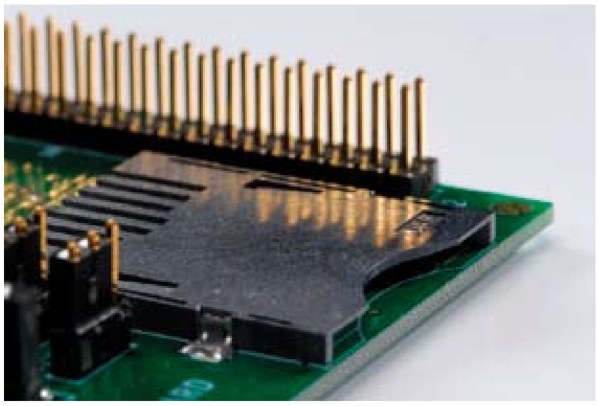
File Systems

Introduction

HCC-Embedded is a specialist developer of embedded firmware, and in particular file systems for embedded applications. Years of work on embedded systems has led to the conclusion that no one solution is suitable for all designs, one size does not fit all. Consequently HCC has developed a coherent range of systems to suit most application needs.

All the file systems have been designed for use on embedded applications, with every thought given to efficient use of resources and extracting the best possible performance within the design criteria of that system. We are confident that these products are second to none. This brochure explains the various options and relevant features in each HCC file system. Understanding these matters will help users to achieve optimal designs for their embedded applications.

Index



About HCC File Systems

All HCC file systems include the following features:

- written in ANSI C
- delivered in full C source code
- standard file API
- RTOS independent
- Royalty free license
- One year of technical support (extensible)
- Complete file system test suite
- Comprehensive documentation
- Selection of standard tested drivers

When buying into HCC's file system products, you are also automatically buying into HCC's experienced development team, which strives to maintain the high standards they have set, as memory storage and microcontroller technology move ever forward.

Contact Information

US office:
444 East 82nd Street
New York
NY 10028
Tel: +1 212 734 1345

Hungarian office:
1133 Budapest
Váci út 76.
Tel.: +36 1 450 1302
Fax: +36 1 450 1303

support@hcc-embedded.com
sales@hcc-embedded.com
www.hcc-embedded.com

About Fail-safety

Whenever possible, HCC encourages embedded developers to use file systems that are robust in the event of unexpected events, particularly power-loss or unexpected reset.

The ubiquitous FAT file system, for example, is not fail-safe. The fundamental problem is that to make a new entry in FAT consistent, more than one area of the disk must be modified. This is logically impossible to achieve. The problems this causes for embedded systems are numerous. And, although a check-disk program can recover some situations, this normally requires user intervention and decision making. For product designers who value or depend on the data stored in their embedded devices, a fail-safe system should be used.

What do we mean by Fail-safe?

HCC's fail-safe file systems are designed to handle unexpected resets of the system, such that the file system always remains in a consistent state. It does not require complex error recovery. At all times a file is in a state that is consistent with its meta-data, and when a file is flushed or closed, it is atomically switched into its new state.

Even complex operations such as seeking back into a file are handled safely and efficiently using a process that mirrors sectors with changed elements. There is never a need to worry about data consistency, or to invoke complex error recovery; the system is always consistent.

Journaling or transaction?

A journaling file system generally guarantees only the integrity of the metadata, and is not always deterministic. A transaction based file system provides integrity for both file data and metadata, though the commit points are normally system wide.

HCC offers the best of all worlds in its fail-safe implementation; that is, all our systems are transaction based but not over the whole system. A single file operation can be done without reference to the state of other files or operations in the system.

Related Issues

There is little point in building a fail-safe file system without defining what is required of the low-level media and driver. It is clearly impossible for a file system to behave in a fail-safe way if the media are not reliable.

With all HCC fail-safe file systems, the requirements of the low-level driver are defined in order to guarantee the reliable operation of the fail-safe system.

It is also important to note that in most systems involving flash storage, careful management of the power to the media can also be critically important.

HCC's development team can offer much insight into the development of reliable file system solutions.

Fail-safe Verification

HCC has invested significant resources in order to insure that the claims of fail-safety hold true.

For all our fail-safe file systems, we have created simulation environments that are designed to insure the robustness of the system through random reset and system verification on restart.

In addition, HCC has built test harnesses for each system, in which an external controller randomly interrupts the power to the target system. In order to insure integrity, these tests are run for weeks on various hardware configurations.

File Systems API and CAPI

All HCC file systems support a comprehensive, standard API. There are differences among the file systems in the initialization, volume and partition control functions, because of their differing functionalities. All the file and directory manipulation functions are entirely standard and 100% compatible across file systems.

Additionally HCC provides its Common API (CAPI) for embedded file systems. This allows any combination of SafeFLASH, FAT and SafeFAT volumes to be used under a single API wrapper. The various drives appear as a standard array of drives with a common API. The files that are managed in the file system may be quite different but the user interface is entirely consistent.

The table below lists the functions that are common to all the file systems.

File Operations

F_FILE *fopen(const char *filename, const char *mode);	Open a file
int f_close(F_FILE *filehandle)	Close a file
int f_flush(F_FILE *filehandle)	Flush a file to disk
long f_write(const void *buf, long size, long size_st, F_FILE *filehandle)	Write data to a file
long f_read(void *buf, long size, long size_st, F_FILE *filehandle)	Read data from a file
long f_seek(F_FILE *filehandle, long offset, long whence)	Seek to a new position in a file
long f_tell(F_FILE *filehandle)	Tell the current file pointer
int f_eof(F_FILE *filehandle)	Test if at end of file
int f_seteof(F_FILE *filehandle)	Set end of file
int f_rewind(F_FILE *filehandle)	Rewinds the file pointer to 0
int f_putc(char ch,F_FILE *filehandle)	Put a character to the file
int f_getc(F_FILE *filehandle)	Get a character from the file
F_FILE *f_truncate(const char *filename, unsigned long length)	Truncate an open file
int f_ftruncate(F_FILE *filehandle, unsigned long length)	Truncate a file
int f_delete(const char *filename)	Delete a file

Directory/Volume Operations

int f_mkdir(const char *dirname)	Make directory
Int f_chdir(const char *dirname)	Change directory
int f_rmdir(const char *dirname)	Remove directory
int f_getdrive(void)	Get current drive
int f_chdrive(int drivenum)	Change drive
int f_getcwd(char *buffer, int maxlen)	Get current working directory
int f_getdcwd(int drivenum, char *buffer, int maxlen)	Get cwd of specified drive
int f_rename(const char *filename, const char *newname)	Rename a file
int f_move(const char *filename, const char *newname)	Move a file

Note: Unicode equivalent functions are provided for those file systems that support Unicode. In this case we use the standard Unicode syntax for functions. For example,

F_FILE *fopen(const char *filename, const char *mode);

Is replaced by

F_FILE *fwopen(const wchar *filename, const wchar *mode)

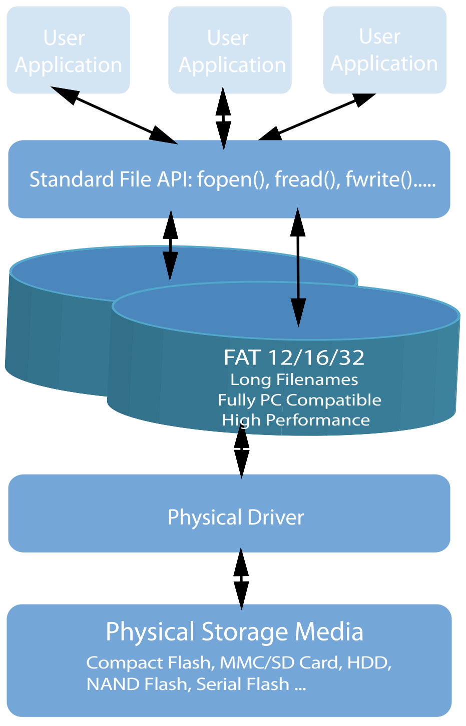
File Systems

FAT File System

FAT is a full featured, high-performance file system for use in embedded applications that need to attach FAT12/16/32 compliant media to a product. Typically this might be an SD card, Compact flash card or USB pen-drive, but may also address any device which is arranged as an array of logical sectors.

Key features:

- Very High performance
- FAT12/16/32
- Long filenames
- 512, 1K, 2K, and 4K sector support
- Unicode 16
- Multiple volumes
- Multiple simultaneous open files
- Multiple users of same open file
- Partition creation and management
- Handles media errors
- Cache options for better performance
- Secure deletion option
- Standard drivers for SD/SDHC/MMC/SafeFTL/USB-MST/HDD/RAM



FAT/SafeFAT/THIN

The low-level driver interface is the link between the FAT and the physical media. Implementation is straightforward, and is accelerated by a set of sample drivers and detailed documentation.

The functions that must be implemented in the driver are:

`GetPhy ()`	get information about the attached device (e.g., number of sectors, FAT type)
`GetStatus ()`	get status of the device (e.g., card removed, card changed, write protect)
`ReadSector ()`	read the specified sector of the attached device
`WriteSector ()`	write data to the specified sector

The following two functions are optional:

`ReadMSectors ()`	read multiple sequential sectors from the device
`WriteMSectors ()`	write multiple sequential sectors to the device

THIN File System

THIN is a full-featured FAT file system designed for embedded device developers, with limited resources available to them, that wish to attach PC compatible media (such as an SD card of USB pen-drive) to their devices. The code has been highly engineered to extract the most out of the target system in terms of speed and performance. The various build options allow the developer to make trade-offs between the system requirements and the available ROM and RAM requirements and the available ROM and RAM.

FAT v THIN

These are the main limitations that THIN imposes on your system:

1. Support for only one volume
2. No multi-sector read/write
3. No cache options
4. No multi-partition support
5. No files open simultaneously by multiple users

One of the primary results of these limitations will be that performance will generally be lower on THIN than FAT. But it should be noted that if FAT uses the same limited RAM allocation as a THIN system the performance difference will be small.

In general, assuming there is no resource limitation issue, FAT should be used since it gives the most flexibility for further development and performance improvement as required.

SafeFAT File System

SafeFAT is identical to the FAT system in all respects except that it has a system of journaling operations that allows it to ensure the system's integrity in the event of an unexpected reset occurring. While the method is journaling the result is a transaction based system where the transactions are operation based rather than system wide.

It is particularly recommended for use on embedded systems with integrated storage that cannot rely on user intervention and check-disk to fix any errors which occur.

Key Features:

- High Performance
- Fail-safe operation
- FAT12/16/32
- Long filenames
- 512, 1K, 2K, and 4K sector support
- Unicode 16
- Multiple volumes
- Multiple simultaneous open files
- Partition creation and management
- Handles media errors
- Cache options for better performance
- Secure deletion option
- Standard drivers for SD/SDHC/MMC/SafeFTL/USB-MST,HDD,RAM

The low level driver interface is identical to that of the FAT system.

SafeFAT has been, and continues to be, very extensively tested by HCC to insure its integrity. This is done both by complex simulated environments and by actual implementations with random reset of the hardware while the system is in operation. This comprehensive testing has given us confidence that SafeFAT is a major breakthrough in providing reliable file system storage for embedded systems.

SafeFAT is fully PC compatible, but there is an exception: Only a system running SafeFAT can recover any damage done by an unexpected reset. A PC cannot do this. HCC also provides a PC utility for repairing SafeFAT drives that have experienced unexpected reset. In normal usage this utility is not required.

SafeFLASH File System

SafeFLASH was designed from its inception to be a high-performance, 100% fail-safe flash file system for use in embedded applications. It can be used with all NOR and NAND flash types as well as any media that can simulate a block structured array. SafeFLASH is highly portable. It has been integrated and proven with many RTOSes on hundreds of products of very varied architectures. The low-level interface is abstracted to give the simplest possible porting layer for easy integration with any target system.

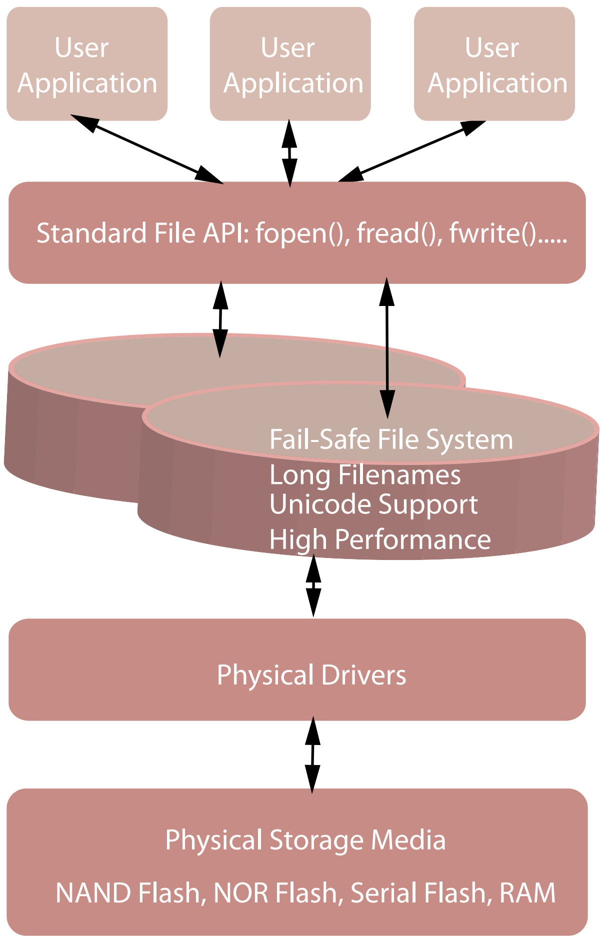
Key features:

- Fail-safe operation
- Long filenames
- Unicode 16
- Multiple volumes
- Multiple media types
- Multiple files open simultaneously
- Multiple users of same open file
- Media error handling
- Static wear-leveling
- Dynamic wear-leveling
- Supports all NOR flash types
- Supports all NAND flash types

The low-level driver is designed for straightforward porting to any target device. Normally it takes only a few hours to integrate with a specific target, particularly if one of our many sample drivers is used as a starting point. To our knowledge Safe-FLASH can operate with all flash devices.

SafeFLASH has three orderable components:

SafeFLASH-BS	the base file system
SafeFLASH-NOR	driver for all NOR flash types
SafeFLASH-NAND	driver for all NAND flash types



SAFE File System

Storing data to NOR or NAND flash on an embedded system, efficiently and reliably, is a non-trivial task. There are many issues to consider such as unexpected power loss, partial write or erase, wear-leveling and error correction. For many product developers it is imperative that this data is handled in a fail-safe way to ensure the stability of their product and the integrity of their data. HCC have developed SafeFLASH to meet this need – and through continuous revision HCC maintains the system for use with the latest devices in an ever developing technology area.

TINY File System

TINY is a full-featured flash file system targeted for use on embedded devices with limited resources.

TINY is specifically designed to work with flash devices that have small erasable sectors. In general this applies to many serial flash types as well as the internal flash of some microcontrollers. It also applies to RAM drives. By small erasable sectors we generally mean <2Kbytes. By limiting the use of TINY to this subset of NOR flash devices it is possible to remove many fragmentation and flash management issues and to thus create a compact and reliable system.

Key features:

- Fail-safe
- Highly scalable
- Minimal footprint
- RAM usage <200 Bytes
- Code usage 4K-10K
- Support for many small sector flash types
- RAM drive
- Multiple simultaneous open files

Examples of supported media:

Atmel Dataflash AT45xxxx

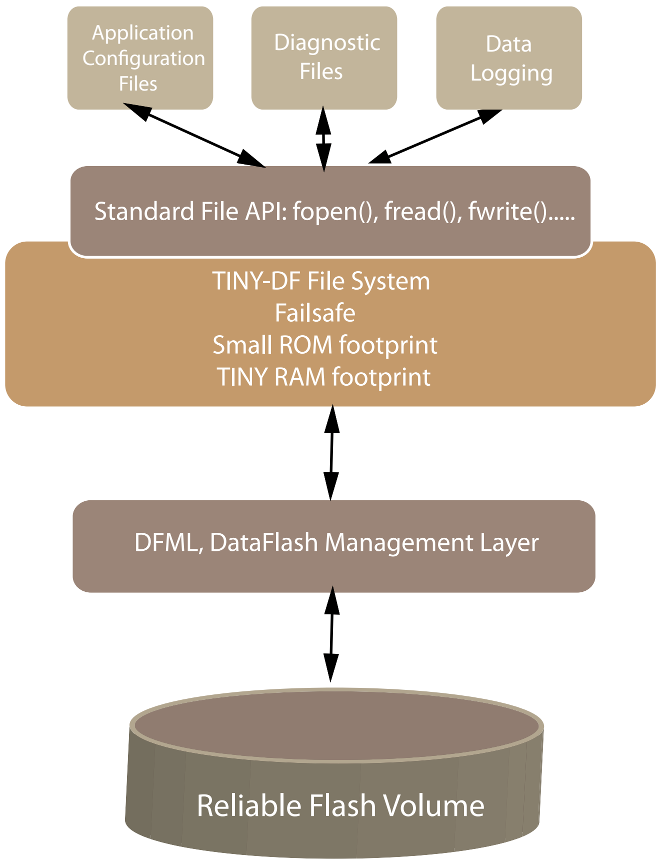
MSP430 internal flash

Ramtron FRAM

SST serial flash

ST M25PExx, M45PExx

Please note that there are two variants of TINY-BW (byte writable) and TINY-PW (page writable). The byte writable variant is for flash that can be written to byte by byte, whereas the PW variant is for devices that can be written to only in complete pages.


Application Configuration Files
Diagnostic Files
Data Logging
Standard File API: fopen(), fread(), fwrite().....
TINY-DF File System
Failsafe
Small ROM footprint
TINY RAM footprint
DFML, DataFlash Management Layer
Reliable Flash Volume

TINY-DF

TINY v SafeFLASH

TINY should only be chosen ahead of the SafeFLASH file system if:

1. The target system has very limited code or data area and
2. The target flash has the possibility to erase in small sectors typically <4K

In all other cases SafeFLASH should be used as this system will give better performance, extensibility and much better wear-level characteristics.

File System Comparison Table

File System Comparison Table					
	SafeFLASH	TINY	THIN	FAT	SafeFAT
Code Size[1]	40K	10K	4-12K	35K	45K
RAM Size	[2]	<256Bytes	0.7K<x<2K	>3K	>6K
Fail-safe	y	y	n	n	y
RTOS Abstraction	y	y	n	y	y
ANSI C	y	y	y	y	y
Long filenames	y	y	y	y	Y
Unicode	y	n	n	y	y
Multiple Open files	y	y	y	y	y
Multiple user of open files	y	n	n	y	y
Multiple Volumes	y	n	n	y	y
Multisector read/write	na	na	n	y	y
Partition Handling	n	n	n	y	y
Media Error Handling	y	n	n	y	y
Test Suite	y	y	y	y	y
Imaging tool	y	d	na	na	na
Performance	High	Low	Low-Medium	V. High	High
Cache Options	y	n	n	y	y
Zero Copy	y	y	y	y	y
Static Wear Leveling	y	n	na	na	na
Dynamic Wear Leveling	y	y	na	na	na
Re-entrant	y	y	n	y	y
CAPI Support	y	n	n	y	y
Secure Delete Option	y[5]	n	n	y[6]	y[6]
FAT12/16/32 Compatible	n	n	y	y	y
Media Types					
NAND Flash[3]	y	n	y[7]	y[7]	y[7]
NOR Flash[4]	y	n	n	n	n
Small sector NOR flash[8]	y	y	n	n	n
MMC/SD/SDHC cards	n	n	y	y	y
CF cards	n	n	y	y	y
HDD	n	n	y	y	y
USB MST	n	n	y	y	Y
RAM	y	y	y	y	y

Key: y – yes; n – no; na – not applicable; d – in development;
Notes:
1. Approximate numbers based on ARM7 at high optimization
2. The RAM usage depends on the configuration and flash type. HCC provides a tool for calculating this number
3. All NAND flash devices are supported; all will operate better with help of a NAND flash controller
4. All types of NOR flash are supported
5. NOR flash only
6. Needs driver support to do this
7. When used with HCC's SafeFTL
8. This is any NOR flash that has small erasable sectors; typically <4K. This includes many types of serial flash and also the integrated flash of some microcontrollers.