver function that describes which drive needs to be mounted. This calls the physical driver function to be associated with it. Standard examples are:

- `fs_mount_ramdrive()` – to use a drive as a RAM drive.
- `fs_mount_flashdrive()` – to use a drive as a NOR flash drive.
- `fs_mount_nandflashdrive()` – to use a drive as a NAND flash drive.

phyfunc

A pointer to a physical driver function for the desired device that is called by the generic mount function to get information about how to use the device. For a RAM drive this function is NULL. Standard examples are:

- **fs_phy_nor_sim()** – for PC emulation of NOR physical.
- **fs_phy_nor_29lvxxx()** – for AMD flash.
- **fs_phy_nand_sim()** – for PC emulation of NAND physical.
- **fs_phy_nand_K9F2816X0C()** – for Samsung NAND flash.

Format

```
int f_mountdrive (
    int          drivenum,
    void *       buffer,
    long         buffsize,
    FS_DRVMOUNT mountfunc,
    FS_PHYGETID phyfunc )
```

Arguments

Argument	Description	Type
drivenum	Number of drive to be mounted (0='A', 1='B', and so on.).	int
buffer	Buffer pointer to be used by file system.	void *
buffsize	Size of buffer.	long
mountfunc	Mount function for selected drive type.	FS_DRVMOUNT
phyfunc	Physical driver for specific chip type.	FS_PHYGETID

Return values

Return value	Description
FS_VOL_OK	Drive successfully mounted.
FS_VOL_NOTMOUNT	Drive not mounted.
FS_VOL_NOTFORMATTED	Drive is mounted but is not formatted.
FS_VOL_NOMEMORY	Not enough memory, drive is not mounted.
FS_VOL_NOMORE	No more drives available (FS_MAXVOLUME).
FS_VOL_DRVERROR	Mount driver error, not mounted.

Example

```
/* this example shows how to mount Ramdrive, FLASH drive and NANDFLASHdrive */
char p0buffer[0x100000]; /* 1M */
void main(void)
{
    char *p1buffer, *p2buffer;
    long memsize;
    f_init();
    f_enterFS();
    f_mountdrive(0,p0buffer,sizeof(p0buffer),fs_mount_ramdrive, 0);
    /* Drive A will be RAM drive */

    memsize=fs_getmem_flashdrive(fs_phy_nor_29lvxxx);
    if (!memsize)
    {
        /* flash is not identified */
    }
    p1buffer=(char*)malloc(memsize);
    if (!p1buffer)
    {
        /* Not enough memory to allocate */
    }
    f_mountdrive(1,p1buffer,memsize,fs_mount_flashdrive,
        fs_phy_nor_29lvxxx);
    /* Drive B will be NOR flash drive, with */
    /* AMD physical driver */
    memsize=fs_getmem_nandflashdrive(fs_phy_nand_K9F2816X0C);
    if (!memsize)
    {
        /* nand flash is not identified, */
    }
    p2buffer=(char*)malloc(memsize);
    if (!p2buffer)
    {
        /* Not enough memory to allocate */
    }
    f_mountdrive(2,p2buffer,memsize,fs_mount_nandflashdrive,
        fs_phy_nand_K9F2816X0C);
    /* Drive C will be NAND flash drive with */
    /* Samsung physical */
}
*/
```

f_unmountdrive

Use this function to unmount an existing volume.

Any open files on the media are marked as closed so that subsequent API accesses to a previously opened file handle return with an error.

This function works independently of the status of the hardware.

Format

```
int f_unmountdrive ( int drivenum )
```

Arguments

Argument	Description	Type
drivenum	Drive to be deleted (0='A', 1='B', and so on).	int

Return values

Return value	Description
F_NO_ERROR	Drive successfully deleted.
Else	See Error Codes .

Example

```
void mydelfs(int num)
{
    int ret;
    /* Unmounts volume 1 */
    if(f_unmountdrive (num))
        printf("Unable to unmount volume %d", num);
    .
    .
    .
}
```

f_chdrive

Use this function to change to a new current drive.

In non-multitasking and multitasking systems, you must call **f_chdrive()** if you need relative path access. In a multitasking system, and in a non-multitasking system after **f_initvolume()**, every **f_enterFS()** must be followed by an **f_chdrive()** function call. In a multitasking system every task has its own current drive.

Format

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

Arguments

Argument	Description	Type
drivenum	Drive number to change to (0='A', 1='B', and so on).	int

Return values

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

Example:

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

f_getdrive

Use this function to get the current drive number.

Format

```
int f_getdrive ( void )
```

Arguments

Argument

None.

Return values

Return value	Description
Current Drive	0='A', 1='B', and so on
Else	See Error Codes .

Example

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

f_checkvolume

Use this function to check the status of a drive that has been initialized.

Format

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

Arguments

Argument	Description	Type
drivenum	Drive to be checked (0='A', 1='B', and so on).	int

Return values

Return value	Description
F_NO_ERROR	The drive is working.
Else	There is an error on the drive; for example, card missing. See Error Codes .

Example

```
void mychkfs(int num)
{
    int ret;
    /* checking volume */
    if(f_checkvolume(num))
    {
        printf("Volume %d is not usable, Error %d", num, ret);
    }
    else
    {
        printf(("Volume %d is working, no error", num);
    }
    .
    .
}
```

f_format

Use this function to format the specified drive.

All data on the drive are destroyed, with the exception of the wear-leveling information on a FLASH device.

Format

```
int f_format ( int drivenum )
```

Arguments

Arguments	Description	Type
drivenum	The drive to be formatted (0='A', 1='B', and so on).	int

Return values

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

Example

```
char buffer[0x30000];

void myinitfs(void)
{
    int ret;

    f_init();
    f_enterFS();

    ret=f_mountdrive(0,buffer,sizeof(buffer),fs_mount_flashdrive,
        fs_phy_nor_29lvxxx);
    /* Drive A will be NOR flash drive */
    if (ret==FS_VOL_OK) return; /* initialized */
    if (ret==FS_VOL_NOTFORMATTED)
    {
        ret=f_format(0); /* format drive A */
        if (ret==F_ERR_NOTERR) return; /* formatted */
    }
    initializationfailed:
}

```


f_get_drive_count

Use this function to get the number of drives currently available to the user.

Format

```
int f_get_drive_count ( void )
```

Arguments

Argument

None.

Return values

Return value	Description
num	Number of active volumes.

Example

```
void mygetvols(void)
{
    printf("there are %d active drives\n",
        f_get_drive_count());
    .
    .
}
```

f_get_drive_list

Use this function to get a list of drives currently available to the user.

Format

```
int f_get_drive_list ( int * buffer )
```

Arguments

Argument	Description	Type
buffer	Where to write the list.	int *

Return values

Return value	Description
number	Number of active volumes.

Example

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

    if (!i) printf ("no active drive found\n");
    for (j=0;j<i;j++)
    {
        printf("Drive %d is active\n", buffer[j]);
    }
}
```

f_getlabel

Use this function to return the label as a function value.

Format

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

Arguments

Argument	Description	Type
drivenum	Drive number (0='A', 1='B', and so on).	int
pLabel	Where to copy the label to. This should be able to hold an 11 character string.	char *
len	Length of storage area.	long

Return values

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

Example

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

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

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

f_setlabel

Use this function to set a volume label.

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

Format

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

Arguments

Argument	Description	Type
drivenum	Drive number (0='A', 1='B', and so on).	int
pLabel	Pointer to the null-terminated string to use.	char *

Return values

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

Example

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

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

f_get_oem

Use this function to return the OEM name in the disk boot record.

Format

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

Arguments

Argument	Description	Type
drivenum	Drive number (0='A', 1='B', and so on).	int
str	Pointer to location to copy label to. This should be able to hold an eight character string.	char *
len	Length of storage area.	long

Return values

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

Example

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

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

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

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

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

Arguments

Argument	Description	Type
<code>drivenum</code>	Drive number (0='A', 1='B', and so on).	int
<code>pspace</code>	Pointer to F_SPACE structure.	F_SPACE *

Return values

Return value	Description
<code>F_NO_ERROR</code>	Successful execution.
Else	See Error Codes .

Example

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

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

Directory Management

f_mkdir

Use this function to create a new directory.

Format

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

Arguments

Argument	Description	Type
dirname	New directory name 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("a:/subfolder/sub3"
    .
    .
}
```


f_chdir

Use this function to change the current working directory.

Every relative path starts from this directory. In a multitasking system every task has its own current working directory.

Format

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

Arguments

Argument	Description	Type
dirname	Null-terminated string with directory to change to.	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 up one directory level */
    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

```
int 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 on the current drive.

Format

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

Arguments

Argument	Description	Type
buffer	Where to store current working directory string.	char *
maxlen	Length of the buffer.	int

Return values

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

Example

```
#define BUFFLEN F_MAXPATH+F_MAXNAME

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

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

f_getdcwd

Use this function to get the current working directory on the selected drive.

Format

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

Arguments

Argument	Description	Type
drivenum	The drive number (0='A', 1='B', and so on).	int
buffer	Where to store the current working directory string.	char *
maxlen	The length of the buffer.	int

Return values

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

Example

```
#define BUFFLEN F_MAXPATH+F_MAXNAME

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

    if (!f_getdcwd(drivenum,buffer, BUFFLEN))
    {
        printf ("current directory is %s",buffer);
        printf ("on drive %c",drivenum+'A');
    }
    else
    {
        printf ("Drive Error")
    }
}
```

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
<code>F_FILE *</code>	Pointer to the associated opened file handle, or zero if it could not be opened.

Example

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

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

f_close

Use this function to close a previously opened file.

Format

```
int 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 opened file to a storage medium.

This is logically equivalent to performing a close and open on a file to ensure the data changed before the flush is committed to the medium.

Format

```
int f_flush ( 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

```
{
    F_FILE *file;
    char *string="ABC";

    file=fopen("myfile.bin","w");
    if (!file)
    {
        printf ("File cannot be opened!");
        return;
    }
    fwrite(string,3,1,file); /*write 3 bytes */

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


f_read

Use this function to read bytes from the current position in the target file.

The file must be opened with "r", "r+", "w+" or "a+".

Format

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

Arguments

Argument	Description	Type
buf	Buffer to store data in.	void *
size	Size of items to be read.	long
size_st	Number of items to be read.	long
filehandle	Handle of target file.	F_FILE *

Return values

Return value	Description
number	Number of items read.
Else	See Error Codes .

Example

```
{
    F_FILE *file=f_open(filename,"r");
    long size=f_filelength(filename);

    if (!file)
    {
        printf ("%s Cannot be opened!",filename);
        return 1;
    }

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

f_write

Use this function to write data into a file at the current position.

The file must be opened with "r+", "w", "w+", "a+" or "a". The file pointer is moved forward by the number of bytes successfully written.

Note: Data is NOT permanently stored to the media until either an **f_flush()** or **f_close ()** has been executed on the file.

Format

```

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

```

Arguments

Argument	Description	Type
buf	Pointer to data to be written.	void *
size	Size of items to be written.	long
size_st	Number of items to be written.	long
filehandle	Handle of target file.	F_FILE *

Return values

Return value	Description
number	Number of items written.
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;
    }

    /* write 3 bytes */
    if(f_write(string,1,3,file)!=3)
    {
        printf ("Error: not all 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.
value	Character read from the file.

Example

```
int myreadfunc(char *filename, char *buffer, long buffsize)
{
    F_FILE *file=f_open(filename,"r");
    while (buffsize-->0)
    {
        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

```
int 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 bufsize)
{
    F_FILE *file=f_open(filename,"r");

    while (!f_eof())
    {
        if (!bufsize) break;
        bufsize--;
        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

```
int 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));
    /* position 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.

The *whence* parameter is one of the following:

- F_SEEK_CUR – current position of file pointer.
- F_SEEK_END – end of file.
- F_SEEK_SET – start of file.

The offset position is relative to *whence*.

Format

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

Arguments

Argument	Description	Type
filehandle	Handle of open target file.	F_FILE *
offset	Byte position relative to <i>whence</i> .	long
whence	Where to calculate the <i>offset</i> from.	long

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

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	Handle of open 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);

        /*rewind file pointer */
        f_rewind(file);

        /*read from beginning */
        f_read(buffer2,4,1,file);

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

f_truncate

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

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

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;
}
```

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 of the open file.	F_FILE *
length	The new length of the file.	unsigned long

Return values

Return value	Description
F_NO_ERROR	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	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 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

```
int 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()** 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 is called with "*" and it is not the root directory, then:

- the first file found is ".", the current directory.
- the second file found is "..", the parent directory.

Format

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

Arguments

Argument	Description	Type
find	File 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("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_move

Use this function to move a file or directory. The original file or directory is lost.

The source and target must be in the same volume. A file can be moved only if it is not open. A directory can be moved only if there are no open files in it.

A file or directory can be moved, irrespective of its attribute settings; the attribute settings are moved with it.

Format

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

Arguments

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

Return values

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

Example

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

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	File or directory name, with or without path.	char *
newname	New name of file or directory.	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_getpermission

Use this function to retrieve 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 6 bits, this field is freely programmable by the developer 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_getpermission (
    const char *    filename,
    unsigned long * psecure )

```

Arguments

Argument	Description	Type
filename	Target file.	char *
psecure	Where to store the permission field.	unsigned long *

Return values

Return value	Description
F_NO_ERROR	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,
    unsigned long  secure )
```

Arguments

Argument	Description	Type
filename	The target file.	char *
secure	32 bit number to associate with the filename.	unsigned long

Return values

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

Example

```
void myfunc(void)
{
    f_mkdir("subfolder"); /* creating directory */
    f_setpermission ("subfolder",0x00ff0000);
}
```


f_gettimedate

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

This field is automatically set by the system when a file or directory is created, and when a file is closed.

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

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

Arguments

Argument	Description	Type
filename	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, sec, min, hour;
  unsigned short day, month, year;

  if (!f_gettimedate("subfolder",&t,&d))
  {
    sec=(t & F_CTIME_SEC_MASK);
    min=((t & F_CTIME_MIN_MASK) >> F_CTIME_MIN_SHIFT);
    hour=((t & F_CTIME_HOUR_MASK) >> F_CTIME_HOUR_SHIFT);
    day= (d & F_CDATE_DAY_MASK);
    month= ((d & F_CDATE_MONTH_MASK) >> F_CDATE_MONTH_SHIFT);
    year=1980+((d & F_CDATE_YEAR_MASK) >> F_CDATE_YEAR_SHIFT);

    printf ("Time: %d:%d:%d",hour,min,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 & 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

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

Arguments

Argument	Description	Type
filename	The file or directory.	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)
{
    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"); /* creating directory */
    f_settimedate("subfolder", ctime, cdate);
}
```

f_fstat

Use this function to get information about a file by using the file handle.

This function retrieves information by filling the [F_STAT](#) structure passed to it. It sets the file size, creation time/date, last access date, modified time/date, and the drive number where the file is located.

Format

```
int f_fstat (
    F_FILE *   p_filehandle,
    F_STAT *   p_stat )
```

Arguments

Argument	Description	Type
p_filehandle	The file handle.	F_FILE *
p_stat	Pointer to F_STAT structure to be filled.	F_STAT *

Return values

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

Example

```
void myfunc ( void )
{
    F_FILE *file;
    F_STAT stat;
    int ret;

    file = f_open( filename, "r" );

    if ( file != NULL )
    {
        ret = f_fstat(file, &stat );

        if ( ret == F_NO_ERROR )
        {
            printf ( "filesize:%d\r\n", stat.filesize );
        }
        else
        {
            printf ( "f_fstat error: %d.\r\n", ret );
        }
        f_close( file );
    }
    else
    {
        printf ( "%s Cannot be opened!\r\n",filename );
    }
}
```

f_stat

Use this function to get information about a file.

This function retrieves information by filling the [F_STAT](#) structure passed to it. It sets file size, creation time/date, last access date, modified time/date, and the drive number where the file is located.

Note: This function can also return with the opened file's current size when [f_findopenseize\(\)](#) is allowed to search through all open file descriptors for its modified size. If this feature is disabled then [f_findopenseize\(\)](#) always returns zero.

Format

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

Arguments

Argument	Description	Type
filename	Name of file.	char *
stat	Pointer to F_STAT structure to be filled.	F_STAT *

Return values

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

Example

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

f_filelength

Use this function to get the length of a file.

Note: This function can also return with the opened file's size when **f_findopenseize()** is allowed to search for it. If **f_findopenseize()** always returns with zero, this feature is disabled.

Format

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

Arguments

Argument	Description	Type
filename	File name, with or without path.	char *

Return values

Return value	Description
filelength	Length of file.
-1	The requested file does not exist or has an error; see Error Codes .

Example

```
int myreadfunc(char *filename, char *buffer, long bufsize)
{
    F_FILE *file=f_open(filename,"r");
    long size=f_filelength(filename);

    if (!file)
    {
        printf ("%s Cannot be opened!",filename);
        return 1;
    }

    if (size>bufsize)
    {
        printf ("Not enough memory!");
        return 2;
    }

    f_read(buffer,size,1,file);
    f_close(file);

    return 0;
}
```

6.3 File System Unicode API

This section describes all the API Unicode functions available with the SafeFLASH file system. It is split into functions for directory management, file access and file management.

Unicode-Specific File System Functions

To enable Unicode API calls in the SafeFLASH file system, enable `HCC_UNICODE` in the `src/config/config_safe.h` file. This makes the functions in this section, as well as their standard API equivalents, available for use.

All functions are exactly the same as their standard API counterparts, except that all character string parameters are changed to “wide character” (`wchar`) strings.

Character and wide character definition with `W_CHAR`

`W_CHAR` is defined as `char` if Unicode is disabled and as `wchar` if it is enabled. Therefore `W_CHAR` is used in structures where the element could be used in either type of system.

Unicode Directory Management

f_wmkdir

Use this function to create a new directory with a Unicode 16 name.

Format

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

Arguments

Argument	Description	Type
dirname	Name of Unicode 16 directory to create.	W_CHAR *

Return values

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

Example

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

f_wchdir

Use this function to change the current working directory (with a Unicode 16 name).

Format

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

Arguments

Argument	Description	Type
dirname	Unicode 16 directory name to change to.	W_CHAR *

Return values

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

Example

```
void myfunc(void)
{
    .
    .
    f_wmkdir("subfolder");
    f_wchdir("subfolder");      /* change directory */
    f_wmkdir("sub2");
    f_wchdir("../");           /* go to upward */
    f_wchdir("subfolder/sub2"); /* goto into sub2 dir */
    .
    .
}
```

f_wrmdir

Use this function to remove a directory with a Unicode 16 name.

The directory must be empty, otherwise an error code is returned and it is not removed.

Format

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

Arguments

Argument	Description	Type
dirname	Unicode 16 name of directory to remove.	W_CHAR *

Return values

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

Example

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

f_wgetcwd

Use this function to get the current working directory on the current drive.

Format

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

Arguments

Argument	Description	Type
buffer	Where to store the current working directory string.	W_CHAR *
maxlen	Length of the buffer.	int

Return values

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

Example

```
void myfunc(int drivenum)
{
    W_CHAR buffer[F_MAXPATH];
    if (!f_wgetdcwd(drivenum,buffer, F_MAXPATH))
    {
        wprintf ("current directory is %s",buffer);
        wprintf ("on drive %c",drivenum+'A');
    }
    else
    {
        wprintf ("Drive Error")
    }
}
```

f_wgetdcwd

Use this function to get the current working directory on the selected drive.

Format

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

Arguments

Argument	Description	Type
drivenum	Drive (0='A', 1='B', and so on).	int
buffer	Where to store the current working directory string.	W_CHAR *
maxlen	Length of the buffer.	int

Return values

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

Example

```
void myfunc(int drivenum)
{
    W_CHAR buffer[F_MAXPATH];
    if (!f_wgetdcwd(drivenum,buffer, F_MAXPATH))
    {
        wprintf ("current directory is %s",buffer);
        wprintf ("on drive %c",drivenum+'A');
    }
    else
    {
        wprintf ("Drive Error")
    }
}
```

Unicode File Access

f_wopen

Use this function to open a file with Unicode 16 filename. The following opening modes are allowed:

Modes	Description
"r"	Open an existing file for reading. The stream is positioned to the beginning of the file.
"r+"	Open an existing file for reading and writing. The stream is positioned to the beginning of the file.
"w"	Truncate file to zero length or create file for writing. The stream is positioned to the beginning of the file.
"w+"	Open for reading and writing. The file is created if it does not exist; otherwise it is truncated. The stream is positioned to the beginning of the file.
"a"	Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned to the end of the file.
"a+"	Open for reading and appending (writing at end of file). The file is created if it does not exist. The stream is positioned to 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 all files to be accessed in binary mode only.

Format

```
F_FILE * f_wopen (
    const W_CHAR * filename,
    const char * mode )
```


Arguments

Argument	Description	Type
filename	Unicode 16 name of target file.	W_CHAR *
mode	The opening mode (see above).	char *

Return values

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

Example

```
void myfunc(void)
{
    F_FILE *file;
    char c;
    file=f_wopen("myfile.bin","r");
    if (!file)
    {
        wprintf ("File cannot be opened!");
        return;
    }
    f_read(&c,1,1,file); /* read lbyte */
    wprintf ("%c' is read from file",c);
    f_close(file);
}
```

f_wtruncate

Use this function to open an existing 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_wtruncate (
    const W_CHAR * filename,
    unsigned long length )
```

Arguments

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

Return values

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

Example

```
int mywtruncatefunc(W_CHAR *filename, unsigned long length)
{
    F_FILE *file=f_wtruncate(filename,length);
    if(!file)
        printf("File not found");
    else
    {
        printf("File %s truncated to %d bytes,
            filename, length);
        f_close(file);
    }
    return 0;
}
```

Unicode File Management

f_wdelete

Use this function to delete a file with a Unicode 16 name.

Format

```
int f_wdelete ( const W_CHAR * filename )
```

Arguments

Argument	Description	Type
filename	Name of file to be deleted, with or without its path.	W_CHAR *

Return values

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

Example

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

f_wmove

Use this function to move a file or directory with a Unicode 16 name.

The source and target must be in the same volume. The original file or directory is lost.

Format

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

Arguments

Argument	Description	Type
filename	Unicode 16 file or directory name, with or without path.	W_CHAR *
newname	New Unicode 16 name of file or directory.	W_CHAR *

Return values

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

Example

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

f_wfilelength

Use this function to obtain the length of a file with a Unicode 16 name.

Format

```
long f_wfilelength ( W_CHAR * filename )
```

Arguments

Argument	Description	Type
filename	Unicode 16 file name, with or without path.	W_CHAR *

Return values

Return value	Description
filelength	Length of file. If this is zero then the file may not exist – check the last error.

Example

```
int myreadfunc(W_CHAR *filename, char *buffer, long bufsize)
{
    F_FILE *file=f_wopen(filename,"r");
    long size=f_wfilelength(filename);
    if (!file)
    {
        printf ("%s Cannot be opened!",filename);
        return 1;
    }
    if (size>bufsize)
    {
        printf ("Not enough memory!");
        return 2;
    }
    f_read(buffer,size,1,file);
    f_close(file);
    return 0;
}
```

f_wfindfirst

Use this function to find the first Unicode 16 file or subdirectory in the specified directory.

First call **f_wfindfirst()** then, if a file is found, get the next file with **f_wfindnext()**.

Format

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

Arguments

Argument	Description	Type
filename	Unicode 16 name of file to find.	W_CHAR *
find	Where to store file information.	F_WFIND *

Return values

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

Example

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

f_wfindnext

Use this function to find the next Unicode 16 file or subdirectory in a specified directory after a previous call to **f_wfindfirst()** or **f_wfindnext()** .

First call **f_wfindfirst ()** then, if a file is found, get the rest of the matching files by repeated calls to **f_wfindnext()**.

Format

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

Arguments

Argument	Description	Type
find	Find structure (from f_wfindfirst()).	F_WFIND *

Return values

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

Example

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

f_rename

Use this function to rename a file or directory with a Unicode 16 name.

Format

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

Arguments

Argument	Description	Type
filename	Unicode 16 file or directory name, with or without path.	W_CHAR *
newname	New Unicode 16 name of file or directory.	W_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:/dir/oldfile.txt", "newfile.txt");
    .
    .
}
```

f_wgetpermission

Use this function to retrieve the file or directory permission field associated with a file with a Unicode 16 name.

Every file/directory in the file system has an associated 32 bit field, known as the permission setting. Except for the top 6 bits, this field is freely programmable by the developer and can, for example, 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_getpermission (
    const W_CHAR *   filename,
    unsigned long *  psecure )
```

Arguments

Argument	Description	Type
filename	Unicode 16 name of the target file.	W_CHAR *
psecure	Where to store the permission field.	unsigned long *

Return values

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

Example

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

f_wsetpermission

Use this function to set the file or directory permission field associated with a file with Unicode 16 name.

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 developer and can, for example, 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_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_wsetpermission (
    const W_CHAR *   filename,
    unsigned long    secure )
```

Arguments

Argument	Description	Type
filename	Unicode 16 name of target file.	W_CHAR *
secure	32 bit number to associate with filename.	unsigned long

Return values

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

Example

```
void myfunc(void)
{
    f_mkdir("subfolder");    /* creating directory */
    f_wsetpermission ("subfolder", 0x00ff0000);
}
```

f_wgettimedate

Use this function to get time and date information for a file or directory with a Unicode 16 name.

This field is automatically set by the system when a file or directory is created, and when a file is closed.

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

```
int f_wgettimedate (
    const W_CHAR *   filename,
    unsigned short * pctime,
    unsigned short * pcdatetime )
```

Arguments

Argument	Description	Type
filename	Unicode 16 name of target file or directory.	W_CHAR *
pctime	Where to store the time.	unsigned short *
pcdatetime	Where to store the 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_wgettimedate("subfolder",&t,&d))
    {
        unsigned short sec=(t & 0x001f) << 1;
        unsigned short minute=((t & 0x07e0) >> 5);
        unsigned short hour=((t & 0x0f800) >> 11);
        unsigned short day= (d & 0x001f);
        unsigned short month= ((d & 0x01e0) >> 5);
        unsigned short year=1980+((d & 0xfe00) >> 9);
        wprintf ("Time: %d:%d:%d",hour,minute,sec);
        wprintf ("Date: %d.%d.%d",year,month,day);
    }
    else
    {
        wprintf ("File time cannot retrieved!");
    }
}
```

f_wsettime

Use this function to set the time and date on a file or directory with a Unicode 16 name.

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

```
int f_settime (
    const W_CHAR *   filename,
    unsigned short   ctime,
    unsigned short   cdate )
```

Arguments

Argument	Description	Type
filename	Unicode 16 name of the file.	W_CHAR *
ctime	Creation time of file or directory.	unsigned short
cdate	Creation date of file or directory.	unsigned short

Return values

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

Example

```
void myfunc(void)
{
    unsigned short ctime,cdate;
    ctime = (15<<11)+(30<<5)+(23>>1);    /* 15:30:22 */

    cdate = ((2002-1980)<<9)+(11<<5)+(3); /* 2002.11.03. */

    f_wmkdir("subfolder");    /* creating directory */
    f_wsettimedate("subfolder",ctime,cdate);
}
```

6.4 Error Codes

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

The header file to include for this list is `src/api/api_fs_err.h`.

Error	Value	Meaning
F_NO_ERROR	0	Successful execution.
F_ERR_INVALIDDRIVE	1	The specified drive does not exist.
F_ERR_NOTFORMATTED	2	The specified volume has not been formatted.
F_ERR_INVALIDDIR	3	The specified directory is invalid.
F_ERR_INVALIDNAME	4	The specified file name is invalid.
F_ERR_NOTFOUND	5	The file or directory could not be found.
F_ERR_DUPLICATED	6	The file or directory already exists.
F_ERR_NOMOREENTRY	7	The volume is full.
F_ERR_NOTOPEN	8	The file access function requires the file to be open.
F_ERR_EOF	9	End of file.
F_ERR_RESERVED	10	Not used.
F_ERR_NOTUSEABLE	11	Invalid parameters for <code>f_seek()</code> .
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 moved 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	Media not recognized.

Error	Value	Meaning
F_ERR_BUSY	22	The caller could not obtain the semaphore within the expiry time.
F_ERR_WRITEPROTECT	23	The physical medium 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_MEDIATOO LARGE	26	Media is too large for the format type requested.
F_ERR_NOTSUPPSECTORSIZE	27	The sector size is not supported. The only supported sector size is 512 bytes.
F_ERR_UNKNOWN	28	An unspecified error has occurred.
F_ERR_DRVALREADYMNT	29	The drive is already mounted.
F_ERR_TOOLONGNAME	30	The name is too long.
F_ERR_NOTFORREAD	31	Not for read.
F_ERR_DELFUNC	32	The delete drive driver function failed.
F_ERR_ALLOCATION	33	psp_malloc() failed to allocate the required memory.
F_ERR_INVALIDPOS	34	An invalid position is selected.
F_ERR_NOMORETASK	35	All task entries are exhausted.
F_ERR_NOTAVAILABLE	36	The called function is not supported by the target volume.
F_ERR_TASKNOTFOUND	37	The caller's task identifier was not registered. This is normally because f_enterFS() has not been called.
F_ERR_UNUSABLE	38	The file system has become unusable. This is normally a result of excessive error rates on the underlying media.
F_ERR_CRCERROR	39	A CRC error has been detected on the file.
F_ERR_CARDCHANGED	40	The card that was being accessed has been replaced with a different card.

6.5 Types and Definitions

W_CHAR: Character and Wide Character Definition

W_CHAR is defined to char if Unicode is disabled and to wchar if it is enabled. Therefore W_CHAR is used in structures where the element could be used in either type of system.

F_FILE: File Handle

This is 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 Structure

The structure takes this form:

Parameter	Description	Type
filename	Long file name.	char[F_MAXPATHNAME]
name	Short file name.	char[F_MAXSNAME]
ext	Short file name extension.	char[F_MAXSEXT]
attr	Attribute setting of the file.	unsigned char
ctime	Creation time.	unsigned short
cdate	Creation date.	unsigned short
filesize	Length of file.	unsigned long
cluster	For internal use only.	unsigned long
findfsname	For internal use only.	F_NAME
pos	For internal use only.	F_POS

F_WFIND Structure

The structure takes this form:

Parameter	Description	Type
filename[F_MAXPATHNAME]	File name plus extension.	W_CHAR
name[F_MAXSNAME]	File extension.	char
ext[F_MAXSEXT]	File name.	char
attr	File attribute setting.	unsigned char
ctime	Creation time.	unsigned short
cdate	Creation date.	unsigned short
filesize	Length of file.	unsigned long
cluster	Current file starting position.	unsigned long
findfsname	Find properties.	F_NAME
pos	Position of the current list.	F_POS

F_STAT Structure

The structure takes this form:

Parameter	Description	Type
filesize	Size of the file.	unsigned long
createdate	Creation date.	unsigned short
createtime	Creation time.	unsigned short
modifieddate	Last modified date.	unsigned short
modifiedtime	Last modified time.	unsigned short
lastaccessdate	Last accessed date.	unsigned short
attr	00ADVSHR	unsigned char
drivenum	Number of the volume.	int

F_SPACE Structure

The F_SPACE structure takes this form:

Parameter	Description	Type
total	Total size in bytes of the disk.	unsigned long
free	Number of free bytes on the disk.	unsigned long
used	Number of used bytes on the disk.	unsigned long
bad	Number of bad bytes on the disk.	unsigned long
total_high	High part of total if greater than 4GB.	unsigned long
free_high	High part of free if greater than 4GB.	unsigned long
used_high	High part of used if greater than 4GB.	unsigned long
bad_high	High part of bad if greater than 4GB.	unsigned long

7 Testing the System

Two test suites are provided for exercising the file system and the flash drivers, and ensuring that all are working correctly.

Both test programs require the functions defined and implemented (as samples) in **testport_ram_s.c**. Port these functions to your system. Refer to the comments and simple code for reference.

7.1 File System Test

This program exercises most of the functionality of the file system, including file read/write/append/seek/file content, directories and file manipulation functions.

To use the test program:

1. Include **test_s.c** and **test_s.h** in your test project.
2. Call the following to execute the test code:

```
void f_dotest( void )
```

7.2 Flash Driver Test

This code tests your ported flash driver in isolation, to ensure that it is ported correctly and is stable.

To use this test program:

1. Include **testdrv_s.c** and **testdrv_s.h** in your test project.
2. Call the following to execute the test code:

```
void f_dotestdrv ( FS_PHYGETID phyfunc )
```

Note: Errors in the execution of this test indicate that there is an error in the implementation of the driver. Contact support@hcc-embedded.com if you need further advice.

8 Integration

This section describes all aspects of the SafeFLASH module that require integration with your target project. This includes porting and configuration of external resources.

8.1 Requirements

The SafeFLASH system is designed to be as open and portable as possible. No assumptions are made about the functionality or behavior of the underlying operating system.

For SafeFLASH to work at its best, perform the porting work outlined below. This is a very straightforward task for an experienced engineer.

Stack Requirements

SafeFLASH file system functions are always called in the context of the calling thread or task. Naturally the functions require stack space and you should allow for this in applications that call file system functions. Typically calls to the file system use <2KB of stack.

Timeouts

Flash devices are normally controlled by hardware control signals. As a result there is no explicit need for any timeouts to control exception conditions. However, some operations on flash devices are relatively slow and it is often worthwhile to schedule other operations while waiting for them to complete. For example, a NOR flash erase typically takes two seconds and a NAND flash erase takes two milliseconds.

- For NOR flash in the **29lvxxx.c** sample driver, the **DataPoll()** function is used to check for the completion of operations. This routine can be modified to force scheduling of the system, or to use the host system's event generation mechanism so that other operations can be performed while waiting.
- For NAND flash in the K9F2816X0C sample driver, the **nandwaitrb()** function is used to check for the completion of operations. This routine can be modified to force scheduling of the system, or to use the host system's event generation mechanism so that other operations can be performed while waiting.

Memory Allocation

Some larger buffers are required by SafeFLASH to handle FATs in RAM, and also to buffer write processes.

A call is made to each driver to get the specific size of memory required for that drive. It is then up to you to allocate this memory from the system.

Buffer sizes depend on the particular chips being used and their configurations. For further information, see the descriptions of the **f_mountdrive()** and **fs_getmem_xxx()** functions in the relevant driver manuals.

8.2 OS Abstraction Layer (OAL)

All HCC modules use the OS Abstraction Layer that allows the module to run seamlessly with a wide variety of RTOSes, or without an RTOS.

This module uses the following OAL components:

OAL resource	Number required
Tasks	0
Mutexes	1 per volume plus 1 Each volume needs protecting by a mutex mechanism to ensure that file access is safe.
Events	0

Note: If the Common API (CAPI) is used (that is, `FS_CAPI_USED` is defined in the file `config_safe.h`), the above mutex functions are replaced by the equivalent functions from the CAPI. See the *HCC File System Common API User's Guide* for details.

Within the standard API there is no support for the current working directory to be maintained on a per-caller basis. By default, the system provides a single `cwd` that can be changed by any user. The `cwd` is maintained on a per-volume basis, or on a per-task basis if multitasking is implemented.

For a multitasking system, do the following:

1. Set `F_MAXTASK` to the maximum number of tasks that can simultaneously maintain access to the file system. This effectively creates a table of `cwds` for each task.
2. Use the OAL's `oal_task_get_id()` function to get a unique identifier for the calling task.
3. Ensure that any task using the file system calls `f_enterFS()` before any other API calls; this ensures that the calling task is registered.
4. Ensure that for any application that has finished using the file system, or is terminated, `f_releaseFS()` is called with the task's unique identifier. This frees that table entry for use by other applications.

Once these steps are implemented, each caller is logged as it acquires the mutex, and a current working directory is associated with it.

Note: If the CAPI is used, `oal_task_get_id()` is replaced by the equivalent function from the CAPI. See the *HCC File System Common API User's Guide* for further information.

8.3 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 SafeFLASH module makes use of the following standard PSP function:

Function	Package	Element	Description
psp_getcurrenttimedate()	psp_base	psp_rtc	Gets the current time and date.

Note: If the Common API (CAPI) is used (that is, F_CAPI_USED is defined in **config_safe.h**) then **psp_getcurrenttimedate()** is replaced by the equivalent function from the CAPI. See the *HCC File System Common API User's Guide* for further information.