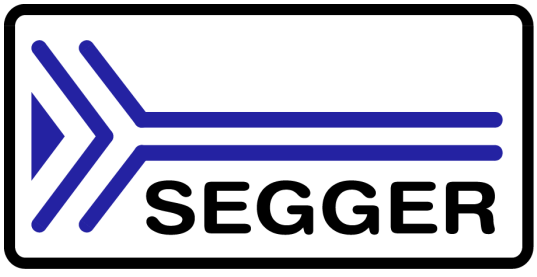
emFile

CPU independent
File System for
embedded applications

User & Reference Guide

Document: UM02001
Software version: 3.34
Revision: 1
Date: January 7, 2015



A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Disclaimer

Specifications written in this document are believed to be accurate, but are not guaranteed to be entirely free of error. The information in this manual is subject to change for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, SEGGER Microcontroller GmbH & Co. KG (SEGGER) assumes no responsibility for any errors or omissions. SEGGER makes and you receive no warranties or conditions, express, implied, statutory or in any communication with you. SEGGER specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

Copyright notice

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller GmbH & Co. KG

In den Weiden 11
D-40721 Hilden

Germany

Tel.+49 2103-2878-0
Fax.+49 2103-2878-28
E-mail: support@segger.com
Internet: http://www.segger.com

Manual versions

This manual describes the current software version. If any error occurs, inform us and we will try to assist you as soon as possible.
Contact us for further information on topics or routines not yet specified.

Print date: January 7, 2015

Software	Revision	Date	By	Description
3.34	1	141215	MD	Section "Logical drivers" * Added Sector write buffer driver. * Added RAID driver. Section "Device drivers -> NAND flash driver -> Additional physical layer functions" * Added FS_NAND_SPI_EnableReadCache() function. * Added FS_NAND_SPI_DisableReadCache() function. Section "Journaling (Add-on) -> Journaling API" * Added FS_JOURNAL_CreateEx() function. Section "Device drivers -> NAND flash driver -> SLC1 driver - FS_NAND_Driver -> Physical layer" * Added pfCopyPage function. Section "Device drivers -> NOR flash driver -> Sector map driver - FS_NOR_Driver -> Physical layer" * Added FS_NOR_PHY_SFDP physical layer. Section "API Functions -> Storage layer functions" * Added FS_STORAGE_SyncSectors() function.
3.34	0	140603	MD	Section "Logical drivers" * Added Sector size adapter driver. Section "API functions -> Operation on files" * Added FS_SetFileSize() function. Section "API functions -> File access functions" * Added FS_FOpenEx() function. Section "API functions -> Error handling functions -> FS_ErrorNo2Test()" * Added more error codes. Section "API functions -> File system extended functions" * Added FS_FreeSectors() function.
3.32	2	140128	MD	Section "API functions -> File system extended functions" * Added FS_GetVolumeInfoEx() function. Section "API functions -> File system extended functions" * Renamed the callback function type of FS_CheckDisk() to FS_CHECKDISK_ON_ERROR_CALLBACK Section "API functions -> File system extended functions -> FS_CheckDisk()" * Replaced magic numbers with symbolic defines. Section "API functions -> Operation on files" * Added FS_ModifyFileAttributes() function. Section "API functions -> Storage layer functions" * Added FS_STORAGE_GetCleanCnt() function. Section "Device drivers -> NOR flash driver -> Block map driver" * Added FS_NOR_BM_GetSectorInfo() function.
3.32	1	130722	MD	Section "API functions -> File system extended functions" * Added new return values for FS_CheckDisk(). Section "API functions -> File system configuration functions" * Renamed FS_ConfigFileBufferFlags() to FS_SetFileBufferFlags() Section "API functions -> Obsolete functions" * Made FS_ConfigUpdateDirOnWrite() obsolete.
3.32	0	130521	MD	Section "Device drivers -> SLC1 driver -> Hardware layer" * Added section "Hardware functions - SPI NAND flash" Section "Logical drivers" * Added Read-ahead driver. Section "API Functions -> Storage layer functions" * Added FS_STORAGE_FreeSectors() function. Section "Device drivers -> MMC/SD card driver" * Marked FS_MMC_CM_Driver4Atmel as deprecated.

Software	Revision	Date	By	Description
3.30	6	130208	MD	Section "Journalig (Add-on) -> Journaling API" * Added FS_JOURNAL_Disable() function. * Added FS_JOURNAL_Enable() function. * Changed the return types of FS_JOURNAL_Begin() and FS_JOURNAL_End() functions to "int".
3.30	5	121214	MD	Section "Device drivers -> NAND flash driver -> Additional physical layer functions" * Added FS_NAND_2048x8_EnableReadCache() function * Added FS_NAND_2048x8_DisableReadCache() function
3.30	4	121119	MD	Section "Device drivers -> MMC/SD card driver -> Additional driver functions" * Added FS_MMC_CM_GetCardId() function.
3.30	3	121107	MD	Section "API functions -> Operation on files" * Added FS_WipeFile() function
3.30	2	121019	MD	Section "API functions -> Extended functions" * Added FS_CreateMBR() function * Added FS_GetPartitionInfo() function Section "API functions -> File system configuration functions" * Added FS_SetFileWriteModeEx() function. Section "Device drivers -> NOR flash driver -> Configuring the driver -> Configuration API" * Added FS_NOR_CFI_SetAddrGap() function.
3.30	1	120903	MD	Section "Device Drivers -> NAND flash driver -> SLC1 dirver -> Physical layer" * Added (*pfConfigureECC)() function. Section "Device drivers -> NAND flash driver -> Universal NAND driver -> Configuring the driver" * Added FS_NAND_ECC_HW_4BIT ECC hook. * Renamed FS_NAND_ECC_NULL to FS_NAND_ECC_HW_NULL. * Renamed FS_NAND_ECC_1BIT to FS_NAND_ECC_SW_1BIT. Section "Logical drivers -> Encryption driver" * Added DES algorithm. Added "Encryption Add-On" chapter.
3.30	0	120803	MD	Chapter "Logical drivers" * Added encryption logical driver
3.28	1	120702	MD	Section "API functions -> File access functions" * Renamed FS_FFlush() to FS_SyncFile()

Software	Revision	Date	By	Description
3.28	0	120619	MD	Updated preface and about information. Merged the description of FS_FAT_CheckDisk() and FS_EFS_CheckDisk() functions to FS_CheckDisk(). Section "API functions -> Extended functions" * Ordered the functions alphabetically * Added FS_GetVolumeFreeSpaceKB() function * Added FS_GetVolumeSizeKB() function * Added FS_CheckDisk() function * Merged the description of FS_FAT_CheckDisk() and FS_EFS_CheckDisk() functions to FS_CheckDisk(). * Added FS_CheckDisk_ErrCode2Text() function * Merged the description of FS_FAT_CheckDisk_ErrCode2Text() and FS_EFS_CheckDisk_ErrCode2Text() functions to FS_CheckDisk_ErrCode2Text(). * Added FS_ON_CHECK_DISK_ERROR_CALLBACK typedef * Added FS_SetMemAccessCallback() function * Added FS_MEMORY_IS_ACCESSIBLE_CALLBACK typedef * Added FS_BUSY_LED_CALLBACK typedef * Renamed FS_QUERY_F_TYPE to FS_ON_CHECK_DISK_ERROR_CALLBACK. Section "API functions -> File system configuration functions" * Ordered the functions alphabetically * Added FS_FAT_ConfigUseFSInfoSector() function * Added FS_FAT_ConfigMaintainFATCopy() function Section "API functions -> File system control functions" * Ordered the functions alphabetically Section "API functions -> File access functions" * Ordered the functions alphabetically * Added FS_FFlush() function. Section "API functions -> Formatting a medium" * Ordered the functions alphabetically Section "API functions -> Error-handling functions" * Ordered the functions alphabetically Added chapter "Logical drivers" Section "Configuration of emFile -> Compile time configuration -> General file system configuration" * Added FS_SUPPORT_CHECK_MEMORY define Section "Optimizing performance - Caching and buffering -> Cache API functions" * Added FS_CACHE_Invalidate() function * Added FS_CACHE_SetAssocLevel() function Section "Device Drivers -> SLC1 driver" * Renamed FS_NAND_SetOnFatalErrorCB() to FS_NAND_SetOnFatalErrorCallback() * Renamed FS_NAND_ON_FATAL_ERROR_CB to FS_NAND_ON_FATAL_ERROR_CALLBACK Section "Device Drivers -> Universal driver" * Renamed FS_NAND_UNI_SetOnFatalErrorCB() to FS_NAND_UNI_SetOnFatalErrorCallback() Section "OS integration -> OS layer API functions" * Added FS_X_OS_Wait() function * Added FS_X_OS_Signal() function Section "API functions -> Directory functions" * Added FS_CreateDir() function. Section "API functions -> Storage layer functions" * Added FS_STORAGE_RefreshSectors() function
3.26	3	120322	MD	Section "API functions -> Storage layer functions" * Added FS_STORAGE_Clean() function * Added FS_STORAGE_CleanOne() function
3.26	2	111205	MD	Section "Device drivers -> NAND flash driver" * Subsection "Additional information" added. * Subsection "Additional physical layer functions" added.
3.26	1	111104	MD	Chapter "API functions" * Function FS_CopyFileEx() added. Section "Journaling Add-On -> Performance and resource usage" * Corrected the computation of dynamic RAM usage.

Software	Revision	Date	By	Description
3.26	0	111005	MD	Section "Device drivers -> NAND flash driver" * Created section "SLC1 driver - FS_NAND_Driver" from section "NAND flash driver" * Section "Universal driver - FS_NAND_UNI_Driver" added Section "Device drivers -> NOR flash driver" * Created section "Sector map - FS_NOR_Driver" from section section "NOR flash driver" * Section "Block map - FS_NOR_BM_Driver" added
3.24	5	110729	MD	Section "Device drivers -> NAND driver" * Added description for the return values of fatal error callback function
3.24	4	110705	MD	Section "Device drivers -> NAND driver" * Function "FS_NAND_SetNumWorkBlocks()" added Chapter "API functions" * Function "FS_Lock()" added * Function "FS_Unlock()" added * Function "FS_LockVolume()" added * Function "FS_UnlockVolume()" added
3.24	3	110318	MD	Section "Journaling (Add on)->Configuration" * Added "Journaling and write caching" section
3.24	2	110209	MD	Chapter "API functions" * Added FS_GetMaxSectorSize() description. Chapter "Device drivers -> NOR flash driver -> Performance and resource usage" * Corrected the performace values Chapter "Journaling (Add on)" * Added "FAQs" section Chapter "Device drivers -> NAND driver" * Function "FS_NAND_SetOnFatalErrorCB()" added
3.24	1	110113	MD	Chapter "Device drivers -> NAND driver" * Section "Partial writes" added.
3.24	0	101208	MD	Chapter "API functions" * Corrected the description of "FS_AssignMemory()" * Corrected the link to "FS_TimeStampToFileTime()" * Documented the return value of "FS_Sync()" Chapter "Device drivers -> MMC/SD card driver" * Added description for MMC cards version 4.x Chapter "Performance and resource usage" * Moved the pervormace measurement into the driver chapters Chapter "Device drivers -> NOR driver" * Added description for the "FS_NOR_SetSectorSize()" Chapter "Device drivers -> NAND driver" * Added description for the "FS_NAND_SetMaxEraseCntDiff()" Chapter "Device drivers -> NAND driver -> Hardware layer" * Removed the "FS_NAND_HW_X_Delayus() function"

Software	Revision	Date	By	Description

3.22	1	101001	MD	Chapter "Journaling"
				  * Corrected the prototypes of functions.
				Chapter "API functions -> File system control functions"
				  * FS_SetAutoMount prototype corrected.
				Chapter "API functions -> File access functions"
				  * FS_Read prototype corrected.
				Chapter "Device drivers -> NOR flash driver
				  -> Resource usage -> Runtime (dynamic) RAM usage"
				  * Simplified the forumula.
				  * Added table showing the RAM usage.
				Chapter "API functions"
				  * Function "FS_AddOnExitHandler()" added.
				  * Function "FS_EFS_CheckDisk()" added.
				  * Function "FS_EFS_CheckDisk_ErrorCode2Text()" added.
				Chapter "Introduction to emFile -> Basic concepts"
				  * Section "Fail safety" added
				  * Section "Wear leveling" added
				  * Section "Implemenation notes" added
				Chapter "Device drivers -> MultiMedia and SD card driver
				  -> Hardware functions - Card mode"
				  * Revised the description of all functions
				Chapter "Device drivers -> NAND flash driver
				  -> Fail-safe operation"
				  * Diagram and explanation of power loss added

3.22	0	100708	AG	Chapter "Running emFile on target hardware"
				  * Section "Adjusting the RAM usage" updated.
				Chapter "API functions"
				  * Function "FS_Mount()" updated.
				  * Function "FS_Sync()" added.
				  * Structure "FS_FORMAT_INFO" description updated.
				  * Function "FS_ConfigFileBufferDefault()" added.
				  * Function "FS_ConfigFileBufferFlags()" added.
				  * Function "FS_SetFileWriteMode()" added.
				Chapter "Device drivers"
				  * Section "NAND flash driver" updated.
				  * Section "WinDrive driver" updated/corrected.
				Chapter "Performance & resource usage"
				  * Section "Memory footprint" updated.
				Chapter "Journaling (Add-on)"
				  * Section "Resource usage" added.
				Chapter "Device drivers"
				  * Section "NOR flash driver",
				  subsection "Resource usage" added.
				Chapter "Porting emFile 2.x to 3.x"
				  * Section "Configuration differences" updated.
				Chapter "Configuration of emFile"
				  * Section "Compile time configuration" updated.

3.20	2	100326	AG	Chapter "Device drivers -> NOR flash driver
				  -> configuring the driver"
				  * Section "Configuration API" added.
				  * Section "Sample configurations" added.

3.20	1	091130	AG	Chapter "API functions" * Function "FS_DeInit()" added.

3.14	0	081215	SK/ SR	Chapter "API functions":
				  * "Cache functions removed.
				Chapter "Optimizing performance - Caching and buffering" added.
				Chapter "Introduction to emFile":
				  * Basic concepts updated.
				Chapter "Performance and resource usage"
				  * RAM requirements added.

3.12	3	080710	SR	Chapter "Performance and Resource Usage":
				  * Divided Memory requirements into different sections.
				Chapter "API functions":
				  * Changed Prototype of FS_Mount.

Software	Revision	Date	By	Description
3.12	2	080605	SR	Chapter "Configuration of emFile": * All configuration samples updated: * Added FS_AssignMemory. * Removed non existing marco: * FS_FAT_OPTIMIZE_SEQ_CLUSTERS * Added FS_DRIVER_ALIGNMENT macro.
3.12	1	080505	SK	Chapter "Introduction": * emFile structure updated. Chapter "Journaling (Add-on)": * FAQ added.
3.12	0	080424	SK	Chapter "Configuration of emFile": * FS_FAT_FWRITE_UPDATE_DIR removed. * FS_EFS_FWRITE_UPDATE_DIR removed. Chapter "API functions": Chapter "Device driver": MMC: * Section "Configuration" updated. * FS_MMC_CM_Allow4bitMode() added. NOR: * Serial NOR flash hardware functions added. Chapter "Journaling (Add-on)" added.
3.10	2	071022	SR	Chapter "Configuration of emFile": * Updated runtime configuration. * Updated Compiletime configuration. Chapter "API functions": * Added new functions: FS_AssignMemory, FS_SetMemHandler, FS_SetMaxSectorSize() FS_DeInit(). * Updated function description: FS_Mount(). Chapter "OS integration": * Added new function FS_X_OS_DeInit().
3.10	1	071008	SK	Chapter "Device driver": * Typos removed.
3.10	0	070927	SK	Chapter "API functions": * Storage layer functions added. Chapter "Running emFile on target hardware": * Structure/Directory names updated. Chapter "Device drivers": * Structure changed * Subsection "Resource usage" added to every driver section. * Section "NAND flash driver" updated and enhanced. * Section "NOR flash driver" updated and enhanced. * Section "Multimedia & SD card driver" enhanced. * Graphics updated. * Subsection Troubleshooting added. * Section "DataFlash driver" removed. The DataFlash driver is now integrated in the NAND driver. Chapter "Performance and resource usage": * Section "Memory footprint" updated.
3.08	5	070719	SK	Chapter "Device drivers": * NAND: Pin description updated. * NAND: Illustrations added. * NOR: Illustrations added.
3.08	4	070716	SK	Chapter "Introduction": * emFile structure picture changed. * Layer description updated.
3.08	3	070703	SK	Chapter "API functions": * FS_InitStorage() updated. * FS_ReadSector() added. * FS_WriteSector() added. * FS_GetDeviceInfo() added. Chapter "Index" * Index updated.
3.08	2	070703	SK	Chapter "Device drivers": * "NAND flash driver" section enhanced.

Software	Revision	Date	By	Description
3.08	1	070618	SK	Chapter "API functions": * FS_UnmountLL added. * FS_GetVolumeStatus() added. * FS_InitStorage() added. Chapter "Porting emFile 2.x to 3.x" chapter.
3.08	0	070618	SK	Chapter "Introduction": * Section "Development environment" added. Chapter "API functions" updated. * FS_Mount() added. * FS_SetAutoMount() added. * FS_UnmountForced() added.
3.04	0	070427	SK	Various improvements. Chapter "Running emFile on target hardware" updated. * Structural changes. * Section "Adjusting the RAM usage" added. Chapter "API functions" updated. * Samples updated. Chapter "Device driver" updated. * Generic flash driver renamed to NOR flash driver. - FS_FLASH_* replaced with FS_NOR_*. - NOR - additional driver functions added. * DataFlash driver added.
3.02	0	070405	SK	Chapter "Running emFile on target hardware" updated. * Some smaller structural changes. * Section "Step 3: Add device driver" simplified. * Section "Step 4: Implement hardware routines" simplified. * Section "Troubleshooting" moved to chapter debugging. Chapter "API functions": * Section "File system configuration functions" added. - FS_AddDevice() moved into this section. - FS_AddPhysDevice() added. - FS_LOGVOL_Create() added. - FS_LOGVOL_AddDevice() added. Chapter "Device drivers": * Section "NAND": - FS_NAND_SetBlockRange() added. Chapter "Configuration of emFile": * Section "Compile-time configuration" - "Miscellaneous configuration" - "FS_NO_CLIB" default value corrected. Chapter "Debugging" - "FS_X_Log()", "FS_X_Warn()", "FS_X_ErrorOut()" : function description enhanced. Chapter "OS Support" updated.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler)
- The C programming language
- The target processor
- DOS command line

If you feel that your knowledge of C is not sufficient, we recommend The C Programming Language by Kernighan and Richie (ISBN 0-13-1103628), which describes the standard in C-programming and, in newer editions, also covers the ANSI C standard.

How to use this manual

This manual explains all the functions and macros that the product offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Keyword`	Text that you enter at the command-prompt or that appears on the display (that is system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections.

Table 1.1:

SEGGER Microcontroller GmbH & Co. KG develops and distributes software development tools and ANSI C software components (middleware) for embedded systems in several industries such as telecom, medical technology, consumer electronics, automotive industry and industrial automation.

SEGGER's intention is to cut software development time for embedded applications by offering compact flexible and easy to use middleware, allowing developers to concentrate on their application.

Our most popular products are emWin, a universal graphic software package for embedded applications, and embOS, a small yet efficient real-time kernel. emWin, written entirely in ANSI C, can easily be used on any CPU and most any display. It is complemented by the available PC tools: Bitmap Converter, Font Converter, Simulator and Viewer. embOS supports most 8/16/32-bit CPUs. Its small memory footprint makes it suitable for single-chip applications.

Apart from its main focus on software tools, SEGGER develops and produces programming tools for flash micro controllers, as well as J-Link, a JTAG emulator to assist in development, debugging and production, which has rapidly become the industry standard for debug access to ARM cores.

Corporate Office:
http://www.segger.com

United States Office:
http://www.segger-us.com

EMBEDDED SOFTWARE (Middleware)

emWin
Graphics software and GUI
emWin is designed to provide an efficient, processor- and display controller-independent graphical user interface (GUI) for any application that operates with a graphical display.

embOS
Real Time Operating System
embOS is an RTOS designed to offer the benefits of a complete multitasking system for hard real time applications with minimal resources.

embOS/IP
TCP/IP stack
embOS/IP a high-performance TCP/IP stack that has been optimized for speed, versatility and a small memory footprint.

emFile
File system
emFile is an embedded file system with FAT12, FAT16 and FAT32 support. Various Device drivers, e.g. for NAND and NOR flashes, SD/MMC and CompactFlash cards, are available.

USB-Stack
USB device/host stack
A USB stack designed to work on any embedded system with a USB controller. Bulk communication and most standard device classes are supported.

SEGGER TOOLS

Flasher
Flash programmer
Flash Programming tool primarily for micro controllers.

J-Link
JTAG emulator for ARM cores
USB driven JTAG interface for ARM cores.

J-Trace
JTAG emulator with trace
USB driven JTAG interface for ARM cores with Trace memory. supporting the ARM ETM (Embedded Trace Macrocell).

J-Link / J-Trace Related Software
Add-on software to be used with SEGGER's industry standard JTAG emulator, this includes flash programming software and flash breakpoints.

Table of Contents

Chapter 1

Introduction to emFile

1.1 What is emFile

emFile is a file system that can be used on any media for which you can provide basic hardware access functions.

emFile is a high-performance library that has been optimized for speed, versatility and memory footprint.

1.2 Features

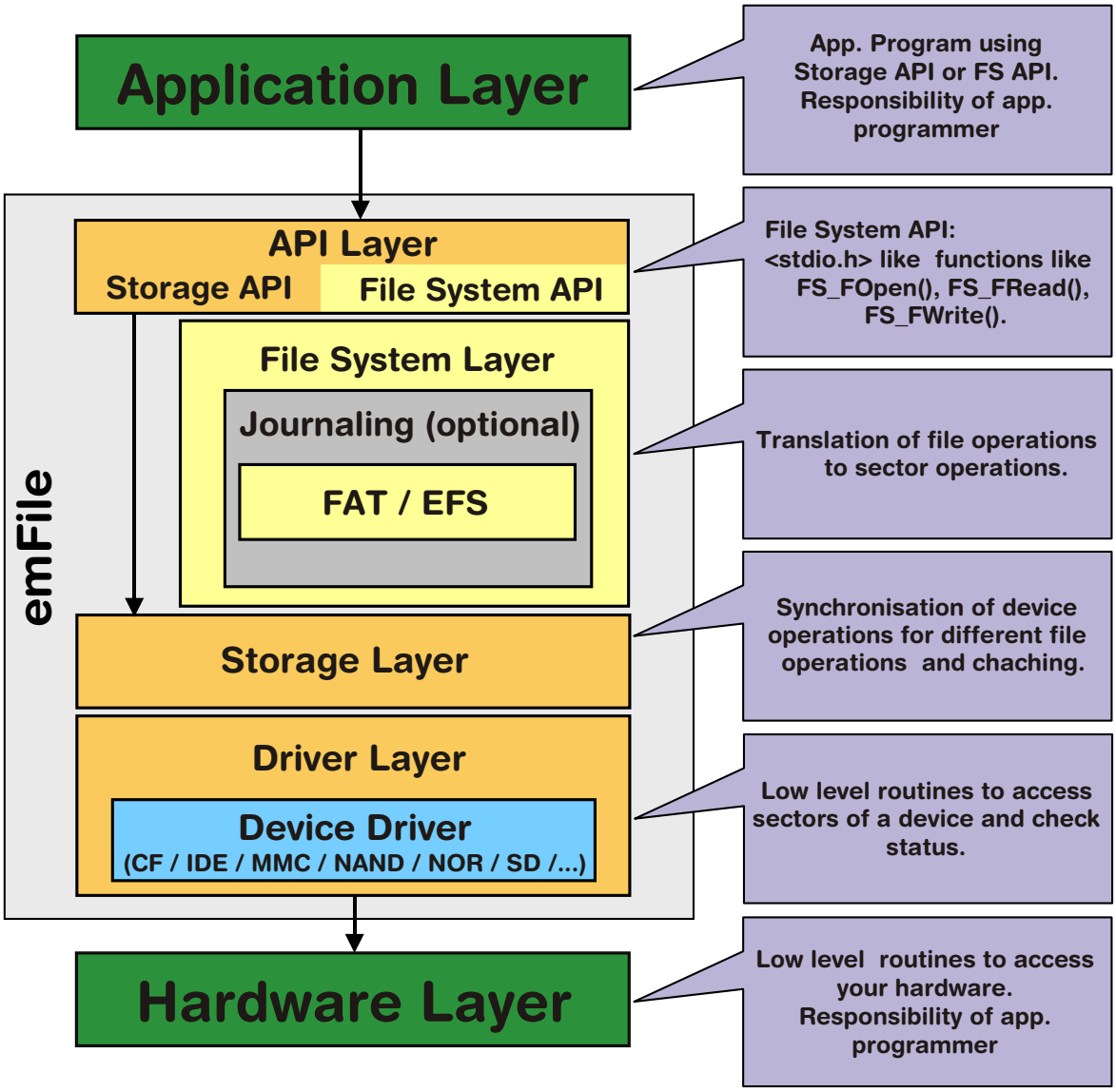
emFile is written in ANSI C and can be used on virtually any CPU.
Some features of emFile:

- MS DOS/MS Windows-compatible FAT12, FAT16 and FAT32 support.
- An optional module that handles long file names of FAT media.
- Multiple device driver support. You can use different device drivers with emFile, which allows you to access different types of hardware with the file system at the same time.
- MultiMedia support. A device driver allows you to access different media at the same time.
- OS support. emFile can be easily integrated into any OS. This allows using emFile in a multi-threaded environment.
- ANSI C `stdio.h`-like API for user applications. An application using the standard C I/O library can easily be ported to use emFile.
- Very simple device driver structure. emFile device drivers need only basic functions for reading and writing blocks. There is a template included. See `/Sample/Driver/DriverTemplate/Driver_Template.c` for more details.
- An optional device driver for NAND flash devices, which can be easily used with any kind of NAND flashes.
- An optional device driver for MultiMedia & SD cards using SPI mode or card mode that can be easily integrated.
- An optional IDE driver, which is also suitable for CompactFlash using either “True IDE” or “Memory Mapped” mode.
- An optional NOR flash (EEPROM) driver that handles different flash sector sizes.
- An optional proprietary file system (EFS) with native long file name support.

1.3 Basic concepts

1.3.1 emFile structure

emFile is organized in different layers, illustrated in the diagram below. A short description of each layer's functionality follows below.



API Layer

The API Layer is the interface between emFile and the user application. It is divided in two parts Storage API and File System API. The File System API declares file functions in ANSI C standard I/O style, such as `FS_FOpen()`, `FS_FWrite()` etc. The API Layer transfers any calls to these functions to the File System Layer. Currently the FAT file system or an optional file system, called EFS, are available for emFile. Right now they cannot be used simultaneously. The Storage API declares the functions which are required to initialize and access a storage medium. The Storage API allows sector read and write operations. The API Layer transfers these calls to the Storage Layer. The Storage API is optimized for applications which do not require file system functionality like file and directory handling. A typical application which uses the Storage API could be a USB mass storage device, where data has to be stored on a medium, but all file system functionality is handled by the host PC.

File System Layer

The file system layer translates file operations to logical block (sector) operations. After such a translation, the file system calls the logical block layer and specifies the corresponding device driver for a device.

Storage Layer

The main purpose of the Storage Layer is to synchronize accesses to a device driver. Furthermore, it provides a simple interface for the File System API. The Storage Layer calls a device driver to perform a block operation. It also contains the cache mechanism.

Driver Layer

Device drivers are low-level routines that are used to access sectors of the device and to check status. It is hardware independent but depends on the storage medium.

Hardware Layer

These layer contains the low-level routines to access your hardware. These routines simply read and store fixed length sectors. The structure of the device driver is simple in order to allow easy integration of your own hardware.

1.3.2 Choice of file system type: FAT vs. EFS

Within emFile, there is a choice among two different file systems. The first, the FAT file system, is divided into three different sub types, FAT12, FAT16 and FAT32. While the other, EFS, is a proprietary file system developed by Segger. Choosing a suitable file system will depend on the environment in which the end application is to operate.

The FAT file system was developed by Microsoft to manage file segments, locate available clusters and reassemble file for use. Released in 1976, the first version of the FAT file system was FAT12, which is no longer widely used. It was created for extremely small storage devices. (The early version of FAT12 did not support managing directories).

FAT16 is good for use on multiple operating systems because it is supported by all versions of Microsoft Windows, including DOS, OS/2 and Linux. The newest version, FAT32, improves upon the FAT16 file system by utilizing a partition/disk much more efficiently. It is supported by Microsoft Windows 98/ME/2000/XP/2003 and Vista and as well on Linux based systems.

The EFS file system has been added to emFile as an alternative to the FAT file system. EFS has been designed for embedded devices. This file system reduces fragmentation of the data by utilizing drive space more efficiently, while still offering faster access to embedded storage devices. Another benefit of EFS is that there are no issues concerning long file name (LFN) support. The FAT file system was not designed for long file name support, limiting names to twelve characters (8.3). LFN support may be added to any of the FAT file systems, but there are legal issues that must be settled with Microsoft before end applications make use of this feature. Long file names are inherent to this proprietary file system relieving it of any legal issues.

1.3.3 Fail safety

Fail safety is the feature of emFile that ensures the consistency of data in case of unexpected loss of power during a write access to a storage medium. emFile will be fail-safe only when both the file system (FAT/EFS) and the device driver are fail-safe. The journaling add-on of emFile to makes the FAT/EFS file systems fail-safe. The device drivers of emFile are all from design fail-safe. You can find detailed information about how the fail-safety works on chapter *Journaling (Add-on)* on page 505 and of the description of individual device drivers.

1.3.4 Wear leveling

This is a feature of the NAND and NOR flash device drivers that increase the lifetime of a storage medium by ensuring that all the storage blocks are equally well used. The flash storage memories have a limited number of program/erase cycles, typically around 100000. The manufacturers do not guarantee that the storage device will work properly if this limit is exceeded. The wear leveling logic implemented in the device drivers tries to keep the number of program-erase cycles of a storage block as low as possible. You can find additional information in the description of the respective device drivers.

1.4 Implementation notes

1.4.1 File system configuration

The file system is designed to be configurable at runtime. This has various advantages. Most of the configuration is done automatically; the linker links in only code that is required. This concept allows to putting the file system in a library. The file system need not to be recompiled when the configuration changes, e.g. a different driver is used. Compile time configuration is kept to a minimum, primarily to select the level of multitasking support and the level of debug information. For detailed information about configuration of emFile, refer to *Configuration of emFile* on page 471.

1.4.2 Runtime memory requirements

Because the configuration is selected at runtime the amount of memory required is not known at compile-time. For this reason a mechanism for runtime memory assignment is required. Runtime memory is typically allocated when required during the initialization and in most embedded systems never freed.

1.4.3 Initializing the file system

The first thing that needs to be done after the system start-up and before any file system function can be used, is to call the function `FS_Init()`. This routine initializes the internals of the file system.While initializing the file system, you have to add your target device to the file system. The function `FS_X_AddDevices()` adds and initializes the device.

```
FS_Init()
 └FS_X_AddDevices()
   ├FS_AssignMemory()
   ├FS_AddDevice()
   └Optional: Other configuration functions
```


1.5 Development environment (compiler)

The CPU used is of no importance; only an ANSI-compliant C compiler complying with at least one of the following international standard is required:

- ISO/IEC/ANSI 9899:1990 (C90) with support for C++ style comments (//)
- ISO/IEC 9899:1999 (C99)
- ISO/IEC 14882:1998 (C++)

If your compiler has some limitations, let us know and we will inform you if these will be a problem when compiling the software. Any compiler for 16/32/64-bit CPUs or DSPs that we know of can be used; most 8-bit compilers can be used as well.

A C++ compiler is not required, but can be used. The application program can therefore also be programmed in C++ if desired.

Chapter 2

Getting started

This chapter provides an introduction to using emFile. It explains how to use the Windows sample, which is an easy way to get a first project with emFile up and running.

2.1 Installation

emFile is shipped as a CD-ROM or as a `.zip` file in electronic form.
In order to install it, proceed as follows:

- If you received a CD, copy the entire contents to your hard drive into any folder of your choice. When copying, keep all files in their respective sub- directories. Make sure the files are not read-only after copying.
- If you received a .zip file, extract it to any folder of your choice, preserving the directory structure of the .zip file.

2.2 Using the Windows sample

If you have MS Visual C++ 6.00 or any later version available, you will be able to work with a Windows sample project using emFile. Even if you do not have the Microsoft compiler, you should read this chapter in order to understand how an application can use emFile.

2.2.1 Building the sample program

Open the workspace `FS_Start.dsw` with MS Visual Studio (for example double-clicking it). There is no further configuration necessary. You should be able to build the application without any error or warning message.

2.2.2 Stepping through the sample

The sample project uses the RAM disk driver for demonstration. The `main` function of the sample application `Start.c` calls the function `MainTask()`. `MainTask()` initializes the file system and executes some basic file system operations.

`The sample application Start.c` step-by-step:

1. `main.c` calls `MainTask()`,
2. `MainTask()` initializes and adds a device to emFile,
3. checks if volume is low- level formatted and formats if required,
4. checks if volume is high-level formatted and formats if required,
5. outputs the volume name,
6. calls `FS_GetFreeVolumeSpace()` and outputs the return value - the available free space of the RAM disk - to console window,
7. creates and opens a file test with write access (`File.txt`) on the device,
8. writes 4 bytes into the file and closes the file handle or outputs an error message,
9. calls `FS_GetFreeVolumeSpace()` and outputs the return value - the available free space of the RAM disk - again to console window,
10. outputs an quit message and runs into an endless loop.

The sample step-by-step

1. After starting the debugger by stepping into the application, your screen should look as the screenshot below. The `main` function calls `MainTask()`.

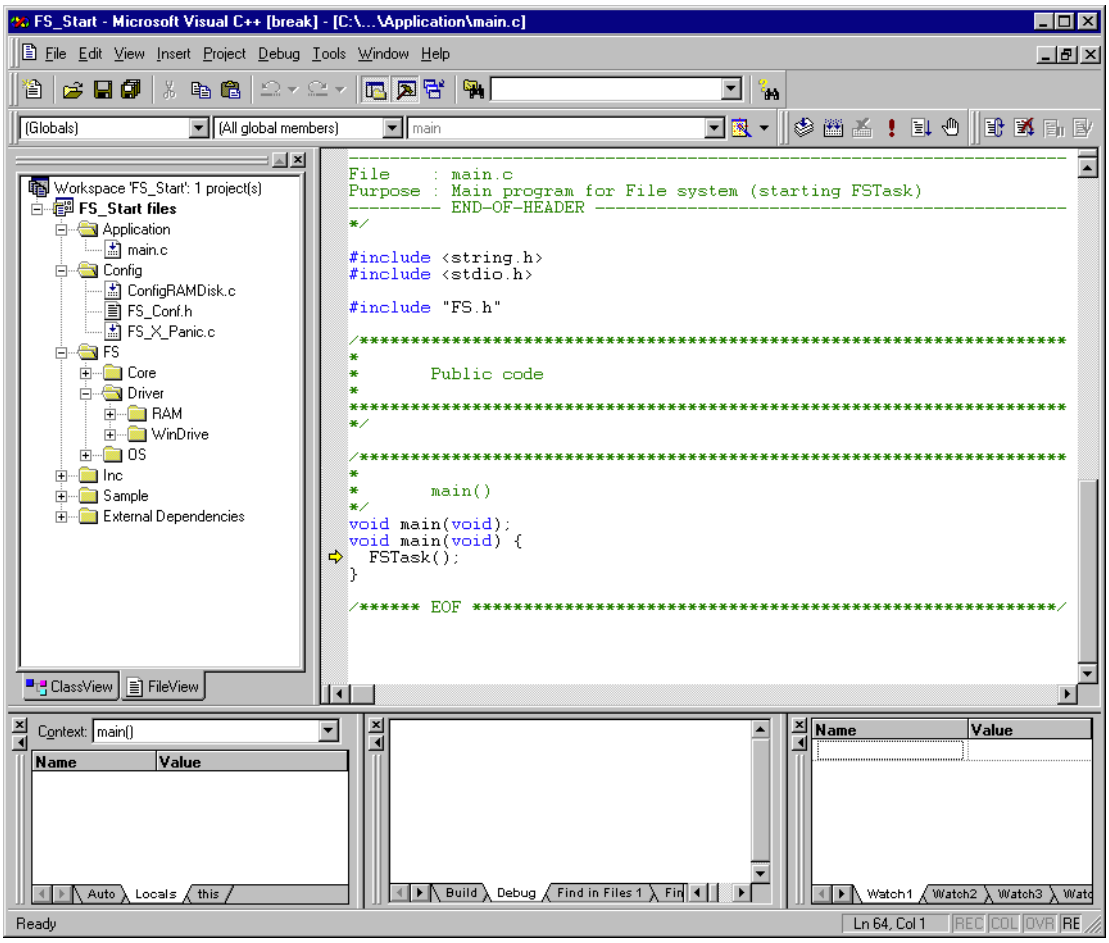


Figure 2.1: FS_Start project - main()

2. The first things called from `MainTask()` is the emFile function `FS_Init()`. This function initializes the file system and calls `FS_X_AddDevices()`. The function `FS_X_AddDevices()` is used to add and configure the used device drivers to the file system. In the example configuration only the RAM disk driver is added. `FS_Init()` must be called before using any other emFile function. You should step over this function.

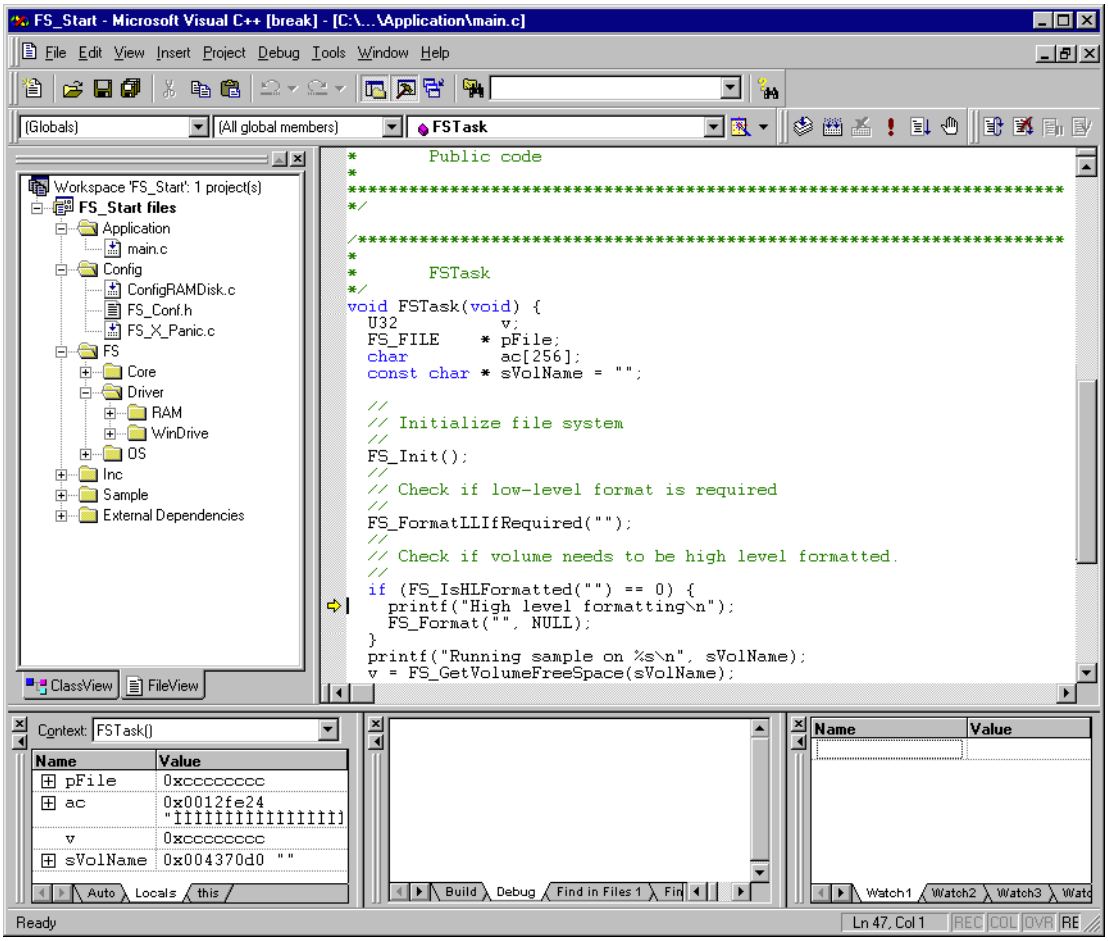


Figure 2.2: FS_Start project - MainTask()

3. If the initialization was successfully, `FS_FormatLLIfRequired()` is called. It checks if the volume is low-level formatted and formats the volume if it is required. You should step over this function.
4. Afterwards `FS_IsHLFormatted()` is called. It checks if the volume is high-level formatted and formats the volume if this is required. You should step over this function.
5. The volume name is printed in the console window.
6. The emFile function `FS_GetVolumeFreeSpace()` is called and the return value is written into the console window.

7. Afterwards, you should get to the emFile function call `FS_FOpen()`. This function creates a file named `file.txt` in the root directory of your RAM disk. Stepping over this function should return the address of an `FS_FILE` structure. In case of any error, it would return 0, indicating that the file could not be created.

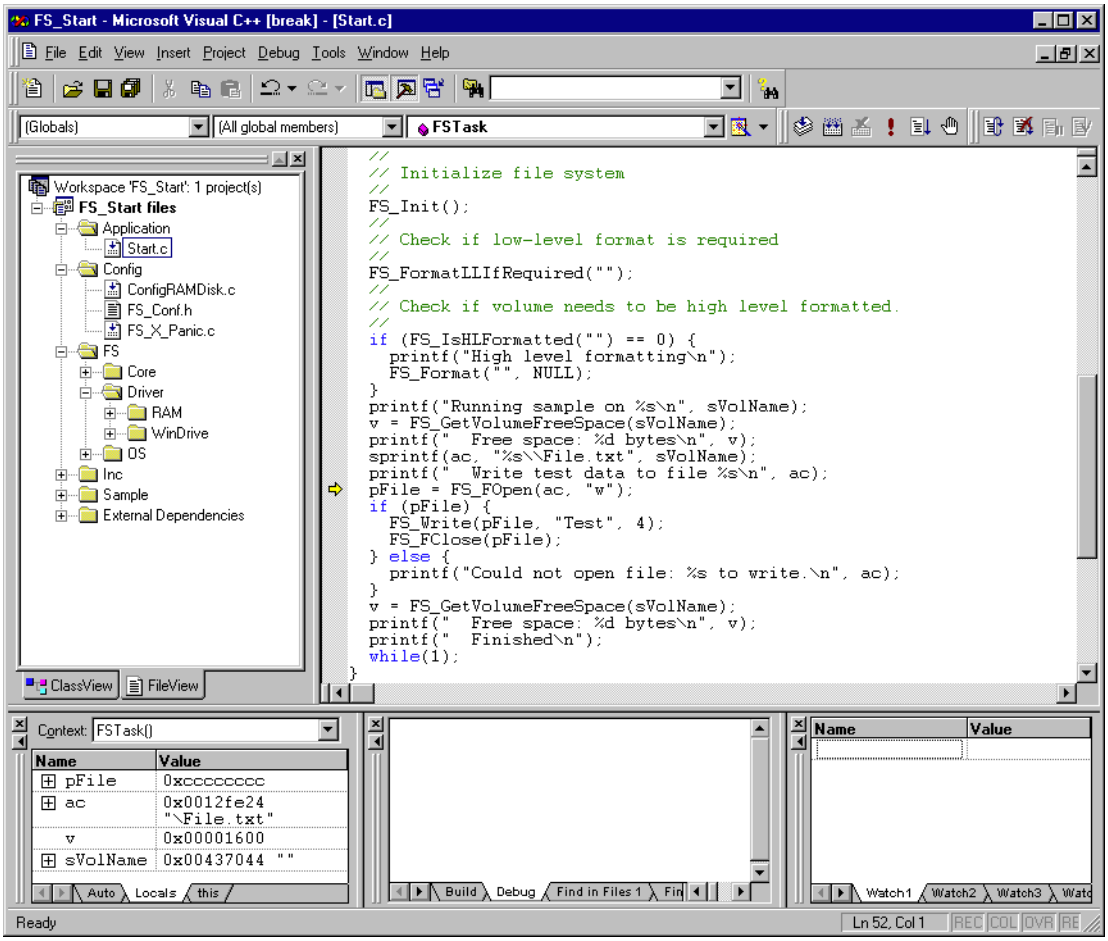


Figure 2.3: FS_Start project - MainTask()

8. If `FS_FOpen()` returns a valid pointer to an `FS_FILE` structure, the sample application will write a small ASCII string to this file by calling the emFile function `FS_FWrite()`. Step over this function. If a problem occurs, compare the return value of `FS_FWrite()` with the length of the ASCII string, which should be written. `FS_FWrite()` returns the number of elements which have been written.
   If no problem occurs the function emFile function `FS_FClose()` should be reached. `FS_FClose()` closes the file handle for `file.txt`. Step over this function.
9. Continue stepping over until you reach the call to the call of `FS_GetVolumeFreeSpace()`.The emFile function `FS_GetVolumeFreeSpace()` returns available free drive space in bytes. After you step over this function, the variable `v` should have a value greater than zero.
10. The return value is written in the console window.

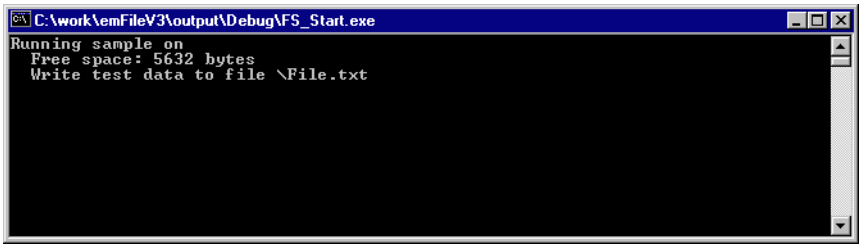


Figure 2.4: FS_Start project - console output

2.2.3 Further source code examples

Further source code examples which demonstrate directory operations and performance measuring are available. All emFile source code examples are located in the `\Sample\API\` directory under your emFile directory.

Chapter 3

Running emFile on target hardware

This chapter explains how to integrate and run emFile on your target hardware. It explains this process step-by-step.

Integrating emFile

The emFile default configuration contains a single device: a RAM disk. This should always be the first step to check the proper function of emFile with your target hardware.

We assume that you are familiar with the tools you have selected for your project (compiler, project manager, linker, etc.). You should therefore be able to add files, add directories to the include search path, and so on. It is also assumed that you are familiar with the OS that you will be using in your target system (if you are using one). In this document the IAR Embedded Workbench® IDE is used for all examples and screenshots, but every other ANSI C toolchain can also be used. It is also possible to use make files; in this case, when we say "add to the project", this translates into "add to the make file".

Procedure to follow

Integration of emFile is a relatively simple process, which consists of the following steps:

- Step 1: Creating a start project without emFile
- Step 2: Adding emFile to the start project
- Step 3: Adding the device driver
- Step 4: Activating the driver
- Step 5: Adjusting the RAM usage

3.1 Step 1: Creating a simple project without emFile

We recommend that you create a small “hello world” program for your system. That project should already use your OS and there should be a way to display text on a screen or serial port.

If you are using embOS, you can use the start project shipped with the OS for this purpose.

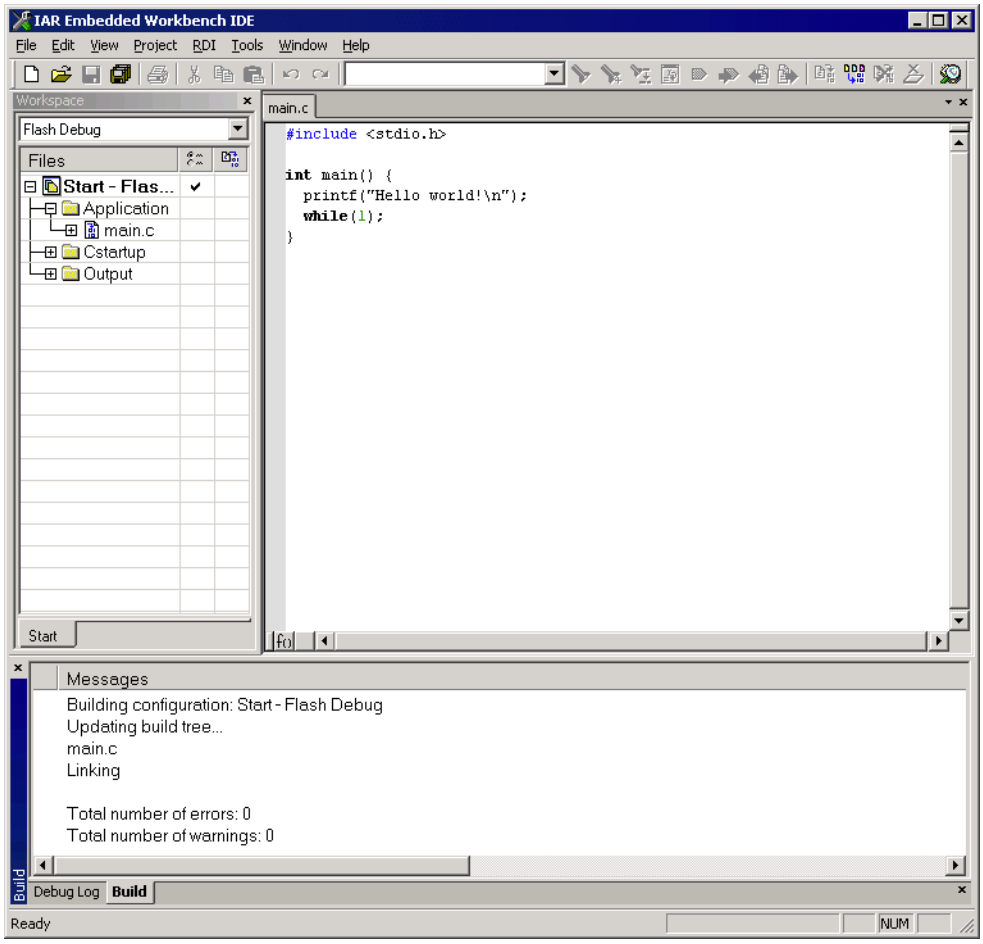


Figure 3.1: Start project

3.2 Step 2: Adding emFile to the start project

Add all source files in the following directories (and their subdirectories) to your project:

- `Application`
- `Config`
- `FS`
- `Sample\Driver\RAM`
- `Sample\OS\` (Optional, add if you use an RTOS. Add only the file compatible to the used operating system.)

It is recommended to keep the provided folder structure.

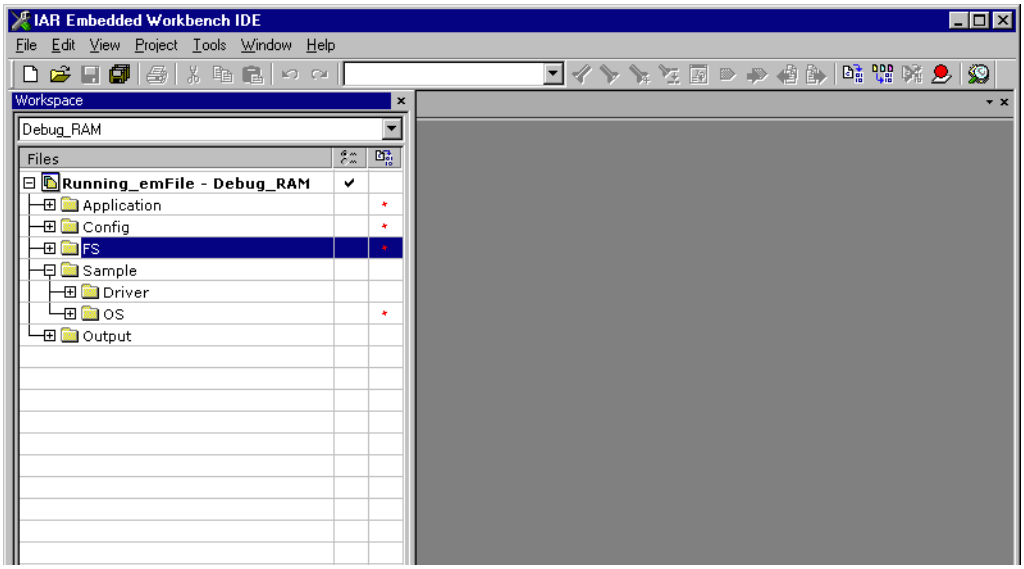


Figure 3.2: emFile project structure

Configuring the include path

The include path is the path in which the compiler looks for include files. In cases where the included files (typically header files, `.h`) do not reside in the same directory as the C file to compile, an include path needs to be set. In order to build the project with all added files, you will need to add the following directories to your include path:

- `Config`
- `FS\`

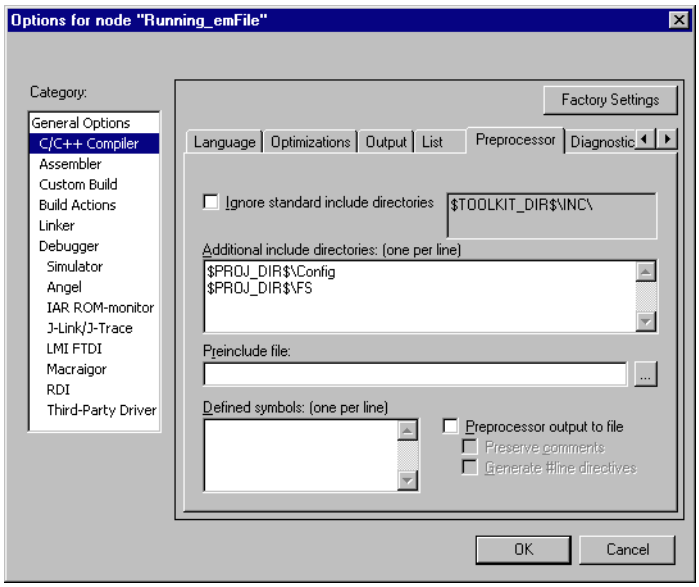


Figure 3.3: Configure the include path

Select the start application

For quick and easy testing of your emFile integration, start with the code found in the folder `Application`. Exclude all files in the `Application` folder of your project except the supplied `main.c` and `Start`.c.

The application performs the following steps:

1. `main.c` calls `MainTask()`,
2. `MainTask()` initializes and adds a device to emFile,
3. checks if volume is low- level formatted and formats if required,
4. checks if volume is high-level formatted and formats if required,
5. outputs the volume name,
6. calls `FS_GetFreeVolumeSpace()` and outputs the return value - the available total space of the RAM disk - to console window,
7. creates and opens a file test with write access (`File.txt`) on the device,
8. writes 4 bytes into the file and closes the file handle or outputs an error message,
9. calls `FS_GetFreeVolumeSpace()` and outputs the return value - the available free space of the RAM disk - again to console window,
10. outputs an quit message and runs into an endless loop.

Build the project and test it

Build the project. It should compile without errors and warnings. If you encounter any problem during the build process, check your include path and your project configuration settings. The start application should print out the storage space of the device twice, once before a file has been written to the device and once afterwards.

3.3 Step 3: Adding the device driver

To configure emFile with a device driver 2 things need to be done at the same time:

- Adding device driver source to project
- Adding hardware routines to project

Every recommended step is explained in the following sections. For example, the implementation of the MMC/SD driver is shown, but all steps should be easy to adapt on every other device driver implementation.

3.3.1 Adding the device driver source to project

Add the driver sources to the project and add the directory to the include path.

Example

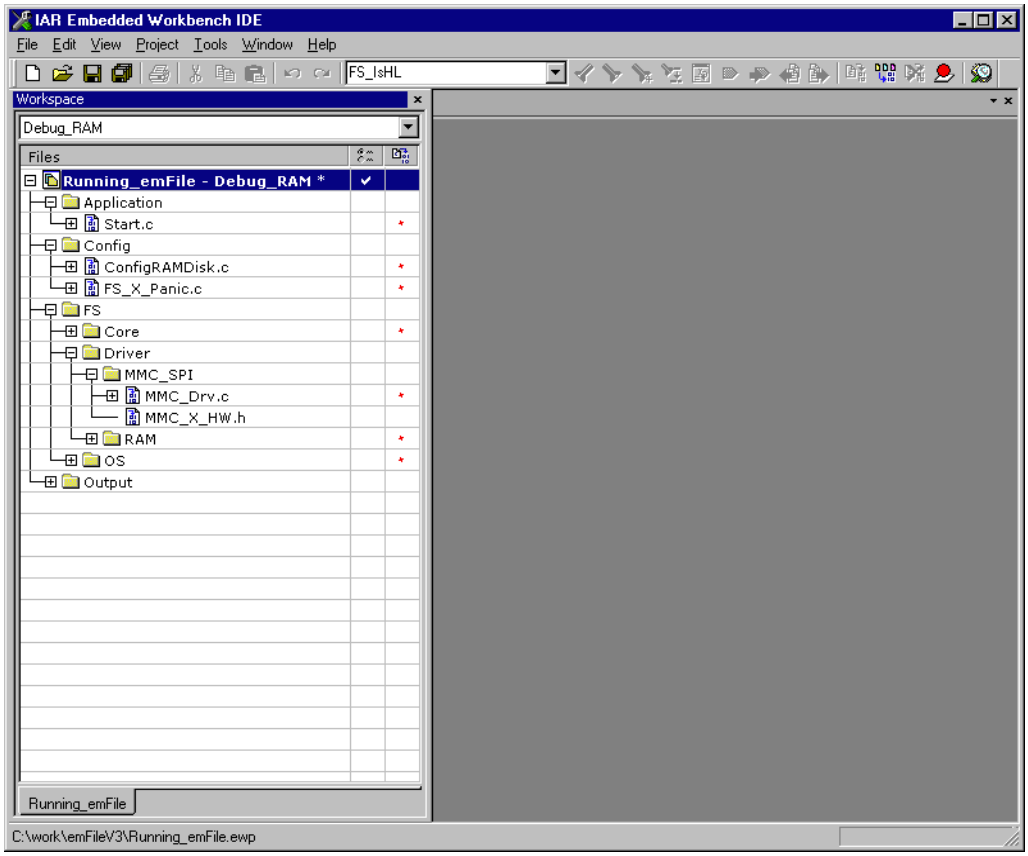


Figure 3.4: Add driver sources to project

Most drivers require additional hardware routines to work with the specific hardware. If your driver requires low-level I/O routines to access the hardware, you will have to provide them.

Drivers which require hardware routines are:

- NAND
- MMC/SD cards
- Compact flash / IDE

Drivers which not require hardware routines are:

- NOR flash
- RAM

Nearly all drivers have to be configured before they can be used. The runtime configuration functions which specify for example the memory addresses and the size of memory are located in the configuration file of the respective driver. All required configurations are explained in configuration section of the respective driver. If you use one of the drivers which do not require hardware routines skip the next section and refer to *Step 4: Activating the driver* on page 40.

3.3.2 Adding hardware routines to project

A template with empty function bodies and in most cases one ore more sample implementations are supplied for every driver which requires hardware routines. The easiest way to start is to use one of the ready-to-use samples. The ready-to-use samples can be found in the subfolders of `Sample\Driver\<DRIVER_DIR>\`. You should check the `Readme.txt` file located in the driver directory to see which samples are included. If there is one which is a good or close match for your hardware, it should be used. Otherwise, use the template to implement the hardware routines.

The template is a skeleton driver which contains empty implementations of the required functions and is the ideal base to start the implementation of hardware specific I/O routines.

What to do

Copy the compatible hardware function sample or the template into a subdirectory of your work directory and add it to your project. The template file is located in the `Sample\Driver\<DRIVER_DIR>\` directory; the example implementations are located in the respective directories. If you start the implementation of hardware routines with the hardware routine template, refer to *Device drivers* on page 213 for detailed information about the implementation of the driver specific hardware functions, else refer to section *Step 4: Activating the driver* on page 40.

Note: You cannot run and test the project with the new driver on your hardware as long as you have not added the proper configuration file for the driver to your project. Refer to section *Step 4: Activating the driver* on page 40 for more information about the activation of the driver with the configuration file.

3.4 Step 4: Activating the driver

After adding the driver source, and if required the hardware function implementation to the project, copy also the `Config<DRIVERNAME>.c` file (for example, `ConfigMMC_SPI.c` for the MMC/SD card driver using the SPI mode) into the `Config` directory of your emFile work directory. Add it afterwards to your project as show below.

Example

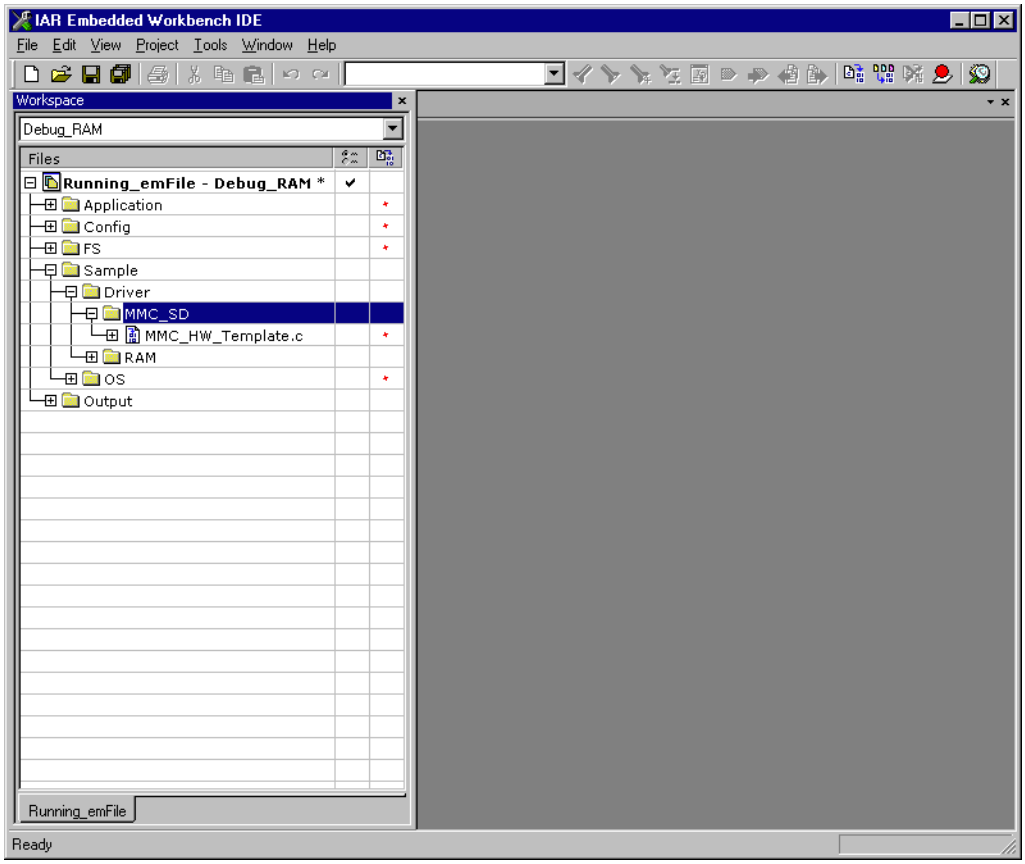


Figure 3.5: Adding template to your project

In the configuration files are all runtime configuration functions of the file system located. The configuration files include a start configuration which allows a quick and easy start with every driver. The most important function for the beginning is `FS_X_AddDevices()`. It activates and configures if required the driver. Driver which not require hardware routines has to configured before they can be used.

3.4.1 Modifying the runtime configuration

The example on the next page adds a single CFI compliant NOR flash chip with a 16-bit interface and a size of 256 Mbytes to the file system. The base address, the start address and the size of the NOR flash are defined using the macros `FLASH0_BASE_ADDR`, `FLASH0_START_ADDR` and `FLASH0_SIZE`. Normally, only the `Defines, configurable` section of the configuration files requires changes for typical embedded systems. The `Public code` section which includes the time and date functions and `FS_X_AddDevices()` does not require modifications in most systems.

Example

```
/*********************************************************************
*
*       Defines, configurable
*
*       This section is the only section which requires changes for
*       typical embedded systems using the NOR flash driver with a
*       single device.
*
**********************************************************************
*/
#define ALLOC_SIZE         0x10000        // Size of memory dedicated to the file
                                          // system. This value should be fine-tuned
                                          // according for your system.
#define FLASH0_BASE_ADDR   0x40000000     // Base addr of the NOR flash device to
                                          // be used as storage
#define FLASH0_START_ADDR  0x40000000     // Start addr of the first sector be used
                                          // as storage. If the entire chip is
                                          // used for file system, it is identical to
                                          // the base addr.
#define FLASH0_SIZE        0x200000       // Number of bytes to be used for storage

/*********************************************************************
*
*       Static data.
*
*       This section does not require modifications in most systems.
*
**********************************************************************
*/
static U32   _aMemBlock[ALLOC_SIZE / 4];    // Memory pool used for semi-dynamic
                                            // allocation in FS_AssignMemory().
/*********************************************************************
*
*       Public code
*
*       This section does not require modifications in most systems.
*
**********************************************************************
*/

/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*    It is supposed to add all devices, using primarily FS_AddDevice().
*/
void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  //  Add driver
  //
  FS_AddDevice(&FS_NOR_Driver);
  //
```



```
  //  Confgure the NOR flash interface
  //
  FS_NOR_SetPhyType(0, &FS_NOR_PHY_CFI_1x16);
  FS_NOR_Configure(0, FLASH0_BASE_ADDR, FLASH0_START_ADDR, FLASH0_SIZE);
  //
  // Configure a read buffer for the file data.
  //
  FS_ConfigFileBufferDefault(512, 0);
}
```

After the driver has been added, the configuration functions (in this example `FS_NOR_SetPhyType()` and `FS_NOR_Configure()`) should be called. Detailed information about the driver configuration can be found in the configuration section of the respective driver.

Refer to section *Runtime configuration* on page 472 for detailed information about the other runtime configuration of the file system.

Before compiling and running the sample application with the added driver, you have to exclude `ConfigRAMDisk.c` from project.

Note for drivers which require hardware routines: If you have only added the template with empty function bodies until now, the project should compile without errors or warning messages. But you can only run the project on your hardware if you have finished the implementation of the hardware functions.

3.5 Step 5: Adjusting the RAM usage

The file system needs RAM for management purposes in various places. The amount of RAM required depends primarily on the configuration, especially the drivers used. The drivers which have their own level of management (such as NOR / NAND drivers) in general need more RAM than the "simple" drivers for hard drives, compact flash or SD cards.

Every driver needs to allocate RAM. The file system allocates RAM in the initialization phase and holds it while the file system is running. The macro `ALLOC_SIZE` which is located in the respective driver configuration file specifies the size of RAM used by the file system. This value should be fine-tuned according to the requirements of your target system.

What to do

Per default, `ALLOC_SIZE` is set to a value which should be appropriate for most target systems. Nevertheless, you should adjust it in order to avoid wasting. Once your file system project is up and running, you can check the real RAM requirement of the driver with the public auxiliary variable `FS_NumBytesAllocated` which is also located in the configuration file of the respective driver. Check the value of `FS_NumBytesAllocated` after the initialization of the file system (`FS_Init()`) and after a volume has been mounted. At this point `FS_NumBytesAllocated` can be used as reference for the dynamic memory usage of emFile. You should reserve a few more bytes for emFile as the value of `FS_NumBytesAllocated` is at this point, since every file which is opened needs dynamic memory for maintenance information. For more information about resource usage of the file handlers, please refer to *Dynamic RAM requirements* on page 501.

Note: If you define `ALLOC_SIZE` with a value which is smaller than the appropriate size, the file system will run into `FS_X_Panic()`. If you define `ALLOC_SIZE` with a value which is above the limits of your target system, the linker will give an error during the build process of the project.

Chapter 4

API functions

In this chapter, you will find a description of each emFile API functions. An application should only access emFile by these functions.

4.1 API function overview

The table below lists the available API functions within their respective categories.

Function	Description
File system control functions	
FS_AddOnExitHandler()	Registers a callback to be invoked when the file system de-initializes.
FS_Init()	Starts the file system.
FS_DeInit()	De-initializes the file system.
FS_Mount()	Mounts a volume.
FS_MountEx()	Mounts a volume.
FS_SetAutoMount()	Sets the mount behavior of the specified volume.
FS_Sync()	Synchronizes the given volume.
FS_Unmount()	Closes all file/directory handles and unmounts the volume.
FS_UnmountForced()	Invalidates all file/directory handles and unmounts the volume.
File system configuration functions	
FS_AddDevice()	Adds and makes a device driver accessible to emFile.
FS_AddPhysDevice()	Adds a device driver physical to emFile.
FS_AssignMemory()	Assigns memory to the file system.
FS_ConfigFileBufferDefault()	Configures the file buffers which can be used by emFile to improve the performance when reading/writing small blocks of data.
FS_ConfigUpdateDirOnWrite()	Enables that writing to a file always updates the directory entry.
FS_FAT_ConfigMaintainFATCopy()	Enables/disables the update of the second FAT allocation table.
FS_FAT_ConfigUseFSInfoSector()	Enables/disables the usage of the information from FSInfoSector.
FS_LOGVOL_Create()	Creates a logical volume.
FS_LOGVOL_AddDevice()	Adds a device to a logical volume.
FS_SetFileBufferFlags()	Changes the file buffer flags of a specified file.
FS_SetFileWriteMode()	Allows the user to modify the file writing mode emFile uses.
FS_SetFileWriteModeEx()	Sets the write mode of a volume.
FS_SetMemHandler()	Sets the memory allocation routines when file system shall use external memory allocation routines.
FS_SetMaxSectorSize()	Configures the max sector size.
File access functions	
FS_FClose()	Closes a file.
FS_FOpen()	Opens a file.
FS_FOpenEx()	Opens a file.
FS_FRead()	Reads data from a file.
FS_FWrite()	Writes data to a file.
FS_Read()	Reads data from a file.
FS_SyncFile()	Cleans the write buffer and updates the management information of a file to storage medium.

Table 4.1: emFile API function overview

Function	Description
FS_Write()	Writes data to a file.
File positioning functions	
FS_FSeek()	Sets position of a file pointer.
FS_FTell()	Returns position of a file pointer.
FS_GetFilePos()	Returns position of a file pointer.
FS_SetFilePos()	Sets position of a file pointer.
Operations on files	
FS_CopyFile()	Copies a file.
FS_CopyFileEx()	Copies a file using a buffer provided by the application.
FS_GetFileAttributes()	Retrieves the attributes of a given file or directory.
FS_GetFileTime()	Retrieves the creation, access or modify timestamp of a given file or directory.
FS_GetFileTimeEx()	Retrieves the timestamp of a given file or directory.
FS_ModifyFileAttributes()	Sets and clears attributes of given file or directory.
FS_Move()	Moves an existing file or a directory, including its children.
FS_Remove()	Deletes a file.
FS_Rename()	Renames a file/directory.
FS_SetEndOfFile()	Sets the end of a file.
FS_SetFileAttributes()	Sets the attributes of a given file or directory.
FS_SetFileTime()	Sets the timestamp of a given file or directory.
FS_SetFileTimeEx()	Sets the creation, access or modify timestamp of a given file or directory.
FS_SetFileSize()	Modifies the size of a file.
FS_Truncate()	Truncates a file to a specified size.
FS_Verify()	Verifies a file with a given data buffer.
FS_WipeFile()	Overwrites the contents of a file with random data.
Directory functions	
FS_CreateDir()	Creates a directory or a path to a directory.
FS_FindClose()	Closes a directory.
FS_FindFirstFile()	Searches for a file in a specified directory.
FS_FindNextFile()	Continues file search in a directory.
FS_MkDir()	Creates a directory.
FS_RmDir()	Removes a directory.
Formatting a medium	
FS_Format()	High-level formats a device.
FS_FormatLLIfRequired()	Checks if a device is low-level formatted and formats it if required.
FS_FormatLow()	Low-level formats a device.
FS_IsHLFormatted()	Checks if a device is high-level formatted.
FS_IsLLFormatted()	Checks if a device is low-level formatted.
File system extended functions	
FS_CheckDisk()	Checks and repairs a FAT volume.
FS_CheckDisk_ErrCode2Text()	Returns an error string to a specific check-disk error code.

Table 4.1: emFile API function overview (Continued)

Function	Description
FS_CreateMBR()	Creates a Master Boot Record.
FS_FileTimeToTimeStamp()	Converts a file time to a timestamp.
FS_FreeSectors()	Informs the storage layer about unused sectors.
FS_GetFileSize()	Retrieves the current file size of a given file pointer.
FS_GetMaxSectorSize()	Returns the logical sector size.
FS_GetNumFilesOpen()	Returns the number of opened files.
FS_GetNumVolumes()	Returns the available volumes.
FS_GetPartitionInfo()	Returns information about a disk partition.
FS_GetVolumeFreeSpace()	Gets the free space of a given volume.
FS_GetVolumeFreeSpaceKB()	Returns the free space of a given volume in kilo bytes.
FS_GetVolumeInfo()	Get volume information.
FS_GetVolumeInfoEx()	Get volume information.
FS_GetVolumeLabel()	Retrieves the label of a given volume index.
FS_GetVolumeName()	Retrieves the name of a given volume index.
FS_GetVolumeSize()	Gets the size of a given volume.
FS_GetVolumeSizeKB()	Returns the size of a given volume in kilo bytes.
FS_GetVolumeStatus()	Returns the status of a volume.
FS_IsVolumeMounted()	Returns if the volume is mounted and has correct file system information.
FS_Lock()	Claims exclusive access to file system.
FS_LockVolume()	Claims exclusive access to a volume.
FS_SetBusyLEDCallback()	Sets a busy LED callback for a specific volume.
FS_SetMemAccessCallback()	Registers a 0-copy check function for a specific volume.
FS_SetVolumeLabel()	Sets a label to a specific volume.
FS_TimeStampToFileTime()	Converts a timestamp to a file time.
FS_Unlock()	Releases exclusive access to file system.
FS_UnlockVolume()	Releases exclusive access to a volume.
Storage layer functions	
FS_STORAGE_Clean()	Performs garbage collection on the storage medium.
FS_STORAGE_CleanOne()	Performs a single garbage collection step on the storage medium.
FS_STORAGE_FreeSectors()	Informs the driver about unused sectors.
FS_STORAGE_GetCounters()	Returns the statistic counters.
FS_STORAGE_GetDeviceInfo()	Returns the device info.
FS_STORAGE_Init()	Initializes the driver and OS if necessary.
FS_STORAGE_ReadSector()	Reads a sector from a device.
FS_STORAGE_ReadSectors()	Reads multiple sectors from a device.
FS_STORAGE_RefreshSectors()	Rewrites a sector with the original data.
FS_STORAGE_ResetCounters()	Sets the statistic counters to 0.
FS_STORAGE_Sync()	Writes cached data to the storage medium.
FS_STORAGE_SyncSectors()	Writes cached sector data to storage medium.
FS_STORAGE_Unmount()	Low-level unmount. Unmounts a volume on driver layer.

Table 4.1: emFile API function overview (Continued)

Function	Description
FS_STORAGE_WriteSector()	Writes a sector to a device.
FS_STORAGE_WriteSectors()	Writes multiple sectors to a device.
FAT related functions	
FS_FAT_GrowRootDir()	Lets the root directory of a FAT32 volume grow.
FS_FAT_SupportLFN()	Add long file name support to the file system.
FS_FAT_DisableLFN()	Disables the support for the long file names.
FS_FormatSD()	High-level formats a device according to the SD card file system specification.
Error-handling functions	
FS_ClearErr()	Clears the error status of a given file pointer.
FS_ErrorNo2Text()	Retrieves text for a given error code.
FS_FEof()	Tests for end-of-file on a given file pointer.
FS_FError()	Returns the error code of a given file pointer.
Obsolete functions	
FS_CloseDir()	Closes a directory stream.
FS_DirEnt2Attr()	Gets the directory entry attributes.
FS_DirEnt2Name()	Gets the directory entry name.
FS_DirEnt2Size()	Gets the directory entry file size.
FS_DirEnt2Time()	Gets the directory entry timestamp.
FS_GetDeviceInfo()	Returns the device info.
FS_GetNumFiles()	Gets the number of files in a directory.
FS_InitStorage()	Initializes the driver and OS if necessary.
FS_OpenDir()	Opens a directory stream.
FS_ReadDir()	Reads next directory entry.
FS_ReadSector()	Reads a sector from a device.
FS_RewindDir()	Resets position of directory stream.
FS_WriteSector()	Writes a sector to a device.
FS_UnmountLL()	Low-level unmount. Unmounts a volume on driver layer.

Table 4.1: emFile API function overview (Continued)

4.2 File system control functions

4.2.1 FS_AddOnExitHandler()

Description

Registers a callback to be invoked when the file system de-initializes.

Prototype

```
void FS_AddOnExitHandler(FS_ON_EXIT_CB * pCB,
                         void          (* pfOnExit)(void));
```

Parameter	Description
pCB	IN: Structure holding the callback information. OUT: ---
pfOnExit	Pointer to the callback function to invoke.

Table 4.2: FS_AddOnExitHandler() parameter list

Additional Information

The pCB memory location is used internally by emFile and it should remain valid from the moment the handler is registered until the FS_DeInit() function is called.

The FS_DeInit() invokes all the registered callback function in reversed order that is the last registered function is called first.

In order to use this function the binary compile time switch FS_SUPPORT_DEINIT has to be enabled (has to be set to "1").

4.2.2 FS_Init()

Description

Starts the file system.

Prototype

```
void FS_Init(void);
```

Additional Information

`FS_Init()` initializes the file system and creates resources required for an OS integration of emFile. This function must be called before any other emFile function.

Example

```
#include "FS.h"

void main(void) {
  FS_Init();
  //
  // Access file system
  //
 }
```


4.2.3 FS_DeInit()

Description

De-initializes the file system. All resources which are occupied by the file system, are freed. All static variables for each layer are reset in order to guarantee that emFile is in a known state after de-initialization.

Please use this function when you are planning to reset emFile during run-time. For example this is the case if your target application uses a software reboot which re-initializes the target application.

Prototype

```
void FS_DeInit(void);
```

Additional information

In order to use this function the binary compile time switch FS_SUPPORT_DEINIT has to be enabled (has to be set to "1").

4.2.4 FS_Mount()

Description

Mounts a volume.

Prototype

```
int FS_Mount(const char * sVolumeName);
```

Parameter	Description
sVolumeName	sVolumeName is the name of a volume. If not specified, the first device in the volume table will be used.

Table 4.3: FS_Mount() parameter list

Return value

```
== 0:   Volume is not mounted
== 1:   Volume is mounted read-only
== 3:   Volume is mounted read/write
<  0:   Error code indicating the failure reason.
        Refer to FS_ErrorNo2Text() on page 177.
```

Additional Information

This function can be useful if the default auto mount behavior has been changed with `FS_SetAutoMount()`. Normally, it is not required to mount a device with `FS_Mount()`, since the file system auto mounts all accessible volumes in read/write mode. Refer to *FS_SetAutoMount()* on page 55 for an overview about the different auto mount types.

4.2.5 FS_MountEx()

Description

Mounts a volume.

Prototype

```
int FS_MountEx(const char * sVolumeName, U8 MountType);
```

Parameter	Description
sVolumeName	sVolumeName is the name of a volume. If not specified, the first device in the volume table will be used.
MountType	Specifies how the volume should be mounted.

Table 4.4: FS_MountEx() parameter list

Permitted values for parameter MountType	
FS_MOUNT_R	The volume will be read only auto mounted.
FS_MOUNT_RW	The volume will be read/write auto mounted.

Return value

== 0: Volume is not mounted
== 1: Volume is mounted read-only
== 3: Volume is mounted read/write
< 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

This function can be useful if the default auto mount behavior has been changed with FS_SetAutoMount(). Normally, it is not required to mount a device with FS_MountEx(), since the file system auto mounts all accessible volumes in read/write mode. Refer to *FS_SetAutoMount()* on page 55 for an overview about the different auto mount types.

4.2.6 FS_SetAutoMount()

Description

Sets the mount behavior of the specified volume.

Prototype

```
void FS_SetAutoMount(const char * sVolumeName,
                     U8           MountType);
```

Parameter	Description
sVolumeName	sVolumeName is the name of a volume. If not specified, the first device in the volume table will be used.
MountType	Specifies the auto mount behavior.

Table 4.5: FS_SetAutoMount() parameter list

Permitted values for parameter MountType	
FS_MOUNT_R	The volume will be read only auto mounted.
FS_MOUNT_RW	The volume will be read/write auto mounted.
0	Disables auto mount for the volume.

Additional Information

The file system auto mounts all volumes default in read/write mode.

4.2.7 FS_Sync()

Description

Writes to storage medium all modifications buffered in RAM by the file system.

Prototype

```
int FS_Sync(const char * sVolumeName);
```

Parameter	Description
sVolumeName	sVolumeName is the name of a volume.

Table 4.6: FS_Sync() parameter list

Return value

== 0: Volume synchronized.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional information

The function cleans the write buffer and updates the management information of all opened file handles. The file handles are not closed. If configured, it also cleans the write cache and the journal. FS_Sync() can be called from the same task as the one writing data or from a different task.

4.2.8 FS_Unmount()

Description

Closes all file/directory handles and unmounts the volume.

Prototype

```
void FS_Unmount(const char * sVolumeName);
```

Parameter	Description
sVolumeName	sVolumeName is the name of a volume. If not specified, the first device in the volume table will be used.

Table 4.7: FS_Unmount() parameter list

Additional Information

FS_Unmount() should be called before a volume is removed. It guarantees that all file handles to this volume are closed and the directory entries for the files are updated. This function is also useful when shutting down the system.

Example

```
#include "FS.h"

void Shutdown(void) {
  FS_Unmount("");  /* Close all file handles and unmount the default volume. */
}
```


4.2.9 FS_UnmountForced()

Description

Invalidates all file/directory handles and unmounts the volume.

Prototype

```
void FS_UnmountForced(const char * sVolumeName);
```

Parameter	Description
sVolumeName	sVolumeName is the name of a volume. If not specified, the first device in the volume table will be used.

Table 4.8: FS_UnmountForced() parameter list

Additional Information

FS_UnmountForced() should be called if a volume has been removed before it could be regular unmounted. It invalidates all file handles. If you use FS_UnmountForced() there is no guarantee that all file handles to this volume are closed and the directory entries for the files are updated.

4.3 File system configuration functions

The file system control functions listed in this section can only be used in the runtime configuration phase. This means in practice that they can only be called from within `FS_X_AddDevices()`, refer to *FS_X_AddDevices()* on page 472 for more information about this function.

4.3.1 FS_AddDevice()

Description

Adds a device to emFile.

This consists of 2 operations:

1. Add physical device. This initialises the driver, allowing the driver to identify the storage device as far as required and allocate memory required for driver level management of the device. This makes sector operations possible.
2. Add the devices as a logical device. This makes it possible to mount the device, making it accessible for the file system and allowing file operations.

Prototype

```
FS_VOLUME * FS_AddDevice(const FS_DEVICE_TYPE * pDevType);
```

Parameter	Description
pDevType	Pointer to device driver table. See *Device driver function table* on page 449 for additional information.

Table 4.9: FS_AddDevice() parameter list

Return value

Pointer of the volume added to emFile.

Additional Information

This function can be used to add an additional device driver.

4.3.2 FS_AddPhysDevice()

Description

Adds a device physical to emFile. This initialises the driver, allowing the driver to identify the storage device as far as required and allocate memory required for driver level management of the device. This makes sector operations possible.

Prototype

```
int FS_AddPhysDevice(const FS_DEVICE_TYPE * pDevType);
```

Parameter	Description
pDevType	Pointer to device driver table. See *Device driver function table* on page 449 for additional information.

Table 4.10: FS_AddDevice() parameter list

Return value

>= 0: Unit number of the device.
< 0: An error has occurred.

Additional Information

Devices that are only physically added to emFile can be combined to a logical volume. Refer to *FS_LOGVOL_Create()* on page 65 and *FS_LOGVOL_AddDevice()* on page 66 for information about logical volumes.

4.3.3 FS_AssignMemory()

Description

Assigns memory to the file system.

Prototype

```
void FS_AssignMemory(U32 * pMem, U32 NumBytes);
```

Parameter	Description
pMem	A pointer to the start of the memory region which should be assigned.
NumBytes	Number of bytes which should be assigned.

Table 4.11: FS_AssignMemory() parameter list

Additional Information

If the internal memory allocation functions (FS_SUPPORT_EXT_MEM_MANAGER == 0) are used, this function is the first function that is called in FS_X_AddDevices(). Otherwise this function does nothing. The memory assigned is used by the file system to satisfy runtime memory requirements.

4.3.4 FS_ConfigFileBufferDefault()

Description

emFile can make use of file buffers in order to increase reading/writing speeds when reading/writing a file in small chunks. In order to use file buffers the compile time switch `FS_USE_FILE_BUFFER` has to be set to 1. For more information about compile time switches, please refer to *Compile time configuration* on page 474.

In order to make file buffers usable for emFile, you have to configure a buffer size, using this function.

Prototype

```
void FS_ConfigFileBufferDefault(int BufferSize, int Flags);
```

Parameter	Description
`BufferSize`	Size of the file buffer. This buffer size will be used for every file.
`Flags`	Allowed values: ==0 Use the buffer for read operations only. ==FS_FILE_BUFFER_WRITE Use the buffer for read and write operations.

Table 4.12: FS_ConfigFileBufferDefault() parameter list

Additional information

It is only allowed to call this function once, in `FS_X_AddDevices()`. Every file has its own file buffer and the buffer size passed to this function is the same for all files. The same buffer is used for read and write operations. The buffer can be configured for read operations only (`Flags` set to 0) and changed to work also as a write buffer using `FS_SetFileBufferFlags()` for specific files.

For maximum performance it is recommended to set the size of the buffer to logical sector size. Smaller buffer sizes can also be used to reduce the RAM usage.

4.3.5 FS_FAT_ConfigMaintainFATCopy()

Description

Sets if the second copy of the FAT allocation table should be kept up to date.

Prototype

```
void FS_FAT_ConfigMaintainFATCopy(int OnOff);
```

Parameter	Description
OnOff	==1 second allocation table should be updated ==0 do not update the backup allocation table (default)

Table 4.13: FS_FAT_ConfigMaintainFATCopy() parameter list

Additional information

The function is available only when the compile-time switch FS_MAINTAIN_FAT_COPY is set to 1. For more information about compile time switches, please refer to *Compile time configuration* on page 474.

4.3.6 FS_FAT_ConfigUseFSInfoSector()

Description

Sets if the information stored in the FSInfoSector of a FAT32 volume should be evaluated.

Prototype

```
void FS_FAT_ConfigUseFSInfoSector(int OnOff);
```

Parameter	Description
OnOff	==1 use the information from FSInfoSector (default) ==0 ignore FSInfoSector information

Table 4.14: FS_FAT_ConfigUseFSInfoSector() parameter list

Additional information

The FSInfoSector stores the number of free clusters and the position of the next free cluster. The number of free clusters is used by emFile to compute the available free space on the storage medium on an efficient way. Without this information the available free space must be determined by visiting all FAT entries and counting which of them are not allocated. This operation is slow on large storage mediums. Unfortunately, the information stored in the FSInfoSector is not 100% reliable. Applications which require very reliable information about the available free space can disable the use of information from FSInfoSector by calling this function withe the OnOff parameter set to 0.

To take effect, the function must be called before any FAT32 volume is mounted. The function is available only when the compile-time switch FS_FAT_USE_FSINFO_SECTOR is set to 1. For more information about compile time switches, please refer to *Compile time configuration* on page 474.

4.3.7 FS_LOGVOL_Create()

Description

Creates a logical volume. A logical volume is the representation of one or more physical devices as a single device. It allows treating multiple physical devices as one larger device; the file system takes care of selecting the correct location on the correct physical device when reading or writing to the logical volume. Logical volumes are typically used if multiple flash devices (NOR or NAND) are present, but should be presented to the application the same way as single device with the combined capacity.

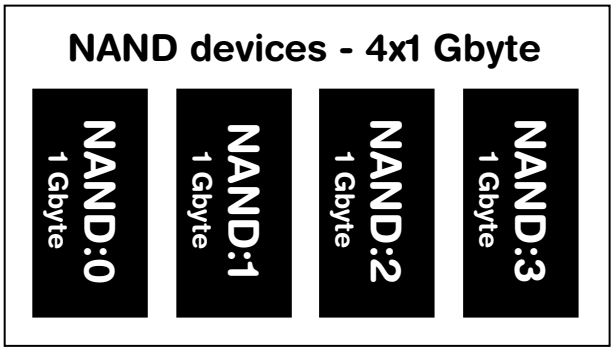
Prototype

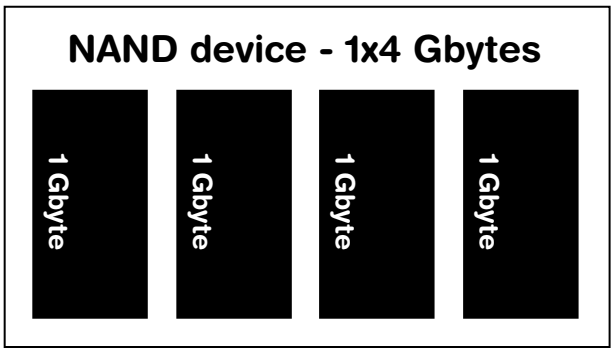
```
int FS_LOGVOL_Create (const char * sVolName);
```

Parameter	Description
sVolName	Name for the logical volume.

Table 4.15: FS_LOGVOL_Create() parameter list

Additional Information





Normally, all devices are added individually using `FS_AddDevice()`. This function adds the devices physically and logically to the file system, this means that every 1 Gbyte NAND devices can be accessed individually. Refer to *FS_AddDevice()* on page 59 for detailed information.

In contrast to adding all devices individually, all devices can be combined in a logical volume with a total size of all combined devices.

To create a logical volume the following steps have to be done:

1. The available device has to be physically added to the file system with `FS_AddPhysDevice()`.
2. A logical volume has to be created. with `FS_LOGVOL_Create()`.
3. The devices which are physically added to the file system has to be added to the logical volume with `FS_LOGVOL_AddDevice()`.

4.3.8 FS_LOGVOL_AddDevice()

Description

Adds a device to a logical volume.

Prototype

```
int FS_LOGVOL_AddDevice(const char *           sLogVolName,
                        const FS_DEVICE_TYPE * pDevice,
                        U8                     Unit,
                        U32                    StartOff,
                        U32                    NumSectors);
```

Parameter	Description
sVolName	Name of the logical volume.
pDevice	Pointer to device type that should be added.
Unit	Number of unit that should be added.
StartOff	Offset to define the start of sector range that should be used.
NumSector	Number of sectors that should be used.

Table 4.16: FS_LOGVOL_AddDevice() parameter list

Additional information

Only devices with an identical sector size can be combined to a logical volume. All additional added devices need to have the same sector size as the first physical device of the logical volume.

Example

```
void FS_X_AddDevices(void) {
  void * pRAM;
  U8     Unit1, Unit2;

  //
  //  Add the RAM drives physical to FS
  //
  Unit1 = FS_AddPhysDevice(&FS_RAMDISK_Driver);
  Unit2 = FS_AddPhysDevice(&FS_RAMDISK_Driver);
  //
  //  Allocate the required memory and configure the RAM drives
  //
  pRAM = FS_Alloc(RAMDISK_NUM_SECTORS * RAMDISK_BYTES_PER_SECTOR);
  FS_RAMDISK_Configure(Unit1, pRAM, RAMDISK_BYTES_PER_SECTOR, RAMDISK_NUM_SECTORS);
  pRAM = FS_Alloc(RAMDISK_NUM_SECTORS * RAMDISK_BYTES_PER_SECTOR);
  FS_RAMDISK_Configure(Unit2, pRAM, RAMDISK_BYTES_PER_SECTOR, RAMDISK_NUM_SECTORS);
  //
  //  Create a logical volume to composite the RAM drives
  //
  FS_LOGVOL_Create("ramc");
  //
  //  Add the devices
  //
  FS_LOGVOL_AddDevice("ramc", &FS_RAMDISK_Driver, Unit1, 0, 0);
  FS_LOGVOL_AddDevice("ramc", &FS_RAMDISK_Driver, Unit2, 0, 0);

  if (FS_IsHLFormatted("ramc") == 0) {
    FS_Format("ramc", NULL);
  }
}
```


4.3.9 FS_SetFileBufferFlags()

Description

Allows to change the file buffer flags for a specific file. This allows the user to have different file buffers (read-only, read/write, etc.) for different files.

Prototype

```
void FS_SetFileBufferFlags(FS_FILE * pFile, int Flags);
```

Parameter	Description
pFile	Handle of file which buffer flags shall be changed
Flags	Allowed values: ==0 Use the buffer for read operations only. ==FS_FILE_BUFFER_WRITE Use the buffer for read and write operations.

Table 4.17: FS_SetFileBufferFlags() parameter list

Additional information

It is only allowed to call this function immediately after opening a file. If read/write operations to the file have already been performed, the file has to be closed and re-opened in order to change the file buffer settings. This is necessary to guarantee, that all the file buffer is synchronized before changing the usage flags.

When the buffer is also used for write operations, the data is written to the storage medium only when the buffer is full or when the data crosses the boundary of a logical sector. This reduces the number of write accesses and can lead to significant performance improvements especially when writing a lot of smaller chunks.

Old name

```
FS_ConfigFileBufferFlags
```

Example

The following sample program shows how to configure the file buffer for read and write operation.

```
void SampleFileBufferFlags(void) {
  FS_FILE * pFile;

  pFile = FS_FOpen("test.txt", "w");
  if (pFile) {
    //
    // Set the file buffer for read and write operation.
    //
    FS_SetFileBufferFlags(pFile, FS_FILE_BUFFER_WRITE);
    //
    // Data is written to file buffer.
    //
    FS_Write(pFile, "Test", 4);
    //
    // Write the data from file buffer to storage and close the file.
    //
    FS_FClose(pFile);
  }
}
```


4.3.10 FS_SetFileWriteMode()

Description

Sets the default write mode.

Prototype

```
void FS_SetFileWriteMode(FS_WRITEMODE WriteMode);
```

Parameter	Description
WriteMode	Write mode which is used by emFile when writing files.

Table 4.18: FS_SetFileWriteMode() parameter list

Valid values for parameter WriteMode are:

Permitted values for parameter WriteMode	
FS_WRITEMODE_SAFE	Allows maximum fail-safe behavior. FAT and directory entry are modified after each file write access. (Slowest performance)
FS_WRITEMODE_MEDIUM	Medium fail-safe. FAT is modified after each file write access. Directory entry is written only if file is synchronized that is when FS_Sync(), FS_FClose() or FS_SyncFile() is called.
FS_WRITEMODE_FAST	Maximum performance. Directory entry is written only if file is synchronized that is when FS_Sync(), FS_FClose() or FS_SyncFile() is called. FAT is modified if necessary.

Additional information

This function can be called to configure which mode emFile should use when writing files. By default emFile uses the safe write mode which allows maximum fail safe behavior, since the FAT and the directory entry is updated on every write. There are different write modes available, which are described above in detail.

If the fast write mode is set the update of the FAT is done using a special algorithm. When writing to the file for the first time, the file system checks how many clusters in series are empty starting from the first one the file occupies. This cluster chain is remembered, so that if the file grows and needs an additional cluster, the FAT must not to be read again in order to find the next free cluster. The FAT is only modified if necessary, which is the case when:

- all clusters of the cached free-cluster-chain are occupied
- the volume or the file is synchronized that is when FS_Sync(), FS_FClose() or FS_SyncFile() is called.
- a different file is written.

Especially when writing big files, the fast write mode allows maximum performance, since usually the file system, has to search for a free cluster in the FAT and link it with the last one the file occupied, so in the worst case multiple FAT sectors have to be read in order to find a free cluster. If the pre-allocation method is used, the file system does not need to search for free clusters as the file grows overtime, but only the file position pointer needs to be modified (a new file end is specified, then the file-position pointer is set back to the old file end, so writing to the file can be resumed from there). Moving the file position pointer back for resume writing, forces the file system to go though the complete cluster chain of the file in order to find the last cluster where writing shall be resumed. Especially on big files the cluster chain can be very long so going through it can cause multiple read-accesses to the FAT and take some time.

4.3.11 FS_SetFileWriteModeEx()

Description

Sets the write mode of a volume.

Prototype

```
void FS_SetFileWriteModeEx(FS_WRITEMODE   WriteMode,
                           const char   * sVolumeName);
```

Parameter	Description
WriteMode	Write mode which is used when writing to a file.
sVolumeName	Name of the volume for which the write mode should be set.

Table 4.19: FS_SetFileWriteModeEx() parameter list

Additional information

When not explicitly set using this function the write mode of a volume is the write mode set in the call to FS_SetFileWriteMode() or the default write mode. Typical usage of this function is in a 2-volume configuration where one volume should be configured for maximum performance (FS_WRITEMODE_FAST) and the other volume should be fail-safe (FS_WRITEMODE_SAFE). Refer to *FS_SetFileWriteMode()* on page 68 for detailed information about the parameters.

4.3.12 FS_SetMemHandler()

Description

Sets the memory allocation routines when file system shall use external memory allocation routines.

Prototype

```
void FS_SetMemHandler(FS_PF_ALLOC * pfAlloc, FS_PF_FREE * pfFree);
```

Parameter	Description
pfAlloc	Pointer to the allocation function (e.g. malloc()).
pfFree	Pointer to the allocation function (e.g. free()).

Table 4.20: FS_SetMemHandler() parameter list

Additional Information

If the external memory allocation functions (`FS_SUPPORT_EXT_MEM_MANAGER` set to 1) should be used, this function is the first function that is called in `FS_X_AddDevices()` to setup the memory allocation functions. Otherwise this function does nothing.

4.3.13 FS_SetMaxSectorSize()

Description

Configures the maximum sector size.

Prototype

```
void FS_SetMaxSectorSize(unsigned MaxSectorSize);
```

Parameter	Description
MaxSectorSize	The max sector size in bytes.

Table 4.21: FS_SetMaxSectorSize() parameter list

Additional Information

The default value for a the max sector size is set 512 bytes. Therefore this function only needs to be called when a device driver is added that handles sector sizes greater than 512 bytes.
This function needs to be called within `FS_X_AddDevices()`.

4.4 File access functions

4.4.1 FS_FClose()

Description

Closes an open file.

Prototype

```
int FS_FClose(FS_FILE * pFile);
```

Parameter	Description
pFile	Pointer to a data structure of type FS_FILE.

Table 4.22: FS_FClose() parameter list

Return value

== 0: File pointer has successfully been closed.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Example

```
void MainTask(void) {
  FS_FILE *pFile;

  pFile = FS_FOpen("test.txt", "r");
  if (pFile != 0) {
    //
    // Access file.
    //
    FS_FClose(pFile);
  }
}
```


4.4.2 FS_FOpen()

Description

Opens an existing file or creates a new file depending on the parameters.

Prototype

```
FS_FILE * FS_FOpen(const char * pName,
                   const char * pMode);
```

Parameter	Description
pName	Pointer to a string that specifies the name of the file to create or open.
pMode	Mode for opening the file.

Table 4.23: FS_FOpen() parameter list

Return value

Returns the address of an FS_FILE data structure, if the file could be opened in the requested mode. NULL in case of any error.

Additional Information

A fully qualified file name looks like:
[DevName:[UnitNum:]][DirPathList]Filename

- DevName is the name of a device. If not specified, the first device in the volume table will be used.
  UnitNum is the number of the unit for this device. If not specified, unit 0 will be used. Note that it is not allowed to specify UnitNum if DevName has not been specified.
- DirPathList means a complete path to an already existing subdirectory; FS_FOpen() does not create directories. The path must start and end with a '\' character. Directory names in the path are separated by '\'. If DirPathList is not specified, the root directory on the device will be used.
- FileName desired
  If FAT is used and long file name support is not enabled, all file names and all directory names have to follow the standard FAT naming conventions (for example 8.3 notation).
  EFS supports long file names. The name length of a file or directory is limited to 235 valid characters.

The parameter pMode points to a string. If the string is one of the following, emFile will open the file in the specified mode:

Permitted values for parameter pMode	
r	Opens text file for reading.
w	Truncates to zero length or creates text file for writing.
a	Appends; opens/creates text file for writing at end-of-file.
rb	Opens binary file for reading.
wb	Truncates to zero length or creates binary file for writing.
ab	Appends; opens/creates binary file for writing at end-of-file.
r+	Opens text file for update (reading and writing).
w+	Truncates to zero length or creates text file for update.
a+	Appends; opens/creates text file for update, writing at end-of-file.

Permitted values for parameter pMode	
r+b or rb+	Opens binary file for update (reading and writing).
w+b or wb+	Truncates to zero length or creates binary file for update.
a+b or ab+	Appends; opens/creates binary file for update, writing at end-of-file.

For more details on the FS_FOpen() function, also refer to the ANSI C documentation regarding the fopen() function.
Note that emFile does not distinguish between binary and text mode; files are always accessed in binary mode.

In order to use long file names with FAT, the FS_FAT_SupportLFN() should be called before the file is opened.

In order to use characters outside of the ASCII range with FAT, emFile should be compiled with the FS_FAT_SUPPORT_UTF8 define to 1 and the support for long file names should be enabled. The name of the file should be encoded in UTF-8 format.

Example

```
FS_FILE * pFile;

/*********************************************************************
*
*       OpenFileSample1
*
*  Function description
*    Opens for reading a file on the default volume.
*/
void OpenFileSample1(void) {
  pFile = FS_FOpen("test.txt", "r");
}

/*********************************************************************
*
*       OpenFileSample2
*
*  Function description
*    Opens for writing a file on the default volume.
*/
void OpenFileSample2(void) {
  pFile = FS_FOpen("test.txt", "w");
}

/*********************************************************************
*
*       OpenFileSample3
*
*  Function description
*    Opens for reading a file in folder 'mysub' on the default volume.
*/
void OpenFileSample3(void) {
  pFile = FS_FOpen("\\mysub\\test.txt", "r");
}

/*********************************************************************
*
*       OpenFileSample4
*
*  Function description
*    Opens for reading a file on the first "ram" volume.
*/
void OpenFileSample4(void) {
  pFile = FS_FOpen("ram:test.txt", "r");
}

/*********************************************************************
*
*       OpenFileSample5
*
*  Function description
*    Opens for reading a file on the second "ram" volume.
*/
void OpenFileSample5(void) {
```



```
  pFile = FS_FOpen("ram:1:test.txt", "r");
}

/*********************************************************************
*
*       OpenFileSample6
*
*  Function description
*    Opens for writing a file with a long name for writing.
*/
void OpenFileSample6(void) {
  FS_FAT_SupportLFN();
  pFile = FS_FOpen("Long file name.text", "w");
}

/*********************************************************************
*
*       OpenFileSample7
*
*  Function description
*    Opens for writing a file with a name encoded in UTF-8 format.
*    The file system should be compiled with FS_SUPPORT_UTF8 define set to 1.
*    The name contains the following characters:
*      small a, umlaut mark
*      small o, umlaut mark
*      small sharp s
*      small u, umlaut mark
*      '.'
*      't'
*      'x'
*      't'
*/
void OpenFileSample7(void) {
  FS_FAT_SupportLFN();
  pFile = FS_FOpen("\xC3\xA4\xC3\xB6\xC3\x9F\xC3\xBC.txt", "w");
}
```


4.4.3 FS_FOpenEx()

Description

Opens an existing file or creates a new file depending on the parameters.

Prototype

```
int FS_FOpenEx(const char *  pName,
               const char *  pMode,
               FS_FILE    ** ppFile);
```

Parameter	Description
pName	Pointer to a string that specifies the name of the file to create or open.
pMode	Mode for opening the file.
ppFile	IN: --- OUT: Pointer to the opened file handle.

Table 4.24: FS_FOpenEx() parameter list

Return value

== 0: OK, file opened
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

For additional information refer to *FS_FOpen()* on page 73.

Example

```
FS_FILE * pFile;

/*********************************************************************
*
*       OpenFileSample
*
*  Function description
*    Opens for reading a file on the default volume.
*/
void OpenFileSample(void) {
  int r;

  r = FS_FOpenEx("test.txt", "r", &pFile);
  if (r) {
    printf("Could not open file (%s)\n", FS_ErrorNo2Text(r));
  }
}
```


4.4.4 FS_FRead()

Description

Reads data from an open file.

Prototype

```
U32 FS_FRead(void    *  pData,
             U32        Size,
             U32        N,
             FS_FILE *  pFile);
```

Parameter	Description
pData	Pointer to a data buffer for storing data transferred from a file.
Size	Size of an element to be transferred from a file to a data buffer.
N	Number of elements to be transferred from the file.
pFile	Pointer to a data structure of type FS_FILE.

Table 4.25: FS_FRead() parameter list

Return value

Number of elements read.

Additional Information

If there is less data transferred than specified, you should check for possible errors by calling the FS_FError() function.

Example

```
char acBuffer[100];

void MainTask(void) {
  FS_FILE* pFile;
  int i;

  pFile = FS_FOpen("test.txt", "r");
  if (pFile != 0) {
    do {
      i = FS_FRead(acBuffer, 1, sizeof(acBuffer) - 1, pFile);
      acBuffer[i] = 0;
      if (i) {
        printf("%s", acBuffer);
      }
    } while (i);
    FS_FClose(pFile);
  }
}
```


4.4.5 FS_FWrite()

Description

Writes data to an open file.

Prototype

```
U32 FS_FWrite(const void * pData,
              U32          Size,
              U32          N,
              FS_FILE    * pFile);
```

Parameter	Description
pData	Pointer to data to be written to the file.
Size	Size of an element to be transferred from a data buffer to a file.
N	Number of elements to be transferred to the file.
pFile	Pointer to a data structure of type FS_FILE.

Table 4.26: FS_FWrite() parameter list

Return value

Number of elements written.

Additional Information

If there is less data transferred than specified, you should check for possible errors by calling the FS_FError() function.

Example

```
const char acText[]="hello world\n";

void MainTask(void) {
  FS_FILE *pFile;

  pFile = FS_FOpen("test.txt", "w");
  if (pFile != 0) {
    FS_FWrite(acText, 1, strlen(acText), pFile);
    FS_FClose(pFile);
  }
}
```


4.4.6 FS_Read()

Description

Reads data from an open file.

Prototype

```
U32 FS_Read(FS_FILE * pFile,
            void    * pData,
            U32       NumBytes);
```

Parameter	Description
pFile	Pointer to a data structure of type FS_FILE.
pData	Pointer to a data buffer for storing data transferred from a file.
NumBytes	Number of bytes to be transferred from the file.

Table 4.27: FS_Read() parameter list

Return value

Number of bytes read.

Additional Information

If there is less data transferred than specified, you should check for possible errors by calling the FS_FError() function.

Example

```
char acBuffer[100];

void MainTask(void) {
  FS_FILE *pFile;

  pFile = FS_FOpen("test.txt", "r");
  if (pFile != 0) {
    do {
      i = FS_Read(pFile, acBuffer, sizeof(acBuffer) - 1);
      acBuffer[i] = 0;
      if (i) {
        printf("%s", acBuffer);
      }
    } while (i);
    FS_FClose(pFile);
  }
}
```


4.4.7 FS_SyncFile()

Description

Clears the write buffer and updates the management information of a file to storage medium.

Prototype

```
int FS_SyncFile(FS_FILE * pFile);
```

Parameter	Description
pFile	Pointer to a data structure of type FS_FILE. If NULL all opened files are updated.

Table 4.28: FS_SyncFile() parameter list

Return value

== 0: Data has been successfully synchronized.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional information

The function performs the same operations as FS_FClose() with the only difference that it leaves the file open. It cleans the write buffer, the directory entry and the allocation table entries of the file to storage medium. FS_SyncFile() is used typically with fast or medium write modes. It can also be called from a different task.

Example

```
void SampleFileSync(void) {
  FS_FILE *pFile;

  FS_SetFileWriteMode(FS_WRITEMODE_FAST);
  pFile = FS_FOpen("test.txt", "w");
  if (pFile != 0) {
    //
    // Write to file...
    //

    FS_SyncFile(pFile);
    //
    // Write to file...
    //

    FS_SyncFile(pFile);
    //
    // Write to file...
    //

    FS_FClose(pFile);
  }
}
```


4.4.8 FS_Write()

Description

Writes data to an open file.

Prototype

```
U32 FS_Write(FS_FILE    * pFile,
             const void * pData,
             U32          NumBytes);
```

Parameter	Description
pFile	Pointer to a data structure of type FS_FILE.
pData	Pointer to data to be written to the file.
NumBytes	Number of bytes that should be written to the file.

Table 4.29: FS_Write() parameter list

Return value

Number of bytes written.

Additional Information

If there is less data transferred than specified, you should check for possible errors by calling the FS_FError() function.

Example

```
const char acText[]="hello world\n";

void MainTask(void) {
  FS_FILE *pFile;

  pFile = FS_FOpen("test.txt", "w");
  if (pFile != 0) {
    FS_Write(pFile, acText, strlen(acText));
    FS_FClose(pFile);
  }
}
```


4.5 File positioning functions

4.5.1 FS_FSeek()

Description

Sets the current position of a file pointer.

Prototype

```
int FS_FSeek(FS_FILE * pFile,
             I32       Offset,
             int       Origin);
```

Parameter	Description
pFile	Pointer to a data structure of type FS_FILE.
Offset	Offset for setting the file pointer position.
Origin	Mode for positioning the file pointer.

Table 4.30: FS_FSeek() parameter list

Return value

== 0: If the file pointer has been positioned according to the parameters.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

The FS_FSeek() function moves the file pointer to a new location that is an offset in bytes from Origin. You can use FS_FSeek() to reposition the pointer anywhere in a file. The pointer can also be positioned beyond the end of the file.
Valid values for parameter Origin are:

Permitted values for parameter Origin	
FS_SEEK_SET	The origin is the beginning of the file.
FS_SEEK_CUR	The origin is the current position of the file pointer.
FS_SEEK_END	The origin is the current end-of-file position.

This function is identical to FS_SetFilePos(). Refer to *FS_SetFilePos()* on page 85 for more information.

Example

```
const char acText[]="some text will be overwritten\n";

void MainTask(void) {
  FS_FILE *pFile;

  pFile = FS_FOpen("test.txt", "w");
  if (pFile != 0) {
    FS_FWrite(acText, 1, strlen(acText), pFile);
    FS_FSeek(pFile, -4, FS_SEEK_CUR);
    FS_FWrite(acText, 1, strlen(acText), pFile);
    FS_FClose(pFile);
  }
}
```


4.5.2 FS_FTell()

Description

Returns the current position of a file pointer.

Prototype

```
I32 FS_FTell(FS_FILE * pFile);
```

Parameter	Description
pFile	Pointer to a data structure of type FS_FILE.

Table 4.31: FS_FTell() parameter list

Return value

>= 0: Current position of the file pointer in the file.
< 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

In this version of emFile, this function simply returns the file pointer element of the file's FS_FILE structure. Nevertheless, you should not access the FS_FILE structure yourself, because that data structure may change in the future.
In conjunction with FS_FSeek(), this function can also be used to examine the file size. By setting the file pointer to the end of the file using FS_SEEK_END, the length of the file can now be retrieved by calling FS_FTell().

This function is identical to FS_GetFilePos(). Refer to *FS_GetFilePos()* on page 84 for more information.

Example

```
const char acText[]="hello world\n";

void MainTask(void) {
  FS_FILE *pFile;
  I32 Pos;

  pFile = FS_FOpen("test.txt", "w");
  if (pFile != 0) {
    FS_FWrite(acText, 1, strlen(acText), pFile);
    Pos = FS_FTell(pFile);
    FS_FClose(pFile);
  }
}
```


4.5.3 FS_GetFilePos()

Description

Returns the current position of a file pointer.

Prototype

```
I32 FS_GetFilePos(FS_FILE * pFile);
```

Parameter	Description
pFile	Pointer to a data structure of type FS_FILE.

Table 4.32: FS_GetFilePos() parameter list

Return value

>= 0: Current position of the file pointer in the file.
== -1: In case of any error.

Additional Information

In this version of emFile, this function simply returns the file pointer element of the file's FS_FILE structure. Nevertheless you should not access the FS_FILE structure yourself, because that data structure may change in the future versions of emFile.
In conjunction with FS_SetFilePos(), FS_GetFilePos() this function can also be used to examine the file size. By setting the file pointer to the end of the file using FS_SEEK_END, the length of the file can now be retrieved by calling FS_GetFilePos().

This function is identical to FS_FTell(). Refer to *FS_FTell()* on page 83 for more information.

Example

```
const char acText[]="hello world\n";

void MainTask(void) {
  FS_FILE *pFile;
  I32 Pos;

  pFile = FS_FOpen("test.txt", "w");
  if (pFile != 0) {
    FS_FWrite(acText, 1, strlen(acText), pFile);
    Pos = FS_GetFilePos(pFile);
    FS_FClose(pFile);
  }
}
```


4.5.4 FS_SetFilePos()

Description

Sets the current position of a file pointer.

Prototype

```
int FS_SetFilePos(FS_FILE * pFile,
                  I32       DistanceToMove,
                  int       MoveMethod);
```

Parameter	Description
pFile	Pointer to a data structure of type FS_FILE.
DistanceToMove	A 32-bit signed value where a positive value moves the file pointer forward in the file, and a negative value moves the file pointer backward.
MoveMethod	The starting point for the file pointer move.

Table 4.33: FS_SetFilePos() parameter list

Return value

== 0: If the file pointer has been positioned according to the parameters.
== -1: In case of any error.

Additional Information

The FS_SetFilePos() function moves the file pointer to a new location that is an offset in bytes from MoveMethod. You can use FS_SetFilePos() to reposition the pointer anywhere in a file. The pointer can also be positioned beyond the end of the file.
Valid values for parameter MoveMethod are:

Permitted values for parameter MoveMethod	
FS_FILE_BEGIN	The starting point is zero or the beginning of the file.
FS_FILE_CURRENT	The starting point is the current value of the file pointer.
FS_FILE_END	The starting point is the current end-of-file position.

This function is identical to FS_FSeek(). Refer to *FS_FSeek()* on page 82 for more information.

Example

```
const char acText[]="some text will be overwritten\n";

void MainTask(void) {
  FS_FILE *pFile;

  pFile = FS_FOpen("test.txt", "w");
  if (pFile != 0) {
    FS_FWrite(acText, 1, strlen(acText), pFile);
    FS_FSeek(pFile, -4, FS_SEEK_CUR);
    FS_FWrite(acText, 1, strlen(acText), pFile);
    FS_FClose(pFile);
  }
}
```


4.6 Operations on files

4.6.1 FS_CopyFile()

Description

Copies an existing file to a new file.

Prototype

```
int FS_CopyFile(const char * sSource,
                const char * sDest);
```

Parameter	Description
sSource	Pointer to a string that specifies the name of an existing file.
sDest	Pointer to a string that specifies the name of the new file.

Table 4.34: FS_CopyFile() parameter list

Return value

== 0: If the file has been copied.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

Valid values for sSource and sDest are the same as for FS_FOpen(). The function overwrites the destination file. Refer to *FS_FOpen()* on page 73 for examples of valid names.

Note: The function allocates 512 bytes on the stack as data buffer.

Example

```
void MainTask(void) {
  FS_CopyFile("test.txt", "ram:\\info.txt");
}
```


4.6.2 FS_CopyFileEx()

Description

Copies the contents of an existing file to a new file.

Prototype

```
int FS_CopyFileEx(const char * sSource,
                  const char * sDest,
                  void       * pBuffer,
                  U32          NumBytes);
```

Parameter	Description
sSource	Pointer to a string that specifies the name of an existing file.
sDest	Pointer to a string that specifies the name of the new file.
pBuffer	Buffer to be used as temporary storage by the copy process.
NumBytes	Capacity of the temporary buffer in bytes.

Table 4.35: FS_CopyFileEx() parameter list

Return value

== 0: If the file has been copied.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

Valid values for sSource and sDest are the same as for FS_FOpen(). The function overwrites the destination file. Refer to *FS_FOpen()* on page 73 for examples of valid names.

Example

```
static U32 _aBuffer[1024 / 4];  // Buffer to be used as temporary storage.

void MainTask(void) {
  FS_CopyFileEx("src.txt", "dest.txt", _aBuffer, sizeof(_aBuffer));
}
```


4.6.3 FS_GetFileAttributes()

Description

The function retrieves attributes for a specified file or directory.

Prototype

```
U8 FS_GetFileAttributes(const char * pName);
```

Parameter	Description
pName	Pointer to a string that specifies the name of a file or directory.

Table 4.36: FS_GetFileAttributes() parameter list

Return value

>= 0x00: Attributes of the given file or directory.
== 0xFF: In case of any error.
The attributes can be one or more of the following values:

Attribute	Description
FS_ATTR_ARCHIVE	The file or directory is an archive file or directory. Applications can use this attribute to mark files for backup or removal.
FS_ATTR_DIRECTORY	The given pName is a directory.
FS_ATTR_HIDDEN	The file or directory is hidden. It is not included in an ordinary directory listing.
FS_ATTR_READ_ONLY	The file or directory is read-only. Applications can read the file but cannot write to it or delete it. In case of a directory, applications cannot delete it
FS_ATTR_SYSTEM	The file or directory is part of, or is used exclusively by, the operating system.

Table 4.37: FS_GetFileAttributes() - list of possible attributes

Additional Information

Valid values for pName are the same as for FS_FOpen(). Refer to *FS_FOpen()* on page 73 for examples of valid names.

Example

```
void MainTask(void) {
  U8 Attributes;
  char ac[100];
  Attributes = FS_GetFileAttributes("test.txt");
  sprintf(ac, "File attribute of test.txt: %d", Attributes);
  FS_X_Log(ac);
}
```


4.6.4 FS_GetFileTime()

Description

Retrieves a timestamp for a specified file or directory.

Prototype

```
int FS_GetFileTime(const char * pName,
                   U32        * pTimeStamp);
```

Parameter	Description
pName	IN: Specifies the name of a file or directory. OUT: ---
pTimeStamp	IN: --- OUT: Timestamp of the file.

Table 4.38: FS_GetFileTime() parameter list

Return value

== 0: The timestamp of the given file was successfully read and stored in pTimeStamp.

!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

Values for pName are the same as for FS_FOpen(). Refer to *FS_FOpen()* on page 73 for examples of valid names.

A timestamp is a packed value with the following format:

Bits	Description
0-4	Second divided by 2
5-10	Minute (0 - 59)
11-15	Hour (0-23)
16-20	Day of month (1-31)
21-24	Month (January -> 1, February -> 2, etc.)
25-31	Year offset from 1980. Add 1980 to get current year.

Table 4.39: FS_GetFileTime() - timestamp format description

To convert a timestamp to a *FS_FILETIME* on page 153 structure, use the function *FS_GetNumFilesOpen()* on page 126.

Example

```
void MainTask(void) {
  char ac[80];
  U32 TimeStamp;
  FS_FILETIME FileTime;
  FS_GetFileTime ("test.txt", &TimeStamp);
  FS_TimeStampToFileTime(TimeStamp, &FileTime);
  sprintf(ac, "File time of test.txt: %d-%.2d-%.2d %.2d:%.2d:%.2d",
          FileTime.Year, FileTime.Month,  FileTime.Day,
          FileTime.Hour, FileTime.Minute, FileTime.Second);
  FS_X_Log(ac);
}
```


4.6.5 FS_GetFileTimeEx()

Description

Retrieves the creation, access or modify timestamp for a specified file or directory.

Prototype

```
int FS_GetFileTime(const char * pName,
                   U32        * pTimeStamp
                   int          Index);
```

Parameter	Description
pName	Pointer to a string that specifies the name of a file or directory.
pTimeStamp	Pointer to a U32 variable that receives the timestamp.
Index	Flag that indicates which timestamp should be returned.

Table 4.40: FS_GetFileTimeEx() parameter list

Permitted values for parameter Index	
FS_FILETIME_CREATE	Indicates that the creation timestamp should be returned.
FS_FILETIME_ACCESS	Indicates that the access timestamp should be returned.
FS_FILETIME_MODIFY	Indicates that the modify timestamp should be returned.

Return value

== 0: The timestamp of the given file was successfully read and stored in pTimeStamp.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

Values for pName are the same as for FS_FOpen(). Refer to *FS_FOpen()* on page 73 for examples of valid names.

A timestamp is a packed value with the following format:

Bits	Description
0-4	Second divided by 2
5-10	Minute (0 - 59)
11-15	Hour (0-23)
16-20	Day of month (1-31)
21-24	Month (January -> 1, February -> 2, etc.)
25-31	Year offset from 1980. Add 1980 to get current year.

Table 4.41: FS_GetFileTime() - timestamp format description

To convert a timestamp to a *FS_FILETIME* on page 153 structure, use the function *FS_GetNumFilesOpen()* on page 126.

4.6.6 FS_ModifyFileAttributes()

Description

Sets and clears the attributes of the specified file or directory.

Prototype

```
U8 FS_ModifyFileAttributes(const char * sName, U8 SetMask, U8 ClrMask);
```

Parameter	Description
sName	Pointer to a string that specifies the name of a file or directory.
SetMask	Bitmask of the attributes to be set
ClrMask	Bitmask of the attributes to be cleared

Table 4.42: FS_ModifyFileAttributes() parameter list

Return value

>= 0x00: The old attributes of the given file or directory.
== 0xFF: In case of any error.
The attributes can be one or more of the following values:

Attribute	Description
FS_ATTR_ARCHIVE	The file or directory is an archive file or directory. Applications can use this attribute to mark files for backup or removal.
FS_ATTR_DIRECTORY	The given pName is a directory.
FS_ATTR_HIDDEN	The file or directory is hidden. It is not included in an ordinary directory listing.
FS_ATTR_READ_ONLY	The file or directory is read-only. Applications can read the file but cannot write to it or delete it. In case of a directory, applications cannot delete it
FS_ATTR_SYSTEM	The file or directory is part of, or is used exclusively by, the operating system.

Table 4.43: FS_ModifyFileAttributes() - list of possible attributes

Additional Information

Valid values for pName are the same as for FS_FOpen(). Refer to *FS_FOpen()* on page 73 for examples of valid names.

Example

```
void ModifyAttributesSample(void) {
  U8 Attr;

  //
  // Set the read-only flag. Clear archive flag.
  //
  Attr = FS_ModifyFileAttributes("test.txt", FS_ATTR_READ_ONLY, FS_ATTR_ARCHIVE);
  printf("Old file attributes: 0x%02x", Attr);
}
```


4.6.7 FS_Move()

Description

Moves an existing file or a directory, including its children.

Prototype

```
int FS_Move(const char * sExistingName,
            const char * sNewName);
```

Parameter	Description
sExistingname	Pointer to a string that names an existing file or directory.
sNewName	Pointer to a string that specifies the name of the new file or directory.

Table 4.44: FS_Move() parameter list

Return value

== 0: If the file was successfully moved.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

Valid values for sExistingName and sNewName are the same as for FS_FOpen(). Refer to *FS_FOpen()* on page 73 for examples of valid names. The FS_Move() function will move either a file or a directory (including its children) either in the same directory or across directories. The file or directory you want to move has to be on the same volume.

Example

```
void MainTask(void) {
  FS_Move("subdir1", "subdir2\\subdir3");
}
```


4.6.8 FS_Remove()

Description

Removes an existing file.

Prototype

```
int FS_Remove(const char * pName);
```

Parameter	Description
pName	Pointer to a string that specifies the file to be deleted.

Table 4.45: FS_Remove() parameter list

Return value

== 0: If the file was successfully removed.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

Valid values for pName are the same as for FS_FOpen(). Refer to *FS_FOpen()* on page 73 for examples of valid names.

Example

```
void MainTask(void) {
  FS_Remove("test.txt");
}
```


4.6.9 FS_Rename()

Description

Renames an existing file or a directory.

Prototype

```
int FS_Rename(const char * sExistingName,
              const char * sNewName);
```

Parameter	Description
sExistingName	Pointer to a string that names an existing file or directory.
sNewName	Pointer to a string that specifies the new name of the file or directory.

Table 4.46: FS_Rename() parameter list

Return value

== 0: If the file was successfully renamed.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

Valid values for sExistingName and sNewName are the same as for FS_FOpen(). Refer to *FS_FOpen()* on page 73 for examples of valid names. sNewName should only specify a valid file or directory name without a path.

Example

```
void MainTask(void) {
  FS_Rename("ram:\\subdir1", "subdir2");
}
```


4.6.10 FS_SetEndOfFile()

Description

Sets the end of file for the specified file.

Prototype

```
int FS_SetEndOfFile(FS_FILE * pFile);
```

Parameter	Description
pFile	Pointer to a data structure of type FS_FILE.

Table 4.47: FS_SetEndOfFile() parameter list

Return value

== 0: End of File was set.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

pFile should point to a file that has been opened with write permission. Refer to *FS_FOpen()* on page 73. This function can be used to truncate or extend a file. If the file is extended, the contents of the file between the old EOF position and the new position are not defined.

Example

```
void MainTask(void) {
  FS_FILE * pFile;

  pFile = FS_FOpen("test.bin", "r+");
  FS_SetFilePos(pFile, 2000);
  FS_SetEndOfFile(pFile);
  FS_FClose(pFile);
}
```


4.6.11 FS_SetFileAttributes()

Description

Sets attributes for a specified file or directory.

Prototype

```
int FS_SetFileAttributes(const char * pName,
                         U8           Attributes);
```

Parameter	Description
pName	Pointer to a string that specifies the name of a file or directory.
Attributes	Attributes to be set to the file or directory.

Table 4.48: FS_SetFileAttributes() parameter list

Permitted values for parameter Attributes	
FS_ATTR_ARCHIVE	The file or directory is an archive file or directory. Applications can use this attribute to mark files for backup or removal.
FS_ATTR_HIDDEN	The file or directory is hidden. It is not included in an ordinary directory listing.
FS_ATTR_READ_ONLY	The file or directory is read-only. Applications can read the file but cannot write to it or delete it. In case of a directory, applications cannot delete it.
FS_ATTR_SYSTEM	The file or directory is part of, or is used exclusively by, the operating system.

Return value

== 0: Attributes have been successfully set.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

Valid values for pName are the same as for FS_FOpen(). Refer to *FS_FOpen()* on page 73 for examples of valid names.

Example

```
void MainTask(void) {
  U8 Attributes;
  char ac[100];
  FS_SetFileAttributes("test.txt", FS_ATTR_HIDDEN);
  Attributes = FS_GetFileAttributes("test.txt");
  sprintf(ac, "File attribute of test.txt: %d", Attributes);
  FS_X_Log(ac);
}
```


4.6.12 FS_SetFileTime()

Description

The FS_SetFileTime function sets the timestamp for a specified file or directory.

Prototype

```
int FS_SetFileTime(const char * pName,
                   U32          TimeStamp);
```

Parameter	Description
pName	Pointer to a string that specifies the name of a file or directory.
TimeStamp	Timestamp to be set to the file or directory.

Table 4.49: FS_SetFileTime() parameter list

Return value

== 0: The timestamp of the given file was successfully set.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

Valid values for pName are the same as for FS_FOpen(). Refer to *FS_FOpen()* on page 73 for examples of valid names.
On a FAT medium, FS_SetFileTime() sets the creation time of a file or directory.
On a EFS medium, FS_SetFileTime() sets the time stamp of a file or directory.

A timestamp is a packed value with the following format.

Bits	Description
0-4	Second divided by 2
5-10	Minute (0 - 59)
11-15	Hour (0-23)
16-20	Day of month (1-31)
21-24	Month (January -> 1, February -> 2, etc.)
25-31	Year offset from 1980. Add 1980 to get current year.

Table 4.50: FS_SetFileTime() - timestamp format description

To convert a FS_FILETIME structure to a timestamp, use the function FS_FileTimeToTimeStamp(). For more information about the conversion have a look at the description of *FS_FileTimeToTimeStamp()* on page 122.

Example

```
void MainTask(void) {
  U32 TimeStamp;
  FS_FILETIME FileTime;

  FileTime.Year   = 2005;
  FileTime.Month  = 03;
  FileTime.Day    = 26;
  FileTime.Hour   = 10;
  FileTime.Minute = 56;
  FileTime.Second = 14;
  FS_FileTimeToTimeStamp (&FileTime, &TimeStamp);
  FS_SetFileTime("test.txt", TimeStamp);
}
```


4.6.13 FS_SetFileTimeEx()

Description

Sets the creation, access or modify timestamp for a specified file or directory.

Prototype

```
int FS_SetFileTimeEx(const char * pName,
                     U32          TimeStamp
                     int          Index);
```

Parameter	Description
pName	Pointer to a string that specifies the name of a file or directory.
TimeStamp	The value of the timestamp to set.
Index	Flag that indicates which timestamp should be set.

Table 4.51: FS_SetFileTimeEx() parameter list

Permitted values for parameter Index	
FS_FILETIME_CREATE	Indicates that the creation timestamp should be set.
FS_FILETIME_ACCESS	Indicates that the access timestamp should be set.
FS_FILETIME_MODIFY	Indicates that the modify timestamp should be set.

Return value

== 0: The timestamp of the given file was successfully set.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

Values for pName are the same as for FS_FOpen(). Refer to *FS_FOpen()* on page 73 for examples of valid names.

The EFS file system has only one timestamp hence it makes no difference which value you use for the Index parameter.

A timestamp is a packed value with the following format:

Bits	Description
0-4	Second divided by 2
5-10	Minute (0 - 59)
11-15	Hour (0-23)
16-20	Day of month (1-31)
21-24	Month (January -> 1, February -> 2, etc.)
25-31	Year offset from 1980. Add 1980 to get current year.

Table 4.52: FS_GetFileTime() - timestamp format description

To convert a timestamp to a FS_FILETIME structure, use the function *FS_GetNumFilesOpen()* on page 126. For more information about the FS_FILETIME structure, refer to *FS_FILETIME* on page 153.

4.6.14 FS_SetFileSize()

Description

Sets the end of file for the specified file.

Prototype

```
int FS_SetFileSize(FS_FILE * pFile, U32 NumBytes);
```

Parameter	Description
pFile	Pointer to a data structure of type FS_FILE.
NumBytes	The new size of the file in bytes.

Table 4.53: FS_SetFileSize() parameter list

Return value

== 0: Size of the file has been modified.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

pFile should point to a file that has been opened with write permission. Refer to *FS_FOpen()* on page 73. This function can be used to truncate or extend a file. If the file is extended, the file pointer is not moved.

Example

```
void MainTask(void) {
  FS_FILE * pFile;

  pFile = FS_FOpen("test.bin", "r+");
  FS_SetFileSize(pFile, 2000);
  FS_FClose(pFile);
}
```


4.6.15 FS_Truncate()

Description

Truncates a file opened with `FS_FOpen()` to the specified size.

Prototype

```
int FS_Truncate(FS_FILE * pFile,
                U32       NewSize);
```

Parameter	Description
pFile	Pointer to a data structure of type `FS_FILE`.
NewSize	New size of the file.

Table 4.54: FS_Truncate() parameter list

Return value

== 0: Truncation was successful.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

This function truncates an open file. Be sure that `pFile` points to a file that has been opened with write permission. For more information about setting write permission for `pFile` have a look at the description of *FS_FOpen()* on page 73.

Example

```
void MainTask(void) {
  FS_FILE * pFile;
  U32    FileSize;
  Int       Success;
  pFile = FS_FOpen("test.bin", "r+");
  FileSize = FS_GetFileSize(pFile);
  Success = FS_Truncate(pFile, FileSize - 200);
  if (Success == 0) {
    FS_X_Log("Truncation was successful.");
  } else {
    FS_X_Log("Truncation was not successful");
  }
  FS_Fclose(pFile);
}
```


4.6.16 FS_Verify()

Description

Validates a file by comparing its contents with a data buffer.

Prototype

```
int FS_Verify(FS_FILE    * pFile,
              const void * pData,
              U32          NumBytes);
```

Parameter	Description
pFile	Pointer to a file handle.
pData	Pointer to a buffer that holds the data to be verified with the file.
NumBytes	Number of bytes to be verified.

Table 4.55: FS_Verify() parameter list

Return value

== 0: If verification was successful.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

If the file contains less bytes than to be verified, only the available bytes are verified.

Note: The position of the file pointer has to set to the beginning of the data that should be verified.

Example

```
const U8 acVerifyData[4] = { 1, 2, 3, 4 };

void MainTask(void) {
  FS_FILE * pFile;
  I32          n;

  FS_Init();
  //
  // Open file and write data into
  //
  pFile = FS_FOpen("test.txt", "w+");
  FS_Write(pFile, acVerifyData, sizeof(acVerifyData));
  //
  // Determine current position of file pointer.
  //
  n = FS_FTell(pFile);
  //
  // Set file pointer to the start of the data that should be verified.
  //
  FS_FSeek(pFile, 0, FS_SEEK_SET);
  //
  // Verify data.
  //
  if (FS_Verify(pFile, acVerifyData, sizeof(acVerifyData)) == 0) {
    FS_X_Log("Verification was successful");
  } else {
    FS_X_Log("Verification failed");
  }
  //
  // Set file pointer to end of data that was written and verified.
  //
  FS_FSeek(pFile, n, FS_SEEK_SET);
  FS_FClose(pFile);

  while (1);
}
```


4.6.17 FS_WipeFile()

Description

Overwrites the contents of the entire file with random data.

Prototype

```
int FS_WipeFile(const char * sFileName);
```

Parameter	Description
sFileName	Pointer to a string that specifies the name of the file.

Table 4.56: FS_WipeFile() parameter list

Return value

== 0: File contents overwritten.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

When a file is removed the file system marks the corresponding directory entry and the clusters in the allocation table as free. The contents of the file is not modified and the file can be restore by using a disk recovery tool. This can be a problem if the file stores sensitive data. Calling `FS_WipeFile()` function before the file is removed makes the recovery of data impossible.

Note: The function allocates a 512 byte buffer on the stack.

Example

```
void WipeFileSample(void) {
  FS_FILE * pFile;

  //
  // Create a file and write data to it.
  //
  pFile = FS_FOpen("test.txt", "w");
  FS_Write(pFile, "12345", 5);
  FS_FClose(pFile);
  //
  // Overwrite the file contents with random data.
  //
  FS_WipeFile("test.txt");
  //
  // Delete the file from storage medium.
  //
  FS_Remove("test.txt");
}
```


4.7 Directory functions

4.7.1 FS_CreateDir()

Description

Creates a new directory or directory path.

Prototype

```
int FS_CreateDir(const char * sDirPath);
```

Parameter	Description
sDirPath	IN: Fully qualified directory name. OUT: ---

Table 4.57: FS_CreateDir() parameter list

Return value

==0 Directory path created.
==1 Directory path already exists.
< 0 Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional information

If a directory in the path does not exist it is created automatically.

Note: The function allocates 260 bytes on the stack.

Example

```
void CreateDirSample(void) {
  int r;

  r = FS_CreateDir("SubDir1\\SubDir2\\SubDir3");
  if (r == 0) {
    printf("Directory path created.\n");
  }
}
```


4.7.2 FS_FindClose()

Description

Closes a directory search.

Prototype

```
void FS_FindClose(FS_FIND_DATA * pfd);
```

Parameter	Description
pfd	Pointer to a FS_FIND_DATA structure.

Table 4.58: FS_FindClose() parameter list

Example

```
typedef struct {
  // Public elements, to be used by application
  U8     Attributes;
  U32    CreationTime;
  U32    LastAccessTime;
  U32    LastWriteTime;
  U32    FileSize;
  char * sFileName;
  // Private elements. Not be used by the application
  int SizeofFileName;
  FS__DIR Dir;
} FS_FIND_DATA;

FindFileSample(void) {
  FS_FIND_DATA fd;
  char acFilename[20];

  if (FS_FindFirstFile(&fd, "\\YourDir\\", acFilename, sizeof(acFilename)) == 0) {
    do {
      printf(acFilename);
    } while (FS_FindNextFile (&fd));
  }
  FS_FindClose(&fd);
}
```


4.7.3 FS_FindFirstFile()

Description

Searches for files and directories in a specified directory.

Prototype

```
int FS_FindFirstFile(FS_FIND_DATA * pfd,
                     const char   * sPath,
                     char         * sFilename,
                     int            sizeofFilename);
```

Parameter	Description
pfd	Pointer to a FS_FIND_DATA structure.
sPath	Pointer to a string containing the name of a directory which should be scanned.
sFilename	Pointer to a buffer used to store the name of a file which has been found.
sizeofFilename	Size of the buffer which contains the name of a file which has been found.

Table 4.59: FS_FindFirstFile() parameter list

Return value

== 0: Directory or file found.
== 1: No entries available in the directory.
else: An error occurred

Additional Information

A fully qualified directory name looks like:

```
[DevName:[UnitNum:]][DirPathList]DirectoryName
```

where:

- DevName is the name of a device, for example "ram" or "mmc". If not specified, the first device in the device table will be used.
  UnitNum is the number for the unit of the device. If not specified, unit 0 will be used. Note that it is not allowed to specify UnitNum if DevName has not been specified.
- DirPathList is a complete path to an existing subdirectory. The path must start and end with a '\' character. Directory names in the path are separated by '\'. If DirPathList is not specified, the root directory on the device will be used.
- DirectoryName and all other directory names have to follow the standard FAT naming conventions (for example 8.3 notation), if support for long file names is not enabled.

To open the root directory, simply use an empty string for sPath.

Refer to *Structure FS_FIND_DATA* on page 109 for more information about the structure pfd points to.

Example

Refer to *FS_FindClose()* on page 104 for an example.

4.7.4 FS_FindNextFile()

Description

Continues a file or directory search from a previous call to the `FS_FindFirstFile()` function.

Prototype

```
int FS_FindNextFile(FS_FIND_DATA * pfd);
```

Parameter	Description
`pfd`	Pointer to a `FS_FIND_DATA` structure.

Table 4.60: FS_FindNextFile() parameter list

Return value

== 1: File found in directory.
== 0: In case of any error.

Example

Refer to *FS_FindClose()* on page 104 for an example.

4.7.5 FS_MkDir()

Description

Creates a new directory.

Prototype

```
int FS_MkDir(const char * pDirName);
```

Parameter	Description
pDirName	Fully qualified directory name.

Table 4.61: FS_MkDir() parameter list

Return value

== 0: The directory was successfully created.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

Refer to *FS_FindFirstFile()* on page 105 for examples of valid fully qualified directory names. Note that `FS_MkDir()` will not create the whole `pDirName`, it will only create a directory in an already existing path.

Example

```
void FSTask1(void) {
  int Err;

  //
  // Create mydir in directory test - default driver on default device
  //
  Err = FS_MkDir("\\test\\mydir");
}

void FSTask2(void) {
  int Err;

  //
  // Create directory mydir - RAM device driver on default device
  //
  Err = FS_MkDir("ram:\\mydir");
}
```


4.7.6 FS_RmDir()

Description

Deletes a directory.

Prototype

```
int FS_RmDir(const char * pDirname);
```

Parameter	Description
pDirname	Fully qualified directory name.

Table 4.62: FS_RmDir() parameter list

Return value

== 0: If the directory has been successfully removed.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

Refer to *FS_FindFirstFile()* on page 105 for examples of valid and fully qualified directory names. `FS_RmDir()` will only delete a directory if it is empty.

Example

```
void FSTask1(void) {
  int Err;

  //
  // Remove mydir in directory test - default driver on default device
  //
  Err = FS_RmDir("\\test\\mydir");
}

void FSTask2(void) {
  int Err;

  //
  // Remove directory mydir - RAM device driver on default device
  //
  Err = FS_RmDir("ram:\\mydir");
}
```


4.7.7 Structure FS_FIND_DATA

Description

The FS_FORMAT_INFO structure represents the information used to access directories and files.

Prototype

```
typedef struct {
  //
  // Public elements, to be used by application
  //
  U8     Attributes;
  U32    CreationTime;
  U32    LastAccessTime;
  U32    LastWriteTime;
  U32    FileSize;
  char * sFileName;
  //
  // Private elements. Not be used by the application
  //
  int SizeofFileName;
  FS__DIR Dir;
} FS_FIND_DATA;
```

Members	Description
Attributes	Specifies the file attributes of the file found.
CreationTime	U32 value containing the time the file was created.
LastAccessTime	U32 value containing the time that the file was last accessed.
LastWriteTime	U32 value containing the time that the file was last written to.
FileSize	U32 value specifies the size of the file.
sFileName	String that is the name of the file.
SizeofFileName	Size of the file name. (Private element. Not to be used by application.)
Dir	Directory administration structure. (Private element. Not to be used by an application.)

Table 4.63: FS_FIND_DATA - list of structure elements

4.8 Formatting a medium

In general, before a medium can be used to read or write to a file, it needs to be formatted. Flash cards are usually already preformatted and do not need to be formatted. Flashes used as storage devices have normally to be reformatted. These devices require a low-level format first, then a high-level format. The low-level format is device-specific, the high-level format depends on the file system only. (FAT-format typically).

4.8.1 FS_Format()

Description

Performs a high-level format of a device. This means putting the management information required by the File system on the medium. In case of FAT, this means primarily initialization of FAT and the root directory, as well as the BIOS parameter block.

Prototype

```
int FS_Format(const char *     pVolumeName
              FS_FORMAT_INFO * pFormatInfo);
```

Parameter	Description
pVolumeName	Name of the device to format.
pFormatInfo	Optional info for formatting.

Table 4.64: FS_Format() parameter list

Return value

== 0: High-level format successful.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

There are many different ways to format a medium, even with one file system. If the second parameter is not specified, reasonable default values are used (auto-format). However, FS_Format() also allows fine-tuning of the parameters used. For details, refer to the sample file FS_Format.c, which is shipped with emFile.

For more information about the structure FS_FORMAT_INFO, refer to *FS_FORMAT_INFO* on page 116.

4.8.2 FS_FormatLLIfRequired()

Description

Checks if the volume is low-level formatted and formats the volume if required.

Prototype

```
int FS_FormatLLIfRequired(const char * pVolumeName);
```

Parameter	Description
pVolumeName	Name of the device to low-level format.

Table 4.65: FS_FormatLLIfRequired() parameter list

Return value

== 0: Ok - low-level format successful.
== 1: Low-level format not required.
< 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

Low-level format is only required for devices which have their own management level. These are the drivers for NOR flashes, NAND flashes. MMC, SD and all other cards do not require a low-level format.

4.8.3 FS_FormatLow()

Description

Low-level formats a device. Required by NAND/NOR flashes prior to format.

Prototype

```
int FS_FormatLow(const char * pDeviceName);
```

Parameter	Description
pVolumeName	Name of the device to low-level format.

Table 4.66: FS_FormatLow() parameter list

Return value

== 0: Low-level format successful.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

Low-level format is only required for devices which have their own management level. These are the drivers for NOR flashes, NAND flashes and SMC cards. MMC, SD and all other cards do not require a low-level format.

4.8.4 FS_IsHLFormatted()

Description

Checks if the volume is high-level formatted.

Prototype

```
int FS_IsHLFormatted(const char * sVolumeName);
```

Parameter	Description
pVolumeName	Name of the device to check.

Table 4.67: FS_IsHLFormatted() parameter list

Return value

== 1: Volume is high-level formatted.
== 0: Volume is not high-level formatted.
< 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional information

This function can be use to determine if the format of a partition is supported by the emFile. If the partition format is unknown the function returns 0.

4.8.5 FS_IsLLFormatted()

Description

Checks if the volume is low-level formatted.

Prototype

```
int FS_IsHLFormatted(const char * sVolumeName);
```

Parameter	Description
sVolumeName	Name of the device to check.

Table 4.68: FS_IsLLFormatted() parameter list

Return value

== 1: Volume is low-level formatted.
== 0: Volume is not low-level formatted.
< 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

Low-level format is only required for devices which have their own management level. These are the drivers for NOR flashes, NAND flashes. MMC, SD and all other cards do not require a low-level format.

4.8.6 FS_FORMAT_INFO

Description

The FS_FORMAT_INFO structure represents the information used to format a volume.

Prototype

```
typedef struct {
  U16 SectorsPerCluster;
  U16 NumRootDirEntries;
  FS_DEV_INFO * pDevInfo;
} FS_FORMAT_INFO;
```

Members	Description
SectorsPerCluster	A cluster is the minimal unit size a file system can handle. Sectors are combined together to form a cluster. Value should be a power of 2, for example 1, 2, 4, 8, 16, 32, 64. Bigger values lead to a higher read/write performance with big files, low values (1) make more efficient use of disk space.
NumRootDirEntries	Represents the number of directory entries the root directory should have. Typically it is only used for FAT12 and FAT16 drives. FAT32 has a dynamically grow table. If this element is used and not set to an invalid value (!= 0), emFile will use a default value of 256. If warnings are enabled, a warning message is output.
pDevInfo	Pointer to a FS_DEV_INFO structure. Optional IN parameter, passing information to the function. Typically NULL, unless some device specifics need to be passed to the function.

Table 4.69: FS_FORMAT_INFO - list of structure elements

4.8.7 FS_DEV_INFO

Description

The FS_DEV_INFO structure contains the medium information.

Prototype

```
typedef struct {
  U16 NumHeads;
  U16 SectorsPerTrack;
  U32 NumSectors;
  U16 BytesPerSector;
} FS_DEV_INFO;
```

Members	Description
NumHeads	Number of heads on the drive. This is relevant for mechanical drives only.
SectorsPerTrack	Number of sectors in each track. This is relevant for mechanical drives only.
NumSectors	Total number of sectors on the medium.
BytesPerSector	Number of bytes per sector.

Table 4.70: FS_DEV_INFO - list of structure elements

4.9 Extended functions

4.9.1 FS_CheckDisk()

Description

Checks and repairs a volume (FAT and EFS).

Prototype

```
int FS_CheckDisk(const char                          * sVolumeName,
                 void                                * pBuffer,
                 U32                                   BufferSize,
                 int                                   MaxRecursionLevel,
                 FS_CHECKDISK_ON_ERROR_CALLBACK * pfOnError);
```

Parameter	Description
sVolumeName	Volume name as a string.
pBuffer	IN: Buffer that will be used by the function as temporary storage. OUT: ---
BufferSize	Size of the specified buffer in bytes.
MaxRecursionLevel	The maximum directory level the function should check.
pfOnError	Pointer to a callback function which is invoked in case of an error.

Table 4.71: FS_CheckDisk() parameter list

Return value

< 0: Invalid parameter or file system error.
==FS_CHECKDISK_RETVAL_OK:
No errors found.
==FS_CHECKDISK_RETVAL_RETRY:
An error has be found and repaired, retry is required.
== FS_CHECKDISK_RETVAL_ABORT:
User specified an abort of disk checking operation through callback.
== FS_CHECKDISK_RETVAL_MAX_RECURSE:
Maximum recursion level reached.

Additional Information

This function can be used to check if there are any errors on a specific volume and if necessary, repair the found errors. Typically, the buffer should be larger than 4 Kbyte. The minimum size of this buffer can be calculated using the following formula:

```
NumBytes = 12 * (BytesPerSector * 8) / BitsPerATEntry
```

Parameter	Description
NumBytes	Required buffer size in bytes
BytesPerSector	Size of a logical sector in bytes
BitsPerATEntry	Size of an allocation table entry in bits: • 12 - FAT12 • 16 - FAT16 • 32 - FAT32 • 32 - EFS

Table 4.72: Buffer size computation parameters

The callback function is used to notify the user about the error that occurred and to ask whether the error should be fixed or not. To get a detailed information string of the error that occurred, the parameter ErrCode can be passed to

FS_CheckDisk_ErrCode2Text(). The return value of the callback function indicates which action should be performed for the encountered error. For more information see *FS_CHECKDISK_ON_ERROR_CALLBACK* on page 150.

The contents of the lost cluster chains the user decides to save are copied to files named FILE<FileIndex>.CHK in directories named FOUND.<DirIndex>. FileIndex is a 0-based 4 digit decimal number and DirIndex is a 0-based 3 decimal number. The first directory will have the name "FOUND.000", the second "FOUND.001", etc. The fist file in the directory will have the name "FILE0000.CHK", the second "FILE0001.CHK", etc.

Before checking the disk the function will close all opened file handles. During the checking it is not allowed to access the medium.

Example

```
#include <stdarg.h>
#include "FS.h"

static U32 _aBuffer[5000];

/*********************************************************************
*
*       _OnError
*/
int _OnError(int ErrCode, ...) {
  va_list ParamList;
  const char * sFormat;
  char  c;
  char ac[1000];

  sFormat = FS_CheckDisk_ErrCode2Text(ErrCode);
  if (sFormat) {
    va_start(ParamList, ErrCode);
    vsprintf(ac, sFormat, ParamList);
    printf("%s\n", ac);
  }
  if (ErrCode != FS_ERRCODE_CLUSTER_UNUSED) {
    printf("  Do you want to repair this? (y/n/a) ");
  } else {
    printf("  * Convert lost cluster chain into file (y)\n"
           "  * Delete cluster chain                 (d)\n"
           "  * Do not repair                        (n)\n"
           "  * Abort                                (a) ");
    printf("\n");
  }
  c = getchar();
  printf("\n");
  if ((c == 'y') || (c == 'Y')) {
    return FS_CHECKDISK_ACTION_SAVE_CLUSTERS;
  } else if ((c == 'a') || (c == 'A')) {
    return FS_CHECKDISK_ACTION_ABORT;
  } else if ((c == 'd') || (c == 'D')) {
    return FS_CHECKDISK_ACTION_DELETE_CLUSTERS;
  }
  return FS_CHECKDISK_ACTION_DO_NOT_REPAIR;     // Do not fix.
}

/*********************************************************************
*
*       MainTask
*/
void MainTask(void) {
  int r;

  FS_Init();
  r = FS_CheckDisk("", &_aBuffer[0], sizeof(_aBuffer), 5, _OnError);
  while (r == FS_CHECKDISK_RETVAL_RETRY) {
    ;
  }
}
```


4.9.2 FS_CheckDisk_ErrCode2Text()

Description

Returns an error string to a specific check-disk error code.

Prototype

```
const char * FS_FAT_CheckDisk_ErrCode2Text(int ErrCode);
```

Parameter	Description
ErrCode	Check-disk error code.

Table 4.73: FS_CheckDisk_ErrCode2Text() parameter list

Return value

A pointer to a statically allocated string holding the error text.

Additional information

The following error codes are defined as

Permitted values for the parameter ErrCode	
FS_CHECKDISK_ERRCODE_0FILE	A file of size zero has allocated cluster(s).
FS_CHECKDISK_ERRCODE_SHORTEN_CLUSTER	A cluster chain for a specific file is longer than its file size.
FS_CHECKDISK_ERRCODE_CROSSLINKED_CLUSTER	A cluster is cross-linked (used for multiple files / directories)
FS_CHECKDISK_ERRCODE_FEW_CLUSTER	Too few clusters allocated to file.
FS_CHECKDISK_ERRCODE_CLUSTER_UNUSED	A cluster is marked as used, but not assigned to a file or directory.
FS_CHECKDISK_ERRCODE_CLUSTER_NOT_EOC	A cluster is not marked as end-of-chain.
FS_CHECKDISK_ERRCODE_INVALID_CLUSTER	A cluster is not a valid cluster.
FS_CHECKDISK_ERRCODE_INVALID_DIRECTORY_ENTRY	A directory entry is invalid.

Typically, this function is used in the callback function for the error handling that is used by FS_CheckDisk(). See *FS_CheckDisk()* on page 118 for an example.

4.9.3 FS_CreateMBR()

Description

Stores a Master Boot Record to the sector 0 of a specified volume.

Prototype

```
int FS_CreateMBR(const char        * sVolumeName,
                 FS_PARTITION_INFO * pPartInfo,
                 int                 NumPartitions);
```

Parameter	Description
sVolumeName	IN: Volume name. If the empty string is specified, the first device in the volume table will be used. NULL is not allowed. OUT: ---
pPartInfo	IN: List of partition entries to create. OUT: ---
NumPartitions	Number of partition entries to create.

Table 4.74: FS_CreateMBR() parameter list

Return value

== 0: MBR created.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

The function overwrites any information stored in the sector 0 of the volume. The partition entries are stored in the order specified in the `pPartInfo` array: `pPartInfo[0]` is the first partition, `pPartInfo[1]` is the second one, etc. If the `Type` field of the `FS_PARTITION_INFO` structure is set to 0 the function will determine the partition type and the CHS addresses (`Type`, `StartAddr` and `EndAddr`) automatically using the values stored in the `StartSector` and `NumSector` fields.

Example

This example creates a MBR with 2 partitions. The first partition is bootable. All parameters are explicitly configured. The second partition is not bootable and the type and CHS addresses are computed by the function.

```
void CreateMBRSample(void) {
  FS_PARTITION_INFO aPartInfo[2];

  memset(aPartInfo, 0, sizeof(aPartInfo));
  //
  // First partition.
  //
  aPartInfo[0].IsActive           = 1;
  aPartInfo[0].StartSector        = 10;
  aPartInfo[0].NumSectors         = 100000;
  aPartInfo[0].Type               = 6;
  aPartInfo[0].StartAddr.Cylinder = 0;
  aPartInfo[0].StartAddr.Head     = 0;
  aPartInfo[0].StartAddr.Sector   = 11;
  aPartInfo[0].EndAddr.Cylinder   = 538;
  aPartInfo[0].EndAddr.Head       = 1;
  aPartInfo[0].EndAddr.Sector     = 10;
  //
  // Second partition.
  //
  aPartInfo[1].StartSector = 200000;
  aPartInfo[1].NumSectors  = 10000;
  FS_CreateMBR("", aPartInfo, 2);
}
```


4.9.4 FS_FileTimeToTimeStamp()

Description

Converts a given FS_FILE_TIME structure to a timestamp.

Prototype

```
void FS_FileTimeToTimeStamp(const FS_FILETIME * pFileTime,
                            U32               * pTimeStamp);
```

Parameter	Description
pFileTime	Pointer to a data structure of type FS_FILETIME, that holds the data to be converted.
pTimeStamp	Pointer to a U32 variable to store the timestamp.

Table 4.75: FS_FileTimeToTimeStamp() parameter list

Additional Information

Refer to *FS_FILETIME* on page 153 to get information about the FS_FILETIME data structure.

4.9.5 FS_FreeSectors()

Description

Informs the storage layer about unused sectors.

Prototype

```
int FS_FreeSectors(const char * pVolumeName);
```

Parameter	Description
pVolumeName	Volume on which to perform the operation.

Table 4.76: FS_FreeSectors() parameter list

Return value

== 0: Operation executed successfully.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

The function checks each entry of the allocation table and if it is not used informs the storage layer that the sectors assigned to the cluster do not store valid data. This information is used by the NAND and NOR driver to optimize the internal copy process of a data block.

4.9.6 FS_GetFileSize()

Description

Gets the current file size of a file.

Prototype

```
U32 FS_GetFileSize(FS_FILE * pFile);
```

Parameter	Description
pFile	Pointer to a data structure of type FS_FILE.

Table 4.77: FS_GetFileSize() parameter list

Return Value

>= 0: File size in bytes (0 - 0xFFFFFFFE).
== 0xFFFFFFFF: In case of any error.

4.9.7 FS_GetMaxSectorSize()

Description

Returns the size of the configured logical sector.

Prototype

```
U32 FS_GetMaxSectorSize(void);
```

Return Value

Size of a logical sector in bytes.

4.9.8 FS_GetNumFilesOpen()

Description

Returns the number of opened files.

Prototype

```
int FS_GetNumFilesOpen(void);
```

Return Value

Number of opened file handles.

4.9.9 FS_GetNumVolumes()

Description

Retrieves the number of available volumes.

Prototype

```
int FS_GetNumVolumes(void);
```

Return Value

The number of available volumes.

Additional Information

This function can be used to get the name of each available volume. Refer to *FS_GetVolumeName()* on page 135 for getting more information.

4.9.10 FS_GetPartitionInfo()

Description

Returns information about a disk partition.

Prototype

```
int FS_GetPartitionInfo(const char        * sVolumeName,
                        FS_PARTITION_INFO * pPartInfo,
                        U8                  PartIndex);
```

Parameter	Description
sVolumeName	IN: Volume name. If the empty string is specified, the first device in the volume table will be used. NULL is not allowed. OUT: ---
pPartInfo	IN: --- OUT: Information about partition
PartIndex	Index of in the partition table to read from (0-3).

Table 4.78: FS_GetPartitionInfo() parameter list

Return Value

== 0: Partition information returned.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

The information is read from the Master Boot Record (MBR) stored on sector 0 of the volume. An error is returned if no MBR is present on the storage medium. If the `Type` field of the `FS_PARTITION_INFO` structure is 0, the partition entry is not valid.

Example

This sample show how to list the contents of the partition list stored in the Master Boot Record. Only the valid entries are displayed.

```
void PartitionInfoSample(void) {
  int               iPart;
  FS_PARTITION_INFO PartInfo;

  for (iPart = 0; iPart < FS_NUM_PARTITIONS; ++iPart) {
    FS_GetPartitionInfo("", &PartInfo, iPart);
    if (PartInfo.Type) {
      printf("  Index:         %u\n"
             "  StartSector:   %lu\n"
             "  NumSectors:    %lu\n"
             "  Type:          %u\n"
             "  IsActive:      %u\n"
             "  FirstCylinder: %u\n"
             "  FirstHead:     %u\n"
             "  FirstSector:   %u\n"
             "  LastCylinder:  %u\n"
             "  LastHead:      %u\n"
             "  LastSector:    %u\n\n", iPart,
                                       PartInfo.StartSector,
                                       PartInfo.NumSectors,
                                       PartInfo.Type,
                                       PartInfo.IsActive,
                                       PartInfo.StartAddr.Cylinder,
                                       PartInfo.StartAddr.Head,
                                       PartInfo.StartAddr.Sector,
                                       PartInfo.EndAddr.Cylinder,
                                       PartInfo.EndAddr.Head,
                                       PartInfo.EndAddr.Sector);
    }
  }
}
```


4.9.11 FS_GetVolumeFreeSpace()

Description

Gets amount of free space on a specific volume.

Prototype

```
U32 FS_GetVolumeFreeSpace(const char * sVolumeName);
```

Parameter	Description
sVolumeName	Pointer to a string that specifies the volume name.

Table 4.79: FS_GetVolumeFreeSpace() parameter list

Return Value

> 0: Amount of free space in bytes. Free space larger than 4 GB is reported as 0xFFFFFFFF (the maximum value of a U32).
== 0: If the volume cannot be found.

Additional Information

Note that free space larger than four Gbytes is reported as 0xFFFFFFFF because a U32 cannot represent bigger values. The function FS_GetVolumeInfo() can be used for larger spaces. If you do not need to know if there is more than 4 GB of free space available, you can still reliably use FS_GetVolumeFreeSpace().

Valid values for sVolumeName have the following structure:

```
[DevName:[UnitNum:]]
```

where:

- DevName is the name of a device. If not specified, the first device in the volume table will be used.
- UnitNum is the number of the unit of the device. If not specified, unit 0 will be used.

Note that to specify UnitNum, also DevName has to be specified.

4.9.12 FS_GetVolumeFreeSpaceKB()

Description

Gets amount of free space on a specific volume in kilo bytes.

Prototype

```
U32 FS_GetVolumeFreeSpaceKB(const char * sVolumeName);
```

Parameter	Description
sVolumeName	Pointer to a string that specifies the volume name.

Table 4.80: FS_GetVolumeFreeSpaceKB() parameter list

Return Value

> 0: Amount of free space in kilo bytes.
== 0: If the volume cannot be found.

Additional Information

Valid values for `sVolumeName` have the following structure:

```
[DevName:[UnitNum:]]
```

where:

- `DevName` is the name of a device. If not specified, the first device in the volume table will be used.
- `UnitNum` is the number of the unit of the device. If not specified, unit 0 will be used.

Note that to specify `UnitNum`, also `DevName` has to be specified.

4.9.13 FS_GetVolumeInfo()

Description

Gets volume information, that is the number of clusters (total and free), sectors per cluster, and bytes per sector. The function collects volume information and stores it into the given FS_DISK_INFO structure.

Prototype

```
int FS_GetVolumeInfo(const char    * sVolumeName,
                     FS_DISK_INFO * pInfo);
```

Parameter	Description
sVolumeName	Volume name as a string.
pInfo	Pointer to a FS_DISK_INFO structure.

Table 4.81: FS_GetVolumeInfo() parameter list

Return Value

== 0: If volume information could be retrieved successfully.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

Valid values for sVolumeName have the following structure:

```
[DevName:[UnitNum:]]
```

where:

- DevName is the name of a device. If not specified, the first device in the volume table will be used.
- UnitNum is the number of the unit of the device. If not specified, unit 0 will be used.

Note that to specify UnitNum, also DevName has to be specified.

Example

```
#include "FS.h"
#include <stdio.h>

void MainTask(void) {
  FS_DISK_INFO Info;

  if (FS_GetVolumeInfo("ram:", &Info) == -1) {
    printf("Failed to get volume information.\n");
  } else {
    printf("Number of total clusters = %d\n"
           "Number of free clusters  = %d\n"
           "Sectors per cluster      = %d\n"
           "Bytes per sector         = %d\n",
           Info.NumTotalClusters,
           Info.NumFreeClusters,
           Info.SectorsPerCluster,
           Info.BytesPerSector);
    }
}
```


4.9.14 FS_GetVolumeInfoEx()

Description

Returns information about the volume into the given FS_DISK_INFO structure.

Prototype

```
int FS_GetVolumeInfoEx(const char    * sVolumeName,
                       FS_DISK_INFO * pInfo,
                       int            Flags);
```

Parameter	Description
sVolumeName	Volume name as a string.
pInfo	Pointer to a FS_DISK_INFO structure.
Flags	Bitmask which controls what add. information should be returned.

Table 4.82: FS_GetVolumeInfoEx() parameter list

Permitted values for parameter Flags	
FS_DISKINFO_FLAG_FREE_SPACE	Information is returned about the number of free clusters.

Return Value

== 0: If volume information could be retrieved successfully.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

Valid values for sVolumeName have the following structure:

```
[DevName:[UnitNum:]]
```

where:

- DevName is the name of a device. If not specified, the first device in the volume table will be used.
- UnitNum is the number of the unit of the device. If not specified, unit 0 will be used.

Note that to specify UnitNum, also DevName has to be specified.

Example

```
#include "FS.h"
#include <stdio.h>

void VolumeInfoSample(void) {
  int r;
  FS_DISK_INFO   Info;
  const char   * sFSType;

  r = FS_GetVolumeInfoEx("", &Info, FS_DISKINFO_FLAG_FREE_SPACE);
  if (r) {
    printf("Failed to get volume information.\n");
  } else {
    switch (Info.FSType) {
    case FS_TYPE_FAT12:
      sFSType = "FAT12";
      break;
    case FS_TYPE_FAT16:
      sFSType = "FAT16";
      break;
    case FS_TYPE_FAT32:
      sFSType = "FAT32";
      break;
    case FS_TYPE_EFS:
      sFSType = "EFS";
      break;
    default:
```



```
      sFSType = "Unknown";
      break;
    }
    sFSType = "";
    printf("Number of total clusters:   %d\n"
           "Number of free clusters:    %d\n"
           "Sectors per cluster:        %d\n"
           "Bytes per sector:           %d\n"
           "Number of entries in root:  %d\n"
           "File system type:           %s\n"
           "Formatted acc. to SD spec.: %s\n",
           Info.NumTotalClusters,
           Info.NumFreeClusters,
           Info.SectorsPerCluster,
           Info.BytesPerSector,
           Info.NumRootDirEntries,
           sFSType,
           Info.IsSDFormatted ? "Yes" : "No");
    }
}
```


4.9.15 FS_GetVolumeLabel()

Description

Returns a volume label name if one exists.

Prototype

```
int FS_GetVolumeLabel(const char * sVolumeName,
                      char       * pVolumeLabel
                      unsigned     VolumeLabelSize);
```

Parameter	Description
sVolumeName	Volume name as a string.
pVolumeLabel	Pointer to a buffer to receive the volume label.
pVolumeLabelSize	Size of the buffer which can used to store pVolumeLabel.

Table 4.83: FS_GetVolumeLabel() parameter list

Return Value

== 0: Volume label stored in the output buffer.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

4.9.16 FS_GetVolumeName()

Description

Retrieves the name of the specified volume index.

Prototype

```
int FS_GetVolumeName(int    Index,
                     char * pBuffer,
                     int    BufferSize);
```

Parameter	Description
Index	Index number of the volume.
pBuffer	Pointer to a buffer that receives the null-terminated string for the volume name.
BufferSize	Size of the buffer to receive the null terminated string for the volume name.

Table 4.84: FS_GetVolumeName() parameter list

Return Value

If the function succeeds, the return value is the length of the string copied to pBuffer, excluding the terminating null character, in bytes.
If the pBuffer buffer is too small to contain the volume name, the return value is the size of the buffer required to hold the volume name plus the terminating null character. Therefore, if the return value is greater than BufferSize, make sure to call the function again with a buffer that is large enough to hold the volume name.

Example

```
void ShowAvailableVolumes(void) {
  int NumVolumes;
  int i;
  int BufferSize;
  char acVolume[12];

  BufferSize = sizeof(acVolume);
  NumVolumes = FS_GetNumVolumes();
  FS_X_Log("Available volumes:\n");
  for (i = 0; i < NumVolumes; i++) {
    if (FS_GetVolumeName(i, &acVolume[0], BufferSize) < BufferSize) {
      FS_X_Log(acVolume);
      FS_X_Log("\n");
    }
  }
}
```


4.9.17 FS_GetVolumeSize()

Description

Gets the total size of a specific volume.

Prototype

```
U32 FS_GetVolumeSize(const char * sVolumeName);
```

Parameter	Description
sVolumeName	Volume name as a string.

Table 4.85: FS_GetVolumeSize() parameter list

Return Value

Volume size in bytes. Volume sizes larger than 4 Gbyte are truncated to 0xFFFFFFFF (the maximum value of a U32).

Additional Information

Note that volume sizes larger than 4 Gbytes are reported as 0xFFFFFFFF because a U32 cannot represent bigger values. The function FS_GetVolumeInfo() can be used for larger media. If you do not need to know if the total space is bigger than 4 Gbytes, you can still reliably use FS_GetVolumeSize().

Valid values for sVolumeName have the following structure:

```
[DevName:[UnitNum:]]
```

where:

- DevName is the name of a device. If not specified, the first device in the volume table will be used.
- UnitNum is the number of the unit of the device. If not specified, unit 0 will be used.

Note that to specify UnitNum, also DevName has to be specified.

4.9.18 FS_GetVolumeSizeKB()

Description

Gets the total size of a specific volume in kilo bytes.

Prototype

```
U32 FS_GetVolumeSizeKB(const char * sVolumeName);
```

Parameter	Description
sVolumeName	Volume name as a string.

Table 4.86: FS_GetVolumeSizeKB() parameter list

Return Value

> 0: Amount of free space in kilo bytes.
== 0: If the volume cannot be found.

Additional Information

Valid values for `sVolumeName` have the following structure:

```
[DevName:[UnitNum:]]
```

where:

- `DevName` is the name of a device. If not specified, the first device in the volume table will be used.
- `UnitNum` is the number of the unit of the device. If not specified, unit 0 will be used.

Note that to specify `UnitNum`, also `DevName` has to be specified.

4.9.19 FS_GetVolumeStatus()

Description

Returns the status of a volume.

Prototype

```
int FS_GetVolumeStatus(const char  * sVolumeName);
```

Parameter	Description
sVolumeName	Volume name as a string.

Table 4.87: FS_GetVolumeStatus() parameter list

Return Value

Return value	Description
FS_MEDIA_STATE_UNKNOWN	The volume state is unknown.
FS_MEDIA_NOT_PRESENT	A volume is not present.
FS_MEDIA_IS_PRESENT	A volume is present.

Table 4.88: FS_GetVolumeStatus() - list of return values

4.9.20 FS_IsVolumeMounted()

Description

Returns if a volume was successfully mounted and has correct file system information.

Prototype

```
int FS_IsVolumeMounted(const char * sVolumeName);
```

Parameter	Description
sVolumeName	Volume name as a string.

Table 4.89: FS_IsVolumeMounted() parameter list

Return Value

== 1: If volume information is mounted.
== 0: In case of error, for example if the volume could not be found, is not detected, or has incorrect file system information.

Example

```
#include "FS.h"
#include <stdio.h>

void MainTask(void) {
  if (FS_IsVolumeMounted("ram:")) {
    printf("Volume is already mounted.\n");
 } else {
    printf("Volume is not mounted.\n");
 }
}
```


4.9.21 FS_Lock()

Description

Claims the exclusive access to file system.

Prototype

```
void FS_Lock(void);
```

Additional information

The execution of the task that calls this function is suspended until the exclusive access to file system can be granted. Typically used by an application when driver specific functions are called from different tasks. These functions are usually not protected against concurrent accesses. No other task can perform file system operations until `FS_Unlock()` is called. If `FS_OS_LOCKING` is not set to 1, the function does nothing.

Note: The file system API functions are multitasking safe. Explicit locking using `FS_Lock()`/`FS_Unlock()` is not required.

4.9.22 FS_LockVolume()

Description

Claims the exclusive access to a given volume.

Prototype

```
void FS_LockVolume(const char * sVolumeName);
```

Parameter	Description
sVolumeName	Volume name as a string.

Additional information

The execution of the task that calls this function is suspended until the exclusive access to file system can be granted. If FS_OS_LOCKING is not set to 2, the function does nothing.

Note: The file system API functions are multitasking safe. Explicit locking using FS_LockVolume()/FS_UnlockVolume() is not required.

4.9.23 FS_SetBusyLEDCallback()

Description

Specifies callback function to control an LED which shows the state of a specific volume.

Prototype

```
void FS_SetBusyLEDCallback(const char            * sVolumeName,
                           FS_BUSY_LED_CALLBACK * pfBusyLEDCallback);
```

Parameter	Description
sVolumeName	Volume name as a string.
pfBusyLEDCallback	Callback function which is invoked when the LED status should be changed.

Table 4.90: FS_SetBusyLEDCallback() parameter list

Additional Information

If you intend to show any volume read/write activity, use this function to set the busy indication for the desired volume.

Example

```
#include "FS.h"

static void _cbBusyLED(U8 OnOff) {
  if (OnOff) {
    HW_SetLED();
  } else {
    HW_ClrLED();
  }
}

void MainTask(void) {
  FS_FILE * pFile;

  FS_Init();
  FS_SetBusyLEDCallback("ram:", _cbBusyLED);
  pFile = FS_FOpen("ram:\\file.txt", "w");
  FS_FClose(pFile);
}
```


4.9.24 FS_SetMemAccessCallback()

Description

Registers a function which should be called before any read and write operation to check if a data buffer can be used in 0-copy operation.

Prototype

```
void FS_SetMemAccessCallback(const char  * sVolumeName,
                             FS_MEMORY_IS_ACCESSIBLE_CALLBACK * pfIsAccessibleCallback);
```

Parameter	Description
sVolumeName	Volume name as a string.
pfIsAccessibleCallback	Pointer to a function which performs the checking.

Table 4.91: FS_SetMemAccessCallback() parameter list

Additional Information

This function is available only if the sources are compiled with the FS_SUPPORT_CHECK_MEMORY switch is set to 1. The file system operations are optimized to avoid the copying of data being written or read. Where possible, 0-copy operations are used. Only a pointer to data being written or read is passed between the file system layers. By registering a callback function an application can control whether a 0-copy operation is allowed or not. This can be useful, for example, if the HW layer uses DMA to transfer the data and the DMA controller can not access a certain memory region. In such a case the callback should return 0 to inform the file system to buffer the data internally.

Example

```
#include "FS.h"

static int _cbIsMemoryAccessible(void * p, U32 NumBytes) {
  if ((U32)p > 0x100000uL) {
    return 1;  // 0-copy allowed.
  } else {
    return 0;  // 0-copy not allowed
  }
}

void MainTask(void) {
  FS_FILE * pFile;

  FS_Init();
  FS_SetMemAccessCallback("ram:", _cbIsMemoryAccessible);
  pFile = FS_FOpen("ram:\\file.txt", "w");
  if (pFile) {
    FS_FWrite("Test\n", 5, 1, pFile);
  }
  FS_FClose(pFile);
}
```


4.9.25 FS_SetVolumeLabel()

Description

Sets a label to a specific volume.

Prototype

```
int FS_SetVolumeLabel(const char * sVolumeName,
                      char       * pVolumeLabel);
```

Parameter	Description
sVolumeName	Volume name as a string.
pVolumeLabel	Pointer to a buffer with the new volume label. NULL indicates, that the volume label should be deleted.

Table 4.92: FS_GetVolumeLabel() parameter list

Return Value

== 0: On Success.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

4.9.26 FS_TimeStampToFileTime()

Description

Converts a given timestamp to a FS_FILE_TIME structure.

Prototype

```
void FS_TimeStampToFileTime(U32           TimeStamp,
                            FS_FILETIME * pFileTime);
```

Parameter	Description
TimeStamp	Timestamp to be converted.
pFileTime	Pointer to a data structure of type FS_FILETIME to store the converted timestamp.

Table 4.93: FS_TimeStampToFileTime() parameter list

Additional Information

A TimeStamp is a packed value with the following format:

Bits	Description
0-4	Second divided by 2
5-10	Minute (0 - 59)
11-15	Hour (0 - 23)
16-20	Day of month (1 - 31)
21-24	Month (January -> 1, February -> 2, etc.)
25-31	Year offset from 1980. Add 1980 to get current year.

Table 4.94: FS_TimeStampToFileTime() - timestamp format description

4.9.27 FS_Unlock()

Description

Releases the exclusive access to file system.

Prototype

```
void FS_Unlock(void);
```

Additional information

Should be called after `FS_Lock()` to give other task access to file system. If `FS_OS_LOCKING` is not set to 1, the function does nothing.

Note: The file system API functions are multitasking safe. Explicit locking using `FS_Lock()`/`FS_Unlock()` is not required.

4.9.28 FS_UnlockVolume()

Description

Releases the exclusive access to a given volume.

Prototype

```
void FS_UnlockVolume(const char * sVolumeName);
```

Parameter	Description
sVolumeName	Volume name as a string.

Additional information

The execution of the task that calls this function is suspended until the exclusive access to file system can be granted. If FS_OS_LOCKING is not set to 2, the function does nothing.

Note: The file system API functions are multitasking safe. Explicit locking using FS_LockVolume()/FS_UnlockVolume() is not required.

4.9.29 FS_BUSY_LED_CALLBACK

Description

Callback function invoked when the LED status should be changed.

Prototype

```
typedef void (FS_BUSY_LED_CALLBACK)(U8 OnOff);
```

Parameter	Description
OnOff	LED status !=0 LED on ==0 LED off

Table 4.95: FS_BUSY_LED_CALLBACK parameter list

Additional information

Refer to *FS_SetBusyLEDCallback()* on page 142 for more information.

4.9.30 FS_MEMORY_IS_ACCESSIBLE_CALLBACK

Description

Callback function invoked at the beginning of a read or write operation to check if a 0-copy operation can be performed on the data buffer.

Prototype

```
typedef int FS_MEMORY_IS_ACCESSIBLE_CALLBACK(void * p, U32 NumBytes);
```

Parameter	Description
p	IN: Data buffer to check. OUT: ---
NumBytes	Number of bytes of data buffer to be checked.

Table 4.96: FS_MEMORY_IS_ACCESSIBLE_CALLBACK parameter list

Return value

==0 The driver can not access the data buffer directly
==1 Data buffer can be passed to device driver

Additional information

Refer to *FS_SetMemAccessCallback()* on page 143 for more information.

4.9.31 FS_CHECKDISK_ON_ERROR_CALLBACK

Description

Callback invoked when an error occurs during a disk check.

Prototype

```
typedef int FS_CHECKDISK_ON_ERROR_CALLBACK(int ErrCode, ...);
```

Parameter	Description
ErrCode	Value which indicates the type of error.

Table 4.97: FS_CHECKDISK_ON_ERROR_CALLBACK parameter list

Return value

==FS_CHECKDISK_ACTION_DO_NOT_REPAIR	Do not repair the error
==FS_CHECKDISK_ACTION_SAVE_CLUSTERS	Save lost cluster chain to file
==FS_CHECKDISK_ACTION_ABORT	Abort disk checking
==FS_CHECKDISK_ACTION_DELETE_CLUSTERS	Delete cluster chain

Additional information

Depending on the error type, additional parameters are passed to this function. They can be used in ca call to a sprinf()-like function to create a text error message. For more information see *FS_CheckDisk()* on page 118.

4.9.32 FS_CHS_ADDR

Description

This structure stores information about the position on a disk.

Prototype

```
typedef struct {
  U16 Cylinder;
  U8  Head;
  U8  Sector;
}
FS_CHS_ADDR;
```

Members	Description
Cylinder	Cylinder number.
Head	Head number.
Sector	Sector number.

Table 4.98: FS_CHS_ADDR - list of structure elements

4.9.33 FS_DISK_INFO

Description

This structure stores information about a volume.

Prototype

```
typedef struct {
  U32 NumTotalClusters;
  U32 NumFreeClusters;
  U16 SectorsPerCluster;
  U16 BytesPerSector;
  U16 NumRootDirEntries;
  U16 FSType;
  U8  IsSDFormatted;
}
FS_DISK_INFO;
```

Members	Description
NumTotalClusters	Number of clusters on the medium.
NumFreeClusters	Number of clusters that are not used
SectorsPerCluster	Number of sectors in a cluster
BytesPerSector	Number of bytes in a sector
NumRootDirEntries	Number of directory entries in the root directory. In case of FAT32 and EFS partitions, where the size of the root directory is not limited, it is always 0xFFFF.
FSType	Partition type. Can be one of: • FS_TYPE_FAT12 • FS_TYPE_FAT16 • FS_TYPE_FAT32 • FS_TYPE_EFS
IsSDFormatted	Set to 1 if the volume is formatted acc. to SD specifications.

Table 4.99: FS_DISK_INFO - list of structure elements

4.9.34 FS_FILETIME

Description

The FS_FILETIME structure represents a timestamp using individual members for the month, day, year, weekday, hour, minute, and second. This can be useful for getting or setting a timestamp of a file or directory.

Prototype

```
typedef struct {
  U16 Year;
  U16 Month;
  U16 Day;
  U16 Hour;
  U16 Minute;
  U16 Second;
} FS_FILETIME;
```

Members	Description
Year	Represents the year. The year must be greater than 1980.
Month	Represents the month, where January = 1, February = 2, etc.
Day	Represents the day of the month (1 - 31).
Hour	Represents the hour of the day (0 - 23).
Minute	Represents the minute of the hour (0 - 59).
Second	Represents the second of the minute (0 - 59).

Table 4.100: FS_FILETIME - list of structure elements

4.9.35 FS_PARTITION_INFO

Description

This structure stores information about a partition.

Prototype

```
typedef struct {
  U32          NumSectors;
  U32          StartSector;
  FS_CHS_ADDR StartAddr;
  FS_CHS_ADDR EndAddr;
  U8           Type;
  U8           IsActive;
}
FS_PARTITION_INFO;
```

Members	Description
NumSectors	Total number of sectors in the partition.
StartSector	Absolute address of the first sector in the partition.
StartAddr	Address of the first sector in the partition in CHS format.
EndAddr	Address of the last sector in the partition in CHS format.
Type	Type of the partition.
IsActive	Set to 1 if the partition is bootable.

Table 4.101: FS_PARTITION_INFO - list of structure elements

4.10 Storage layer functions

4.10.1 FS_STORAGE_Clean()

Description

Performs garbage collection on a storage medium.

Prototype

```
int FS_STORAGE_Clean(const char * sVolumeName);
```

Parameter	Description
sVolumeName	IN: Volume name. If the empty string is specified, the first device in the volume table will be used. NULL is not allowed. OUT: ---

Table 4.102: FS_STORAGE_Clean() parameter list

Return value

== 0: Storage medium cleaned.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional information

The function can be used only on storage layers managed by the file system. Typically, these are the volumes mounted on NAND flash and NOR flash devices. All the blocks/sectors which contain invalid data are erased. Depending on the storage type the function can block for a long period of time preventing the access of other tasks to file system. For the situations where this is not desired, an alternative function is provided which performs only one garbage collection step. Refer to *FS_STORAGE_CleanOne()* on page 156 for more information. The operations performed by the driver are documented in the relevant "Garbage collection" section. Not all the device drivers support this functionality. The function can be called from a different task than the task performing the file access operations.

Example

```
void CleanSample(void) {
  FS_FILE * pFile;

  //
  // Perform garbage collection on the storage medium.
  //
  FS_STORAGE_Clean("");
  //
  // The write to file is fast since no garbage collection is required.
  //
  pFile = FS_FOpen("file.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Test", 4);
    FS_FClose(pFile);
  }
}
```


4.10.2 FS_STORAGE_CleanOne()

Description

Performs a single garbage collection step on a storage medium.

Prototype

```
int FS_STORAGE_CleanOne(const char * sVolumeName,
                        int        * pMore);
```

Parameter	Description
sVolumeName	IN: Volume name. If the empty string is specified, the first device in the volume table will be used. NULL is not allowed. OUT: ---
pMore	IN: --- OUT: !=0 medium has not been cleaned completely ==0 medium is completely clean

Table 4.103: FS_STORAGE_CleanOne() parameter list

Return value

== 0: Clean operation successful.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional information

The function can be used only on storage layers managed by the file system. Typically, these are the volumes mounted on NAND flash and NOR flash devices. Usually, one block/sector which contain invalid data is erased. The operations performed by the driver are documented in the relevant "Garbage collection" section. Not all the device drivers support this functionality. The function can be called from a different task than the task performing the file access operations.

Example

```
void CleanOneSample(void) {
  FS_FILE * pFile;
  int       More;

  //
  // Perform garbage collection on the storage medium.
  //
  More = 0;
  do {
    FS_STORAGE_CleanOne("", &More);
  } while (More);
  //
  // The write to file is fast since no garbage collection is required.
  //
  pFile = FS_FOpen("file.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Test", 4);
    FS_FClose(pFile);
  }
}
```


4.10.3 FS_STORAGE_FreeSectors()

Description

Informs the driver about unused sectors.

```
nt FS_STORAGE_FreeSectors(const char * sVolumeName,
                          U32          FirstSector,
                          U32          NumSectors);
```

Parameter	Description
sVolumeName	IN: Volume name. If not specified, the first device in the volume table will be used. OUT: ---
FirstSector	Index of the first sector which is no longer used
NumSectors	Number of sectors not used anymore

Table 4.104: FS_STORAGE_FreeSectors() parameter list

Return value

==0: Sectors freed
!=0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional information

Typically called by the application to mark the data sectors as not used. The NAND and NOR driver use this information to optimize relocation of data blocks. The data of the sectors marked as not used are not copied anymore which improves the write performance. This is equivalent to trim command for SSDs (Solid State Drives).

4.10.4 FS_STORAGE_GetCleanCnt()

Description

Returns the number of clean operations which should be performed before all invalid data on the storage medium is erased.

```
int FS_STORAGE_GetCleanCnt(const char * sVolumeName,
                           U32        * pCnt);
```

Parameter	Description
sVolumeName	IN: Volume name. If not specified, the first device in the volume table will be used. OUT: ---
pCnt	IN: --- OUT: Number of clean operations

Table 4.105: FS_STORAGE_GetCleanCnt() parameter list

Return value

== 0: OK, number of clean operations returned.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

4.10.5 FS_STORAGE_GetCounters()

Description

Returns the device status.

Prototype

```
void FS_STORAGE_GetCounters(FS_STORAGE_COUNTERS * pStat);
```

Parameter	Description
pStat	IN: --- OUT: Contents of statistic counters.

Table 4.106: FS_STORAGE_GetCounters() parameter list

4.10.6 FS_STORAGE_GetDeviceInfo()

Description

Returns the device status.

Prototype

```
int FS_STORAGE_GetDeviceInfo (const char  * sVolumeName,
                              FS_DEV_INFO * pDevInfo);
```

Parameter	Description
sVolumeName	Name of the device to check.
pDeviceInfo	Pointer to a data structure of type FS_DEV_INFO.

Table 4.107: FS_STORAGE_GetDeviceInfo() parameter list

Return Value

== 0: OK
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

4.10.7 FS_STORAGE_Init()

Description

This function only initializes the driver and OS if necessary.

Prototype

```
void FS_STORAGE_Init(void);
```

Return value

The return value is the number of drivers can be used at the same time. These number of drivers is relevant for the high-level initialization function `FS_Init()`. `FS_Init()` uses these information to allocate the sector buffers which are necessary for a file system operation.

Additional information

The function initializes the storage layer of a driver. If you use `FS_STORAGE_Init()` instead of `FS_Init()`, only the storage layer functions like `FS_STORAGE_ReadSector()` or `FS_STORAGE_WriteSector()` are available. This means that the file system can be used as a pure sector read/write software. This can be useful when using the file system as a USB mass storage client driver.

4.10.8 FS_STORAGE_ReadSector()

Description

Reads a sector from a device.

Prototype

```
int FS_STORAGE_ReadSector(const char * sVolumeName,
                          const void * pData,
                          U32          SectorIndex);
```

Parameter	Description
sVolumeName	Volume name. If not specified, the first device in the volume table will be used.
pData	Pointer to a buffer where the read data will be stored.
SectorIndex	Index of the sector from which data should be read.

Table 4.108: FS_STORAGE_ReadSector() parameter list

Return value

== 0: Sector data read.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

4.10.9 FS_STORAGE_ReadSectors()

Description

Reads multiple sectors from a device.

Prototype

```
int FS_STORAGE_ReadSectors(const char * sVolumeName,
                           void       * pData,
                           U32          FirstSector,
                           U32          NumSectors);
```

Parameter	Description
sVolumeName	Volume name. If not specified, the first device in the volume table will be used.
pData	Pointer to a data buffer where the read data should be stored.
FirstSector	First sector to read.
NumSectors	Number of sectors which should be read.

Table 4.109: FS_STORAGE_ReadSectors() parameter list

Return value

== 0: Sector data read.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

4.10.10 FS_STORAGE_RefreshSectors()

Description

Rewrites one or more sectors with the original data.

Prototype

```
int FS_STORAGE_RefreshSectors(const char * sVolumeName,
                              U32          FirstSector,
                              U32          NumSectors,
                              void       * pBuffer,
                              U32          NumBytes);
```

Parameter	Description
sVolumeName	IN: Volume name. If not specified, the first device in the volume table will be used. OUT: ---
FirstSector	Index of the first sector to be refreshed
NumSectors	Number of sectors to be refreshed
pBuffer	Buffer to be used as storage for the read sectors. Must be at least one sector large.
NumBytes	Number of bytes in the buffer.

Table 4.110: FS_STORAGE_RefreshSectors() parameter list

Return value

== 0: Sector data refreshed.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional information

Typically called to prevent the loss of data when the sector data is not modified for long periods of time. This can be useful to handle read disturbs of NAND flashes or errors cause by the data reaching the retention limit. Refer to *Read disturbs* on page 230 for more information.

Example

```
static U32 _aBuffer[2048 / 4];

void SectorRefresSample(void) {
  int r;
  //
  // Refresh the first 100 sectors of the storage medium.
  //
  r = FS_STORAGE_RefreshSectors("", 0, 100, _aBuffer, sizeof(_aBuffer));
  if (r) {
    printf("Sectors 0-99 have been refreshed.\n");
  }
}
```


4.10.11 FS_STORAGE_ResetCounters()

Description

Sets the statistic counters of the storage layer to 0.

Prototype

```
void FS_STORAGE_ResetCounters(void);
```


4.10.12 FS_STORAGE_Sync()

Description

Writes cached data to the storage medium and sends a command to the driver to finalize all pending tasks.

Prototype

```
void FS_STORAGE_Sync(const char * sVolumeName);
```

Parameter	Description
sVolumeName	Volume name. If not specified, the first device in the volume table will be used.

Table 4.111: FS_STORAGE_Sync() parameter list

4.10.13 FS_STORAGE_SyncSectors()

Description

Writes cached sector data to storage medium.

```
nt FS_STORAGE_SyncSectors(const char * sVolumeName,
                          U32          FirstSector,
                          U32          NumSectors);
```

Parameter	Description
sVolumeName	IN: Volume name. If not specified, the first device in the volume table will be used. OUT: ---
FirstSector	Index of the first sector to be synchronized
NumSectors	Number of sectors to be synchronized

Table 4.112: FS_STORAGE_SyncSectors() parameter list

Return value

==0: Sectors synchronized
!=0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional information

When called on a RAID volume it updates the contents of the spcified sectors on the secondary storage with the contents of the corresponding sectors on the primary storage if the sector data is different.

4.10.14 FS_STORAGE_Unmount()

Description

Unmounts a given volume at the driver layer. The function also sends an unmount command to the driver, and marks the volume as unmounted and uninitialized.

Prototype

```
void FS_STORAGE_Unmount(const char * sVolumeName);
```

Parameter	Description
sVolumeName	sVolumeName is the name of a volume. If not specified, the first device in the volume table will be used.

Table 4.113: FS_STORAGE_Unmount() parameter list

4.10.15 FS_STORAGE_WriteSector()

Description

Writes one sector to a device.

Prototype

```
int FS_STORAGE_WriteSector(const char * sVolumeName,
                           const void * pData,
                           U32          SectorIndex);
```

Parameter	Description
sVolumeName	Volume name. If not specified, the first device in the volume table will be used.
pData	Pointer to the data which should be written to the device.
SectorIndex	Index of the sector to which data should be written.

Table 4.114: FS_STORAGE_WriteSector() parameter list

Return value

== 0: Sector data written.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

4.10.16 FS_STORAGE_WriteSectors()

Description

Writes multiple sectors to a device.

Prototype

```
int FS_STORAGE_WriteSectors (const char * sVolumeName,
                             const void * pData,
                             U32          FirstSector,
                             U32          NumSectors)
```

Parameter	Description
sVolumeName	Volume name. If not specified, the first device in the volume table will be used.
pData	Pointer to the data which should be written to the device.
FirstSector	Start sector of the write operation.
NumSectors	Number of sectors that should be written.

Table 4.115: FS_STORAGE_WriteSectors() parameter list

Return value

== 0: Sector data written.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

4.10.17 Structure FS_STORAGE_COUNTERS

Description

This structure describes the statistic counters of the storage layer.

Prototype

```
typedef struct {
  U32 ReadOperationCnt;
  U32 ReadSectorCnt;
  U32 ReadSectorCachedCnt;
  U32 WriteOperationCnt;
  U32 WriteSectorCnt;
  U32 WriteSectorCntCleaned;
  U32 ReadSectorCntMan;
  U32 ReadSectorCntDir;
  U32 WriteSectorCntMan;
  U32 WriteSectorCntDir;
} FS_STORAGE_COUNTERS;
```

Members	Description
ReadOperationCnt	Total number of read operations.
ReadSectorCnt	Total number of sectors read.
ReadSectorCachedCnt	Number of times a sector was found in cache.
WriteOperationCnt	Total number of write operations.
WriteSectorCnt	Total number of sectors written.
WriteSectorCntCleaned	Number of sectors written by the cache module to storage in order to make room for other data.
ReadSectorCntMan	Number of management sectors read (before cache).
ReadSectorCntDir	Number of directory sectors (which store directory entries) read (before cache).
WriteSectorCntMan	Number of management sectors written (before cache).
WriteSectorCntDir	Number of directory sectors (which store directory entries) written (before cache).

Table 4.116: FS_STORAGE_COUNTERS - list of structure elements

4.11 FAT related functions

4.11.1 FS_FAT_GrowRootDir()

Description

Enlarges the default size of the root directory of a FAT32 volume.

Prototype

```
U32 FS_FAT_GrowRootDir (const char * sVolumeName, U32 NumAddEntries);
```

Parameter	Description
pVolumeName	Name of the device.
NumAddEntries	Numbers of directories entries to be added.

Table 4.117: FS_FAT_GrowRootDir() parameter list

Return value

>= 0: Number of entries added.
== 0: Clusters after root directory are not free.
== 0xFFFFFFFF: An error has occurred.

Additional Information

This function has to be called after formatting the volume. If the function is not called after format or called for a FAT12/16 volume the function will fail. In opposite to FAT12 and FAT16 which have a fixed root directory size, the root directory of a FAT32 formatted device can be of variable size. By default, one cluster is reserved for the root directory entries. Therefore, it can speed up performance to reserve additional clusters for root directory entries after formatting the medium.

4.11.2 FS_FormatSD()

Description

Performs a high-level format of a device according to the SD Specification - File system specification.

Prototype

```
int FS_FormatSD (const char * pVolumeName);
```

Parameter	Description
pVolumeName	Name of the device to format.

Table 4.118: FS_FormatSD() parameter list

Return value

== 0: Format was successful.
!= 0: Error code indicating the failure reason.
Refer to *FS_ErrorNo2Text()* on page 177.

Additional Information

For further information refer to SD Specification - Part 2 - File System Specification (May 9, 2006, *www.sdcard.org*).

4.11.3 FS_FAT_SupportLFN()

Description

Adds long file name support to the file system.

Prototype

```
void FS_FAT_SupportLFN(void);
```

Additional Information

The FAT file system was not designed for long file name (LFN) support, limiting names to twelve characters (8.3). LFN support may be added to any of the FAT file systems, but there are legal issues that must be settled with Microsoft before end applications make use of this feature. Long file names are inherent to this proprietary file system relieving it of any legal issues.

Note: The LFN package is required to support long file names.

4.11.4 FS_FAT_DisableLFN()

Description

Disables the support for long file names for the FAT file system.

Prototype

```
void FS_DisableLFN(void);
```


4.12 Error handling functions

4.12.1 FS_ClearErr()

Description

Clears the error status of a file.

Prototype

```
void FS_ClearErr(FS_FILE * pFile);
```

Parameter	Description
pFile	Pointer to a data structure of type FS_FILE.

Table 4.119: FS_ClearErr() parameter list

Additional Information

This routine should be called after you have detected an error so that you can check for success of the next file operations.

Example

```
void MainTask(void) {
  FS_FILE *pFile;
  int Err;

  pFile = FS_FOpen("test.txt", "r");
  if (pFile != 0) {
    Err = FS_FError(pFile);
    if (Err != FS_ERR_OK) {
      FS_ClearErr(pFile);
    }
    FS_FClose(pFile);
  }
}
```


4.12.2 FS_ErrorNo2Text()

Description

Retrieves text for a given error code.

Prototype

```
const char * FS_ErrorNo2Text (int ErrCode);
```

Parameter	Description
ErrCode	The returned error code.

Table 4.120: FS_ErrorNo2Text() parameter list

Return value

Returns the string according to the ErrCode.

Additional information

The following error codes are available:

Code	Description
FS_ERRCODE_OK	No error.
FS_ERRCODE_EOF	End-of-file has been reached.
FS_ERRCODE_VOLUME_FULL	Unable to write data because there is no more space on the media.
FS_ERRCODE_INVALID_PARA	An emFile function has been called with an illegal parameter.
FS_ERRCODE_WRITE_ONLY_FILE	A read operation has been made on a file open for writing only.
FS_ERRCODE_READ_ONLY_FILE	A write operation has been made on a file open for reading only.
FS_ERRCODE_READ_FAILURE	An error occurred during a read operation.
FS_ERRCODE_WRITE_FAILURE	An error occurred during a write operation.
FS_ERRCODE_FILE_IS_OPEN	Trying to delete an opened file.
FS_ERRCODE_PATH_NOT_FOUND	Path to file or directory not found.
FS_ERRCODE_FILE_DIR_EXISTS	A file or directory with the same name already exists.
FS_ERRCODE_NOT_A_FILE	The API function operates only on files.
FS_ERRCODE_TOO_MANY_FILES_OPEN	Trying to open more files at once than the trial version allows.
FS_ERRCODE_INVALID_FILE_HANDLE	The file handle is no longer valid.
FS_ERRCODE_VOLUME_NOT_FOUND	The volume name specified in a path is does not exist.
FS_ERRCODE_READ_ONLY_VOLUME	Trying to write to a volume mounted in read-only mode.
FS_ERRCODE_VOLUME_NOT_MOUNTED	Trying access a volume which is not mounted.
FS_ERRCODE_NOT_A_DIR	The API function operates only on directories.
FS_ERRCODE_FILE_DIR_NOT_FOUND	File or directory not found.

Code	Description
FS_ERRCODE_NOT_SUPPORTED	Functionality not supported by the active file system type.
FS_ERRCODE_CLUSTER_NOT_FREE	Trying to allocate a cluster which is not free.
FS_ERRCODE_INVALID_CLUSTER_CHAIN	Detected a shorter than expected cluster chain.
FS_ERRCODE_STORAGE_NOT_PRESENT	Trying to access a removable storage which is not inserted.
FS_ERRCODE_BUFFER_NOT_AVAILABLE	No more sector buffers available.
FS_ERRCODE_STORAGE_TOO_SMALL	Not enough sectors on the storage medium.
FS_ERRCODE_STORAGE_NOT_READY	Storage device can not be accessed.
FS_ERRCODE_BUFFER_TOO_SMALL	Sector buffer smaller than sector size of storage medium.
FS_ERRCODE_INVALID_FS_FORMAT	Storage medium is not formatted or the format is not valid.
FS_ERRCODE_INVALID_FS_TYPE	The type of file system is invalid or not configured.
FS_ERRCODE_FILENAME_TOO_LONG	The name of the file is too long.
FS_ERRCODE_VERIFY_FAILURE	Data verification failure.
FS_ERRCODE_DIR_NOT_EMPTY	Trying to delete a directory which is not empty.
FS_ERRCODE_IOCTL_FAILURE	Error while executing a driver control command.
FS_ERRCODE_INVALID_MBR	Invalid information in the Master Boot Record.

Example

```
void MainTask(void) {
  FS_FILE *pFile;
  int      ErrCode;

  ErrCode = FS_FOpenEx("test.txt", "r", &pFile);
  if (ErrCode) {
    FS_X_Log("Open file error: ");
    FS_X_Log(FS_ErrorNo2Text(ErrCode));
  }
}
```


4.12.3 FS_FEof()

Description

Tests for end-of-file on a given file pointer.

Prototype

```
int FS_FEof (FS_FILE * pFile);
```

Parameter	Description
pFile	Pointer to a data structure of type FS_FILE.

Table 4.121: FS_FEof() parameter list

Return value

== 0: If the end of file has not been reached.
== 1: If the end of file has been reached.

Additional Information

The FS_FEof function determines whether the end of a given file pointer has been reached. When end of file is reached, read operations return an end-of-file indicator until the file pointer is closed or until FS_FSeek, or FS_ClearErr is called against it.

Example

```
int ReadFile(FS_File * pFile, char * pBuffer, int NumBytes) {
  FS_FILE * pFile;
  char      acBuffer[100];
  char      acLog[100];
  int       Count;
  int       Total;
  I16       Error;

  Total = 0;
  pFile = FS_FOpen("default.txt", "r");
  if (pFile == NULL) {
    FS_X_ErrorOut("Could not open file.");'
  }
  /* Cycle until end of file reached: */
  while (!FS_FEof(pFile)) {
    Count = FS_Read(pFile, &acBuffer[0], sizeof(acBuffer));
    Error = FS_FError(pFile);
    if (Error) {
      sprintf(acLog, "Could not read from file:\nReason = %s",
              FS_ErrorNo2Text(Error));
      FS_X_ErrorOut(acLog);
      break;
    }
    /* Total up actual bytes read */
    Total += Count;
  }
  sprintf(acLog, "Number of read bytes = %d\n", Total);
  FS_X_Log(acLog);
  FS_FClose(pFile);
}
```


4.12.4 FS_FError()

Description

Returns the current error status of a file.

Prototype

```
I16 FS_FError (FS_FILE * pFile);
```

Parameter	Description
pFile	Pointer to a data structure of type FS_FILE.

Table 4.122: FS_FError() parameter list

Return value

FS_ERR_OK if no errors.
A value not equal to FS_ERR_OK if a file operation caused an error.

Additional Information

The return value is not FS_ERR_OK only when a file operation caused an error and the error was not cleared by calling FS_ClearErr() or any other operation that clears the previous error status.

Example

```
void MainTask(void) {
  FS_FILE *pFile;

  pFile = FS_FOpen("test.txt", "r");
  if (pFile != 0) {
    I16 Err;
    Err = FS_FError(pFile);
    FS_FClose(pFile);
  }
}
```


4.13 Obsolete functions

This section contains reference information for obsolete functions.

4.13.1 FS_CloseDir()

Description

Closes a directory referred to by the parameter pDir.

Prototype

```
int FS_CloseDir (FS_DIR * pDir);
```

Parameter	Description
pDir	Pointer to a data structure of type FS_DIR.

Table 4.123: FS_CloseDir() parameter list

Return Value

== 0: If the directory was successfully closed.
== -1: In case of any error.

Example

```
void MainTask(void) {
  FS_DIR    *pDir;
  FS_DIRENT *pDirEnt;

  pDir = FS_OpenDir("");          /* Open the root directory of default device */
  if (pDir) {
    do {
      char acDirName[20];
      pDirEnt = FS_ReadDir(pDir);
      FS_DirEnt2Name(pDirEnt, acDirName); /* Get name of the current DirEntry */
      if ((void*)pDirEnt == NULL) {
        break;                                /* No more files or directories */
      }
      sprintf(_acBuffer,"  %s\n", acName);
      FS_X_Log(_acBuffer);
    } while (1);
    FS_CloseDir(pDir);
  } else {
    FS_X_ErrorOut("Unable to open directory\n");
}
```


4.13.2 FS_ConfigUpdateDirOnWrite()

Description

Configures if the file system should update the directory entry on date write.

Prototype

```
void FS_ConfigUpdateDirOnWrite(char OnOff);
```

Parameter	Description
OnOff	==1 means enable update directory after write (Default). ==0 means do not update directory.

Table 4.124: FS_ConfigUpdateDirOnWrite() parameter list

Additional Information

Use the FS_SetFileWriteMode() function instead.

4.13.3 FS_DirEnt2Attr()

Description

Retrieves the attributes of the directory entry referred to by pDirEnt.

Prototype

```
void FS_DirEnt2Attr (FS_DIRENT * pDirEnt,
                     U8 *        pAttr);
```

Parameter	Description
pDirEnt	Pointer to a directory entry, read by FS_ReadDir().
pAttr	Pointer to U8 variable in which the attributes should be stored.

Table 4.125: FS_DirEnt2Attr() parameter list

Additional Information

These attributes are available:

Parameter	Description
FS_ATTR_DIRECTORY	pDirEnt is a directory.
FS_ATTR_ARCHIVE	pDirEnt has the ARCHIVE attribute set.
FS_ATTR_READ_ONLY	pDirEnt has the READ ONLY attribute set.
FS_ATTR_HIDDEN	pDirEnt has the HIDDEN attribute set.
SYSTEM	pDirEnt has the SYSTEM attribute set.

Table 4.126: FS_DirEnt2Attr() - list of possible attributes

pDirEnt should point to a valid FS_DIRENT structure. FS_DirEnt2Attr() checks if the pointer is valid. To get a valid pointer, FS_ReadDir() should be called before using FS_DirEnt2Attr(). Refer to *FS_ReadDir()* on page 191 for more information.

Example

```
void MainTask(void) {
  FS_DIR    *pDir;
  FS_DIRENT *pDirEnt;
  char acBuffer[200];
  pDir = FS_OpenDir("");      /* Open root directory of default device */
  if (pDir) {
    do {
      char acName[20];
      U8 Attr;
      pDirEnt = FS_ReadDir(pDir);
      FS_DirEnt2Name(pDirEnt, acName);
      FS_DirEnt2Attr(pDirEnt, &Attr);
      if ((void*)pDirEnt == NULL) {
        break;             /* No more files */
      }
      sprintf(_acBuffer,"  %s %s Attributes: %s%s%s%s\n", acName,
                        (Attr & FS_ATTR_DIRECTORY) ? "(Dir)" : "     ",
                        (Attr & FS_ATTR_ARCHIVE)   ? "A"     : "-",
                        (Attr & FS_ATTR_READ_ONLY) ? "R"     : "-",
                        (Attr & FS_ATTR_HIDDEN)    ? "H"     : "-",
                        (Attr & FS_ATTR_SYSTEM)    ? "S"     : "-");
      FS_X_Log(acBuffer);
    } while (1);
    FS_CloseDir(pDir);
  } else {
    FS_X_ErrorOut("Unable to open directory\n");
  }
}
```


4.13.4 FS_DirEnt2Name()

Description

Retrieves the name of the directory entry referred to by pDirEnt.

Prototype

```
void FS_DirEnt2Name (FS_DIRENT * pDirEnt,
                     char *      pBuffer);
```

Parameter	Description
pDirEnt	Pointer to a directory entry, read by FS_ReadDir().
pBuffer	Pointer to the buffer that will receive the text.

Table 4.127: FS_DirEnt2Name() parameter list

Additional Information

If pDirEnt and pBuffer are valid, the name of the directory is copied to the buffer that pBuffer points to. Otherwise pBuffer is NULL.
pDirEnt should point to a valid FS_DIRENT structure. FS_DirEnt2Name() checks if the pointers are valid. To get a valid pointer, FS_ReadDir() should be called before using FS_DirEnt2Name(), otherwise pBuffer is NULL. Refer to *FS_ReadDir()* on page 191 for more information.

Example

```
void MainTask(void) {
  char acDirName[20];
  FS_DIR    *pDir ;
  FS_DIRENT *pDirEnt ;

  pDir = FS_OpenDir("");      /* Open root directory of default device */
  pDirEnt = FS_ReadDir(pDir); /* Read the first directory entry */
  FS_DirEnt2Name(pDirEnt, acDirName);
  FS_X_Log(acDirName);
}
```


4.13.5 FS_DirEnt2Size()

Description

Returns the size in bytes of the directory entry referred to pDirEnt.

Prototype

```
U32 FS_DirEnt2Size (FS_DIRENT * pDirEnt);
```

Parameter	Description
pDirEnt	Pointer to a directory entry, read by FS_ReadDir().

Table 4.128: FS_DirEnt2Size() parameter list

Return value

File size in bytes.
0 in case of any error.

Additional Information

If pDirEnt is valid, the size of the directory entry will be returned. Otherwise the return value is 0.
pDirEnt should point to a valid FS_DIRENT structure. FS_DirEnt2Name() checks if the pointers are valid. To get a valid pointer, FS_ReadDir() should be called before using FS_DirEnt2Size(). Refer to *FS_ReadDir()* on page 191 for more information.

Example

```
void MainTask(void) {
  U32    FileSize;
  FS_DIR    *pDir ;
  FS_DIRENT *pDirEnt ;

  pDir = FS_OpenDir("");       /* Open root directory of default device */
  pDirEnt = FS_ReadDir(pDir); /* Read the first directory entry */
  FileSize = FS_DirEnt2Size(pDirEnt);
  if (FileSize) {
    char ac[50] ;
    sprintf(ac, "File size = %lu\n", FileSize);
    FS_X_Log(ac) ;
}
```


4.13.6 FS_DirEnt2Time()

Description

Returns the timestamp of the directory entry referred to by pDirEnt.

Prototype

```
U32 FS_DirEnt2Size (FS_DIRENT * pDirEnt);
```

Parameter	Description
pDirEnt	Pointer to a directory entry, read by FS_ReadDir().

Table 4.129: FS_DirEnt2Time() parameter list

Return value

The timestamp of the current directory entry.

Additional Information

If pDirEnt is valid, the timestamp of the directory entry will be returned. Otherwise, the return value is 0.
pDirEnt should point to a valid FS_DIRENT structure. FS_DirEnt2Name() checks if the pointer is valid. To get a valid pointer, FS_ReadDir() should be called before using FS_DirEnt2Size(). Refer to *FS_ReadDir()* on page 191 for more information.
A timestamp is a packed value with the following format.

Bits	Description
0-4	Second divided by 2
5-10	Minute (0 - 59)
11-15	Hour (0-23)
16-20	Day of month (1-31)
21-24	Month (January -> 1, February -> 2, etc.)
25-31	Year offset from 1980. Add 1980 to get current year.

Table 4.130: FS_DirEnt2Time() - timestamp format description

To convert a timestamp to a FS_FILETIME structure, use the function FS_TimeStampToFileTime().

Example

```
void MainTask(void) {
  U32      TimeStamp;
  FS_DIR    * pDir ;
  FS_DIRENT * pDirEnt ;
  char        acLog[100] ;
  char        acFileName[40];
  FS_FILETIME FileTime;

  pDir      = FS_OpenDir("");      /* Open root directory of default device */
  pDirEnt   = FS_ReadDir(pDir);    /* Read the first directory entry */
  FS_DirEnt2Name(pDirEnt, &acFileName[0]);
  TimeStamp = FS_DirEnt2Time(pDirEnt);
  FS_TimeStampToFileTime(TimeStamp, &FileTime);
  sprintf(ac, "File time of %s: %d-.2d-%.2d %.2d:%.2d:%.2d",
          acFileName,
          FileTime.Year, FileTime.Month,  FileTime.Day,
          FileTime.Hour, FileTime.Minute, FileTime.Second);
  FS_X_Log(ac);
}
```


4.13.7 FS_GetDeviceInfo()

Description

Returns the device status.

Prototype

```
int FS_GetDeviceInfo(const char  * sVolumeName,
                     FS_DEV_INFO * pDevInfo);
```

Parameter	Description
sVolumeName	Name of the device to check.
pDeviceInfo	Pointer to a data structure of type FS_DEV_INFO.

Table 4.131: FS_GetDeviceInfo() parameter list

Additional information

This function is obsolete. Use instead *FS_STORAGE_GetDeviceInfo()* on page 160.

Return Value

==0: Ok
==-1: Device is not ready or a general error has occurred.

4.13.8 FS_GetNumFiles()

Description

Returns the number of files in a directory opened by `FS_OpenDir()`.

Prototype

```
U32 FS_GetNumFiles (FS_DIR * pDir);
```

Parameter	Description
pDir	Pointer to a `FS_FILE` data structure.

Table 4.132: FS_GetNumFiles() parameter list

Return value

Number of files in a directory.
0xFFFFFFFF as return value indicates an error.

Additional Information

If `pDir` is valid, the number of files in the directory will be returned. To get a valid pointer, `FS_OpenDir()` should be called before using `FS_GetNumFiles()`. Refer to *FS_OpenDir()* on page 190 for more information.

Example

```
void NumFilesInDirectory(void) {
  U32    NumFilesInDir;
  FS_DIR    *pDir ;

  pDir = FS_OpenDir("");      /* Open root directory of default device */
  NumFilesInDir = FS_GetNumFiles(pDir);
  if (NumFilesInDir) {
    char ac[50] ;
    sprintf(ac, "NumFilesInDir = %lu\n", NumFilesInDir);
    FS_X_Log(ac) ;
  }
}
```


4.13.9 FS_InitStorage()

Description

This function only initializes the driver and OS if necessary.

Prototype

```
void FS_InitStorage (void);
```

Return value

The return value indicates the caller how many drivers can be used at the same time. The function will accordingly allocate the sector buffers that are necessary for a file system operation.

Additional information

If `FS_InitStorage()` is used to initialize a driver only the hardware layer functions `FS_ReadSector()`, `FS_WriteSector()`, and `FS_GetDeviceInfo()` are available.

This function is obsolete. Use instead *FS_STORAGE_Init()* on page 161.

4.13.10 FS_OpenDir()

Description

Opens an existing directory for reading.

Prototype

```
FS_DIR *FS_OpenDir (const char * pDirname);
```

Parameter	Description
pDirName	Fully qualified directory name.

Table 4.133: FS_OpenDir() parameter list

Return value

Returns the address of an FS_DIR data structure if the directory was opened.
In case of any error the return value is 0.

Additional Information

A fully qualified directory name looks like:

```
[DevName:[UnitNum:]][DirPathList]DirectoryName
```

where:

- DevName is the name of a device, for example "ram" or "mmc". If not specified, the first device in the device table will be used.
  UnitNum is the number for the unit of the device. If not specified, unit 0 will be used. Note that it is not allowed to specify UnitNum if DevName has not been specified.
- DirPathList is a complete path to an existing subdirectory. The path must start and end with a '\' character. Directory names in the path are separated by '\'. If DirPathList is not specified, the root directory on the device will be used.
- DirectoryName and all other directory names have to follow the standard FAT naming conventions (for example 8.3 notation), if support for long file names is not enabled.

To open the root directory, simply use an empty string for pDirName.

Example

```
FS_DIR *pDir;

void FSTask1(void) {
  /* Open directory test - default driver on default device */
  pDir = FS_OpenDir("test");
}

void FSTask2(void) {
  /* Open root directory - RAM device driver on default device */
  pDir = FS_OpenDir("ram:");
}
```


4.13.11 FS_ReadDir()

Description

Reads next directory entry in directory specified by pDir.

Prototype

```
FS_DIRENT *FS_ReadDir (FS_DIR * pDir);
```

Parameter	Description
pDir	Pointer to an opened directory.

Table 4.134: FS_ReadDir() parameter list

Return value

Returns a pointer to a directory entry.
If there are no more entries in the directory or in case of any error, 0 is returned.

Example

Refer to *FS_CloseDir()* on page 181.

4.13.12 FS_ReadSector()

Description

Reads a sector from a device.

Prototype

```
int FS_ReadSector(const char * sVolumeName,
                  const void * pData,
                  U32          SectorIndex);
```

Parameter	Description
sVolumeName	Volume name.
pData	Pointer to a data buffer where the read data should be stored.
SectorIndex	Index of the sector from which data should be read.

Table 4.135: FS_ReadSector() parameter list

Return value

== 0: On success
!= 0: On error

Additional information

This function is obsolete. Use instead *FS_STORAGE_ReadSector()* on page 162.

4.13.12.1FS_RewindDir()

Description

Sets the current pointer for reading a directory entry to the first entry in the directory.

Prototype

```
void FS_RewindDir (FS_DIR * pDir);
```

Parameter	Description
pDir	Pointer to directory structure.

Table 4.136: FS_RewindDir() parameter list

Example

```
void MainTask(void) {
  FS_DIR    *pDir;
  FS_DIRENT *pDirEnt;
  char acDirName[20];

  pDir = FS_OpenDir("");          /* Open the root directory of default device */
  if (pDir) {
    do {
      char acDirName[20];
      pDirEnt = FS_ReadDir(pDir);
      FS_DirEnt2Name(pDirEnt, acDirName);  /* Get name of the current DirEntry */
      if ((void*)pDirEnt == NULL) {
        break;                                  /* No more files or directories */
      }
      sprintf(_acBuffer,"  %s\n", acName);
      FS_X_Log(_acBuffer);
    } while (1);
    /* rewind to 1st entry */
    FS_RewindDir(dirp);
    /* display directory  again */
    do {
      pDirEnt = FS_ReadDir(pDir);
      FS_DirEnt2Name(pDirEnt, acDirName); /* Get name of the current DirEntry */
      if ((void*)pDirEnt == NULL) {
        break;                                  /* No more files or directories */
      }
      sprintf(_acBuffer,"  %s\n", acName);
      FS_X_Log(_acBuffer);
    } while (1);
    FS_CloseDir(pDir);
  }
  else {
    FS_X_ErrorOut("Unable to open directory\n");
  }
}
```


4.13.13 FS_UnmountLL()

Description

Unmounts a given volume at driver layer. Sends an unmount command to the driver, marks the volume as unmounted and uninitialized.

Prototype

```
void FS_Unmount (const char * sVolumeName);
```

Parameter	Description
sVolumeName	sVolumeName is the name of a volume. If not specified, the first device in the volume table will be used.

Table 4.137: FS_UnmountLL() parameter list

Additional information

This function is obsolete. Use instead *FS_STORAGE_Init()* on page 161.

4.13.14 FS_WriteSector()

Description

Writes a sector to a device.

Prototype

```
int FS_WriteSector(const char * sVolumeName,
                   const void * pData,
                   U32          SectorIndex);
```

Parameter	Description
sVolumeName	Volume name.
pData	Pointer to the data which should be written to the device.
SectorIndex	Index of the sector to which data should be written.

Table 4.138: FS_WriteSector() parameter list

Return value

== 0: On success
!= 0: On error

Additional information

This function is obsolete. Use instead FS_STORAGE_Init() on page 104.

Chapter 5

Optimizing performance - Caching and buffering

This chapter gives an introduction into emFile’s cache handling. Furthermore, it contains the function description and an example.

5.1 Introduction

A cache is a storage area where frequently used data can be stored for fast access. In many cases, this can enhance the average execution time. Applications which do not use a cache data will always be read from the storage medium even if it has been used before. A cache stores accessed and processed data. If the data should be processed again, it will be copied out of the cache instead of refetching it from the storage medium. This is called a "hit". When the data is not present in the cache and must be read from the storage it is called a "miss".

Write cache and journaling

Do not use a write cache when the journaling is enabled. The journaling will not work properly if any form of write cache is configured. More detailed information can be found in the section *Journaling and write caching* on page 512.

5.2 Types of caches

emFile supports the usage of different cache modules as listed in the following table:

Cache module	Description
FS_CACHE_ALL	This module is a pure read cache. All sectors that are read from a volume are cached. This module does not need to be configured.
FS_CACHE_MAN	This module is also a pure read cache. In contrast to the FS_CACHE_ALL, this module does only cache the management sector of a file system (for example, the FAT sectors).
FS_CACHE_RW	FS_CACHE_RW is a configurable cache module. This module can be either used as read, write or as read/write cache. Additionally, the sectors that should be cached are also configurable.
FS_CACHE_RW_QUOTA	FS_CACHE_RW_QUOTA is a configurable cache module. This module can be either used as read, write or as read/write cache.
FS_CACHE_MULTI_WAY	Configurable cache module which can be used as read, write or read/write cache. The associativity level is also configurable.

Table 5.1: Cache types

5.3 Cache API functions

The following functions are required to enable, configure and control the emFile cache modules:

Function	Description
FS_AssignCache()	Adds a cache to a specific device.
FS_CACHE_Clean()	Cleans the caches and writes dirty sectors to the volume.
FS_CACHE_Invalidate()	Removes all the entries from the cache.
FS_CACHE_SetAssocLevel()	Sets the associativity level of a FS_CACHE_MULTI_WAY cache module.
FS_CACHE_SetMode()	Sets the mode for the cache.
FS_CACHE_SetQuota()	Sets the quotas for the different sector types in the FS_CACHE_RW_QUOTA cache module.

Table 5.2: emFile cache functions

5.3.1 FS_AssignCache()

Description

Adds a cache to a specific volume.

Prototype

```
I32 FS_AssignCache(const char    * pVolumeName,
                   void          * pCacheData,
                   U32             NumBytes
                   FS_INIT_CACHE * pfInit);
```

Parameter	Description
pVolumeName	Name of the volume for which the cache should be enabled/disabled. If not specified, the first volume will be used.
pCacheData	Pointer to a buffer that should be used as cache.
NumBytes	Size of the specified buffer in bytes.
pfInit	Pointer to the initialization function of the desired cache module. The following values can be used: FS_CACHE_ALL FS_CACHE_MAN FS_CACHE_RW FS_CACHE_RW_QUOTA FS_CACHE_MULTI_WAY

Table 5.3: FS_AssignCache() parameter list

Return value

> 0: Buffer is used as cache for the specified device.
== 0: Buffer cannot be used as cache for this device.

Additional Information

To disable the cache for a specific device, call `FS_AssignCache()` with `NumBytes set to 0`. In this case the return value will be 0.
There are four different available cache modules that can be assigned to a specific device. These modules are the following:

Cache module	Description
FS_CACHE_ALL	This module is a pure read cache. All sectors that are read from a volume are cached. This module does not need to be configured. Caching is enabled right after calling `FS_AssignCache()`.
FS_CACHE_MAN	This module is also a pure read cache. In contrast to the `FS_CACHE_ALL`, this module does only cache the management sector of a file system (for example FAT sectors). Caching is enabled right after calling `FS_AssignCache()`.
FS_CACHE_RW	`FS_CACHE_RW` is a configurable cache module. This module can be either used as read, write or as read/write cache. Additionally, the sectors that should be cached are also configurable. Refer to `FS_CACHE_SetMode()` to configure the `FS_CACHE_RW` module.

Cache module	Description
FS_CACHE_RW_QUOTA	FS_CACHE_RW_QUOTA is a configurable cache module. This module can be either used as read, write or as read/write cache. To configure the cache module properly, FS_CACHE_SetMode() and FS_CACHE_SetQuota need to be called. Otherwise the functionality inside the cache is disabled.
FS_CACHE_MULTI_WAY	It is a configurable cache module which can be used as read, write or read/write cache. The associativity level is also configurable and is by default 2.

The function expects the size of the cache buffer to be specified in bytes. A part of this buffer is used by the cache module as management data. The following defines can help an application allocate a cache buffer large enough to store a given number of sectors:

Define	Description
FS_SIZEOF_CACHE_ALL()	Computes the size in bytes of a FS_CACHE_ALL cache buffer.
FS_SIZEOF_CACHE_MAN()	Computes the size in bytes of a FS_CACHE_MAN cache buffer.
FS_SIZEOF_CACHE_RW()	Computes the size in bytes of a FS_CACHE_RW cache buffer.
FS_SIZEOF_CACHE_RW_QUOTA()	Computes the size in bytes of a FS_CACHE_QUOTA cache buffer.
FS_SIZEOF_CACHE_MULTI_WAY()	Computes the size in bytes of a FS_CACHE_MULTI_WAY cache buffer.

All the above macros take the following two arguments:

Parameter	Description
NumSectors	The number of sectors to store in the cache.
SectorSize	Size of a logical sector in bytes.

Table 5.4: Cache size parameter list

Example

```
#include "FS.h"

#define CACHE_SIZE   FS_SIZEOF_CACHE_ALL(200, 512)

static char _acCache[CACHE_SIZE]; // Allocate RAM for cache buffer

void Function(void) {
  //
  // Assign a cache to the first available device
  //
  FS_AssignCache("", _acCache, sizeof(_acCache), FS_CACHE_ALL);
  //
  // Do some work
  //
  DoWork();
  //
  // Disable the read cache
  //
  FS_AssignCache("", 0, 0, 0);
}
```


5.3.2 FS_CACHE_Clean()

Description

Cleans a cache if sectors that are marked as dirty need to be written to the device.

Prototype

```
void FS_CACHE_Clean(const char * pVolumeName);
```

Parameter	Description
pVolumeName	Name of the volume for which the cache should be cleaned. If not specified, the first volume will be used.

Table 5.5: FS_CACHE_Clean() parameter list

Additional Information

Because only write or read/write caches need to be cleaned, this function should be called for volumes where the `FS_CACHE_RW` module is assigned. The other cache modules ignore the cache clean operation.
Cleaning of the cache is also performed when the volume is unmounted through `FS_Unmount()` or disabling or reassigning the cache through `FS_AssignCache()`.

5.3.3 FS_CACHE_Invalidate()

Description

Removes all the sectors from the cache.

Prototype

```
void FS_CACHE_Invalidate(const char * pVolumeName);
```

Parameter	Description
pVolumeName	Name of the volume for which the cache should be invalidated. If not specified, the first volume will be used.

Table 5.6: FS_CACHE_Invalidate() parameter list

Additional Information

This function does not write the sectors marked as dirty to device. After calling `FS_CACHE_Invalidate()` the contents of dirty sectors are lost.

5.3.4 FS_CACHE_SetAssocLevel()

Description

Configures the number of entries (ways) in the cache which can store the contents of the same sector.

Prototype

```
int FS_CACHE_SetAssocLevel(const char * pVolumeName,
                           int          AssocLevel);
```

Parameter	Description
pVolumeName	Name of the volume for which the cache should be configured. If not specified, the first volume will be used.
AssocLevel	Number of entries in the cache as power of 2 (1 for 2-way associative, 2 for 4-way associative, etc.)

Table 5.7: FS_CACHE_SetAssocLevel() parameter list

Return value

== 0 Associativity level configured
!= 0 An error occurred

Additional Information

This function makes sense only for FS_CACHE_MULTI_WAY cache modules. Calling the function on any other cache module types returns an error. The cache replacement policy uses the associativity level to decide where to store the contents of a sector in the cache. Caches with higher associativity levels tend to have a higher hit rates.

5.3.5 FS_CACHE_SetMode()

Description

Sets the mode for the cache.

Prototype

```
int FS_CACHE_SetMode(const char * pVolumeName,
                     int          TypeMask,
                     int          ModeMask);
```

Parameter	Description
pVolumeName	Name of the volume for which the cache should be configured. If not specified, the first volume will be used.
TypeMask	Specifies the sector types that should be cached. This parameter can be an OR-combination of the following sector type mask.
ModeMask	Specifies the cache mode that should be used. Use one of the following parameters as cache mode mask.

Table 5.8: FS_CACHE_SetMode() parameter list

Permitted values for parameter TypeMask (OR-combinable)	
FS_SECTOR_TYPE_MASK_DATA	Caches all data sectors.
FS_SECTOR_TYPE_MASK_DIR	Caches all directory sectors.
FS_SECTOR_TYPE_MASK_MAN	Caches all management sectors.
FS_SECTOR_TYPE_MASK_ALL	Caches all sectors by an OR-combination of: FS_SECTOR_TYPE_MASK_DATA FS_SECTOR_TYPE_MASK_DIR FS_SECTOR_TYPE_MASK_MAN

Permitted values for parameter ModeMask (OR-combinable)	
FS_CACHE_MODE_R	Sectors of types defined in TypeMask are copied to cache when read from volume.
FS_CACHE_MODE_WT	Sectors of types defined in TypeMask are copied to cache and also written to the volume. (write through cache)
FS_CACHE_MODE_WB	Sector types defined in TypeMask are lazily written back to the device. (write back cache)

Return value

== 0: Setting the mode of the cache module was successful.
== -1: Setting the mode of the cache module was not successful.

Additional Information

This function is only usable with the FS_CACHE_RW and FS_CACHE_RW_QUOTA module, after the FS_CACHE_RW cache has been assigned to a volume. The cache module needs to be configured with this function. Otherwise, neither read nor write operations are cached.

5.3.6 FS_CACHE_SetQuota()

Description

Sets the quotas for the different sector types in the CacheRW_Quota cache module.

Prototype

```
int FS_CACHE_SetMode (const char * pVolumeName,
                      int          TypeMask,
                      U32          NumSectors);
```

Parameter	Description
pVolumeName	Name of the volume for which the cache should be configured. If not specified, the first volume will be used.
TypeMask	Specifies the sector types that should be cached. This parameter can be an OR-combination of the following sector type mask.
NumSectors	Specifies the number of sectors each sector type that is defined by TypeMask should reserve.

Table 5.9: FS_CACHE_SetQuota() parameter list

Permitted values for parameter TypeMask (OR-combinable)	
FS_SECTOR_TYPE_MASK_DATA	Caches all data sectors.
FS_SECTOR_TYPE_MASK_DIR	Caches all directory sectors.
FS_SECTOR_TYPE_MASK_MAN	Caches all management sectors.
FS_SECTOR_TYPE_MASK_ALL	All sectors are cached. This is an OR-combination of FS_SECTOR_TYPE_MASK_DATA FS_SECTOR_TYPE_MASK_DIR FS_SECTOR_TYPE_MASK_MAN

Return value

== -1: Setting the quota of the cache module was not successful.
== 0: Setting the quota of the cache module was successful.

Additional Information

This function is currently only usable with the FS_CACHE_RW_QUOTA module. After the FS_CACHE_RW_QUOTA cache has been assigned to a volume and the cache mode has been set, the quotas for the different sector types need to be configured with this function. Otherwise neither read nor write operations are cached.

Example

```
#include "FS.h"

#define CACHE_SIZE   FS_SIZEOF_CACHE_RW_QUOTA(200, 512)

static char _acCache[CACHE_SIZE]; // Allocate RAM for cache buffer

void MainTask(void) {
  //
```



```
  // Assign a cache to the first available device
  //
  FS_AssignCache("", _acCache, sizeof(_acCache), FS_CACHE_RW_QUOTA);
  //
  // Set the FS_CACHE_RW module to cache all sectors
  // Sectors are cached for read and write. Write back operation to volume
  // are delayed.
  //
  FS_CACHE_SetMode("", FS_SECTOR_TYPE_MASK_ALL, FS_CACHE_MODE_FULL);
  //
  // Set the quotas for directory and data sector types
  // in the CACHE_RW_QUOTA module to 10 sectors each
  //
  FS_CACHE_SetQuota("", FS_SECTOR_TYPE_MASK_DATA | FS_SECTOR_TYPE_MASK_DIR, 10);
  //
  // Do some work
  //
  DoWork();
  FS_CACHE_Clean("");
  DoOtherWork();
  //
  // Disable the cache.
  //
  FS_AssignCache("", 0, 0, 0);
}
```


5.3.7 FS_CACHE_SetAssocLevel()

Description

Configures the number of entries (ways) in the cache which can store the contents of the same sector.

Prototype

```
int FS_CACHE_SetAssocLevel(const char * pVolumeName,
                           int          AssocLevel);
```

Parameter	Description
pVolumeName	Name of the volume for which the cache should be configured. If not specified, the first volume will be used.
AssocLevel	Number of entries in the cache as power of 2 (1 for 2-way associative, 2 for 4-way associative, etc.)

Table 5.10: FS_CACHE_SetAssocLevel() parameter list

Return value

== 0 Associativity level configured
!= 0 An error occurred

Additional Information

This function makes sense only for FS_CACHE_MULTI_WAY cache modules. Calling the function on any other cache module types returns an error. The cache replacement policy uses the associativity level to decide where to store the contents of a sector in the cache. Caches with higher associativity levels tend to have a higher hit rates.

5.4 Example applications

This example applications can be used to check the gain of performance with enabled cache. The following example applications are available:

Function	Description
FS_50Files.c	

Table 5.11: emFile cache example applications

The listed performance values depend on the compiler options, the compiler version, the used CPU, the storage medium and the defined cache size. The performance values presented in the tables below have been measured on a system as follows:

Detail	Description
CPU	ATMEL AT91SAM7S256
Tool chain	IAR Embedded Workbench for ARM V4.41A
Model	ARM7, Thumb instructions, no interwork
Compiler options	Highest speed optimization
Storage medium	SD card

Table 5.12: ARM7 sample configuration

5.4.1 Example application: FS_50Files.c

Note: The example application FS_50Files.c uses the time measurement function `OS_GetTime()` of embOS, Segger's Real Time Operating System. For more information about embOS, refer to *www.segger.com*.

The application step by step:

1. Initialize the file system.
2. Perform ma high-level format if required.
3. Create 50 files without a cache.
4. Write the time which was required for creation in the terminal I/O window.
5. Enable a read and write cache.
6. Create 50 files with the enabled read and write cache.
7. Write the time which was required for creation in the terminal I/O window.
8. Flush the cache.
9. Write the time which was required for flushing in the terminal I/O window.
10. Disable the cache.
11. Create again 50 files without a cache.
12. Write the time which was required for creation in the terminal I/O window.

Terminal output:

```
Cache disabled
Creation of 50 files took: 685 ms
Cache enabled
Creation of 50 files took: 43 ms
Cache flush took: 17 ms
Cache disabled
Creation of 50 files took: 687 ms
```


5.4.1.1 Source code listing: FS_50Files.c

```
#include <stdio.h>
#include <string.h>
#include "FS.h"
#include "RTOS.h"

/*********************************************************************
*
*       Defines configurable
*
**********************************************************************
*/
#define NUM_FILES    50

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/
static U32 _aCache[0x400];
static char _aacFileName[NUM_FILES][13];

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/
/*********************************************************************
*
*       _CreateFiles
*/
static void _CreateFiles(void) {
  int       i;
  U32       Time;
  FS_FILE * pFile[NUM_FILES];

  Time = OS_GetTime();
  for (i = 0; i < NUM_FILES; i++) {
    pFile[i] = FS_FOpen(&_aacFileName[i][0], "w");
  }
  Time = OS_GetTime() - Time;
  printf("Creation of %d files took: %d ms\n", NUM_FILES, Time);
  for (i = 0; i < NUM_FILES; i++) {
    FS_FClose(pFile[i]);
  }
}

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       MainTask
*/
void MainTask(void);
void MainTask(void) {
  const char * sVolName = "";
  int          i;
  U32       Time;

  //
  // Initialize file system
  //
  FS_Init();
  //
  // Check if low-level format is required
  //
  FS_FormatLLIfRequired("");
  //
  // Check if volume needs to be high level formatted.
  //
  if (FS_IsHLFormatted("") == 0) {
    printf("High level formatting\n");
    FS_Format("", NULL);
```



```
  }
  //
  //  Prepare strings in advance
  //
  for (i = 0; i < NUM_FILES; i++) {
    sprintf(&_aacFileName[i][0], "file%.2d.txt", i);
  }
  //
  //  Create and measure the time used to create the files.
  //
  printf("Cache disabled\n");
  _CreateFiles();
  //
  //  Create and measure the time used to create the files.
  //  R/W CACHE enabled.
  //
  FS_AssignCache(sVolName, _aCache, sizeof(_aCache), FS_CACHE_RW);
  FS_CACHE_SetMode(sVolName, FS_SECTOR_TYPE_MASK_ALL, FS_CACHE_MODE_WB);
  printf("Cache enabled\n");
  _CreateFiles();
  Time = OS_GetTime();
  FS_CACHE_Clean(sVolName);
  Time = OS_GetTime() - Time;
  printf("Cache flush took: %d ms", Time);
  //
  //  Create and measure the time used to create the files.
  //  R/W CACHE disabled.
  //
  printf("Cache disabled\n");
  FS_AssignCache(sVolName, NULL, 0, NULL);
  _CreateFiles();

  while(1);
}
```


Chapter 6

Device drivers

emFile has been designed to cooperate with any kind of hardware. To use specific hardware with emFile, a so-called device driver for that hardware is required. The device driver consists of basic I/O functions for accessing the hardware and a global table that holds pointers to these functions.

6.1 General information

6.1.1 Default device driver names

By default the following identifiers are used for each driver.

Driver (Device)	Identifier	Name
Hard disk/CompactFlash	FS_IDE_Driver	"ide:"
MMC/SD (SPI mode)	FS_MMC_SPI_Driver	"mmc:"
MMC/SD (card mode)	FS_MMC_CardMode_Driver	"mmc:"
MMC/SD (card mode for ATMEL MCUs)	FS_MMC_CM_Driver4Atmel	"mmc:"
NAND flash and ATMEL's DataFlash	FS_NAND_Driver	"nand:"
NOR flash (sector map)	FS_NOR_Driver	"nor:"
RAM disk	FS_RAMDISK_Driver	"ram:"
WINDrive	FS_WINDRIVE_Driver	"win:"
NOR flash (block map)	FS_NOR_BM_Driver	"nor:"
NAND flash for SLCs and MLCs	FS_NAND_UNI_Driver	"nand:"

Table 6.1: List of default device driver labels

Note: FS_MMC_CM_Driver4Atmel is deprecated. Use FS_MMC_CardMode_Driver instead.

To add a driver to emFile, FS_AddDevice() should be called with the proper identifier to mount the device driver to emFile before accessing the device or its units. Refer to *FS_AddDevice()* on page 59 for detailed information.

6.1.2 Unit number

Most driver functions as well as most of the underlying hardware functions receive the unit number as the first parameter. The unit number allows differentiation between the different instances of the same device types. If there are for example 2 NAND flashes which operate as 2 different devices, the first one is identified as unit 0, the second one as unit 1. If there is just a single instance (as in most systems), the unit parameter can be ignored by the underlying hardware functions.

6.1.3 Hardware layer

Some drivers, such as the MMC/SD drivers or the NAND driver, require a hardware layer. The implementation of this hardware layer is user responsibility. The hardware layer can be implemented in different ways:

- polled mode
- interrupt driven

6.1.3.1 Polled mode

In the polled mode the software actively queries the completion of the I/O operation. No operating system is required to implement the driver.

Example

```
/*---------------------------------------------------------------------
File        : HWLayer_PolledDriven.c
Purpose     : Sample hardware layer to demonstrate the fundamentals
              of an polled driven hardware layer
----------------------------END-OF-HEADER-----------------------------
*/
#include "FS.h"
#include "FS_OS.h"

/*********************************************************************
*
*             FS_HW_Write
*
*  Function description
*    FS hardware layer function. Writes a specified number of bytes via SPI
*/
void FS_HW_Write(U8 Unit, const U8 * pData, int NumBytes) {
  //
  // Start transmission using DMA
  //
  // TBD by implementer

  //
  // Make sure transmission is completed (in case interrupt came to early)
  //
  while (_IsCompleted());   // TBD by implementer
}

/************************** End of file ****************************/
```

6.1.3.2 Interrupt driven hardware layer

In the interrupt driven mode the completion of an I/O operation is signaled through an interrupt. This mode requires the support of an operating system.

Example

```
/*---------------------------------------------------------------------
File        : HWLayer_InterruptDriven.c
Purpose     : Sample hardware layer to demonstrate the fundamentals
              of an interrupt driven hardware layer
----------------------------END-OF-HEADER-----------------------------
*/
#include "FS.h"
#include "FS_OS.h"

/*********************************************************
*
*       _IrqHandler
*/
static void _IrqHandler(void) {
  //
  // Disable further interrupts
  //
  // TBD by implementer

  //
  // Signal (wake) the task waiting
  //
  FS_OS_SIGNAL();
}

/*********************************************************************
*
*             FS_HW_Write
*
*  Function description
*    FS hardware layer function. Writes a specified number of bytes via SPI
*/
void FS_HW_Write(U8 Unit, const U8 * pData, int NumBytes) {
```



```
  //
  // Start transmission using DMA
  //
  // TBD by implementer

  //
  // For larger blocks of data, enable "transmission complete" interrupt
  // and suspend task to save CPU time (if an OS is present)
  //
  if (NumBytes >= 512) {
    // Enable interrupt: TBD by implementer
    FS_OS_WAIT(1000);       // Suspend task with timeout
  }
  //
  // Make sure transmission is completed (in case interrupt came to early)
  //
  while (_IsCompleted());   // TBD by implementer
}

/*************************** End of file ****************************/
```


6.2 RAM disk driver

emFile comes with a simple RAM disk driver that makes it possible to use a portion of your system RAM as drive for data storage. This can be very helpful to examine your system performance and may also be used as a in-system test procedure.

6.2.1 Supported hardware

The RAM driver can be used with every target with enough RAM. The size of the disk is defined as the number of sectors reserved for the drive.

6.2.2 Theory of operation

A RAM disk is a portion of memory that you allocate to use as a partition. The RAM disk driver takes some of your memory and pretends that it is a hard drive that you can format, mount, save files to, etc.

Remember that every bit of RAM is important for the well being of your system and the bigger your RAM disk is, the less memory there is available for your system.

6.2.3 Fail-safe operation

When power is lost, the data of the RAM drive is typically lost as well except for systems with Battery backup for the RAM used as storage device.

For this reason, fail-safety is relevant only for systems which provide such battery backup.

Unexpected Reset

In case of an unexpected reset the data will be preserved. However, if the Power failure / unexpected Reset interrupts a write operation, the data of the sector may contain partially invalid data.

Power failure

Power failure causes an unexpected reset and has the same effects.

6.2.4 Wear leveling

The RAM disk driver does not require wear leveling.

6.2.5 Configuring the driver

6.2.5.1 Adding the driver to emFile

To add the driver, use `FS_AddDevice()` with the driver label `FS_RAMDISK_Driver`. This function has to be called from within `FS_X_AddDevices()`. Refer to *FS_X_AddDevices()* on page 472 for more information.

Example

```
FS_AddDevice(&FS_RAMDISK_Driver);
```

6.2.5.2 FS_RAMDISK_Configure()

Description

Configures a single RAM disk instance. This function has to be called from within `FS_X_AddDevices()` after adding an instance of the `RAMDisk` driver. Refer to *FS_X_AddDevices()* on page 472 for more information.

Prototype

```
void FS_RAMDISK_Configure(U8     Unit,
                          void * pData,
                          U16    BytesPerSector,
                          U32    NumSectors);
```

Parameter	Description
`Unit`	Unit number (0…N).
`pData`	Pointer to a data buffer.
`BytesPerSector`	Number of bytes per sector.
`NumSectors`	Number of sectors.

Table 6.2: FS_RAMDISK_Configure() parameter list

Additional information

The size of the disk is defined as the number of sectors reserved for the drive. Each sector consists of 512 bytes. The minimum value for `NumSectors` is 7. `BytesPerSector` defines the size of each sector on the RAM disk. A FAT file system needs a minimum sector size of 512 bytes.

Example

```
/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*    It is supposed to add all devices, using primarily FS_AddDevice().
*
*  Note
*    (1) Other API functions
*        Other API functions may NOT be called, since this function is called
*        during initialisation. The devices are not yet ready at this point.
*/
void FS_X_AddDevices(void) {
  void  * pRamDisk = NULL;

  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Allocate memory for the RAM disk
  //
  pRamDisk = FS_Alloc(RAMDISK_NUM_SECTORS * RAMDISK_BYTES_PER_SECTOR);
  //
  // Add driver
  //
  FS_AddDevice(&FS_RAMDISK_Driver);
  //
  // Configure driver
  //
  FS_RAMDISK_Configure(0, pRamDisk, RAMDISK_BYTES_PER_SECTOR, RAMDISK_NUM_SECTORS);
  //
  // Configure a file buffer for reading.
  //
  FS_ConfigFileBufferDefault(512, 0);
}
```


6.2.6 Hardware functions

The RAM disk driver does not need any hardware function.

6.2.7 Additional information

6.2.7.1 Formatting

A RAM disk is unformatted after each startup. Exceptions from this rule are RAM disks, which are memory backed up with a battery.

You have to format every unformatted RAM disk with the `FS_Format()` function, before you can store data on it. If you use only one RAM disk in your application `FS_FORMAT()` can be called with an empty string as device name. For example, `FS_Format("", NULL);`

If you use more then one RAM disk, you have to specify the device name. For example, `FS_FORMAT("ram:0:", NULL);` for the first device and `FS_FORMAT("ram:1:", NULL);` for the second. Refer to *FS_Format()* on page 111 for more detailed information about the high-level format function of emFile.

6.2.8 Performance and resource usage

6.2.8.1 Performance

These performance measurements are in no way complete, but they give an approximation of the length of time required for common operations on various targets. The tests were performed as described in *Performance* on page 502.

All values are in Mbytes/sec.

Device	CPU speed	W	R
Atmel AT91SAM9261	200 MHz	128.0	128.0
LogicPD LH79520	51 MHz	20.0	20.0

Table 6.3: Performance values for sample configurations

6.3 NAND flash driver

emFile supports the use of NAND flashes. Two optional drivers for NAND flashes are available:

- SLC1 driver - works only with SLC flashes which require 1-bit error correction. It also supports the ATMEL's DataFlashes.
- Universal driver - works with all modern SLC and MLC NAND flashes. It can use the ECC engine build into NAND flashes to correct bit errors.

To use the drivers in your system, you will have to provide basic I/O functions for accessing your flash device.

How to select which driver to use

The first factor is the type of device used. ATMEL's DataFlashes are supported only by the SLC1 driver. NAND flashes are supported by both drivers.

The bit error correction requirements of the NAND flash is the next factor. It indicates how many bit errors the error correcting code (ECC) must be able to detect and correct. If the NAND flash requires only 1-bit correction capability then the SLC1 driver can be used. The SLC1 driver will perform the bit error detection and correction. For more than 1-bit correction capability the Universal driver is required. In order to use the Universal driver, the following conditions must be met:

- Page size of minimum 2048 bytes.
  More specifically: the size spare area corresponding to 512 bytes in the data area must be greater than 16. For more information about the internals of a NAND flash, refer to *NAND flash organization* on page 222.
- Hardware support for error correction.
  Either a NAND flash with internal ECC engine or another way to compute the ECC in hardware (MCU, FPGA, etc.)
- The ECC must not exceed 8 bytes in size.

Multiple driver configuration

Both drivers store management information to spare area of a page. The layout and the content of this information is different for each driver which means that data written using one driver is not recognized when read using the other one. If an application used the SLC1 driver to write to NAND flash it can not use the Universal driver to access it. The application should decide at runtime in the `FS_X_AddDevices()` function which driver to configure. emFile supports the configuration of a driver based on the type of NAND flash connected to host. For an example, refer to *FS_NAND_PHY_ReadDeviceId()* on page 308.

6.3.1 SLC1 driver - FS_NAND_Driver

This driver for NAND flashes requires very little RAM, it can work with sector sizes of 512 bytes or 2 Kbytes (small sectors even on large page NAND flashes) and is extremely efficient. The driver is designed to support one or multiple SLC (Single Level Cell) NAND flashes which require 1-bit ECC. The NAND flash driver can also be used to access ATMEL's DataFlash chips.

This section first describes which devices are supported and describes all hardware access functions required by the NAND flash driver.

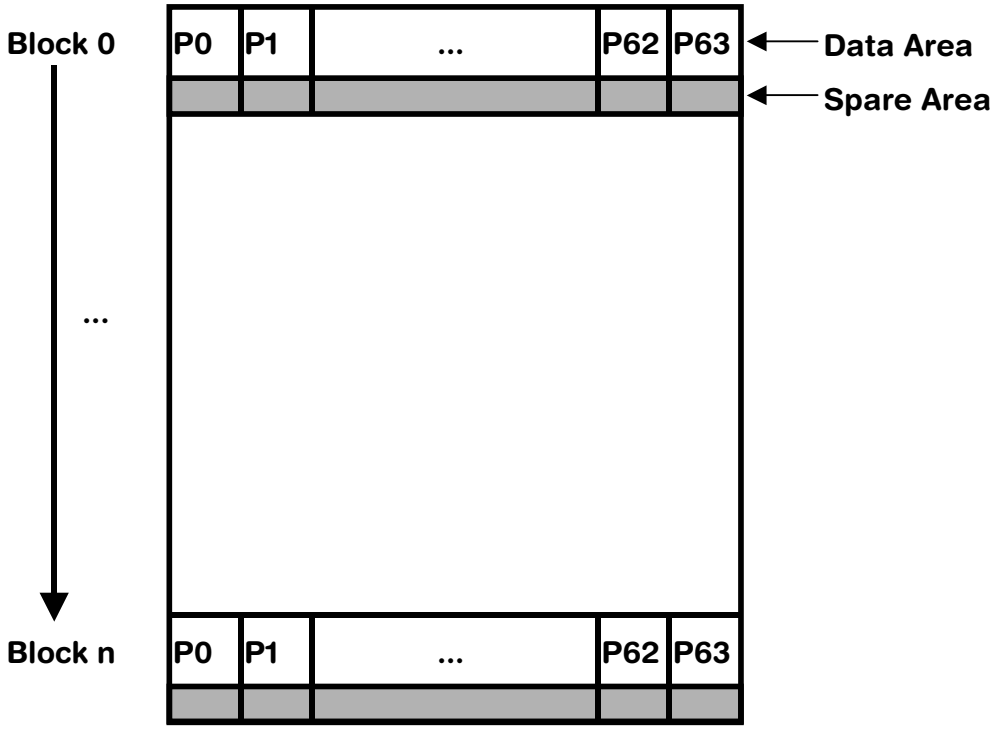
6.3.1.1 NAND flash organization

A NAND flash is a serial-type memory device which utilizes the I/O pins for both address and data input/output as well as for command inputs. The erase and program operations are automatically executed. To store data on the NAND flash device, it has to be low-level formatted.

NAND flashes consist of a number of blocks. Every block contains a number of pages, typically 64. The pages can be written to individually, one at a time. When writing to a page, bits can only be written from 1 to 0. Only whole blocks (all pages in the block) can be erased. Erasing means bringing all memory bits in all pages of the block to logical 1.

Small NAND flashes (up to 256 Mbytes) have a page size of 528 bytes, 512 for data + 16 spare bytes for storing relevant information (ECC, etc.) to the page. Large NAND devices (256 Mbytes or more) have a page size of 2112 bytes, 2048 bytes for data + 64 bytes for storing relevant information to the page.

NAND flash organization



For example, a typical NAND flash with a size of 256 Mbytes has 2048 blocks of 64 pages of 2112 bytes (2048 bytes for data + 64 bytes).

6.3.1.2 Supported hardware

6.3.1.2.1 Tested and compatible NAND flashes

In general, the driver supports almost all Single-Level Cell NAND flashes (SLC). This includes NAND flashes with page sizes of 512+16 and 2048+64 bytes.

The table below shows the NAND flashes that have been tested or are compatible with a tested device:

Manufacturer	Device	Page size [Bytes]	Size [Bits]
Hynix	HY27xS08281A	512+16	16Mx8
	HY27xS08561M	512+16	32Mx8
	HY27xS08121M	512+16	64Mx8
	HY27xA081G1M	512+16	128Mx8
Samsung	K9F6408Q0xx	512+16	8Mx8
	K9F6408U0xx	512+16	8Mx8
	K9F2808Q0xx	512+16	16Mx8
	K9F2808U0xx	512+16	16Mx8
	K9F5608Q0xx	512+16	32Mx8
	K9F5608D0xx	512+16	32Mx8
	K9F5608U0xx	512+16	32Mx8
	K9F1208Q0xx	512+16	64Mx8
	K9F1208D0xx	512+16	64Mx8
	K9F1208U0xx	512+16	64Mx8
	K9F1208R0xx	512+16	64Mx8
	K9K1G08R0B	512+16	128Mx8
	K9K1G08B0B	512+16	128Mx8
	K9K1G08U0B	512+16	128Mx8
	K9K1G08U0M	512+16	128Mx8
	K9T1GJ8U0M	512+16	128Mx8
ST-Microelectronics	NAND128R3A	512+16	16Mx8
	NAND128W3A	512+16	16Mx8
	NAND256R3A	512+16	32Mx8
	NAND256W3A	512+16	32Mx8
	NAND512R3A	512+16	64Mx8
	NAND512W3A	512+16	64Mx8
	NAND01GR3A	512+16	128Mx8
	NAND01GW3A	512+16	128Mx8
Toshiba	TC5816BFT	512+16	2Mx8
	TC58V32AFT	512+16	4Mx8
	TC58V64BFTx	512+16	8Mx8
	TC58256AFT	512+16	32Mx8
	TC582562AXB	512+16	32Mx8
	TC58512FTx	512+16	64Mx8
	TH58100FT	512+16	256Mx8
Hynix	HY27UF082G2M	2048+64	256Mx8
	HY27UF084G2M	2048+64	512Mx8
	HY27UG084G2M	2048+64	512Mx8
	HY27UG084GDM	2048+64	512Mx8

Table 6.4: List of supported NAND flashes

Manufacturer	Device	Page size [Bytes]	Size [Bits]
Micron	MT29F2G08AAB	2048+64	256Mx8
	MT29F2G08ABD	2048+64	256Mx8
	MT29F4G08AAA	2048+64	512Mx8
	MT29F4G08BAB	2048+64	512Mx8
	MT29F2G16AAD	2048+64	128Mx16
Samsung	K9F1G08x0A	2048+64	256Mx8
	K9F2G08U0M	2048+64	256Mx8
	K9K2G08R0A	2048+64	256Mx8
	K9K2G08U0M	2048+64	256Mx8
	K9F4G08U0M	2048+64	512Mx8
	K9F8G08U0M	2048+64	1024Mx8
ST-Microelectronics	NAND01GR3B	2048+64	128Mx8
	NAND01GW3B	2048+64	128Mx8
	NAND02GR3B	2048+64	256Mx8
	NAND02GW3B	2048+64	256Mx8
	NAND04GW3	2048+64	512Mx8

Table 6.4: List of supported NAND flashes

Support for devices not in this list

Most other NAND flash devices are compatible with one of the supported devices. Thus, the driver can be used with these devices or may only need a little modification, which can be easily done. Get in touch with us, if you have questions about support for devices not in this list.

6.3.1.2.2 Tested and compatible DataFlash chips

The NAND flash driver fully supports the ATMEL DataFlash®/DataFlash Cards series up to 128 MBit. Currently the following devices are supported:

Manufacturer	Device
ATMEL	AT45DB011B
	AT45DB021B
	AT45DB041B
	AT45DB081B
	AT45DB161B
	AT45DB321C
	AT45BR3214B
	AT45DCB002
	AT45DCB002
	AT45DB642D
	AT45DB1282

Table 6.5: List of supported serial flash devices

Note: DataFlash chips with a page size that is power of 2 are not supported by this driver.

6.3.1.2.3 Pin description - NAND flashes

Pin	Driver (Device)
$\overline{CE}$	CHIP ENABLE The $\overline{CE}$ input enables the device. Signal is active low. If the signal is inactive, device is in standby mode.
$\overline{WE}$	WRITE ENABLE The $\overline{WE}$ input controls writes to the I/O port. Commands, address and data are latched on the rising edge of the WE pulse.
$\overline{RE}$	READ ENABLE The $\overline{RE}$ input is the serial data-out control. When active (low) the device outputs data.
CLE	COMMAND LATCH ENABLE The CLE input controls the activating path for commands sent to the command register. When active high, commands are latched into the command register through the I/O ports on the rising edge of the WE signal.
ALE	ADDRESS LATCH ENABLE The ALE input controls the activating path for address to the internal address registers. Addresses are latched on the rising edge of WE with ALE high.
$\overline{WP}$	WRITE PROTECT Typically connected to VCC (recommended), but may also be connected to port pin.

Table 6.6: NAND flash pin description

R/$\overline{B}$	READY/$\overline{B}$USY OUTPUT The R/$\overline{B}$ output indicates the status of the device operation. When low, it indicates that a program, erase or read operation is in process. It returns to high state when the operation is completed. It is an open drain output. Should be connected to a port pin with pull-up. If available a port pin which can trigger an interrupt should be used.
I/O_0- I/O_7	DATA INPUTS/OUTPUTS The I/O pins are used to input command, address and data, and to output data during read operations.
I/O_8- I/O_{15}	DATA INPUTS/OUTPUTS I/O_8-I/O_{15} 16-bit flashes only.

6.3.1.2.4 Pin description - DataFlashes

DataFlash chips are commonly used when low pin count and easy data transfer are required. DataFlash devices use the following pins:

Pin	Meaning
CS	ChipSelect This pin selects the DataFlash device. The device is selected, when CS pin is driven low.

Table 6.7: DataFlash chip pin function description

Pin	Meaning
SCLK	Serial Clock The SCLK pin is an input-only pin and is used to control the flow of data to and from the DataFlash. Data is always clocked into the device on the rising edge of SCLK and clocked out of the device on the falling edge of SCLK.
SI	Serial Data In The SI pin is an input-only pin and is used to transfer data into the device. The SI pin is used for all data input including opcodes and address sequences.
SO	Serial Data Out This SO pin is an output pin and is used to transfer data serially out of the device.

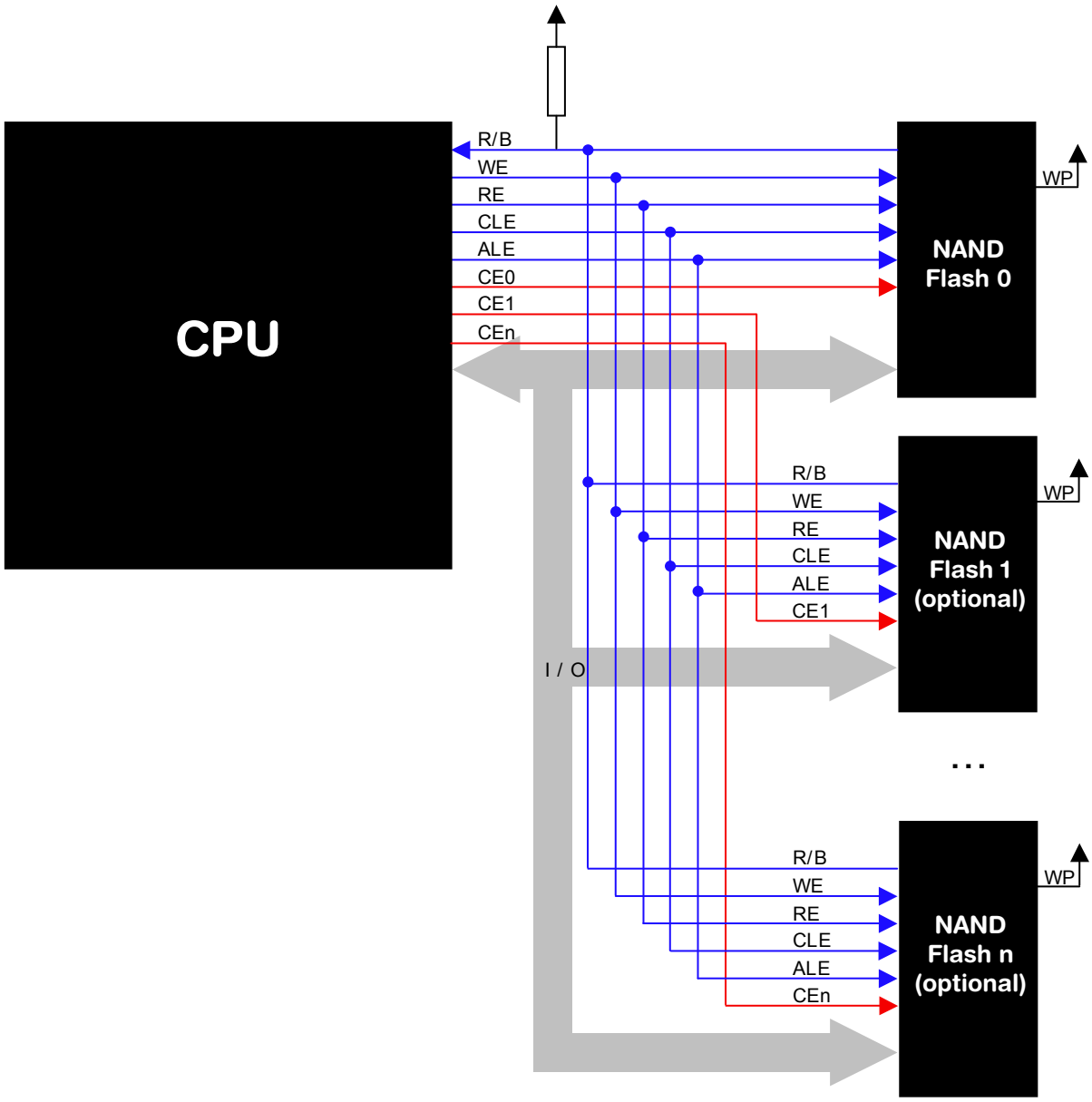
Table 6.7: DataFlash chip pin function description

Additionally the following requirements need to be fulfilled by your host system:

- Data transfer width is 8 bit.
- Chip Select (CS) sets the card active at low-level and inactive at high level.
- Clock signal must be generated by the target system. The serial flash chips are always in slave mode.
- Bit order requires most significant bit (MSB) to be sent out first.

To setup all these requirements, the NAND flash driver will call the function `FS_DF_HW_X_Init()`, therefore the function `FS_DF_HW_X_Init()` can be used to initialize the SPI bus. Refer to *FS_DF_HW_X_Init()* on page 269 for further details.

6.3.1.2.5 Sample block schematics



6.3.1.3 Theory of operation

NAND flash devices are divided into physical blocks and physical pages. One physical block is the smallest erasable unit; one physical page is the smallest writable unit. Small block NAND flashes contain multiple pages. One block contain typically 16 / 32 / 64 pages per block. Every page has a size of 528 bytes (512 data bytes + 16 spare bytes). Large block NAND Flash devices contain blocks made up of 64 pages, each page containing 2112 bytes (2048 data bytes + 64 spare bytes).

The driver uses the spare bytes for the following purposes:

1. To check if the data status byte and block status are valid.
   If they are valid the driver uses this sector. When the driver detects a bad sector, the whole block is marked as invalid and its content is copied to a non-defective block.
2. To store/read an ECC (Error Correction Code) for data reliability.
   When reading a sector, the driver also reads the ECC stored in the spare area of the sector, calculates the ECC based on the read data and compares the ECCs. If the ECCs are not identical, the driver tries to recover the data, based on the read ECC.

When writing to a page the ECC is calculated based on the data the driver has to write to the page. The calculated ECC is then stored in the spare area.

6.3.1.3.1 Error correction code (ECC)

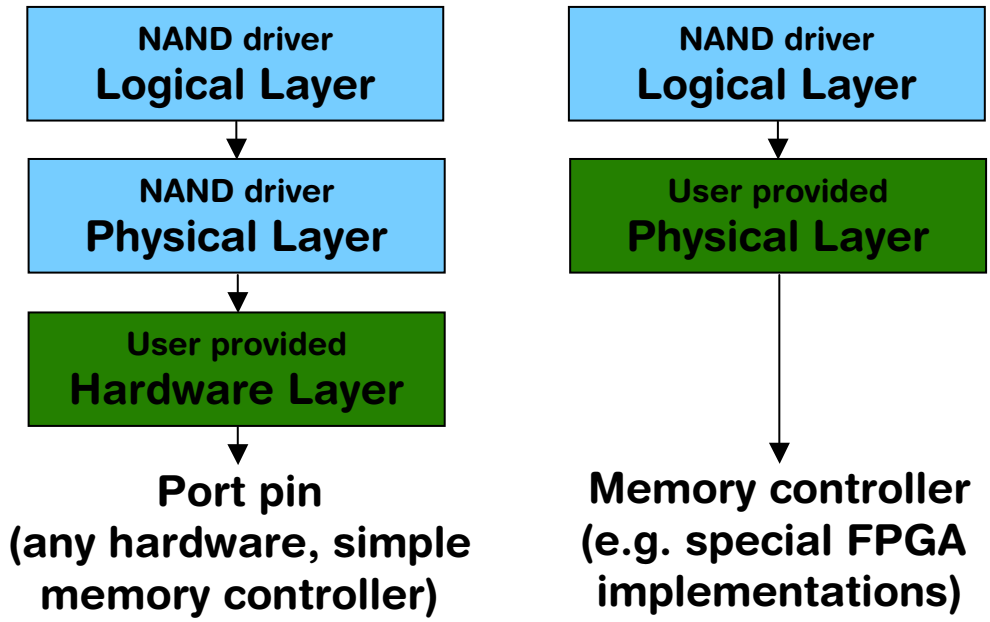
The emFile NAND driver is highly speed optimized and offers a better error detection and correction than a standard memory controller ECC. The ECC is capable of single bit error correction and 2-bit random detection. When a block for which the ECC is computed has 2 or more bit errors, the data cannot be corrected.

Standard memory controllers compute an ECC for the complete blocksize (512 / 2048 bytes). The emFile NAND driver computes the ECC for data chunks of 256 bytes (e.g. a page with 2048 bytes is divided into 8 parts of 256 bytes), so the probability to detect and also correct data errors is much higher. This enhancement is realized with a very good performance. The ECC computation of the emFile NAND driver is highly optimized, so that a performance of 18 Mbytes/second can be achieved with an ARM7 based MCU running at 48 MHz.

We suggest the use of the emFile NAND driver without enabling the hardware ECC of the memory controller, because the performance of the driver is very high and the error correction is much better if it is controlled from driver side.

6.3.1.3.2 Software structure

The NAND Flash driver is split up into different layers, which are shown in the illustration below.



It is possible to use the NAND driver with custom hardware. If port pins or a simple memory controller are used for accessing the flash memory, only the hardware layer needs to be ported, normally no changes to the physical layer are required. If the NAND driver should be used with a special memory controller (for example special FPGA implementations), the physical layer needs to be adapted. In this case, the hardware layer is not required, because the memory controller manages the hardware access.

6.3.1.4 Fail-safe operation

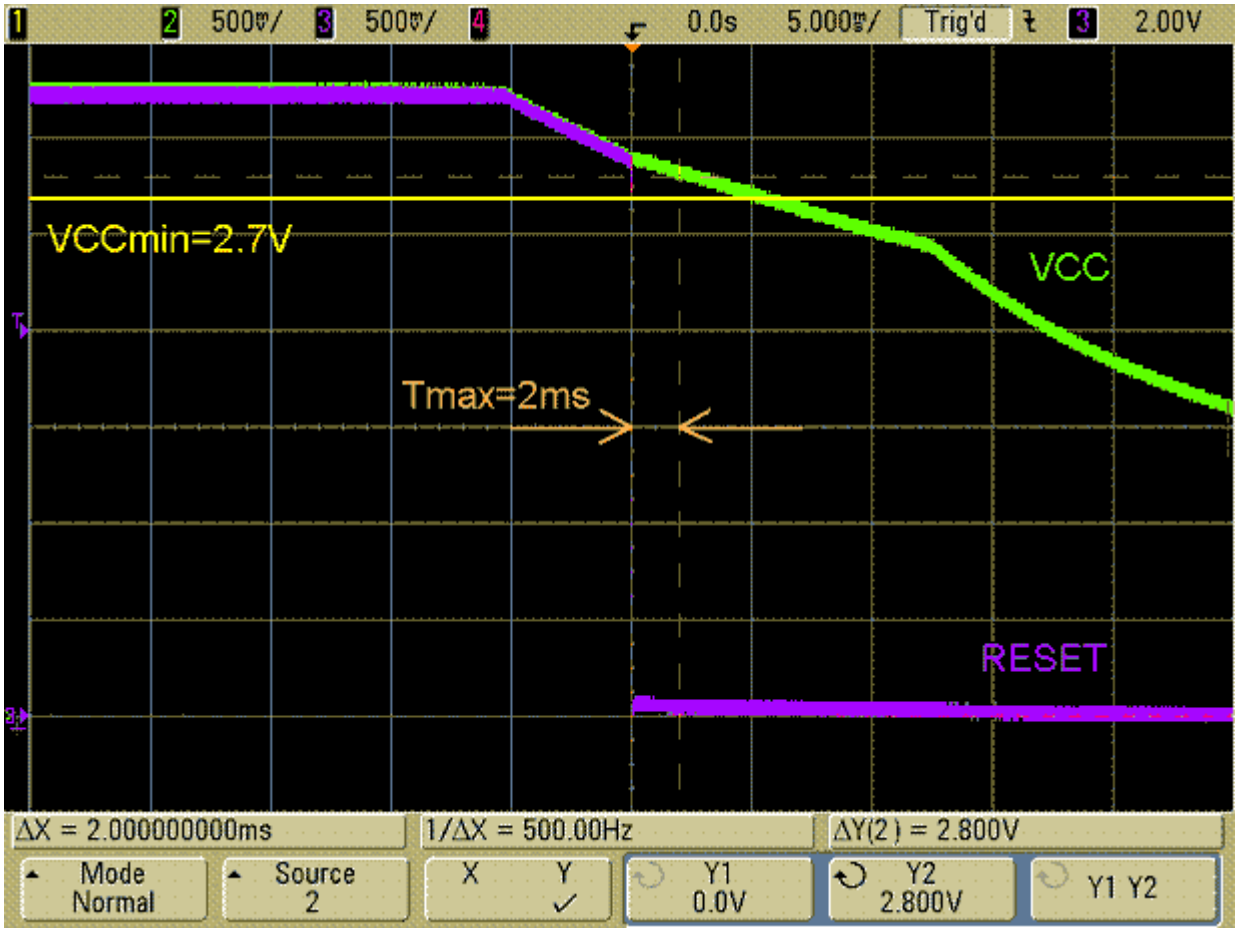
The emFile NAND flash driver is fail-safe which means that the driver makes only atomic actions and takes the responsibility that the data managed by the file system is always valid. In case of a power loss or a power reset during a write operation, it is always assured that only valid data is stored to NAND flash. If the power loss interrupts the write operation, the old data will be kept and the data is not corrupted.

In case of a power loss the fail-safe operation is only guaranteed if the NAND flash is able to fully complete the last command it received from the CPU.

Below is an oscilloscope capture which shows an example power down sequence meeting the requirements needed for a fail-safe operation of a NAND flash.

- VCC is the main power supply voltage.
- RESET is a signal driven high by a program running on the CPU. This signal goes low when the CPU stops running indicating the point in time when the last command could have been sent to NAND flash.
- VCCmin is the minimum supply voltage required for the NAND flash to properly operate.
- Tmax is the time it takes for the longest NAND flash operation to complete which is 2 ms for the NAND flash used in the test.

As it can be seen in the picture the supply voltage stays above VCCmin long enough to allow for any NAND flash command to finish.



6.3.1.5 Wear leveling

Wear leveling is supported by the driver. The procedure ensures that the number of erase cycles remains approximately for all the blocks. The maximum allowed erase count difference is runtime configurable and is by default 5000.

6.3.1.6 Partial writes

Most of the NAND devices allow a write operation to change an arbitrary number of bytes starting from any byte offset inside a page. A write operation that does not change all the bytes in page is called partial write or partial programming. The driver makes extensively use of this feature to increase the write speed and to reduce the RAM usage. But there is a limitation of this method imposed by the NAND technology. The manufacturer does not guarantee the integrity of the data if a page is partially

written more than a number of times without an intermediate erase operation. The maximum number of partial writes is usually 4. Exceeding the maximum number of partial writes does not lead automatically to the corruption of data in that page but it will increase the probability of a bit error. The driver will be able to correct the bit error using the ECC. For some combinations of logical sector size and NAND page size the driver might exceed this limit. The table below summarizes the maximum number of partial writes performed by the driver:

NAND page size [bytes]	Logical sector size [bytes]	Maximum number of partial writes
512	512	4
2048	2048	4
2048	1024	5
2048	512	8

Table 6.8: Maximum number of partial writes

6.3.1.7 Read disturbs

Read disturbs are bit errors which occur when a large number of read operations (several hundred thousand to one million) are preformed on a NAND flash block without being erased in between. These errors must be handled by the application by rewriting the data. It can be done using the `FS_STORAGE_RefreshSectors()` storage layer API function which is able to refresh several sectors in a single call.

6.3.1.8 Configuring the driver

6.3.1.8.0.1 Adding the driver to emFile

To add the driver, use `FS_AddDevice()` with the driver label `FS_NAND_Driver` or `FS_NAND_UNI_Driver`. This function has to be called from within `FS_X_AddDevices()`. Refer to *FS_X_AddDevices()* on page 472 for more information.

Example

```
void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  FS_AddDevice(&FS_NAND_Driver);
  FS_NAND_SetPhyType(0, &FS_NAND_PHY_x8); // Set the physical
                                          // interface of the NAND flash
  FS_NAND_SetBlockRange(0, 2, 128);       // Skip 2 blocks (256 Kbytes in case of 2K
                                          // device)
                                          // Size is 128 blocks
                                          // For 2k devices, this means
                                          // 2 Kbytes * 64 * 128 = 16 Mbytes
}
```

6.3.1.8.1 Specific configuration functions

Routine	Explanation
`FS_NAND_SetPhyType()`	Configures the physical type of NAND device.
`FS_NAND_SetBlockRange()`	Configures the range of physical blocks managed by the driver.
`FS_NAND_SetMaxEraseCntDiff()`	Configures the threshold for the wear-leveling.
`FS_NAND_SetOnFatalErrorCallback()`	Configures a function to be invoked when the driver encounters a fatal error.
`FS_NAND_SetNumWorkBlocks()`	Configures the number of work blocks.

Table 6.9: FS_NAND_Driver - list of configuration functions.

6.3.1.8.1.1 FS_NAND_SetPhyType()

Description

Sets the physical type of the device. NAND flash is organized in pages of either 512 or 2048 bytes and has an 8-bit or 16-bit interface. The driver needs to know the correct combination of page and interface width.

Prototype

```
void FS_NAND_SetPhyType(U8                       Unit,
                        const FS_NAND_PHY_TYPE * pPhyType);
```

Parameter	Meaning
Unit	Unit number.
pPhyType	IN: Physical type of device. OUT: ---

Table 6.10: FS_NAND_SetPhyType() parameter list

Permitted values for parameter pPhyType	
FS_NAND_PHY_512x8	Supports NAND flash devices with 512 bytes per page and 8-bit width
FS_NAND_PHY_2048x8	Supports NAND flash devices with 2048 bytes per page and 8-bit width
FS_NAND_PHY_2048x16	Supports NAND flash devices with 2048 bytes per page and 16-bit width
FS_NAND_PHY_4096x8	Supports NAND flash devices with 4096 bytes per page and 8-bit width
FS_NAND_PHY_x	Supports the following NAND flash devices: • 512 bytes per page and 8-bit width • 2048 bytes per page and 8-bit width • 2048 bytes per page and 16-bit width
FS_NAND_PHY_x8	Supports the following NAND flash devices: • 512 bytes per page and 8-bit width • 2048 bytes per page and 8-bit width
FS_NAND_PHY_DataFlash	Supports ATMEL DataFlashes. The physical layer driver accesses these chips using the SPI mode. To use the driver with ATMEL DataFlash chips in your system, you will have to provide basic I/O functions which are divergent to the hardware functions of the other physical layers. Refer to *Hardware layer* on page 253 for detailed information.
FS_NAND_PHY_ONFI	Supports NAND flash devices which are compatible to ONFI specification. ONFI (Open NAND Flash Interface) is an standard aimed at increasing the compatibility between NAND devices. More information can be found at *www.onfi.org*
FS_NAND_PHY_SPI	Supports NAND flash devices with SPI serial interface.

Additional information

This function needs to be called for every NAND device added.

Example

Refer to *Adding the driver to emFile* on page 230 for an example.

6.3.1.8.1.2 FS_NAND_SetBlockRange()

Description

Sets a limit for which blocks of the NAND flash can be controlled by the driver.

Prototype

```
void FS_NAND_SetBlockRange(U8  Unit,
                           U16 FirstBlock,
                           U16 MaxNumBlocks);
```

Parameter	Meaning
Unit	Unit number.
FirstBlock	Zero-based index of the first block to use. Specifies the number of blocks at the beginning of the device to skip. 0 means that no blocks are skipped.
MaxNumBlocks	Maximum number of blocks to use. 0 means use all blocks after FirstBlock.

Table 6.11: FS_NAND_SetBlockRange() parameter list

Additional information

This function is optional. By default, the driver controls all blocks of the NAND flash, making the entire NAND flash available. If a part of the NAND flash should be used for another purpose (for example to store the application program used by a boot-loader) and therefore is not controlled by the driver, this function can be used. Limiting the number of blocks used by the driver also reduces the amount of memory used by the driver.

Note: The read optimization of the FS_NAND_PHY_2048x8 physical layer must be disabled when this function is used to divide the same NAND flash device into 2 or more partitions. It can be done by calling the FS_NAND_2048x8_DisableReadCache() function.

Example

Refer to *Adding the driver to emFile* on page 230 for an example.

6.3.1.8.1.3 FS_NAND_SetMaxEraseCntDiff()

Description

Sets the maximum difference between block erase counts that triggers the active wear leveling.

Prototype

```
void FS_NAND_SetMaxEraseCntDiff(U8  Unit,
                                U32 EraseCntDiff);
```

Parameter	Meaning
Unit	Unit number.
EraseCntDiff	Maximum allowed difference between the erase counts.

Table 6.12: FS_NAND_SetMaxEraseCntDiff() parameter list

Additional information

This function controls how the driver performs the wear leveling. The wear leveling algorithm chooses first the next available block from the list of free blocks. Then the difference between the erase count of the chosen block and the lowest erase count of used blocks is computed. If this value is greater than EraseCntDiff the block with the lowest erase count is freed and made available for use.

6.3.1.8.1.4 FS_NAND_SetOnFatalErrorCallback()

Description

Registers a function that should be invoked when a fatal error occurs.

Prototype

```
void FS_NAND_SetOnFatalErrorCallback(
     FS_NAND_ON_FATAL_ERROR_CALLBACK * pfOnFatalError);
```

Parameter	Meaning
pfOnFatalError	Pointer to callback function.

Table 6.13: FS_NAND_SetOnFatalErrorCallback() parameter list

Additional information

The type of the callback function is defined as follows:

```
typedef int FS_NAND_ON_FATAL_ERROR_CALLBACK(
            FS_NAND_FATAL_ERROR_INFO * pFatalErrorInfo);
```

The parameter is a structure defined like this:

```
typedef struct {
  U8 Unit;
} FS_NAND_FATAL_ERROR_INFO;
```

Unit is the number of NAND driver that encountered the fatal error.

If the callback function returns a 0 the driver marks the NAND flash as read-only. In this state all further write operations are rejected by the driver with an error. A low-level format is required to make the NAND flash writable. The callback function can also return a 1 in which case the medium is not marked as read-only.

6.3.1.8.1.5 FS_NAND_SetNumWorkBlocks()

Description

Sets number of work blocks the driver uses for write operations.

Prototype

```
void FS_NAND_SetNumWorkBlocks(U8  Unit,
                              U32 NumWorkBlocks);
```

Parameter	Meaning
Unit	Unit number.
NumWorkBlocks	Number of work blocks.

Table 6.14: FS_NAND_SetNumWorkBlocks() parameter list

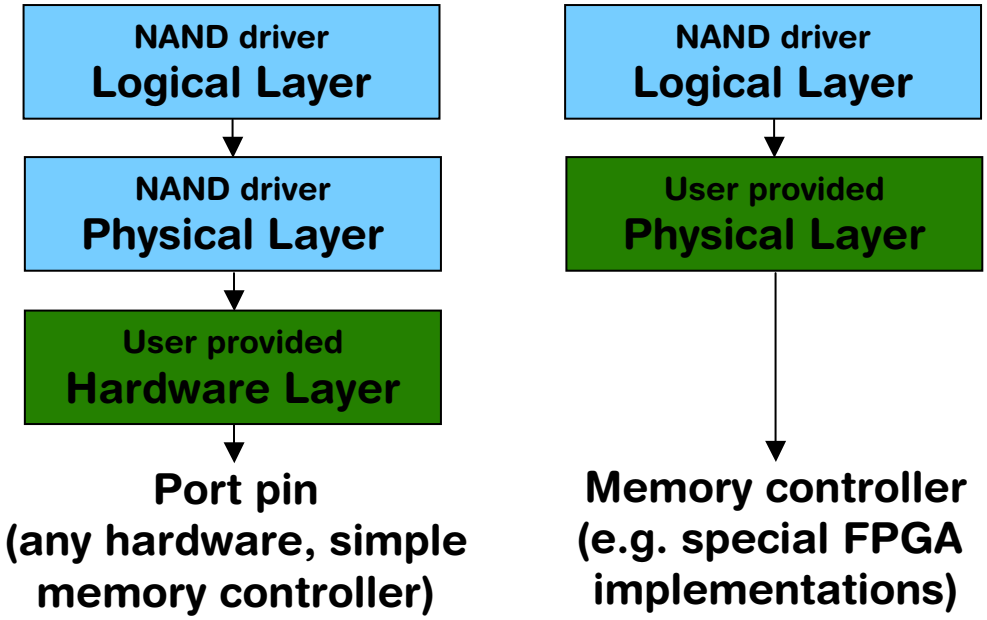
Additional information

Work blocks are physical blocks which the driver uses to temporarily store the data written to NAND flash. This function can be used to change the number of work blocks according to the requirements of an application. Usually, the write performance of the NAND driver improves when the number work blocks is increased. Please note that increasing the number of work blocks will also increase the RAM usage. By default, the NAND driver allocates 10% from the total number of blocks available but no more than 10 blocks. The minimum number of work blocks allocated by default depends whether journaling is used or not. If the journal is active the 4 work blocks are allocated else 3.

6.3.1.9 Physical layer

There is normally no need to change the physical layer of the NAND driver, only the hardware layer has to be adapted.

In some special cases, when the low-level hardware routines provided by the driver are not compatible with the target hardware (e.g. special FPGA implementations of a memory controller), the physical layer has to be adapted.



6.3.1.9.1 Available physical layers

The following physical layers are available. Refer to *Configuring the driver* on page 230 for detailed information about how to add the required physical layer to your application.

Available physical layers	
FS_NAND_PHY_512x8	Supports NAND devices with 512 bytes per page and 8-bit width
FS_NAND_PHY_2048x8	Supports NAND devices with 2048 bytes per page and 8-bit width
FS_NAND_PHY_2048x16	Supports NAND devices with 2048 bytes per page and 16-bit width
FS_NAND_PHY_4096x8	Supports NAND devices with 4096 bytes per page and 8-bit width
FS_NAND_PHY_x	Supports the following NAND devices: • 512 bytes per page and 8-bit width • 2048 bytes per page and 8-bit width • 2048 bytes per page and 16-bit width • 4096 bytes per page and 8-bit width
FS_NAND_PHY_x8	Supports the following NAND devices: • 512 bytes per page and 8-bit width • 2048 bytes per page and 8-bit width • 4096 bytes per page and 8-bit width

Table 6.15: Available physical layers

Available physical layers	
FS_NAND_PHY_DataFlash	Supports ATMEL DataFlashes. The physical layer driver accesses these chips using the SPI mode. To use the driver with ATMEL DataFlash chips in your system, you will have to provide basic I/O functions which are divergent to the hardware functions of the other physical layers. Refer to *Hardware layer* on page 253 for detailed information.
FS_NAND_PHY_ONFI	Supports NAND flash devices which are compatible to ONFI specification.
FS_NAND_PHY_SPI	Supports NAND flash devices with SPI serial interface.

Table 6.15: Available physical layers

6.3.1.9.2 Physical layer functions

If there is a reason to change the physical layer anyhow, the functions which have to be changed are organized in a function table. The function table is implemented in a structure of type FS_NAND_PHY_TYPE.

```
typedef struct FS_NAND_PHY_TYPE {
  int    (*pfEraseBlock)        (U8                    Unit,
                                 U32                   Block);
  int    (*pfInitGetDeviceInfo) (U8                    Unit,
                                 FS_NAND_DEVICE_INFO * pDevInfo);
  int    (*pfIsWP)              (U8                    Unit);
  int    (*pfRead)              (U8                    Unit,
                                 U32                   PageNo,
                                 void *                pData,
                                 unsigned              Off,
                                 unsigned              NumBytes);
  int    (*pfReadEx)            (U8                    Unit,
                                 U32                   PageNo,
                                 void *                pData,
                                 unsigned              Off,
                                 unsigned              NumBytes,
                                 void *                pSpare,
                                 unsigned              OffSpare,
                                 unsigned              NumBytesSpare);
  int    (*pfWrite)             (U8                    Unit,
                                 U32                   PageNo,
                                 const void *          pData,
                                 unsigned              Off,
                                 unsigned              NumBytes);
  int    (*pfWriteEx)           (U8                    Unit,
                                 U32                   PageNo,
                                 const void *          pData,
                                 unsigned              Off,
                                 unsigned              NumBytes,
                                 const void *          pSpare,
                                 unsigned              OffSpare,
                                 unsigned              NumBytesSpare);
  int (*pfEnableECC)            (U8                    Unit);
  int (*pfDisableECC)           (U8                    Unit);
  int (*pfConfigureECC)         (U8                    Unit,
                                 U8                    NumBitsCorrectable,
                                 U16                   BytesPerECCBlock);
  int (*pfCopyPage)             (U8                    Unit,
                                 U32                   PageNoSrc,
                                 U32                   PageNoDest);
} FS_NAND_PHY_TYPE;
```


If the physical layer should be modified, the following members of the structure FS_NAND_PHY_TYPE have to be adapted:

Routine	Explanation
(*pfEraseBlock)()	Erases a chosen block of the device.
(*pfInitGetDeviceInfo)()	Initializes the devices and retrieves the device information.
(*pfIsWP)()	Checks if the device is write protected.
(*pfRead)()	Reads data from the device.
(*pfReadEx)()	Reads data from the device and the spare area.
(*pfWrite)()	Writes data to the device.
(*pfWriteEx)()	Writes data to the device and the spare area.
(*pfEnableECC)()	Enables the HW ECC.
(*pfDisableECC)()	Disables the HW ECC.
(*pfConfigureECC)()	Configures the HW ECC.
(*pfCopyPage)()	Copies the contents of one page to an other one.

Table 6.16: NAND device driver physical layer functions

6.3.1.9.2.1 (*pfEraseBlock)()

Description

Erases one block of the device. A block is the smallest erasable unit.

Prototype

```
int (*pfEraseBlock) (U8 Unit, U32 PageIndex);
```

Parameter	Meaning
Unit	Unit number (0…N).
PageIndex	Zero-based index of the first page in the block to be erased. If the device has 64 pages per block, then the following values are permitted: PageIndex == 0 -> block 0, PageIndex == 64 -> block 1, PageIndex == 128 -> block 2, etc.

Table 6.17: (*pfEraseBlock)() parameter list

Return value

== 0: On success, block erased.
==-1: In case of an error.

6.3.1.9.2.2 (*pfInitGetDeviceInfo)()

Description

Initializes hardware layer, resets NAND flash and tries to identify the NAND flash. If the NAND flash can be handled, `FS_NAND_DEVICE_INFO` is filled.

Prototype

```
int (*pfInitGetDeviceInfo) (U8                    Unit,
                            FS_NAND_DEVICE_INFO * pDevInfo) {
```

Parameter	Meaning
`Unit`	Unit number (0…N).
`pDevInfo`	Pointer to a structure of type `FS_NAND_DEVICE_INFO`.

Table 6.18: (*pfInitGetDeviceInfo)() parameter list

Return value

== 0: On success.
== 1: In case of an error.

6.3.1.9.2.3 (*pfIsWP)()

Description

Checks if the device is write protected. This is done by reading bit 7 of the status register. Typical reason for write protection is that either the supply voltage is too low or the /WP-pin is active (low).

Prototype

```
int (*pfIsWP)(U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0…N).

Table 6.19: (*pfIsWP)() parameter list

Return value

== 0: Device is not write protected.
== 1: Device is write protected.

6.3.1.9.2.4 (*pfRead)()

Description

This function can be used to read from the data or spare area of the device. The spare area is assumed to be located right after the main area.

Prototype

```
int (*pfRead) (U8          Unit,
               U32         PageIndex,
               void      * pData,
               unsigned    Off,
               unsigned    NumBytes);
```

Parameter	Meaning
Unit	Unit number (0...N).
PageIndex	Zero-based index of page to be read. Needs to be smaller than page size.
pData	Pointer to a buffer for read data.
Off	Byte offset within the page.
NumBytes	Number of bytes to read

Table 6.20: (*pfRead)() parameter list

Return value

== 0: Data successfully transferred.
!= 0: An error has occurred.

Additional information

If the parameter Off is smaller than the page size, the data area is accessed. An offset greater than the page size indicates that the spare area should be accessed.

6.3.1.9.2.5 (*pfReadEx)()

Description

Reads from both the data and the spare area of a page.

Prototype

```
int     (*pfReadEx) (U8          Unit,
                     U32         PageIndex,
                     void      * pData,
                     unsigned    Off,
                     unsigned    NumBytes,
                     void      * pSpare,
                     unsigned    OffSpare,
                     unsigned    NumBytesSpare);
```

Parameter	Meaning
Unit	Unit number (0...N).
PageIndex	Number of page that should be read.
pData	Pointer to a buffer for read data.
Off	Byte offset within the page, which needs to be smaller than the page size.
NumBytes	Number of bytes to read.
pSpare	Pointer to a buffer for spare data.
OffSpare	Offset from the start of the spare area to the point where spare data should be read. First byte of the spare area has the same offset as the page size. Example: Page size: 512 OffSpare == 512 -> First byte of spare area OffSpare == 513 -> Second byte of spare area Page size: 2048 OffSpare == 2048 -> First byte of spare area OffSpare == 2049 -> Second byte of spare area
NumBytesSpare	Number of spare bytes to read.

Table 6.21: (*pfReadEx)() parameter list

Return value

== 0: Data successfully transferred.
!= 0: An error has occurred.

6.3.1.9.2.6 (*pfWrite)()

Description

Writes data into a complete or a part of a page. This code is identical for main memory and spare area; the spare area is located right after the main area.

Prototype

```
int (*pfWrite) (U8           Unit,
                U32          PageIndex,
                const void * pData,
                unsigned     Off,
                unsigned     NumBytes);
```

Parameter	Meaning
Unit	Unit number (0...N).
PageIndex	Zero-based index of page to be written.
pData	Pointer to a buffer of data which should be written.
Off	Byte offset within the page, which needs to be smaller than the page size.
NumBytes	Number of bytes which should be written.

Table 6.22: (*pfWrite)() parameter list

Return value

== 0: Data successfully transferred.
!= 0: An error has occurred.

6.3.1.9.2.7 (*pfWriteEx)()

Description

Writes data to 2 parts of a page. Typically used to write both the data and spare area of a page in one step.

Prototype

```
int (*pfWriteEx) (U8           Unit,
                  U32          PageIndex,
                  const void * pData,
                  unsigned     Off,
                  unsigned     NumBytes,
                  const void * pSpare,
                  unsigned     OffSpare,
                  unsigned     NumBytesSpare);
```

Parameter	Meaning
Unit	Unit number (0...N).
PageIndex	Number of page that should be written.
pData	Pointer to a buffer of data which should be written.
Off	Byte offset within the page, which needs to be smaller than the page size.
NumBytes	Number of bytes to write.
pSpare	Pointer to a buffer data which should be written to the spare area.
OffSpare	Offset from the start of the spare area to the point where spare data should be written. First byte of the spare area has the same offset as the page size. Example: Page size: 512 OffSpare == 512 -> First byte of spare area OffSpare == 513 -> Second byte of spare area Page size: 2048 OffSpare == 2048 -> First byte of spare area OffSpare == 2049 -> Second byte of spare area
NumBytesSpare	Number of spare bytes to write.

Table 6.23: (*pfWriteEx)() parameter list

Return value

== 0: Data successfully transferred.
!= 0: An error has occurred.

6.3.1.9.2.8 (*pfEnableECC)()

Description

This function activates the HW ECC.

Prototype

```
int (*pfEnableECC)(U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0...N).

Table 6.24: (*pfEnableECC)() parameter list

Return value

==0: HW ECC enabled.
!= 0: An error has occurred.

Additional information

With the HW ECC enabled the (*pfRead)() and (*pfReadEx)() functions will return corrected data. The function is called only by the Universal NAND driver.

6.3.1.9.2.9 (*pfDisableECC)()

Description

This function deactivates the HW ECC.

Prototype

```
int (*pfDisableECC)(U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0...N).

Table 6.25: (*pfDisableECC)() parameter list

Return value

==0: HW ECC disabled.
!= 0: An error has occurred.

Additional information

With the HW ECC disabled the (*pfRead)() and (*pfReadEx)() functions will return uncorrected data. The function is called only by the Universal NAND driver.

6.3.1.9.2.10 (*pfConfigureECC)()

Description

This function configures the HW ECC.

Prototype

```
int (*pfConfigureECC)(U8  Unit,
                      U8  NumBitsCorrectable,
                      U16 BytesPerECCBlock);
```

Parameter	Meaning
Unit	Unit number (0...N).
NumBitsCorrectable	Number of error bits the HW ECC should correct.
BytesPerECCBlock	The ECC is computed over this number of bytes.

Table 6.26: (*pfConfigureECC)() parameter list

Return value

==0: HW ECC configured.
!= 0: An error has occurred.

Additional information

This function is optional and is called only by the Universal NAND driver. It must be implemented only when the NAND flash device is interfaced to a NAND flash controller with HW ECC. The NAND flash controller should be configured to correct NumBitsCorrectable bit errors over BytesPerECCBlock bytes of data and spare area.

6.3.1.9.2.11 (*pfCopyPage)()

Description

Copies the contents of an entire NAND page (including the spare area) to an other page.

Prototype

```
int (*pfCopyPage)(U8 Unit, U32 PageNoSrc, U32 PageNoDest);
```

Parameter	Meaning
Unit	Unit number (0...N).
PageNoSrc	Index of the source page.
PageNoDest	Index of the destination page.

Table 6.27: (*pfCopyPage)() parameter list

Return value

==0: Page copied.
!= 0: An error occurred.

Additional information

This function is optional and is called only by the Universal NAND driver. It has to be implemented only when the NAND flash device supports HW ECC. The page should be copied by reading the source page to internal register of NAND flash device followed by a write to destination page.

6.3.1.9.2.12 FS_NAND_DEVICE_INFO

Description

This structure stores information about a NAND device.

Prototype

```
typedef struct FS_NAND_DEVICE_INFO {
  U8               BPP_Shift;
  U8               PPB_Shift;
  U16              NumBlocks;
  U16              BytesPerSpareArea;
  FS_NAND_ECC_INFO ECC_Info;
} FS_NAND_DEVICE_INFO;
```

Members	Description
BPP_Shift	Number of bytes in a page. Usually 9 (512 bytes) or 11 (2048 bytes).
PPB_Shift	Number of pages in NAND flash block. Usually 6 (64 pages) or 7 (128 pages).
NumBlocks	Number of NAND blocks.
BytesPerSpareArea	Number of bytes in the spare area. Usually 16 for 512 byte pages or 64 for 2048 byte pages. If 0 the NAND driver computes it from the page size as: BPP_Shift / 32
ECC_Info	Information about the ECC requirements. Usually taken from the ONFI parameters.

Table 6.28: FS_NAND_DEVICE_INFO - list of structure elements

6.3.1.9.2.13 FS_NAND_ECC_INFO

Description

This structure stores information about error correction requirements.

Prototype

```
typedef struct {
  U8 NumBitsCorrectable;
  U8 ldBytesPerBlock;
} FS_NAND_ECC_INFO;
```

Members	Description
NumBitsCorrectable	Number of bits the ECC should be able to correct.
ldBytesPerBlock	ECC must be computed over this number of data bytes. Usually 9 (512 bytes). In addition, the ECC should protect 4 bytes of spare area starting from byte offset 2.

Table 6.29: FS_NAND_ECC_INFO - list of structure elements

6.3.1.10 Hardware layer

6.3.1.10.1 Hardware functions - NAND flash

Routine	Explanation
FS_NAND_HW_X_SetAddrMode()	CLE low and ALE high for the specified device.
FS_NAND_HW_X_SetCmdMode()	CLE high and ALE low for the specified device.
FS_NAND_HW_X_SetDataMode()	CLE low and ALE low for the specified device.
FS_NAND_HW_X_DisableCE()	Disables CE.
FS_NAND_HW_X_EnableCE()	Enables CE.
FS_NAND_HW_X_WaitWhileBusy()	Waits while the device is busy.
FS_NAND_HW_X_Read_x8()	For 8-bit NAND flashes: Reads data from the NAND flash device.
FS_NAND_HW_X_Read_x16()	For 16-bit NAND flashes: Reads data from the NAND flash device.
FS_NAND_HW_X_Write_x8()	For 8-bit NAND flashes: Writing data to the NAND flash, using the I/O 0-7 lines of the NAND flash device.
FS_NAND_HW_X_Write_16()	For 16-bit NAND flashes: Writing data to the NAND flash, using the I/O 0-15 lines of the NAND flash device.
FS_NAND_HW_X_Init_x8()	For 8-bit NAND flashes: Initializes the NAND flash device.
FS_NAND_HW_X_Init_x16()	For 16-bit NAND flashes: Initializes the NAND flash device.

Table 6.30: NAND device driver hardware layer functions

6.3.1.10.1.1 FS_NAND_HW_X_SetAddrMode()

Description

Sets CLE low and ALE high for the specified device.

Prototype

```
void FS_NAND_HW_X_SetAddrMode (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0…N).

Table 6.31: FS_NAND_HW_X_SetAddrMode() parameter list

Additional Information

This function is called to start the address data transfer.

Example

```
void FS_NAND_HW_X_SetAddrMode(U8 Unit) {
  FS_USE_PARA(Unit);
  /* CLE low, ALE high */
  NAND_CLR_CLE();
  NAND_SET_ALE();}
}
```


6.3.1.10.1.2 FS_NAND_HW_X_SetCmdMode()

Description

Sets CLE high and ALE low for the specified device.

Prototype

```
void FS_NAND_HW_X_SetCmdMode (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0…N).

Table 6.32: FS_NAND_HW_X_SetCmdMode() parameter list

Additional Information

This function is called to start the command transfer.

Example

```
void FS_NAND_HW_X_SetCmdMode(U8 Unit) {
  FS_USE_PARA(Unit);
  /* CLE high, ALE low  */
  NAND_SET_CLE();
  NAND_CLR_ALE();
}
```


6.3.1.10.1.3 FS_NAND_HW_X_SetDataMode()

Description

Sets CLE low and ALE low for the specified device.

Prototype

```
void FS_NAND_HW_X_SetDataMode (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0…N).

Table 6.33: FS_NAND_HW_X_SetDataMode() parameter list

Additional Information

This function is called to the start data transfer.

Example

```
void FS_NAND_HW_X_SetData(U8 Unit) {
  FS_USE_PARA(Unit);
  /* CLE low, ALE low  */
  NAND_CLR_CLE();
  NAND_CLR_ALE();
}
```


6.3.1.10.1.4 FS_NAND_HW_X_DisableCE()

Description

Disables NAND CE.

Prototype

```
void FS_NAND_HW_X_DisableCE (U8 Unit);
```

Parameter	Description
Unit	Unit number (0…N).

Table 6.34: FS_NAND_HW_X_DisableCE() parameter list

6.3.1.10.1.5 FS_NAND_HW_X_EnableCE()

Description

Enables NAND CE.

Prototype

```
void FS_NAND_HW_X_EnableCE (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0…N).

Table 6.35: FS_NAND_HW_X_EnableCE() parameter list

Example

```
/*********************************************************************
*
*       FS_NAND_HW_X_EnableCE
*/
void FS_NAND_HW_X_EnableCE(U8 Unit) {
  PIOB_CODR = (1 << 18); // Enable NAND CE
}
```


6.3.1.10.1.6 FS_NAND_HW_X_WaitWhileBusy()

Description

Checks whether the device is busy.

Prototype

```
int FS_NAND_HW_X_WaitWhileBusy (U8       Unit,
                                unsigned us);
```

Parameter	Meaning
Unit	Unit number (0…N).
us	Time in µs to wait.

Table 6.36: FS_NAND_HW_X_WaitWhileBusy() parameter list

Return value

0 if the device is not busy.
Any other value means that an operation is pending.

Additional Information

If your hardware allows you to monitor the nR/B line, you can use the status of that line and return when the device is not busy. Otherwise, the function should return 1. In this case, the physical layer will perform a software-status-check of the device or wait for the time required by the current operation.

Example

```
int FS_NAND_HW_X_WaitWhileBusy(U8 Unit, unsinged us) {
  int IsReady;
  do {
    IsReady = NAND_GET_RDY() ? 0 : 1;
  } while(IsReady == 0);
  return IsReady;
}
```


6.3.1.10.1.7 FS_NAND_HW_X_Read_x8()

Description

Reads data from an 8-bit NAND flash device, using the I/O 0-7 lines.

Prototype

```
void FS_NAND_HW_X_Read_x8 (U8        Unit,
                           U8 *      pBuffer,
                           unsigned  NumBytes);
```

Parameter	Meaning
Unit	Unit number (0…N).
pBuffer	Pointer to a buffer to store the read data.
NumBytes	Number of bytes that should be stored into the buffer.

Table 6.37: FS_NAND_HW_X_Read_x8() parameter list

Additional Information

When reading from the device, usually you will not have to take care of handling the RE line because that is done automatically by the hardware.
If you do have to control the RE line, make sure that timing is according to your NAND flash device specification.

Example

```
void FS_NAND_HW_X_Read_x8(U8 Unit, U8 * pBuffer, unsigned NumBytes) {
  SET_DATA2INPUT();
  do {
    NAND_CLR_RE();      /*  RE is active low */
    NAND_GET_DATA(*pBuffer);
    pBuffer++;
    NAND_SET_RE();     /* disable RE */
  } while (--NumBytes);
}
```


6.3.1.10.1.8 FS_NAND_HW_X_Read_x16()

Description

Reads data from a 16-bit NAND flash device, using the I/O 0-15 lines.

Prototype

```
void FS_NAND_HW_X_Read_x16 (U8        Unit,
                            U8 *      pBuffer,
                            unsigned  NumBytes);
```

Parameter	Meaning
Unit	Unit number (0…N).
pBuffer	Pointer to a buffer to store the read data.
NumBytes	Number of bytes that should be stored into the buffer.

Table 6.38: FS_NAND_HW_X_Read_x16() parameter list

Additional Information

When reading from the device, usually you will not have to take care of handling the RE line because that is done automatically by the hardware.
If you do have to control the RE line, make sure that timing is according to your NAND flash device specification.

6.3.1.10.1.9 FS_NAND_HW_X_Write_x8()

Description

Writes data to an 8-bit NAND flash, using the I/O 0-7 lines of the NAND flash device.

Prototype

```
void FS_NAND_HW_X_Write_x8 (U8         Unit,
                            const U8 * pBuffer,
                            unsigned   NumBytes);;
```

Parameter	Meaning
Unit	Unit number (0…N).
pBuffer	Pointer to a buffer of data which should be written.
NumBytes	Number of bytes that should be transferred to the NAND flash.

Table 6.39: FS_NAND_HW_X_Write_x8() parameter list

Additional Information

When writing data to the device, usually you will not have to take care of handling the WE line because that is done automatically by the hardware.
If you do have to control the WE line, make sure that timing is according to your NAND flash device specifications.

Example

```
void FS_NAND_HW_X_Write_x8(U8 Unit, U8 * pBuffer, unsigned NumBytes) {
  SET_DATA2OUTPUT();
  do {
    NAND_CLR_WE();      /*  WE is active low */
    NAND_SET_DATA(*pBuffer);
    pBuffer++;
    NAND_SET_WE();     /* disable WE */
  } while (--NumBytes);
 }
```


6.3.1.10.1.10 FS_NAND_HW_X_Write_x16()

Description

Writing data to a 16-bit NAND flash, using the I/O 0-15 lines of the NAND flash device.

Prototype

```
void FS_NAND_HW_X_Write_x16 (U8          Unit,
                             const U8 * pBuffer,
                             unsigned   NumBytes);;
```

Parameter	Meaning
Unit	Unit number (0…N).
pBuffer	Pointer to a buffer of data which should be written.
NumBytes	Number of bytes that should be transferred to the NAND flash.

Table 6.40: FS_NAND_HW_X_Write_x16() parameter list

Additional Information

When writing data to the device, usually you will not have to take care of handling the WE line because that is done automatically by the hardware.
If you do have to control the WE line, make sure that timing is according to your NAND flash device specifications.

6.3.1.10.1.11 FS_NAND_HW_X_Init_x8()

Description

Initializes a NAND flash device with an 8-bit interface.

Prototype

```
void FS_NAND_HW_X_Init_x8 (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0…N).

Table 6.41: FS_NAND_HW_X_Init_x8() parameter list

Additional Information

This function is called before any access to the NAND flash device is made. Use this function to initialize the hardware.

Example

```
int FS_NAND_HW_X_Init_x8(U8 Unit) {
 FS_USE_PARA(Unit);
 _Timer2Config();
 _NANDFlashInit();
}
```


6.3.1.10.1.12 FS_NAND_HW_X_Init_x16()

Description

Initializes a NAND flash device with a 16-bit interface.

Prototype

```
void FS_NAND_HW_X_Init_x16 (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0…N).

Table 6.42: FS_NAND_HW_X_Init_x16() parameter list

Additional Information

This function is called before any access to the NAND flash device is made. Use this function to initialize the hardware.

6.3.1.10.2 Hardware functions - ATMEL DataFlash

Routine	Explanation
Control line functions	
FS_DF_HW_X_EnableCS()	Activates chip select signal (CS) of the DataFlash chip.
FS_DF_HW_X_DisableCS()	Deactivates chip select signal (CS) of the DataFlash chip.
FS_DF_HW_X_Init()	Initializes the SPI hardware.
Data transfer functions	
FS_DF_HW_X_Read()	Receives a number of bytes from the DataFlash.
FS_DF_HW_X_Write()	Sends a number of bytes to the DataFlash.

Table 6.43: DataFlash device driver hardware functions

6.3.1.10.2.1 FS_DF_HW_X_EnableCS()

Description

Activates chip select signal (CS) of the specified DataFlash.

Prototype

```
void FS_DF_HW_X_EnableCS (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0…N).

Table 6.44: FS_DF_HW_X_EnableCS() parameter list

Additional Information

The CS signal is used to address a specific DataFlash chip connected to the SPI. Enabling is equal to setting the CS line to low.

Example

```
void FS_DF_HW_X_EnableCS(U8 Unit) {
  SPI_CLR_CS();
}
```


6.3.1.10.2.2 FS_DF_HW_X_DisableCS()

Description

Deactivates chip select signal (CS) of the specified DataFlash.

Prototype

```
void FS_DF_HW_X_DisableCS (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0…N).

Table 6.45: FS_DF_HW_X_DisableCS() parameter list

Additional Information

The CS signal is used to address a specific DataFlash connected to the SPI. Disabling is equal to setting the CS line to high.

Example

```
void FS_DF_HW_X_DisableCS(U8 Unit) {
  SPI_SET_CS();
}
```


6.3.1.10.2.3 FS_DF_HW_X_Init()

Description

Initializes the SPI hardware.

Prototype

```
int FS_DF_HW_X_Init (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0…N).

Table 6.46: FS_DF_HW_X_Init() parameter list

Return value

== 0 Initialization was successful.
== 1 Initialization failed.

Additional Information

The `FS_DF_HW_X_Init()` can be used to initialize the SPI hardware. As described in the previous section. The SPI should be initialized as follows:

- 8-bit data length
- MSB should be sent out first
- CS signal should be initially high
- The set clock frequency should not exceed the max clock frequency that are specified by the Serial Flash devices (Usually: 20MHz).

Example

```
void FS_DF_HW_X_Init(U8 Unit) {
  SPI_SETUP_PINS();
}
```


6.3.1.10.2.4 FS_DF_HW_X_Read()

Description

Receives a number of bytes from the DataFlash.

Prototype

```
void FS_DF_HW_X_Read (U8   Unit,
                      U8 * pData,
                      int  NumBytes);
```

Parameter	Meaning
Unit	Unit number (0…N).
pData	Pointer to a buffer for data to be receive.
NumBytes	Number of bytes to receive.

Table 6.47: FS_DF_HW_X_Read() parameter list

Example

```
void FS_DF_HW_X_Read (U8 Unit, U8 * pData, int NumBytes) {
  do {
    c = 0;
    bpos = 8; /* get 8 bits */
    do {
      SPI_CLR_CLK();
      c <<= 1;
      if (SPI_DATAIN()) {
        c |= 1;
      }
      SPI_SET_CLK();
    } while (--bpos);
    *pData++ = c;
  } while (--NumBytes);
}
```


6.3.1.10.2.5 FS_DF_HW_X_Write()

Description

Sends a number of bytes from memory buffer to the dedicated DataFlash.

Prototype

```
void FS_DF_HW_X_Write (U8         Unit,
                       const U8 * pData,
                       int        NumBytes);
```

Parameter	Meaning
Unit	Unit number (0…N).
pData	Pointer to a buffer for data to be receive.
NumBytes	Number of bytes to be written.

Table 6.48: FS_DF_HW_X_Write() parameter list

Example

```
void FS_DF_HW_X_Write(U8 Unit, const U8 * pData, int NumBytes) {
  int i;
  U8 mask;
  U8 data;
  for (i = 0; i < NumBytes; i++) {
    data = pData[i];
    mask = 0x80;
    while (mask) {
      if (data & mask) {
        SPI_SET_DATAOUT();
      } else {
        SPI_CLR_DATAOUT();
      }
      SPI_CLR_CLK();
      SPI_DELAY();
      SPI_SET_CLK();
      SPI_DELAY();
      mask >>= 1;
    }
  }
  SPI_SET_DATAOUT(); /* default state of data line is high */
}
```


6.3.1.10.3 Hardware functions - SPI NAND flash

Routine	Explanation
FS_NAND_HW_SPI_X_Delay()	Blocks the execution for the specified number of milliseconds.
FS_NAND_HW_SPI_X_DisableCS()	Deactivates chip select signal (CS) of the NAND flash.
FS_NAND_HW_SPI_X_EnableCS()	Activates chip select signal (CS) of the NAND flash.
FS_NAND_HW_SPI_X_Init()	Initializes the SPI interface.
FS_NAND_HW_SPI_X_Read()	Receives a number of bytes from the NAND flash.
FS_NAND_HW_SPI_X_Write()	Sends a number of bytes to the NAND flash.

Table 6.49: Hardware functions - SPI NAND flash

6.3.1.10.3.1 FS_NAND_HW_SPI_X_Delay()

Description

Blocks the execution for the specified number of milliseconds.

Prototype

```
void FS_NAND_HW_SPI_X_Delay(U8 Unit, int ms);
```

Parameter	Meaning
Unit	Unit number (0-based).
ms	Number of milliseconds to wait.

Table 6.50: FS_NAND_HW_SPI_X_Delay() parameter list

Additional information

The function is called after a reset command is sent to NAND flash. The routine can delay longer that the number of milliseconds specified.

6.3.1.10.3.2 FS_NAND_HW_SPI_X_DisableCS()

Description

Disables the access to NAND flash.

Prototype

```
void FS_NAND_HW_SPI_X_DisableCS(U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0-based).

Table 6.51: FS_NAND_HW_SPI_X_DisableCS() parameter list

Additional information

Typically, the CS signal is active low which means that the CS signal must be held high to disable the NAND flash. The NAND flash ignores any command sent with the CS signal disabled.

6.3.1.10.3.3 FS_NAND_HW_SPI_X_EnableCS()

Description

Enables the access to NAND flash.

Prototype

```
void FS_NAND_HW_SPI_X_EnableCS(U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0-based).

Table 6.52: FS_NAND_HW_SPI_X_EnableCS() parameter list

Additional information

Typically, the CS signal is active low which means that the CS signal must be held low to enable the NAND flash.

6.3.1.10.3.4 FS_NAND_HW_SPI_X_Init()

Description

Performs the initialization of SPI interface.

Prototype

```
void FS_NAND_HW_SPI_X_Init(U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0-based).

Table 6.53: FS_NAND_HW_SPI_X_Init() parameter list

Additional information

This function is called before any other function of the HW layer. It should be used to initialize the HW.

6.3.1.10.3.5 FS_NAND_HW_SPI_X_Read()

Description

Receives a number of bytes form NAND flash via SPI.

Prototype

```
void FS_NAND_HW_SPI_X_Read(U8 Unit, void * pData, unsigned NumBytes);
```

Parameter	Meaning
Unit	Unit number (0-based).
pData	IN: --- OUT: Data received from NAND flash
NumBytes	Number of bytes to be received

Table 6.54: FS_NAND_HW_SPI_X_Read() parameter list

6.3.1.10.3.6 FS_NAND_HW_SPI_X_Write()

Description

Sends a number of bytes to NAND flash via SPI.

Prototype

```
void FS_NAND_HW_SPI_X_Write(U8 Unit, const void * pData, unsigned NumBytes);
```

Parameter	Meaning
Unit	Unit number (0-based).
pData	IN: Data to be sent to NAND flash OUT: ---
NumBytes	Number of bytes to be sent

Table 6.55: FS_NAND_HW_SPI_X_Write() parameter list

6.3.1.11 Additional driver functions

Routine	Explanation
FS_NAND_GetDiskInfo()	Delivers information about NAND flash.
FS_NAND_GetBlockInfo()	Delivers information about a physical block of NAND flash.

Table 6.56: FS_NAND_Driver - list of additional functions.

6.3.1.11.1 FS_NAND_GetDiskInfo()

Description

Returns information about the NAND flash.

Prototype

```
void FS_NAND_GetDiskInfo(U8 Unit, FS_NAND_DISK_INFO * pDiskInfo);
```

Parameter	Meaning
Unit	Unit number of driver.
pDiskInfo	IN: --- OUT: Information about NAND flash.

Table 6.57: FS_NAND_GetDiskInfo() parameter list

Example

```
void ShowDiskInfo(U32 Unit) {
  FS_NAND_DISK_INFO DiskInfo;

  printf("Retrieving disk information for nand:%d:\n", Unit);
  FS_NAND_GetDiskInfo((U8)Unit, &DiskInfo);
  printf("  NumPhyBlocks       = %d\n"
         "  NumLogBlocks       = %d\n"
         "  NumPagesPerBlock   = %d\n"
         "  NumSectorsPerBlock = %d\n"
         "  BytesPerPage       = %d\n"
         "  BytesPerSector     = %d\n"
         "  NumUsedPhyBlocks   = %d\n"
         "  NumBadPhyBlocks    = %d\n"
         "  EraseCntMin        = %u\n"
         "  EraseCntMax        = %u\n"
         "  EraseCntAvg        = %u\n"
         "  IsWriteProtected   = %d\n"
         "  HasFatalError      = %d\n"
         "  ErrorType          = %d\n"
         "  ErrorSectorIndex   = %d\n", DiskInfo.NumPhyBlocks,
                                        DiskInfo.NumLogBlocks,
                                        DiskInfo.NumPagesPerBlock,
                                        DiskInfo.NumSectorsPerBlock,
                                        DiskInfo.BytesPerPage,
                                        DiskInfo.BytesPerSector,
                                        DiskInfo.NumUsedPhyBlocks,
                                        DiskInfo.NumBadPhyBlocks,
                                        DiskInfo.EraseCntMin,
                                        DiskInfo.EraseCntMax,
                                        DiskInfo.EraseCntAvg,
                                        DiskInfo.IsWriteProtected,
                                        DiskInfo.HasFatalError,
                                        DiskInfo.ErrorType,
                                        DiskInfo.ErrorSectorIndex);
}
```


6.3.1.11.2 FS_NAND_GetBlockInfo()

Description

Returns information about a physical block of NAND flash.

Prototype

```
void FS_NAND_GetBlockInfo(U8                    Unit,
                          U32                   PhyBlockIndex,
                          FS_NAND_BLOCK_INFO * pBlockInfo);
```

Parameter	Meaning
Unit	Unit number of driver.
PhyBlockIndex	Index of the physical block.
pDiskInfo	IN: --- OUT: Information about the physical block.

Table 6.58: FS_NAND_GetBlockInfo() parameter list

Example

```
void _ShowBlockInfo(U32 Unit, U32 PhyBlockIndex) {
  FS_NAND_BLOCK_INFO BlockInfo;

  printf("Retrieving block information for nand:%d:, block index: 0x%.8x\n",
          Unit, PhyBlockIndex);
  FS_NAND_GetBlockInfo((U8)Unit, PhyBlockIndex, &BlockInfo);
  printf("  sType                     = %s\n"
         "  EraseCnt                  = 0x%.8x\n"
         "  lbi                       = %d\n"
         "  NumSectorsBlank           = %d\n"
         "  NumSectorsECCCorrectable  = %d\n"
         "  NumSectorsErrorInECC      = %d\n"
         "  NumSectorsECCError        = %d\n"
         "  NumSectorsInvalid         = %d\n"
         "  NumSectorsValid           = %d\n", BlockInfo.sType,
                                               BlockInfo.EraseCnt,
                                               BlockInfo.lbi,
                                               BlockInfo.NumSectorsBlank,
                                               BlockInfo.NumSectorsECCCorrectable,
                                               BlockInfo.NumSectorsErrorInECC,
                                               BlockInfo.NumSectorsECCError,
                                               BlockInfo.NumSectorsInvalid,
                                               BlockInfo.NumSectorsValid);
}
```


6.3.1.11.3 FS_NAND_DISK_INFO

Description

The structure contains information about the NAND flash.

Declaration

```
typedef struct {
  U32 NumPhyBlocks;
  U32 NumLogBlocks;
  U32 NumUsedPhyBlocks;
  U32 NumBadPhyBlocks;
  U32 NumPagesPerBlock;
  U32 NumSectorsPerBlock;
  U32 BytesPerPage;
  U32 BytesPerSector;
  U32 EraseCntMin;
  U32 EraseCntMax;
  U32 EraseCntAvg;
  U8  IsWriteProtected;
  U8  HasFatalError;
  U8  ErrorType;
  U32 ErrorSectorIndex;
} FS_NAND_DISK_INFO;
```

Members	Description
NumPhyBlocks	Number of physical NAND flash blocks managed by the driver.
NumLogBlocks	Number of blocks available to file system.
NumUsedPhyBlocks	Number of physical blocks currently in use.
NumBadPhyBlocks	Number of physical blocks marked as bad.
NumPagesPerBlock	Number of pages in a NAND flash block. Typ. 64 or 256.
NumSectorsPerBlock	Number of file system sectors that fit in a NAND flash block.
BytesPerPage	Number of bytes in a NAND flash page. Typ. 512 or 2048.
BytesPerSector	Number of bytes in a file system sector.
EraseCntMin	Smallest erase count from all the physical blocks.
EraseCntMax	Greatest erase count from all the physical blocks.
EraseCntAvg	Average erase count from all the physical blocks.
IsWriteProtected	Set to 1 to indicate that the file system is not allowed to write to NAND flash.
HasFatalError	Set to 1 to indicate that the data stored to NAND flash is corrupted.
ErrorType	Type of fatal error encountered.
ErrorSectorIndex	Index of the physical sector where the fatal error occurred.

Table 6.59: FS_NAND_DISK_INFO - list of structure elements

6.3.1.11.4 FS_NAND_BLOCK_INFO

Description

The structure contains information about the NAND flash.

Declaration

```
typedef struct {
  U32          EraseCnt;
  U32          lbi;
  U16          NumSectorsBlank;
  U16          NumSectorsValid;
  U16          NumSectorsInvalid;
  U16          NumSectorsECCError;
  U16          NumSectorsECCCorrectable;
  U16          NumSectorsErrorInECC;
  const char * sType;
  U8           Type;
} FS_NAND_BLOCK_INFO;
```

Members	Description
EraseCnt	Number of times the block has been erased.
lbi	Logical index of the physical block.
NumSectorsBlank	Number of sectors that were not written yet.
NumSectorsValid	Number of sectors that contain valid data.
NumSectorsInvalid	Number of sectors that have been invalidated.
NumSectorsECCError	Number of sectors where more bit errors are present than the ECC is not able to correct.
NumSectorsECCCorrectable	Number of sectors where bit errors were found and corrected.
NumSectorsErrorInECC	Number of sectors where bit errors were found in the ECC itself.
sType	Pointer to a 0 terminated string holding the block type.
Type	Type of data block. Can take one of these values: • NAND_BLOCK_TYPE_UNKNOWN • NAND_BLOCK_TYPE_BAD • NAND_BLOCK_TYPE_EMPTY • NAND_BLOCK_TYPE_WORK • NAND_BLOCK_TYPE_DATA

Table 6.60: FS_NAND_BLOCK_INFO - list of structure elements

6.3.1.12 Test hardware

The SEGGER "NAND-Flash EVAL" board is an easy to use and cost effective testing tool designed to evaluate the features and the performance of the emFile NAND driver.

The NAND driver can be used with emFile or emUSB-Device, in which case the board behaves like a Mass Storage Device (USB-Stick).

Common evaluation boards are usually used to perform these tests but this approach brings several disadvantages. Software and hardware development tools are required to build and load the application into the target system. Moreover, the tests are restricted to the type of NAND flash which is soldered on the board.

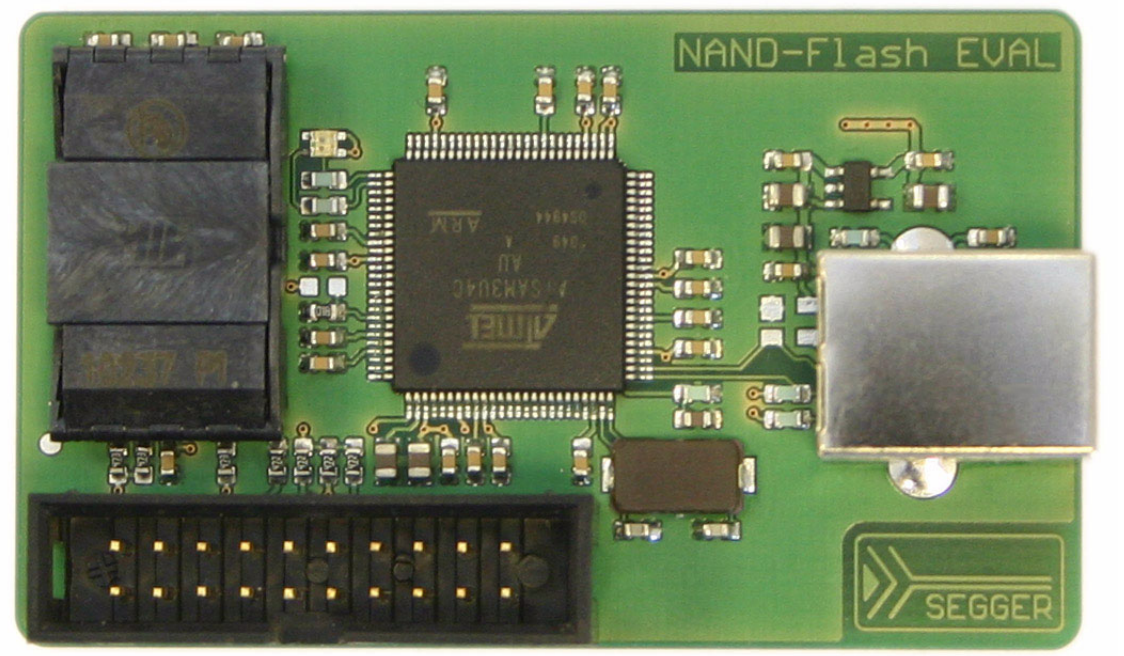
The "NAND-Flash EVAL" board was designed to overcome these limitations and provides the user with an affordable alternative.

The main feature is that the NAND flash is not directly soldered on the board. A 48-pin TSOP socket is used instead which allows the user to experiment with different types of NAND flashes. This helps finding the right NAND flash for an application and thus reducing costs.

A further important feature is that the "NAND-Flash EVAL" board comes preloaded with a USB-MSD application. When connected to a PC over USB, the board shows up as a removable storage on the host operating system. Performance and functionality tests of NAND flash can be performed in this way without the need of an expensive development environment. All current operating systems will recognize the board out of the box.

Trial software packages

The "NAND-Flash EVAL" board comes with a ready to use USB-MSD application in binary form. emFile is provided in object code form together with a start project which can be easily modified to create custom applications. For programming and debugging a JTAG debug probe like J-Link is required. The package also contains the schematics of the board.



Feature list

- Atmel ATSAM3U4C ARM Cortex-M3 microcontroller
- NAND flash socket
- 2 color LED
- 20-pin JTAG header
- High speed USB interface

- USB powered

6.3.1.13 Performance and resource usage

6.3.1.13.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the NAND driver presented in the tables below have been measured on a system as follows: ARM7, IAR Embedded workbench V4.41A, Thumb mode, Size optimization.

Module	ROM [Kbytes]
emFile NAND driver	4.5

In addition, one of the following physical layers is required:

Physical layer	Description	ROM [Kbytes]
FS_NAND_PHY_512x8	Physical layer for small NAND devices with an 8-bit interface.	1.1
FS_NAND_PHY_2048x8	Physical layer for large NAND devices with an 8-bit interface.	1.0
FS_NAND_PHY_2048x16	Physical layer for large NAND devices with an 16-bit interface.	1.0
FS_NAND_PHY_x8	Physical layer for large and small NAND devices with an 8-bit interface.	2.3
FS_NAND_PHY_x	Physical layer for large and small NAND devices with an 8-bit or 16-bit interface.	3.3
FS_NAND_PHY_ONFI	Physical layer for NAND flashes which support ONFI.	1.5

6.3.1.13.2 Static RAM usage

Static RAM usage is the amount of RAM required by the driver for variables inside of the driver. The number of bytes can be seen in a compiler list file

Static RAM usage of driver: 32 bytes

6.3.1.13.3 Runtime RAM usage

Runtime RAM usage is the amount of RAM allocated by the driver at runtime. The amount required depends on the runtime configuration and the connected device.

The approximately RAM usage for the NAND driver can be calculated as follows:

Every NAND device requires:
`160 + 2 * NumberOfUsedBlocks + 4 * SectorsPerBlock + 1.04 * MaxSectorSize`

Example: 2 GBit NAND flash with 2K pages, 2048 blocks used, 512-byte sectors

One block consists of 64 pages, each page holds 4 sectors of 512 bytes.

SectorsPerBlock = 256
NumberOfUsedBlocks = 2048
MaxSectorSize = 512

RAM usage = (160 + 2 * 2048 + 4 * 256 + 1.04 * 512) bytes
RAM usage = 5813 bytes

Example: 2 GBit NAND flash with 2K pages, 2048 blocks used, 2048-byte sectors

One block consists of 64 pages, each page holds 1 sector of 2048 bytes.

SectorsPerBlock = 64
NumberOfUsedBlocks = 2048
MaxSectorSize = 2048

RAM usage = (160 + 2 * 2048 + 4 * 64 + 1.04 * 2048) bytes
RAM usage = 6642bytes

Example: 512 MBit NAND flash with 512 pages, 4096 blocks used, 512-byte sectors

One block consists of 64 pages, each page holds 1 sector of 512 bytes.

SectorsPerBlock = 32
NumberOfUsedBlocks = 8192
MaxSectorSize = 512

RAM usage = (160 + 2 * 4096 + 4 * 32 + 1.04 * 512) bytes
RAM usage = 9013 bytes

6.3.1.13.4 Performance

These performance measurements are in no way complete, but they give an approximation of the length of time required for common operations on various targets. The tests were performed as described in *Performance* on page 502.

All values are in Mbytes/sec.

Device	CPU speed	Medium	W	R
Atmel AT91SAM7S	48 MHz	NAND flash with 512 bytes per page using Port mode.	0.8	2.0
Atmel AT91SAM7S	48 MHz	NAND flash with 2048 bytes per page and a sector size of 512 bytes using Port mode.	0.7	2.0
Atmel AT91SAM7S	48 MHz	NAND flash with 2048 bytes per page and a sector size of 2048 bytes using the built-in NAND controller/ external bus-interface.	1.3	2.3
Atmel AT91SAM7SE	48 MHz	NAND flash with 2048 bytes per page and a sector size of 512 bytes using the built-in NAND controller/ external bus-interface.	1.6	3.1
Atmel AT91SAM7SE	48 MHz	NAND flash with 2048 bytes per page and a sector size of 2048 bytes using the built-in NAND controller/ external bus-interface.	3.8	5.9
Atmel AT91SAM9261	200 MHz	NAND flash with 2048 bytes per page and a sector size of 512 bytes using the built-in NAND controller/ external bus-interface.	2.3	6.5
Atmel AT91SAM9261	200 MHz	NAND flash with 2048 bytes per page and a sector size of 2048 bytes using the built-in NAND controller/ external bus-interface.	5.1	9.8

Table 6.61: Performance values for sample configurations

Device	CPU speed	Medium	W	R
Atmel AT91SAM9G45	384 MHz	NAND flash with 2048 bytes per page and a sector size of 2048 bytes using the built-in NAND controller/ external bus-interface.	4.7	12.4
Atmel AT91SAM3U	96 MHz	NAND flash with 2048 bytes per page and a sector size of 2048 bytes using the built-in NAND controller/ external bus-interface.	5	5.9
Atmel AT91SAM3U	96 MHz	NAND flash with 2048 bytes per page and a sector size of 512 bytes using the built-in NAND controller/ external bus-interface.	2.3	4.7

Table 6.61: Performance values for sample configurations

6.3.1.14 FAQs

Q: Are Multi-Level Cell NAND flashes (MLCs) supported?
A: Yes, the Universal NAND driver does support MLCs.

Q: Are NAND flashes with 4-Kbytes pages supported?
A: Yes, they are supported. The `FS_NAND_PHY_4096x8` physical layer should be used.

6.3.2 Universal driver - FS_NAND_UNI_Driver

This driver for NAND flashes is designed to support SLC and MLC NAND flashes. It can correct multiple bit errors by using the internal ECC of NAND flashes or by calling ECC computation routines provided by the application. The ECC protects the sector data and the driver management data stored in the spare area of a page. Sector size is equal to page size and must be at least 2048 bytes. Smaller sector sizes are possible using an additional file system layer. The driver requires very little RAM and is extremely efficient.

This section first describes which devices are supported and describes all hardware access functions required by the NAND flash driver.

6.3.2.1 Supported hardware

In general, the driver supports almost all Single-Level Cell NAND flashes (SLC) with a page size greater than 2048+64 bytes.

The table below shows the NAND flashes that have been tested or are compatible with a tested device:

Manufacturer	Device	Page size [Bytes]	Size [Bits]	Internal ECC
Hynix	HY27UF082G2M	2048+64	256Mx8	no
	HY27UF084G2M	2048+64	512Mx8	no
	HY27UG084G2M	2048+64	512Mx8	no
	HY27UG084GDM	2048+64	512Mx8	no
Micron	MT29F2G08AAB	2048+64	256Mx8	no
	MT29F2G08ABD	2048+64	256Mx8	no
	MT29F4G08AAA	2048+64	512Mx8	no
	MT29F4G08BAB	2048+64	512Mx8	no
	MT29F2G16AAD	2048+64	128Mx16	no
	MT29F2G08ABAEA	2048+64	2Gx8	yes
	MT29F8G08ABABA	4069+224	8Gx8	no
	MT29F1G01AAADD	2048+64	1Gx1	yes
	MT29F2G01AAAED	2048+64	2Gx1	yes
	MT29F4G01AAADD	2048+64	4Gx1	yes
Samsung	K9F1G08x0A	2048+64	256Mx8	no
	K9F2G08U0M	2048+64	256Mx8	no
	K9K2G08R0A	2048+64	256Mx8	no
	K9K2G08U0M	2048+64	256Mx8	no
	K9F4G08U0M	2048+64	512Mx8	no
	K9F8G08U0M	2048+64	1024Mx8	no
ST-Microelectronics	NAND01GR3B	2048+64	128Mx8	no
	NAND01GW3B	2048+64	128Mx8	no
	NAND02GR3B	2048+64	256Mx8	no
	NAND02GW3B	2048+64	256Mx8	no
	NAND04GW3	2048+64	512Mx8	no

Table 6.62: List of supported NAND flashes

Support for devices not in this list

Most other NAND flash devices are compatible with one of the supported devices. Thus, the driver can be used with these devices or may only need a little modification, which can be easily done. Get in touch with us, if you have questions about support for devices not in this list.

Additional information

For a description of the NAND flash hardware interface, refer to *Pin description - NAND flashes* on page 225. Sample schematics showing how to connect more than one NAND flash to a single MCU can be found on the chapter *Sample block schematics* on page 227.

6.3.2.2 Theory of operation

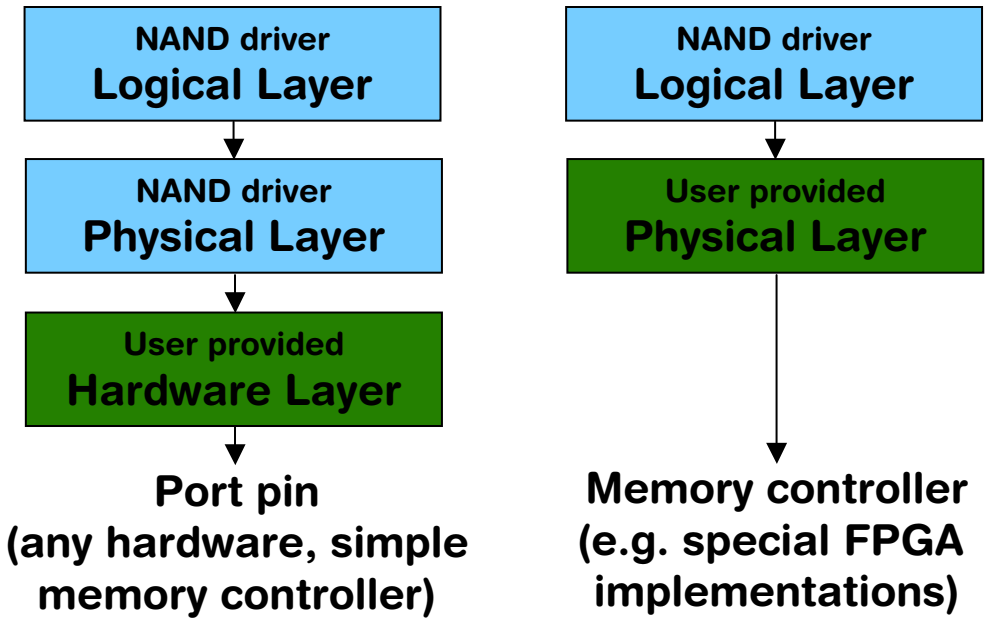
NAND flash devices are divided into physical blocks and physical pages. One physical block is the smallest erasable unit; one physical page is the smallest writable unit. Large block NAND Flash devices contain blocks made up of 64 pages, each page containing 2112 bytes (2048 data bytes + 64 spare bytes). The first page of a block is reserved for management data.

The driver uses the spare bytes for the following purposes:

1. To check if the block is valid.
   If they are valid the driver uses this sector. When the driver detects a bad sector, the whole block is marked as invalid and its content is copied to a non-defective block.
2. To store/load management information
   This includes the mapping of pages to logical sectors, the number of times a block has been erased and whether a page contains valid data or not.
3. To store/load an ECC (Error Correction Code) for data reliability.
   When reading a sector, the driver also reads the ECC stored in the spare area of the sector, calculates the ECC based on the read data and compares the ECCs. If the ECCs are not identical, the driver tries to recover the data, based on the read ECC.
   When writing to a page the ECC is calculated based on the data the driver has to write to the page. The calculated ECC is then stored in the spare area.

6.3.2.2.1 Software structure

The NAND Flash driver is split up into different layers, which are shown in the illustration below.



It is possible to use the NAND driver with custom hardware. If port pins or a simple memory controller are used for accessing the flash memory, only the hardware layer needs to be ported, normally no changes to the physical layer are required. If the NAND driver should be used with a special memory controller (for example special FPGA implementations), the physical layer needs to be adapted. In this case, the hardware layer is not required, because the memory controller manages the hardware access.

6.3.2.3 Fail-safe operation

The emFile NAND driver is fail-safe. That means that the driver makes only atomic actions and takes the responsibility that the data managed by the file system is always valid. In case of a power loss or a power reset during a write operation, it is always assured that only valid data is stored in the flash. If the power loss interrupts the write operation, the old data will be kept and the data is not corrupted.

For additional information, refer to *Fail-safe operation* on page 228.

6.3.2.4 Wear leveling

Wear leveling is supported by the driver. The procedure ensures that the number of erase cycles remains approximately for all the blocks. The maximum allowed erase count difference is runtime configurable and is by default 5000.

6.3.2.5 Partial writes

The driver writes only once in any page of the NAND flash between two block erase cycles. The number of partial writes is 1 making the driver conform with any SLC/ MLC device.

6.3.2.6 Configuring the driver

6.3.2.6.1 Adding the driver to emFile

To add the driver, use `FS_AddDevice()` with the driver label `FS_NAND_UNI_Driver`. This function has to be called from within `FS_X_AddDevices()`. Refer to *FS_X_AddDevices()* on page 472 for more information.

Example

```
void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  FS_AddDevice(&FS_NAND_UNI_Driver);
  //
  // Set the physical interface of the NAND flash.
  //
  FS_NAND_UNI_SetPhyType(0, &FS_NAND_PHY_ONFI);
  //
  // Configure the driver to use the internal ECC of NAND flash for error correction.
  //
  FS_NAND_UNI_SetECCHook(0, &FS_NAND_ECC_HW_NULL);
}
```

6.3.2.6.2 Specific configuration functions

Routine	Explanation
`FS_NAND_UNI_SetPhyType()`	Configures the physical type of NAND device.
`FS_NAND_UNI_SetBlockRange()`	Configures the range of physical blocks managed by the driver.
`FS_NAND_UNI_SetMaxEraseCntDiff()`	Configures the threshold for the wear-leveling.
`FS_NAND_UNI_SetOnFatalErrorCallback()`	Configures a function to be invoked when the driver encounters a fatal error.
`FS_NAND_UNI_SetNumWorkBlocks()`	Configures the number of work blocks.
`FS_NAND_UNI_SetECCHook()`	Configures the ECC algorithm.

Table 6.63: FS_NAND_UNI_Driver - list of configuration functions.

6.3.2.6.2.1 FS_NAND_UNI_SetPhyType()

Description

Sets the physical type of the device. NAND flash is organized in pages of either 512 or 2048 bytes and has an 8-bit or 16-bit interface. The driver needs to know the correct combination of page and interface width.

Prototype

```
void FS_NAND_UNI_SetPhyType(U8                         Unit,
                            const FS_NAND_PHY_TYPE * pPhyType);
```

Parameter	Meaning
Unit	Unit number.
pPhyType	IN: Physical type of device. OUT: ---

Table 6.64: FS_NAND_UNI_SetPhyType() parameter list

For additional information, refer to *FS_NAND_SetPhyType()* on page 231.

6.3.2.6.2.2 FS_NAND_UNI_SetBlockRange()

Description

Sets a limit for which blocks of the NAND flash can be controlled by the driver.

Prototype

```
void FS_NAND_UNI_SetBlockRange(U8  Unit,
                               U16 FirstBlock,
                               U16 MaxNumBlocks);
```

Parameter	Meaning
Unit	Unit number.
FirstBlock	Zero-based index of the first block to use. Specifies the number of blocks at the beginning of the device to skip. 0 means that no blocks are skipped.
MaxNumBlocks	Maximum number of blocks to use. 0 means use all blocks after FirstBlock.

Table 6.65: FS_NAND_UNI_SetBlockRange() parameter list

For additional information, refer to *FS_NAND_SetBlockRange()* on page 233.

6.3.2.6.2.3 FS_NAND_UNI_SetMaxEraseCntDiff()

Description

Sets the maximum difference between block erase counts that triggers the active wear leveling.

Prototype

```
void FS_NAND_UNI_SetMaxEraseCntDiff(U8  Unit,
                                    U32 EraseCntDiff);
```

Parameter	Meaning
Unit	Unit number.
EraseCntDiff	Maximum allowed difference between the erase counts.

Table 6.66: FS_NAND_UNI_SetMaxEraseCntDiff() parameter list

For additional information, refer to *FS_NAND_SetMaxEraseCntDiff()* on page 234.

6.3.2.6.2.4 FS_NAND_UNI_SetOnFatalErrorCallback()

Description

Registers a function that should be invoked when a fatal error occurs.

Prototype

```
void FS_NAND_UNI_SetOnFatalErrorCallback(
     FS_NAND_ON_FATAL_ERROR_CALLBACK * pfOnFatalError);
```

Parameter	Meaning
pfOnFatalError	Pointer to callback function.

Table 6.67: FS_NAND_UNI_SetOnFatalErrorCB() parameter list

For additional information, refer to *FS_NAND_SetOnFatalErrorCallback()* on page 235.

6.3.2.6.2.5 FS_NAND_UNI_SetNumWorkBlocks()

Description

Sets number of work blocks the driver uses for write operations.

Prototype

```
void FS_NAND_UNI_SetNumWorkBlocks(U8  Unit,
                                  U32 NumWorkBlocks);
```

Parameter	Meaning
Unit	Unit number (0 based).
NumWorkBlocks	Number of work blocks.

Table 6.68: FS_NAND_UNI_SetNumWorkBlocks() parameter list

For additional information, refer to *FS_NAND_SetNumWorkBlocks()* on page 236.

6.3.2.6.2.6 FS_NAND_UNI_SetECCHook()

Description

Configures the ECC algorithm to be used by the NAND driver.

Prototype

```
void FS_NAND_UNI_SetECCHook(U8 Unit, const FS_NAND_ECC_HOOK * pECCHook);
```

Parameter	Meaning
Unit	Unit number (0 based)
pECCHook	IN: Pointer to a FS_NAND_ECC_HOOK structure containing the API functions and the attributes of ECC algorithm. OUT: ---

Table 6.69: FS_NAND_UNI_SetECCHook() parameter list

Additional information

This function allows an application to configure the functions which the driver should call to compute and correct the bit errors. Following ECC algorithms are provided by the driver:

Values for parameter pECCHook	
FS_NAND_ECC_HW_NULL	This is a pseudo algorithm. It requests the driver to use the internal HW ECC of NAND flash.
FS_NAND_ECC_HW_4BIT	This is a pseudo algorithm an it should be used with physical layers which provide ECC error correction. For example when the physical layer uses a dedicated NAND flash controller with HW ECC. When used the pseudo algorithm requests the physical layer to use 4-bit error correction
FS_NAND_ECC_SW_1BIT	This is an algorithm which is able to correct 1 bit errors and detect 2 bit errors. More specifically it can correct 1 bit errors on each 256 byte stripe of the 512 data area and a 1 bit errors on the 4 bytes of spare area.

By default, the driver uses the FS_NAND_ECC_1BIT ECC algorithm. The application must provide an ECC algorithm for the cases where the NAND flash has no ECC engine and a better error correction is required than provided by the default algorithm. The algorithm can be implemented in software or it can use a dedicated ECC hardware if available on the target system. For details about the computation routines, refer to *FS_NAND_ECC_HOOK* on page 300.

6.3.2.6.2.7 FS_NAND_ECC_HOOK

Description

The structure contains pointers to API functions and attributes related to ECC algorithm.

Declaration

```
typedef struct {
  void     (*pfCompute)         (const U32 * pData, U8 * pSpare);
  int      (*pfApply)           (      U32 * pData, U8 * pSpare);
  unsigned NumBitsCorrectable;
} FS_NAND_ECC_HOOK;
```

Parameter	Meaning
pfCompute	Pointer to a function which computes the ECC over a block of 516 bytes.
pfApply	Pointer to a function which checks and corrects the ECC protected data.
NumBitsCorrectable	Number of bits the ECC algorithm is able to correct in a 516 byte block.

Table 6.70: FS_NAND_ECC_HOOK - list of structure elements

Additional information

The ECC is always computed over a 512 byte data area and 4 byte spare area. `pData` points to a data area of 512 bytes and `pSpare` points to whole 16 byte spare area. The spare area is organized as follows:

Byte offset	Meaning
0-3	Reserved
4-7	4 bytes of data which must be protected by ECC. The driver stores here management information. This area is used only by the driver.
7-15	8 bytes of ECC. The `(*pfCompute)()` function stores here the calculated ECC of the data and spare areas. The `(*pfApply)()` loads The ECC is loaded from this location by the `(*pfApply)()` function which performs the error checking and correction. This area is used only by the ECC routines.

The `(*pfCompute)()` function calculates the ECC. It is called when data is written to NAND flash. The ECC covers the whole data area and 4 bytes of spare area from byte offset 4. The function stores the resulted 8 byte ECC to `pSpare` at byte offset 8.

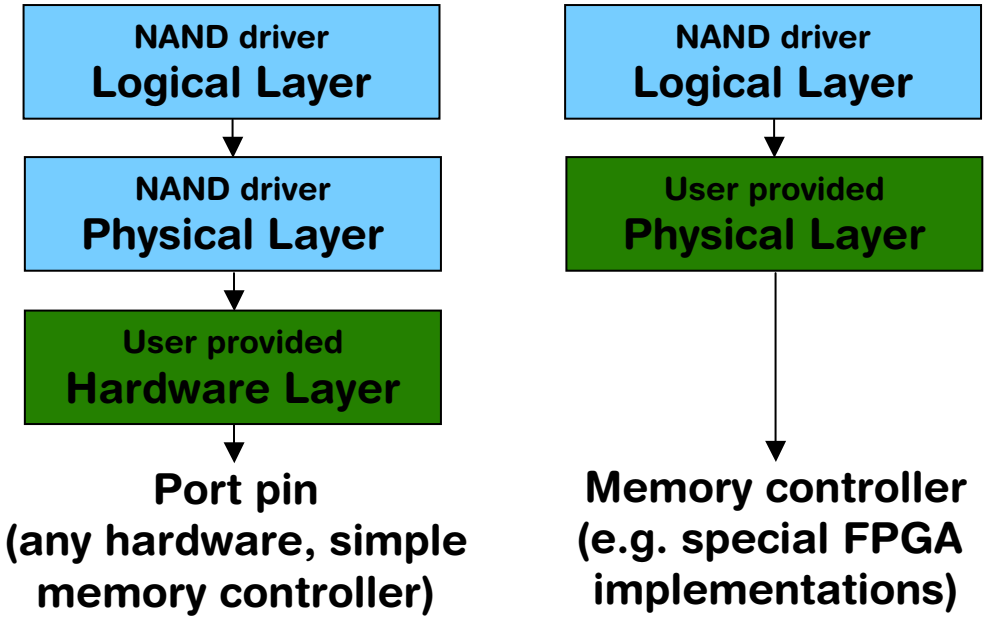
The error checking and correction is performed by the `(*pfApply)()` function. This function is called when data is read from NAND flash. First, the routine calculates the ECC in the same way `(*pfCompute)()` function does. Then it compares the calculated ECC against the 8 byte ECC loaded from the byte offset 8 of `pSpare`. If the ECC values are equal there are no bit errors and the function returns. Else the function determines whether the errors can be corrected and if so, it corrects them in the `pData` and `pSpare` buffers. Following values can be returned:

Value	Meaning
0	No error detected.
1	Bit errors corrected. Data is OK.
2	Error in ECC detected. Data is OK.
3	Uncorrectable bit error. Data is corrupted.

6.3.2.7 Physical layer

There is normally no need to change the physical layer of the NAND driver, only the hardware layer has to be adapted.

In some special cases, when the low-level hardware routines provided by the driver are not compatible with the target hardware (e.g. special FPGA implementations of a memory controller), the physical layer has to be adapted.



6.3.2.7.1 Available physical layers

The following physical layers are available. Refer to *Configuring the driver* on page 230 for detailed information about how to add the required physical layer to your application.

Available physical layers	
FS_NAND_PHY_2048x8	Supports NAND flash devices with 2048 bytes per page and 8-bit width
FS_NAND_PHY_2048x16	Supports NAND flash devices with 2048 bytes per page and 16-bit width
FS_NAND_PHY_4096x8	Supports NAND flash devices with 4096 bytes per page and 8-bit width
FS_NAND_PHY_x	Supports the following NAND flash devices: • 512 bytes per page and 8-bit width • 2048 bytes per page and 8-bit width • 2048 bytes per page and 16-bit width • 4096 bytes per page and 8-bit width
FS_NAND_PHY_x8	Supports the following NAND flash devices: • 512 bytes per page and 8-bit width • 2048 bytes per page and 8-bit width • 4096 bytes per page and 8-bit width
FS_NAND_PHY_ONFI	Supports NAND flash devices compliant with the ONFI specification.
FS_NAND_PHY_SPI	Supports NAND flash devices with SPI serial interface.

Table 6.71: Available physical layers

For a description of the physical layer functions, refer to *Physical layer functions* on page 238.

6.3.2.8 Hardware layer

The driver uses the same hardware layer as the SLC1 driver. For additional information, refer to *Hardware functions - NAND flash* on page 253.

6.3.2.9 Additional driver functions

Routine	Explanation
FS_NAND_UNI_GetDiskInfo()	Delivers information about NAND flash.
FS_NAND_UNI_GetBlockInfo()	Delivers information about a physical block of NAND flash.

Table 6.72: FS_NAND_UNI_Driver - list of additional functions.

6.3.2.9.1 FS_NAND_UNI_GetDiskInfo()

Description

Returns information about the NAND flash.

Prototype

```
void FS_NAND_UNI_GetDiskInfo(U8 Unit, FS_NAND_DISK_INFO * pDiskInfo);
```

Parameter	Meaning
Unit	Unit number of driver.
pDiskInfo	IN: --- OUT: Information about NAND flash.

Table 6.73: FS_NAND_UNI_GetDiskInfo() parameter list

Example

For an example, refer to *FS_NAND_GetDiskInfo()* on page 280.

6.3.2.9.2 FS_NAND_UNI_GetBlockInfo()

Description

Returns information about a physical block of NAND flash.

Prototype

```
void FS_NAND_UNI_GetBlockInfo(U8                    Unit,
                              U32                   PhyBlockIndex,
                              FS_NAND_BLOCK_INFO * pBlockInfo);
```

Parameter	Meaning
Unit	Unit number of driver.
PhyBlockIndex	Index of the physical block.
pDiskInfo	IN: --- OUT: Information about the physical block.

Table 6.74: FS_NAND_UNI_GetBlockInfo() parameter list

Example

```
void _ShowBlockInfo(U32 Unit, U32 PhyBlockIndex) {
  FS_NAND_BLOCK_INFO BlockInfo;
  const char * sType;

  printf("Retrieving block information for nand:%d:, block index: 0x%.8x\n",
         Unit, PhyBlockIndex);
  FS_NAND_GetBlockInfo((U8)Unit, PhyBlockIndex, &BlockInfo);
  switch (BlockInfo.Type) {
  case NAND_BLOCK_TYPE_BAD:
    sType = "Bad block";
    break;
  case NAND_BLOCK_TYPE_EMPTY:
    sType = "Block not in use";
    break;
  case NAND_BLOCK_TYPE_WORK:
    sType = "Work block";
    break;
  case NAND_BLOCK_TYPE_DATA:
    sType = "Data block";
    break;
  case NAND_BLOCK_TYPE_UNKNOWN:
  default:
    sType = "Unknown";
    break;
  }
  printf("  sType                     = %s\n"
         "  EraseCnt                  = 0x%.8x\n"
         "  lbi                       = %d\n"
         "  NumSectorsBlank           = %d\n"
         "  NumSectorsECCCorrectable  = %d\n"
         "  NumSectorsErrorInECC      = %d\n"
         "  NumSectorsECCError        = %d\n"
         "  NumSectorsInvalid         = %d\n"
         "  NumSectorsValid           = %d\n", sType,
                                               BlockInfo.EraseCnt,
                                               BlockInfo.lbi,
                                               BlockInfo.NumSectorsBlank,
                                               BlockInfo.NumSectorsECCCorrectable,
                                               BlockInfo.NumSectorsErrorInECC,
                                               BlockInfo.NumSectorsECCError,
                                               BlockInfo.NumSectorsInvalid,
                                               BlockInfo.NumSectorsValid);
}
```


6.3.2.10 Test hardware

For more information, refer to *Test hardware* on page 284.

6.3.2.11 Performance and resource usage

6.3.2.11.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the NAND driver presented in the tables below have been measured on a system as follows: ARM7, IAR Embedded workbench V4.41A, Thumb mode, Size optimization.

Module	ROM [Kbytes]
emFile Universal NAND driver	7.9

In addition, one of the following physical layers is required:

Physical layer	Description	ROM [Kbytes]
FS_NAND_PHY_2048x8	Physical layer for large page NAND flash devices with an 8-bit interface.	1.0
FS_NAND_PHY_2048x16	Physical layer for large page NAND flash devices with an 16-bit interface.	1.0
FS_NAND_PHY_4096x8	Physical layer for NAND flash devices with an 8-bit interface and 4096 bytes per page.	0.9
FS_NAND_PHY_x8	Physical layer for large and small NAND devices with an 8-bit interface.	2.3
FS_NAND_PHY_x	Physical layer for large and small NAND devices with an 8-bit or 16-bit interface.	3.3
FS_NAND_PHY_ONFI	Physical layer for NAND flash devices which support ONFI.	1.5
FS_NAND_PHY_SPI	Physical layer for NAND flash devices with an SPI serial interface.	1.2

6.3.2.11.2 Static RAM usage

Static RAM usage is the amount of RAM required by the driver for variables inside of the driver. The number of bytes can be seen in a compiler list file

Static RAM usage of the NAND driver: 32 bytes

6.3.2.11.3 Runtime RAM usage

Runtime RAM usage is the amount of RAM allocated by the driver at runtime. The amount required depends on the runtime configuration and the connected device.

The approximately RAM usage for the NAND driver can be calculated as follows:

Every NAND flash device requires:


```
MemAllocated = 148 + 2 * NumBlocks
             + ((PagesPerBlock - 1) + 18) * NumWorkBlocks
             + 1.04 * PageSize
```

Parameter	Description
MemAllocated	Number of bytes allocated.
NumBlocks	Number of blocks in the NAND flash.
NumWorkBlocks	Number of physical sectors the driver reserves as temporary storage for the written data. Typically 3 physical sectors or the number specified in the call to the FS_NAND_UNI_SetNumWorkBlocks() configuration function
PageSize	Number of bytes in a page.

Table 6.75: Runtime RAM usage parameters for FS_NAND_UNI_Driver

Example: 2 GBit NAND flash with 2K pages

One block consists of 64 pages, each page holds 1 sector of 2048 bytes.

```
PagesPerBlock = 64
NumBlocks     = 2048
NumWorkBlocks = 4
PageSize      = 2048

MemAllocated = 148 + 2 * 2048 + (64 - 1 + 18) * 4 + 1.04 * 2048
             = 148 + 4096 + 324 + 2129
             = 6397 bytes
```

6.3.2.11.4 Performance

These performance measurements are in no way complete, but they give an approximation of the length of time required for common operations on various targets. The tests were performed as described in *Performance* on page 502.

All values are in Mbytes/sec.

Device	CPU speed	Medium	W	R
Atmel AT91SAM3U	96 MHz	NAND with 2048 bytes per page and a sector size of 2048 bytes with internal ECC enabled using the built-in NAND controller/external bus-interface.	1.6	7.5
Atmel AT91SAM7S	48MHz	NAND with 2048 bytes per page and a sector size of 2048 bytes, 1-bit ECC, controller/external bus-interface.	1.5	5.5

Table 6.76: Performance values for sample configurations

6.3.3 Additional Information

Low-level format

Before using the NAND flash as a storage device, a low-level format has to be performed. Refer to *FS_FormatLow()* on page 113 and *FS_FormatLLIfRequired()* on page 112 for detailed information about low-level format.

6.3.4 Additional physical layer functions

Routine	Explanation
FS_NAND_PHY_ReadDeviceId()	Reads the ID information from NAND flash.
FS_NAND_PHY_ReadONFIPara()	Reads the ONFI parameters from NAND flash.
FS_NAND_2048x8_EnableReadCache()	Activates the read page optimization.
FS_NAND_2048x8_DisableReadCache()	Deactivates the read page optimization.
FS_NAND_SPI_EnableReadCache()	Activates the read page optimization.
FS_NAND_SPI_DisableReadCache()	Deactivates the read page optimization.

Table 6.77: List of additional physical layer functions.

6.3.4.0.1 FS_NAND_PHY_ReadDeviceId()

Description

Executes the READ ID command to read information from NAND flash.

Prototype

```
void FS_NAND_PHY_ReadDeviceId(U8 Unit, U8 * pId, U32 NumBytes);
```

Parameter	Meaning
Unit	Unit number of HW layer.
pId	IN: --- OUT: Information about NAND flash.
NumBytes	Number of bytes to read.

Table 6.78: FS_NAND_PHY_ReadDeviceId() parameter list

Additional information

This function can be used to query the type of NAND flash connected to host. It can be called from the function FS_X_AddDevices() as it invokes only functions of the NAND HW layer. No instance of NAND driver is required.

Example

This example shows how an application can configure at runtime different NAND drivers based on the type of the NAND flash.


```
/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*    It is supposed to add all devices, using primarily FS_AddDevice().
*
*  Note
*    (1) Other API functions
*        Other API functions may NOT be called, since this function is called
*        during initialisation. The devices are not yet ready at this point.
*/
void FS_X_AddDevices(void) {
  U8  Id;

  FS_AssignMemory(_aMemBlock, sizeof(_aMemBlock));
  //
  // Read the first byte of the identification array
  // which stores the manufacturer type.
  //
  FS_NAND_PHY_ReadDeviceId(0, &Id, sizeof(Id));
  if (Id == 0xEC) {
    //
    // Found a Samsung NAND flash. Use the SLC1 NAND driver.
    // ECC is performed by the NAND driver
    //
    FS_AddDevice(&FS_NAND_Driver);
    FS_NAND_SetPhyType(0, &FS_NAND_PHY_2048x8);
  } else if (Id == 0x2C) {
    //
    // Found a Micron NAND flash. Use the Universal NAND driver.
    // The ECC is performed by the NAND flash.
    //
    FS_AddDevice(&FS_NAND_UNI_Driver);
    FS_NAND_UNI_SetPhyType(0, &FS_NAND_PHY_ONFI);
    FS_NAND_UNI_SetECCHook(0, &FS_NAND_ECC_HW_NULL);
  } else {
    //
    // NAND flash from another manufacturer, Use auto-identification.
    //
    FS_AddDevice(&FS_NAND_Driver);
    FS_NAND_SetPhyType(0, &FS_NAND_PHY_x8);
  }
}
```


6.3.4.0.2 FS_NAND_PHY_ReadONFIPara()

Description

Reads ONFI parameters from NAND flash.

Prototype

```
int FS_NAND_PHY_ReadONFIPara(U8 Unit, void * pPara);
```

Parameter	Meaning
Unit	Unit number of HW layer.
pPara	IN: --- OUT: Read ONFI parameters. It must be at least 256 bytes large.

Table 6.79: FS_NAND_PHY_ReadONFIPara() parameter list

Return value

==0 ONFI parameters read.
!=0 NAND flash does not support ONFI or an error occurred.

Additional information

This function can be used to read the ONFI information stored in a NAND flash. It can be called from the function FS_X_AddDevices() as it invokes only functions of the NAND hardware layer. No instance of NAND driver is required. The pPara parameter can be NULL in which case the function returns 0 when the NAND flash is ONFI compatible.

Example

This example shows how an application can configure at runtime different NAND drivers based on the type of the NAND flash.

```
/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*    It is supposed to add all devices, using primarily FS_AddDevice().
*
*  Note
*    (1) Other API functions
*        Other API functions may NOT be called, since this function is called
*        during initialisation. The devices are not yet ready at this point.
*/
void FS_X_AddDevices(void) {
  int r;

  FS_AssignMemory(_aMemBlock, sizeof(_aMemBlock));
  //
  // Check whether the NAND flash supports ONFI.
  //
  r = FS_NAND_PHY_ReadONFIPara(0, NULL);
  if (r) {
    //
    // Found a NAND flash which does not support ONFI.
    //
    FS_AddDevice(&FS_NAND_Driver);
    FS_NAND_SetPhyType(0, &FS_NAND_PHY_2048x8);
  } else {
    //
    // Found a NAND flash which supports ONFI.
    //
    FS_AddDevice(&FS_NAND_UNI_Driver);
    FS_NAND_UNI_SetPhyType(0, &FS_NAND_PHY_ONFI);
    FS_NAND_UNI_SetECCHook(0, &FS_NAND_ECC_HW_NULL);
  }
}
```


6.3.4.0.3 FS_NAND_2048x8_EnableReadCache()

Description

Activates the read page optimization.

Prototype

```
void FS_NAND_2048x8_EnableReadCache(U8 Unit);
```

Parameter	Meaning
Unit	Unit number of physical layer.

Table 6.80: FS_NAND_2048x8_EnableReadCache() parameter list

Additional information

A page read operation consists of 2 steps. On the first step the page data is read from memory array to internal page register of NAND flash device. On the second step the data is transferred from internal page register of NAND flash device to host CPU. With the optimization enabled the first step is skipped when possible.

The optimization is enabled by default and should be disabled if 2 or more instances of NAND driver are configured to access the same NAND flash device.

6.3.4.0.4 FS_NAND_2048x8_DisableReadCache()

Description

Deactivates the read page optimization.

Prototype

```
void FS_NAND_2048x8_DisableReadCache(U8 Unit);
```

Parameter	Meaning
Unit	Unit number of physical layer.

Table 6.81: FS_NAND_2048x8_DisableReadCache() parameter list

Additional information

For additional information, refer to *FS_NAND_2048x8_EnableReadCache()* on page 311.

6.3.4.0.5 FS_NAND_SPI_EnableReadCache()

Description

Activates the read page optimization of the FS_NAND_PHY_SPI physical layer.

Prototype

```
void FS_NAND_SPI_EnableReadCache(U8 Unit);
```

Parameter	Meaning
Unit	Unit number of physical layer.

Table 6.82: FS_NAND_SPI_EnableReadCache() parameter list

Additional information

For additional information, refer to *FS_NAND_2048x8_EnableReadCache()* on page 311.

6.3.4.0.6 FS_NAND_SPI_DisableReadCache()

Description

Deactivates the read page optimization of the FS_NAND_PHY_SPI physical layer.

Prototype

```
void FS_NAND_SPI_DisableReadCache(U8 Unit);
```

Parameter	Meaning
Unit	Unit number of physical layer.

Table 6.83: FS_NAND_SPI_DisableReadCache() parameter list

Additional information

For additional information, refer to *FS_NAND_2048x8_EnableReadCache()* on page 311.

6.4 NOR flash driver

emFile supports the use of NOR flashes. Two optional drivers for NOR flashes are available:

- Sector map driver - optimized for read/write speed.
- Block map driver - optimized for reduced RAM usage.

They can work with almost any NOR flash and are extremely efficient. The difference between the drivers consists in the way they are managing the mapping of file system sectors to NOR flash storage.

The Sector map driver was designed with the goal to access the data fast at a time when the NOR flashes had a relatively small capacity. To achieve this, the driver maintains a mapping table at sector granularity. This approach has been proven to be efficient, but modern NOR flashes with capacities over 1MB the RAM usage of the driver increases. This is the reason why the Block map driver was developed.

The design goal of the Block map driver was to use as few RAM as possible. The driver maps blocks of file system sectors to NOR storage. In this way the RAM requirements of the driver are kept to a minimum.

6.4.1 Sector map driver - FS_NOR_Driver

This section describes the NOR driver which is optimized for fast write speed. It works by mapping single logical sectors to locations on the NOR flash memory.

6.4.1.1 Supported hardware

The NOR flash drivers can be used with almost any NOR flash. This includes NOR flashes with 1x8-bit and 1x16-bit parallel interfaces, as well as 2x16-bit interfaces in parallel, as well as serial NOR flashes.

Requirements

To be more precise, any NOR flash which fulfills the following requirements:

- Minimum of 2 physical sectors. At least 2 sectors need to be identical in size.
- Physical sectors need to be at least 2048 bytes each.
- Physical sectors do not need to be uniform
  (for example, 8 * 8 Kbytes + 3 * 64 Kbytes is permitted).
- Flash needs to be re-writable without erase: The same location can be written to multiple times without erase, as long as only 1-bits are converted to 0-bits.
- Erase clears all bits in a physical sector to 1.

Physical layer

The driver requires a physical layer for the flash device.

The following physical layers are available:

- FS_NOR_PHY_CFI_1x16 - CFI compliant parallel NOR flash with 1x16-bit interface
- FS_NOR_PHY_CFI_2x16 - CFI compliant parallel NOR flash with 2x16-bit interface
- FS_NOR_PHY_ST_M25 - Serial flash (ST M25Pxx family)
- FS_NOR_PHY_SFDP - Serial flash compliant with the JEDEC JESD216 standard.
- Physical layer template

Common flash interface (CFI)

The NOR flash drivers can be used with any CFI-compliant 16-bit chip. The Common Flash Memory Interface (CFI) is an open specification which may be implemented freely by flash memory vendors in their devices. It was developed jointly by Intel, AMD, Sharp, and Fujitsu.

The idea behind CFI was the interchangeability of current and future flash memory devices offered by different vendors. If you use only CFI compliant flash memory chips, you are able to use one driver for different flash products by reading identifying information out of the flash chip itself.

The identifying information for the device, such as memory size, byte/word configuration, block configuration, necessary voltages, and timing information, is stored directly on the chip.

6.4.1.1.1 Tested and compatible NOR flashes

In general, the drivers supports almost all serial and parallel NOR flashes which fulfill the listed requirements. This includes NOR flashes with 1x8-bit, 1x16-bit and 2x16-bit interfaces.

The table below shows the serial NOR flashes that have been tested or are compatible with a tested device:

Manufacturer	Device	Size
ST Microelectronics	M25P40 M25P80 M25P16 M25P32 M25P128	4 Mbit (512 Kbytes) 8 Mbit (1Mbytes) 16 Mbit (2Mbytes) 32 Mbit (4Mbytes) 128 Mbit (16Mbytes)
Micron	N25Q064 N25Q256	64 Mbit (8Mbytes) 256 Mbit (32Mbytes)
Numonyx	M25P64	64 Mbit (8Mbytes)
Winbond	W25Q64	64 Mbit (8Mbytes)
Macronix	MX25L256 MX66L51235F	256 Mbit (32Mbytes) 512 Mbit (64Mbytes)

Table 6.84: List of supported serial NOR flashes

The table below shows the parallel NOR flashes that have been tested or are compatible with a tested device:

Manufacturer	Device	Size [Bits]
Intel	Intel 28FxxxP30 Intel 28FxxxP33	64 Mbytes - 1 Gbytes 64 Mbytes - 512 Mbytes
ST-Microelectronics	M28W160 M28W320 M28W640 M29F080 M29W160 M29W320 M29W640 M58LW064	16 Mbytes (1 Mbytes x 16) 32 Mbytes (2 Mbytes x 16) 64 Mbytes (4 Mbytes x 16) 8 Mbytes (1 Mbytes x 8) 16 Mbytes (2 Mbytes x 8 or 1 Mbytes x 16) 32 Mbytes (4 Mbytes x 8 or 2 Mbytes x 16) 64 Mbytes (8 Mbytes x 8 or 4 Mbytes x 16) 64 Mbytes (8 Mbytes x 8, 4Mbytes x 16)
Micron	MT28F128 MT28F256 MT28F320 MT28F640	128 Mbytes 256 Mbytes 32 Mbytes 64 Mbytes

Table 6.85: List of supported parallel NOR flashes

Support for devices not available in this list

Most other NOR flash devices are compatible with one of the supported devices. Thus the driver can be used with these devices or may only need a little modification, which can be easily done. Get in touch with us, if you have questions about support for devices not in this list.

6.4.1.2 Theory of operation

Differentiating between “logical sectors” or “blocks” and “physical sectors” is very essential to understand this section. A logical sector/block is the base unit of any file system, its usual size is 512 bytes. A physical sector is an array of bytes on the flash chip that are erased together (typically between 2 Kbytes - 128 Kbytes). The flash chip driver is an abstraction layer between these two types of sectors.

Every time a logical sector is being updated, it is marked as invalid and the new content of this sector is written into another area of the flash. The physical address and the order of physical sectors can change with every write access. Hence, there cannot exist a direct relation between the sector number and its physical location.

The flash driver manages the logical sector numbers by writing it into special headers. It does not matter to the upper layer were the logical sector is stored or how much flash memory is used as a buffer. All logical sectors (starting with Sector #0) do always exist and are always available for user access.

6.4.1.2.1 Using the same NOR flash for code and data

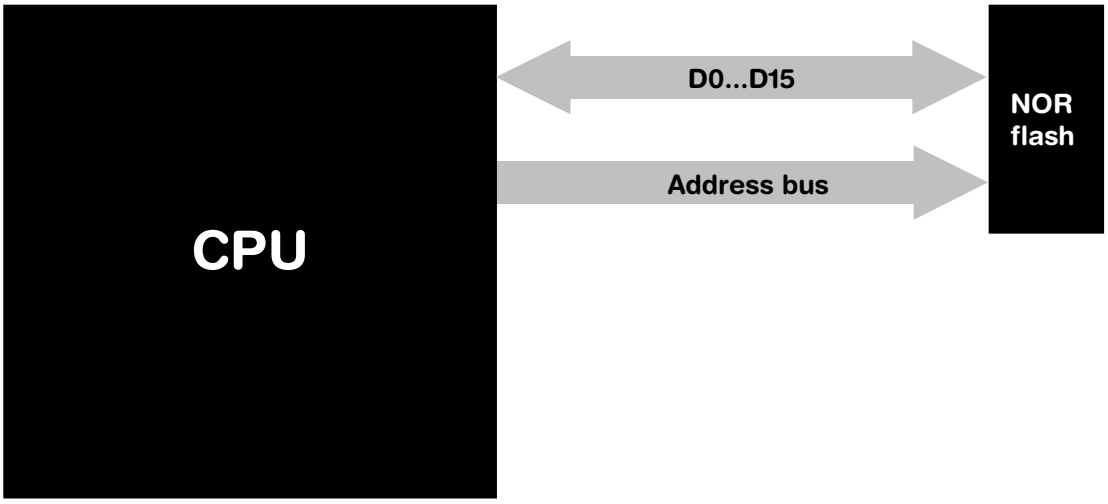
Most NOR flashes cannot be read out during a program, erase or identify operation. This means that code cannot be read from the NOR flash during a program or erase operation. If code which resides in the same NOR flash used for data storage is executed during program or erase, a program crash is almost certain.

There are multiple options to solve this:

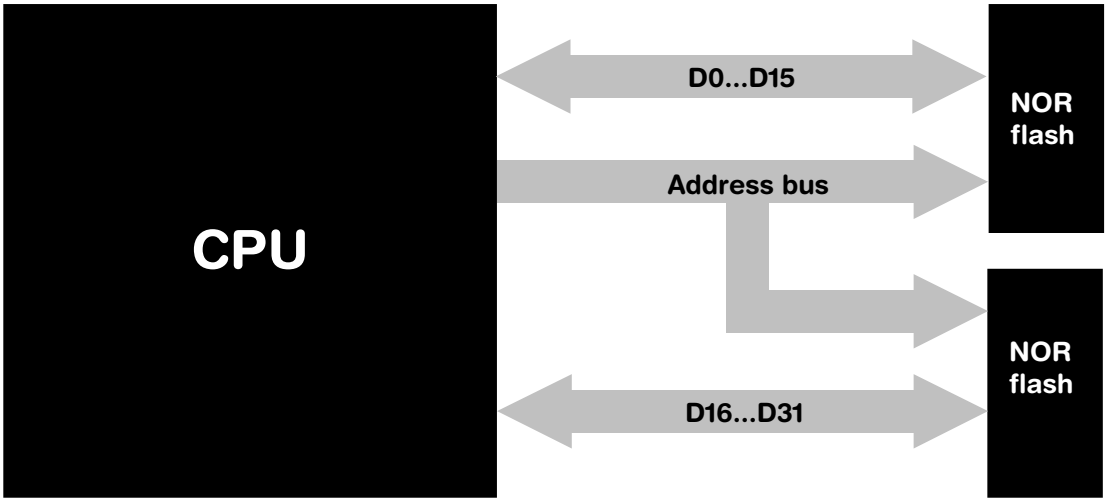
1. Use multiple NOR flashes. Use one flash for code and one for data.
2. Use a NOR flash with multiple banks, which allows reading Bank A while Bank B is being programmed.
3. Make sure the hardware routines which program, erase or identify the NOR flash are located in RAM and interrupts are disabled.

6.4.1.2.2 Physical interfaces

A device can consist of a single or two identical CFI compliant flash interfaces with a 16-bit interface. The most common is a CFI compliant NOR flash chip with a 16-bit interface.



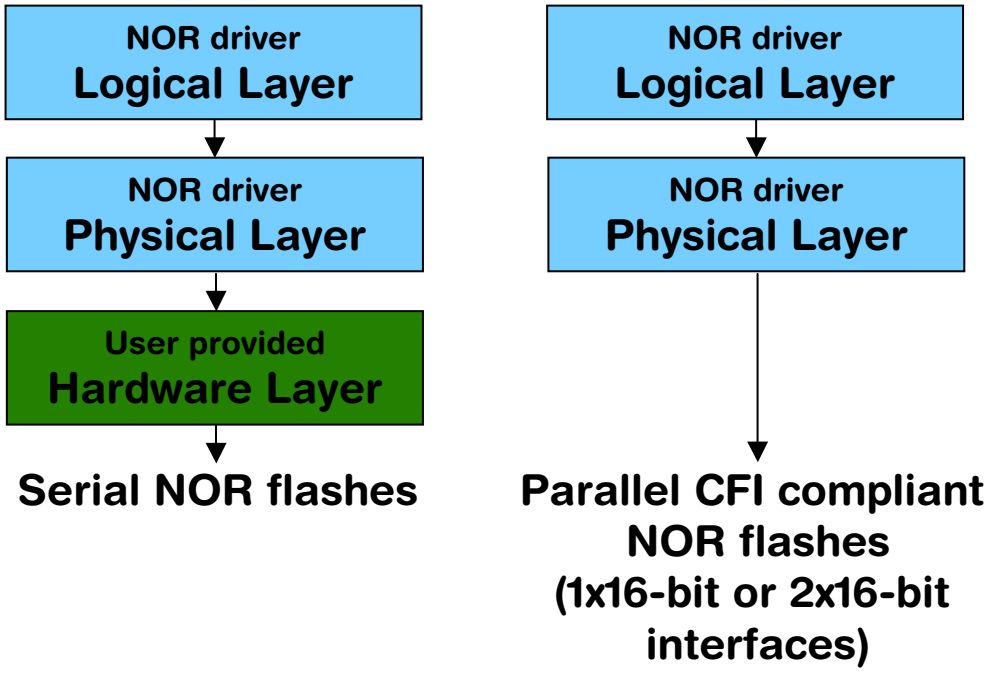
Beside this solution, emFile supports two CFI compliant NOR flash chips with a 16-bit interface which are connected to the same address bus.



The emFile NOR flash driver supports both options.

6.4.1.2.3 Software structure

The NOR flash driver is divided into different layers, which are shown in the illustration below.



It is possible to use the NOR flash drivers also with serial NOR flashes. Only the hardware layer needs to be ported. Normally no changes to the physical layer are required. If the physical layer needs to be adapted, a template is available.

6.4.1.3 Fail-safe operation

The emFile NOR driver is fail-safe. That means that the driver makes only atomic actions and takes the responsibility that the data managed by the file system is always valid. In case of power loss or power reset during a write operation it is always assured that only valid data is stored in the flash. If the power loss interrupts the write operation, the old data will be kept and not corrupted.

6.4.1.4 Wear leveling

Wear leveling is supported by the driver. Wear leveling makes sure that the number of erase cycles remains approximately equal for each sector. Maximum erase count difference is set to 5. This value specifies a maximum difference of erase counts for different physical sectors before the wear leveling uses the sector with the lowest erase count.

6.4.1.5 Garbage collection

The driver performs the garbage collection automatically. When data is written to storage medium and there are no more free logical sectors left new free logical sectors are created. A physical sector is erased and the valid logical sectors of an other physical sector are copied to it. This operation can take a potentially long time affecting the write performance. For applications which require maximum write throughput the garbage collection can be triggered explicitly. Typically, the operation can be performed when the file system is idle.

Two API functions are provided: `FS_STORAGE_Clean()` and `FS_STORAGE_CleanOne()`. They can be called directly from the task which is performing the write or from a background task. The `FS_STORAGE_Clean()` function blocks until all the invalid logical sectors are converted to free logical sectors. A write operation following the call of this function runs at maximum speed. The other function, `FS_STORAGE_CleanOne()`,

converts the invalid sectors of a single physical sector. Depending on the number of invalid logical sectors, several calls to this function are required to clean up the whole storage medium.

6.4.1.6 Configuring the driver

6.4.1.6.1 Adding the driver to emFile

To add the driver, use `FS_AddDevice()` with the driver label `FS_NOR_Driver`. This function has to be called from `FS_X_AddDevices()`. Refer to *FS_X_AddDevices()* on page 472 for more information.

Example

```
FS_AddDevice(&FS_NOR_Driver);
```

6.4.1.6.2 Configuration API

Routine	Explanation
FS_NOR_Configure()	Configures the NOR flash.
FS_NOR_SetPhyType()	Sets the physical type of NOR device.
FS_NOR_SetSectorSize()	Sets the size of a logical sector.
FS_NOR_SPI_Configure()	Configures manually a SPI NOR flash.
FS_NOR_CFI_SetAddrGap()	Defines a discontinuity in the address space of the CFI NOR flash.

Table 6.86: FS_NOR_Driver - list of configuration functions

6.4.1.6.2.1 FS_NOR_Configure()

Description

Configures the NOR flash drive. Needs to be called for CFI flashes. Typically, this function has to be called from `FS_X_AddDevices()` after adding the device driver to file system. Refer to *FS_X_AddDevices()* on page 472 for more information.

Prototype

```
void FS_NOR_Configure(U8  Unit,
                      U32 BaseAddr,
                      U32 StartAddr,
                      U32 NumBytes);
```

Parameter	Description
`Unit`	Unit number (0…N).
`BaseAddr`	Base address of the NOR flash chip. This is the address of the first byte of the NOR flash.
`StartAddr`	Start address of the NOR flash disk. This is the address of the first byte of the NOR flash to be used as flash disk. It needs to be >= `BaseAddr`.
`NumBytes`	Specifies the size of the NOR flash device in bytes. The size of the flash disk will be: `min(NumBytes, DeviceSize - (StartAddr - BaseAddr)` where `DeviceSize` is the size of the NOR flash found.

Table 6.87: FS_NOR_Configure() parameter list

Additional information

If your consists of two identical CFI compliant NOR flash chips with 16 bit interface `FS_NOR_Configure()` configures both flash chips. Refer to *FS_NOR_SetPhyType()* on page 324 for more information about the different physical type of your device.

Example

Configuration with a single NOR flash devices:

```
void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add driver
  //
  FS_AddDevice(&FS_NOR_Driver);
  //
  // Set physical type, single CFI compliant NOR flash chips with 16 bit interface
  //
  FS_NOR_SetPhyType(0, &FS_NOR_PHY_CFI_1x16);
  //
  // Configure a single NOR flash interface (2 Mbytes)
  //
  FS_NOR_Configure(0, 0x1000000, 0x1000000, 0x200000);
}
```

Configuration with 2 identical NOR flash devices:

```
void FS_X_AddDevices(void) {
  //
  // Add driver
  //
  FS_AddDevice(&FS_NOR_Driver);
  //
  // Set physical type, 2 identical CFI compliant NOR flash chips
  // with 16 bit interface
  //
  FS_NOR_SetPhyType(0, &FS_NOR_PHY_CFI_2x16);
  //
```



```
  //  Configure two NOR flash interfaces (2 Mbytes each)
  //
  FS_NOR_Configure(0, 0x1000000, 0x1000000, 0x400000);
}
```

Configuration with 2 different NOR flash devices:

```
void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add and configure the first NOR driver. Volume name "nor:0:"
  // Set physical type, single CFI compliant NOR flash chips with 16 bit interface.
  //
  FS_AddDevice(&FS_NOR_Driver);
  FS_NOR_SetPhyType(0, &FS_NOR_PHY_CFI_1x16);
  FS_NOR_Configure(0, 0x1000000, 0x1000000, 0x200000);
  //
  // Add and configure the second NOR driver. Volume name "nor:1:"
  // Set physical type, single CFI compliant NOR flash chips with 16 bit interface.
  //
  FS_AddDevice(&FS_NOR_Driver);
  FS_NOR_SetPhyType(1, &FS_NOR_PHY_CFI_1x16);
  FS_NOR_Configure(1, 0x4000000, 0x4000000, 0x200000);
}

void main(void) {
  FS_Init();
  //
  // Format first volume.
  //
  FS_FormatLLIfRequired("nor:0:");
  if (FS_IsHLFormatted("nor:0:") == 0) {
    FS_Format("nor:0:", NULL);
  }
  //
  // Format second volume.
  //
  FS_FormatLLIfRequired("nor:1:");
  if (FS_IsHLFormatted("nor:1:") == 0) {
    FS_Format("nor:1:", NULL);
  }
}
```


6.4.1.6.2.2 FS_NOR_SetPhyType()

Description

Sets the physical type of the device. The NOR flash driver comes with different physical interfaces. The most common is a CFI compliant NOR flash chip with a 16 bit interface. A device can consist of a single or two identical CFI compliant flash interfaces with a 16 bit interface. Set `pPhyType` to `FS_NOR_PHY_CFI_1x16` if you use a single NOR flash chip. If your device consists of two identical NOR flash chips, set `pPhyType` to `FS_NOR_PHY_CFI_2x16`.

This function has to be called from within `FS_X_AddDevices()` after adding the device driver to file system. Refer to *FS_X_AddDevices()* on page 472 for more information.

Prototype

```
void FS_NOR_SetPhyType(U8 Unit, const FS_NOR_PHY_TYPE * pPhyType);
```

Parameter	Meaning
`Unit`	Unit number (0…N).
`pPhyType`	Pointer to physical type.

Table 6.88: FS_NOR_SetPhyType() parameter list

Permitted values for parameter `pPhyType`	
`FS_NOR_PHY_CFI_1x16`	One CFI compliant NOR flash chip with 16 bit interface.
`FS_NOR_PHY_CFI_2x16`	Two CFI compliant NOR flash chip with 16 bit interfaces.
`FS_NOR_PHY_ST_M25`	Serial NOR flashes.

Additional information

If you want to access special flash devices (for example, the internal NOR flash of a microcontroller), you can define your own physical type. Use the supplied template `NOR_Phy_Template.c` for the implementation. The template is located in the `\Sample\Driver\NOR\` directory.

Note: Most NOR flashes cannot be read out during a program, erase or identify operation. This means that code cannot be read from the NOR flash during a program or erase operation. If code which resides in the same NOR flash used for data storage is executed during program or erase, a program crash is almost certain. To avoid this, you have to make sure that routines which program, erase or identify are located in RAM and interrupts are disabled. The responsibility therefor is on user side.

Example

Refer to *FS_NOR_Configure()* on page 322 for an example of usage.

6.4.1.6.2.3 FS_NOR_SetSectorSize()

Description

Configures the size of a logical sector on the NOR flash drive. Typically, this function has to be called from `FS_X_AddDevices()` after adding the device driver to file system. Refer to *FS_X_AddDevices()* on page 472 for more information.

Prototype

```
void FS_NOR_SetSectorSize(U8  Unit,
                          U16 SectorSize);
```

Parameter	Description
`Unit`	Unit number.
`SectorSize`	Number of bytes in a logical sector.

Table 6.89: FS_NOR_SetSectorSize() parameter list

Additional information

The logical sector size must be equal or less than the logical sector size of the file system. This is typically 512 bytes or the value configured in the call to API function `FS_SetMaxSectorSize()`.

6.4.1.6.2.4 FS_NOR_SPI_Configure()

Description

Specifies the parameters of a NOR flash connected over SPI.

Prototype

```
void FS_NOR_SPI_Configure(U8 Unit, U32 SectorSize, U16 NumSectors);
```

Parameter	Description
`Unit`	Unit number (0 based).
`SectorSize`	Number of bytes in a physical sector.
`NumSectors`	Number of physical sectors in the NOR flash.

Table 6.90: FS_NOR_SPI_Configure() parameter list

Additional information

Calling of this function is optional. The physical layer tries to auto-detect the capacity of device. It uses the value of the 3rd byte returned in response to a Read Identification (0x9F) command. The following mapping table is used:

3rd byte in response	Device size [Mbits]
0x11	1
0x12	2
0x13	4
0x14	8
0x15	16
0x16	32
0x17	64
0x18	128

Table 6.91: NOR SPI device IDs

It is required to call this function, only if the device does not identify itself with one of the above device IDs. `SectorSize` must be set to the size of the storage area erased by the Block Erase (0xD8) command. `NumSectors` is the device capacity in bytes divided by `SectorSize`.

Example

```
/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*    It is supposed to add all devices, using primarily FS_AddDevice().
*
*  Note
*    (1) Other API functions
*        Other API functions may NOT be called, since this function is called
*        during initialisation. The devices are not yet ready at this point.
*/
void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  //  Add driver.
  //
  FS_AddDevice(&FS_NOR_Driver);
  //
  //  Confgure manually a 16Mbit SPI NOR flash.
  //
  FS_NOR_SetPhyType(0, &FS_NOR_PHY_ST_M25);
  FS_NOR_SPI_Configure(0, 0x10000, 32);
  FS_NOR_Configure(0, 0, 0, 0x10000 * 32);
}
```

6.4.1.6.2.5 FS_NOR_CFI_SetAddrGap()

Description

Defines a discontinuity in the address space of NOR flash.

Prototype

```
void FS_NOR_CFI_SetAddrGap(U8 Unit, U32 StartAddr, U32 NumBytes);
```

Parameter	Description
Unit	Unit number (0 based).
StartAddr	Address of the first byte in the gap.
NumBytes	Number of bytes in the gap.

Table 6.92: FS_NOR_CFI_SetAddrGap() parameter list

Additional information

Any access to an address equal to or greater than `StartAddr` is offset by `NumBytes`. `StartAddr` and `NumBytes` should be aligned to physical sector boundaries.

Example

```
#define ALLOC_SIZE            0x400000
#define FLASH_BASE_ADDR       0x80000000
#define FLASH_START_ADDR      0x80000000
#define FLASH_SIZE            0x00400000
#define FLASH_GAP_START_ADDR  0x80200000
#define FLASH_GAP_SIZE        0x00200000

/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*    It is supposed to add all devices, using primarily FS_AddDevice().
*
*  Note
*    (1) Other API functions
*        Other API functions may NOT be called, since this function is called
*        during initialisation. The devices are not yet ready at this point.
*/
void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add and configure the driver for a 4MB NOR flash.
  //
  FS_AddDevice(&FS_NOR_Driver);
  FS_NOR_SetPhyType(0, &FS_NOR_PHY_CFI_1x16);
  FS_NOR_Configure(0, FLASH_BASE_ADDR, FLASH_START_ADDR, FLASH_SIZE);
  //
  // Configure a 2MB gap in the address space of NOR flash.
  //
  FS_NOR_CFI_SetAddrGap(0, FLASH_GAP_START_ADDR, FLASH_GAP_SIZE);
}
```

6.4.1.6.3 Sample configurations

In the following some sample configurations how to create multiple volumes, logical volumes etc., using the NOR driver are shown. All configuration steps have to be performed inside the `FS_X_AddDevices()` function. For more information about the `FS_X_AddDevices()` function, please refer to *FS_X_AddDevices()* on page 472.

Creating multiple volumes on a single NOR flash chip

The following example illustrates how to create multiple volumes on a single NOR flash chip. In this sample we create 2 volumes on one NOR flash.

```
//
// Config:  1 NOR flash, where NOR flash size -> 2 MB
//          2 volumes, , where volume 0 size -> 1MB, volume 1 -> 0.5MB
//
#define FLASH_BASE_ADDR             0x80000000

#define FLASH_VOLUME_0_START_ADDR   0x80000000
#define FLASH_VOLUME_0_SIZE         0x00100000  // 1 MByte

#define FLASH_VOLUME_1_START_ADDR   0x80100000
#define FLASH_VOLUME_1_SIZE         0x00080000  // 0.5 MByte

//
// Volume 0
//
FS_AddDevice(&FS_NOR_Driver);
FS_NOR_SetPhyType(0, &FS_NOR_PHY_CFI_1x16);
FS_NOR_Configure(0, FLASH_BASE_ADDR, FLASH_VOLUME_0_START_ADDR,
FLASH_VOLUME_0_SIZE);
//
// Volume 1
//
FS_AddDevice(&FS_NOR_Driver);
FS_NOR_SetPhyType(1, &FS_NOR_PHY_CFI_1x16);
FS_NOR_Configure(1, FLASH_BASE_ADDR, FLASH_VOLUME_1_START_ADDR,
```



```
    FLASH_VOLUME_1_SIZE);
```


Creating multiple volumes with multiple NOR flash chips

The following example illustrates how to create multiple volumes on multiple NOR flash chips. In this sample we create 2, each on one NOR flash.

```
//
// Config:  2 NOR flash chips, where NOR flash 0 size -> 2 MB, NOR flash 1 -> 16MB
//          2 volumes, volume 0 size -> complete NOR 0, volume 1 -> complete NOR 1
//
#define FLASH0_BASE_ADDR            0x80000000
#define FLASH_VOLUME_0_START_ADDR   FLASH0_BASE_ADDR
#define FLASH_VOLUME_0_SIZE         0xFFFFFFFF  // Use the complete flash

#define FLASH1_BASE_ADDR            0x40000000
#define FLASH_VOLUME_1_START_ADDR   FLASH1_BASE_ADDR
#define FLASH_VOLUME_1_SIZE         0xFFFFFFFF  // Use the complete flash

//
// Volume 0
//
FS_AddDevice(&FS_NOR_Driver);
FS_NOR_SetPhyType(0, &FS_NOR_PHY_CFI_1x16);
FS_NOR_Configure(  0,
                   FLASH0_BASE_ADDR,
                   FLASH_VOLUME_0_START_ADDR,
                   FLASH_VOLUME_0_SIZE
                );
//
// Volume 1
//
FS_AddDevice(&FS_NOR_Driver);
FS_NOR_SetPhyType(1, &FS_NOR_PHY_CFI_1x16);
FS_NOR_Configure(  1,
                   FLASH1_BASE_ADDR,
                   FLASH_VOLUME_1_START_ADDR,
                   FLASH_VOLUME_1_SIZE
                );
```


Creating volumes which spread over multiple NOR flash chips

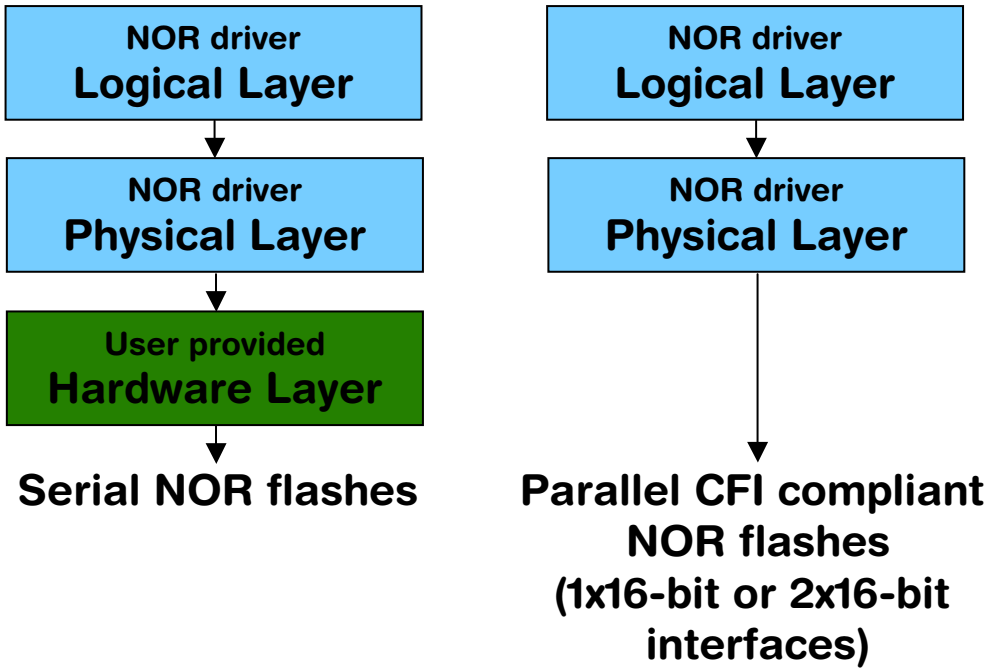
The following example illustrates how to create a volume which spreads over multiple NOR flash chips. This is achieved by using the logical volume functions. In this sample a logical volume which spreads over 2 NOR flash chips is created.

```
//
// Config:  2 NOR flash chips, where NOR flash 0 size -> 2 MB, NOR flash 1 -> 16MB
//          1 volume, where volume is NOR flash 0 + NOR flash 1
//
#define FLASH0_BASE_ADDR            0x80000000
#define FLASH_VOLUME_0_START_ADDR   FLASH0_BASE_ADDR
#define FLASH_VOLUME_0_SIZE         0xFFFFFFFF  // Use the complete flash

#define FLASH1_BASE_ADDR            0x40000000
#define FLASH_VOLUME_1_START_ADDR   FLASH1_BASE_ADDR
#define FLASH_VOLUME_1_SIZE         0xFFFFFFFF  // Use the complete flash

//
// Create physical device 0, this device will not be visible as a volume
//
FS_AddPhysDevice(&FS_NOR_Driver);
FS_NOR_SetPhyType(0, &FS_NOR_PHY_CFI_1x16);
FS_NOR_Configure(  0,
                   FLASH0_BASE_ADDR,
                   FLASH_VOLUME_0_START_ADDR,
                   FLASH_VOLUME_0_SIZE
                );
//
// In order to know whether the volume is low-level-formatted, we do the check here.
// When the device is added to the logical volume,
// a single check for low-level-format can not be performed.
//
if (FS_NOR_IsLLFormatted(0) == 0) {
  FS_NOR_FormatLow(0);
}
//
// Create physical device 1, this device will not be visible as a volume
//
FS_AddPhysDevice(&FS_NOR_Driver);
FS_NOR_SetPhyType(1, &FS_NOR_PHY_CFI_1x16);
FS_NOR_Configure(  1,
                   FLASH1_BASE_ADDR,
                   FLASH_VOLUME_1_START_ADDR,
                   FLASH_VOLUME_1_SIZE
                );
//
// In order to know whether the volume is low-level-formatted, we do the check here.
// When the device is added to the logical volume,
// a single check for low-level-format can not be performed.
//
if (FS_NOR_IsLLFormatted(1) == 0) {
  FS_NOR_FormatLow(1);
}
//
// Now create a logical volume, containing the physical devices
//
FS_LOGVOL_Create("LogVol");
FS_LOGVOL_AddDevice("LogVol", &FS_NOR_Driver, 0, 0, 0);
FS_LOGVOL_AddDevice("LogVol", &FS_NOR_Driver, 1, 0, 0);
```


6.4.1.7 Physical layer

There is normally no need to change the physical layer of the NOR driver, only the hardware layer has to be adapted if a non CFI compliant NOR flash chip is used in your hardware.

In some special cases, when the low-level hardware routines provided by the driver are not compatible with the target hardware the physical layer has to be adapted.



6.4.1.7.1 Available physical layers

The following physical layers are available. Refer to *Configuring the driver* on page 321 for detailed information about how to add the required physical layer to your application.

Available physical layers	
FS_NOR_PHY_CFI_1x16	One CFI compliant NOR flash chip with 16 bit interface.
FS_NOR_PHY_CFI_2x16	Two CFI compliant NOR flash chip with 16 bit interfaces.
FS_NOR_PHY_ST_M25	Serial NOR flashes.
FS_NOR_PHY_SFDP	Serial NOR flash compliant with the JEDEC JESD216 standard.

Table 6.93: Available physical layer

6.4.1.7.2 Physical layer functions

If there is a reason to change the physical layer anyhow, the functions which have to be changed are organized in a function table. The function table is implemented in a structure of type `FS_NOR_PHY_TYPE`.

```
struct FS_NOR_PHY_TYPE {
  int  (*pfWriteOff)     (U8 Unit, U32 Off, const void * pSrc, U32 Len);
  int  (*pfReadOff)      (U8 Unit, void * pDest, U32 Off, U32 Len);
  int  (*pfEraseSector)  (U8 Unit, unsigned int SectorIndex);
  void (*pfGetSectorInfo)(U8 Unit, unsigned int SectorIndex, U32 * pOff, U32 * pLen);
  int  (*pfGetNumSectors)(U8 Unit);
  void (*pfConfigure)    (U8 Unit, U32 BaseAddr, U32 StartAddr, U32 NumBytes);
  void (*pfOnSelectPhy)  (U8 Unit);
  void (*pfDeInit)       (U8 Unit);
} FS_NOR_PHY_TYPE;
```


If the physical layer should be modified, the following members of the structure FS_NOR_PHY_TYPE have to be adapted:

Routine	Explanation
(*pfWriteOff)()	Writes data into any section of the flash.
(*pfReadOff)()	Reads data from the given offset of the flash.
(*pfEraseSector)()	Erases one sector.
(*pfGetSectorInfo)()	Returns the offset and length of the given sector.
(*pfGetNumSectors)()	Returns the number of flash sectors.
(*pfConfigure)()	Configures a single instance of the driver.
(*pfOnSelectPhy)()	Retrieves information from flash.

Table 6.94: Physical layer hardware functions

6.4.1.7.2.1 (*pfWriteOff)()

Description

This routine writes data into any section of the flash. It does not check if this section has been previously erased; this is in the responsibility of the user program. Data written into multiple sectors at a time can be handled by this routine.

Prototype

```
int (*pfWriteOff) (U8           Unit,
                   U32          Off,
                   const void * pSrc,
                   unsigned     NumBytes);
```

Parameter	Meaning
Unit	Unit number (0...N).
Off	Zero-based byte offset.
pSrc	Pointer to a buffer of data which should be written.
NumBytes	Number of bytes which should be written.

Table 6.95: (*pfWriteOff)() parameter list

Return value

== 0: Data successfully transferred.
!= 0: An error has occurred.

6.4.1.7.2.2 (*pfReadOff)()

Description

Reads data from the given offset of the flash.

Prototype

```
int (*pfReadOff) (U8           Unit,
                  const void * pDest,
                  U32          Off,
                  unsigned     NumBytes);
```

Parameter	Meaning
Unit	Unit number (0...N).
pDest	Pointer to a buffer of data which should be read.
Off	Zero-based byte offset.
NumBytes	Number of bytes which should be written.

Table 6.96: (*pfReadOff)() parameter list

Return value

== 0: Data successfully transferred.
!= 0: An error has occurred.

6.4.1.7.2.3 (*pfEraseSector)()

Description

Erases one sector.

Prototype

```
int (*pfEraseSector) (U8  Unit,
                      U32 SectorIndex);
```

Parameter	Meaning
Unit	Unit number (0...N).
SectorIndex	zero-based index.

Table 6.97: (*pfEraseSector)() parameter list

Return value

== 0: OK. Sector is erased.
!= 0: An error has occurred; sector might not be erased.

6.4.1.7.2.4 (*pfGetSectorInfo)()

Description

Returns the offset and length of the given sector.

Prototype

```
void (*pfGetSectorInfo) (U8    Unit,
                         U32   SectorIndex,
                         U32 * pOff,
                         U32 * pLen);
```

Parameter	Meaning
Unit	Unit number (0...N).
SectorIndex	Zero-based sector index.
pOff	Buffer to store the offset of the specified sector.
pLen	Buffer to store the length of the specified sector.

Table 6.98: (*pfGetSectorInfo)() parameter list

6.4.1.7.2.5 (*pfGetNumSectors)()

Description

Returns the number of flash sectors.

Prototype

```
int (*pfGetNumSectors) (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0...N).

Table 6.99: (*pfGetNumSectors)() parameter list

Return value

Number of flash sectors.

6.4.1.7.2.6 (*pfConfigure)()

Description

Configures a single instance of the driver.

Prototype

```
void (*pfConfigure) (U8  Unit,
                     U32 BaseAddr,
                     U32 StartAddr,
                     U32 NumBytes);
```

Parameter	Meaning
Unit	Unit number (0...N).
BaseAddr	Base address of the flash.
StartAddr	Start address that should be used for the device.
NumBytes	Number of bytes which should be used for the device.

Table 6.100: (*pfConfigure)() parameter list

6.4.1.7.2.7 (*pfOnSelectPhy)()

Description

This function might be neccessary to retrieve the information from flash. It is called right after selection of the physical layer.

Prototype

```
void (*pfOnSelectPhy) (U8  Unit);
```

Parameter	Meaning
Unit	Unit number (0...N).

Table 6.101: (*pfOnSelectPhy)() parameter list

6.4.1.8 Hardware functions

Depending on the used NOR flash type and the corresponding physical layer, different hardware functions are required. CFI compliant NOR flashes do not need any hardware function, refer to *Hardware functions - Serial NOR flashes* on page 340 for detailed information about the hardware functions required by the physical layer for serial NOR flashes.

6.4.1.8.1 Hardware functions - CFI compliant chips

The NOR flash driver for CFI compliant chips does not need any hardware function.

6.4.1.8.2 Hardware functions - Serial NOR flashes

Routine	Explanation
Control line functions	
FS_NOR_SPI_HW_X_EnableCS()	Activates chip select signal (CS) of the serial NOR flash chip.
FS_NOR_SPI_HW_X_DisableCS()	Deactivates chip select signal (CS) of the DataFlash chip.
FS_NOR_SPI_HW_X_Init()	Initializes the SPI hardware.
Data transfer functions	
FS_NOR_SPI_HW_X_Read()	Receives a number of bytes from the serial NOR flash.
FS_NOR_SPI_HW_X_Write()	Sends a number of bytes to the serial NOR flash.
FS_NOR_SPI_HW_X_Read_x2()	Receives a number of bytes from the serial NOR flash using 2 data lines.
FS_NOR_SPI_HW_X_Write_x2()	Sends a number of bytes to the serial NOR flash using 2 data lines.
FS_NOR_SPI_HW_X_Read_x4()	Receives a number of bytes from the serial NOR flash using 4 data lines.
FS_NOR_SPI_HW_X_Write_x4()	Sends a number of bytes to the serial NOR flash using 4 data lines.

Table 6.102: Serial NOR flash device driver hardware functions

6.4.1.8.2.1 FS_NOR_SPI_HW_X_EnableCS()

Description

Activates chip select signal (CS) of the specified serial NOR flash.

Prototype

```
void FS_NOR_SPI_HW_X_EnableCS (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0…N).

Table 6.103: FS_NOR_SPI_HW_X_EnableCS() parameter list

Additional Information

The CS signal is used to address a specific serial NOR flash chip connected to the SPI. Enabling is equal to setting the CS line to low.

Example

```
/* Excerpt from NOR SPI hardware layer for Atmel AT91SAM9261. */

void FS_NOR_SPI_HW_X_EnableCS(U8 Unit) {
  _SPI_CLR_CS();
}
```


6.4.1.8.2.2 FS_NOR_SPI_HW_X_DisableCS()

Description

Deactivates chip select signal (CS) of the specified serial NOR flash chip.

Prototype

```
void FS_NOR_SPI_HW_X_DisableCS (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0…N).

Table 6.104: FS_NOR_SPI_HW_X_DisableCS() parameter list

Additional Information

The CS signal is used to address a specific serial NOR flash connected to the SPI. Disabling is equal to setting the CS line to high.

Example

```
/* Excerpt from NOR SPI hardware layer for Atmel AT91SAM9261. */

void FS_NOR_SPI_HW_X_DisableCS(U8 Unit) {
  _SPI_SET_CS();
}
```


6.4.1.8.2.3 FS_NOR_SPI_HW_X_Init()

Description

Initializes the SPI hardware.

Prototype

```
int FS_NOR_SPI_HW_X_Init (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0…N).

Table 6.105: FS_NOR_SPI_HW_X_Init() parameter list

Return value

== 0 Initialization was successful.
== 1 Initialization failed.

Example

```
/* Excerpt from NOR SPI hardware layer for Atmel AT91SAM9261. */

void FS_NOR_SPI_HW_X_Init(U8 Unit) {
  _SPI_SETUP_PINS();
}
```


6.4.1.8.2.4 FS_NOR_SPI_HW_X_Read()

Description

Receives a number of bytes from the serial NOR flash chip.

Prototype

```
void FS_NOR_SPI_HW_X_Read ( U8   Unit,
                            U8 * pData,
                            int  NumBytes );
```

Parameter	Meaning
Unit	Unit number (0…N).
pData	Pointer to a buffer for data to be receive.
NumBytes	Number of bytes to receive.

Table 6.106: FS_NOR_SPI_HW_X_Read() parameter list

Example

```
/* Excerpt from NOR SPI hardware layer for Atmel AT91SAM9261. */

void FS_NOR_SPI_HW_X_Read (U8 Unit, U8 * pData, int NumBytes) {
  do {
    SPI_TDR = 0xff;
    while ((SPI_SR & (1 << 9)) == 0);
    while ((SPI_SR & (1 << 0)) == 0);
    *pData++ = SPI_RDR;
  } while (--NumBytes);
}
```


6.4.1.8.2.5 FS_NOR_SPI_HW_X_Write()

Description

Sends a number of bytes to the card.

Prototype

```
void FS_NOR_SPI_HW_X_Write (       U8    Unit,
                            const U8   * pData,
                                  int   NumBytes );
```

Parameter	Meaning
Unit	Unit number (0…N).
pData	Pointer to a buffer for data to be receive.
NumBytes	Number of bytes to be written.

Table 6.107: FS_NOR_SPI_HW_X_Write() parameter list

Example

```
/* Excerpt from NOR SPI hardware layer for Atmel AT91SAM9261. */

void FS_NOR_SPI_HW_X_Write(U8 Unit, const U8 * pData, int NumBytes) {
  do {
    SPI_TDR = *pData++;
    while ((SPI_SR & (1 << 9)) == 0);
  } while (--NumBytes);
}
```


6.4.1.8.2.6 FS_NOR_SPI_HW_X_Read_x2()

Description

Receives a number of bytes from the serial NOR flash using 2 data lines.

Prototype

```
void FS_NOR_SPI_HW_X_Read_x2(U8    Unit,
                             U8  * pData,
                             int   NumBytes);
```

Parameter	Meaning
Unit	Unit number (0…N).
pData	Pointer to a buffer for data to be receive.
NumBytes	Number of bytes to receive.

Table 6.108: FS_NOR_SPI_HW_X_Read_x2() parameter list

Additional information

This function reads 2 data bits of data on each clock period. Typically, the data is transferred via the DataOut and DataIn lines of the NOR flash where the even numbered bits of a byte are sent on the DataIn line. The function is called only by the FS_NOR_PHY_SFDP physical layer.

6.4.1.8.2.7 FS_NOR_SPI_HW_X_Write_x2()

Description

Sends a number of bytes to serial NOR flash using 2 data lines.

Prototype

```
void FS_NOR_SPI_HW_X_Write_x2(      U8    Unit,
                              const U8  * pData,
                                    int   NumBytes );
```

Parameter	Meaning
Unit	Unit number (0…N).
pData	Pointer to a buffer for data to be receive.
NumBytes	Number of bytes to be written.

Table 6.109: FS_NOR_SPI_HW_X_Write_x2() parameter list

Additional information

This function writes 2 data bits of data on each clock period. Typically, the data is transferred via the DataOut and DataIn of the NOR flash where the even numbered bits of a byte are sent on the DataIn line. The function is called only by the FS_NOR_PHY_SFDP physical layer.

6.4.1.8.2.8 FS_NOR_SPI_HW_X_Read_x4()

Description

Receives a number of bytes from the serial NOR flash using 4 data lines.

Prototype

```
void FS_NOR_SPI_HW_X_Read_x4(U8    Unit,
                             U8  * pData,
                             int   NumBytes);
```

Parameter	Meaning
Unit	Unit number (0…N).
pData	Pointer to a buffer for data to be receive.
NumBytes	Number of bytes to receive.

Table 6.110: FS_NOR_SPI_HW_X_Read_x4() parameter list

Additional information

This function reads 4 data bits of data on each clock period. Typically, the data is transferred via the DataOut, DataIn, WP, and Hold lines of the NOR flash. A byte is read as follows: bits 0 and 4 are sent on the DataIn line, bits 1 and 5 on the DataOut line, bits 3 and 6 on the WP line, and bits 3 and 7 on the Hold line. The function is called only by the FS_NOR_PHY_SFDP physical layer.

6.4.1.8.2.9 FS_NOR_SPI_HW_X_Write_x4()

Description

Sends a number of bytes to serial NOR flash using 4 data lines.

Prototype

```
void FS_NOR_SPI_HW_X_Write_x4(      U8    Unit,
                              const U8  * pData,
                                    int   NumBytes );
```

Parameter	Meaning
Unit	Unit number (0…N).
pData	Pointer to a buffer for data to be receive.
NumBytes	Number of bytes to be written.

Table 6.111: FS_NOR_SPI_HW_X_Write_x4() parameter list

Additional information

This function writes 4 data bits of data on each clock period. Typically, the data is transferred via the DataOut, DataIn, WP, and Hold lines of the NOR flash. A byte is written as follows: bits 0 and 4 are sent on the DataIn line, bits 1 and 5 on the DataOut line, bits 3 and 6 on the WP line, and bits 3 and 7 on the Hold line. The function is called only by the FS_NOR_PHY_SFDP physical layer.

6.4.1.9 Additional information

Low-level format

Before using the NOR flash as storage device. A low-level format has to be performed. Refer to *FS_FormatLow()* on page 113 and *FS_FormatLLIfRequired()* on page 112 for detailed information about low-level formatting.

Further reading

For more technical details about CFI compliant flash memory, check the documents and specifications that are available free of charge:

- *Common Flash Interface (CFI) and Command Sets*
  Intel - Application Note 646 - April 2000
- *Common Flash Memory Interface Specification*
  AMD - Revision 2.0 - December 1, 2001

6.4.1.10 Additional driver functions

Routine	Explanation
FS_NOR_GetDiskInfo()	Returns information about NOR flash.
FS_NOR_GetSectorInfo()	Returns information about a physical sector.

Table 6.112: FS_NOR_Driver - list of additional functions

6.4.1.10.1 FS_NOR_GetDiskInfo()

Description

Returns information about the flash disk.

Prototype

```
void FS_NOR_GetDiskInfo(U8 Unit, FS_NOR_DISK_INFO * pDiskInfo);
```

Parameter	Description
Unit	Unit number.
pDiskInfo	Pointer to a structure of type FS_NOR_DISK_INFO.

Table 6.113: FS_NOR_GetDiskInfo() parameter list

Additional information

Refer to *FS_NOR_DISK_INFO* on page 353 for more information about the structure elements.

6.4.1.10.2 FS_NOR_GetSectorInfo()

Description

Returns info about a particular physical sector.

Prototype

```
void FS_NOR_GetSectorInfo(U8                   Unit,
                          U32                  PhysSectorIndex,
                          FS_NOR_SECTOR_INFO * pSectorInfo);
```

Parameter	Description
Unit	Unit number.
PhysSectorIndex	Index of physical sector.
pDiskInfo	Pointer to a structure of type FS_NOR_DISK_INFO.

Table 6.114: FS_NOR_GetSectorInfo() parameter list

Additional information

Refer to *FS_NOR_SECTOR_INFO* on page 354 for more information about the structure elements.

Example

```
/*********************************************************************
*
*       ShowDiskInfo
*
*/
void ShowDiskInfo(FS_NOR_DISK_INFO* pDiskInfo) {
  char acBuffer[80];

  FS_X_Log("Disk Info: \n");
  FS_NOR_GetDiskInfo(0, pDiskInfo);
  sprintf(acBuffer," Physical sectors: %d\n"
                   " Logical sectors : %d\n"
                   " Used sectors: %d\n", pDiskInfo->NumPhysSectors,
                                          pDiskInfo->NumLogSectors,
                                          pDiskInfo->NumUsedSectors);
  FS_X_Log(acBuffer);
}

/*********************************************************************
*
*       ShowSectorInfo
*/
void ShowSectorInfo(FS_NOR_SECTOR_INFO* pSecInfo, U32 PhysSectorIndex) {
  char acBuffer[400];

  FS_X_Log("Sector Info: \n");
  FS_NOR_GetSectorInfo(0, PhysSectorIndex, pSecInfo);
  sprintf(acBuffer," Physical sector No.     : %d\n"
                   " Offset                  : %d\n"
                   " Size                    : %d\n"
                   " Erase Count             : %d\n"
                   " Used logical sectors    : %d\n"
                   " Free logical sectors    : %d\n"
                   " Erasable logical sectors: %d\n", PhysSectorIndex,
                                                      pSecInfo->Off,
                                                      pSecInfo->Size,
                                                      pSecInfo->EraseCnt,
                                                      pSecInfo->NumUsedSectors,
                                                      pSecInfo->NumFreeSectors,
                                                     pSecInfo->NumEraseableSectors);
  FS_X_Log(acBuffer);
}

/*********************************************************************
*
*       MainTask
*/
void MainTask(void) {
  U32                i, j;
  char               ac[0x400];
```



```
  FS_NOR_DISK_INFO   DiskInfo;
  FS_NOR_SECTOR_INFO SecInfo;

  FS_FILE * pFile;
  FS_Init();
  FS_FormatLLIfRequired("");
  for(i = 0; i < strlen(ac); i++) {
    ac[i] = 'A';
  }
  //
  // Check if volume needs to be high-level formatted.
  //
  if (FS_IsHLFormatted("") == 0) {
    printf("High level formatting\n");
    FS_Format("", NULL);
  }
  ShowDiskInfo(&DiskInfo);
  for (i = 0;  i < 1000; i++) {
    pFile = FS_FOpen("Test.txt","w");
    if(pFile != 0) {
      FS_Write(pFile, &ac, strlen(ac));
      FS_FClose(pFile);
      printf("Loop cycle: %d\n", i);
      for(j = 0; j < DiskInfo.NumPhysSectors; j++) {
        ShowSectorInfo(&SecInfo, j);
      }
    }
  }
  while(1);
}
```


6.4.1.10.3 FS_NOR_DISK_INFO

Description

The structure contains information about the NOR flash.

Declaration

```
typedef struct {
  U32 NumPhysSectors;
  U32 NumLogSectors;
  U32 NumUsedSectors;
} FS_NOR_DISK_INFO;
```

Members	Description
NumPhysSectors	Number physical sectors of the chip.
NumLogSectors	Number of logical sectors of the chip.
NumUsedSectors	Number of used sectors of the chip.

Table 6.115: FS_NOR_DISK_INFO - list of structure elements

6.4.1.10.4 FS_NOR_SECTOR_INFO

Description

The structure contains physical and logical sector information.

Declaration

```
typedef struct {
  U32 Off;
  U32 Size;
  U32 EraseCnt;
  U16 NumUsedSectors;
  U16 NumFreeSectors;
  U16 NumEraseableSectors;
} FS_NOR_SECTOR_INFO;
```

Members	Description
Off	Offset of the physical sector.
Size	Size of the physical sector.
EraseCnt	Erase count of sector.
NumUsedSectors	Number of used logical sector inside the physical sector.
NumFreeSectors	Number of free logical sector inside the physical sector.
NumEraseableSectors	Number of erasable logical sector inside the physical sector.

Table 6.116: FS_NOR_SECTOR_INFO - list of structure elements

6.4.1.11 Performance and resource usage

This section describes the ROM and RAM (static + dynamic) RAM usage of the emFile NOR driver.

6.4.1.11.1 ROM usage

The ROM usage depends on the compiler options, the compiler version, the used CPU and the physical layer which is used. The memory requirements of the NOR driver have been measured on a system as follows: ARM7, IAR Embedded workbench V5.50.1, Thumb mode, Size optimization.

Module	ROM [Kbytes]
emFile sector map NOR driver	4.0

In addition, one of the following physical layers is required:

Physical layer	Description	ROM [Kbytes]
FS_NOR_PHY_ST_M25	Physical layer for SPI NOR flash devices (ST M25Pxx family).	1.1
FS_NOR_PHY_CFI_1x16	Physical layer for CFI-compliant parallel NOR flash devices with a configuration of 1x16 (1 chip, 16-bits buswidth)	2.1
FS_NOR_PHY_CFI_2x16	Physical layer for CFI-compliant parallel NOR flash devices with a configuration of 2x16 (2 chips, 16-bits buswidth)	1.5

6.4.1.11.2 Static RAM usage

Static RAM usage is the amount of RAM required by the driver for static variables inside the driver. The number of bytes can be seen in a compiler list file

Module	RAM [bytes]
emFile sector map NOR driver	20
Physical layer: SPI	50
Physical layer: CFI 1x16	100
Physical layer: CFI 2x16	100

6.4.1.11.3 Runtime (dynamic) RAM usage

Runtime (dynamic) RAM usage is the amount of RAM allocated by the driver at runtime. The amount required depends on the runtime configuration and the connected device. The approximately RAM usage of the NOR flash driver can be calculated as follows:

```
MemAllocated = 500 + 2 * FlashSize / LogSectorSize
```

Parameter	Description
MemAllocated	Number of bytes allocated.
FlashSize	Size in bytes of a NOR flash.
LogSectorSize	Size in bytes of a file system sector. Typically 512 bytes or the value set in the call to FS_SetMaxSectorSize() configuration function.

Table 6.117: Runtime RAM usage parameters for FS_NOR_Driver

The following table lists the approximate amount of RAM required for different combinations of NOR flash size and logical sector size:

		Logical sector size		
		512 bytes	1024 bytes	2048 bytes
Flash Size	1Mbyte	4.6Kbyte	2.5Kbyte	1.5Kbyte
	2Mbyte	8.7Kbyte	4.6Kbyte	2.5Kbyte
	4Mbyte	16.8Kbyte	8.7Kbyte	4.6Kbyte
	8Mbyte	33.2Kbyte	16.8Kbyte	8.7Kbyte

Table 6.118: Runtime RAM usage examples for FS_NOR_Driver

Note: When choosing a bigger logical sector size keep in mind that the RAM usage of the file system increases as more space is needed for the sector buffers. There is an optimal logical sector size that depends on the flash size. For a 1Mbyte flash memory the ideal configuration is 1Kbyte sectors.

6.4.1.11.4 Performance

These performance measurements are in no way complete, but they give an approximation of the length of time required for common operations on various targets. The tests were performed as described in *Performance* on page 502.

All values are in Kbytes/sec

Device	CPU speed	Medium	W	R
ST STR912	96 MHz	Winbond W25Q32BV (SPI)	75	2625
NXP LPC2478	57.6MHz	SST SST39VF201 (CFI, 16-bit, without "write burst")	53.5	2560
ST STM32F103	72MHz	ST M29W128 (CFI, 16-bit, with "write burst")	60.4	8000
ST STM32F103	72MHz	ST M25P64 (SPI)	62.8	1125

Table 6.119: Performance values for FS_NOR_Driver

6.4.1.12 FAQs

Q: How many physical sectors are reserved by the driver?
A: The driver reserves 2 physical sectors for its internal use.

6.4.2 Block map - FS_NOR_BM_Driver

This section describes the NOR driver which is optimized for reduced RAM usage. It works by mapping blocks of logical sectors to locations on the NOR flash memory.

6.4.2.1 Supported hardware

The NOR flash drivers can be used with almost any NOR flash. This includes NOR flashes with 1x8-bit and 1x16-bit parallel interfaces, as well as 2x16-bit interfaces in parallel, as well as serial NOR flashes.

For additional information, refer to *Supported hardware* on page 315.

6.4.2.2 Theory of operation

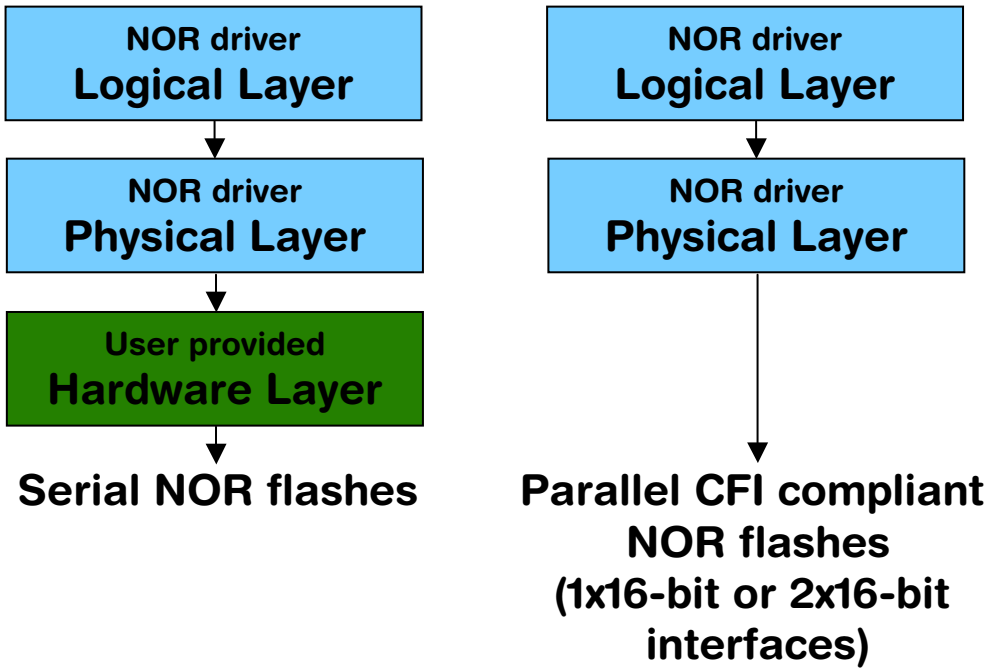
Differentiating between "logical sectors" or "blocks" and "physical sectors" is very essential to understand this section. A logical sector/block is the base unit of any file system, its usual size is 512 bytes. A physical sector is an array of bytes on the flash chip that are erased together (typically between 2 Kbytes - 128 Kbytes). The flash chip driver is an abstraction layer between these two types of sectors.

The driver maintains a table that maps ranges of logical sectors, called logical blocks, to physical sectors on the NOR flash. The number of logical sectors in a logical block depends on how many logical sectors fit in a physical sector. Every time a logical sector is being updated, its content is written to a special physical sector called work block. A work block is a temporary storage for the modified data of logical sectors. A work block is later converted into a data block when an empty work block is allocated.

For additional information, refer to *Using the same NOR flash for code and data* on page 317 and *Physical interfaces* on page 318.

6.4.2.2.1 Software structure

The NOR flash driver is divided into different layers, which are shown in the illustration below.



It is possible to use the NOR flash drivers also with serial NOR flashes. Only the hardware layer needs to be ported. Normally no changes to the physical layer are required. If the physical layer needs to be adapted, a template is available.

6.4.2.3 Fail-safe operation

The emFile NOR driver is fail-safe. That means that the driver makes only atomic actions and takes the responsibility that the data managed by the file system is always valid. In case of power loss or power reset during a write operation it is always assured that only valid data is stored in the flash. If the power loss interrupts the write operation, the old data will be kept and not corrupted.

6.4.2.4 Wear leveling

Wear leveling is supported by the driver. Wear leveling makes sure that the number of erase cycles remains approximately equal for each sector. Maximum erase count difference is set by default to 5000. This value specifies a maximum difference of erase counts for different physical sectors before the wear leveling uses the sector with the lowest erase count.

6.4.2.5 Configuring the driver

6.4.2.5.1 Adding the driver to emFile

To add the driver, use `FS_AddDevice()` with the driver label `FS_NOR_BM_Driver`. This function has to be called from `FS_X_AddDevices()`. Refer to *FS_X_AddDevices()* on page 472 for more information.

Example

```
FS_AddDevice(&FS_NOR_BM_Driver);
```

6.4.2.5.2 Configuration API

Routine	Explanation
`FS_NOR_BM_Configure()`	Configures the NOR flash.
`FS_NOR_BM_SetPhyType()`	Sets the physical type of NOR device.
`FS_NOR_BM_SetSectorSize()`	Sets the size of a logical sector.
`FS_NOR_BM_SetMaxEraseCntDiff()`	Configures the threshold for the wear-leveling.
`FS_NOR_BM_SetNumWorkBlocks()`	Sets the number of work blocks.

Table 6.120: FS_NOR_BM_Driver - list of configuration functions

6.4.2.5.2.1 FS_NOR_BM_Configure()

Description

Configures the NOR flash drive. Needs to be called for CFI flashes. Typically, this function has to be called from `FS_X_AddDevices()` after adding the device driver to file system. Refer to *FS_X_AddDevices()* on page 472 for more information.

Prototype

```
void FS_NOR_BM_Configure(U8  Unit,
                         U32 BaseAddr,
                         U32 StartAddr,
                         U32 NumBytes);
```

Parameter	Description
Unit	Unit number (0…N).
BaseAddr	Base address of the NOR flash chip. This is the address of the first byte of the NOR flash.
StartAddr	Start address of the NOR flash disk. This is the address of the first byte of the NOR flash to be used as flash disk. It needs to be >= BaseAddr.
NumBytes	Specifies the size of the NOR flash device in bytes. The size of the flash disk will be: `min(NumBytes, DeviceSize - (StartAddr - BaseAddr)` where DeviceSize is the size of the NOR flash found.

Table 6.121: FS_NOR_BM_Configure() parameter list

Additional information

The driver is designed to work only with physical sectors of the same size. If the configured memory area contains physical sectors of different sizes the driver chooses the range containing the highest number of physical sectors of the same size. From the selected physical sectors several are reserved for internal use: 1 to store the format information, 1 for the copy operations and 1 for each configured work block.

Example

For configuration examples, refer to *FS_NOR_Configure()* on page 322.

6.4.2.5.2.2 FS_NOR_BM_SetPhyType()

Description

Sets the physical type of the device. The NOR flash driver comes with different physical interfaces. The most common is a CFI compliant NOR flash chip with a 16 bit interface. A device can consist of a single or two identical CFI compliant flash interfaces with a 16 bit interface. Set `pPhyType` to `FS_NOR_PHY_CFI_1x16` if you use a single NOR flash chip. If your device consists of two identical NOR flash chips, set `pPhyType` to `FS_NOR_PHY_CFI_2x16`.

This function has to be called from within `FS_X_AddDevices()` after adding the device driver to file system. Refer to *FS_X_AddDevices()* on page 472 for more information.

Prototype

```
void FS_NOR_BM_SetPhyType(U8 Unit, const FS_NOR_PHY_TYPE * pPhyType);
```

Parameter	Meaning
`Unit`	Unit number (0…N).
`pPhyType`	Pointer to physical type.

Table 6.122: FS_NOR_BM_SetPhyType() parameter list

For additional information, refer to *FS_NOR_SetPhyType()* on page 324.

6.4.2.5.2.3 FS_NOR_BM_SetSectorSize()

Description

Configures the size of a logical sector on the NOR flash drive. Typically, this function has to be called from `FS_X_AddDevices()` after adding the device driver to file system. Refer to *FS_X_AddDevices()* on page 472 for more information.

Prototype

```
void FS_NOR_BM_SetSectorSize(U8  Unit,
                             U16 SectorSize);
```

Parameter	Description
`Unit`	Unit number.
`SectorSize`	Number of bytes in a logical sector.

Table 6.123: FS_NOR_BM_SetSectorSize() parameter list

For additional information, refer to *FS_NOR_SetSectorSize()* on page 325.

6.4.2.5.2.4 FS_NOR_BM_SetMaxEraseCntDiff()

Description

Configures the maximum difference between the number of erase cycles of two any physical sectors. This value is used by the wear-leveling algorithm to decide which physical sector to erase.

Prototype

```
void FS_NOR_BM_SetMaxEraseCntDiff(U8  Unit,
                                  U16 EraseCntDiff);
```

Parameter	Description
Unit	Unit number.
EraseCntDiff	Maximum difference between erase counts.

Table 6.124: FS_NOR_BM_SetMaxEraseCntDiff() parameter list

Additional information

Each physical sector stores the number of times it has been erased since the last low-level format. This count is used by the driver to ensure that the physical sectors are equally-well used. When a write operation required a new physical sector, the driver takes the next free one. It then computes the difference between the erase count of the chosen physical sector and the minimum erase count of all physical sectors. When the difference is greater than the value configured by this function the physical sector with the minimum erase count is selected as the next physical sector to write to.

6.4.2.5.2.5 FS_NOR_BM_SetNumWorkBlocks()

Description

Number of logical blocks to be used as temporarily storage for the data written to NOR flash.

Prototype

```
void FS_NOR_BM_SetNumWorkBlocks(U8  Unit,
                                U16 NumWorkBlocks);
```

Parameter	Description
Unit	Unit number.
NumWorkBlocks	Number of work blocks the driver should use for write operations.

Table 6.125: FS_NOR_BM_SetNumWorkBlocks() parameter list

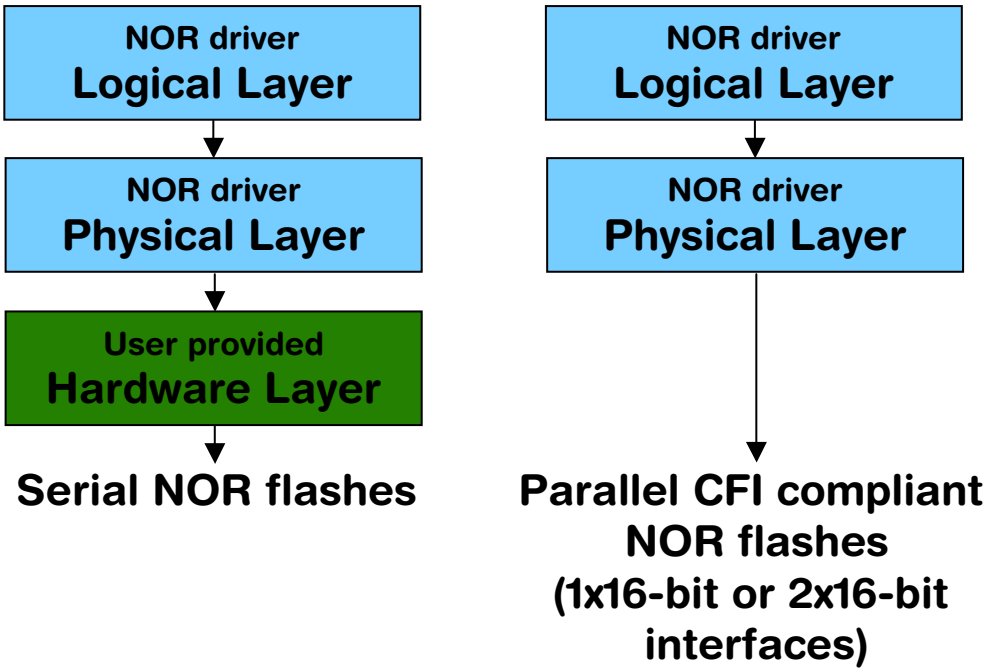
Additional information

Work blocks are physical sectors which the driver uses to temporarily store the data written to NOR flash. This function can be used to change the number of work blocks according to the requirements of an application. Usually, the write performance of the driver improves when the number work blocks is increased. Please note that increasing the number of work blocks will also increase the RAM usage. By default, the driver allocates 10% from the total number of blocks available but no more than 10 blocks. The minimum number of work blocks allocated by default depends whether journaling is used or not. If the journal is active the 4 work blocks are allocated else 3.

6.4.2.6 Physical layer

There is normally no need to change the physical layer of the NOR driver, only the hardware layer has to be adapted if a non CFI compliant NOR flash chip is used in your hardware.

In some special cases, when the low-level hardware routines provided by the driver are not compatible with the target hardware the physical layer has to be adapted.



6.4.2.6.1 Available physical layers

The following physical layers are available. Refer to *Configuring the driver* on page 358 for detailed information about how to add the required physical layer to your application.

Available physical layers	
FS_NOR_PHY_CFI_1x16	One CFI compliant NOR flash chip with 16 bit interface.
FS_NOR_PHY_CFI_2x16	Two CFI compliant NOR flash chip with 16 bit interfaces.
FS_NOR_PHY_ST_M25	Serial NOR flashes.

Table 6.126: Available physical layer

For a detailed description of the physical layer functions, refer to *Physical layer* on page 331.

6.4.2.7 Hardware functions

Depending on the used NOR flash type and the corresponding physical layer, different hardware functions are required. CFI compliant NOR flashes do not need any hardware function, refer to *Hardware functions* on page 340 for detailed information about the hardware functions required by the physical layer for serial NOR flashes.

6.4.2.8 Additional information

Low-level format

Before using the NOR flash as storage device. A low-level format has to be performed. Refer to *FS_FormatLow()* on page 113 and *FS_FormatLLIfRequired()* on page 112 for detailed information about low-level formatting.

Further reading

For more technical details about CFI compliant flash memory, check the documents and specifications that are available free of charge:

- *Common Flash Interface (CFI) and Command Sets*
  Intel - Application Note 646 - April 2000
- *Common Flash Memory Interface Specification*
  AMD - Revision 2.0 - December 1, 2001

6.4.2.9 Additional driver functions

Routine	Explanation
FS_NOR_BM_GetDiskInfo()	Returns information about NOR flash.
FS_NOR_BM_ReadOff()	Reads data from NOR flash memory.

Table 6.127: FS_NOR_BM_Driver - list of configuration functions

6.4.2.9.1 FS_NOR_BM_GetDiskInfo()

Description

Returns information about the NOR flash.

Prototype

```
void FS_NOR_BM_GetDiskInfo(U8 Unit, FS_NOR_BM_DISK_INFO * pDiskInfo);
```

Parameter	Description
Unit	Unit number.
pDiskInfo	Pointer to a structure of type FS_NOR_DISK_INFO.

Table 6.128: FS_NOR_BM_GetDiskInfo() parameter list

Additional information

Refer to *Structure FS_NOR_BM_DISK_INFO* on page 369 for more information about the structure elements.

6.4.2.9.2 FS_NOR_BM_GetSectorInfo()

Description

Returns info about a particular physical sector.

Prototype

```
void FS_NOR_GetSectorInfo(U8                       Unit,
                          U32                      PhysSectorIndex,
                          FS_NOR_BM_SECTOR_INFO * pSectorInfo);
```

Parameter	Description
Unit	Unit number.
PhysSectorIndex	Index of physical sector.
pDiskInfo	Pointer to a structure of type FS_NOR_BM_DISK_INFO.

Table 6.129: FS_NOR_BM_GetSectorInfo() parameter list

Additional information

Refer to *FS_NOR_SECTOR_INFO* on page 354 for more information about the structure elements.

6.4.2.9.3 FS_NOR_BM_ReadOff()

Description

Reads data from NOR flash memory.

Prototype

```
int FS_NOR_BM_ReadOff(U8 Unit, void * pData, U32 Off, U32 NumBytes)
```

Parameter	Description
Unit	Unit number.
pData	IN: --- OUT: Data read from NOR flash memory,
Off	Byte offset from the configured base address.
NumBytes	Number of bytes to read

Table 6.130: FS_NOR_BM_ReadOff() parameter list

Return value

==0 OK
!=0 An error occurred

Additional information

This function can be used to dump a part or the whole contents of a NOR flash. It works even if the NOR flash is not low-level formatted.

6.4.2.9.4 Structure FS_NOR_BM_DISK_INFO

Description

The structure contains information about the NOR flash.

Declaration

```
typedef struct {
  U16 NumPhySectors;
  U16 NumLogBlocks;
  U16 NumUsedPhySectors;
  U16 LSectorsPerPSector;
  U16 BytesPerSector;
  U32 EraseCntMax;
  U32 EraseCntMin;
  U32 EraseCntAvg;
  U8  HasFatalError;
  U8  ErrorType;
  U16 ErrorPSI;
} FS_NOR_DISK_INFO;
```

Members	Description
NumPhysSectors	Number physical sectors on the NOR flash.
NumLogBlock	Number of logical blocks on the NOR flash.
NumUsedPhySectors	Number of used physical sectors.
LSectorsPerPSector	Number of logical sectors stored in a physical sector.
BytesPerSector	Number of bytes in a logical sector.
EraseCntMax	Maximum erase count of all physical sectors.
EraseCntMin	Minimum erase count of all physical sectors.
EraseCntAvg	Average erase count.
HasFatalError	Set to 1 if the driver detected a fatal error.
ErrorType	Type of fatal error occurred.
ErrorPSI	Index of physical sector where the error occurred.

Table 6.131: FS_NOR_BM_DISK_INFO - list of structure elements

6.4.2.9.5 FS_NOR_BM_SECTOR_INFO

Description

The structure contains physical and logical sector information.

Declaration

```
typedef struct {
  U32 Off;
  U32 Size;
  U32 EraseCnt;
  U16 lbi;
  U8  Type;
} FS_NOR_SECTOR_INFO;
```

Members	Description
Off	Offset of the first byte relative to begin of NOR flash.
Size	Size of the physical sector in bytes.
EraseCnt	Number of times the physical sector has been erased.
lbi	Index of the logical block stored in the physical sector.
Type	Type of data stored in the physical sector.

Table 6.132: FS_NOR_BM_SECTOR_INFO - list of structure elements

Permitted values for Types	
FS_NOR_BLOCK_TYPE_UNKNOWN	The type of data stored to physical sector is unknown.
FS_NOR_BLOCK_TYPE_DATA	The physical sector contains valid data block.
FS_NOR_BLOCK_TYPE_WORK	The physical sector contains a work block.
FS_NOR_BLOCK_TYPE_EMPTY_ERASED	The physical sector is blank.
FS_NOR_BLOCK_TYPE_EMPTY_NOT_ERASED	The physical sector contains old and invalid data.

6.4.2.10 Performance and resource usage

This section describes the ROM and RAM (static + dynamic) RAM usage of the emFile NOR driver.

6.4.2.10.1 ROM usage

The ROM usage depends on the compiler options, the compiler version, the used CPU and the physical layer which is used. The memory requirements of the NOR driver have been measured on a system as follows: ARM7, IAR Embedded workbench V5.50.1, Thumb mode, Size optimization.

Module	ROM [Kbytes]
emFile block map NOR driver:	5.5

In addition, one of the following physical layers is required:

Physical layer	Description	ROM [Kbytes]
FS_NOR_PHY_ST_M25	Physical layer for SPI NOR flash devices (ST M25Pxx family).	1.1
FS_NOR_PHY_CFI_1x16	Physical layer for CFI-compliant parallel NOR flash devices with a configuration of 1x16 (1 chip, 16-bits buswidth)	2.1
FS_NOR_PHY_CFI_2x16	Physical layer for CFI-compliant parallel NOR flash devices with a configuration of 2x16 (2 chips, 16-bits buswidth)	1.5

6.4.2.10.2 Static RAM usage

Static RAM usage is the amount of RAM required by the driver for static variables inside the driver. The number of bytes can be seen in a compiler list file

Module	RAM [bytes]
emFile block map NOR driver:	72
Physical layer: SPI	50
Physical layer: CFI 1x16	100
Physical layer: CFI 2x16	100

6.4.2.10.3 Runtime (dynamic) RAM usage

Runtime (dynamic) RAM usage is the amount of RAM allocated by the driver at runtime. The amount required depends on the runtime configuration and the connected device. The approximately amount of RAM required by the driver can be computed using the following formula:

```
MemAllocated = 84
             + (24 + PhySectorSize / LogSectorSize) * NumWorkBlocks
             + 1.5 * NumPhySectors
```

Parameter	Description
MemAllocated	Number of bytes allocated.
PhySectorSize	Size in bytes of a NOR flash physical sector.

Table 6.133: Runtime RAM usage parameters for FS_NOR_BM_Driver

Parameter	Description
LogSectorSize	Size in bytes of a file system sector. Typically 512 bytes or the value set in the call to FS_SetMaxSectorSize() configuration function.
NumWorkBlocks	Number of physical sectors the driver reserves as temporary storage for the written data. Typically 3 physical sectors or the number specified in the call to the FS_NOR_BM_SetNumWorkBlocks() configuration function
NumPhySectors	Number of physical sectors managed by the driver.

Table 6.133: Runtime RAM usage parameters for FS_NOR_BM_Driver

6.4.2.10.4 Performance

These performance measurements are in no way complete, but they give an approximation of the length of time required for common operations on various targets. The tests were performed as described in *Performance* on page 502.

All values are in Kbytes/sec

Device	CPU speed	Medium	W	R
NXP LPC2478	57.6MHz	SST SST39VF201 (CFI, 16-bit, without "write burst")	45.5	2064
ST STM32F103	72MHz	ST M25P64 (SPI)	59.3	1110

Table 6.134: Performance values for FS_NOR_BM_Driver

6.5 MMC/SD card driver

emFile supports the use of MultiMediaCard (MMC) and SecureDigital (SD) cards through the use of optional drivers. MMC/SD cards are mechanically small, removable mass storage devices. The main design goal of these devices are to provide a very low cost mass storage product, implemented as a card with a simple controlling unit, and a compact, easy-to-implement interface. These requirements lead to a reduction of the functionality of each card to an absolute minimum. In order to have a flexible design, MMC/SD cards are designed to be used in different I/O modes:

- SPI mode
- card mode

Separate drivers are available for both of these modes. The drivers require very little RAM and are extremely efficient. To use one of these drivers, you need first to select one according to the access mode. Then you need to configure the selected MMC/SD driver and provide basic I/O functions for accessing your card reader hardware.

This section describes how to enable each of these drivers and what hardware access functions these drivers require.

6.5.1 Supported hardware

The following card types are supported:

- MMC: MMC, RS-MMC, RS-MMC DV, MMCplus, MMCmobile, MMCmicro, eMMC
- SD: SD, miniSD, microSD, SDHC, SDXC (FAT32 formatted)

Note: The MMC cards conforming with the version 4.x (MMCplus, MMCmobile, MMCmicro, eMMC) work only with the card mode driver as they don't support the SPI mode.

The difference between MMC and SD cards are that SD cards can operate with a higher clock frequency. In normal mode the clock range can be between 0 and 25MHz, whereas MMCs can only operate up to 20MHz. The newer MMC cards that adhere to the version 4.x of the MMC system specification can also operate at higher frequencies up to 26MHz. In high speed mode an SD card can operate with a clock frequency up to 50MHz. The MMC cards conforming to the 3.x standard or lower didn't have a high speed mode. The 4.x improved this and allows the MMC cards to operate with a clock frequency of up to 52MHz in high speed mode.

Additionally SD cards have a write protect switch, which can be used to lock the data on the card.

MMC and SD cards also differ in the number of pins. SD cards have typically more pins than MMCs. Which pins are used depends on which mode is configured.

In card mode

MMC cards use a seven pin interface: command, clock, data and 3 power lines. In contrast to the MMC cards, SD cards use a 9 pin interface: command, clock, 1 or 4 data lines and 3 power lines. The MMC cards version 4.x can have 1, 4 or 8 data lines.

In SPI mode

Both card systems use the same pin interface: chip select, data input, data output, clock and 3 power lines.

6.5.1.1 Pin description for MMC/SD card in Card mode

Pin No.	Name	Type	Description
1	CD/ DAT[3]	Input/Output using push pull drivers	Card Detect / Data line [Bit 3] After power up this line is input with 50-kOhm pull-up resistor. This can be used for card detection; relevant only for SD cards. The pull-up resistor is disabled after the initialization procedure for using this line as DAT3, Data line [Bit 3], for data transfer.
2	CMD	Push Pull	Command/Response CMD is a bidirectional command channel used for card initialization and data transfer commands. The CMD signal has two operation modes: open-drain for initialization mode and push-pull for fast command transfer. Commands are sent from the MultiMediaCard bus master (card host controller) to the card and responses are sent from the cards to the host.
3	V_{SS}	Power supply	Supply voltage ground.
4	V_{DD}	Power supply	Supply voltage.
5	CLK	Input	Clock signal With each cycle of this signal an one bit transfer on the command and data lines is done. The frequency may vary between zero and the maximum clock frequency.
6	V_{SS2}	Power supply	Supply voltage ground.
7	DAT0	Input/Output using push pull drivers	Data line [Bit 0] DAT is a bidirectional data channel. The DAT signal operates in push-pull mode. Only one card or the host is driving this signal at a time. Relevant only for SD cards: For data transfers, this line is the Data line [Bit 0].
8	DAT1	Input/Output using push pull drivers	Data line [Bit 1] On MMC card this line does not exist. Relevant only for SD cards: For data transfer, this line is the Data line [Bit 1]. Connect an external pull-up resistor to this data line even if only DAT0 is to be used. Otherwise, non-expected high current consumption may occur due to the floating inputs of DAT1.
9	DAT2	Input/Output using push pull drivers	Data line [Bit 2] On MMC card this line does not exist. Relevant only for SD cards: For data transfer, this line is the Data line [Bit 2]. Connect an external pull-up resistor to this data line even if only DAT0 is to be used. Otherwise, non-expected high current consumption may occur due to the floating inputs of DAT2.
10	DAT4	Input/Output using push pull drivers	Data line [Bit 4] Relevant only for eMMC cards. Not used for data transfers over four data lines. For data transfer over eight data lines, this line is carries the bit 5. On SD cards this line does not exist.

Table 6.135: MMC/SD card pin description (card mode)

Pin No.	Name	Type	Description
11	DAT5	Input/Output using push pull drivers	Data line [Bit 5] Relevant only for eMMC cards. Not used for data transfers over four data lines. For data transfer over eight data lines, this line is carries the bit 5. On SD cards this line does not exist.
12	DAT6	Input/Output using push pull drivers	Data line [Bit 6] Relevant only for eMMC cards. Not used for data transfers over four data lines. For data transfer over eight data lines, this line is carries the bit 6. On SD cards this line does not exist.
13	DAT7	Input/Output using push pull drivers	Data line [Bit 7] Relevant only for eMMC cards. Not used for data transfers over four data lines. For data transfer over eight data lines, this line is carries the bit 7. On SD cards this line does not exist.

Table 6.135: MMC/SD card pin description (card mode)

Pin No.	Name	Type	Description
1	DAT2	Input/Output using push pull drivers	Data line [Bit 2] On MMC card this line does not exist. Relevant only for SD cards: For data transfer, this line is the Data line [Bit 2]. Connect an external pull-up resistor to this data line even if only DAT0 is to be used. Otherwise, non-expected high current consumption may occur due to the floating inputs of DAT2.
2	CD/ DAT[3]	Input/Output using push pull drivers	Card Detect / Data line [Bit 3] After power up this line is input with 50-kOhm pull-up resistor. This can be used for card detection; relevant only for SD cards. The pull-up resistor is disabled after the initialization procedure for using this line as DAT3, Data line [Bit 3], for data transfer.
3	CMD	Push Pull	Command/Response CMD is a bidirectional command channel used for card initialization and data transfer commands. The CMD signal has two operation modes: open-drain for initialization mode and push-pull for fast command transfer. Commands are sent from the MultiMediaCard bus master (card host controller) to the card and responses are sent from the cards to the host.
4	V_{DD}	Power supply	Supply voltage
5	CLK	Input	Clock signal With each cycle of this signal an one bit transfer on the command and data lines is done. The frequency may vary between zero and the maximum clock frequency.

Table 6.136: microSD card pin description (card mode)

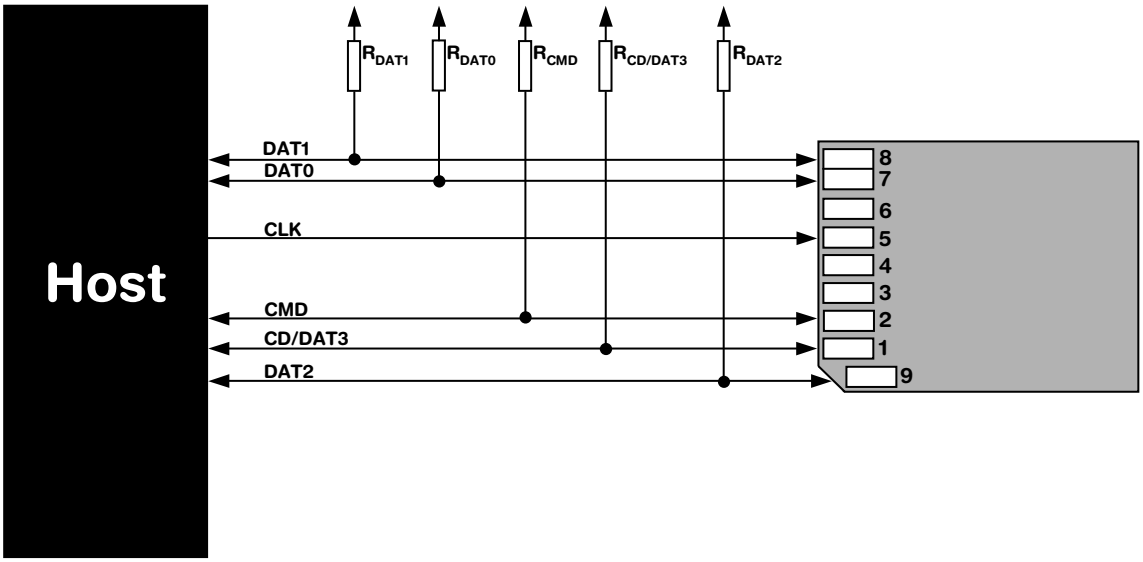
Pin No.	Name	Type	Description
6	V_{SS}	Power supply	Supply ground
7	DAT0	Input/Output using push pull drivers	Data line [Bit 0] DAT is a bidirectional data channel. The DAT signal operates in push-pull mode. Only one card or the host is driving this signal at a time. Relevant only for SD cards: For data transfers, this line is the Data line [Bit 0].
8	DAT1	Input/Output using push pull drivers	Data line [Bit 1] On MMC card this line does not exist. Relevant only for SD cards: For data transfer, this line is the Data line [Bit 1]. Connect an external pull-up resistor to this data line even if only DAT0 is to be used. Otherwise, non-expected high current consumption may occur due to the floating inputs of DAT1.

Table 6.136: microSD card pin description (card mode)

Additional information

- External pull-up resistors must be connected to all data lines even it they are not used. Otherwise, non-expected high current consumption may occur due to the floating of these inputs.

Sample schematic for MMC/SD card in Card mode



6.5.1.2 Pin description for MMC/SD card in SPI mode

Pin No.	Name	Type	Description
1	CS	Input	Chip Select It sets the card active at low-level and inactive at high level.
2	DataIn (MOSI)	Input	Data Input (Master Out Slave In) Transmits data to the card.
3	V_{SS}	Supply ground	Power supply ground Supply voltage ground.
4	V_{DD}	Supply voltage	Supply voltage
5	SCLK	Input	Clock signal It must be generated by the target system. The card is always in slave mode.
6	V_{SS2}	Supply ground	Supply ground
7	DataOut (MISO)	Output	Data Output (Master In Slave Out) Line to transfer data to the host.
8	Reserved	Not used	The reserved pin is a floating input. Therefore, connect an external pull-up resistor to it. Otherwise, non-expected high current consumption may occur due to the floating input.
9	Reserved	Not used	The reserved pin is a floating input. Therefore, connect an external pull-up resistor to it. Otherwise, non-expected high current consumption may occur due to the floating input.

Table 6.137: MMC/SD card pin description (SPI mode)

Pin No.	Name	Type	Description
1	Reserved	Not used	The reserved pin is a floating input. Therefore, connect an external pull-up resistor to it. Otherwise, non-expected high current consumption may occur due to the floating input.
2	CS	Input	Chip Select It sets the card active at low-level and inactive at high level.
3	DataIn (MOSI)	Input	Data Input (Master Out Slave In) Transmits data to the card.
4	V_{DD}	Supply voltage	Supply voltage
5	SCLK	Input	Clock signal It must be generated by the target system. The card is always in slave mode.
6	V_{SS}	Supply ground	Supply ground
7	DataOut (MISO)	Output	Data Output (Master In Slave Out) Line to transfer data to the host.
8	Reserved	Not used	The reserved pin is a floating input. Therefore, connect an external pull-up resistor to it. Otherwise, non-expected high current consumption may occur due to the floating input.

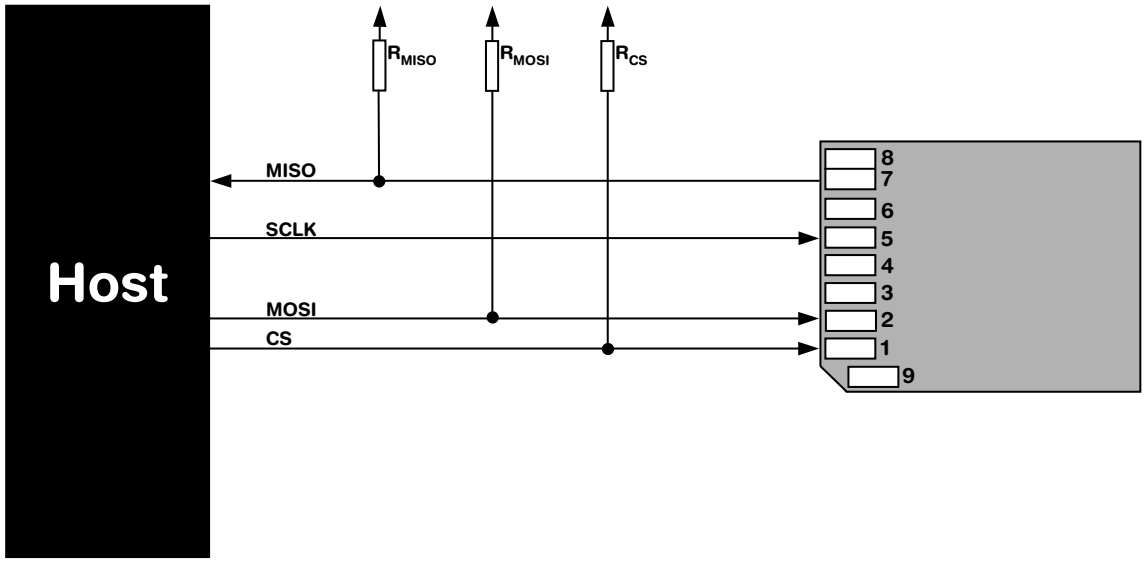
Table 6.138: microSD card pin description (SPI mode)

Additional information

- The data transfer width is 8 bits.
- Data should be output on the falling edge and must remain valid until the next period. Rising edge means data is sampled (i.e. read).
- The bit order requires most significant bit (MSB) to be sent out first.
- Data polarity is normal, which means a logical "1" is represented with a high level on the data line and a logical "0" is represented with low-level.
- MMC/SD cards support different voltage ranges. Initial voltage should be 3.3V.

Power control should be considered when creating designs using the MMC and/or SD cards. The ability to have software power control of the cards makes the design more flexible and robust. The host will be able to turn power to the card on or off independent of whether the card is inserted or removed. This can improve card initialization when there is a contact bounce during card insertion. The host waits a specified time after the card is inserted before powering up the card and starting the initialization process. Also, if the card goes into an unknown state, the host can cycle the power and start the initialization process again. When card access is unnecessary, allowing the host to power-down the bus can reduce the overall power consumption.

Sample schematic for MMC/SD card in SPI mode



6.5.2 Theory of operation

The Serial Peripheral Interface (SPI) bus is a very loose de facto standard for controlling almost any digital electronics that accepts a clocked serial stream of bits. SPI operates in full duplex (sending and receiving at the same time).

6.5.3 Fail-safe operation

Unexpected Reset

The data will be preserved.

Power failure

Power failure can be critical: If the card does not have sufficient time to complete a write operation, data may be lost. Countermeasures: make sure the power supply for the card drops slowly.

6.5.4 Wear leveling

MMC/SD cards are controlled by an internal controller, this controller also handles wear leveling. Therefore, the driver does not need to handle wear-leveling.

6.5.5 Configuration

6.5.5.1 Adding the driver to emFile

To add the driver use `FS_AddDevice()` with either the driver label `FS_MMC_SPI_Driver` or `FS_MMC_CardMode_Driver`. This function has to be called from within `FS_X_AddDevices()`. Refer to *FS_X_AddDevices()* on page 472 for more information.

Example

SPI mode: `FS_AddDevice(&FS_MMC_SPI_Driver);`

Card mode: `FS_AddDevice(&FS_MMC_CardMode_Driver);`

6.5.5.2 Enable 4-bit mode (card mode only)

To enable the 4-bit mode of the card mode driver, call `FS_MMC_CM_Allow4bitMode()`. Refer to *FS_MMC_CM_Allow4bitMode()* on page 417 for detailed information.

6.5.5.3 Cyclic redundancy check (CRC)

The cyclic redundancy check (CRC) is a method to produce a checksum. The checksum is a small, fixed number of bits against a block of data. The checksum is used to detect errors after transmission or storage. A CRC is computed and appended before transmission or storage, and verified afterwards by the recipient to confirm that no changes occurred on transit. CRC is a good solution for error detection, but reduces the transmission speed, because a CRC checksum has to be computed for every data block which will be transmitted.

The following functions can be used for controlling CRC calculation in emFile.

Function	Description
CRC configuration	
`FS_MMC_ActivateCRC()`	Activates the CRC functionality in SPI mode.
`FS_MMC_DeactivateCRC()`	Deactivates the CRC functionality in SPI mode. By default, CRC is deactivated.

Table 6.139: SPI mode configuration functions

6.5.5.4 FS_MMC_ActivateCRC()

Description

Activates the cyclic redundancy check.

Prototype

```
void FS_MMC_ActivateCRC(void);
```

Additional information

By default, the cyclic redundancy check is deactivated for speed reasons. The driver supports cyclic redundancy check both for all transmissions and just for critical transmissions. You can activate and deactivate the cyclic redundancy check as it fits to the requirements of your application.

6.5.5.5 FS_MMC_DeactivateCRC()

Description

Deactivates the cyclic redundancy check.

Prototype

```
void FS_MMC_DeactivateCRC(void);
```

Additional information

By default, the cyclic redundancy check is deactivated for speed reasons. The driver supports cyclic redundancy check both for all transmissions and just for critical transmissions. You can activate and deactivate the cyclic redundancy check as it fits to the requirements of your application.

6.5.6 Hardware functions - SPI mode

Routine	Explanation
Control line functions	
FS_MMC_HW_X_EnableCS()	Activates chip select signal (CS) of the specified card slot.
FS_MMC_HW_X_DisableCS()	Deactivates chip select signal (CS) of the specified card slot.
Operation condition detection and adjusting	
FS_MMC_HW_X_SetMaxSpeed()	Sets the SPI clock speed. The value is represented in thousand cycles per second (kHz).
FS_MMC_HW_X_SetVoltage()	Sets the operating voltage range for the MultiMedia & SD card slot.
Medium status functions	
FS_MMC_HW_X_IsWriteProtected()	Checks the status of the mechanical write protection of a SD card.
FS_MMC_HW_X_IsPresent()	Checks whether a card is present or not.
Data transfer functions	
FS_MMC_HW_X_Read()	Receives a number of bytes from the card.
FS_MMC_HW_X_Write()	Sends a number of bytes to the card.

Table 6.140: SPI mode hardware functions

6.5.6.1 FS_MMC_HW_X_EnableCS()

Description

Activates chip select signal (CS) of the specified card slot.

Prototype

```
void FS_MMC_HW_X_EnableCS(U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0-based).

Table 6.141: FS_MMC_HW_X_EnableCS() parameter list

Additional Information

The CS signal is used to address a specific card slot connected to the SPI. Enabling is equal to setting the CS line to low-level.

Example

```
void FS_MMC_HW_X_EnableCS(U8 Unit) {
  SPI_CLR_CS();
}
```


6.5.6.2 FS_MMC_HW_X_DisableCS()

Description

Deactivates chip select signal (CS) of the specified card slot.

Prototype

```
void FS_MMC_HW_X_DisableCS(U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0-based).

Table 6.142: FS_MMC_HW_X_DisableCS() parameter list

Additional Information

The CS signal is used to address a specific card slot connected to the SPI. Disabling is equal to setting the CS line to high.

Example

```
void FS_MMC_HW_X_DisableCS(U8 Unit) {
  SPI_SET_CS();
}
```


6.5.6.3 FS_MMC_HW_X_SetMaxSpeed()

Description

Sets the maximum SPI speed. If the hardware is unable to use this speed, a lower frequency can always be selected. The value is given in kHz.

Prototype

```
U16 FS_MMC_HW_X_SetMaxSpeed(U8  Unit,
                            U16 MaxFreq);
```

Parameter	Meaning
Unit	Unit number (0-based).
MaxFreq	Clock speed (kHz) between host and card.

Table 6.143: FS_MMC_HW_X_SetMaxSpeed() parameter list

Return value

Actual frequency in thousand cycles per second (kHz)
0 if the frequency could not be set.

Additional Information

Make sure your SPI interface never generates a higher clock than MaxFreq specifies. You can always run MultiMedia & SD cards at lower or equal, but never on higher frequencies. The initial frequency must be 400kHz or less. If the precise frequency is unknown (typical for implementation using port-pins "bit-banging"), the return value should be less than the maximum frequency, leading to longer timeout values, which is in general unproblematic. You have to return the actual clock speed of your SPI interface, because emFile needs the actual frequency to calculate timeout values.

Example using port pins

```
#define MMC_MAXFREQUENCY    400

U16 FS_MMC_HW_X_SetMaxSpeed(U8 Unit, U16 MaxFreq) {
  _Init();
  return MMC_MAXFREQUENCY;    /* We are not faster than this */
}
```

Example using SPI mode

```
U16 FS_MMC_HW_X_SetMaxSpeed(U8 Unit, U16 MaxFreq) {
  U32 InFreq;
  U32 SPIFreq;

  if (MaxFreq < 400) {
    MaxFreq = 400;
  }
  SPIFreq = 1000 * MaxFreq;
  if (SPIFreq >= 200000) {
    InFreq = 48000000;
  }
  _sbcr = (InFreq + SPIFreq - 1) / SPIFreq;
  _InitSPI();
  return MaxFreq;    /* We are not faster than this */
}
```


6.5.6.4 FS_MMC_HW_X_SetVoltage()

Description

Sets the operating voltage range for the MultiMedia & SD card slot.

Prototype

```
char FS_MMC_HW_X_SetVoltage(U8  Unit,
                            U16 Vmin,
                            U16 Vmax);
```

Parameter	Meaning
Unit	Unit number (0-based).
Vmin	Minimum supply voltage in mV.
Vmax	Maximum supply voltage in mV.

Table 6.144: FS_MMC_HW_X_SetVoltage() parameter list

Return value

== 1: Card slot works within the given range.
== 0: Card slot cannot provide a voltage within given range.

Additional Information

The values are in mill volts (mV). 1mV is 0.001V. All cards work with the initial voltage of 3.3V. If you want to save power you can adjust the card slot supply voltage within the given range of Vmin and Vmax.

Example

```
#define FS__MMC_DEFAULTSUPPLYVOLTAGE  3300 /* example means 3.3V */

char FS_MMC_HW_X_SetVoltage(U8 Unit, U16 Vmin, U16 Vmax) {
  /* voltage range check */
  char r;
  if((Vmin <= MMC_DEFAULTSUPPLYVOLTAGE) && (Vmax >= MMC_DEFAULTSUPPLYVOLTAGE)) {
    r = 1;
  } else {
    r = 0;
  }
  return r;
}
```


6.5.6.5 FS_MMC_HW_X_IsWriteProtected()

Description

Checks the status of the mechanical write protection of a SD card.

Prototype

```
char FS_MMC_HW_X_IsWriteProtected(U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0-based).

Table 6.145: FS_MMC_HW_X_IsWriteProtected() parameter list

Return value

== 0: If the card is not write protected.
== 1: Means that the card is write protected.

Additional Information

MultiMedia cards do not have mechanical write protection switches and should always return 0. If you are using SD cards, be aware that the mechanical switch does not really protect the card physically from being overwritten; it is the responsibility of the host to respect the status of that switch.

Example

```
char FS_MMC_HW_X_IsWriteProtected(U8 Unit) {
  return 0; /* If the card slot has no write switch detector, return 0 */
}
```


6.5.6.6 FS_MMC_HW_X_IsPresent()

Description

Checks whether a card is present or not.

Prototype

```
char FS_MMC_HW_X_IsPresent(U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0-based).

Table 6.146: FS_MMC_HW_X_IsPresent() parameter list

Return value

Return value	Description
FS_MEDIA_STATE_UNKNOWN	The card state is unknown.
FS_MEDIA_NOT_PRESENT	A card is not present.
FS_MEDIA_IS_PRESENT	A card is present.

Table 6.147: FS_MMC_HW_X_IsPresent() - list of return values

Additional Information

Usually, a card slot provides a hardware signal that can be used for card presence determination. The sample code below is for a specific hardware that does not have such a signal. Therefore, the presence of a card is unknown and you have to return FS_MEDIA_STATE_UNKNOWN. Then emFile tries reading the card to figure out if a valid card is inserted into the slot.

Example

```
char FS_MMC_HW_X_IsPresent(U8 Unit) {
  return FS_MEDIA_STATE_UNKNOWN;
}
```


6.5.6.7 FS_MMC_HW_X_Read()

Description

Receives a number of bytes from the card.

Prototype

```
void FS_MMC_HW_X_Read(U8    Unit,
                      U8  * pData,
                      int   NumBytes);
```

Parameter	Meaning
Unit	Unit number (0-based).
pData	Pointer to a buffer for data to receive.
NumBytes	Number of bytes to receive.

Table 6.148: FS_MMC_HW_X_Read() parameter list

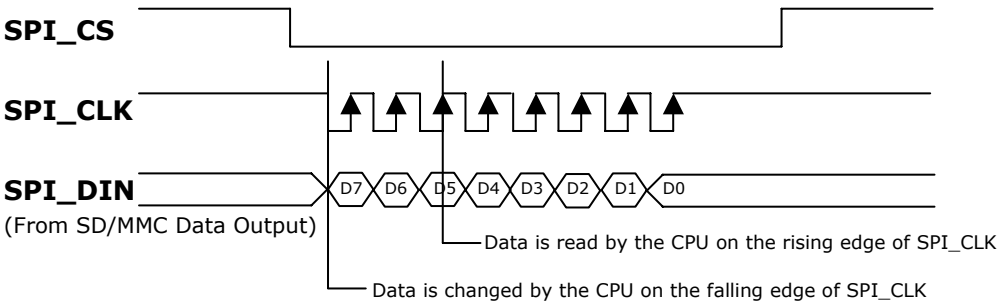
Additional Information

This function is used to read a number of bytes from the card to buffer memory. According to SD specification DOUT (MOSI) signal must be driven high during the data transfer, otherwise the SD card will not work properly.

Example

```
void FS_MMC_HW_X_Read(U8 Unit, U8 * pData, int NumBytes) {
  do {
    c = 0;
    bpos = 8; /* get 8 bits */
    do {
      SPI_CLR_CLK();
      c <<= 1;
      if (SPI_DATAIN()) {
        c |= 1;
      }
      SPI_SET_CLK();
    } while (--bpos);
    *pData++ = c;
  } while (--NumBytes);
}
```

Timing diagram for read access



6.5.6.8 FS_MMC_HW_X_Write()

Description

Sends a number of bytes to the card.

Prototype

```
void FS_MMC_HW_X_Write(U8         Unit,
                       const U8 * pData,
                       int        NumBytes);
```

Parameter	Meaning
Unit	Unit number (0-based).
pData	Pointer to a buffer that contains the data to be written to the card.
NumBytes	Number of bytes to write.

Table 6.149: FS_MMC_HW_X_Write() parameter list

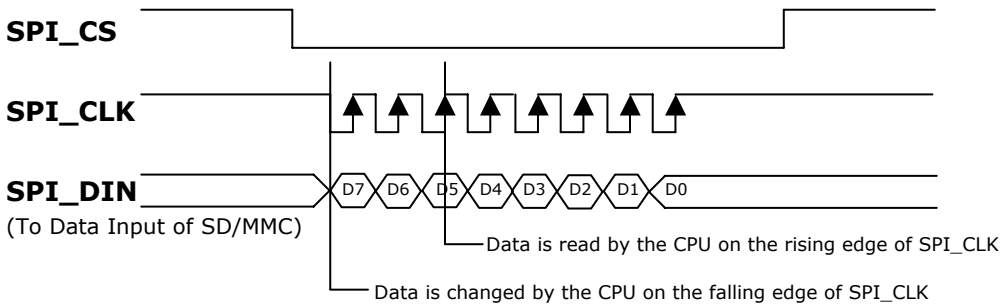
Additional Information

This function is used to send a number of bytes from a memory buffer to the card.

Example

```
void FS_MMC_HW_X_Write(U8 Unit, const U8 * pData, int NumBytes) {
  int i;
  U8 mask;
  U8 data;
  for (i = 0; i < NumBytes; i++) {
    data = pData[i];
    mask = 0x80;
    while (mask) {
      if (data & mask) {
        SPI_SET_DATAOUT();
      } else {
        SPI_CLR_DATAOUT();
      }
      SPI_CLR_CLK();
      SPI_DELAY();
      SPI_SET_CLK();
      SPI_DELAY();
      mask >>= 1;
    }
  }
  SPI_SET_DATAOUT(); /* default state of data line is high */
}
```

Timing diagram for write access



6.5.7 Hardware functions - Card mode

Routine	Explanation
Operation condition detection and adjusting	
FS_MMC_HW_X_SetMaxSpeed()	Sets the output clock speed. The value is represented in thousand cycles per second (kHz).
FS_MMC_HW_X_SetResponseTimeOut()	Sets the card host controller timeout value for receiving response from card.
FS_MMC_HW_X_SetReadDataTimeOut()	Sets the card host controller timeout value for receiving data from card.
FS_MMC_HW_X_SetHWBlockLen()	Sets the card host controller block size value for a block.
FS_MMC_HW_X_SetHWNumBlocks()	Tells the card host controller how many block will be transferred to or received from card.
Medium status functions	
FS_MMC_HW_X_IsWriteProtected()	Checks the status of the mechanical write protection of a card.
FS_MMC_HW_X_IsPresent()	Checks whether a card is present or not.
Data transfer functions	
FS_MMC_HW_X_GetResponse()	Retrieves the response after sending a command to the card.
FS_MMC_HW_X_ReadData()	Receives a number of bytes from the card.
FS_MMC_HW_X_SendCmd()	Sends and setups the controller to send a specific command to card.
FS_MMC_HW_X_WriteData()	Writes a number of block to the card.
Time functions	
FS_MMC_HW_X_Delay()	Waits for a specific time in ms.

Table 6.150: Card mode hardware functions

6.5.7.1 FS_MMC_HW_X_SetMaxSpeed()

Description

Sets the maximum output clock speed. If the hardware is unable to use this speed, a lower frequency can always be selected. The value is given in kHz.

Prototype

```
U16 FS_MMC_HW_X_SetMaxSpeed(U8  Unit,
                            U16 MaxFreq);
```

Parameter	Meaning
Unit	Unit number (0-based).
MaxFreq	Clock speed (kHz) between host and card.

Table 6.151: FS_MMC_HW_X_SetMaxSpeed() parameter list

Return value

Actual frequency in thousand cycles per second (kHz)
0 if the frequency could not be set.

Additional Information

Make sure your card host controller never generates a higher clock than MaxFreq specifies. You can always run the cards at lower or equal, but never on higher frequencies. The initial frequency must be 400kHz or less. You have to return the actual clock speed of your hardware interface, because emFile needs the actual frequency to calculate timeout values.

Example

```
U16 FS_MMC_HW_X_SetMaxSpeed(U8 Unit, U16 MaxFreq) {
  U32 Prediv;
  U32 Rate;

  if (Freq <= 400) {
    Prediv =  8;     //  HCLK / 8, where HCLK is 100MHz. -> SDClock = 12.5 MHz
    Rate   =  5;     //  Card clock frequency = SDClock / (1 << Rate) = 390kHz.
  } else {
    Prediv =  5;     // HCLK / 5, where HCLK is 100MHz, SDClock = 20 MHz
    Rate   =  0;     //  Card clock frequency = SDClock / (1 << Rate) = 20 MHz.
  }
  __SDMMC_PREDIV    = (1 << 5)        // Use Poll mode instead of DMA
                    | (1 << 4)        // Enable the Controller
                    | (Prediv & 0x0f); // Set the predivisor value
  __SDMMC_RATE      = Rate;            // Set rate value
  return Freq;
}
```


6.5.7.2 FS_MMC_HW_X_SetResponseTimeOut()

Description

Sets the timeout of card host controller for receiving response from card.

Prototype

```
void FS_MMC_HW_X_SetResponseTimeOut(U8  Unit,
                                    int Value);
```

Parameter	Meaning
Unit	Unit number (0-based).
Value	Number of output clock cycles to wait before a response timeout occurs.

Table 6.152: FS_MMC_HW_X_SetResponseTimeOut() parameter list

Example

```
void FS_MMC_HW_X_SetResponseTimeOut(U8 Unit, int Value) {
  __SDMMC_RES_TO    = Value;           // Set the timeout for Card Response
}
```


6.5.7.3 FS_MMC_HW_X_SetReadDataTimeOut()

Description

Sets the timeout of card host controller for receiving data from card.

Prototype

```
void FS_MMC_HW_X_SetReadDataTimeOut(U8  Unit,
                                    int Value);
```

Parameter	Meaning
Unit	Unit number (0-based).
Value	Number of card clock cycles to wait before a read data timeout occurs.

Table 6.153: FS_MMC_HW_X_SetReadDataTimeOut() parameter list

Example

```
void FS_MMC_HW_X_SetReadDataTimeOut(U8 Unit, int Value) {
  __SDMMC_READ_TO    = Value;          // Set the read timeout
}
```


6.5.7.4 FS_MMC_HW_X_SetHWBlockLen()

Description

Sets the card host controller block size value for a block.

Prototype

```
void FS_MMC_HW_X_SetHWBlockLen(U8  Unit,
                               U16 BlockSize);
```

Parameter	Meaning
Unit	Unit number (0-based).
BlockSize	Block size given in number of bytes.

Table 6.154: FS_MMC_HW_X_SetHWBlockLen() parameter list

Additional Information

Card host controller sends data to or receives data from the card in block chunks. This function typically sets the card host controller's block length register.

Example

```
void FS_MMC_HW_X_SetHWBlockLen(U8 Unit, U16 BlockSize) {
  __SDMMC_BLK_LEN  = BlockSize;
}
```


6.5.7.5 FS_MMC_HW_X_SetHWNumBlocks()

Description

Tells the card host controller how many block will be transferred to or received from card.

Prototype

```
void FS_MMC_HW_X_SetHWNumBlocks (U8  Unit,
                                 U16 NumBlocks);
```

Parameter	Meaning
Unit	Unit number (0-based).
NumBlocks	Number of blocks to be transferred.

Table 6.155: FS_MMC_HW_X_SetHWNumBlocks() parameter list

Additional Information

Before sending the command to read or write data from or to the card. This functions tells the card host controller, how many blocks need to be transferred/received.

Example

```
void FS_MMC_HW_X_SetHWNumBlocks(U8 Unit, U16 NumBlocks) {
  __SDMMC_NUM_BLK  = NumBlocks;
}
```


6.5.7.6 FS_MMC_HW_X_IsWriteProtected()

Description

Checks the status of the mechanical write protection of a card.

Prototype

```
int FS_MMC_HW_X_IsWriteProtected(U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0-based).

Table 6.156: FS_MMC_HW_X_IsWriteProtected() parameter list

Return value

== 0: If the card is not write protected.
== 1: Means that the card is write protected.

Additional Information

MultiMedia cards do not have mechanical write protection switches and should always return 0. If you are using SD cards, be aware that the mechanical switch does not really protect the card physically from being overwritten; it is the responsibility of the host to respect the status of that switch.

Example

```
int FS_MMC_HW_X_IsWriteProtected(U8 Unit) {
  return 0; /* Card slot has no write switch detector, return 0 */
}
```


6.5.7.7 FS_MMC_HW_X_IsPresent()

Description

Checks whether a card is present or not.

Prototype

```
int FS_MMC_HW_X_IsPresent(U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0-based).

Table 6.157: FS_MMC_HW_X_IsPresent() parameter list

Return value

Return value	Meaning
FS_MEDIA_STATE_UNKNOWN	State of the media is unknown.
FS_MEDIA_NOT_PRESENT	No card is present.
FS_MEDIA_IS_PRESENT	Card is present.

Table 6.158: FS_MMC_HW_X_IsPresent() - list of return values

Additional Information

Usually, a card slot provides a hardware signal that can be used for card presence determination. The sample code below is for a specific hardware that does not have such a signal. Therefore, the presence of a card is unknown and you have to return FS_MEDIA_STATE_UNKNOWN. Then emFile tries reading the card to figure out if a valid card is inserted into the slot.

Example

```
int FS_MMC_HW_X_IsPresent(U8 Unit) {
  __GPIO_PFDD &=  ~(1 << 5);  // Set PE.5 as input for card detect signal
  return ((__GPIO_PFD >> 5) & 1) ? FS_MEDIA_NOT_PRESENT : FS_MEDIA_IS_PRESENT;
}
```


6.5.7.8 FS_MMC_HW_X_GetResponse()

Description

Retrieves the card response to a sent command.

Prototype

```
int FS_MMC_HW_X_GetResponse(U8     Unit,
                            void * pBuffer,
                            U32    Size);
```

Parameter	Meaning
Unit	Unit number (0-based).
pBuffer	IN: --- OUT: response bytes.
NumBytes	Response size in bytes.

Table 6.159: FS_MMC_HW_X_GetResponse() parameter list

Return value

Return value	Meaning
FS_MMC_CARD_NO_ERROR	All data have been read successfully.
FS_MMC_CARD_RESPONSE_TIMEOUT	Card did not send the response in appropriate time.
FS_MMC_CARD_RESPONSE_CRC_ERROR	The received response failed the CRC check of card host controller.

Table 6.160: FS_MMC_HW_X_GetResponse() - list of return values

Additional information

The MMC/SD card standard describes the structure of a response in terms of bit units with bit 0 being the first transmitted over the CMD line. The following table shows you at which byte offsets the response should be stored into pBuffer:

Byte offset	Bit range (48-bit)	Bit range (136-bit)
0	47-40	135-128
1	39-32	127-120
2	31-24	119-112
3	23-16	111-104
4	15-8	103-96
5	7-0	95-88
6	-	87-80
7	-	79-72
8	-	71-64
9	-	63-56
10	-	55-48
11	-	47-40
12	-	39-32
13	-	31-24
14	-	23-16
15	-	15-8
16	-	7-0

Table 6.161: FS_MMC_HW_X_GetResponse() - mapping of response bits into buffer

Note: In the case of card

Note: Some card controllers forward only the payload of a response, i.e. the first and the last byte, which carry control and checking information, are discarded. In this case you need not set the missing bytes in the pBuffer.

Example

```
int FS_MMC_HW_X_GetResponse(U8 Unit, void * pBuffer, U32 Size) {
  U16 * pResponse;
  U32   Index;
  U32   Status;

  pResponse = (U16 *) pBuffer;
  // Wait for response
  while (1) {
    Status = __SDMMC_STATUS;
    if (Status & MMC_STATUS_CLOCK_DISABLED) {
      _StartMMCClock(Unit);
    }
    if (Status & MMC_STATUS_END_COMMAND_RESPONSE) {
      break;
    }
    if (Status & MMC_STATUS_RESPONSE_TIMEOUT) {
      return FS_MMC_CARD_RESPONSE_TIMEOUT;
    }
    if (Status & MMC_STATUS_RESPONSE_CRC_ERROR) {
      return FS_MMC_CARD_RESPONSE_CRC_ERROR;
    }
  }
  // Read the necessary number of response words from the response FIFO
  for (Index = 0; Index < (Size/ 2); Index++) {
    pResponse[Index] = __SDMMC_RES_FIFO;
  }
  return FS_MMC_CARD_NO_ERROR;
}
```


6.5.7.9 FS_MMC_HW_X_ReadData()

Description

Receives a number of bytes from the card.

Prototype

```
int FS_MMC_HW_X_ReadData(U8          Unit,
                         void      * pBuffer,
                         unsigned    NumBytes,
                         unsigned    NumBlocks);
```

Parameter	Meaning
Unit	Unit number (0-based).
pBuffer	IN: --- OUT: received data.
NumBytes	Number of bytes to receive.
NumBlocks	Number of blocks to receive.

Table 6.162: FS_MMC_HW_X_ReadData() parameter list

Return value

Return value	Meaning
FS_MMC_CARD_NO_ERROR	All data have been read successfully.
FS_MMC_CARD_READ_TIMEOUT	Card did not send the data in appropriate time.
FS_MMC_CARD_READ_CRC_ERROR	The received response failed the CRC check of card host controller.

Table 6.163: FS_MMC_HW_X_ReadData() - list of return values

Additional Information

This function is used to read the data is coming from MMC/SD card to the host controller through the DAT0 line or DAT[0:3] lines.

Example

```
int FS_MMC_HW_X_ReadData(U8 Unit, void * pBuffer, unsigned NumBytes,
                         unsigned NumBlocks) {
  U16 * pBuf = (U16 *)pBuffer;
  int i;
  do {
    i = 0;
    // Wait until transfer is complete
    while ((__SDMMC_STATUS & MMC_STATUS_FIFO_FULL) == 0);
    if (__SDMMC_STATUS & MMC_STATUS_READ_CRC_ERROR) {
      return FS_MMC_CARD_READ_CRC_ERROR;
    }
    if (__SDMMC_STATUS & MMC_STATUS_READDATA_TIMEOUT) {
      return FS_MMC_CARD_READ_TIMEOUT;
    }
    // Continue reading data until FIFO is empty
    while(((__SDMMC_STATUS & MMC_STATUS_FIFO_EMPTY) == 0) && (i < (NumBytes >> 1))) {
      // Any data in the FIFO
      if ((__SDMMC_STATUS & MMC_STATUS_FIFO_EMPTY) == 0) {
        *pBuf = __SDMMC_DATA_FIFO;
        pBuf++;
        i++;
      }
    }
  } while (--NumBlocks);
  return 0;
}
```


6.5.7.10 FS_MMC_HW_X_SendCmd()

Description

Sends a command to card.

Prototype

```
void    FS_MMC_HW_X_SendCmd (U8       Unit,
                             unsigned Cmd,
                             unsigned CmdFlags,
                             unsigned ResponseType,
                             U32      Arg);
```

Parameter	Meaning
Unit	Unit number (0-based).
Cmd	Command to be sent to the card. This is the command number from the SD card specification.
CmdFlags	Additional command flags, that are necessary for this command.
ResponseType	Specifies the response format that is expected after sending this command.
Arg	Argument sent with command.

Table 6.164: FS_MMC_HW_X_SendCmd() parameter list

Additional Information

This function should send the command specified by Cmd. Each command may have additional command flags. One or a combination of these is possible:

Command Flag	Meaning
FS_MMC_CMD_FLAG_DATATRANSFER	This flags tells the card controller, that the sent command initiate a data transfer.
FS_MMC_CMD_FLAG_WRITETRANSFER	This flags tells the card controller, that the sent command initiate a data transfer and will write to the card.
FS_MMC_CMD_FLAG_SETBUSY	The card may be in busy state after sending this command. The card host controller may wait after the card ready for next command.
FS_MMC_CMD_FLAG_INITIALIZE	The card host controller should send the initialization sequence to the card.
FS_MMC_CMD_FLAG_USE_SD4MODE	This tells the card host controller to use all four data lines DAT[0:3] rather than only DAT0 line. Note, that this command flag is only set when FS_MMC_SUPPORT_4BIT_MODE is set.
FS_MMC_CMD_FLAG_STOP_TRANS	The card host controller shall stop transferring data to the card.

Table 6.165: FS_MMC_HW_X_SendCmd() - list of possible command flags

Most of the commands require a response from the card. The type of the expected response can be one of the following:

Response Type	Meaning
FS_MMC_RESPONSE_FORMAT_NONE	No response is expected from card.

Table 6.166: FS_MMC_HW_X_SendCmd() - list of possible responses

Response Type	Meaning
FS_MMC_RESPONSE_FORMAT_R1	Response type 1 is expected from card. (48 Bit data stream is sent by card through the CMD line.)
FS_MMC_RESPONSE_FORMAT_R2	Response type 2 is expected from card. (136 Bit data stream is sent by card through the CMD line.)
FS_MMC_RESPONSE_FORMAT_R3	Response type 3 is expected from card. (48 Bit data stream is sent by card through the CMD line.)

Table 6.166: FS_MMC_HW_X_SendCmd() - list of possible responses

If the specified command expects a response, FS_MMC_HW_X_GetResponse() will be called after FS_MMC_HW_X_SendCmd().

Example

```
void FS_MMC_HW_X_SendCmd(U8 Unit, unsigned Cmd, unsigned CmdFlags,
                         unsigned ResponseType, U32 Arg) {
  U32 CmdCon;
  _StopMMCClock(Unit);
  CmdCon = ResponseType;
  if (CmdFlags & FS_MMC_CMD_FLAG_DATATRANSFER) { /* If data transfer */
    CmdCon |= (1 << 8)     /* Set big endian flag for data transfers
                              since this is how the data is in the 16-bit fifo */
           |  (1 << 2);    // Set DATA_EN
  }
  if (CmdFlags & FS_MMC_CMD_FLAG_WRITETRANSFER) {    /* Abort transfer ? */
    CmdCon |= (1 << 3);    // Set WRITE bit
  }
  if (CmdFlags & FS_MMC_CMD_FLAG_SETBUSY) {         /* Set busy ? */
    CmdCon |= (1 << 5);    // Set ABORT bit
  }
  if (CmdFlags & FS_MMC_CMD_FLAG_INITIALIZE) {    /* Init ? */
    CmdCon |= (1 << 6);    // Set ABORT bit
  }
  if (CmdFlags & FS_MMC_CMD_FLAG_USE_SD4MODE) {    /* 4 bit mode ? */
    CmdCon |= (1 << 7);    // Set WIDE bit
  }
  if (CmdFlags & FS_MMC_CMD_FLAG_STOP_TRANS) {    /* Abort transfer ? */
    CmdCon |= (1 << 13);    // Set ABORT bit
  }
  __SDMMC_CMD      = Cmd;
  __SDMMC_CMDCON   = CmdCon;
  __SDMMC_ARGUMENT = Arg;
  _StartMMCClock(Unit);
}
```


6.5.7.11 FS_MMC_HW_X_WriteData()

Description

Writes a number of blocks to the card.

Prototype

```
int FS_MMC_HW_X_WriteData(U8           Unit,
                          const void * pBuffer,
                          unsigned     NumBytes,
                          unsigned     NumBlocks);
```

Parameter	Meaning
Unit	Unit number (0-based).
pBuffer	IN: Data to send. OUT: ---
NumBytes	Number of bytes for each block to send.
NumBlocks	Number of blocks to send.

Table 6.167: FS_MMC_HW_X_WriteData() parameter list

Return value

Return Flag	Meaning
FS_MMC_CARD_NO_ERROR	All data have been sent successfully and card has programmed the data.
FS_MMC_CARD_WRITE_CRC_ERROR	During the data transfer to the card a CRC error occurred.

Table 6.168: FS_MMC_HW_X_WriteData() - list of return values

Additional Information

This function is used to write a specified number of blocks to the card. Each block is NumBytes long.

Example

```
int FS_MMC_HW_X_WriteData(U8 Unit, const void * pBuffer,
                          unsigned NumBytes, unsigned NumBlocks) {
  int         i;
  const U16 * pBuf;
  pBuf = (const U16 *)pBuffer;
  do {
    while((__SDMMC_STATUS & MMC_STATUS_FIFO_EMPTY) == 0);
    for (i = 0; i < (NumBytes >> 1); i++) {
      __SDMMC_DATA_FIFO =  *pBuf++;
    }
    _StartMMCClock(Unit);
    if (__SDMMC_STATUS & MMC_STATUS_WRITE_CRC_ERROR) {
      return FS_MMC_CARD_WRITE_CRC_ERROR;
    }
  } while (--NumBlocks);
  // Wait until transfer operation has ended
   while ((__SDMMC_STATUS & MMC_STATUS_DATA_TRANFER_DONE) == 0);
  // Wait until write operation has ended
  while ((__SDMMC_STATUS & MMC_STATUS_DATA_PROGRAM_DONE) == 0);
  return 0;
}
```


6.5.7.12 FS_MMC_HW_X_Delay()

Description

Waits for a specific time in ms.

Prototype

```
void FS_MMC_HW_X_Delay(int ms);
```

Parameter	Meaning
ms	Milliseconds to wait.

Table 6.169: FS_MMC_HW_X_Delay() parameter list

Additional Information

The delay specified is a minimum delay. The actual delay is permitted to be longer. This can be helpful when using an RTOS. Every RTOS has a delay API function, but the accuracy is typically 1 tick, which is 1 ms in most cases. Therefore, a delay of 1 tick is typically between 0 and 1 ms. To compensate for this, the equivalent of 1 tick (typically 1) should be added to the delay parameter before passing it to an RTOS delay function.

Example

```
void   FS_MMC_HW_X_Delay(int ms) {
  OS_Delay(ms + 1);   // Make sure we delay at least <ms> milliseconds
}
```


6.5.8 Hardware functions - Card mode for ATMEL devices

Note: `FS_MMC_CM_Driver4Atmel` is deprecated. Use `FS_MMC_CardMode_Driver` instead.

Routine	Description
Operation condition detection and adjusting	
`FS_MCI_HW_EnableClock()`	Enable/disable the master clock of the MCI module.
`FS_MCI_HW_EnableISR()`	Install the ISR handler of the MCI module.
`FS_MCI_HW_GetMCIInfo()`	Used to get the base address of the MCI module and which MCI slot.
`FS_MCI_HW_GetMClk()`	Returns the MCLK of an AT91SAM9x.
`FS_MCI_HW_Init()`	This function shall initialize all necessary hardware modules that depend on the MCI.
Medium status functions	
`FS_MCI_HW_IsCardPresent()`	Checks whether a card is present or not.
`FS_MCI_HW_IsCardWriteProtected()`	Checks the status of the mechanical write protection of a SD card.
Cache handling functions	
`FS_MCI_HW_CleanDCacheRange()`	Clean data cache range.
`FS_MCI_HW_InvalidateDCache()`	Invalidate data cache.

Table 6.170: Card mode for ATMEL hardware functions

6.5.8.1 FS_MCI_HW_EnableClock()

Description

Enables or disables the master clock of the MCI module. This is done by setting the appropriate bit in the PMC_PCER/PMC_PCDR register.

Prototype

```
void FS_MCI_HW_EnableClock(U8       Unit,
                           Unsigned OnOff);
```

Parameter	Description
Unit	Unit number (0-based).
OnOff	1: Enable the clock 0: Disable the clock

Table 6.171: FS_MCI_HW_EnableClock() parameter list

Example

```
/*********************************************************************
*
*       FS_MCI_HW_EnableClock
*
*  Function description:
*    This function shall enable or disable the master clock of the
*    MCI module. This is done by setting the appropiate bit in the
*    PMC_PCER/PMC_PCDR register.
*
*  Parameters:
*    Unit    - MCI Card unit that shall be used
*    OnOff   - 1 - Enable the clock
*              0 - Disable the clock
*
*/
void FS_MCI_HW_EnableClock(U8 Unit, unsigned OnOff) {
  if (OnOff) {
    WRITE_SFR_REG(PMC_BASE, PMC_PCER, (1 << MCI_ID)); // Enable the MCI
                                                      // peripheral clock.
  } else {
    WRITE_SFR_REG(PMC_BASE, PMC_PCDR, (1 << MCI_ID)); // Disable the MCI
                                                      // peripheral clock.
  }
}
```


6.5.8.2 FS_MCI_HW_EnableISR()

Description

Installs the ISR handler of the MCI module.

Prototype

```
void FS_MCI_HW_EnableISR(U8         Unit,
                         ISR_FUNC * pISRHandler);
```

Parameter	Description
Unit	Unit number (0-based).
pISRHandler	Pointer to the ISR handler that shall be installed.

Table 6.172: FS_MCI_HW_EnableISR() parameter list

Additional Information

The ISR handler is defined in the header file `MMC_MCI_HW.h`:

```
typedef void(ISR_FUNC)(void);
```

Example

```
/*********************************************************************
*
*       FS_MCI_HW_EnableISR
*
*  Function description:
*    The function is called during initialization to install the
*    ISR handler of the MCI module.
*
*  Parameters:
*    Unit         - MCI Card unit that shall be used
*    pISRHandler  - Pointer to the ISR handler that shall be installed.
*
*/
void FS_MCI_HW_EnableISR(U8 Unit, ISR_FUNC * pISRHandler) {
  //
  // Install interrupt service routine
  //
  OS_ARM_InstallISRHandler(MCI_ID, pISRHandler);
  OS_ARM_EnableISR(MCI_ID);
}
```


6.5.8.3 FS_MCI_HW_GetMCIInfo()

Description

Gets the base address of the MCI module and the information which slot is used.

Prototype

```
void FS_MCI_HW_GetMCIInfo(U8        Unit,
                          ISR_FUNC * pISRHandler);
```

Parameter	Description
Unit	Unit number (0-based).
pISRHandler	Pointer a MCI_INFO structure.

Table 6.173: FS_MCI_HW_GetMCIInfo() parameter list

Additional Information

The MCI_INFO structure is defined in the header file MMC_MCI_HW.h. It has the following elements:

```
typedef struct {
  U32 BaseAddr;
  U32 Mode;
} MCI_INFO;
```

Example

```
/*********************************************************************
*
*       FS_MCI_HW_GetMCIInfo
*
*  Function description:
*    This function is used to get the base address of the MCI module
*    and which MCI slot shall be used.
*
*  Parameters:
*    Unit    - MCI Card unit that shall be used
*    pInfo   - Pointer a MCI_INFO structure that shall be filled
*              by this function.
*
*/
void FS_MCI_HW_GetMCIInfo(U8 Unit, MCI_INFO * pInfo) {
  if (pInfo) {
    pInfo->BaseAddr = (U32)MCI_BASE_ADDR;
    pInfo->Mode     = MCI_SD_SLOTB;
  }
}
```


6.5.8.4 FS_MCI_HW_GetMClk()

Description

Returns the master clock in Hz.

Prototype

```
void FS_MCI_HW_GetMClk(U8 Unit);
```

Parameter	Description
Unit	Unit number (0-based).

Table 6.174: FS_MCI_HW_GetMClk() parameter list

Example

```
/*********************************************************************
*
*       FS_MCI_HW_GetMClk
*
*  Function description:
*    The internal MCLK of an AT91SAM9x that was configured shall be returned.
*
*  Parameters:
*    Unit    - MCI Card unit that shall be used
*
*  Return value:
*    The AT91 master clock (MCLK) given in Hz.
*
*/
U32 FS_MCI_HW_GetMClk(U8 Unit) {
  return MCLK;
}
```


6.5.8.5 FS_MCI_HW_Init()

Description

Initializes all necessary hardware modules.

Prototype

```
void FS_MCI_HW_Init(U8 Unit);
```

Parameter	Description
Unit	Unit number (0-based).

Table 6.175: FS_MCI_HW_Init() parameter list

Example

```
/*********************************************************************
*
*       FS_MCI_HW_Init
*
*  Function description:
*    This function shall initialize all necessary hardware modules
*    that depend on the MCI.
*    In normal cases PIO configuration needs to be done.
*
*  Parameters:
*    Unit    - MCI Card unit that shall be used
*
*/
void FS_MCI_HW_Init(U8 Unit) {
  // Configure SDcard pins
  _ConfigurePIO(_SDPins, COUNTOF(_SDPins));
}
```


6.5.8.6 FS_MCI_HW_IsCardPresent()

Description

Checks whether a card is present or not.

Prototype

```
char FS_MMC_HW_X_IsPresent(U8 Unit);
```

Parameter	Description
Unit	Unit number (0-based).

Table 6.176: FS_MCI_HW_IsCardPresent() parameter list

Return value

Return value	Description
FS_MEDIA_STATE_UNKNOWN	The card state is unknown.
FS_MEDIA_NOT_PRESENT	A card is not present.
FS_MEDIA_IS_PRESENT	A card is present.

Table 6.177: FS_MCI_HW_IsCardPresent() - list of return values

Additional Information

Usually, a card slot provides a hardware signal that can be used for card presence determination. The example code below is for a specific hardware that does not have such a signal. Therefore, the presence of a card is unknown and you have to return FS_MEDIA_STATE_UNKNOWN. Then emFile tries reading the card to figure out if a valid card is inserted into the slot.

Example

```
/*********************************************************************
*
*       FS_MCI_HW_IsCardPresent
*
*  Function description:
*    Returns whether a card is inserted or not.
*    When a card detect pin is not available. The function shall return
*    FS_MEDIA_STATE_UNKNOWN. The driver above will check, whether there
*    a valid card
*
*  Parameters:
*    Unit    - MCI Card unit that shall be used
*
*  Return value:
*    FS_MEDIA_STATE_UNKNOWN  - Card state is unknown, no card detect pin available
*    FS_MEDIA_NOT_PRESENT    - No Card is inserted in slot.
*    FS_MEDIA_IS_PRESENT     - Card is inserted in slot.
*/
int FS_MCI_HW_IsCardPresent(U8 Unit) {
  U8 r;
  r = FS_MEDIA_STATE_UNKNOWN;
  if (CARD_DETECT_PIN_AVAILABLE) {
    r = READ_SFR_REG(CARD_DETECT_PIN_PIO_BASE, PIO_PDSR)
        & (1 << CARD_DETECT_PIN) ? FS_MEDIA_NOT_PRESENT : FS_MEDIA_IS_PRESENT;
  }
  return r;
}
```


6.5.8.7 FS_MCI_HW_IsCardWriteProtected()

Description

Checks the status of the mechanical write protection of a SD card.

Prototype

```
char FS_MCI_HW_IsCardWriteProtected(U8 Unit);
```

Parameter	Description
Unit	Unit number (0-based).

Table 6.178: FS_MCI_HW_IsCardWriteProtected() parameter list

Return value

== 0: If the card is not write protected.
== 1: Means that the card is write protected.

Additional Information

MultiMedia cards do not have mechanical write protection switches and should always return 0. If you are using SD cards, be aware that the mechanical switch does not really protect the card physically from being overwritten; it is the responsibility of the host to respect the status of that switch.

Example

```
/*********************************************************************
*
*       FS_MCI_HW_IsCardWriteProtected
*
*  Function description:
*    Checks whether a card is write protected or not.
*
*  Parameters:
*    Unit    - MCI Card unit that shall be used
*
*  Return value:
*    0       - Card is not write protected.
*    1       - Card is     write protected.
*
*/
U8 FS_MCI_HW_IsCardWriteProtected(U8 Unit) {
  U8 r;
  r = 0;
  if (WRITE_PROTECT_PIN_AVAILABLE) {
    r = READ_SFR_REG(WRITE_PROTECT_PIN_PIO_BASE, PIO_PDSR)
      & (1 << WRITE_PROTECT_PIN) ? 0 : 1;
  }
  return r;
}
```


6.5.8.8 FS_MCI_HW_CleanDCacheRange()

Description

Used to clean a range in the data cache memory to ensure that the data is written from the data cache into the memory. This function can be empty if data cache is not used.

Prototype

```
void FS_MCI_HW_CleanDCacheRange(void *   p,
                                unsigned NumBytes);
```

Parameter	Description
p	Pointer to the region that shall be flushed from cache.
NumBytes	Number of bytes to flush.

Table 6.179: FS_MCI_HW_CleanDCacheRange() parameter list

Example

```
/*********************************************************************
*
*       FS_MCI_HW_CleanDCacheRange
*
*  Parameters:
*    p          - Pointer to the region that shall be flushed from cache.
*    NumBytes   - Number of bytes to flush
*
*/
void FS_MCI_HW_CleanDCacheRange(void * p, unsigned NumBytes) {
  OS_ARM_DCACHE_CleanRange(p,  NumBytes);
}
```


6.5.8.9 FS_MCI_HW_InvalidateDCache()

Description

Used to invalidate a memory area in the data cache. Invalidating means, mark all entries in the specified area as invalid. Invalidation forces re-reading the data from memory into the cache, when the specified area is accessed again. This function can be empty if data cache is not used.

Prototype

```
void FS_MCI_HW_InvalidateDCache(void *   p,
                                unsigned NumBytes);
```

Parameter	Description
p	Pointer to the region that shall be flushed from cache.
NumBytes	Number of bytes to flush.

Table 6.180: FS_MCI_HW_InvalidateDCache() parameter list

Example

```
/*********************************************************************
*
*       FS_MCI_HW_InvalidateDCache
*
*  Parameters:
*    p          - Pointer to the buffer that shall be invalidated in cache.
*    NumBytes   - Number of bytes to invalidate
*
*/
void FS_MCI_HW_InvalidateDCache(void * p, unsigned NumBytes) {
  OS_ARM_DCACHE_InvalidateRange(p, NumBytes);
}
```


6.5.9 Additional information

For more technical details about MMC and SD cards, check the documents and specifications available on the following internet web pages:

www.jedec.org

www.sdcard.org

6.5.10 Additional driver functions

6.5.10.1 FS_MMC_CM_Allow4bitMode()

Description

Allows the driver to use 4 data lines (4-bit mode) when exchanging data with SD and eMMC cards.

Prototype

```
void FS_MMC_CM_Allow4bitMode(U8 Unit, U8 OnOff);
```

Parameter	Description
Unit	Unit number (0-based).
OnOff	1 enable 4-bit mode. 0 disable 4-bit mode.

Table 6.181: FS_MMC_CM_Allow4bitMode() parameter list

Additional information

This function shall only be used when configuring the driver in `FS_X_AddDevices()`. Refer to *FS_X_AddDevices()* on page 472 for more information. The 4-bit mode is disabled by default.

6.5.10.2 FS_MMC_CM_Allow8bitMode()

Description

Allows the driver to use 8 data lines when exchanging data with eMMC cards.

Prototype

```
void FS_MMC_CM_Allow8bitMode(U8 Unit, U8 OnOff);
```

Parameter	Description
Unit	Unit number (0-based).
OnOff	1 enable 8-bit data transfer. 0 disable 8-bit data transfer.

Table 6.182: FS_MMC_CM_Allow8bitMode() parameter list

Additional information

This function shall only be used when configuring the driver in `FS_X_AddDevices()`. Refer to *FS_X_AddDevices()* on page 472 for more information. The data transfer over 8 data lines is disabled by default. It has no effect when used with SD cards.

6.5.10.3 FS_MMC_CM_AllowHighSpeedMode()

Description

Allows the driver to use the highest communication speed the card supports.

Prototype

```
void FS_MMC_CM_AllowHighSpeedMode(U8 Unit, U8 OnOff);
```

Parameter	Description
Unit	Unit number (0-based).
OnOff	1 enable high speed mode. 0 disable high speed mode.

Table 6.183: FS_MMC_CM_AllowHighSpeedMode() parameter list

Additional information

This function shall only be used when configuring the driver in `FS_X_AddDevices()`. Refer to *FS_X_AddDevices()* on page 472 for more information. The high speed mode is disabled by default. The maximum communication speed is typically 50MHz for SD cards and 52MHz for eMMC cards.

6.5.10.4 FS_MMC_CM_GetCardId()

Description

Reads the contents of the CID register.

Prototype

```
int FS_MMC_CM_GetCardId(U8 Unit, MMC_CARD_ID * pCardId);
```

Parameter	Description
Unit	Unit number (0-based).
pCardId	IN: --- OUT: CID register contents

Table 6.184: FS_MMC_CM_GetCardId() parameter list

Additional information

The CID (Card Identification) register stores information which can be used to uniquely identify the card such as the serial number, product name and manufacturer ID.

Example

The following example shows how to read and interpret the contents of the CID register.

```
void SampleGetCardID(void) {
  U8     ManId;
  char   acOEMId[2 + 1];
  char   acProductName[5 + 1];
  U8     ProductRevMajor;
  U8     ProductRevMinor;
  U32    ProductSN;
  U8     MfgMonth;
  U16    MfgYear;
  U16    MfgDate;
  U8   * p;
  MMC_CARD_ID CardId;

  FS_MMC_CM_GetCardId(0, &CardId);
  p = CardId.aData;
  ++p;      // Skip the start of message.
  ManId = *p++;
  strncpy(acOEMId, (char *)p, 2);
  acOEMId[2] = '\0';
  p += 2;
  strncpy(acProductName, (char *)p, 5);
  acProductName[5] = '\0';
  p += 5;
  ProductRevMajor  = *p >> 4;
  ProductRevMinor  = *p++ & 0xF;
  ProductSN        = (U32)*p++ << 24;
```



```
  ProductSN       |= (U32)*p++ << 16;
  ProductSN       |= (U32)*p++ << 8;
  ProductSN       |= (U32)*p++;
  MfgDate          = (U16)*p++;
  MfgDate         |= (U16)*p++;
  MfgMonth         = (U8)(MfgDate & 0xF);
  MfgYear          = MfgDate >> 4;
  printf("SD card info:\n");
  printf("  Manufacturer Id:       0x%02x\n"
         "  OEM/Application Id:    %s\n"
         "  Product name:          %s\n"
         "  Product revision:      %lu.%lu\n"
         "  Product serial number: 0x%08lx\n"
         "  Manufacturing date:    %lu-%lu\n", ManId,
                                                acOEMId,
                                                acProductName,
                                                (U32)ProductRevMajor,
                                                (U32)ProductRevMinor,
                                                ProductSN,
                                                (U32)MfgMonth, (U32)MfgYear);
}
```


6.5.11 Performance and resource usage

6.5.11.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the MMC/SD driver have been measured on a system as follows: ARM7, IAR Embedded workbench V4.41A, Thumb mode, Size optimization.

Module	ROM [Kbytes]
emFile SD card SPI mode driver	2.8
emFile SD card mode driver	3.9

6.5.11.2 Static RAM usage

Static RAM usage is the amount of RAM required by the driver for variables inside of the driver. The number of bytes can be seen in a compiler list file

Static RAM usage of the SD card driver in SPI mode: 12 bytes
Static RAM usage of the SD card driver in card mode: 12 bytes

6.5.11.3 Performance

These performance measurements are in no way complete, but they give an approximation of the length of time required for common operations on various targets. The tests were performed as described in *Performance* on page 502.

All values are in Mbytes/sec.

Device	CPU speed	Medium	W	R
Atmel AT91SAM7S	48 MHz	MMC/SD using SPI with 24MHz	2.3	2.3
Atmel AT91SAM9261	178 MHz	MMC/SD driver using SPI with 24 MHz.	2.3	2.5
Atmel AT91SAM9263	200 MHz	MMC/SD card mode driver using card controller with 25 MHz.	10.0	9.3
LogicPD LH79520	51 MHz	MMC using SPI with 12MHz	0.5	1.3
NXP LPC2478	57 MHz	MMC/SD card mode driver using card controller with 25 MHz.	2.4	3.1
NXP LPC3250	208 MHz	MMC/SD card mode driver using card controller with 25 MHz.	3.9	8.4
Toshiba TMPA910	192 MHz	MMC/SD card mode driver using card controller with 25 MHz.	3.9	8.4

Table 6.185: Performance values for sample configurations

6.5.12 Troubleshooting

If the driver test fails or if the card cannot be accessed at all, please follow the trouble shooting guidelines below.

6.5.12.1 SPI mode troubleshooting guide

Verify SPI configuration

If an SPI is used, you should verify that it is set up as follows:

- 8 bits per transfer
- Most significant bit first
- Data changes on falling edge
- Data is sampled on rising edge.

Verify signals during initialization of the card

The oscilloscope has been set up as follows:

Color	Description
RED	MOSI - Master Out Slave In (Pin 2)
PURPLE	MISO - Master In Slave Out (Pin 7)
GREEN	CLK - Clock (Pin 5)
YELLOW	CS - Chip Select (Pin 1)

Table 6.186: Screenshot descriptions

Trigger: Single, falling edge of CS

To check if your implementation of the hardware layer works correct, compare your output of the relevant lines (SCLK, CS, MISO, MOSI) with the correct output which is shown in the following screenshots. The output of your card should be similar.

In the example, MISO has a pull-up and a pull-down of equal value. This means that the MISO signal level is at 50% (1.65V) when the output of the card is inactive. On other target hardware, the inactive level can be low (in case a pull-down is used) or high (if a pull-up is used).

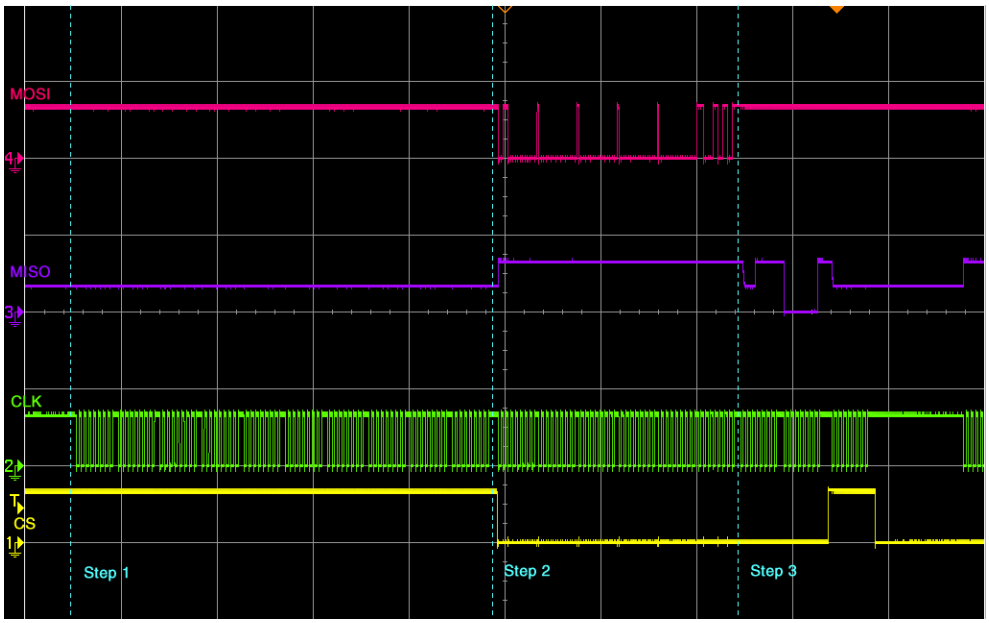
Initial communication sequence

The initial communication sequence consists of the following three parts:

1. Outputs 10 dummy bytes with CS disabled, MOSI = 1.
2. Sets CS low and send a 6-byte command (`GO_IDLE_STATE` command).
3. Receives two bytes, sets CS high and outputs 1 dummy byte with CS disabled, MOSI = 1.

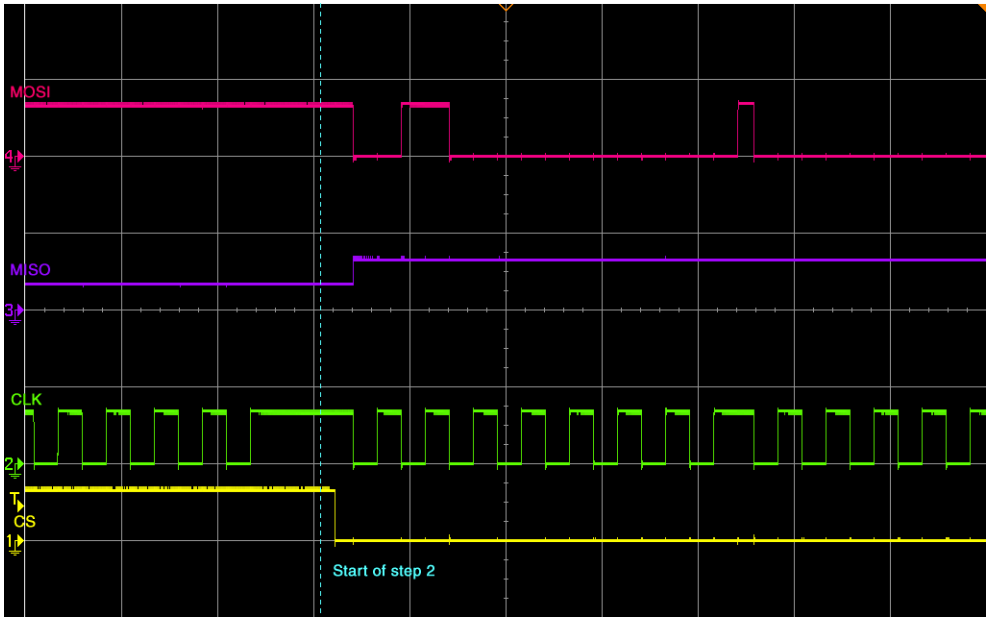
Overview

The screenshot shows the data flow of a correct initialization. It has been captured with an oscilloscope.



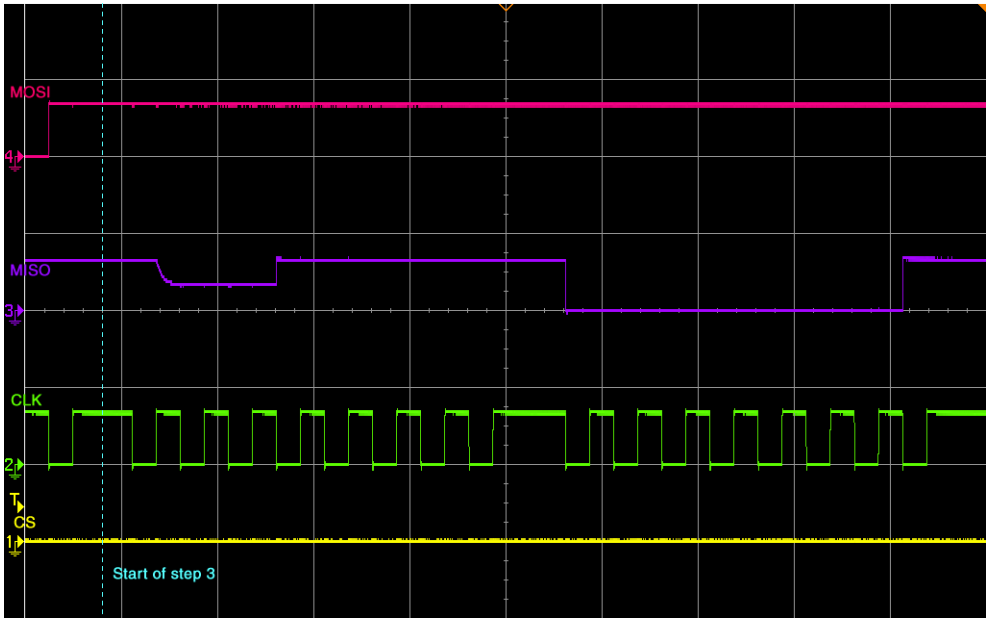
Verify command transfer (Step 2)

After sending 8 dummy bytes to the card, CS is activated and the `GO_IDLE_STATE` command is sent to the card. The first byte is 0x40 or b01000000. You can see (and should verify that MOSI changes on the falling edge of CLK. The `GO_IDLE_STATE` command is the reset command. It sets the card into idle state regardless of the current card state.



Check output of card (Step 3)

The card responses to the command with two bytes. The SD Card Association defines that the first byte of the response should always be ignored. The second byte is the answer from the card. The answer to `GO_IDLE_STATE` command should be 0x01. This means that the card is in idle state.

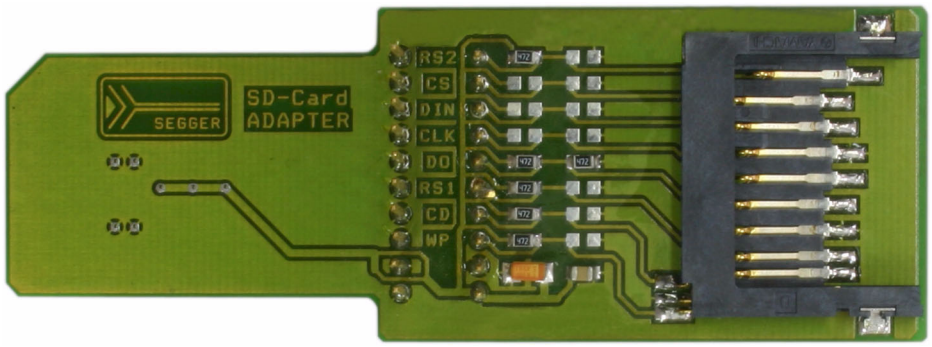


If your card does not return 0x01, check your initialization sequence. After the command sequence CS has to be deselected.

6.5.13 Test hardware

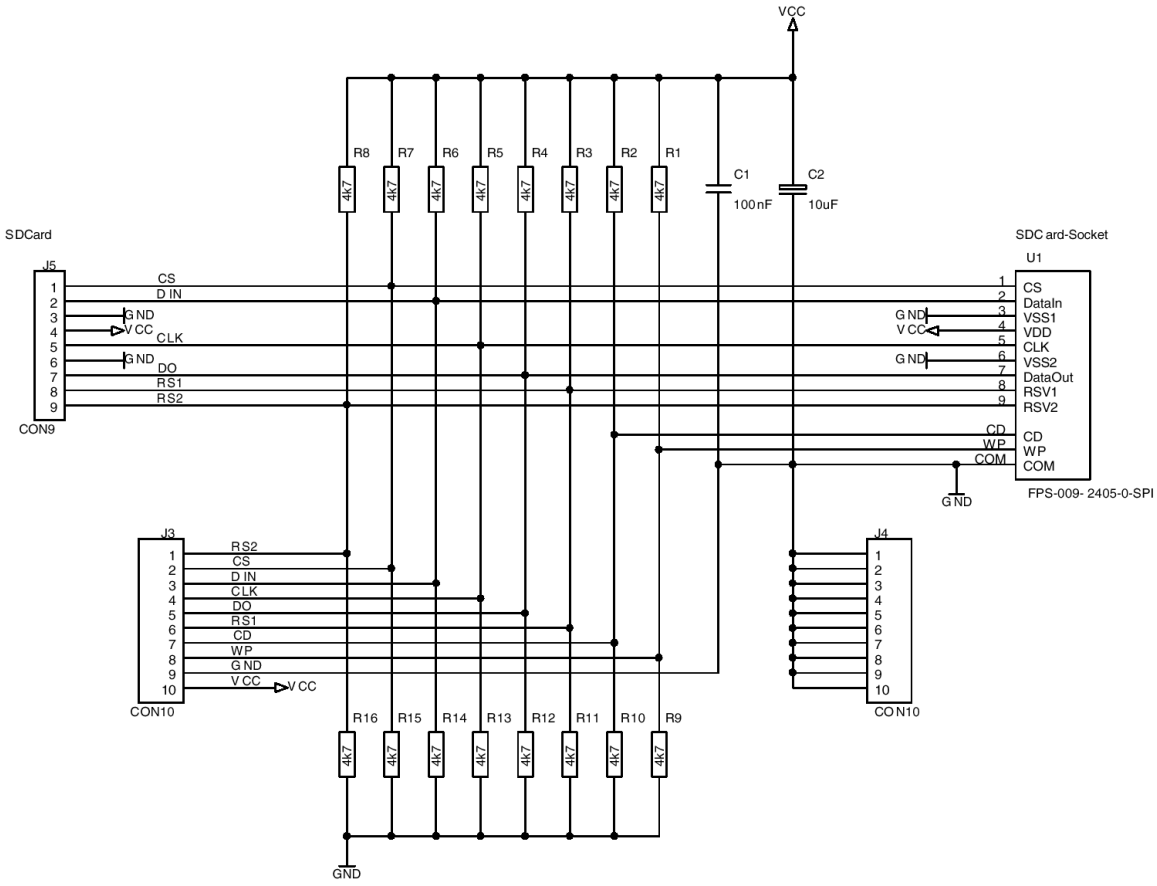
On some evaluation boards the pins required for measuring are not accessible, so that an oscilloscope or logic analyzer cannot capture the outputs. An adapter which can be inserted between the card slot and the card, is the best solution in those situations.

An example adapter is shown below and is available from Segger.



Adapter schematics

Use the schematic below to build an compatible adapter.



6.6 CompactFlash card and IDE driver

emFile supports the use of CompactFlash & IDE devices. An optional generic driver for CompactFlash & IDE devices is available.

To use the driver with your specific hardware, you will have to provide basic I/O functions for accessing the ATA I/O registers. This section describes all these routines.

6.6.1 Supported Hardware

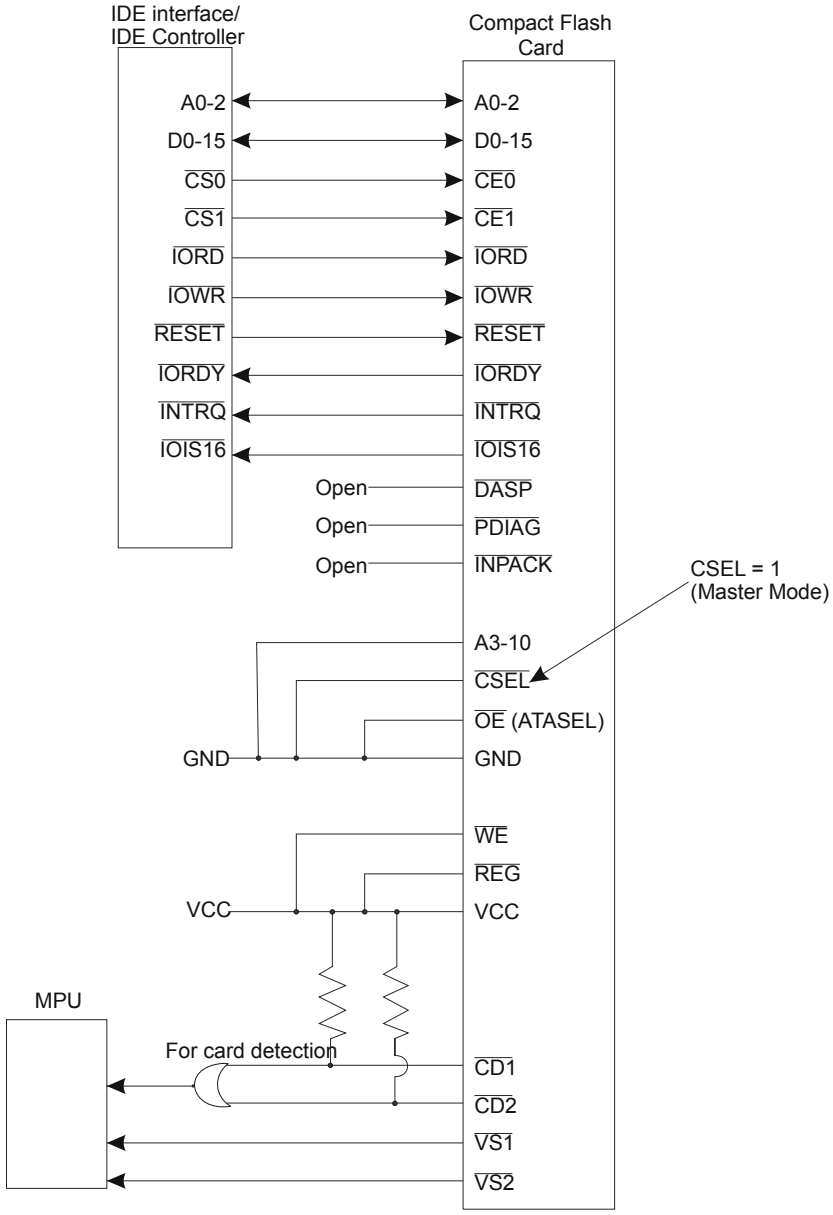
emFile's CompactFlash & IDE device driver can be used to access most ATA HD drives or CompactFlash storage cards also known as CF using true IDE or Memory card mode.

True IDE mode pin functions

Signal name	Dir	Pin	Description
A2-A0	I	18, 19, 20	Only A[2:0] are used to select one of eight registers in the Task File, the remaining address lines should be grounded by the host.
PDIAG	I/O	46	This input / output is the Pass Diagnostic signal in the Master / Slave handshake protocol.
DASP	I/O	45	This input/output is the Disk Active/Slave Present signal in the Master/Slave handshake protocol.
$\overline{CD1}$, $\overline{CD2}$	O	26, 25	These Card Detect pins are connected to ground on the CompactFlash Storage Card or CF+ Card. They are used by the host to determine that the CompactFlash Storage Card or CF+ Card is fully inserted into its socket.
$\overline{CS0}$, $\overline{CS1}$	I	7, 32	CS0 is the chip select for the task file registers while CS1 is used to select the Alternate Status Register and the Device Control Register.
CSEL	I	39	This internally pulled up signal is used to configure this device as a Master or a Slave when configured in True IDE Mode. When this pin is grounded, the device is configured as a Master. When the pin is open, the device is configured as a Slave.
D15 - D00	I/O	27 - 31 47 - 49 2 - 6 21 - 23	All Task File operations occur in byte mode on the low order bus D00-D07 while all data transfers are 16 bit using D00-D15.
GND	--	1, 5	Ground.
IORD	I	34	This is an I/O Read strobe generated by the host. This signal gates I/O data onto the bus from the CompactFlash Storage Card or CF+ Card when the card is configured to use the I/O interface.
IOWR	I	35	I/O Write strobe pulse is used to clock I/O data on the Card Data bus into the CompactFlash Storage Card or CF+ Card controller registers when the CompactFlash Storage Card or CF+ Card is configured to use the I/O interface. The clocking will occur on negative to positive edge of the signal (trailing edge).
$\overline{OE}$ (ATA SEL)	I	9	To enable True IDE Mode this input should be grounded by the host.
INTRQ	O	37	Signal is the active high interrupt request to the host.
REG	I	44	This input signal is not used and should be connected to VCC by the host.
RESET	I	41	This input pin is the active low hardware reset from the host.
VCC	--	13, 38	+5V, +3.3V power.
$\overline{VS1}$, $\overline{VS2}$	O	33, 4	Voltage Sense Signals. -VS1 is grounded so that the CompactFlash Storage Card or CF+ Card CIS can be read at 3.3 volts and -VS2 is reserved by PCMCIA for a secondary voltage.
IORDY	O	42	This output signal may be used as IORDY.
WE	I	36	This input signal is not used and should be connected to VCC by the host.
IOIS16	O	24	This output signal is asserted low when the device is expecting a word data transfer cycle.

Table 6.187: True IDE pin functions

Sample block schematic



Memory card mode pin functions

Signal name	Dir	Pin	Description
A10 - A0	I	8, 10, 11, 12, 14, 15, 16, 17, 18, 19, 20	These address lines along with the -REG signal are used to select the following: the I/O port address registers within the CompactFlash Storage Card or CF+ Card, the memory mapped port address registers within the CompactFlash Storage Card or CF+ Card, a byte in the card's information structure and its configuration control and status registers.
BVD1	I/O	46	This signal is asserted high, as BVD1 is not supported.
BVD2	I/O	45	This signal is asserted high, as BVD2 is not supported.

Table 6.188: Pin functions in memory card mode

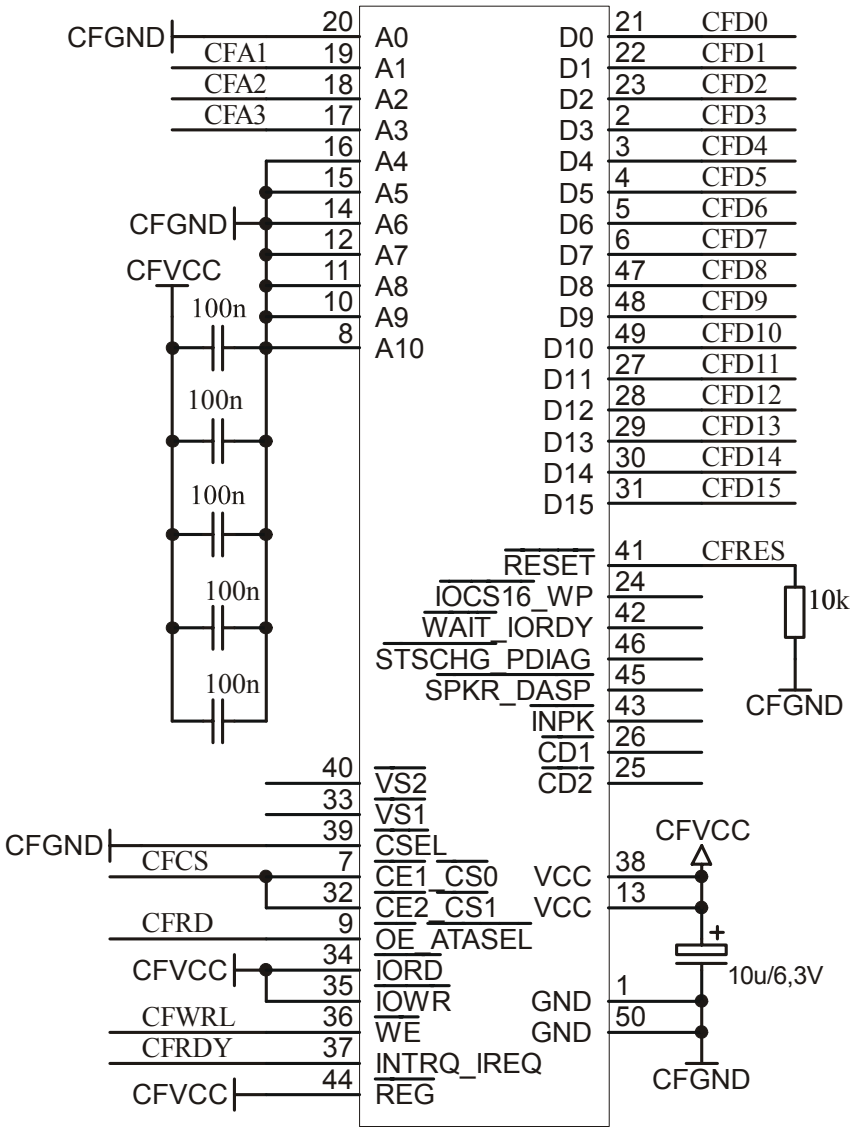
Signal name	Dir	Pin	Description
$\overline{CD1}$, $\overline{CD2}$	O	26, 25	These Card Detect pins are connected to ground on the CompactFlash Storage Card or CF+ Card. They are used by the host to determine that the CompactFlash Storage Card or CF+ Card is fully inserted into its socket.
$\overline{CE1}$, $\overline{CE2}$	I	7, 32	These input signals are used both to select the card and to indicate to the card whether a byte or a word operation is being performed. -CE2 always accesses the odd byte of the word. We recommend connecting these pins together.
CSEL	I	39	This signal is not used for this mode, but should be grounded by the host.
D15 - D00	I/O	27 - 31 47 - 49 2 - 6 21 - 23	These lines carry the Data, Commands and Status information between the host and the controller. D00 is the LSB of the Even Byte of the Word. D08 is the LSB of the Odd Byte of the Word.
GND	--	1, 5	Ground.
INPACK	O	43	This signal is not used in this mode.
IORD	I	34	This signal is not used in this mode.
IOWR	I	35	This signal is not used in this mode.
$\overline{OE}$ (ATA SEL)	I	9	This is an Output Enable strobe generated by the host interface. It is used to read data from the CompactFlash Storage Card or CF+ Card in Memory Mode and to read the CIS and configuration registers.
READY	O	37	In Memory Mode, this signal is set high when the CompactFlash Storage Card or CF+ Card is ready to accept a new data transfer operation and is held low when the card is busy. At power up and at Reset, the READY signal is held low (busy) until the CompactFlash Storage Card or CF+ Card has completed its power up or reset function. No access of any type should be made to the CompactFlash Storage Card or CF+ Card during this time.Note, however, that when a card is powered up and used with +RESET continuously disconnected or asserted, the reset function of this pin is disabled and consequently the continuous assertion of +RESET will not cause the READY signal to remain continuously in the busy state.
REG	I	44	This signal is used during Memory Cycles to distinguish between Common Memory and Register (Attribute) Memory accesses. High for Common Memory, Low for Attribute Memory. To use it with emFile, this signal should be high.
RESET	I	41	When the pin is high, this signal Resets the CompactFlash Storage Card or CF+ Card. The CompactFlash Storage Card or CF+ Card is reset only at power up if this pin is left high or open from power-up.
VCC	--	13, 38	+5 V, +3.3 V power.
$\overline{VS1}$, $\overline{VS2}$	O	33, 4	Voltage Sense Signals. -VS1 is grounded so that the CompactFlash Storage Card or CF+ Card CIS can be read at 3.3 volts and -VS2 is reserved by PCMCIA for a secondary voltage.

Table 6.188: Pin functions in memory card mode (Continued)

Signal name	Dir	Pin	Description
WAIT	O	42	The -WAIT signal is driven low by the CompactFlash Storage Card or CF+ Card to signal the host to delay completion of a memory or I/O cycle that is in progress.
WE	I	36	This is a signal driven by the host and used for strobing memory write data to the registers of the CompactFlash Storage Card or CF+ Card when the card is configured in the memory interface mode.
WP	O	24	The CompactFlash Storage Card or CF+ Card does not have a write protect switch. This signal is held low after the completion of the reset initialization sequence.

Table 6.188: Pin functions in memory card mode (Continued)

Sample block schematic



6.6.2 Theory of operation

6.6.2.1 CompactFlash

CompactFlash is a mechanically small, removable mass storage device. The CompactFlash Storage Card contains a single chip controller and flash memory module(s) in a matchbox-sized package with a 50-pin connector consisting of two rows of 25 female contacts each on 50 mil (1.27 mm) centers. The controller interfaces with a host system allowing data to be written to and read from the flash memory module(s).

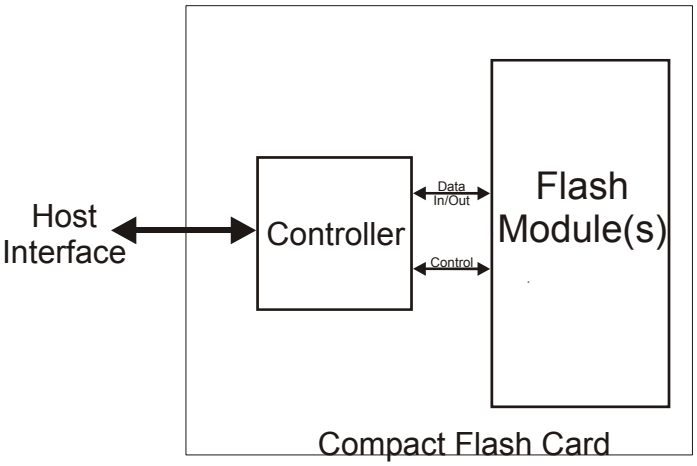
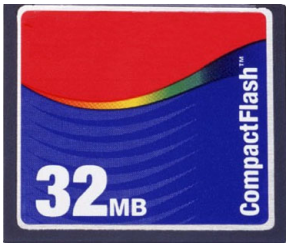


Figure 6.1: CompactFlash schematic

There are two different Compact Flash Types, namely CF Type I and CF Type II.
The only difference between CF Type I and CF Type II cards is the card thickness. CF Type I is 3.3 mm thick and CF Type II cards are 5mm thick. A CF Type I card will operate in a CF Type I or CF Type II slot. A CF Type II card will only fit in a CF Type II slot. The electrical interfaces are identical. CompactFlash is available in both CF Type I and CF Type II cards, though predominantly in CF Type I cards. The Microdrive is a CF Type II card. Most CF I/O cards are CF Type I, but there are some CF Type II I/O cards.



CompactFlash cards are designed with flash technology, a nonvolatile storage solution that does not require a battery to retain data indefinitely.
The CompactFlash card specification version 2.0 supports data rates up to 16MB/sec and capacities up to 137GB.
CF cards consume only five percent of the power required by small disk drives.

CompactFlash cards support both 3.3V and 5V operation and can be interchanged between 3.3V and 5V systems. This means that any CF card can operate at either voltage. Other small form factor flash cards may be available to operate at 3.3V or 5V, but any single card can operate at only one of the voltages.
CF+ data storage cards are also available using magnetic disk (IBM Microdrive).

Modes of operation (interface modes)

Compact Flash cards can operate in three modes:

- Memory card mode
- I/O Card mode
- True IDE mode

Supported modes of operation (interface modes)

Currently, TRUE IDE and MEMORY CARD mode are supported.

Memory card mode pin functions

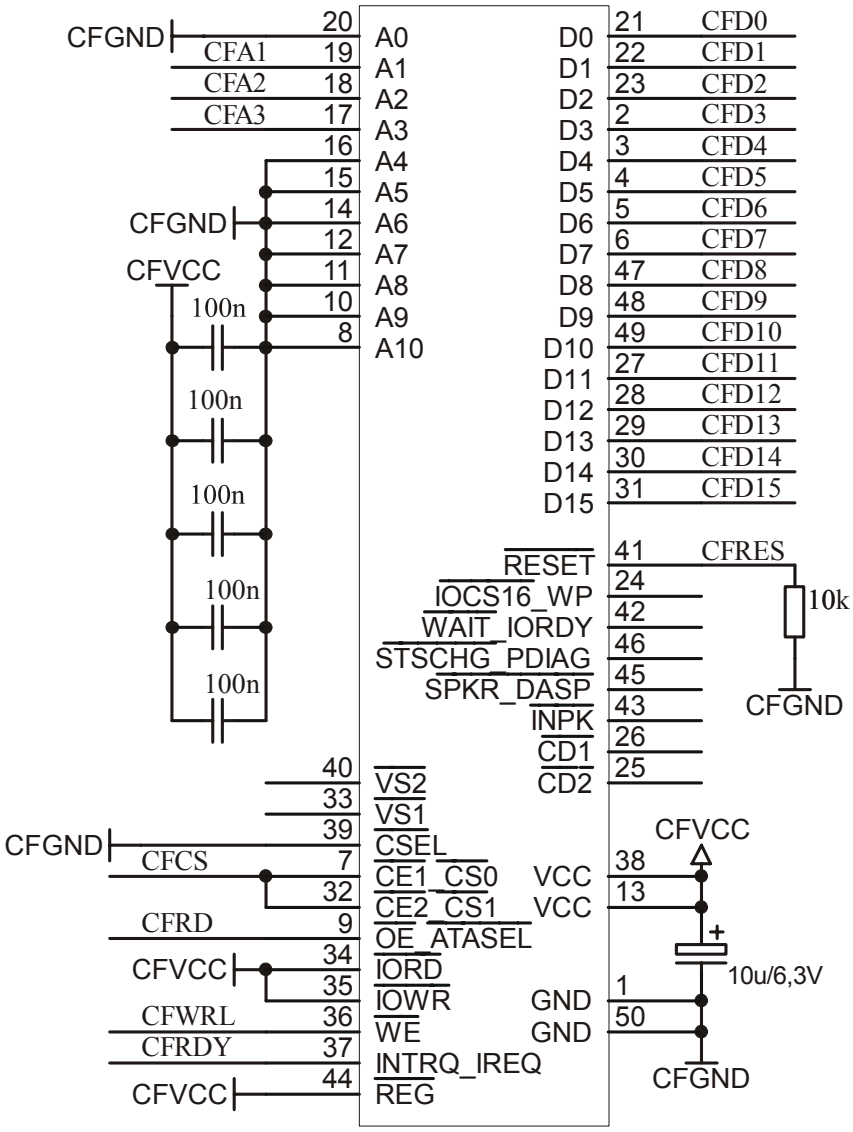
Signal name	Dir	Pin	Description
A10 - A0	I	8, 10, 11, 12, 14, 15, 16, 17, 18, 19, 20	These address lines along with the -REG signal are used to select the following: the I/O port address registers within the CompactFlash Storage Card or CF+ Card, the memory mapped port address registers within the CompactFlash Storage Card or CF+ Card, a byte in the card's information structure and its configuration control and status registers.
BVD1	I/O	46	This signal is asserted high, as BVD1 is not supported.
BVD2	I/O	45	This signal is asserted high, as BVD2 is not supported.
$\overline{CD1}$, $\overline{CD2}$	O	26, 25	These Card Detect pins are connected to ground on the CompactFlash Storage Card or CF+ Card. They are used by the host to determine that the CompactFlash Storage Card or CF+ Card is fully inserted into its socket.
$\overline{CE1}$, $\overline{CE2}$	I	7, 32	These input signals are used both to select the card and to indicate to the card whether a byte or a word operation is being performed. -CE2 always accesses the odd byte of the word. We recommend connecting these pins together.
CSEL	I	39	This signal is not used for this mode, but should be grounded by the host.
D15 - D00	I/O	27 - 31 47 - 49 2 - 6 21 - 23	These lines carry the Data, Commands and Status information between the host and the controller. D00 is the LSB of the Even Byte of the Word. D08 is the LSB of the Odd Byte of the Word.
GND	--	1, 5	Ground.
INPACK	O	43	This signal is not used in this mode.
IORD	I	34	This signal is not used in this mode.
IOWR	I	35	This signal is not used in this mode.
$\overline{OE}$ (ATA SEL)	I	9	This is an Output Enable strobe generated by the host interface. It is used to read data from the CompactFlash Storage Card or CF+ Card in Memory Mode and to read the CIS and configuration registers.
READY	O	37	In Memory Mode, this signal is set high when the CompactFlash Storage Card or CF+ Card is ready to accept a new data transfer operation and is held low when the card is busy. At power up and at Reset, the READY signal is held low (busy) until the CompactFlash Storage Card or CF+ Card has completed its power up or reset function. No access of any type should be made to the CompactFlash Storage Card or CF+ Card during this time.Note, however, that when a card is powered up and used with +RESET continuously disconnected or asserted, the reset function of this pin is disabled and consequently the continuous assertion of +RESET will not cause the READY signal to remain continuously in the busy state.

Table 6.189: Pin functions in memory card mode

Signal name	Dir	Pin	Description
REG	I	44	This signal is used during Memory Cycles to distinguish between Common Memory and Register (Attribute) Memory accesses. High for Common Memory, Low for Attribute Memory. To use it with emFile, this signal should be high.
RESET	I	41	When the pin is high, this signal Resets the CompactFlash Storage Card or CF+ Card. The CompactFlash Storage Card or CF+ Card is reset only at power up if this pin is left high or open from power-up.
VCC	--	13, 38	+5 V, +3.3 V power.
$\overline{VS1}$, $\overline{VS2}$	O	33, 4	Voltage Sense Signals. -VS1 is grounded so that the CompactFlash Storage Card or CF+ Card CIS can be read at 3.3 volts and -VS2 is reserved by PCMCIA for a secondary voltage.
WAIT	O	42	The -WAIT signal is driven low by the CompactFlash Storage Card or CF+ Card to signal the host to delay completion of a memory or I/O cycle that is in progress.
WE	I	36	This is a signal driven by the host and used for strobing memory write data to the registers of the CompactFlash Storage Card or CF+ Card when the card is configured in the memory interface mode.
WP	O	24	The CompactFlash Storage Card or CF+ Card does not have a write protect switch. This signal is held low after the completion of the reset initialization sequence.

Table 6.189: Pin functions in memory card mode (Continued)

Sample block schematic



6.6.2.2 IDE (ATA) Drives

Just like Compact Flash cards, ATA drives have a built-in controller to drive and control the mechanical hardware in a drive. Actually there are two types of connecting ATA drives. 5.25 and 3.5 inch drives are using a 40 pin male interface to connect to an IDE controller. 2.5 and 1.8 inch drives, mostly used in Notebooks and embedded systems, have a 50 pin male interface.

Modes of operation (interface modes)

ATA drives can operate in a variety of different modes:

- PIO (Programmed I/O)
- Multiword DMA
- Ultra DMA

Supported modes of operation (interface modes)

Currently, only PIO mode through TRUE IDE is supported.

ATA drives: True IDE mode pin functions

Refer to *True IDE mode pin functions* on page 426 for information.

ATA drives: Hardware interfaces

Signal	Pin	Pin	Signal
RESETÐ	1	2	Ground
DD7	3	4	DD8
DD6	5	6	DD9
DD5	7	8	DD10
DD4	9	10	DD11
DD3	11	12	DD12
DD2	13	14	DD13
DD1	15	16	DD14
DD0	17	18	DD15
Ground	19	20	key (no pin)
DMARQ	21	22	Ground
DIOWÐ	23	24	Ground
DIORÐ	25	26	Ground
IORDY	27	28	SPSYNC:CSEL
DMACKÐ	29	30	Ground
INTRQ	31	32	IOCS16Ð
DA1	33	34	PDIAGÐ
DA0	35	36	DA2
CS1FXÐ	37	38	CS3FXÐ
DASPÐ	39	40	Ground

Signal / use	Pin	Pin	Signal / use
master/slave jumper	A	B	master/slave jumper
master/slave jumper	C	D	master/slave jumper
no pin			no pin
RESET–	1	2	Ground
DD7	3	4	DD8
DD6	5	6	DD9
DD5	7	8	DD10
DD4	9	10	DD11
DD3	11	12	DD12
DD2	13	14	DD13
DD1	15	16	DD14
DD0	17	18	DD15
Ground	19	20	key (no pin)
DMARQ	21	22	Ground
DIOW–	23	24	Ground
DIOR–	25	26	Ground
IORDY	27	28	SPSYNC:CSEL
DMACK–	29	30	Ground
INTRQ	31	32	IOCS16–
DA1	33	34	PDIAG–
DA0	35	36	DA2
CS1FX–	37	38	CS3FX–
DASP–	39	40	Ground
+5V (logic)	41	42	+5V (motor)
+Ground	43	44	Type

6.6.3 Fail-safe operation

Unexpected Reset

The data will be preserved.

Power failure

Power failure can be critical: If the card does not have sufficient time to complete a write operation, data may be lost. Countermeasures: make sure the power supply for the card drops slowly.

6.6.4 Wear-leveling

CompactFlash card are controlled by an internal controller, this controller also handles wear leveling. Therefore, the driver does not need to handle wear-leveling.

6.6.5 Configuring the driver

6.6.5.1 Adding the driver to emFile

To add the driver, use `FS_AddDevice()` with the driver label `FS_IDE_Driver`. This function has to be called from within `FS_X_AddDevices()`. Refer to *FS_X_AddDevices()* on page 472 for more information.

Example

```
FS_AddDevice(&FS_IDE_Driver);
```

6.6.5.2 FS_IDE_Configure()

Description

Configures the IDE/CF drive. This function has to be called from `FS_X_AddDevices()`. `FS_IDE_Configure()` can be called before or after adding the device driver to the file system. Refer to *FS_X_AddDevices()* on page 472 for more information.

Prototype

```
void FS_IDE_Configure(U8  Unit, U8  IsSlave);
```

Parameter	Description
Unit	Unit number (0…N).
ISSlave	Specifies whether the unit is connected

Table 6.190: FS_IDE_Configure() parameter list

Additional information

This function only needs to be called when the device does not use the default IDE master/slave configuration. By default, all even units (0,2,4...) are master, all odd units are slave (1, 3, 5...).

Example

Configure 2 different IDE/CF devices:

```
void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  //  Add 2 instances of the IDE driver
  //
  FS_AddDevice(&FS_IDE_Driver);
  FS_AddDevice(&FS_IDE_Driver);
  //
  //  Set the first unit as MASTER
  //
  FS_IDE_Configure(0, 0);
  //
  //  Set the second unit as MASTER
  //
  FS_IDE_Configure(1, 0);
}
```


6.6.6 Hardware functions

Routine	Explanation
Control line function	
FS_IDE_HW_Reset()	Resets the bus interface.
FS_IDE_HW_Delay400ns()	Waits 400ns.
FS_IDE_HW_IsPresent()	Checks if a device is present.
ATA I/O register access functions	
FS_IDE_HW_ReadReg()	Reads an IDE register. Data from the IDE register are read 16-bit wide.
FS_IDE_HW_WriteReg()	Write an IDE register. Data to the IDE register are written 16-bit wide.
FS_IDE_HW_ReadData()	Reads data from the IDE data register.
FS_IDE_HW_WriteData()	Writes data to the IDE data register.

Table 6.191: CompactFlash / IDE device driver functions

6.6.6.1 FS_IDE_HW_Reset()

Description

Resets the bus interface.

Prototype

```
void FS_IDE_HW_Reset (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0…N).

Table 6.192: FS_IDE_HW_Reset() parameter list

Additional Information

This function is called, when the driver detects a new media is present. For ATA HD drives, there is no action required and this function can be empty.

Example

```
void FS_IDE_HW_X_Reset(U8 Unit) {
  FS_USE_PARA(Unit);
}
```


6.6.6.2 FS_IDE_HW_Delay400ns()

Description

Waits 400ns.

Prototype

```
void FS_IDE_HW_Delay400ns (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0…N).

Table 6.193: FS_IDE_HW_Delay400ns() parameter list

Additional Information

`FS_IDE_HW_X_Delay400ns()` is always called when a command is sent or parameters are set in the IDE/CF drive. The integrated logic may need a delay of 400ns.
When using slow IDE/CF drives with fast processors this function should guarantee that a delay of 400ns is kept.
However this function may be empty if you intend to use fast drives (Modern CF-Cards and IDE drives are faster than 400ns when executing commands.)

Example

```
void FS_IDE_HW_X_Delay400ns(U8 Unit) {
  FS_USE_PARA(Unit);
}
```


6.6.6.3 FS_IDE_HW_IsPresent()

Description

Checks if the device is connected.

Prototype

```
U8 FS_IDE_HW_IsPresent (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0…N).

Table 6.194: FS_IDE_HW_IsPresent() parameter list

Return value

== 1: Device is connected.
== 0: Device is not connected.

Example

```
int FS_IDE_HW_IsPresent(U8 Unit) {
  FS_USE_PARA(Unit);
  return 1;
}
```


6.6.6.4 FS_IDE_HW_ReadReg()

Description

Reads an IDE register. Data from the IDE register are read 16-bit wide.

Prototype

```
U16 FS_IDE_HW_ReadReg (U8       Unit,
                       unsigned AddrOff);
```

Parameter	Meaning
Unit	Unit number (0…N).
AddrOff	Address offset that specifies which IDE register should be read.

Table 6.195: FS_IDE_HW_ReadReg() parameter list

Return value

Data read from the IDE register.

Example

```
U16  FS_IDE_HW_ReadReg(U8 Unit, unsigned AddrOff) {
  volatile U16 * pIdeReg;

  FS_USE_PARA(Unit);
  pIdeReg = _Getp(AddrOff);
  return *pIdeReg;
}
```


6.6.6.5 FS_IDE_HW_WriteReg()

Description

Writes an IDE register. Data to the IDE register are written 16-bit wide.

Prototype

```
void FS_IDE_HW_WriteReg (U8       Unit,
                         unsigned AddrOff,
                         U16      Data);
```

Parameter	Meaning
Unit	Unit number (0…N).
AddrOff	Address offset that specifies which IDE register should be written.
Data	Value that should be written to the register.

Table 6.196: FS_IDE_HW_WriteReg() parameter list

Example

```
void FS_IDE_HW_WriteReg(U8 Unit, unsigned AddrOff, U16 Data) {
  volatile U16 * pIdeReg;

  FS_USE_PARA(Unit);
  pIdeReg = _Getp(AddrOff);
  *pIdeReg = Data;
}
```


6.6.6.6 FS_IDE_HW_ReadData()

Description

Reads data from the IDE data register.

Prototype

```
void FS_IDE_HW_ReadData (U8        Unit,
                         U16       pData,
                         unsigned  NumBytes);
```

Parameter	Meaning
Unit	Unit number (0…N).
pData	Pointer to a read buffer.
NumBytes	Number of bytes that should be read.

Table 6.197: FS_IDE_HW_ReadData() parameter list

Example

```
void FS_IDE_HW_ReadData(U8 Unit, U8 * pData, unsigned NumBytes) {
  unsigned       NumItems;
  volatile U16 * pIdeReg;
  U16          * pData16;

  pIdeReg = _Getp(AddrOff);
  NumItems = NumBytes >> 1;
  pData16 = (U16 *)pData;
  do {
    *pData16++ = *pIdeReg;
  }  while (--NumItems);
}
```


6.6.6.7 FS_IDE_HW_WriteData()

Description

Writes data to the IDE data register.

Prototype

```
void FS_IDE_HW_WriteData (U8        Unit,
                          U16       Data,
                          unsigned  NumBytes);
```

Parameter	Meaning
Unit	Unit number (0…N).
pData	Pointer to a buffer of data which should be written.
NumBytes	Number of bytes that should be read.

Table 6.198: FS_IDE_HW_WriteData() parameter list

Example

```
void FS_IDE_HW_WriteData(U8 Unit, const U8 * pData, unsigned NumBytes) {
  unsigned       NumItems;
  volatile U16 * pIdeReg;
  U16          * pData16;

  pIdeReg = _Getp(AddrOff);
  NumItems = NumBytes >> 1;
  pData16 = (U16 *)pData;
  do {
    *pIdeReg = *pData16++;
  }  while (--NumItems);
}
```


6.6.7 Additional information

The emFile's generic CompactFlash & IDE device driver can be used to access most ATA HD drives or CompactFlash storage cards also known as CF using true IDE or Memory card mode. For details on CompactFlash cards, check the specification, which is available at:

http://www.compactflash.org/

Information about the AT Attachment interface can be found at the Technical Committee T13, who is responsible for the ATA standard:

http://www.t13.org/

6.6.8 Performance and resource usage

6.6.8.1 ROM usage

The ROM usage depends on the compiler options, the compiler version, and the used CPU. The memory requirements of the IDE/CF driver displayed in the table have been measured on a system as follows: ARM7, IAR Embedded Workbench V4.41A, Thumb mode, Size optimization.

Module	ROM [Kbytes]
emFile IDE/CF driver	1.6

6.6.8.2 Static RAM usage

Static RAM usage is the amount of RAM required by the driver for variables inside the driver. The number of bytes can be seen in a compiler list file

Static RAM usage of the IDE/CF driver: 24 bytes.

6.6.8.3 Performance

These performance measurements are in no way complete, but they give an approximation of the length of time required for common operations on various targets. The tests were performed as described in *Performance* on page 502.

All values are in Mbytes/sec.

Device	CPU speed	Medium	W	R
LogicPD LH79520	51 MHz	IDE mem-mapped	1.4	1.7
Cogent EP7312	74 MHz	CompacFlash card, True IDE mode	1.9	2.5
Cogent EP7312	74 MHz	HDD, True IDE mode	1.7	2.4

Table 6.199: Performance values for sample configurations

6.7 *WinDrive* driver

The purpose of this driver is to run emFile for test and simulation purposes on a PC running Windows. Refer to the chapter *Getting started* on page 27 for a sample using the WinDrive driver.

6.7.1 Supported hardware

This driver is compatible with use any Windows logical driver on a Windows NT system.

Be aware, that Win9X is not supported, because it cannot access logical drives with "CreateFile".

6.7.2 Theory of operation

emFile supports in this version FAT and EFS file systems only. NTFS logical drives cannot be accessed by emFile. It can be used either to store/access files on a floppy disk or using an USB-Card reader for accessing flash cards. It works also on FAT formatted hard disks or partitions.

Note: Do not use this driver on partitions containing important data. It is primarily meant to be used for evaluation purposes. Problems may occur if the program using emFile is debugged or terminated using the task manager.

6.7.3 Fail-safe operation

Although not important since the driver is not designed to be used in an embedded device, the data is normally safe. Data safety is handled by the underlying operating system and hardware.

6.7.4 Wear leveling

The driver does not need wear leveling.

6.7.5 Configuring the driver

6.7.5.1 Adding the driver to emFile

To add the driver use `FS_AddDevice()` with the driver label `FS_WINDRIVE_Driver`. This function has to be called from within `FS_X_AddDevices()`. Refer to *FS_X_AddDevices()* on page 472 for more information.

Example

```
FS_AddDevice(&FS_WINDRIVE_Driver);
```

6.7.5.2 WINDRIVE_Configure()

Description

Configures a windows drive instance. This function has to be called from within `FS_X_AddDevices()` after adding an instance of the `Windrive` driver. Refer to *FS_X_AddDevices()* on page 472 for more information.

Prototype

```
void WINDRIVE_Configure(U8 Unit, const char * sDriveName);
```

Parameter	Description
Unit	Unit number (0…n).
sDriveName	Pointer to string which contains the windows drive name. For example:"\\\\.\\a:"

Table 6.200: FS_Windrive_Configure() parameter list

Additional information

If sDriveName is NULL a configuration dialog will be opened to select which drive should be used.

6.7.6 Hardware functions

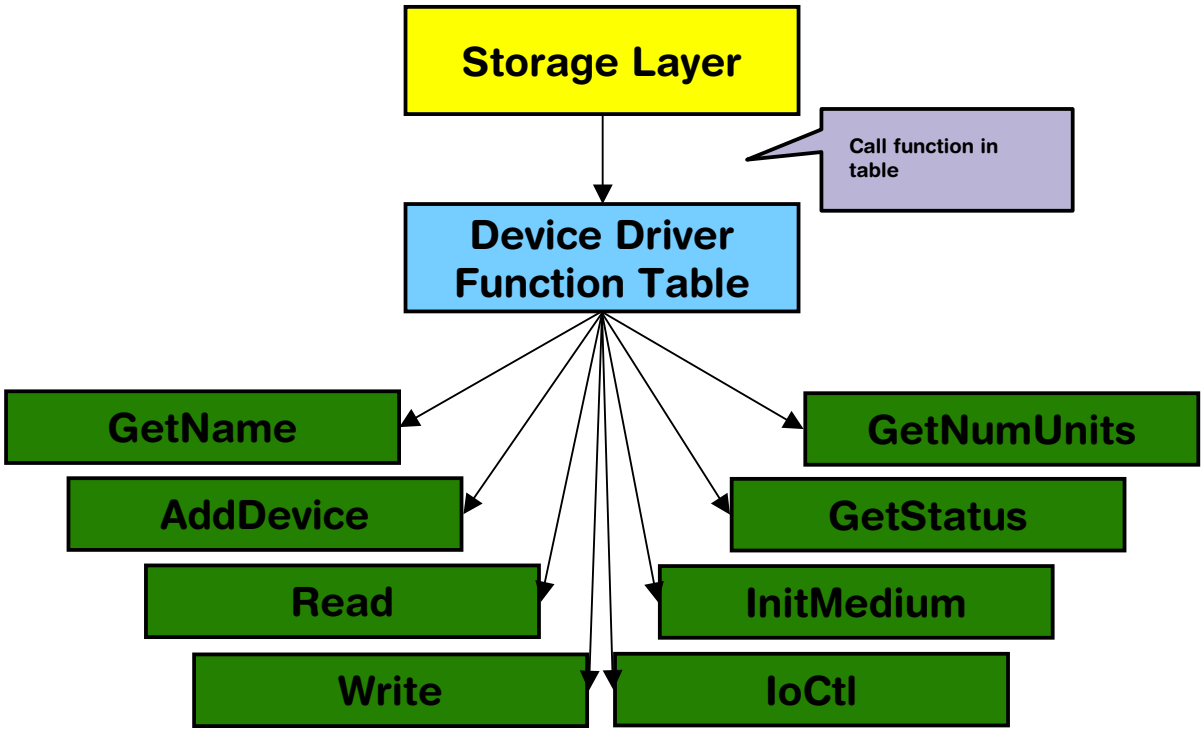
The WinDrive driver does not need any hardware functions.

6.7.7 Additional information

None.

6.8 Writing your own driver

If you are going to use emFile with your own hardware, you may have to write your own device driver. This section describes which functions are required and how to integrate your own device driver into emFile.



6.8.1 Device driver functions

This section provides descriptions of the device driver functions required by emFile. Note that the names used for these functions are not really relevant for emFile because the file system accesses them through a function table.

Routine	Explanation
AddDevice()	Adds a device to file system.
GetName()	Returns the name of the device.
GetNumUnits()	Returns the number of units.
GetStatus()	Returns the Status of the device.
InitMedium()	Initializes the device.
IoCtl()	Executes a special command on a device.
Read()	Reads data from a device.
Write()	Writes data to a device.

Table 6.201: Device driver functions

6.8.2 Device driver function table

emFile uses function tables to call the appropriate driver function for a device.

Data structure

```
typedef struct {
  const char *      (*pfGetName)      (U8            Unit);
  int               (*pfAddDevice)    (void);
  int               (*pfRead)         (U8            Unit,
                                       U32           SectorNo,
                                       void *        pBuffer,
                                       U32           NumSectors);
  int               (*pfWrite)        (U8            Unit,
                                       U32           SectorNo,
                                       const void *  pBuffer,
                                       U32           NumSectors,
                                       U8            RepeatSame);
  int               (*pfIoCtl)        (U8            Unit,
                                       I32           Cmd,
                                       I32           Aux,
                                       void *        pBuffer);
  int               (*pfInitMedium)   (U8            Unit);
  int               (*pfGetStatus)    (U8            Unit);
  int               (*pfGetNumUnits)  (void);
} FS_DEVICE_TYPE;
```

Elements of FS_DEVICE_TYPE

Element	Meaning
pfGetName	Pointer to a function that returns the name of the driver.
pfRead	Pointer to the device read sector function.
pfWrite	Pointer to the device write sector function.
pfIoCtl	Pointer to the device IoCtl function.
pfInitMedium	Pointer to the medium initialization function. (optional)
pfGetStatus	Pointer to the device status function.
pfGetNumUnits	Pointer to a function that returns the number of available devices.

Table 6.202: FS_DEVICE_TYPE - List of structure member variables

Example

```
/* sample implementation taken from the RAM device driver */

const FS_DEVICE_TYPE FS_RAMDISK_Driver = {
  _GetDriverName,
  _AddDevice,
  _Read,
  _Write,
  _IoCtl,
  NULL,
  _GetStatus,
  _GetNumUnits
};
```


6.8.3 Integrating a new driver

There is an empty skeleton driver called `generic` in the `Sample\Driver\DriverTemplate\` folder. This driver can be easily modified to get any block oriented storage device working with the file system.

To add the driver to emFile, `FS_AddDevice()` should be called from within `FS_X_AddDevices()` to mount the device driver to emFile before accessing the device or its units. Refer to *FS_X_AddDevices()* on page 472 for more information.

Chapter 7

Logical drivers

Optional software components located between file system layer and device driver layer which extend the functionality of emFile.

7.1 General information

7.1.1 Default logical driver names

By default the following identifiers are used for each driver.

Driver (Logical)	Identifier	Name
Storage partitioning	FS_DISKPART_Driver	"diskpart:"
Encryption	FS_CRYPT_Driver	"crypt:"
Sector read-ahead	FS_READAHEAD_Driver	"rah:"
Sector size adapter	FS_SECSIZE_Driver	"secsize:"
Sector write buffer	FS_WRBUF_Driver	"wrbuf:"
RAID	FS_RAID1_Driver	"raid:"

Table 7.1: List of default logical driver labels

To add a logical driver to emFile, `FS_AddDevice()` should be called with the proper identifier. Refer to *FS_AddDevice()* on page 59 for detailed information.

7.1.2 Unit number

Most driver functions receive the unit number as the first parameter. The unit number allows distinction between the different instances of the same driver type.

7.2 Disk partition driver

This logical driver can be used to access storage medium partitions as defined in a Master Boot Record (MBR). MBR contains information about how the storage medium is divided and an optional machine code for bootstrapping PC-compatible computers. It is always stored on the first physical sector of the storage medium. The partitioning information is stored in a partition table which contains 4 entries each 16 byte large. Each entry stores the following information about the partition:

Offset [bytes]	Size [byte]	Description
0	1	Partition status: • 0x80 - bootable • 0x00 - non-bootable • else - invalid
1	3	First sector in the partition as cylinder/head/sector address
4	1	Partition type
5	3	Last sector in the partition as cylinder/head/sector address
8	4	First sector in the partition as logical block address
12	4	Number of sectors in the partition

Table 7.2: Partition table entry layout

The driver uses only the information stored in a valid partition table entry. Invalid partition table entries are ignored. The position and the size of the partition are taken from the last 2 fields. The cylinder/head/sector information and the partition type are also ignored.

A separate volume is assigned to each driver instance. The volumes can be accessed using the following names: `"diskpart:0:"`, `"diskpart:1:"`, etc.

Note: This logical driver is not required if an application should access only the first storage medium partition, as emFile will use this partition by default.

7.2.1 Configuring the driver

To add the driver, use `FS_AddDevice()` with the driver identifier set to `FS_DISKPART_Driver`. This function has to be called from within `FS_X_AddDevices()`. Refer to *FS_X_AddDevices()* on page 472 for more information.

7.2.1.1 FS_DISKPART_Configure()

Description

Configures an instance of the logical driver. This function has to be called from within `FS_X_AddDevices()` after adding the logical driver instance. Refer to *FS_X_AddDevices()* on page 472 for more information.

Prototype

```
void FS_DISKPART_Configure(U8                      Unit,
                           const FS_DEVICE_TYPE * pDevice,
                           U8                      DeviceUnit,
                           U8                      PartIndex);
```

Parameter	Description
`Unit`	Unit number of the instance to configure.

Table 7.3: FS_DISKPART_Configure() parameter list

Parameter	Description
pDevice	IN: Device driver used to access the storage medium OUT: ---
DeviceUnit	Unit number of device driver.
PartIndex	Index in the partition table of the partition to be accessed. 0 is the fist partition, 1 is the second, etc.

Table 7.3: FS_DISKPART_Configure() parameter list

Additional information

The function does not access the storage medium. It simply stores the parameters to driver instance. The size and the position of the partition is read from MBR on the first access to storage medium.

Example

This example demonstrates how to configure emFile to access the first 2 MBR partitions of an SD card.

```
#define ALLOC_SIZE  2048    // Size of emFile memory pool

static U32 _aMemBlock[ALLOC_SIZE / 4];

/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*    It is supposed to add all devices, using primarily FS_AddDevice().
*
*  Note
*    (1) Other API functions
*        Other API functions may NOT be called, since this function is called
*        during initialisation. The devices are not yet ready at this point.
*/
void FS_X_AddDevices(void) {
  U8 DeviceUnit;
  U8 PartIndex;
  U8 Unit;

  FS_AssignMemory(_aMemBlock, sizeof(_aMemBlock));
  //
  // Add SD/MMC card device driver.
  //
  DeviceUnit = 0;
  FS_AddPhysDevice(&FS_MMC_CardMode_Driver);
  //
  // Configure logical driver to access the first MBR partition.
  // Partition will be mounted as volume "diskpart:0:".
  //
  PartIndex = 0;
  Unit      = 0;
  FS_AddDevice(&FS_DISKPART_Driver);
  FS_DISKPART_Configure(Unit, &FS_MMC_CardMode_Driver, DeviceUnit, PartIndex);
  //
  // Configure logical driver to access the second MBR partition.
  // Partition will be mounted as volume "diskpart:1:".
  //
  PartIndex = 1;
  Unit      = 1;
  FS_AddDevice(&FS_DISKPART_Driver);
  FS_DISKPART_Configure(Unit, &FS_MMC_CardMode_Driver, DeviceUnit, PartIndex);
}
```


7.2.2 Performance and resource usage

7.2.2.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the driver presented in the tables below have been measured using the following system: Cortex-M4, IAR Embedded Workbench V6.30, Size optimization.

Module	ROM [kBytes]
emFile Disk partition driver	1.5

7.2.2.2 Static RAM usage

Static RAM usage is the amount of RAM required by the driver for variables inside of the driver. The number of bytes can be seen in a compiler list file.

Static RAM usage of driver: 20 bytes

7.2.2.3 Runtime RAM usage

Runtime RAM usage is the amount of RAM allocated by the driver at runtime.

Runtime RAM usage of driver: 18 bytes

7.3 Encryption driver

This is an additional logical driver which can be used to protect the file system data against unauthorized access. The data is encrypted using a very efficient implementation of the Data Encryption Standard and of the Advanced Encryption Standard (AES) algorithm. AES algorithms are provided for 128-bit and 256-bit key lengths.

The logical driver can be used with both FAT and EFS file systems and with any supported storage medium.

A separate volume is assigned to each driver instance. The volumes can be accessed using the following names: `"crypt:0:", "crypt:1:"`, etc.

Theory of operation

The sector data is transformed to make it unreadable for anyone which tries to read it directly. The operation which makes the data unreadable is called encryption and is performed when the file system writes the sector data. When the contents of a sector is read the reversed operation takes place which makes the data readable. This is called decryption. Both operations use a cryptographic algorithm and a key to transform the data. The same key is used for encryption and decryption. Without the knowledge of the key it is not possible to decrypt the data.

7.3.1 Configuring the driver

To add the driver, call `FS_AddDevice()` with the driver identifier set to `FS_CRYPT_Driver`. This function has to be called from within `FS_X_AddDevices()`. Refer to *FS_X_AddDevices()* on page 472 for more information.

7.3.1.1 FS_CRYPT_Configure()

Description

Configures a driver instance. The function must be called from within `FS_X_AddDevices()` after the creation of driver instance. Refer to *FS_X_AddDevices()* on page 472 for more information.

Prototype

```
void FS_CRYPT_Configure(U8                          Unit,
                        const FS_DEVICE_TYPE     * pDevice,
                        U8                          DeviceUnit,
                        const FS_CRYPT_ALGO_TYPE * pAlgoType,
                        void                     * pContext,
                        const U8                 * pKey);
```

Parameter	Description
Unit	Unit number of the driver instance to configure.
pDevice	IN: Device driver used to access the storage medium OUT: ---
DeviceUnit	Unit number of device driver.
pAlgoType	IN: Type of encryption algorithm OUT: ---
pContext	IN: Data specific to inception algorithm OUT: ---
pKey	IN: Password for data encryption/decryption OUT: ---

Table 7.4: FS_CRYPT_Configure() parameter list

Permitted values for parameter pAlgoType	
FS_CRYPT_ALGO_DES	DES encryption using a 56-bit key.
FS_CRYPT_ALGO_AES128	AES encryption using a 128-bit key.
FS_CRYPT_ALGO_AES256	AES encryption using a 256-bit key.

Additional information

The pContext memory location is passed to as parameter to encryption/decryption routines. It should remain valid from the moment the driver is configured until the FS_DeInit() function is called.

The number of bytes in pKey array should match the size of the key required by the encryption algorithm.

Example

This example demonstrates how to configure emFile to secure the data of an SD card using the AES algorithm with 128-bit key.

```
#define ALLOC_SIZE  2048    // Size of emFile memory pool

static U32            _aMemBlock[ALLOC_SIZE / 4];
static FS_AES_CONTEXT _Context;

/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*    It is supposed to add all devices, using primarily FS_AddDevice().
*
*  Note
*    (1) Other API functions
*        Other API functions may NOT be called, since this function is called
*        during initialisation. The devices are not yet ready at this point.
*/
void FS_X_AddDevices(void) {
  U8 DeviceUnit;
  U8 PartIndex;
  U8 Unit;

  U8 aPass[16] = {'s', 'e', 'c', 'r', 'e', 't'};

  FS_AssignMemory(_aMemBlock, sizeof(_aMemBlock));
  //
  // Add SD/MMC card device driver.
  //
  DeviceUnit = 0;
  FS_AddPhysDevice(&FS_MMC_CardMode_Driver);
  FS_MMC_CM_Allow4bitMode(0, 1);
  //
  // Add the encryption driver. The storage can be accessed as volume "crypt:0:".
  //
  Unit = 0;
  FS_AddDevice(&FS_CRYPT_Driver);
  FS_CRYPT_Configure(Unit,
                     &FS_MMC_CardMode_Driver,
                     DeviceUnit,
                     &FS_CRYPT_ALGO_AES128,
                     &_Context,
                     aPass);
}
```


7.3.2 Performance and resource usage

7.3.2.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the driver presented in the tables below have been measured using the following system: Cortex-M4, IAR Embedded Workbench V6.30, Size optimization.

Module	ROM [Kbytes]
emFile Encryption driver	1.3

In addition, one of the following cryptographic algorithms is required:

Physical layer	Description	ROM [Kbytes]
FS_CRYPT_ALGO_DES	DES encryption algorithm.	3.2
FS_CRYPT_ALGO_AES128	AES encryption algorithm using an 128-bit key.	12.0
FS_CRYPT_ALGO_AES256	AES encryption algorithm using a 256-bit key.	12.0

7.3.2.2 Static RAM usage

Static RAM usage is the amount of RAM required by the driver for variables inside of the driver. The number of bytes can be seen in a compiler list file.

Static RAM usage of driver: 24 bytes

7.3.2.3 Runtime RAM usage

Runtime RAM usage is the amount of RAM allocated by the driver at runtime. The amount required depends on the runtime configuration and on the selected encryption algorithm.

Every driver instance requires 16 bytes. In addition the context of the AES encryption algorithm requires 480 bytes and of the DES encryption algorithm 128 bytes.

7.3.2.4 Performance

These performance measurements are in no way complete, but they give an approximation of the length of time required for common operations on various targets. The tests were performed as described in *Performance* on page 502.

All values are in kBytes/sec.

Device	CPU speed	Medium	W	R
ST STM32F207	96 MHz	SD card as storage medium using AES with an 128-bit key	639	661
Freescale Kinetis K60	120 MHz	NAND flash interfaced via 8-bit bus using AES with an 128-bit key.	508	550

Table 7.5: Performance values for sample configurations

7.4 Sector read-ahead driver

The driver reads in advance more sectors than requested and caches them to provided buffer. The maximum number of sectors which fit in the buffer are read at once. If the requested sectors are present in the buffer the driver returns the cached sector contents and the storage medium is not accessed. The driver should be used on SD/MMC/eMMC storage mediums where reading single sectors is less efficient than reading all the sectors at once. By default the driver is not active. The file system activates the driver when the allocation table is searched for free clusters. This will improve performance in the case where the whole allocation table needs to be scanned. To activate the support for read-ahead in the file system the `FS_SUPPORT_READ_AHEAD` define must be set to 1 in `FS_Conf.h`. Both FAT and EFS file systems support read-ahead.

7.4.1 Configuring the driver

To add the driver, call `FS_AddDevice()` with the driver identifier set to `FS_READAHEAD_Driver`. This function has to be called from within `FS_X_AddDevices()`. Refer to *FS_X_AddDevices()* on page 472 for more information.

7.4.1.1 FS_READAHEAD_Configure()

Description

Configures an instance of the logical driver. This function has to be called from within `FS_X_AddDevices()` after adding the logical driver instance. Refer to *FS_X_AddDevices()* on page 472 for more information.

Prototype

```
void FS_READAHEAD_Configure(U8                       Unit,
                            const FS_DEVICE_TYPE * pDevice,
                            U8                       DeviceUnit,
                            U32                    * pData,
                            U32                      NumBytes);
```

Parameter	Description
`Unit`	Unit number of the driver instance to configure.
`pDevice`	IN: Device driver used to access the storage medium OUT: ---
`DeviceUnit`	Unit number of device driver.
`pData`	Buffer to store the sector data read from storage medium.
`NumBytes`	Number of bytes in the read buffer.

Table 7.6: FS_READAHEAD_Configure() parameter list

Additional information

The read buffer should be at least one sector large.

Example

This example demonstrates how to configure the access to an SD card.


```
#define ALLOC_SIZE  2048    // Size of emFile memory pool
#define BUFFER_SIZE 4096

static U32 _aMemBlock[ALLOC_SIZE / 4];
static U32 _aReadBuffer[BUFFER_SIZE / 4];

/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*    It is supposed to add all devices, using primarily FS_AddDevice().
*
*  Note
*    (1) Other API functions
*        Other API functions may NOT be called, since this function is called
*        during initialisation. The devices are not yet ready at this point.
*/
void FS_X_AddDevices(void) {

  FS_AssignMemory(_aMemBlock, sizeof(_aMemBlock));
  //
  // Add SD/MMC card device driver.
  //
  FS_AddPhysDevice(&FS_MMC_CardMode_Driver);
  //
  // Add and configure the read-ahead driver. Volume name: "rah:0:"
  //
  FS_AddDevice(&FS_READAHEAD_Driver);
  FS_READAHEAD_Configure(0, &FS_MMC_CardMode_Driver, 0,
                         _aReadBuffer, sizeof(_aReadBuffer));
}
```

7.4.2 Performance and resource usage

7.4.2.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the driver presented in the tables below have been measured using the following system: Cortex-M4, IAR Embedded Workbench V6.30, Size optimization.

Module	ROM [kBytes]
emFile Read-ahead driver	1.4

7.4.2.2 Static RAM usage

Static RAM usage is the amount of RAM required by the driver for variables inside of the driver. The number of bytes can be seen in a compiler list file.

Static RAM usage of driver: 20 bytes

7.4.2.3 Runtime RAM usage

Runtime RAM usage is the amount of RAM allocated by the driver at runtime.

Runtime RAM usage of driver: 30 bytes

7.4.2.4 Performance

The detection of free space is 2 times faster when a 4KB read ahead buffer is used (measured on 4GB SD card formatted with 4KB clusters).

7.5 Sector size adapter driver

The logical driver supports the access to a storage medium using a sector size different than that of the underlying layer (typically a storage driver). The sector size of logical driver is configurable and can be larger or smaller than the sector size of the below layer.

Typically, the logical driver is placed between the file system and a storage driver and it is configured with a sector size smaller than that of the storage layer to help reduce the RAM usage of the internal sector buffers of the file system.

7.5.1 Configuring the driver

To add the driver, call `FS_AddDevice()` with the driver identifier set to `FS_SECSIZE_Driver`. This function has to be called from within `FS_X_AddDevices()`. Refer to *FS_X_AddDevices()* on page 472 for more information.

7.5.1.1 FS_SECSIZE_Configure()

Description

Configures an instance of the logical driver. This function has to be called from within `FS_X_AddDevices()` after adding the logical driver instance. Refer to *FS_X_AddDevices()* on page 472 for more information.

Prototype

```
void FS_SECSIZE_Configure(U8                       Unit,
                          const FS_DEVICE_TYPE * pDevice,
                          U8                       DeviceUnit,
                          U16                      BytesPerSector);
```

Parameter	Description
`Unit`	Unit number of the driver instance to configure.
`pDevice`	IN: Device driver used to access the storage medium OUT: ---
`DeviceUnit`	Unit number of device driver.
`BytesPerSector`	Sector size in bytes presented to file system.

Table 7.7: FS_SECSIZE_Configure() parameter list

Additional information

The sector size should be a power of 2 value.

Example

This example demonstrates how to configure the logical driver to access a NAND flash via the Universal NAND driver.


```
#define ALLOC_SIZE  0x8000    // Size of emFile memory pool

static U32 _aMemBlock[ALLOC_SIZE / 4];

/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*    It is supposed to add all devices, using primarily FS_AddDevice().
*
*  Note
*    (1) Other API functions
*        Other API functions may NOT be called, since this function is called
*        during initialisation. The devices are not yet ready at this point.
*/
void FS_X_AddDevices(void) {

  FS_AssignMemory(_aMemBlock, sizeof(_aMemBlock));
  //
  // Add NAND flash device driver.
  //
  FS_AddPhysDevice(&FS_NAND_UNI_Driver);
  FS_NAND_UNI_SetPhyType(0, &FS_NAND_PHY_ONFI);
  FS_NAND_UNI_SetECCHook(0, &FS_NAND_ECC_HW_NULL);
  //
  // Add sector conversion logical driver.
  //
  FS_AddDevice(&FS_SECSIZE_Driver);
  FS_SECSIZE_Configure(0, &FS_NAND_UNI_Driver, 0, 512);
}
```

7.5.2 Performance and resource usage

7.5.2.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the driver presented in the tables below have been measured using the following system: Cortex-M4, IAR Embedded Workbench V6.30, Size optimization.

Module	ROM [kBytes]
emFile Sector size driver	1.1

7.5.2.2 Static RAM usage

Static RAM usage is the amount of RAM required by the driver for variables inside of the driver. The number of bytes can be seen in a compiler list file.

Static RAM usage of driver: 20 bytes

7.5.2.3 Runtime RAM usage

Runtime RAM usage is the amount of RAM allocated by the driver at runtime.

Runtime RAM usage of driver: 30 bytes + sector size of storage driver if it is larger than the sector size configured for the logical driver.

7.6 Sector write buffer driver

This driver has been designed to help improving the write performance of the file system. It operates by temporarily storing the sector data to RAM which takes significantly less time than writing it directly to a storage. The sector data is written later to storage at the request of the application or when the internal buffer is full. The sectors are written to storage in the same order in which they were written by the file system.

Some of the advantages of using this driver are:

- a file system write operation blocks for a very short period of time
- the number write operations is reduced when the same sector is written in succession
- the driver can be used with activated Journal since the write order is preserved

The internal buffer can be cleaned from application by calling the `FS_STORAGE_Sync()` API function. The function blocks until all sectors stored in the internal buffer are written to storage. Typically, this function should be called from a low priority task when the application does not access the file system.

7.6.1 Configuring the driver

To add the driver, call `FS_AddDevice()` with the driver identifier set to `FS_WRBUF_Driver`. This function has to be called from within `FS_X_AddDevices()`. Refer to *FS_X_AddDevices()* on page 472 for more information.

7.6.1.1 FS_WRBUF_Configure()

Description

Configures an instance of the logical driver. This function has to be called from within `FS_X_AddDevices()` after adding the logical driver instance. Refer to *FS_X_AddDevices()* on page 472 for more information.

Prototype

```
void FS_WRBUF_Configure(U8                      Unit,
                        const FS_DEVICE_TYPE * pDevice,
                        U8                      DeviceUnit,
                        void                  * pBuffer,
                        U32                     NumBytes);
```

Parameter	Description
`Unit`	Unit number of the driver instance to configure.
`pDevice`	IN: Device driver used to access the storage medium OUT: ---
`DeviceUnit`	Unit number of device driver.
`pBuffer`	Memory to store the sector list.
`NumBytes`	Number of bytes allocated for the sector list.

Table 7.8: FS_WRBUF_Configure() parameter list

Additional information

The FS_SIZEOF_WRBUF() define can be used to compute the number of bytes required to store a given number of sectors. The first parameter specifies the number of sectors while the second one represents the size of the sector in bytes.

Example

This example demonstrates how to configure the logical driver to access a NOR flash via the NOR flash block-mode driver.


```
#define ALLOC_SIZE  0x1000    // Size of emFile memory pool

static U32 _aMemBlock[ALLOC_SIZE / 4];
static U32 _aWriteBuffer[FS_SIZEOF_WRBUF(8, 512) / 4];

/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*    It is supposed to add all devices, using primarily FS_AddDevice().
*
*  Note
*    (1) Other API functions
*        Other API functions may NOT be called, since this function is called
*        during initialisation. The devices are not yet ready at this point.
*/
void FS_X_AddDevices(void) {

  FS_AssignMemory(_aMemBlock, sizeof(_aMemBlock));
  //
  // Add and configure the NOR flash driver.
  //
  FS_AddPhysDevice(&FS_NOR_BM_Driver);
  FS_NOR_BM_SetPhyType(0, &FS_NOR_PHY_ST_M25);
  FS_NOR_BM_Configure(0, 0, 0, 0x100000uL);
  //
  // Add and configure the write buffer driver.
  //
  FS_AddDevice(&FS_WRBUF_Driver);
  FS_WRBUF_Configure(0, &FS_MMC_CardMode_Driver, 0,
                     _aWriteBuffer, sizeof(_aWriteBuffer));
}
```

7.6.2 Performance and resource usage

7.6.2.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the driver presented in the tables below have been measured using the following system: Cortex-M4, IAR Embedded Workbench V6.30, Size optimization.

Module	ROM [kBytes]
emFile Sector write buffer driver	1.2

7.6.2.2 Static RAM usage

Static RAM usage is the amount of RAM required by the driver for variables inside of the driver. The number of bytes can be seen in a compiler list file.

Static RAM usage of driver: 20 bytes

7.6.2.3 Runtime RAM usage

Runtime RAM usage is the amount of RAM allocated by the driver at runtime.

Runtime RAM usage of driver: 32 bytes.

7.7 RAID1 driver

The driver provides increased data integrity by keeping a copy of each sector data. It uses a primary (main) storage and a secondary (mirror) storage. The data is written to both primary and secondary storage and is read from the primary storage. In case of a read error the data is recovered by reading it from the secondary storage.

The driver can be configured to store the data on a single volume or on two separate volumes. In the case a single volume is configured the first half is used as primary storage. When using two volumes they do not need to have the same number of sectors. The number of sectors available to file system will be that of the smallest volume. It is required that both volumes have the same sector size. If necessary the sector size can be adapted using the SECSIZE logical driver. For more information refer to *Sector size adapter driver* on page 461.

NAND flash error recovery

The Universal NAND driver can make use of the RAID driver to avoid a data loss when an ECC error happens during a read operation. This feature is by default disabled and it can be enabled at compile time by setting the `FS_NAND_ENABLE_ERROR_RECOVERY` switch to 1 in `FS_Conf.h`.

Sector data synchronization

A sudden reset which interrupts a write operation can lead to an inconsistency where the data of the last written sector is stored only to primary storage but not to secondary storage. After restart the file system will continue to operate correctly but in case of an read error affecting this sector old data is read from secondary storage which might cause a data corruption. This situation can be prevented by synchronizing all the sectors on the storage. The application can perform the synchronization by calling the `FS_STORAGE_SyncSectors()` API function in a low priority task or after the file system initialization. For an example refer to *FS_RAID1_SetSyncBuffer()* on page 468.

7.7.1 Configuring the driver

7.7.1.1 FS_RAID1_Configure()

Description

Configures an instance of the logical driver. This function has to be called from within `FS_X_AddDevices()` after adding the logical driver instance. Refer to *FS_X_AddDevices()* on page 472 for more information.

Prototype

```
void FS_RAID1_Configure(U8                      Unit,
                        const FS_DEVICE_TYPE * pPrimaryDeviceType,
                        U8                      PrimaryDeviceUnit,
                        const FS_DEVICE_TYPE * pSecondaryDeviceType,
                        U8                      SecondaryDeviceUnit);;
```

Parameter	Description
`Unit`	Unit number of the driver instance to configure.
`pPrimaryDeviceType`	Device driver used to access the main storage medium.
`PrimaryDeviceUnit`	Unit number of primary device driver.
`pSecondaryDeviceType`	Device driver used to access the mirror storage medium.
`SecondaryDeviceUnit`	Unit number of secondary device driver.

Table 7.9: FS_RAID1_Configure() parameter list

Additional information

If the same device type and unit number is specified for the primary and the secondary storage the first half of the storage medium will be used for the primary (main) storage while the second half will be used for the secondary (mirror) storage.

Example

The following example shows how to configure RAID on a single volume.

```
#define ALLOC_SIZE      0x8000            // Size of the memory pool in bytes

static U32 _aMemBlock[ALLOC_SIZE / 4];    // Memory pool used for
                                          // semi-dynamic allocation.

/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*    It is supposed to add all devices, using primarily FS_AddDevice().
*
*  Note
*    (1) Other API functions
*        Other API functions may NOT be called, since this function is called
*        during initialization. The devices are not yet ready at this point.
*/
void FS_X_AddDevices(void) {
  //
  // Give the file system some memory to work with.
  //
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Set the file system sector size.
  //
  FS_SetMaxSectorSize(2048);
  //
  // Add and configure the first NAND driver.
  //
  FS_AddDevice(&FS_NAND_UNI_Driver);
  FS_NAND_UNI_SetPhyType(0, &FS_NAND_PHY_ONFI);
  FS_NAND_UNI_SetECCHook(0, &FS_NAND_ECC_HW_NULL);
  //
  // Add and configure the RAID driver.
  //
  FS_AddDevice(&FS_RAID1_Driver);
  FS_RAID1_Configure(0, &FS_NAND_UNI_Driver, 0, &FS_NAND_UNI_Driver, 0);
}
```

The next example demonstrates how to configure a RAID on 2 separate volumes.

```
#define ALLOC_SIZE      0x8000              // Size of the memory pool in bytes

static U32 _aMemBlock[ALLOC_SIZE / 4];      // Memory pool used for
                                            // semi-dynamic allocation.

/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*    It is supposed to add all devices, using primarily FS_AddDevice().
*
*  Note
*    (1) Other API functions
*        Other API functions may NOT be called, since this function is called
*        during initialization. The devices are not yet ready at this point.
*/
void FS_X_AddDevices(void) {
  //
  // Give the file system some memory to work with.
  //
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Set the file system sector size.
```



```
  //
  FS_SetMaxSectorSize(2048);
  //
  // Add and configure the NAND driver for the primary storage.
  //
  FS_AddDevice(&FS_NAND_UNI_Driver);
  FS_NAND_UNI_SetPhyType(0, &FS_NAND_PHY_ONFI);
  FS_NAND_UNI_SetECCHook(0, &FS_NAND_ECC_HW_NULL);
  //
  // Add and configure the NAND driver for the secondary storage.
  //
  FS_AddDevice(&FS_NAND_UNI_Driver);
  FS_NAND_UNI_SetPhyType(1, &FS_NAND_PHY_ONFI);
  FS_NAND_UNI_SetECCHook(1, &FS_NAND_ECC_HW_NULL);
  //
  // Add and configure the RAID driver.
  //
  FS_AddDevice(&FS_RAID1_Driver);
  FS_RAID1_Configure(0, &FS_NAND_UNI_Driver, 0, &FS_NAND_UNI_Driver, 1);
}
```

7.7.1.2 FS_RAID1_SetSectorRanges()

Description

Specifies an area which should be used as storage.

Prototype

```
void FS_RAID1_SetSectorRanges(U8  Unit,
                              U32 NumSectors,
                              U32 PrimaryStartSector,
                              U32 SecondaryStartSector);
```

Parameter	Description
Unit	Unit number of the driver instance to configure.
NumSectors	Number of the sectors to be used a storage.
PrimaryStartSector	Index of the first sector to be used on the primary storage.
SecondaryStartSector	Index of the first sector to be used on the secondary storage.

Table 7.10: FS_RAID1_SetSectorRanges() parameter list

Additional information

This function is optional and it must be called from within `FS_X_AddDevices()` after adding the logical driver instance. Refer to *FS_X_AddDevices()* on page 472 for more information. An error is reported when trying to access the RAID volume if the sector range is invalid or it does not fit into device.

Example

This example configures a RAID volume of 10000 sectors. The primary storage starts at sector index 0 and the secondary storage at sector index 10000.


```
#define ALLOC_SIZE      0x8000            // Size of the memory pool in bytes

static U32 _aMemBlock[ALLOC_SIZE / 4];    // Memory pool used for
                                          // semi-dynamic allocation.

/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*    It is supposed to add all devices, using primarily FS_AddDevice().
*
*  Note
*    (1) Other API functions
*        Other API functions may NOT be called, since this function is called
*        during initialization. The devices are not yet ready at this point.
*/
void FS_X_AddDevices(void) {
  //
  // Give the file system some memory to work with.
  //
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Set the file system sector size.
  //
  FS_SetMaxSectorSize(2048);
  //
  // Add and configure the first NAND driver.
  //
  FS_AddDevice(&FS_NAND_UNI_Driver);
  FS_NAND_UNI_SetPhyType(0, &FS_NAND_PHY_ONFI);
  FS_NAND_UNI_SetECCHook(0, &FS_NAND_ECC_HW_NULL);
  //
  // Add and configure the RAID driver.
  //
  FS_AddDevice(&FS_RAID1_Driver);
  FS_RAID1_Configure(0, &FS_NAND_UNI_Driver, 0, &FS_NAND_UNI_Driver, 0);
  FS_RAID1_SetSectorRanges(0, 10000, 0, 10000);
}
```

7.7.1.3 FS_RAID1_SetSyncBuffer()

Description

Provides a buffer for the synchronization operation.

Prototype

```
void FS_RAID1_SetSyncBuffer(U8     Unit,
                            void * pBuffer,
                            U32    NumBytes);
```

Parameter	Description
Unit	Unit number of the driver instance to configure.
pBuffer	Pointer to a memory location to be used as buffer.
NumBytes	Number of bytes in the buffer.

Table 7.11: FS_RAID1_SetSyncBuffer() parameter list

Additional information

This function is optional and it must be called from within `FS_X_AddDevices()` after adding the logical driver instance. Refer to *FS_X_AddDevices()* on page 472 for more information. The buffer must be large enough to hold the data of at least 2 sectors else the synchronization operation will fail. A larger buffer allows the driver to read/write several sectors a once which increases the performance of the synchronization operation.

Example

The following example configures a synchronization buffer of 16KB.


```
#define ALLOC_SIZE      0x8000              // Size of the memory pool in bytes
#define BUFFER_SIZE     0x8000              // Size of the sync buffer in bytes

static U32 _aMemBlock[ALLOC_SIZE / 4];      // Memory pool used for
                                            // semi-dynamic allocation.
static U32 _aSyncBuffer[BUFFER_SIZE / 4];   // Buffer for the RAID synchronization.

/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*    It is supposed to add all devices, using primarily FS_AddDevice().
*
*  Note
*    (1) Other API functions
*        Other API functions may NOT be called, since this function is called
*        during initialization. The devices are not yet ready at this point.
*/
void FS_X_AddDevices(void) {
  //
  // Give the file system some memory to work with.
  //
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Set the file system sector size.
  //
  FS_SetMaxSectorSize(2048);
  //
  // Add and configure the first NAND driver.
  //
  FS_AddDevice(&FS_NAND_UNI_Driver);
  FS_NAND_UNI_SetPhyType(0, &FS_NAND_PHY_ONFI);
  FS_NAND_UNI_SetECCHook(0, &FS_NAND_ECC_HW_NULL);
  //
  // Add and configure the RAID driver.
  //
  FS_AddDevice(&FS_RAID1_Driver);
  FS_RAID1_Configure(0, &FS_NAND_UNI_Driver, 0, &FS_NAND_UNI_Driver, 0);
  FS_RAID1_SetSyncBuffer(0, _aSyncBuffer, sizeof(_aSyncBuffer));
}
```

The following sample function can be used to synchronize the RAID volume after an unexpected reset.

```
/*********************************************************************
*
*       _SyncRAID
*
*  Function description
*    Performs RAID synchronization, Should be called from a low-priority task
*    in order to minimize the effect on the normal file system activity.
*/
static void _SyncRAID(void) {
  U32         iSector;
  FS_DEV_INFO DevInfo;

  //
  // Get the number of sectors on the storage.
  //
  FS_STORAGE_GetDeviceInfo("", &DevInfo);
  //
  // Synchronize one sector at a time to avoid
  // blocking the file system for a too long time.
  //
  for (iSector = 0; iSector < DevInfo.NumSectors; ++iSector) {
    FS_STORAGE_SyncSectors("", iSector, 1);
  }
}
```


7.7.2 Performance and resource usage

7.7.2.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the driver presented in the tables below have been measured using the following system: Cortex-M4, IAR Embedded Workbench V6.30, Size optimization.

Module	ROM [kBytes]
emFile RAID1 driver	1.5

7.7.2.2 Static RAM usage

Static RAM usage is the amount of RAM required by the driver for variables inside of the driver. The number of bytes can be seen in a compiler list file.

Static RAM usage of driver: 20 bytes

7.7.2.3 Runtime RAM usage

Runtime RAM usage is the amount of RAM allocated by the driver at runtime.

Runtime RAM usage of driver: 48 bytes.

Chapter 8

Configuration of emFile

emFile can be used without the need for changing any of the compile time flags. All compile time configuration flags are preconfigured with valid values, which matches the requirements of the most applications. Device drivers can be added at runtime.

The default configuration of emFile can be changed via compile time flags which can be added to `FS_Conf.h`. This is the main configuration file for the file system.

Every driver folder includes a configuration file (e.g. `FS_ConfigRamDisk.c`) with implementations of runtime configuration functions explained in this chapter. The configuration files are a good start, to run emFile “out of the box”.

8.1 Runtime configuration

Every driver folder includes a configuration file (e.g. FS_ConfigRamDisk.c) with implementations of runtime configuration functions explained in this chapter. These functions can be customized.

8.1.1 Driver handling

FS_X_AddDevices() is called by the initialization of the file system from FS_Init(). This function should help to bundle the process of adding and configuring the driver.

8.1.1.1 FS_X_AddDevices()

Description

Helper function called by FS_Init() to add devices to the file system and configure them.

Prototype

```
void FS_X_AddDevices(void);
```

Example

```
/*********************************************************************
*
*       FS_X_AddDevices
*/
void FS_X_AddDevices(void) {
  void  * pRamDisk;

  FS_AssignMemory(_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Allocate memory for the RAM disk
  //
  pRamDisk = FS_Alloc(RAMDISK_NUM_SECTORS * RAMDISK_BYTES_PER_SECTOR);
  //
  // Add driver
  //
  FS_AddDevice(&FS_RAMDISK_Driver);
  //
  // Configure driver
  //
  FS_RAMDISK_Configure(0, pRamDisk, RAMDISK_BYTES_PER_SECTOR, RAMDISK_NUM_SECTORS);
}
```

For a detailed description of the function used in this example, refer to *File system configuration functions* on page 59.

8.1.2 System configuration

8.1.2.1 FS_X_GetTimeDate()

Description

Returns the current time and date.

Prototype

```
U32 FS_X_OS_GetTimeDate(void);
```

Return value

Current time and date as U32 in a format suitable for the file system.

Additional Information

The format of the time is arranged as follows:
Bit 0-4: 2-second count (0-29)
Bit 5-10: Minutes (0-59)

Bit 11-15: Hours (0-23)
Bit 16-20: Day of month (1-31)
Bit 21-24: Month of year (1-12)
Bit 25-31: Number of years since 1980 (0-127)

Example

```
U32 FS_X_GetTimeDate(void) {
  U32 r;
  U16 Sec, Min, Hour, Day, Month, Year;

  Sec   = FS_X_GET_SECOND();
  Min   = FS_X_GET_MINUTE();
  Hour  = FS_X_GET_HOUR();
  Day   = FS_X_GET_DAY();
  Month = FS_X_GET_MONTH();
  Year  = FS_X_GET_YEAR();

  r   = Sec / 2 + (Min << 5) + (Hour  << 11);
  r  |= (Day + (Month << 5) + (Year  << 9)) << 16;
  return r;
}
```

8.1.2.2 FS_X_Panic()

Description

Handler for unrecoverable errors.

Prototype

```
void FS_X_Panic(int ErrorCode);
```

Parameter	Description
ErrorCode	Type of fatal error.

Table 8.1: FS_X_Panic() parameter list

Additional Information

Typically, the function is called when the file system runs out of memory or when invalid parameters are passed to some API functions. Compiled in only when debugging is turned on (FS_DEBUG_LEVEL greater than 0). The default implementation is an endless loop.

8.1.2.3 Logging functions

Logging is used in higher debug levels only. The typical target build does not use logging and does therefore not require any of the logging functions. For a release build without logging the functions may be eliminated from configuration file to save some space. (If the linker is not function aware and eliminates unreferenced functions automatically). Refer to the chapter *Debugging* on page 491 for further information about the different logging functions.

8.2 Compile time configuration

The following types of configuration macros exist:

Binary switches “B”

Switches can have a value of either 0 or 1, for deactivated and activated respectively. Actually, anything other than 0 works, but 1 makes it easier to read a configuration file. These switches can enable or disable a certain functionality or behavior. Switches are the simplest form of configuration macros.

Numerical values “N”

Numerical values are used somewhere in the code in place of a numerical constant. A typical example is the configuration of the sector size of a storage medium.

8.2.1 General file system configuration

Type	Macro	Default	Description
B	FS_SUPPORT_FAT	1	Defines if emFile should use the FAT file system layer.
B	FS_SUPPORT_EFS	0	Defines if emFile should use the optional EFS file system layer.
B	FS_SUPPORT_CACHE	1	Determines whether FS_AssignCache() can be used. FS_AssignCache() allows runtime assignment of a cache. Refer to *FS_AssignCache()* on page 201 for further information. **Note:** FS_AssignCache() needs to be called to activate the cache functionality for a specific device.
B	FS_MULTI_HANDLE_SAFE	0	If you intend to open a file simultaneously for read/write, set this macro to 1.
String	FS_DIRECTORY_DELIMITER	'\\'	Defines the character/string that should be used to delimit directories in a path.
N	FS_DRIVER_ALIGNMENT	4	Defines the minimum alignment in bytes a driver needs.
B	FS_USE_FILE_BUFFER	1	Disables/Enables file buffer support. File buffers make file access faster when reading/writing files in small chunks. When using file buffers, emFile requires a bit more ROM and RAM. By default, file buffers are enabled in emFile, but not used, since the buffer size has to be configured before they can be used. For more information about how to configure the file buffers, please refer to *FS_ConfigFileBufferDefault()* on page 62.
B	FS_SUPPORT_DEINIT	0	Allows to deinitialize the file system. This can be useful when device may not longer use the file system and the resources shall be used for other purposes. *ON:* FS_DeInit() is enabled and will free all resource that have been used, including all memory block that have been used. For more information about FS_DeInit() please refer to *FS_DeInit()* on page 52. *OFF:* FS_DeInit() is disabled and therefore resources are not freed.

Table 8.2: General file system configuration macros

Type	Macro	Default	Description
B	FS_SUPPORT_EXT_MEM_MANAGER	0	Defines whether the internal or an external memory allocation function should be used. *ON:* The file system shall use external memory allocation routines. These routines shall be set by calling the function FS_SetMemHandler(). *OFF:* The internal memory allocation routines of the file system should be used.
B	FS_VERIFY_WRITE	0	Verify every write sector operation (tests the driver and hardware). This switch should always be off for production code. It is normally switched on only when investigating driver problems.
B	FS_SUPPORT_CHECK_MEMORY	0	Selects if the access to data buffers passed to device driver should be checked for 0-copy operations. The check is performed by a callback function registered by invoking FS_SetMemAccessCallback(). *ON:* On each read/write operation the registered callback function is invoked to check if a 0-copy operation can be performed. *OFF:* No checking is performed. If possible, the data buffer is passed directly to device driver (0-copy operations).

Table 8.2: General file system configuration macros

8.2.2 FAT configuration

The current version of emFile supports FAT12/FAT16/FAT32.

Type	Macro	Default	Description
B	FS_FAT_SUPPORT_FAT32	1	To enable support for FAT32 media, define this macro to 1.
B	FS_FAT_USE_FSINFO_SECTOR	1	When retrieving the free disk amount on large FAT32 volumes, this may take a long time, since the FAT table can extend to many Mbytes. To improve this, this macro should be set to 1. This will enable the feature of using the FAT32 specific FSInfo sector. This sector stores the information of the free clusters that are available and the last known free cluster. *ON:* Higher speed, Bigger code. *OFF:* Lower speed, Smaller code.
B	FS_FAT_OPTIMIZE_DELETE	1	When deleting a large contiguous file on a FAT system, it may take some time to delete the FAT entries for the file. This macro set to 1 enables a sequence to accelerate this operation. *ON:* Higher speed, Bigger code. *OFF:* Lower speed, Smaller code.
B	FS_FAT_SUPPORT_UTF8	0	When using the LFN package, the file/directory name is stored as Unicode string. This macros enables the support for accessing such files and directories, where characters in the file/directory name are others than the standard Latin characters such as Greek or Cyrillic. To open such a file the string should be UTF-8 encoded.
B	FS_MAINTAIN_FAT_COPY	0	Enables the update of the second FAT allocation table.

Table 8.3: FAT configuration macros

8.2.3 EFS configuration

Type	Macro	Default	Description
B	FS_EFS_CASE_SENSITIVE	0	If EFS file/directory operations should be case sensitive, define this macro to 1.

Table 8.4: EFS configuration macros

8.2.4 OS support

emFile can be used with operating systems. For no OS support at all, set all of them to 0. If you need support for an additional OS, you will have to provide functions described in the chapter *OS integration* on page 481.

Type	Macro	Default	Description
N	FS_OS_LOCKING	0	Set this to 1 determines that an operating system should be used. When using an operating system, generally every file system operation is locked by a semaphore. When this macro is defined to 1 only one lock is used to lock each file system function (Coarse lock granularity). If FS_OS_LOCKING is defined to 2 the file system locks on every critical file system operation. (Fine lock granularity). Fine lock granularity requires more semaphores.

Table 8.5: Operating system support macros

Default setting of emFile is not configured for a multitasking environment.

8.2.5 Debugging

emFile can be configured to generate useful debug information which can help you analyze a potential problem. You can control the amount of generated information by changing the value of the FS_DEBUG_LEVEL define.

The following table lists the permitted values for FS_DEBUG_LEVEL:

Value	Symbolic name	Explanation
0	FS_DEBUG_LEVEL_NOCHECK	No runtime checks are performed.
1	FS_DEBUG_LEVEL_CHECK_PARA	Parameter checks are performed to avoid crashes. (Default for target system)
2	FS_DEBUG_LEVEL_CHECK_ALL	Parameter checks and consistency checks are performed.
3	FS_DEBUG_LEVEL_LOG_ERRORS	Errors are recorded.
4	FS_DEBUG_LEVEL_LOG_WARNINGS	Errors and warnings are recorded. (Default for PC-simulation)
5	FS_DEBUG_LEVEL_LOG_ALL	Errors, warnings and messages are recorded.

Table 8.6: Debug level macros

emFile outputs the debug information in text form using logging routines (see *Debugging* on page 491). These routines can be left empty as they are not required for the proper function of emFile. This is typically the case for release (production) builds which usually use the lowest debug level.

The following table lists the logging functions and on which debug level they are active:

Function	Debug level	Explanation
FS_X_ErrorOut()	>= 3	Fatal errors.
FS_X_Warn()	>= 4	Warnings.
FS_X_Log()	>= 5	Execution trace.

Table 8.7: Logging functions

8.2.6 Miscellaneous configurations

Type	Macro	Default	Description
B	FS_NO_CLIB	0	Setting this macro to 1, emFile does not use the standard C library functions (such as strcmp() etc.) that come with the compiler.

Table 8.8: Miscellaneous configuration macros

8.2.7 Sample configuration

The emFile configuration file `FS_Conf.h` is located in the `\Config` directory of your shipment. emFile compiles and runs without any problem with the default settings. If you want to change the default configuration, insert the corresponding macros in the delivered `FS_Conf.h`.

```
/*********************************************************************
*                SEGGER MICROCONTROLLER GmbH & Co. KG                *
*        Solutions for real time microcontroller applications        *
**********************************************************************
*                                                                    *
*        (c) 2002 - 2007  SEGGER MICROCONTROLLER GmbH & Co. KG       *
*                                                                    *
*        Internet: www.segger.com    Support:  support@segger.com    *
*                                                                    *
**********************************************************************

**** emFile file system for embedded applications ****
emFile is protected by international copyright laws. Knowledge of the
source code may not be used to write a similar product. This file may
only be used in accordance with a license and should not be re-
distributed in any way. We appreciate your understanding and fairness.

----------------------------------------------------------------------
File        : FS_Conf.h
Purpose     : emFile compile-time configuration settings
---------------------------END-OF-HEADER------------------------------
*/
#ifndef _FS_CONF_H_
#define _FS_CONF_H_

#define FS_DEBUG_LEVEL        1
#define FS_MAX_SECTOR_SIZE  512

#endif  /* Avoid multiple inclusion */
```


Chapter 9

OS integration

emFile is suitable for any multithreaded environment. To ensure that different tasks can access the file system concurrently, you need to implement a few operating system-dependent functions.

For embOS and MS Windows, you will find implementations of these functions in the file system's source code. This chapter provides descriptions of the functions required to fully support emFile in multithreaded environments. If you do not use an OS, or if you do not make file access from different tasks, you can leave these functions empty.

You may also add date and time support functions for use by the FAT file system. The sample implementations provided with emFile use ANSI C standard functions to obtain the correct date and time.

9.1 OS layer API functions

To use emFile with an operating system set the define `FS_OS_LOCKING` to `1` for coarse lock granularity (or alternatively to 2 for file lock granularity) in `FS_Conf.h`. Set this to 1 determines that an operating system should be used. When using an operating system, generally every file system operation is locked by a semaphore. When this macro is defined to 1 only one lock is used to lock each file system function (Coarse lock granularity). If `FS_OS_LOCKING` is defined to 2 the file system locks on every critical file system operation. (Fine lock granularity). Fine lock granularity requires more semaphores. You have to implement the following functions to integrate emFile into your operating system. Samples for the implementation of an operating system can be found in the directory `\Sample\OS\`.

Example

```
/*********************************************************************
*                 SEGGER MICROCONTROLLER GmbH & Co. KG               *
*        Solutions for real time microcontroller applications        *
**********************************************************************
*                                                                    *
*        (c) 2006         SEGGER MICROCONTROLLER GmbH & Co. KG       *
*                                                                    *
*        Internet: www.segger.com    Support:  support@segger.com    *
*                                                                    *
**********************************************************************

**** emFile file system for embedded applications ****
emFile is protected by international copyright laws. Knowledge of the
source code may not be used to write a similar product. This file may
only be used in accordance with a license and should not be re-
distributed in any way. We appreciate your understanding and fairness.
----------------------------------------------------------------------
File        : FS_Conf.h
Purpose     : File system configuration
---------------------------END-OF-HEADER------------------------------
*/

#ifndef _FS_CONF_H_
#define _FS_CONF_H_

#define FS_OS_LOCKING        1
#endif  /* Avoid multiple inclusion */
```


9.1.1 FS_X_OS_Init()

Description

Initializes the OS resources. Specifically, you will need to create at least NumLocks binary semaphores.

Prototype

```
void FS_X_OS_Init(unsigned NumLocks);
```

Parameter	Meaning
NumLocks	Number of binary semaphores/mutexes that should be created.

Table 9.1: FS_X_OS_Init() parameter list

Additional Information

This function is called by FS_Init(). You should create all resources required by the OS to support multithreading of the file system.

9.1.2 FS_X_OS_DeInit()

Description

Frees the OS resources.

Prototype

```
void FS_X_OS_DeInit(void);
```

Additional Information

This function is optional and is called by `FS_DeInit()` which is only available when FS_SUPPORT_DEINIT is set to 1. You should delete all resources what were required by the OS to support multithreading of the file system.

9.1.3 FS_X_OS_Lock()

Description

Locks a specific file system operation.

Prototype

```
void FS_X_OS_Lock(unsigned LockIndex);
```

Parameter	Meaning
LockIndex	Index number of the binary semaphore/mutex created before in FS_X_OS_Init().

Table 9.2: FS_X_OS_Lock() parameter list

Additional Information

This routine is called by the file system before it accesses the device or before using a critical internal data structure. It blocks other threads from entering the same critical section using a resource semaphore/mutex until FS_X_OS_Unlock() has been called with the same LockIndex.
When using a real time operating system, you normally have to increment a counting resource semaphore.

9.1.4 FS_X_OS_Unlock()

Description

Unlocks a file system operation.

Prototype

```
void FS_X_OS_Unlock(unsigned LockIndex);
```

Parameter	Meaning
LockIndex	Index number of the binary semaphore/mutex created before in FS_X_OS_Init().

Table 9.3: FS_X_OS_Unlock() parameter list

Additional Information

This routine is called by the file system after accessing the device or after using a critical internal data structure. When using a real time operating system, you normally have to decrement a counting resource semaphore.

9.1.5 FS_X_OS_Wait()

Description

Blocks the calling task for a specified time or until the file system event is triggered.

Prototype

```
void FS_X_OS_Wait(int Timeout);
```

Parameter	Meaning
Timeout	Number of OS ticks the function should wait for the event to be signaled.

Table 9.4: FS_X_OS_Wait() parameter list

Additional Information

The file system has only one event which is created at the initialization in the FS_X_OS_Init() function and later released in FS_X_OS_DeInit(). The event can be triggered by calling FS_X_OS_Signal(). This routine is not called directly by the file system but it can be used to implement event-driven HW layers in a portable way.

9.1.6 FS_X_OS_Signal()

Description

Signals the file system event.

Prototype

```
void FS_X_OS_Signal(void);
```

Additional Information

Typically, this routine is called from an interrupt to wake-up the task which is blocked in a call to `FS_X_OS_Wait()` function. This routine is not called directly by the file system but it can be used to implement event-driven HW layers in a portable way.

9.1.7 FS_X_OS_GetTime()

Description

Returns the number of OS ticks elapsed since the start of OS.

Prototype

```
U32 FS_X_OS_GetTime(void);
```

Return value

Number of OS ticks elapsed since the start of OS.

Additional Information

Typically, this function is called for performance measurements. An OS tick is usually 1ms long.

9.1.8 Examples

OS interface routine for embOS

The following example shows an adaptation for embOS (excerpt from file `FS_X_embOS.c` located in the folder `\FS\OS\`):

```
#include  "FS_Int.h"
#include  "FS_OS.h"
#include "RTOS.h"

static OS_RSEMA * _FS_Sema;

void FS_X_OS_Lock(unsigned LockIndex) {
  OS_RSEMA * pSema;

  pSema = _paSema + LockIndex;
  OS_Use(pSema);
}

void FS_X_OS_Unlock(unsigned LockIndex) {
  OS_RSEMA * pSema;

  pSema = _paSema + LockIndex;
  OS_Unuse(pSema);
}

void FS_X_OS_Init(unsigned NumLocks) {
  unsigned i;
  OS_RSEMA * pSema;

  _paSema = (OS_RSEMA *)FS_AllocZeroed(NumLocks* sizeof(OS_RSEMA));
  pSema =_paSema;
  for (i = 0; i < NumLocks; i++) {
    OS_CREATERSEMA(pSema++);
  }
}
```

OS interface routines for uC/OS

The following example shows an adaptation for µC/OS (excerpt from file `FS_X_uCOS_II.c` located in the folder `\Sample\OS\`):

```
#include  "FS_Int.h"
#include  "FS_OS.h"
#include  "ucos_ii.h"

static  OS_EVENT  **FS_SemPtrs;

void  FS_X_OS_Init (unsigned nlocks) {
    unsigned    i;
    OS_EVENT  **p_sem;

    FS_SemPtrs  = (OS_EVENT  **)FS_AllocZeroed(nlocks * sizeof(OS_EVENT *));
    p_sem       =  FS_SemPtrs;

    for(i = 0; i < nlocks; i++) {
       *p_sem   = OSSemCreate(1);
        p_sem  += 1;
    }
}


void  FS_X_OS_Unlock (unsigned index) {
    OS_EVENT  *p_sem;

    p_sem    = *(FS_SemPtrs + index);
    OSSemPost(p_sem);
}

void  FS_X_OS_Lock (unsigned index) {
    INT8U       err;
    OS_EVENT  *p_sem;

    p_sem    = *(FS_SemPtrs + index);
    OSSemPend(p_sem, 0, &err);
}
```


Chapter 10

Debugging

For debug purpose the functions in this chapter are helpful. The functions display information on a display or through a serial communication port.

10.1 FS_X_Log()

Description

Outputs debug information from emFile. This function has to integrated into your application if `FS_DEBUG_LEVEL >= 5`. Refer to section *Debugging* on page 479 of the Configuration chapter for further information about the different debug-level.

Prototype

```
void FS_X_Log (const char * s);
```

Parameter	Meaning
s	Pointer to the string to be sent.

Table 10.1: FS_X_Log() parameter list

Example

```
/* sample using ANSI C printf function */

U16 FS_X_Log(const char* s) {
  printf("%s", s);
}
```


10.2 FS_X_Warn()

Description

Outputs warnings from emFile. This function has to integrated into your application if `FS_DEBUG_LEVEL >= 4`. Refer to section *Debugging* on page 479 of the Configuration chapter for further information about the different debug-level.

Prototype

```
void FS_X_Warn (const char * s);
```

Parameter	Meaning
s	Pointer to the string to be sent.

Table 10.2: FS_X_Warn() parameter list

Example

```
/* sample using ANSI C printf function */

U16 FS_X_Warn(const char* s) {
  printf("%s", s);
}
```


10.3 FS_X_ErrorOut()

Description

Outputs errors from emFile. This function has to integrated into your application if `FS_DEBUG_LEVEL >= 3`. Refer to section *Debugging* on page 479 of the Configuration chapter for further information about the different debug-level.

Prototype

```
void FS_X_ErrorOut (const char * s);
```

Parameter	Meaning
s	Pointer to the string to be sent.

Table 10.3: FS_X_ErrorOut() parameter list

Example

```
/* sample using ANSI C printf function */

U16 FS_X_ErrorOut(const char* s) {
  printf("%s", s);
}
```


10.4 Troubleshooting

If you are used to C-like file operations, you already know the `fopen()` function. In emFile, there is an equivalent function called `FS_FOpen()`. You specify a name, an access mode and if this kind of file access is allowed and no error occurs, you get a pointer to a file handle in return. For more information about the parameters refer to *FS_FOpen()* on page 73: Open a file

```
FS_FILE * pfile;
pfile = FS_FOpen("test.txt","r");
if (pFile == 0) {
  return -1; /* report error */
} else {
  return 0; /* file system is up and running! */
}
```

If this pointer is zero after calling `FS_FOpen()`, there was a problem opening the file. There are basically some common reasons why this could happen:

- The file or path does not exist
- The drive could not be read or written
- The drive contains an invalid BIOS parameter block or partition table

These faults can be caused by corrupted media. To verify the validity of your medium, either check if the medium is physically okay or check the medium with another operation system (for example Windows).

But there are also faults that are relatively seldom but also possible:

- A compiler/linker error has occurred
- Stack overflow
- Memory failure
- Electro-magnetic influence (EMC, EMV, ESD)

To find out what the real reason for the error is, you may just try reading and writing a raw sector. Here is an example function that tries writing a single sector to your device. After reading back and verifying the sector data, you know if sectored access to the device is possible and if your device is working.

```
#define BYTES_PER_SECTOR    512   // Should match the sector size of storage medium

int WriteSector(void) {
  U8  acBufferOut[BYTES_PER_SECTOR];
  U8  acBufferIn[BYTES_PER_SECTOR];
  U32 SectorIndex;
  int r;
  int i;

  //
  // Do not write on the first sectors. They contain
  // information about partitioning and media geometry.
  //
  SectorIndex = 80;
  //
  // Fill the buffer with data.
  //
  for (i = 0; i < BYTES_PER_SECTOR; i++) {
    acBufferOut[i] = i % 256;
  }
  //
  // Write one sector.
  //
  r = FS_STORAGE_WriteSector("", acBufferOut, SectorIndex);
  if (r) {
    FS_X_Log("Cannot write to sector.\n");
    return -1;
  }
  //
  // Read back the sector contents.
```



```
  //
  r = FS_STORAGE_ReadSector("", acBufferIn, SectorIndex);
  if (r) {
    FS_X_Log("Cannot read from sector.\n");
    return -1;
  }
  //
  // Compare the sector contents.
  //
  for (i = 0; i < BYTES_PER_SECTOR; i++) {
    if (acBufferIn[i] != acBufferOut[i]) {
      FS_X_Log("Sector not correctly written.\n");
      return -1;
    }
  }
  return 0;
}
```

If you still receive no valid file pointer although the sectors of the device is accessible and other operating systems report the device to be valid, you may have to take a look into the running system by stepping through the function `FS_FOpen()`.

Chapter 11

Performance and resource usage

11.1 Memory footprint

The file system is designed to fit many kinds of embedded design requirements. Several features can be excluded from a build to get a minimal system that can efficiently access any FAT media.

The operation area of emFile is very different and the memory requirements (RAM and ROM) differs in depending on the used features. The following section will show the memory requirements of different modules which are used in typical applications.

Note that the values are valid for the given configuration. Features can affect the size of others. For example, if FAT32 is deactivated, the format function gets smaller because the 32 bit specific part of format is not added into the compilation.

11.1.1 System

The following table shows the hardware and the toolchain details of the project:

Detail	Description
CPU	ARM7
Tool chain	IAR Embedded Workbench for ARM V4.41A
Model	ARM7, Thumb instructions; no interwork;
Compiler options	Highest size optimization;
Device driver	Empty dummy driver. For information about the memory usage of a specific emFile device driver refer to the Unit number section of the respective driver in the *Device drivers* on page 213.

Table 11.1: ARM7 sample configuration

11.1.2 File system configuration

The following excerpts of `FS_Conf.h` shows the used configuration options:

```
#define FS_OS_LOCKING               0 // Disable OS support
#define FS_SUPPORT_FAT              1 // Support the FAT file system if enabled
#define FS_DEBUG_LEVEL              0 // Set debug level
```

11.1.3 Sample project

We use the following code to calculate the memory resources of commonly used functions. You can easily reproduce the measurement when you compile the following sample. Build the application listed below and generate a linker listing to get the memory requirements of an application which only includes startup code and the empty `main()` function. Afterwards, set the value of the macro `STEP` to 1 to get the memory requirement of the minimum file system. Subtract the ROM requirements from `STEP==0` from the ROM requirements of `STEP==1` to get the exact ROM requirements of a minimal file system. Increment the value of the macro STEP to include more file system functions and repeat your calculation.

```
#include "FS.h"'
#include "FS_Int.h"

/*********************************************************************
*
*       defines, configurable
*
**********************************************************************/
#define STEP    0     // Change this line to adjust which portions of code are linked

/*********************************************************************
*
*       Public code
```



```
*
**********************************************************************/

/*********************************************************************
*
*       main
*/
void main(void) {
#if STEP >= 1            // Step 1: Minimum file system
  FS_FILE    * pFile;
  FS_Init();
  pFile = FS_FOpen("File.txt", "w");
#endif
#if STEP >= 2           // Step 2: Write a file
  FS_Write(pFile, "Test", 4);
#endif
#if STEP >= 3           // Step 3: Remove a file
  FS_Remove("File.txt");
#endif
#if STEP >= 4           // Step 4: Open a directory
  FS_FIND_DATA fd;
  FS_FindFirstFile(&fd, "\\YourDir\\", "File.txt", 8);
  FS_FindClose(&fd);
#endif
#if STEP >= 5          // Step 5: Create a directory
  FS_MkDir ("");
#endif
#if STEP >= 6          // Step 6: Add long file name support
  FS_FAT_SupportLFN();
#endif
#if STEP >= 7          // Step 7: Low-level format a medium
  FS_FormatLow("");
#endif
#if STEP >= 8          // Step 8: High-level format a medium
  FS_Format("", NULL);
#endif
#if STEP >= 9          // Step 9: Assign cache - Cache module: FS_CACHE_ALL
  FS_AssignCache("", NULL, 0, FS_CACHE_ALL);
//  FS_AssignCache("", NULL, 0, FS_CACHE_MAN);
//  FS_AssignCache("", NULL, 0, FS_CACHE_RW);
//  FS_AssignCache("", NULL, 0, FS_CACHE_RW_QUOTA);
#endif
#if STEP >= 10         // Step 10: Checkdisk
  FS_FAT_CheckDisk("", NULL, 0, 0, NULL);
#endif
#if STEP >= 11         // Step 11: Get device info
  FS_GetDeviceInfo("", NULL);
#endif
#if STEP >= 12         // Step 12: Get the size of a file
  FS_GetFileSize(NULL);
#endif
#if STEP >= 1            // Step 1: Minimum file system
  FS_FClose(pFile);
#endif
}
```


11.1.4 Static ROM requirements

The following table shows the ROM requirement of the used functions:

Description	ROM [Kbytes]
Step 1: File system core (without driver) Contains the following functionality: Init / Configuration Open file	7.0
Step 2: Read file	1.1
Step 3: Write file	1.1
Step 4: Remove file	0.1
Step 5: Open directory	0.5
Step 6: Create directory	0.5
Step 7: Long file name support	2.0
Step 8: Low-level format a medium	0.2
Step 9: High-level format a medium	1.8
Step 10: Assign a cache - `FS_CACHE_ALL`	0.4
Assign a cache - `FS_CACHE_MAN`	0.7
Assign a cache - `FS_CACHE_RW`	0.7
Assign a cache - `FS_CACHE_RW_QUOTA`	1.0
Step 11: Checkdisk	3.3
Step 12: Get device info	0.1
Step 13: Get the size of a file	0.1

Summary

A simple system will typically use around 10 KByte of ROM. To compute the overall ROM requirements, the ROM requirements of the driver need to be added.

11.1.4.1 ROM requirements for long filename support

This section describes the additional ROM usage of emFile if the long filename support is used. Please note that long filename support is not part of the emFile FAT packet, but is sold separately.

Module	ROM [Kbytes]
emFile LFN	2.2

RAM requirements for long filename support

The long filename support of emFile does not require any additional RAM.

11.1.5 Static RAM requirements

The static RAM requirement of the file system without any driver is around 150 bytes.

11.1.6 Dynamic RAM requirements

During the initialization emFile will dynamically allocate memory depending on the number of added devices, the number of simultaneously opened files and the OS locking type:

Type	Size [Bytes]	Count
FS_FILE	16	Maximum number of simultaneously open files. Depends on application, minimum is 1.
FS_FILE_OBJ	40	Maximum number of simultaneously open files. Depends on application, minimum is 1.
FS_VOLUME	88	Number of FS_AddDevice() calls.
No operating system should be used: FS_OS_LOCKING == 0		
FS_SECTOR_BUFFER	8 + SectorSize By default, SectorSize is 512 bytes.	2
OS should be used: FS_OS_LOCKING == 1		
FS_SECTOR_BUFFER	8 + SectorSize By default, SectorSize is 512 bytes.	2
OS_LOCKS	sizeof(SEMAPHORE)	1
OS should be used. Every critical operations is locked: FS_OS_LOCKING == 2		
FS_SECTOR_BUFFER	8 + SectorSize By default, SectorSize is 512 bytes.	2 * Number of used drivers
OS_LOCKS	1 + sizeof(SEMAPHORE)	Number of used drivers
DRIVER_LOCK_TABLE	16	Number of used drivers

Note: FS_FILE and FS_FILE_OBJ structures can be allocated even after initialization depending on how many files are simultaneously opened.

11.1.7 RAM usage example

For example, a small file system application with the following configuration:

- only one file is opened at a time
- no operating system support
- using the SD card driver

requires approximately 1300 bytes.

11.2 Performance

A benchmark is used to measure the speed of the software on available targets. This benchmark is in no way complete, but it gives an approximation of the length of time required for common operations on various targets. You can find the measurement results in the chapter describing the individual drivers.

11.2.1 Description of the performance tests

The performance tests are executed as described and in the order below.
Performance test procedure:

1. Format the drive.
2. Create and open a file for writing.
   W: Start measuring of write performance.
3. Write a multiple of 8 Kbytes.
   W: Stop measuring of write performance.
4. Close the file
5. Reopen the file.
   R: Start measuring of read performance.
6. Read a multiple of 8 Kbytes.
   R: Stop measuring of read performance.
7. Close the file
8. Show the performance results.

The performance tests can be reproduced. Include `FS_PerformanceSimple.c` (located in the folder `.\Sample\API`) into your project. Compile and run the project on your target hardware.

11.2.2 How to improve the performance

If you find that the performance of emFile on your hardware is not what you expect there are several ways you can improve it.

Use the proper write mode

The default behavior of the file system is to update the allocation table and the directory entry after each write operation. If several write operations are performed between the opening and the closing of the file it is recommended to set the write mode to `FS_WRITEMODE_MEDIUM` or `FS_WRITEMODE_FAST` to increase the performance. In these modes the allocation table and the directory entry are updated only when the file is closed. Refer to *FS_SetFileWriteMode()* on page 68 to learn how the write mode can be configured. Please note that these write modes can not be used when the journaling is enabled.

Write multiple of sector size

The file system implements a 0-copy mechanism in which the data written to a file using the FS_FWrite() and FS_Write() functions is passed directly to the device driver if the data is written at a sector boundary and the number of bytes written is a multiple of sector size. In any other case the file system uses a read-modify-write operation which increases the number of I/O operations and reduces the performance. The file system makes sure that the contents of a file always begins at a sector boundary.

Use the file buffer

It is recommended to activate the file buffering when the application reads and writes amounts of data smaller than the sector size. Refer to *FS_ConfigFileBufferDefault()* on page 62 to see how this can be done. The file buffer is a small cache which helps reducing the number of times storage medium is accessed and thus increasing the performance. Please note that a file buffer in write mode can not be used when the journaling is enabled.

Use a sector cache

The sector cache can be enabled to increase the overall performance of the file system. For more information refer to *Optimizing performance - Caching and buffering* on page 197.

Configure a read-ahead driver

The read-ahead driver is useful when a storage medium is used which is more efficient when several sectors are read or written at once. This includes storage media such as CompactFlash cards, SD and MMC cards and USB sticks. Normally, the file system reads the allocation table one sector at a time. If configured, the file system activates the read-ahead driver at run time when the allocation table is accessed reducing, for example, the time it takes to determine the amount of free space. For more information refer to *Sector read-ahead driver* on page 459.

Optimize the hardware layer

Ensure that the routines of the hardware layer are fast. How you do that depends on your compiler. Some compilers have the option to define a function as running from RAM which is faster compared to running it from flash.

Use the FS_OPTIMIZE macro

The definitions of time critical functions in emFile are prefixed with the macro FS_OPTIMIZE. As default it expands to nothing. You can use this macro to enable the compiler optimization only for these functions. How you define this macro depends on your compiler.

Chapter 12

Journaling (Add-on)

This chapter documents and explains emFile's journaling add-on. Journaling is an extension to emFile that makes the file system layer fail-safe.

12.1 Introduction

emFile Journaling is an additional component which sits on top of the file system and makes the file system layer fail-safe. File systems without journaling support (for example, FAT) are not fail-safe. Journaling means that a file system logs all changes to a journal before committing them to the main file system.

Driver fail-safety

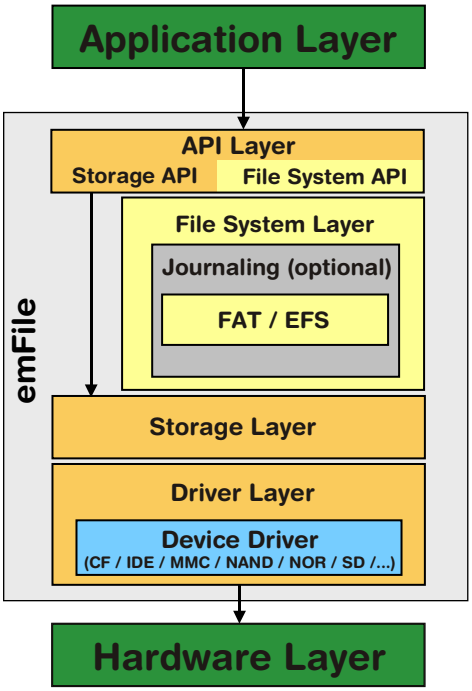
Data can be lost in case of unexpected Reset in either the file system Layer (FAT or EFS) or in the driver layer. The entire system is fail-safe only if BOTH layers are fail-safe. The journaling add-on makes only the file system layer fail-safe. For fail-safety of the driver layer, refer to *Device drivers* on page 213.

12.2 Features

- Non fail-safe file systems will be fail-safe.
- Fully compatible to standard file system implementations (e.g. FAT)
- Every storage solution can be used.
  No reformat required.
- Multiple write accesses to the storage medium can be combined in user application.

12.3 Backgrounds

emFile is typically used with non fail-safe file systems like FAT. Loss of data can occur in either the driver layer or the file system layer. The driver layer is typically fail-safe so the only place for typical data loss is the file system layer. The file system can be corrupted through an interrupted write access for example in the event of power failure or system crash. This is by design of FAT and true for all implementations from any vendor. The emFile journaling add-on adds journaling to the file system layer.



The goal of this additional layer is to guarantee a file system that is always in a consistent state. Operations on File System Layer are mostly not atomic. For example, a single call of `FS_FWrite()` to write data into a new file causes the execution of the following three Storage Layer operations:

1. Allocate cluster and update FAT
2. Write user data
3. Update directory entry

An unexpected interrupt (such as a power failure) in this process can corrupt the file system. To prevent such corruptions the Journaling Layer stores every write access to achieve an always consistent state of the file system. All changes to the file system are stored in a journal. The data stored in the journal is copied into the file system only if the File System Layer operation has been finished without interruption. This procedure guarantees an always consistent state of the file system, because an interruption of the copy process leads not to data loss. The interrupted copy process will be restarted after a restart of the target.

12.3.1 File System Layer error scenarios

The following table lists the possible error scenarios:

	Moment of error	State	Data
1.	Journal empty.	Consistent	---
2.	While writing into journal.	Consistent	Lost
3.	While finalizing of the journal.	Consistent	Lost
4.	After finalization.	Consistent	Obtained

Table 12.1: Error scenarios

	Moment of error	State	Data
5.	While copying from journal into file system.	Consistent	Obtained
6.	After copy process, before invalidating of the journal.	Consistent	Obtained
7.	While invalidating of the journal.	Consistent	Obtained

Table 12.1: Error scenarios

12.3.2 Write optimization

The journaling add-on has been optimized for write performance. When data is written at the end of a file, which is usually the case in the most applications, only the management information (allocation table and directory entry) goes through journal. The file contents are written directly to final destination on the storage medium, thus improving the write performance. The fail-safety of the File System Layer is not affected as the file contents overwrites storage blocks which are not allocated to any file. The optimization is disabled as soon as a file or directory is deleted during a journaling transaction. This is done in order to make sure that the data of the deleted file or directory is contained in case of an unexpected reset.

12.4 How to use journaling

12.4.1 What do I need to do to use journaling?

Using journaling is very simple from a user perspective.

You have to

1. Enable journaling in the emFile configuration.
   Refer to *Configuration* on page 512 for detailed information.
2. Call `FS_JOURNAL_Create()` after formatting the volume.
   Refer to *FS_JOURNAL_Create()* on page 515 for detailed information.

That's it. Everything else is done by the emFile Journaling extension.

12.4.2 How can I use journaling in my application?

Journaling can also be used in your application. You can combine multiple write accesses in your application. Start the section that should use the journal with a call of `FS_JOURNAL_Begin()` and finish the section with a call of `FS_JOURNAL_End()` to assure that only all write operations of the section or non will be executed.

Example

```
void FailSafeSample(void) {
  FS_FILE * pFile;

  //
  // Create journal on first device of the volume.
  // Size: 200 KBytes.
  //
  FS_JOURNAL_Create("", 200 * 1024);
  //
  // Begin an operations which have to be be fail-safe.
  // All following steps will be stored into journal.
  //
  FS_JOURNAL_Begin("");
  pFile = FS_FOpen("File001.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Test...", 7);
    FS_FClose(pFile);
  }
  pFile = FS_FOpen("File002.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Another Test...", 15);
    FS_FClose(pFile);
  }
  //
  // End an operation which has to be be fail-safe.
  // Data will be copied from journal into file system.
  //
  FS_JOURNAL_End("");
}
```

12.4.3 Keeping the consistency of file contents

The journaling can be used to make sure that the contents of the whole file are consistent. This means that after recovering from an unexpected reset, which occurred during the writing, the file will contain either the previous data or the new data but not a combination of both. It also applies to the particular case were the application writes to an empty file. The only condition that must be satisfied is for the file system to generate 1 journaling transaction. This can be realized by surrounding the write operation within `FS_JOURNAL_Begin()`/`FS_JOURNAL_End()`. Here is an example code:


```
void WriteWholeFileSample1(void) {
  U8        aBuffer[128];
  FS_FILE * pFile;

  pFile = FS_FOpen("File.txt", "w");
  if (pFile) {
    FS_JOURNAL_Begin("");
    memset(aBuffer, 'a', sizeof(aBuffer));
    FS_Write(pFile, aBuffer, sizeof(aBuffer));
    memset(aBuffer, 'b', sizeof(aBuffer));
    FS_Write(pFile, aBuffer, sizeof(aBuffer));
    //
    // Changes are committed in a single journaling transaction
    // assuming the journal is large enough to store all the changes.
    //
    FS_JOURNAL_End("");
    FS_FClose(pFile);
  }
}
```

The advantage of this method is that a relatively small writing buffer can be used.

Another possibility is to use a buffer large enough to store the contents of the entire file and to call FS_FWrite()/FS_Write() 1 time with this buffer as parameter. Example code follows:

```
void WriteWholeFileSample2(void) {
  U8        aBuffer[256];
  FS_FILE * pFile;

  pFile = FS_FOpen("File.txt", "w");
  if (pFile) {
    memset(aBuffer, 'a', 128);
    memset(&aBuffer[128], 'b', 128);
    //
    // Upon function return the changes are committed
    // in a single journaling transaction assuming the journal is large enough.
    //
    FS_Write(pFile, aBuffer, sizeof(aBuffer));
    FS_FClose(pFile);
  }
}
```

Both methods require that the journaling is large enough to store all the changes made to the storage medium. When writing to a file which is not empty the journal should be able to store the management data and the file contents. If the file is empty a smaller journal is required to store only the management data. In this case the contents of the file are written directly to the place on the storage where the data should actually be stored.

12.5 Configuration

The following types of configuration macros exist:

Binary switches “B”

Switches can have a value of either 0 or 1, for deactivated and activated respectively. Actually, anything other than 0 works, but 1 makes it easier to read a configuration file. These switches can enable or disable a certain functionality or behavior. Switches are the simplest form of configuration macros.

12.5.1 Journaling file system configuration

The configuration of emFile can be changed via compile time flags which can be added to `FS_Conf.h`. `FS_Conf.h` is the main configuration file for the file system.

Type	Macro	Default	Description
B	`FS_SUPPORT_JOURNAL`	0	Defines if emFile should enable journaling for the used file system.

Table 12.2: Journaling configuration macros

For detailed information about the configuration of emFile, refer to *Configuration of emFile* on page 471.

12.5.2 Journaling and write caching

It is recommended to disable any form of write caching when the journaling is enabled. A write cache buffers temporarily the data in RAM to increase the write throughput. If a power failure occurs, the data stored in the write cache is lost. After the target restart, the journaling is no more be able to recover the data of the last write operation. The optimal operation of journaling is achieved if the file system is configured as follows:

- Write cache should be disabled.
  If your application uses a cache, make sure it is a read cache. This is always true for the `FS_CACHE_ALL` and the `FS_CACHE_MAN` cache types which are pure read caches. For the `FS_CACHE_RW` and `FS_CACHE_RW_QUOTA` cache types the `FS_CACHE_SetMode()` function must be called to configure them as read caches. For detailed information about the configuration and usage of caches, refer to *Optimizing performance - Caching and buffering* on page 197.
- Directory entries should be updated after each write.
  Call the `FS_ConfigUpdateDirOnWrite()` function with the `OnOff` parameter set to 1 to activate this feature.
- Write mode should be set to “safe”.
  To configure this, call the `FS_SetFileWriteMode()` function with the `WriteMode` parameter set to `FS_WRITEMODE_SAFE`.
- Write buffer should be disabled.
  Make sure the `FS_USE_FILE_BUFFER` define is set to 0.

12.6 Journaling API

The table below lists the available API functions within their respective categories.

Function	Description
FS_JOURNAL_Begin()	Start data caching in the journal.
FS_JOURNAL_Create()	Creates the journal.
FS_JOURNAL_CreateEx()	Creates the journal.
FS_JOURNAL_Disable()	Deactivates the journal.
FS_JOURNAL_Enable()	Activates the journal.
FS_JOURNAL_End()	End data caching in the journal.

Table 12.3: emFile Journaling API function overview

12.6.1 FS_JOURNAL_Begin()

Description

Starts the data buffering in the journal. This means all relevant data is written to the journal, instead of the "real destination".

Prototype

```
int FS_JOURNAL_Begin(const char * sVolumeName);
```

Parameter	Description
sVolumeName	IN: name of a volume. If not specified, the first device in the volume table will be used. OUT: ---

Table 12.4: FS_JOURNAL_Begin() parameter list

Return value

==0	Transaction opened
!=0	Error code indicating the failure reason Refer to *FS_ErrorNo2Text()* on page 177.

Example

Refer to *How can I use journaling in my application?* on page 510.

12.6.2 FS_JOURNAL_Create()

Description

Creates the journal.

Prototype

```
int FS_JOURNAL_Create(const char * sVolumeName,
                      U32          NumBytes);
```

Parameter	Description
sVolumeName	IN: name of a volume. If not specified, the first device in the volume table will be used. OUT: ---
NumBytes	Sets the size of the journal.

Table 12.5: FS_JOURNAL_Create() parameter list

Return value

== 0: O.K., journal created
== 1: O.K., journal already exists
else: Error code indicating the failure reason
Refer to *FS_ErrorNo2Text()* on page 177.

Additional information

The size of the journal file can be computed by using this formula:

```
JournalSize = 3 * BytesPerSector + (16 + BytesPerSector) * NumSectors
```

Parameter	Description
JournalSize	Size of the journal file in bytes. This value should be passed as second parameter to FS_JOURNAL_Create().
BytesPerSector	Size of the logical sector in bytes.
NumSectors	Number of sectors the journal should be able to store.

Table 12.6: Parameters for journal size computation

The number of sectors the journal file should be able to store depends on the file system operations performed by the application. The table below can be used to compute the number of sectors which are changed during a specific file system operation.

Function	Number of sectors
FS_FClose()	1 sector if the file has been modified else no sectors.
FS_FOpen()	1 sector when creating the file else no sectors.
FS_FWrite()	see FS_Write() below
FS_SyncFile()	1 sector if the file has been modified else no sectors.
FS_Write()	Uses the remaining free space on the journal. 2 sectors and about 9 percent are reserved for allocation table and directory entry updates. The remaining sectors are used to store the actual data. If more data is written as the free space in the journal file the operation is split into several journal transactions.
FS_Rename()	1 sector
FS_SetFileAttributes()	1 sector
FS_SetFileTime()	1 sector
FS_SetFileTimeEx()	1 sector
FS_MkDir()	2 sectors + SectorsPerCluster
FS_RmDir()	2 sectors
FS_SetVolumeLabel()	1 sector

Table 12.7: Number of sectors modified by API functions

Example

Refer to *How can I use journaling in my application?* on page 510.

12.6.3 FS_JOURNAL_CreateEx()

Description

Creates the journal.

Prototype

```
int FS_JOURNAL_CreateEx(const char * sVolumeName,
                        U32          NumBytes,
                        U8           SupportFreeSector);
```

Parameter	Description
sVolumeName	IN: name of a volume. If not specified, the first device in the volume table will be used. OUT: ---
NumBytes	Sets the size of the journal.
SupportFreeSector	Set to 1 if the storage driver should be informed about unused sectors.

Table 12.8: FS_JOURNAL_CreateEx() parameter list

Return value

== 0: O.K., journal created
== 1: O.K., journal already exists
else: Error code indicating the failure reason
Refer to *FS_ErrorNo2Text()* on page 177.

Additional information

Refer to *FS_JOURNAL_Create()* on page 515.

12.6.4 FS_JOURNAL_Disable()

Description

Deactivates the journal.

Prototype

```
int FS_JOURNAL_Disable(const char * sVolumeName);
```

Parameter	Description
sVolumeName	IN: name of a volume. If not specified, the first device in the volume table will be used. OUT: ---

Table 12.9: FS_JOURNAL_Disable() parameter list

Return value

==0 Journal disabled
!=0 Error code indicating the failure reason
Refer to *FS_ErrorNo2Text()* on page 177.

Additional information

Following the call to this function, the modifications made to storage medium are not fail-safe anymore. The function can be called at any time after the file system inital-ization. It closes any pending journal transactions and it does nothing if the journal is already disabled. The journal remains disabled if the corresponding volume is re-mounted but it is activated if file system re-initialized. The FS_JOURNAL_Enable() function can be used to explicitly activate the journal.

12.6.5 FS_JOURNAL_Enable()

Description

Activates the journal.

Prototype

```
int FS_JOURNAL_Enable(const char * sVolumeName);
```

Parameter	Description
sVolumeName	IN: name of a volume. If not specified, the first device in the volume table will be used. OUT: ---

Table 12.10: FS_JOURNAL_Enable() parameter list

Return value

==0 Journal enabled
!=0 Error code indicating the failure reason
Refer to *FS_ErrorNo2Text()* on page 177.

Additional information

Calling of this function is optional. The journal is activated by default when the file system is initialized.

12.6.6 FS_JOURNAL_End()

Description

Ends the data buffering in the journal. This means all journal data should be written to the real destination.

Prototype

```
int FS_JOURNAL_End(const char * sVolumeName);
```

Parameter	Description
sVolumeName	IN: name of a volume. If not specified, the first device in the volume table will be used. OUT: ---

Table 12.11: FS_JOURNAL_End() parameter list

Return value

==0	Transaction closed
!=0	Error code indicating the failure reason Refer to *FS_ErrorNo2Text()* on page 177.

Example

Refer to *How can I use journaling in my application?* on page 510.

12.7 Performance and resource usage

In this section the RAM (static and dynamic) and ROM resource usage of the journaling add-on is described.

12.7.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the journaling have been measured on a system as follows: ARM7, IAR Embedded workbench V5.50.1, Thumb mode, Size optimization.

Module	ROM [Kbytes]
emFile journal	1.9

12.7.2 Static RAM usage

Static RAM usage is the amount of RAM required by the journal module for static variables. The number of bytes can be seen in a compiler list file:

Static RAM usage of the journaling add-on: 16 bytes

12.7.3 Runtime (dynamic) RAM usage

Runtime (dynamic) RAM usage is the amount of RAM allocated by the journaling add-on at runtime. The amount required depends the journal size and on the number of volumes on which the journaling add-on is enabled.

The approximately runtime RAM usage for the journaling add-on can be calculated as follows:

```
MemAllocated = (JournalSize / (BytesPerSector + 16) * 4 + 56)
             * NumVolumes
```

Parameter	Description
MemAllocated	Number of bytes allocated.
JournalSize	Size of the journal file in bytes. This is the second argument specified in the call to FS_JOURNAL_Create().
BytesPerSector	Size of a file system sector in bytes.
NumVolumes	Number of volumes on which the journaling is active.

Table 12.12: Runtime RAM usage parameters for journaling add-on

12.7.4 Performance

These performance measurements are in no way complete, but they give an approximation of the length of time required for common operations on various targets. The tests were performed as described in *Performance* on page 502.

All values are in Kbytes/sec.

Device	CPU speed	Medium	W	R
NXP LPC2478	57.6MHz	SST39VF201 (1x16 bit, no "burst write")	5.6	2534
ST STM32F103	72MHz	M29W128 (1x16, with "write burst", 64 bytes)	18.5	7877

Table 12.13: Performance values for sample configurations

12.8 FAQs

Q: Can a journal be created when other files are already present on the disk?
A: Yes. The journal saves its data in a normal file called `Journal.dat`. If there is enough space left on the medium there is no problem to create the journal even if other files are present.

Q: Can a journal can be re-created?
A: Yes. Follow this procedure to recreate the journal:
- remove `Journal.dat` file
- unmount the file system
- mount the file system
- re-create the journal

Q: Can a journal be deleted?
A: Yes, by deleting the `Journal.dat` file.

Q: What if the journal isn't big enough?
A: If the journal is not big enough the data already stored on the journal is saved to the real location on the medium and an error message is generated.

Q: Can multiple tasks use a journal at the same time?
A: Yes, the journal is multitasking safe.

Q: Can `FS_JOURNAL_Begin()` and `FS_JOURNAL_End()` be nested?
A: Yes. There is a reference counter that is incremented with each invocation of `FS_JOURNAL_Begin()` and is decremented when `FS_JOURNAL_End()` is called. When the reference counter reaches zero the data is transferred from journal to the real destination on the medium and the journal is cleared.

Chapter 13

Encryption (Add-on)

This chapter documents and explains emFile's encryption add-on. Encryption is an extension to emFile which allows for the data to be stored in a secure way.

13.1 Introduction

emFile Encryption is an additional component which can be used to secure the data of the entire volume or of individual files. Without encryption support all the data is stored in a readable form. Using the encryption the data can be made unreadable before being stored using a key. Without the knowledge of the key it is not possible to make the data readable again.

13.2 Features

- Can be used with both FAT and EFS file systems.
- All storage types such as NAND, NOR, SD/MMC/CompactFlash cards are supported.
- Only minor changes of application are required.
- DES and AES with 128-bit and 256-bit key lengths are supported.
- Encryption of entire media or of individual files.
- A tool available to decrypt/encrypt files on a PC.

13.3 How to use encryption

13.3.1 What do I need to do to use file encryption?

Using file encryption is very simple from a user perspective.

You have to:

1. Enable file encryption in the emFile configuration.
   Refer to *Compile time configuration* on page 527 for detailed information.
2. Call `FS_CRYPT_Prepare()` to initialize an encryption object. It must be performed only once. Refer to *FS_CRYPT_Prepare()* on page 529 for detailed information.
3. Open a file and call `FS_SetEncryptionObject()` to assign the encryption object to file handle. Refer to *FS_SetEncryptionObject()* on page 532 for detailed information.

That's it. Everything else is done by the emFile Encryption extension.

Example

This sample function opens a file and writes a text message to it. The file contents is encrypted using the DES encryption algorithm. The changes required to an application to support encryption are marked in magenta.

```
void FileEncryptionSample(void) {
  FS_FILE               * pFile;
  const U8                aKey[8] = {1, 2, 3, 4};
  FS_CRYPT_OBJ            CryptObj;
  static FS_DES_CONTEXT   _Context;
  static int              _IsInited;

  //
  // Create the encryption object. It contains all the necessary information
  // for the encryption/decryption of data. This step must be performed only once.
  //
  if (_IsInited == 0) {
    FS_CRYPT_Prepare(&CryptObj, &FS_CRYPT_ALGO_DES, &_Context, 512, aKey);
    _IsInited = 1;
  }
  pFile = FS_FOpen("cipher.bin", "w");
  if (pFile) {
    //
    // Assign the created encryption object to file handle.
    //
    FS_SetEncryptionObject(pFile, &CryptObj);
    //
    // Write data to file using encryption.
    //
    FS_Write(pFile, "This message has been encrypted using SEGGER emFile.\n", 53);
    FS_FClose(pFile);
  }
}
```

13.3.2 How can I use volume encryption?

Refer to *Encryption driver* on page 456.

13.4 Compile time configuration

The following types of configuration macros exist:

Binary switches “B”

Switches can have a value of either 0 or 1, for deactivated and activated respectively. Actually, anything other than 0 works, but 1 makes it easier to read a configuration file. These switches can enable or disable a certain functionality or behavior. Switches are the simplest form of configuration macros.

The configuration of emFile can be changed via compile time flags which can be added to `FS_Conf.h`. `FS_Conf.h` is the main configuration file for the file system.

Type	Macro	Default	Description
B	FS_SUPPORT_ENCRYPTION	0	Specifies whether support for the file encryption should be compiled in or not.

Table 13.1: Encryption configuration macros

For detailed information about the configuration of emFile, refer to *Configuration of emFile* on page 471.

13.5 Encryption API

The table below lists the available API functions within their respective categories.

Function	Description
FS_CRYPT_Prepare()	Initializes an encryption object.
FS_CRYPT_Decrypt()	Decrypts data on PC.
FS_CRYPT_Encrypt()	Encrypts data on PC.
FS_SetEncryptionObject()	Assigns an encryption object to a file handle.

Table 13.2: emFile Encryption API function overview

13.5.1 FS_CRYPT_Prepare()

Description

Initializes an encryption object which contains all the informations necessary for the encryption/decryption of a file.

Prototype

```
void FS_CRYPT_Prepare(FS_CRYPT_OBJ             * pCryptObj,
                      const FS_CRYPT_ALGO_TYPE * pAlgoType,
                      void                     * pContext,
                      U32                        BytesPerBlock,
                      const U8                 * pKey);
```

Parameter	Description
pCryptObj	IN: --- OUT: Encryption object to be initialized
pAlgoType	IN: Type of encryption algorithm OUT: ---
pContext	IN: Context of encryption algorithm OUT: ---
BytesPerBlock	Number of bytes to encrypt at once.
pKey	IN: The encryption/decryption password OUT: ---

Table 13.3: FS_CRYPT_Prepare() parameter list

Additional information

The following encryption algorithms are defined:

Permitted values for the parameter pAlgoType	
FS_CRYPT_ALGO_DES	Data Encryption Standard with 56-bit key length
FS_CRYPT_ALGO_AES128	Advanced Encryption Standard with 128-bit key length
FS_CRYPT_ALGO_AES256	Advanced Encryption Standard with 256-bit key length

The pContext parameter points to a structure of type FS_DES_CONTEXT when the FS_CRYPT_ALGO_DES algorithm is used or to FS_AES_CONTEXT structure when the FS_CRYPT_ALGO_AES128 or the FS_CRYPT_ALGO_AES256 are specified. The context pointer is saved to object structure and must point to a valid memory location as long as the encryption object is in use.

The BytesPerBlock parameter is a power of 2 value which should be smaller than or equal to the sector size of the volume which stores the file. A block size of 512 bytes is a good value.

The size of pKey byte depends on the algorithm type:

Algorithm type	pKey size [bytes]
FS_CRYPT_ALGO_DES	8
FS_CRYPT_ALGO_AES128	16
FS_CRYPT_ALGO_AES256	32

The encryption object can be shared between different files.

Example

Refer to *What do I need to do to use file encryption?* on page 526.

13.5.2 FS_CRYPT_Decrypt()

Description

Decrypts one or more blocks of data.

Prototype

```
void FS_CRYPT_Decrypt(const FS_CRYPT_OBJ * pCryptObj,
                      U8                 * pDest,
                      const U8           * pSrc,
                      U32                  NumBytes,
                      U32                * pBlockIndex);
```

Parameter	Description
pCryptObj	IN: Encryption object to be used for the decrypt operation OUT: ---
pDest	IN: --- OUT: Decrypted data
pSrc	IN: Data to be decrypted OUT: ---
NumBytes	Number of bytes to decrypt
pBlockIndex	IN: Index of the first block to decrypt OUT: Index of the next block to decrypt

Table 13.4: FS_CRYPT_Decrypt() parameter list

Additional information

The function should be used on a PC to decrypt a file encrypted on a target system. On a target system the data is decrypted automatically by the file system. pBlockIndex parameter can be used to start the decryption at an arbitrary block index inside the file. The size of the block is the value passed to BytesPerBlock in a call to FS_CRYPT_Prepare() which initialized the encryption object.

Example

For an example take a look at the source of the FSFileEncrypter.exe tool located in the Windows\FS_FileEncrypter\Src folder of the emFile shipment.

13.5.3 FS_CRYPT_Encrypt()

Description

Encrypts one or more blocks of data.

Prototype

```
void FS_CRYPT_Encrypt(const FS_CRYPT_OBJ * pCryptObj,
                      U8                 * pDest,
                      const U8           * pSrc,
                      U32                  NumBytes,
                      U32                * pBlockIndex);
```

Parameter	Description
pCryptObj	IN: Encryption object to be used for the decrypt operation OUT: ---
pDest	IN: --- OUT: Encrypted data
pSrc	IN: Data to be encrypted OUT: ---
NumBytes	Number of bytes to encrypt
pBlockIndex	IN: Index of the first block to encrypt OUT: Index of the next block to encrypt

Table 13.5: FS_CRYPT_Encrypt() parameter list

Additional information

The function should be used on a PC to encrypt a file encrypted on a target system. On a target system the data is encrypted automatically by the file system when the application writes to file. pBlockIndex parameter can be used to start the encryption at an arbitrary block index inside the file. The size of the block is the value passed to BytesPerBlock in a call to FS_CRYPT_Prepare() which initialized the encryption object.

Example

For an example take a look at the source of the FSFileEncrypter.exe tool located in the Windows\FS_FileEncrypter\Src folder of the emFile shipment.

13.5.4 FS_SetEncryptionObject()

Description

Assigns an encryption object to a file handle.

Prototype

```
void FS_SetEncryptionObject(FS_FILE      * pFile,
                            FS_CRYPT_OBJ * pCryptObj);
```

Parameter	Description
pFile	IN: Handle to opened file. OUT: ---
pCryptObj	IN: --- OUT: Encryption object to be initialized.

Table 13.6: FS_SetEncryptionObject() parameter list

Additional information

The function must be called right after the file is opened before any read or write operation. The pointer to encryption object is saved internally by emFile. This means that the memory it points to should be valid until the file is closed or until the FS_SetEncryptionObject() is called again with pCryptObj set to NULL.

Example

What do I need to do to use file encryption? on page 526.

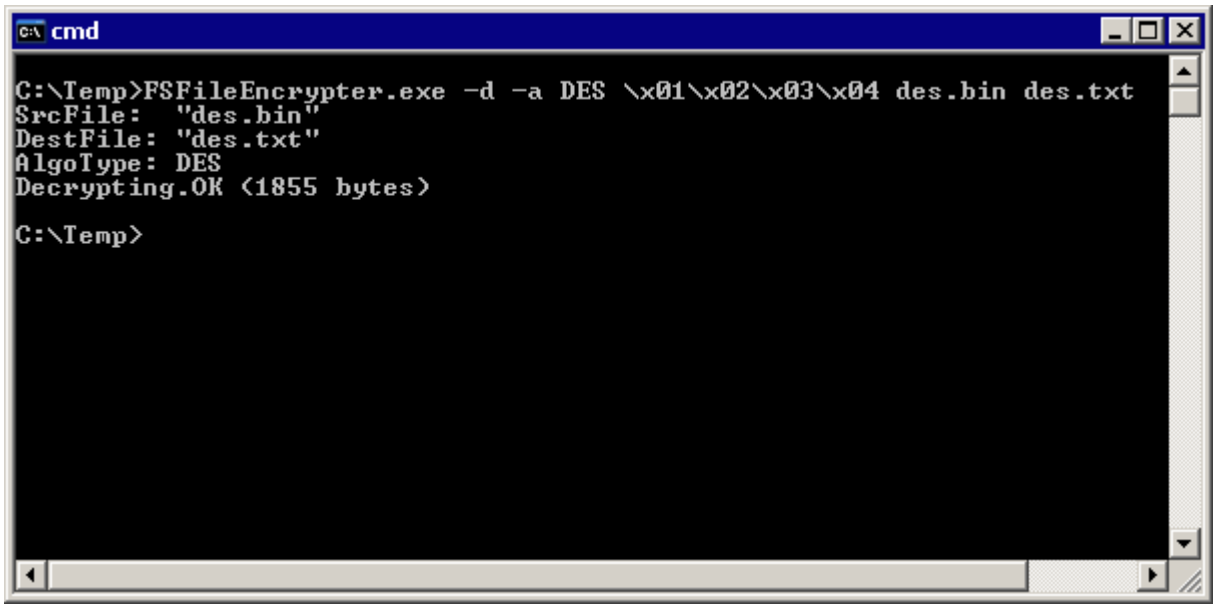
13.6 Encryption tool

emFile comes with command line tools to allow the encryption/decryption of files on a PC. Due to export regulations of encryption software two separate executable are provided: `FSFileEncrypter.exe` and `FSFileEncrypterES.exe`. which support different encryption strengths. `FSFileEncrypter.exe` supports only the encryption algorithms with a key smaller than or equal to 56-bit which includes the DES algorithm. Encryption/decryption with any key strength can be done using the `FSFileEncrypterES.exe` executable. The tool supports the DES and AES encryption algorithms.

13.6.1 Using the file encryption tools

The tools can be invoked directly form the command line or via a batch file. To use the tools directly a terminal window must be opened first. On the command line the name of the executable, either `FSFileEncrypter.exe` or `FSFileEncrypterES.exe`, should be input first followed by optional and required arguments. By pressing the Enter key the tool will perform encryption or decryption as specified.

Below is a screenshot of the `FSFileEncrypter.exe` decrypting the contents of the `des.bin` file to `des.txt` file. The encryption algorithm is DES as specified with the -a option. Information about the decrypting process is shown on the terminal. In case of an error a message is displayed and the executable returns with a status of 1. No destination file is created in this case.



13.6.2 Command line options

Parameters which can be omitted when invoking the tools from the command line.

13.6.2.1 -a

Description

Selects the encryption algorithm. Default encryption algorithm is DES.

Syntax

-a <AlgoType>

Additional information

The following table lists all valid values for <AlgoType>:

Permitted values for parameter <AlgoType>	
DES	Data Encryption Standard, 56-bit key length
AES128	Advanced Encryption Standard, 128-bit key length (supported only by FSFileEncrypterES.exe)
AES256	Advanced Encryption Standard, 256-bit key length (supported only by FSFileEncrypterES.exe)

Example

Shows how to encrypt the contents of the file plain.txt file to cipher.bin file using the DES algorithm. The encryption key is the string "secret".

```
C:>FSFileEncrypter -a DES secret plain.txt cipher.bin
```

13.6.2.2 -b

Description

Sets the size of the encryption block. Default block size is 512 bytes.

Syntax

-b <BlockSize>

Additional information

The parameter should be a power of 2 value and represents the number of bytes in the block. It should be equal to the BytesPerBlock passed in the call to FS_CRYPT_Prepare() function which initializes the encryption object on the target system.

Example

Shows how to encrypt the contents of the file plain.txt file to cipher.bin file using the DES algorithm. The encryption key is the string "secret" and the size of the encryption block is 2048 bytes.

```
C:>FSFileEncrypter -b 2048 secret plain.txt cipher.bin
```

13.6.2.3 -d

Description

Performs decryption. Default is encryption.

Syntax

-d

Example

Shows how to decrypt the contents of the file cipher.bin file to plain.txt file using the DES algorithm. The encryption key is the string "secret".

```
FSFileEncrypter -d secret cipher.bin plain.txt
```

13.6.2.4 -h

Description

Show the usage message and exit.

Syntax

-h

Example

```
C:>FSFileEncrypterES -h
DESCRIPTION
  File encryption/decryption utility for SEGGER emFile.
USAGE
  FSFileEncrypterES [-a <AlgoType>] [-b <BlockSize>]
    [-d] [-h] [-q] [-v] <Key> <SrcFile> <DestFile>
OPTIONS
  -a <AlgoType>   Type of the encryption algorithm. AlgoType can be one of:
                    DES    Data Encryption Standard, 56-bit key length
                    AES128 Advanced Encryption Standard, 128-bit key length
                    AES256 Advanced Encryption Standard, 256-bit key length
                  Default is DES.
  -b <BlockSize>  Number of bytes to be encrypted/decrypted at once.
                  BlockSize must be a power of 2 value. When encrypting a file
                  BlockSize should be smaller than or equal to the sector size
                  of the file system volume. When decrypting a file, BlockSize
                  should be equal to the value used to encrypt the file.
                  Default is 512 bytes.
  -d              Perform decryption. Default is encryption.
  -h              Show this help information.
  -q              Do not show log messages.
  -v              Show version information.
ARGUMENTS
  <Key>           Encryption/decryption key as ASCII string. Non-printable
                  characters can be specified as 2 hexadecimal characters
                  prefixed by the sequence '\x'.
                  Ex: the key value 1234 can be specified as \x04\xD2.
  <SrcFile>       Path to file to be encrypted/decrypted.
  <DestFile>      Path to encrypted/decrypted file.
```

13.6.2.5 -q

Description

Do not show log information. Default is to log messages to console.

Syntax

-q

13.6.2.6 -v

Description

Show version information and exit.

Syntax

-v

Example

```
C:>FSFileEncrypterES -v
SEGGER FS File Encrypter (Extra Strong) V1.01a ('?' or '-h' for help)
Compiled on Sep  4 2012 16:18:23
```

13.6.3 Command line arguments

Mandatory parameters which must be specified on the command line in the order they are described below.

13.6.3.1 <Key>

Description

A string which specifies the encryption key. Non-printable characters can be input in hexadecimal form by prefixing them with the string ’\x’. The key is case sensitive.

Example

The file `plain.txt` is encrypted using DES and the result is saved to `cipher.bin`. The password looks like this in binary form: 0x70 0x61 0x73 0x73 0x01 0x02 0x03 0x04.

```
C:>FSFileEncrypterES pass\x01\x02\x03\x04 plain.txt cipher.bin
```

13.6.3.2 <SrcFile>

Description

Path to the file to read from.

Additional information

It specifies the plain text file in case when encryption is performed. When decrypting the parameter specifies the cipher text file. The parameters <SrcFile> and <DestFile> must specify 2 different files.

Example

Shows how to encrypt the contents of the file `plain.txt` file to `cipher.bin` file using the AES encryption algorithm. `plain.txt` is the source file.

```
C:>FSFileEncrypterES -a AES128 pass plain.txt cipher.bin
```

13.6.3.3 <DestFile>

Description

Path to the file to write to.

Additional information

It specifies the cipher text file in case when encryption is performed. When decrypting the parameter specifies the plain text file. The parameters <SrcFile> and <DestFile> must specify 2 different files.

Example

Shows how to decrypt the contents of the file `cipher.bin` file to `plain.txt` file using the AES encryption algorithm. `plain.txt` is the destination file.

```
C:>FSFileEncrypterES -d -a AES256 pass cipher.bin plain.txt
```


13.7 Performance and resource usage

In this section the RAM (static and dynamic) and ROM resource usage of the file encryption is described. Refer to *Performance and resource usage* on page 458 for the performance and resource usage of volume encryption.

13.7.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the journaling have been measured on a system as follows: Cortex-M, IAR Embedded Workbench V6.30, Size optimization.

Module	ROM [Kbytes]
emFile File Encryption	0.4

In addition, one of the following cryptographic algorithms is required:

Physical layer	Description	ROM [Kbytes]
FS_CRYPT_ALGO_DES	DES encryption algorithm.	3.2
FS_CRYPT_ALGO_AES128	AES encryption algorithm using an 128-bit key.	12.0
FS_CRYPT_ALGO_AES256	AES encryption algorithm using a 256-bit key.	12.0

13.7.2 Static RAM usage

Static RAM usage is the amount of RAM required by the journal module for static variables. No static RAM is used by the file encryption.

13.7.3 Runtime (dynamic) RAM usage

Runtime (dynamic) RAM usage is the amount of RAM allocated by the file encryption at runtime. The file encryption requires one sector buffer for the encryption/decryption of data. By default, the size of the sector buffer is 512 and can be configured using the FS_SetMaxSectorSize().

13.7.4 Performance

These performance measurements are in no way complete, but they give an approximation of the length of time required for common operations on various targets. The tests were performed as described in *Performance* on page 502.

All values are in Kbytes/sec.

Device	CPU speed	Medium	W	R
Freescale Kinetis K60	120 MHz	NAND flash interfaced via 8-bit bus using AES with an 128-bit key.	522	553
ST STM32F4	96 MHz	SD card as storage medium using AES with an 128-bit key	500	530

Table 13.7: Performance values for sample configurations

Chapter 14

Porting emFile 2.x to 3.x

14.1 Differences from version 2.x to 3.x

Most of the differences from emFile version 2.x to version 3.x are internal. The API of emFile version 2.x is a subset of the API of version 3.x. Only few functions are completely removed. Refer to section *API differences* on page 540 for a complete overview of the removed and obsolete functions.

emFile version 3 has a new driver handling. You can include drivers and allocate the required memory for the accordant driver without the need to recompile the whole file system. Refer to *Configuration of emFile* on page 471 for detailed information about the integration of a driver into emFile. For detailed information to the emFile device drivers, refer to the chapter *Device drivers* on page 213.

Because of these differences, we recommend to start with a new file system project and include your application code, if the start project runs without any problem. Refer to the chapter *Running emFile on target hardware* on page 33 for detailed information about the best way to start the work with emFile version 3.x.

The following sections gives an overview about the changes from emFile version 2.x. to emFile version 3 in table form.

14.2 API differences

Function	Description
Changed functions	
`FS_GetFreeSpace()`	Number of parameters reduced. Parameter `DevIndex` removed.
`FS_GetTotalSpace()`	Number of parameters reduced. Parameter `DevIndex` removed.
Removed functions	
`FS_Exit()`	Should be removed from your application source code.
`FS_CheckMediumPresent()`	
Obsolete directory handling functions	
`FS_CloseDir()`	The directory handling has been changed in emFile version 3.x. The functions should be replaced. Refer to *FS_FindClose()* on page 104 for an example of the new way of directory handling.
`FS_DirEnt2Attr()`	
`FS_DirEnt2Name()`	
`FS_DirEnt2Size()`	
`FS_DirEnt2Time()`	
`FS_GetNumFiles()`	
`FS_OpenDir()`	
`FS_ReadDir()`	
`FS_RewindDir()`	
Obsolete file system extended functions	
`FS_IoCtl()`	`FS_IoCtl()` should not be used in emFile version 3.x. Use `FS_IsLLFormatted()` to check if a low-level format is required and `FS_GetDeviceInfo()` to get the device information.

Table 14.1: Differences between emFile v.2.x / v.3.x - API differences

In emFile version 3 is the header file `FS_Api.h` renamed to `FS.h`, therefore change the name of the file system header file in your application.

14.3 Configuration differences

The configuration of emFile version 3.x has been simplified compared to emFile v2.x. emFile v3.x can be used "out of the box". You can use it without the need for changing any of the compile time flags. All compile time configuration flags are preconfigured with valid values, which matches the requirements of the most applications.

A lot of the compile time flags of emFile v.2.x are removed and replaced with runtime configuration function.

Removed/replaced configuration macros

In version 3.x removed macros	In version 3.x used macros
File system configuration	
FS_MAXOPEN	--
FS_POSIX_DIR_SUPPORT	--
FS_DIR_MAXOPEN	FS_NUM_DIR_HANDLES
FS_DIRNAME_MAX	--
FS_SUPPORT_BURST	--
FS_DRIVER_ALIGNMENT	--
FAT configuration macros	
FS_FAT_SUPPORT_LFN	Replaced by FS_FAT_SupportLFN(). Refer to *FS_FAT_SupportLFN()* on page 174 for more information.

Table 14.2: Differences between emFile v.2.x / v.3.x - removed/replaced configuration macros

14.4 Device driver

14.4.1 Renamed drivers

Old driver names	Driver names in emFile version 3.x
NAND2K	In emFile version 3.x, the NAND driver could be used to access small and large block NAND flashes similarly. The driver is therefore renamed from NAND2K to NAND.
SMC	In emFile version 3, the SMC / small block NAND driver is integrated in the NAND driver. The NAND driver could be used to access small and large block NAND flashes similarly.
SFLASH	The serial flash driver is renamed into DataFlash driver.
FLASH	FLASH driver renamed to NOR flash driver.

Table 14.3: Differences between emFile v.2.x / v.3.x - list of renamed device drivers

14.4.2 Integrating a device driver into emFile

In version 2.x, you have to enable a device driver with a macro which has to be set has to be set in the emFile configuration file `FS_Conf.h` and recompile your file system project. emFile version 3.x is run time configurable, so that you can add all device drivers by calling the `FS_AddDevice()` function with the proper parameter for the accordant driver.

In version 3.x removed macros	Alternative
FS_USE_FLASH_DRIVER	FS_AddDevice(&FS_NOR_Driver)
FS_USE_IDE_DRIVER	FS_AddDevice(&FS_IDE_Driver)
FS_USE_MMC_DRIVER	FS_AddDevice(&FS_MMC_SPI_Driver) FS_AddDevice(&FS_MMC_CardMode_Driver)
FS_USE_RAMDISK_DRIVER	FS_AddDevice(&FS_RAMDISK_Driver)
FS_USE_SFLASH_DRIVER	FS_AddDevice(&FS_DataFlash_Driver)
FS_USE_SMC_DRIVER	FS_AddDevice(&FS_NAND_Driver)
FS_USE_NAND2K_DRIVER	FS_AddDevice(&FS_NAND_Driver)
FS_USE_WINDRIVE_DRIVER	FS_AddDevice(&FS_WINDRIVE_Driver)

Table 14.4: Differences between emFile v.2.x / v.3.x - adding a driver

14.4.3 RAM disk driver differences

In version 3.x removed macros	Alternative
FS_USE_RAMDISK_DRIVER	FS_AddDevice(&FS_RAMDISK_Driver)
FS_RAMDISK_NUM_SECTORS	FS_RAMDISK_Configure() - Refer to *FS_RAMDISK_Configure()* on page 218 for detailed information.
FS_RAMDISK_MAXUNIT	
FS_RAMDISK_ADDR	
FS_RAMDISK_SECTOR_SIZE	

Table 14.5: Differences between emFile v.2.x / v.3.x - removed RAMDISK macros

Refer to the section *RAM disk driver* on page 217 for detailed information about the RAM disk driver in emFile version 3.x.

14.4.4 NAND driver differences

In version 3.x removed macros	Alternative
FS_USE_NAND2K_DRIVER	FS_AddDevice(&FS_NAND_Driver)
FS_NAND2K_MAXUNIT	FS_NAND_SetPhyType() - Refer to *FS_NAND_SetPhyType()* on page 231 for detailed information.
FS_NAND2K_MAX_NUM_PHY_BLOCKS	FS_NAND_SetBlockRange() - Refer to *FS_NAND_SetBlockRange()* on page 233 for detailed information.

Table 14.6: Differences between emFile v.2.x / v.3.x - removed NAND driver macros

Hardware interface version 2.x	Hardware interface version 3.x
FS_NAND2K_HW_X_SetAddr()	FS_NAND_HW_X_SetAddrMode()
FS_NAND2K_HW_X_SetCmd()	FS_NAND_HW_X_SetCmdMode()
FS_NAND2K_HW_X_SetData()	FS_NAND_HW_X_SetDataMode()
FS_NAND2K_HW_X_SetStandby()	FS_NAND_HW_X_SetStandby()
FS_NAND2K_HW_X_WaitWhileBusy()	FS_NAND_HW_X_WaitWhileBusy()
FS_NAND2K_HW_X_IsWriteProtected()	FS_NAND_HW_X_IsWriteProtected()
FS_NAND2K_HW_X_Read()	FS_NAND_HW_X_Read()
FS_NAND2K_HW_X_Write()	FS_NAND_HW_X_Write()
FS_NAND2k_HW_X_Delayus()	FS_NAND_HW_X_Delayus()
FS_NAND2K_HW_X_Init()	FS_NAND_HW_X_Init()
--	FS_NAND_HW_X_DisableCE()
--	FS_NAND_HW_X_EnableCE()

Table 14.7: Differences between emFile v.2.x / v.3.x - IDE driver hardware interface differences

Refer to the section *NAND flash driver* on page 221 for detailed information about the NAND driver in emFile version 3.x.

14.4.5 NAND driver differences

In version 3.x removed macros	Alternative
FS_USE_SMC_DRIVER	FS_AddDevice(&FS_NAND_Driver)
FS_SMC_MAXUNIT	FS_NAND_SetPhyType() - Refer to *FS_NAND_SetPhyType()* on page 231 for detailed information.
FS_SMC_HW_SUPPORT_BSYLINE_CHECK	FS_NAND_SetBlockRange() - Refer to *FS_NAND_SetBlockRange()* on page 233 for detailed information.

Table 14.8: Differences between emFile v.2.x / v.3.x - adding a driver

Hardware interface version 2.x	Hardware interface version 3.x
FS_SMC_HW_X_SetAddr()	In emFile version 3, the SMC / small block NAND driver is integrated in the NAND driver. The NAND driver could be used to access small and large block NAND flashes similarly. Refer to *NAND flash driver* on page 221 for detailed information about the NAND driver in emFile version 3.x
FS_SMC_HW_X_SetCmd()	
FS_SMC_HW_X_SetData()	
FS_SMC_HW_X_SetStandby()	
FS_SMC_HW_X_VccOff()	
FS_SMC_HW_X_VccOn()	
FS_SMC_HW_X_ChkBusy()	
FS_SMC_HW_X_ChkCardIn()	
FS_SMC_HW_X_ChkPower()	
FS_SMC_HW_X_ChkStatus()	
FS_SMC_HW_X_ChkWP()	
FS_SMC_HW_X_DetectStatus()	
FS_SMC_HW_X_InData()	
FS_SMC_HW_X_OutData()	
FS_SMC_HW_X_ChkTimer()	
FS_SMC_HW_X_SetTimer()	
FS_SMC_HW_X_StopTimer()	
FS_SMC_HW_X_WaitTimer()	

Table 14.9: Differences between emFile v.2.x / v.3.x - IDE driver hardware interface differences

Refer to the section *NAND flash driver* on page 221 for detailed information about the NAND driver in emFile version 3.x.

14.4.6 MMC driver differences

In version 3.x removed macros	Alternative
FS_USE_MMC_DRIVER	FS_AddDevice(&FS_MMC_CardMode_Driver)
FS_MMC_USE_SPI_MODE	FS_AddDevice(&FS_MMC_SPI_Driver)
FS_MMC_MAXUNIT	--
FS_USE_CRC	FS_MMC_ActivateCRC() / FS_MMC_DeactivateCRC()
FS_MMC_SUPPORT_4BIT_MODE	--

Table 14.10: Differences between emFile v.2.x / v.3.x - removed MMC macros

Refer to the section *MMC/SD card driver* on page 373 for detailed information about the MMC driver in emFile version 3.x.

14.4.7 CF/IDE driver differences

In version 3.x removed macros	Alternative
FS_USE_IDE_DRIVER	FS_AddDevice(&FS_IDE_Driver)
FS_IDE_MAXUNIT	--

Table 14.11: Differences between emFile v.2.x / v.3.x - removed CF/IDE macros

In version 3.x is the hardware interface of the CF/IDE driver simplified. Only 6 hardware functions have to implemented.

Hardware interface version 2.x	Hardware interface version 3.x
FS_IDE_HW_X_HWReset()	FS_IDE_HW_X_HWReset()
FS_IDE_HW_X_Delay400ns()	FS_IDE_HW_X_Delay400ns()
FS_IDE_HW_X_GetAltStatus()	--
FS_IDE_HW_X_GetCylHigh()	--
FS_IDE_HW_X_GetCylLow()	--
FS_IDE_HW_X_GetData()	FS_IDE_HW_X_ReadData()
FS_IDE_HW_X_GetError()	--
FS_IDE_HW_X_GetSectorCount()	--
FS_IDE_HW_X_GetSectorNo()	--
FS_IDE_HW_X_GetStatus()	--
FS_IDE_HW_X_SetCommand()	--
FS_IDE_HW_X_SetCylHigh()	--
FS_IDE_HW_X_SetCylLow()	--
FS_IDE_HW_X_SetData()	FS_IDE_HW_X_WriteData()
FS_IDE_HW_X_SetDevControl()	--
FS_IDE_HW_X_SetDevice()	--
FS_IDE_HW_X_SetFeatures()	--
FS_IDE_HW_X_SetSectorCount()	--
FS_IDE_HW_X_SetSectorNo()	--
FS_IDE_HW_X_DetectStatus()	--
--	FS_IDE_HW_X_ReadReg()
--	FS_IDE_HW_X_WriteReg()

Table 14.12: Differences between emFile v.2.x / v.3.x - CF/IDE driver hardware interface differences

Refer to the section *CompactFlash card and IDE driver* on page 425 for detailed information about the CF/IDE driver in emFile version 3.x.

14.4.8 Flash / NOR flash differences

In version 3.x removed macros	Alternative
FS_USE_FLASH_DRIVER	FS_AddDevice(&FS_NOR_Driver)
FS_FLASH_MAX_ERASE_CNT_DIFF	FS_NOR_Configure() - Refer to *FS_NOR_Configure()* on page 322 for detailed information. FS_NOR_SetPhyType() - Refer to *FS_NOR_SetPhyType()* on page 324 for detailed information.
FS_FLASH_NUM_FREE_SECTORCACHE	
FS_FLASH_CHECK_INFO_SECTOR	
FLASH_BASEADR	
FLASH_USER_START	
FLASH_BYTEMODE	
FLASH_RELOCATECODE	
FS_FLASH_CAN_REWRITE	
FS_FLASH_LINE_SIZE	
FS_FLASH_SECTOR_SIZE	

Table 14.13: Differences between emFile v.2.x / v.3.x - removed Flash / NOR flash macros

Refer to the section *NOR flash driver* on page 315 for detailed information about the NOR flash driver in emFile version 3.x.

14.4.9 Serial Flash / DataFlash differences

In version 3.x removed macros	Alternative
FS_USE_SFLASH_DRIVER	FS_AddDevice(&FS_DataFlash_Driver);
FS_SFLASH_MAXUNIT	--

Table 14.14: Differences between emFile v.2.x / v.3.x - removed Serial Flash / DataFlash macros

Note: The DataFlash support is integrated into the NAND flash driver since version 3.10. Refer to *NAND flash driver* on page 221 for detailed information.

14.4.10 Windrive differences

In version 3.x removed macros	Alternative
FS_WD_DEV0NAME	FS_Windrive_Configure() - Refer to *WINDRIVE_Configure()* on page 446 for detailed information.
FS_WD_DEV1NAME	

Table 14.15: Differences between emFile v.2.x / v.3.x - removed Windrive macros

Refer to the section *WinDrive driver* on page 446 for detailed information about the Windrive driver in emFile version 3.x.

14.5 OS Integration

In version 3.x removed macros	In version 3.x used macros
OS configuration macros	
FS_OS_LOCKING_PER_FILE	Removed. If you want to use emFile version 3.x with an RTOS, define FS_OS_LOCKING in your FS_Conf.h. Refer to *OS integration* on page 481 for information about he functions which has to be implemented to use emFile with an RTOS.
FS_OS_EMBOS	
FS_OS_UCOS_II	
FS_OS_WINDOWS	
FS_OS	

Table 14.16: Differences between emFile v.2.x / v.3.x - removed/replaced configuration macros

Function	Description
Changed functions	
FS_X_OS_Init()	In emFile version 3.x gets FS_X_OS_Init() an additional parameter. Refer to FS_X_OS_Init()
Removed functions	
FS_X_OS_ Exit()	--
Time and date functions	
FS_X_OS_GetDate()	In emfile version 3.x is only one version used to handle the time and date function-ality. Refer to *FS_X_GetTimeDate()* on page 472 for more information.
FS_X_OS_GetDateTime()	

Table 14.17: Differences between emFile v.2.x / v.3.x - Changes in the OS interface

Chapter 15

FAQs

You can find in this chapter a collection of frequently asked questions (FAQs) together with answers.

15.1 FAQs

Q: Is my data safe, when an unexpected RESET occurs?
A: In general, the data which is already on the medium is safe. If a read operation is interrupted, this is completely harmless. If a write operation is interrupted, the data written in this operation may or may not be stored on the medium, depending on when the unexpected RESET occurred. In any case, the data which was on the media prior to the write operation is not affected; directory entries are not messed up, the file-allocation-table is kept in order. This is true if your storage medium is not affected by the RESET, meaning that it is able to complete a pending write operation. (Which is typically the case with Flash memory cards other than SMC)

Q: I use FAT and I can only create a limited number of root directory entries. Why?
A: With FAT12 and FAT16 the root directory is special because it has a fixed size. During media format one can determine the size, but once formatted this value is constant and determines the number of entries the root directory can hold. FAT32 does not have this limitation and the root directory's size can be variable.

Microsoft's “FAT32 File System Specification” says on page 22: “For FAT12 and FAT16 media, the root directory is located in a fixed location on the disk immediately following the last FAT and is of a fixed size in sectors computed from the BPB_RootEntCnt value [...] For FAT32, the root directory can be of variable size and is a cluster chain, just like any other directory is.”. Here BPB_RootEntCnt specifies the count of 32-byte directory entries in the root directory and as the citation says, the number of sectors is computed from this value.

In addition, which file system is used depends on the size of the medium, that is the number of clusters and the cluster size, where each cluster contains one or more sectors. Using small cluster sizes (for example cluster size = 512 bytes) one can use FAT32 on media with more than 32 MB. (FAT16 can address at least 216 clusters with a 512 byte cluster size. That is 65536 * 512 = 33554432 bytes = 32768 KB = 32 MB). If the media is smaller than or equal to 32 MB or the cluster size is greater than 512 bytes, FAT32 cannot be used.

To actually set a custom root directory size for FAT12/FAT16 one can use the emFile API function `int FS_Format(const char * pDevice, FS_FORMAT_INFO * pFormatInfo);` where `FS_FORMAT_INFO` is declared as:

```
typedef struct {
  U16 SectorsPerCluster;
  U16 NumRootDirEntries;
  FS_DEV_INFO * pDevInfo;
} FS_FORMAT_INFO;
```

Set `NumRootDirEntries` to the desired number of root directory entries you want to store.

Index

F

I

L

M

N

O

S

T

W

Mouser Electronics

Authorized Distributor

Click to View Pricing, Inventory, Delivery & Lifecycle Information:

Segger Microcontroller:

2.50.04 2.50.02