Application Note

AN101

Choosing An Appropriate File System

Contents

Choosing an Appropriate File System

HCC provides five file systems, each designed to achieve the best balance of performance and resource utilisation in their targeted embedded configuration. These file systems share a common API, to ensure portability, and can interface with any type of sector-based media. This document helps designers choose the file system that best suits their needs.

FAT Compatible File Systems

HCC supply three FAT based file systems

- **FAT:** full featured FAT file system that can be optimized to suit the memory or performance requirements of an embedded application.
- **THIN:** provides most functionality of HCC's FAT file system, optimized to use minimal ROM/RAM.
- **SafeFAT:** comprehensive FAT file system designed to be truly fail-safe. SafeFAT protects against unexpected reset or power loss.

Flash File Systems

HCC has two Flash file systems designed to meet the demands of high performance flash media in embedded systems.

- **SafeFlash** is a high performance truly fail-safe file system that can be used with all NOR and NAND flash or any media that can simulate a block-structured array. SafeFLASH supports dynamic and static wear- leveling and provides a highly efficient solution for products in which data integrity is critical.
- **TINY** is a full-featured, fail-safe flash file system for use in resource-constrained applications. TINY is designed for use with NOR Flash with erasable sectors <4kB. Typical devices supported include all common serial flash and embedded versions such as TI MSP430 internal flash.

Flash Translation Layer

SafeFTL is an advanced fail-safe Flash Translation Layer used to provide a logical, sector-based interface for any application or file system using NAND or NOR Flash devices. SafeFTL manages the underlying complexity of Flash based media devices and can be used in conjunction with any of HCC's FAT file systems.

Advanced Fail Safety

HCC has invested a great deal of research, test and development effort to design truly fail-safe file systems that will always recover from unexpected system events such as power loss or reset. A standard FAT file system is not fail-safe and therefore risks becoming corrupt – this is not normally acceptable in an embedded environment. The fundamental problem is that to make a new entry in a FAT consistent, more than one area of the disk must be modified in a single, uninterrupted action. This is logically impossible to achieve. 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 is strongly recommended.

Journal based file systems generally guarantee only the integrity of the metadata and are 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 employs a hybrid approach for its fail-safe file systems and all our implementations are transaction based on a file-by-file basis. This has the advantage that a single file operation can be executed without reference to the state of other files or operations, meaning each application using the file system can operate safely and independently.

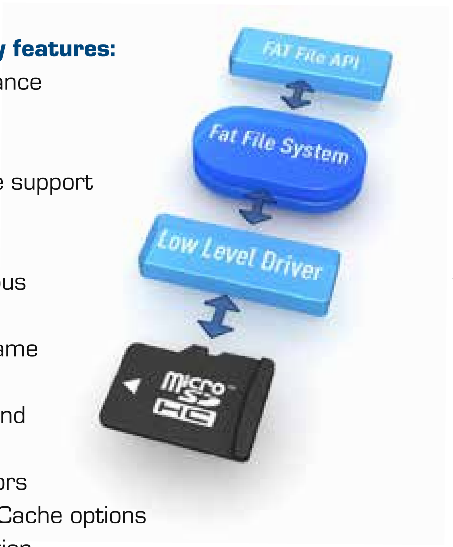
Any file system claiming fail-safety must define what is required of the low-level media driver to guarantee fail-safety. With all HCC fail-safe file systems, the requirements of the low-level driver are clearly defined. This enables designers to create systems that will survive unexpected reset or power failure. It is important to note that in most systems involving flash storage, careful management of the power to the target media is critical. HCC's experienced team can offer insight into the design of reliable file system solutions.

For all fail-safe file systems HCC have created simulation environments that are designed to ensure the robustness of the system through random reset and system verification on restart. HCC develops test harnesses for each system, in which an external controller randomly interrupts power to the target system. In order to ensure integrity, these tests are run continuously for weeks using multiple hardware configurations.

* Note: ATMEL® and DataFlash® are the registered trademarks of Atmel.

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 that is arranged as an array of logical sectors.



FAT File System Key features:

- Very High performance
- FAT12/16/32
- Long filenames
- Variable sector size support
- Unicode 16
- Multiple volumes
- Multiple simultaneous open files
- Multiple users of same open file
- Partition creation and management
- Handles media errors
- High Performance Cache options
- Secure deletion option
- Standard drivers available for SD /SDHC /SDXC /MMC /eMMC /SafeFTL /USB-MST /HDD /RAM

THIN File System

THIN is a full-featured FAT file system designed for embedded MCUs with limited system resources that wish to attach PC compatible media like SD cards or pen-drives to their devices. The software has been carefully engineered to enable the optimal balance of performance, functionality and available resource.

THIN has the following limitations compared to the full-featured FAT file system:

- Support for single volume only
- No multi-sector read/write
- No cache options
- No multi-partition support
- No files open simultaneously by multiple users
- Only supports 512byte sector size
- RAM usage 0.7–2kB
- Code usage 4–12kB

One of the main limitations if using THIN is that performance will generally be lower than that achievable using FAT – but it should be noted that if FAT uses the same limited RAM allocation as a THIN system, the performance will be similar. If there are no resource limitations, use of the FAT file system is recommended, because it provides flexibility for future development and performance improvement.

SafeFAT File System

SafeFAT is an enhanced version of FAT. It has an identical API but uses a hybrid transaction and journaling system to ensure file system integrity in the event of unexpected reset or power loss. HCC's hybrid approach is particularly recommended for use with integrated storage that cannot rely on user intervention and check-disk to fix errors.

Key Features:

- High Performance
- Fail-safe operation
- FAT12/16/32
- Long filenames
- Variable sector size 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 available for SD /SDHC /SDXC / MMC /eMMC /SafeFTL /USB-MST /HDD /RAM

SafeFAT has been extensively tested to ensure its integrity using both complex simulation environments and physical tests with injection of random failure events. This rigorous approach makes SafeFAT the ideal file system for high availability applications.

SafeFLASH File System

Storing data on NOR or NAND flash efficiently and reliably is a non-trivial task. SafeFLASH is a truly fail-safe flash file system designed specifically for embedded MCUs. It can be used with any NOR or NAND flash or any media that can simulate a block structured array. SafeFLASH is highly portable and has been used by hundreds of companies worldwide to ensure the integrity of their applications.

- Fail-safe operation
- Long filenames
- Unicode 16
- Multiple volumes
- Multiple media types
- Multiple files open simultaneously
- Multiple users of same open file
- Optional CRC file integrity
- ECC handling
- Media error handling
- Static wear-leveling
- Dynamic wear-leveling
- Bad block management
- Garbage collection
- Supports all NOR flash types
- Supports all NAND flash types
- MCU/NAND controller support

When integrating NAND flash with an embedded system, there are many design considerations e.g. unexpected power loss, partial write or erase, wear-leveling and error correction. NAND flash devices vary widely in their requirement to use Error Correction Codes (ECC) to guarantee the specified erase/write life. HCC can supply software based ECC algorithms or can supply drivers that support hardware based ECC implementations.

All flash devices (NOR & NAND) suffer from 'wear' due to repeated write/erase operation, SafeFLASH includes comprehensive and efficient wear-management algorithms.

TINY Flash File System

TINY is a full-featured flash file system for use on MCUs with limited resources. TINY is specifically designed for flash devices that have small erasable sectors, typically <4kB. This includes many serial flash devices or even the internal flash on some MCUs. By limiting the application of TINY to this subset of NOR flash devices, many fragmentation and flash management issues have been eliminated to make TINY a compact and reliable file 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:

- AT45 like serial flash (e.g. Adesto, Spansion, Microchip)
- MSP430 internal flash
- Ramtron FRAM
- SST serial flash (Microchip)
- ST M25PExx, M45PExx (Micron)
- Winbond, Macronix and many more

TINY should only be chosen in preference to SafeFLASH if the target system has limited memory available.

Smart-meter File System

Instead of using a traditional file based system, HCC has taken the radical approach of defining a system built around the needs of smart-meters. Metering applications usually have well defined record structures and HCC has used its' extensive flash experience to take advantage of this characteristic. By taking a data focused, and not a file focused approach, it is possible reduce the required number of write/erase cycles by an order of magnitude. Traditional file systems do not have built-in cyclic buffer logic for storing records and this can add complexity, significantly increasing the number of times flash must be accessed. SMFS uses a structured database to reduce complexity of the application which can improve the performance in almost every way; speed, power consumption, and flash life.

Benefits of SMFS:

- Fail-safe data storage for guaranteed system recovery.
- Persistent data storage for 15 years or more.
- Features developed to significantly lower manufacturing BOM cost.
- Minimum number of flash operations to preserve both the flash and the battery.
- Deterministic behavior in the event of unexpected reset.

File System Feature and Performance Comparison

	FAT	THIN	SafeFAT	SafeFLASH	TINY
Code Size[1]	~24kB	4-12.5kB	~31KB	17-20kB	8.2kB
RAM	>3KB	0.7->2KB	>6KB	varies[2]	<256bytes
Fail Safety	N	N	Y	Y	Y
ANSI 'C'	Y	Y	Y	Y	Y
Long Filename Support	Y	Y	Y	Y	Y
Unicode	Y	N	Y	Y	N
Multiple Open Files	Y	Y	Y	Y	Y
Multiple Open File Users	Y	N	Y	Y	N
Multiple Volumes	Y	N	Y	Y	N
Multi-sector R/W	Y	N	Y	n/a	n/a
Partition Handling	Y	N	Y	N	N
Media Error Handling	Y	N	Y	Y	N
CRC on files (optional)	N	N	N	Y	N
Test Suite	Y	Y	Y	Y	Y
Relative Performance	very high	med/low	high	high	low
Cache Option	Y	N	Y	Y	N
Zero Copy	Y	Y	Y	Y	Y
Static Wear Leveling	n/a	n/a	n/a	Y	N
Dynamic Wear Leveling	n/a	n/a	n/a	Y	Y
Re-entrant	Y	N	Y	Y	Y
CAPI Support	Y	N	Y	Y	N
Secure Delete Option	Y[4]	N	Y[4]	Y[3]	N
FAT Compatible	Y	Y	Y	N	N

1. Approximate numbers based on Cortex-M3 at high optimisation – long file names active, can be reduced further with LFN off.
2. RAM usage depends on the configuration and flash type. HCC provides a tool for calculating this number
3. NOR flash only
4. Needs special driver support

Supported Media

	FAT	THIN	SafeFAT	SafeFLASH	TINY
NAND Flash	Y*	Y*	Y*	Y	N
NOR Flash	Y*	Y*	Y*	Y	Y
Small Sector NOR	Y*	Y*	Y*	Y	Y
MMC/eMMC/SD/SDHC/SDXC	Y	Y	Y	N	N
Compact Flash	Y	Y	Y	N	N
SSD Flash	Y	Y	Y	N	N
USB Mass Storage	Y	Y	Y	N	N
RAM	Y	Y	Y	Y	Y

* Requires SafeFTL flash translation layer.

Appendix A:
Using Multiple File Systems, Common API (CAPI)

HCC provide five different file systems for embedded controllers to ensure that maximum performance can be reached and that resource limitations can be managed effectively. There are differences between each file system in the initialization, volume and partition control functions because of their differing capabilities. However all file and directory manipulation functions are entirely standard and 100% compatible across every file system.

The Common API (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.

File Operations

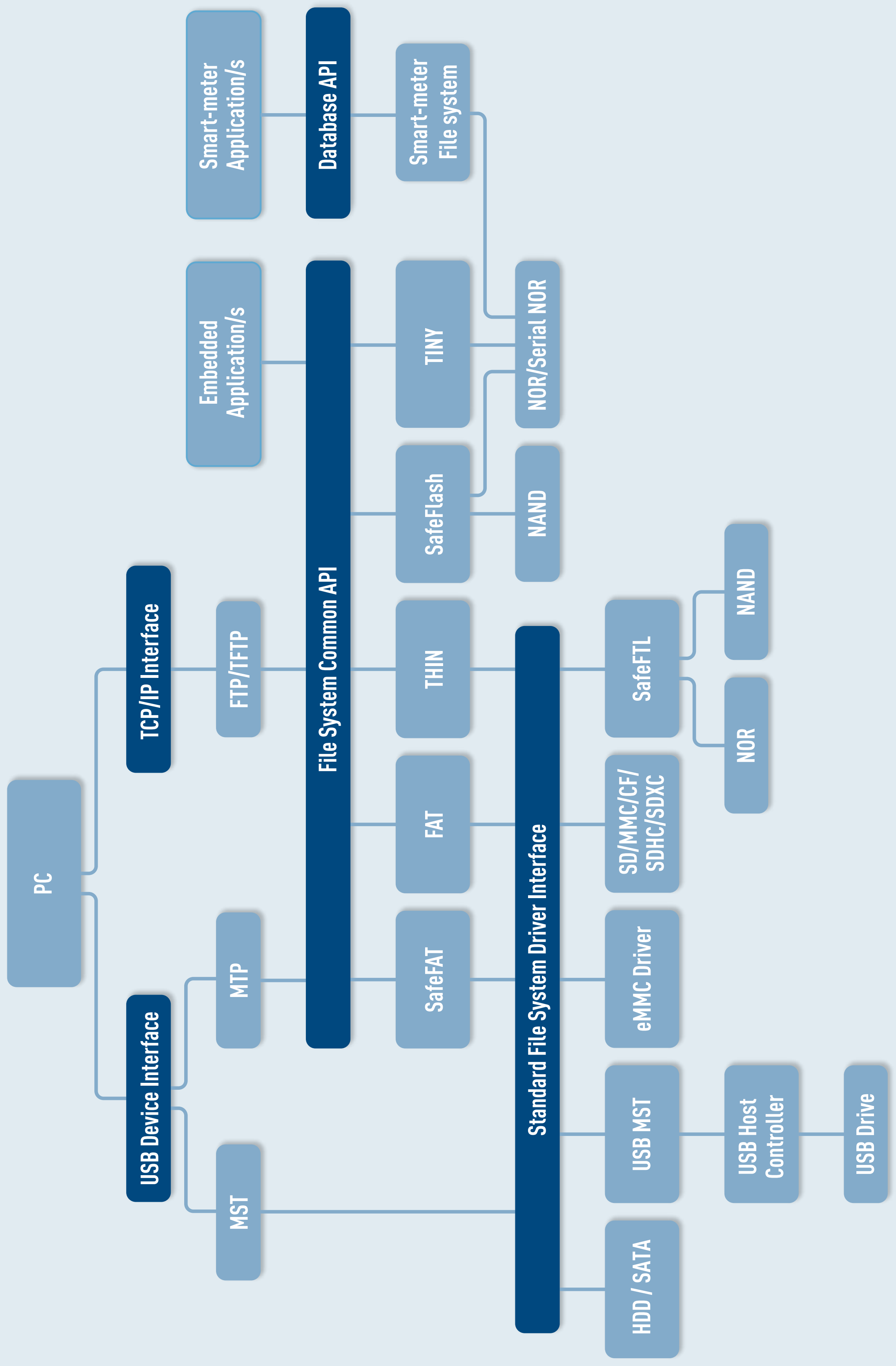
F_FILE *fopen(const char *filename, const char *mode);	Open a file
int f_close(F_FILE *filehandle)	Close a file
int f_flush(F_FILE *filehandle)	Flush a file to disk
long f_write(const void *buf, long size, long size_st, F_FILE *filehandle)	Write data to a file
long f_read(void *buf, long size, long size_st, 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 zero
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 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.

Appendix B:
Media Connectivity Schematic



HCC embedded

US sales office: Los Angeles, California • **Tel:** +1 310 265-1304
European sales office:
1133 Budapest, Váci út 76., Hungary • **Tel.:** +36 1 450 1302
info@hcc-embedded.com • sales@hcc-embedded.com • **www.hcc-embedded.com**