File System Common API User Guide

Version 2.10

For use with File System Common API Versions 3.03 and above

Date: 12-Jun-2018 10:23

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

1 System Overview

This chapter contains the fundamental information for this module.

The component sections are as follows:

- Introduction – describes the main elements of the module.
- Feature Check – summarizes the main features of the module as bullet points.
- Building a CAPI System – shows how to build a CAPI system. Once these instructions are completed, the unified system will be ready for use.
- Drive Numbers – shows which drive numbers to use and how these are mapped.
- Packages and Documents – the *Packages* section lists the packages that you need in order to use this module. The *Documents* section lists the relevant user guides.
- Change History – lists the earlier versions of this manual, giving the software version that each manual describes.

1.1 Introduction

This guide is for those who want to implement a single file system combining the benefits of HCC Embedded's FAT, SAFEFAT, and SafeFLASH file systems.

The Common Application Programming Interface (CAPI) is designed to allow HCC's SAFE, FAT, and SafeFLASH file systems to be accessed through a single consistent API, presenting one array of drives that can be used identically by applications. This gives you all the benefits of an efficient and fail-safe file system for internal flash devices, while being able to use the same API to address devices in which the media are PC-compatible with either the SAFEFAT or FAT file systems.

The CAPI is provided to allow any combination of HCC file system volumes to be used under a single API wrapper. Drives appear as a standard array of drives with a common API. The file system on each drive may be different but the user interface is entirely consistent.

Because of their differing capabilities, there are differences between the HCC file systems in the initialization, volume, and partition control functions. However all file and directory manipulation functions are entirely standard and 100% compatible across every file system. All of the available functions are listed in this manual and functions that are specific to the CAPI are described in detail.

Note: HCC Embedded offers hardware and firmware development consultancy to assist developers in implementing various types of file system.

1.2 Feature Check

The system incorporates all the features of the FAT, SafeFAT, and SafeFLASH file systems under a common API.

1.3 Building a CAPI System

To build a CAPI system, do the following:

> **Note:** If either FAT or SAFE is not required but CAPI is, simply omit the instructions below relating to the unwanted system.

1. Install and test FAT as directed in the *HCC FAT and SafeFAT File System User Guide*. Run the test software to ensure that the module is working correctly.
2. Install and test SafeFLASH as directed in the *HCC SafeFLASH File System User Guide*. Run the test software to ensure that the module is working correctly.
3. In the configuration file **src/config/config_capi.h** define FW_FAT_USED and FW_SAFE_USED as required.
4. In the SafeFAT configuration file, set FN_CAPI_USED to 1.
5. In the SafeFLASH configuration file, set FS_CAPI_USED to 1.
6. Set the parameter FW_MAXFILE in **src/api/api_capi.h** to the number of open files allowed. This defines the maximum number of files that can be opened simultaneously through the common API.
7. Set up Unicode. If Unicode file names are required, then enable Unicode 16 in both FAT and SafeFLASH by setting HCC_UNICODE to 1 in both the SafeFAT and SafeFLASH configuration files. If Unicode is enabled in either system, the CAPI Unicode API functions are built automatically.

Your unified system is now ready for use.

1.4 Drive Numbers

You can use any drive number from 0 to 25 when mounting a drive. These map directly to A:(0) to Z:(25).

If the function **f_mountfat()** is called with drive number 2, the C: drive is available as a FAT drive. Similarly, if **f_mountsafe()** is called with drive number 4, a SAFE drive E: is available. CAPI manages the mapping between drives so all operations are transparent to the user.

1.5 Packages and Documents

Packages

The table below lists the packages that you need in order to use this module:

Package	Description
hcc_base_doc	This contains the two guides that will help you get started.
fs_capi	The Common API package.

The table below lists the file system packages that you may use with the CAPI:

Package	Description
fs_fat	The FAT file system package.
fs_fat_safe	The SafeFAT package that contains extensions to the FAT system.
fs_safe	The SafeFLASH file system base package.

Documents

For an overview of HCC file systems and guidance on choosing a file system, see Product Information on the main HCC website.

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 File System Common API User Guide

This is this document.

HCC FAT and SafeFAT File System User Guide

This document describes the FAT and SafeFAT file system.

HCC SafeFLASH File System User Guide

This document describes the SafeFLASH file system.

1.6 Change History

To view or download manuals, see File System PDFs.

For the history of changes made to the package code itself, see History: fs_capi.

The current version of this manual is 2.10.

Manual version	Date	Software version	Reason for change
2.10	2018-06-12	3.03	Simplified list in *Feature Check*.
2.00	2018-06-05	3.03	First online version.

2 Source File List

This section lists all the source code files included in the 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 files and test suite porting files.

2.1 API File

The file **src/api/api_capi.h** should be included by any application using the system. It includes all that is required to access the system.

2.2 Configuration Files

These files are in the directory **src/config/** contain all the configurable parameters of the system. Configure these as required.

File	Description
config_capi.h	System configuration options.
config_capi_test.h	Test configuration options.

2.3 Source Code Files

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

File	Description
fw_file.c	Source code.
fw_multi.h	Header file.

2.4 Version File

The file **src/version/ver_capi.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.

2.5 Test Suite

These files are in the directory **src/capi/test**.

File	Description
fw_test.c	Test source code.
fw_test.h	Header file.
fw_testport_f.c	Porting file for testing a FAT drive.
fw_testport_s.c	Porting file for testing a SAFE drive.

3 Configuration Options

Set the configuration options in the files **src/config/config_capi.h**. and **src/config/config_capi_test.h**. This section lists the available options and their default values.

3.1 config_capi.h

Set the following system configuration options in the file **src/config/config_capi.h**.

General Options

FW_FAT_USED, FW_SAFE_USED

Enable the option for each file system being used in the target system. The default for both of these is 1.

HCC_UNICODE

This option enables support for Unicode 16 formatted characters. (Unicode 7/8 formats are supported as standard.) The default is 0.

To enable this option, set it to 1. This forces any build to include the Unicode 16 API, making the Unicode 16 API calls documented in File System Unicode API available.

Use of Unicode 16:

- Implies that the host system has wchar (“wide character”) support or an equivalent definition.
- Creates additional resource requirements because all string and path accesses effectively use twice the space. Therefore, use of this option is recommended only if Unicode 16 is genuinely required.

Note: To allow the file system to generate consistent short file names from the Unicode file name, you must provide conversion tables in the code. For details, see Unicode Translation.

F_FILE_CHANGED_EVENT

This notifies any change in the file or directory structure of the file system. The default is 0.

Enabling this is useful when the system is used in conjunction with other file system interfaces such as MTP or NFS, where the other system needs notifying of any changes to the directory or file structures in the system.

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 define is enabled and the file is opened with the special mode for this, then only the lower half (8bits) will be stored for all data written by the file system, and all data read out of the file system will be 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 then the file data will be handled normally.

> **Note:** When using devices in which the pointer wraps at 64K 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 for details.

F_SUPPORT_TI64K

The default is 0. Enabling this option ensures that read and write operations do not cross over 64K boundaries. The system automatically breaks the results of these operations into units which do not cross these boundaries.

This option is provided because certain devices, in particular TI C2000 and C5000 series DSPs, do not handle pointer increments over 64K boundaries.

USE_TASK_SEPARATED_CWD

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

If this is set to 0, there is one cwd per volume so, if any task changes it, it is changed for all tasks accessing that volume.

File and Volume Definitions

FS_MAXVOLUME

The maximum number of SAFE volumes allowed on the system. The default is 1.

> **Note:** The system is designed so that access to a specific volume is entirely independent of any other volumes. That is, if an operation is being performed on one volume, this does not block access to other volumes.

FN_MAXVOLUME

The maximum number of FAT volumes allowed on the system. The default is 1.

FW_MAXTASK

The number of tasks that are allowed to access the file system simultaneously.

If this is set to 1 (the default), it implies that no OS is used, or that all accesses are controlled through a single task. If this is set to any value greater than 1, the OS Abstraction Layer (OAL) must be included in the project.

FW_CURRDRIVE

This determines which drive of the system is used at system startup. If -1 is set there is no default current drive. The default is 0.

F_MAXPATHNAME

The maximum file path. The default is 256. You can decrease this to reduce the resource requirements, in particular the stack.

FW_PATH_SEPARATOR

The default is '/'. Set this to '\\' if you want FAT to use backslash as the pathname separator character.

FW_DRIVE_SEPARATOR

The drive name separator character in full pathnames. The default is ':'.

3.2 config_capi_test.h

Set the following test configuration options in the file **src/config/config_capi_test.h**.

TEST_FAT_MEDIA

The media used for test formatting. The default is F_FAT32_MEDIA.

RAMDRIVETEST

Set this to 1 if RAM drive is tested. The default is 0.

MAX_BUFFERSIZE

The size of the buffer used for test functions. A smaller buffer size results in fewer tests. Valid values are: 1024, 2048, 4096, 8192, 16384 and 32768. The default is 16384.

RIT_NUM_OF_RECORDS

The number of RIT tests. The default is 100.

4 Application Programming Interface

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

4.1 Module Management

There is just one function:

Function	Description
f_init()	Initializes the CAPI. This call initializes the CAPI's structures and resources and then calls the initialization functions of each file system that it controls.

f_init

Use this function to initialize the CAPI. This call initializes the CAPI's structures and resources and then calls the initialization functions of each file system that it controls.

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()
{
    /* Initialize CAPI and file systems */
    if(f_init() == F_NO_ERROR)
    {
        /* system initialized successfully */
        .
        .
    }
    else
    {
        /* file system initialization failed */
        .
        .
    }
}
```

4.2 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.

General Management

The functions are the following:

Function	Description
f_enterFS()	Creates resources for the calling task in the file system and allocates a current working directory for that task.
f_releaseFS()	Releases a previously assigned unique task ID.
f_getlasterror()	Gets the last error code.
f_get_oem()	Gets the OEM name from the disk boot record.

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 management API functions.
- For the correct operation of this function, **oal_get_task_id()** in the OS Abstraction Layer must have 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 also needs to 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 the file system from the calling task.

This function releases the entry so another slot is available for tasks to be able to use the file system. You must call it if a given task is released or no longer exists.

Format

```
void f_releaseFS ( void )
```

Arguments

Argument
None.

Return values

Return value
None.

Example

```
void task_destructor()
{
    f_releaseFS(); /* Release the 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_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	The drive number.	int
str	A pointer to the location to copy the name to. This must be able to hold an eight character string.	char *
len	The length of the 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);
}
```

Volume Management

The following functions are only available in the CAPI. These are described on the following pages.

Function	Description
f_mountfat()	Mounts a drive from FAT.
f_mountfatpartition()	Mounts a drive from FAT when the medium (for example, hard disk) contains multiple FAT partitions.
f_mountsafe()	Mounts a drive from SafeFLASH.

Note: The API functions **f_getdrive()**, **f_chdrive()** and **f_getdcwd()** use the term "drive" because this is the convention. This is equivalent to the term "volume".

The following functions are the same in SafeFAT and SafeFLASH. Refer to the relevant manual for more details.

Function	Description
f_chdrive()	Changes to a new current drive.
f_getdrive()	Gets the current drive number.
f_checkvolume()	Checks the status of a drive that has been initialized.
f_format()	Formats the specified drive.
f_setlabel()	Sets a volume label.
f_getlabel()	Returns the label as a function value.
f_get_oem()	Returns the OEM name in the disk boot record.
f_getfreespace()	Fills a structure with information about the drive space usage: total space, free space, used space, and bad (damaged) size.

The following functions are available in one of the file systems. Refer to the relevant manual for more details.

FAT/SafeFAT	SafeFLASH	Description
	f_mountdrive()	Mounts and maps a new drive.
	f_unmountdrive()	Unmounts an existing volume.
	f_get_drive_count()	Gets the number of drives currently available to the user.
	f_get_drive_list()	Gets a list of drives currently available to the user.
f_initvolume()		Initializes a volume.
f_initvolume_nonsafe()		Initializes a volume as a standard FAT drive. This is not protected by the SafeFAT journaling mechanisms.
f_delvolume()		Deletes an existing volume.
f_setvolname()		Sets the name of a volume.
f_getvolname()		Gets the name of a volume.
f_get_volume_count()		Gets the number of active volumes.
f_get_volume_list()		Gets a list of all the active volumes.
f_initvolumepartition()		Initializes a volume on an existing partition.
f_initvolumepartition_nonsafe()		Initializes a volume on an existing partition as a standard FAT drive. This is not protected by the SafeFAT journaling mechanisms.
f_createpartition()		Creates one or more partitions on a drive, or removes partitions by overwriting the current partition table.
f_createpartition_align()		Creates one or more partitions on a drive, aligned to given sector boundaries, or removes partitions by overwriting the current partition table.
f_getpartition()		Gets the used sectors and system indication byte from a partitioned medium.
f_createdriver()		Initializes a driver.
f_releasedriver()		Releases a driver when it is no longer required.

f_mountfat

Use this function to mount a drive from a FAT filesystem.

For detailed information about usage, please consult the **f_initvolume()** section of the *HCC FAT and SafeFAT File System User Guide*.

Format

```
int f_mountfat (
    int             drivenum,
    F_DRIVERINIT    driverinit,
    unsigned long   user_ptr)
```

Arguments

Argument	Description	Type
drivenum	Drive to be initialized (0:A, 1:B, ...).	int
driverinit	A pointer to the initialization function for the drive.	F_DRIVERINIT
user_ptr	This can optionally be used to pass information to the low level driver.	unsigned long

Return values

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

Example

```
 void main()
{
   int ret;

   f_init();

   f_enterFS();

   ret = f_mountfat( 0, ram_initfunc, 0 );

   printf ( "RAM drive is mounted with error code: %d", ret );
}
```

f_mountfatpartition

Use this function to mount a drive from FAT.

For detailed information about usage, see **f_initvolumepartition()** in the *HCC FAT and SafeFAT File System User Guide*.

Note:

- This function is only needed if the medium (for example, hard disk) contains multiple FAT partitions. In all other cases **f_mountfat()** can be used (there is only one partition on the drive).
- To get the proper F_DRIVER pointer for this function, please refer to the *HCC FAT and SafeFAT File System User Guide* and use the **f_createdriver()** and **f_releasedriver()** functions. Please also check the partition functions **f_createpartition()** and **f_getpartition()** in that user guide.

Format

```
int fw_mountfatpartition (
    int          drivenum,
    F_DRIVER *   driver,
    int          partition)
```

Arguments

Argument	Description	Type
drivenum	Drive to be initialized (0=A, 1=B, and so on).	int
driver	The initialized driver.	F_DRIVER *
partition	The partition requested from the media.	int

Return values

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

f_mountsafe

Use this function to mount a drive from SAFE.

For detailed information about usage, see the *HCC FAT and SafeFAT File System User Guide*.

Format

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

Arguments

Argument	Description	Type
drivenum	The number of the drive to mount (0:A, 1:B, ...).	int
buffer	A buffer pointer to be used by the file system.	void *
buffsize	The size of the buffer.	long
mountfunc	The mount function for the selected drive type.	FS_DRVMOUNT
phyfunc	The physical driver for the specific chip type.	FS_PHYGETID

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_mountsafe(0, buffer, sizeof(buffer),
                fs_mount_flashdrive, fs_phy_nor_29lvxxx);

    /* Drive A will be NOR flash drive */
    if (ret == F_NO_ERROR) return;  /* initialized */
    if (ret == F_ERR_NOTFORMATTED)
    {

        ret = f_format(0);              /* format drive A */
        if (ret == F_NO_ERROR) return; /* formatted */
    }
initializationfailed:
    /* Perform fatal error handling */
}
```

Directory Management

All the following functions are the same in SafeFAT and SafeFLASH.

Function	Description
f_mkdir()	Creates a new directory.
f_chdir()	Changes the current working directory.
f_rmdir()	Removes a directory.
f_getcwd()	Gets the current working directory on the current drive.
f_getdcwd()	Gets the current working directory on the selected drive.

File Access

The following functions are available. Refer to the relevant manual for more details.

FAT/SafeFAT	SafeFLASH	Description
f_open()	**f_open()**	Opens a file.
f_open_nonsafe()		Opens a file without the journaling enabled.
f_close()	**f_close()**	Closes a previously opened file.
f_abortclose()		Closes a previously opened file, aborting all operations.
f_flush()	**f_flush()**	Flushes an opened file to a storage medium.
f_read()	**f_read()**	Reads bytes from the current position in the specified file.
f_write()	**f_write()**	Writes data into a file at the current position.
f_getc()	**f_getc()**	Reads a character from the current position in the specified open file.
f_putc()	**f_putc()**	Writes a character to the specified open file at the current file position.
f_eof()	**f_eof()**	Checks whether the current position in the specified open file is the end of file.
f_seteof()	**f_seteof()**	Moves the end of file (EOF) to the current file pointer.
f_tell()	**f_tell()**	Obtains the current read-write position in the specified open file.
f_seek()	**f_seek()**	Moves the stream position in the specified file.
f_rewind()	**f_rewind()**	Sets the file position in the specified open file to the start of the file.
f_truncate()	**f_truncate()**	Opens an existing file for writing and truncates it to the specified length.
f_ftruncate()	**f_truncate()**	Truncates a file that is open for writing to the specified length.

File Management

The following functions are available. Refer to the relevant manual for more details.

FAT/SafeFAT	SafeFLASH	Description
f_delete()	**f_delete()**	Deletes a file.
f_deletecontent()		Deletes a file and also its contents; that is, sets all the content to 0xFF.
f_findfirst()	**f_findfirst()**	Finds the first file or subdirectory in a specified directory.
f_findnext()	**f_findnext()**	Finds the next file or subdirectory in a specified directory.
f_move()	**f_move()**	Moves a file or directory. The original file or directory is lost.
f_rename()	**f_rename()**	Renames a file or directory.
f_getattr()		Gets the attributes of a file.
f_setattr()		Sets the attributes of a file.
f_gettimedate()	**f_gettimedate()**	Gets time and date information from a file or directory.
f_settimedate()	**f_settimedate()**	Sets time and date information for a file or directory.
	f_getpermission()	Gets the file or directory permission field associated with a file.
	f_setpermission()	Sets the file or directory permission field associated with a file.
f_stat()	**f_stat()**	Gets information about a file.
f_fstat()	**f_fstat()**	Gets information about a file by using its file handle.
f_filelength()	**f_filelength()**	Gets the length of a file.

4.3 File System Unicode API

Unicode Directory Management

All the following functions are the same in SafeFAT and SafeFLASH.

Function	Description
f_wmkdir()	Creates a new directory with a Unicode 16 name.
f_wchdir()	Changes the current working directory.
f_wrmdir()	Removes a Unicode 16 directory.
f_wgetcwd()	Gets the current working directory on the current drive.
f_wgetdcwd()	Gets the current working directory on the selected drive.

Unicode File Access

The following functions are available. Refer to the relevant manual for more details.

FAT/SafeFAT	SafeFLASH	Description
f_wopen()	**f_wopen()**	Opens a file with a Unicode 16 filename.
f_wopen_nonsafe()		Opens a file without the journaling enabled.
f_wtruncate()	**f_wtruncate()**	Opens an existing file for writing and truncates it to the specified length.

Unicode File Management

The following functions are available. Refer to the relevant manual for more details.

FAT/SafeFAT	SafeFLASH	Description
f_wdelete()	**f_wdelete()**	Deletes a file with a Unicode 16 name.
f_wdeletecontent()		Deletes a Unicode 16 file and also its contents.
f_wfindfirst()	**f_wfindfirst()**	Finds the first Unicode 16 file or subdirectory in the specified directory.
f_wfindnext()	**f_wfindnext()**	Finds the next Unicode 16 file or subdirectory in a specified directory after a previous call to **f_wfindfirst()** or **f_wfindnext()**.
f_wmove()	**f_wmove()**	Moves a Unicode 16 file or directory.
f_wrename()	**f_wrename()**	Renames a Unicode 16 file or directory.
f_wgetattr()		Gets the attributes of a Unicode 16 file.
f_wsetattr()		Sets the attributes of a Unicode 16 file.
f_wgettimedate()	**f_wgettimedate()**	Gets time and date information for a Unicode 16 file or directory.
f_wsettimedate()	**f_wsettimedate()**	Sets time and date information for a Unicode 16 file or directory.
f_wfilelength()	**f_wfilelength()**	Gets the length of a Unicode 16 file.
f_wstat()		Gets information about a Unicode 16 file.
	f_wgetpermission()	Gets the permission field of a Unicode 16 file or directory.
	f_wsetpermission()	Sets the permission field of a Unicode 16 file or directory.

Unicode Translation

The following functions are only available in FAT or SafeFAT. See the *HCC FAT and SafeFAT File System User Guide* for more details.

Function	Description
f_set_ascii_to_unicode()	Converts one or two single byte ASCII characters to a single Unicode wide-byte character.
f_set_unicode_to_ascii()	Converts a single Unicode wide-byte character to one or two single byte ASCII characters.

4.4 Error Codes

The following table lists the error codes from the two systems in a single list. For detailed information about their causes, consult the respective file systems.

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 **f_seek()**.
F_ERR_LOCKED	12	The file has already been opened for writing /appending.
F_ERR_ACCESSDENIED	13	The necessary physical read and/or write functions are not present for this volume.
F_ERR_NOTEMPTY	14	The directory to be 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_MEDIATOOLARGE	26	Media is too large for the format type requested.
F_ERR_NOTSUPPSECTORSIZE	27	The sector size is not supported. The only supported sector size is 512 bytes.
F_ERR_UNKNOWN	28	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 – normally because **f_enterFS()** has not been called.
F_ERR_UNUSABLE	38	The file system has become unusable, normally due to 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.

4.5 Types and Definitions

This section describes the main elements that are defined in the API Header file.

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

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

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

F_FIND

The *F_FIND* structure takes this form:

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

F_WFIND

The *F_WFIND* structure takes this form:

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

F_SPACE

The *F_SPACE* structure takes this form:

Element	Type	Description
total	unsigned long	The total size in bytes of the disk.
free	unsigned long	The number of free bytes on the disk.
used	unsigned long	The number of used bytes on the disk.
bad	unsigned long	The number of bad bytes on the disk.
total_high	unsigned long	The high part of *total* if greater than 4GB.
free_high	unsigned long	The high part of *free* if greater than 4GB.
used_high	unsigned long	The high part of *used* if greater than 4GB.
bad_high	unsigned long	The high part of *bad* if greater than 4GB.

F_STAT

The *F_STAT* structure takes this form:

Element	Type	Description
filesize	unsigned long	The size of the file.
createdate	unsigned short	The creation date.
createtime	unsigned short	The creation time.
modifieddate	unsigned short	The last modified date.
modifiedtime	unsigned short	The last modified time.
lastaccessdate	unsigned short	The last accessed date.
attr	unsigned char	00ADVSHR
drivenum	int	The number of the volume.

Directory Entry Attributes

Directory entries, meta-description elements for files and directories, can have attributes assigned to them. These are detailed in the table below.

Attribute	Description
F_ATTR_ARC	Is an archived file or directory.
F_ATTR_DIR	Is a directory.
F_ATTR_VOLUME	Is a volume.
F_ATTR_SYSTEM	Is a system file or directory.
F_ATTR_HIDDEN	Is a hidden file or directory.
F_ATTR_READONLY	Is a read-only file or directory.

5 System Test

Test programs are supplied for exercising the file system and ensuring that it works correctly. Most of the functions of the file system are exercised by the programs, including file read, write, append, seek, file content, directories, and file manipulation functions. To use the test programs, include **fw_test.c** and **fw_test.h** in your test project.

Call the following to execute the test code:

```
void f_dotest(void)
```

The FAT or SafeFLASH system can be tested separately in the CAPI. Include **fw_tesport_f.c** for FAT and **fw_testport_s.c** for SafeFLASH.

These files must be ported for your operating system (for example, if that system uses a different function for printing than **printf()**, or uses drives other than RAM drives). For details, see the implementation example and the comments in the test files.

6 Integration

This section describes all aspects of the file system that require integration with your target project.

This includes porting and configuration of external resources.

6.1 OS Abstraction Layer

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

The file system uses the following OAL components:

OAL Resource	Number required if FW_MAXTASK is 1	Number required if FW_MAXTASK > 1
Tasks	0	0
Mutexes	0	1 + FAT_MAXVOLUME
Events	0	0

Multiple Tasks, Mutexes and Reentrancy

Each file system implements its own volume protection but one additional mutex is required by the CAPI to control access to the global table for volumes from both systems. This mutex will only be held for very sort periods of time when the CAPI global volume table is being modified.

> **Note:** If your system has multiple tasks that access the file system, you must implement this section.

6.2 PSP Porting

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

The module makes use of the following standard PSP functions:

Function	Package	Element	Description
psp_memcpy()	psp_base	psp_string	Copies a block of memory. The result is a binary copy of the data.
psp_memset()	psp_base	psp_string	Sets the specified area of memory to the defined value.