emFile

CPU-independent file system
for embedded applications

User Guide & Reference Manual

Document: UM02001
Software Version: 5.2.0
Revision: 0
Date: June 29, 2020

A product of SEGGER Microcontroller GmbH

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 (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

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel.	+49 2173-99312-0
Fax.	+49 2173-99312-28
E-mail:	support@segger.com*
Internet:	www.segger.com

*By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at https://www.segger.com/legal/privacy-policy/.

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: June 29, 2020

Software	Revision	Date	By	Description
5.0.0	0	200301	MD	Reworked version of the manual.
5.0.1	0	200316	MD	Corrected structure member names.
5.0.1	1	200317	MD	Corrected a sample.
5.1.0	0	200504	MD	Added `FS_GetMountType()` function. Added `FS_NOR_SPIFI_SetSectorSize()` function. Updated the list of supported NOR flash devices. Added `FS_EFS_SUPPORT_FREE_CLUSTER_CACHE` configuration define. Added `FS_SUPPORT_SECTOR_BUFFER_CACHE` configuration define. Added `FS_GetMemInfo()` function. Added more information about the fail-safety of the NOR drivers. Added compile time configuration information for the Block Map NOR driver.
5.1.0	1	200411	MD	Updated the resource usage of EFS.
5.2.0	0	200527	MD	Improved wording of the FAQs chapter. Added description of the BigFileMerger utility. Added `FS_NOR_BM_SetInvalidSectorError()` function.

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 of contents

Chapter 1

Introduction to emFile

This section presents an overview of emFile, its structure, and its capabilities.

1.1 What is emFile

emFile is a file system design for embedded applications which supports NAND, DataFlash, NOR and SPI Flash, SD and MMC Memory Cards, RAM and USB mass storage devices. emFile is a high performance library optimized for high speed, versatility and a minimal memory footprint of both RAM and ROM. It is written in ANSI C and can be used on any CPU.

1.2 Features

The main features of emFile are:

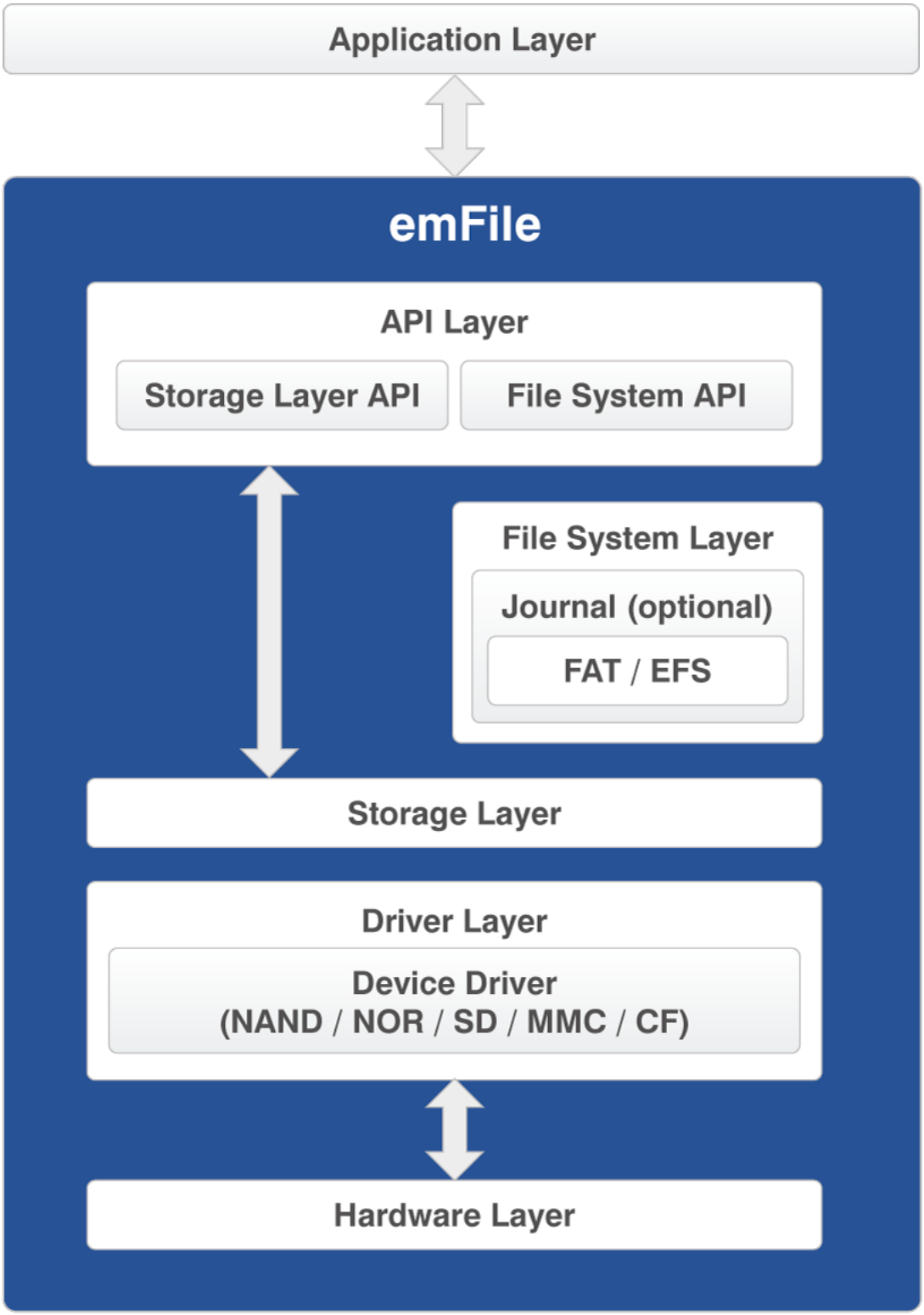
- Two file systems variants: FAT or SEGGER’s proprietary Embedded File System (EFS).
- FAT supports MS DOS/MS Windows-compatible FAT12, FAT16 and FAT32.
- EFS natively supports Long File Name (LFN). Add-on for FAT LFN available.
- Multiple device driver support; the same driver can support multiple storage media.
- Multiple media support; device drivers allow concurrent access to different storage media types.
- Cache support via RAM for optimized performance.
- Fail-safe and Task-safe, works with any operating system.
- ANSI C stdio.h-like API. Applications using standard C I/O library can easily be ported to emFile.
- Simple device driver structure, sample code trial versions and extensive API documentation.
- NAND Flash driver for SLC and MLC NAND and DataFlash with ECC and wear leveling.
- NOR Flash driver for NOR, SPI and QSPI Flash with wear leveling.
- Driver for Memory Card devices such as MMC, SD, SDHC, eMMC using bus and SPI mode.
- IDE Driver, Compact Flash, True-IDE and memory mapped mode.
- Journaling, RAID1 and RAID5 options to enhance data integrity.
- FAT Long File Name (LFN).
- Encryption (DES) and Extra Strong Encryption (DES and AES.)
- Profiling via SEGGER SystemView.
- Image creator tools for NOR and NAND.
- NAND flash evaluation board available.
- SQLite integration is available as sample upon request.
- MISRA C:2012 compliant.
- Support for Shift-JIS encoded Japanese file names.
- Support for file larger than 4 GBytes via BigFile component.

1.3 Basic concepts

This section describes the software structure of emFile as well as other general concepts such as fail safety and wear leveling.

1.3.1 emFile structure

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



1.3.1.1 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. Both file systems can 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.

1.3.1.2 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.

1.3.1.3 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.

1.3.1.4 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.

1.3.1.5 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. The other file system EFS, is a proprietary file system developed by SEGGER. The choice of the suitable file system depends 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 files 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 and Linux. The newest version, FAT32, improves upon the FAT16 file system by utilizing a partition/disk much more efficiently. It is supported by all Microsoft Windows versions newer than Windows 98 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 format). LFN support may be added to any of the FAT file systems. Long file names are inherent to this proprietary file system.

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 makes the FAT/EFS file systems fail-safe. The device drivers of emFile are all fail-safe by design. You can find detailed information about how the fail-safety works in chapter *Journaling* on page 1000 and as part 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 100,000. The manufacturers do not guarantee that the storage device will work properly once 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

This section provides information about the implementation of emFile

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 builds in only code that is required. This concept allows to put the file system in a library. The file system does not need 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 927.

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
```

1.4.4 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 standards 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 Package content and installation

emFile is provided in source code and contains everything needed to compile it on any platform. The following table shows the contents of the emFile package:

Files	Description
`Application`	Sample applications.
`BSP`	Support files for different evaluation boards
`Config`	Configuration header files.
`Doc`	emFile documentation.
`FS`	emFile source code.
`Inc`	Global header files.
`Linux`	Utility applications for Linux.
`Sample`	Sample drivers and applications.
`SEGGER`	Utility source code.
`Simulation`	Support files for PC simulation.
`Windows`	Utility applications for Windows.
`FS_Start.*`	PC simulation project MS Visual Studio / C++ and FAT.
`FS_EFS_Start.*`	PC simulation project MS Visual Studio / C++ and EFS.
`FS_STORAGE_Start.*`	PC simulation project MS Visual Studio / C++ and storage layer.

emFile is shipped in electronic form in a `.zip` file. In order to install it, extract the `.zip` file to any folder of your choice, preserving the directory structure of the `.zip` file.

2.2 Using the Windows sample application

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 application

Open the workspace `FS_Start.sln` 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 Overview of the sample application

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:

```
void main(void);
void main(void) {
  MainTask();                                                    ❶
}
```

```
void MainTask(void) {
  U32          v;
  FS_FILE    * pFile;
  char         ac[256];
  char         acFileName[32];
  const char * sVolumeName = "";

  FS_X_Log("Start\n");
  //
  // Initialize file system
  //
  FS_Init();                                                     ❷
  //
  // Check if low-level format is required
  //
  FS_FormatLLIfRequired(sVolumeName);                            ❸
  //
  // Check if volume needs to be high level formatted.
  //
  if (FS_IsHLFormatted(sVolumeName) == 0) {                      ❹
    FS_X_Log("High-level format\n");
    FS_Format(sVolumeName, NULL);
  }
  sprintf(ac, "Running sample on \"%s\"\n", sVolumeName);
  FS_X_Log(ac);                                                  ❺
  v = FS_GetVolumeFreeSpaceKB(sVolumeName);                      ❻
  if (v < 0x8000) {
    sprintf(ac, "  Free space: %lu KBytes\n", v);
  } else {
    v >>= 10;
    sprintf(ac, "  Free space: %lu MBytes\n", v);
  }
  FS_X_Log(ac);
  sprintf(acFileName, "%s\\File.txt", sVolumeName);
  sprintf(ac, "  Write test data to file %s\n", acFileName);
  FS_X_Log(ac);
  pFile = FS_FOpen(acFileName, "w");                             ❼
  if (pFile) {
    FS_Write(pFile, "Test", 4);                                  ❽
```



```
    FS_FClose(pFile);
  } else {
    sprintf(ac, "Could not open file: %s to write.\n", acFileName);
    FS_X_Log(ac);
  }
  v = FS_GetVolumeFreeSpaceKB(sVolumeName);                    ❾
  if (v < 0x8000) {
    sprintf(ac, "  Free space: %lu KBytes\n", v);
  } else {
    v >>= 10;
    sprintf(ac, "  Free space: %lu MBytes\n", v);
  }
  FS_X_Log(ac);
  FS_Unmount(sVolumeName);
  FS_X_Log("Finished\n");                                      ❿
  while (1) {
    ;
  }
}
```

❶ Application start

`main.c` calls `MainTask()`.

❷ File system initialization

`MainTask()` initializes and adds a device to emFile.

❸ Low-level formatting

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

❹ High-level formatting

Checks if volume is high-level formatted and formats if required.

❺ Volume name

Outputs the volume name.

❻ Free space before operation

Calls `FS_GetVolumeFreeSpace()` and outputs the return value - the available free space of the RAM disk - to console window.

❼ File creation

Creates and opens a file test with write access (`File.txt`) on the device.

❽ File write

Writes 4 bytes into the file and closes the file handle or outputs an error message.

❾ Free space after operation

Calls `FS_GetVolumeFreeSpace()` and outputs the return value - the available free space of the RAM disk - again to console window.

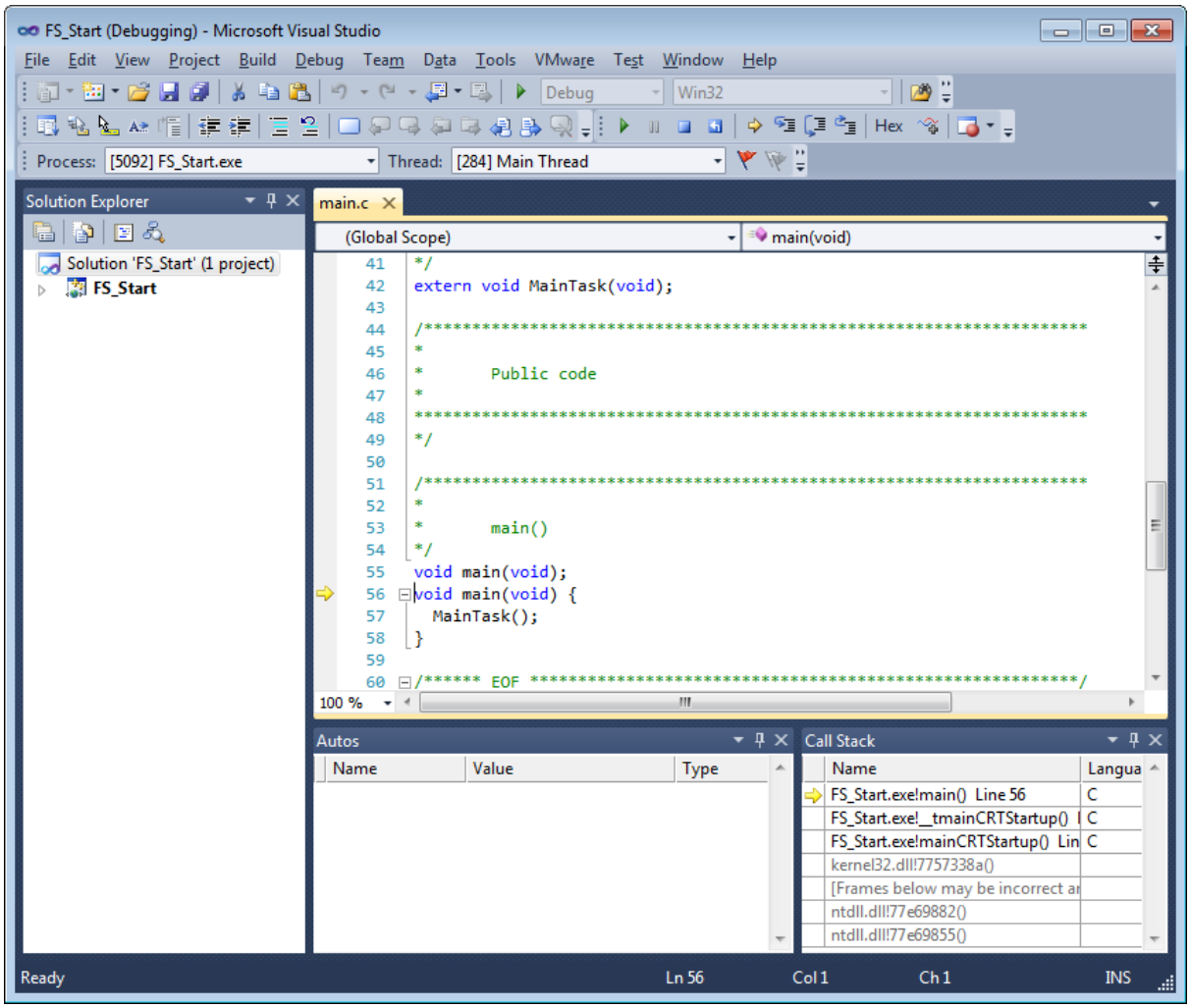
❿ End of application

Outputs a quit message and runs into an endless loop.

2.2.3 Stepping through the sample application

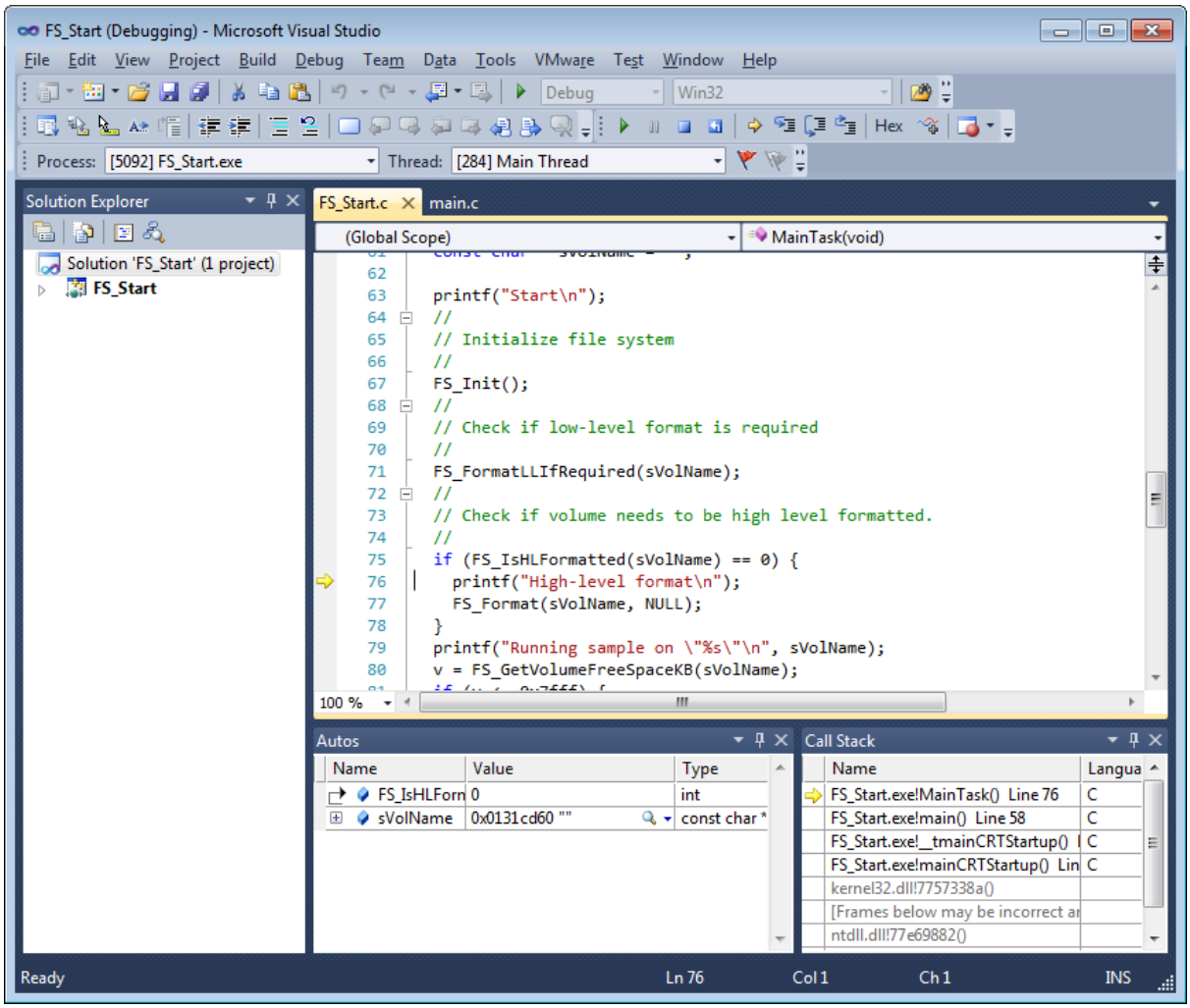
❶ Application start

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



❷ File system initialization

The first thing 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.



❸ Low-level formatting

If the initialization was successful, `FS_FormatLLIfRequired()` is called. It checks if the volume is low-level formatted and formats the volume if required. You should step over this function.

❹ High-level formatting

Afterwards `FS_IsHLFormatted()` is called. It checks if the volume is high-level formatted and formats the volume if required. You should step over this function.

❺ Volume name

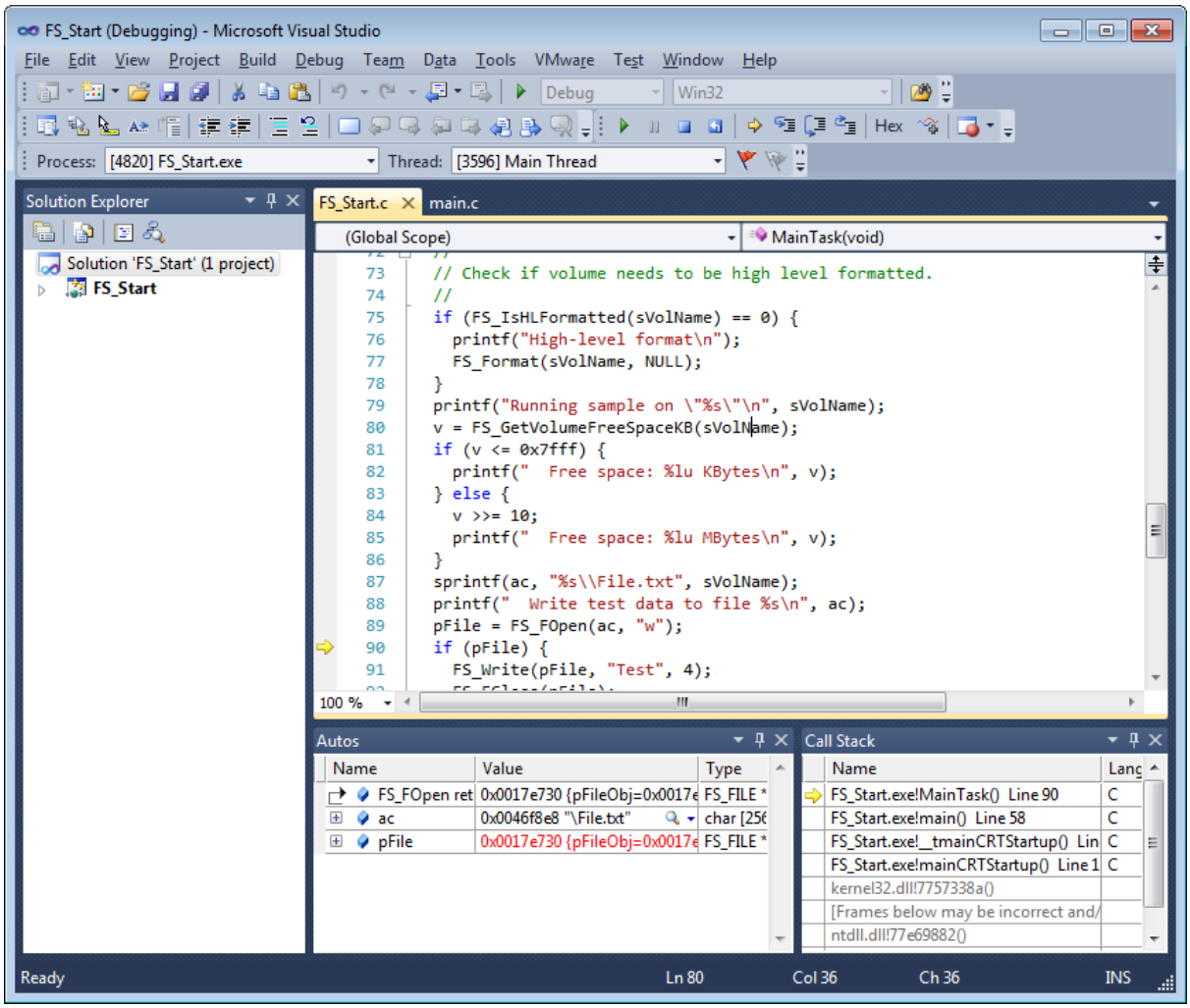
The volume name is printed in the console window.

❻ Free space before operation

The emFile function `FS_GetVolumeFreeSpace()` is called and the return value is written into the console window.

❼ File creation

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.



❽ File write

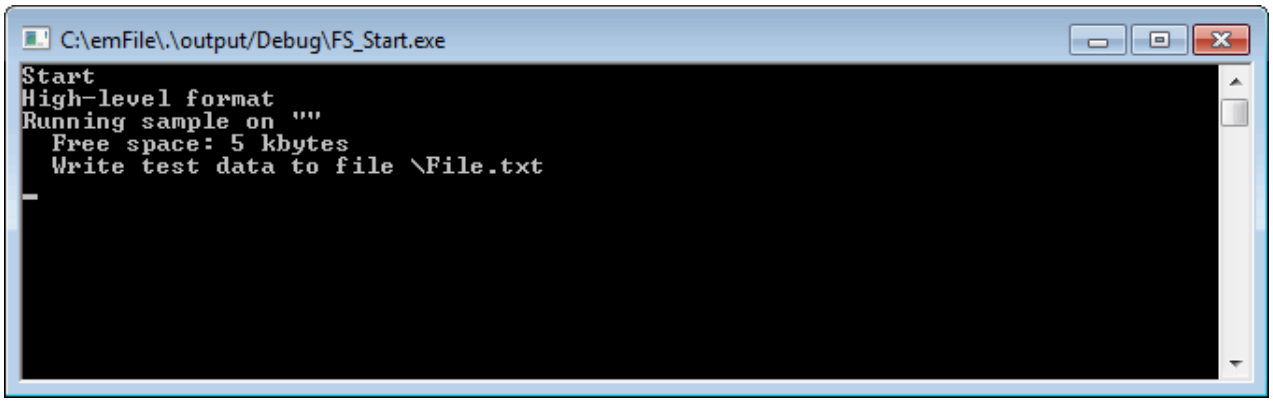
If `FS_FOpen()` returns a valid pointer to an `FS_FILE` structure, the sample application will write a small ASCII string to this file by calling the emFile function `FS_Write()`. Step over this function. If a problem occurs, compare the return value of `FS_Write()` with the length of the ASCII string, which should be written. `FS_Write()` 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.

❾ Free space after operation

Continue stepping over until you reach the place where the function `FS_GetVolumeFreeSpace()` is called. 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.

❿ End of application

The return value is written in the console window.



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/FS/Application` folder of the emFile shipment.

2.4 Recommended project structure

We recommend keeping emFile separate from your application files. It is good practice to keep all the program files (including the header files) together in the FS subdirectory of your project's root directory. This practice has the advantage of being very easy to update to newer versions of emFile by simply replacing the FS directory. Your application files can be stored anywhere.

Note

When updating to a newer emFile version: as files may have been added, moved or deleted, the project directories may need to be updated accordingly.

Note

Always make sure that you have only one version of each file!

It is frequently a major problem when updating to a new version of emFile if you have old files included and therefore mix different versions. If you keep emFile in the directories as suggested (and only in these), this type of problem cannot occur. When updating to a newer version, you should be able to keep your configuration files and leave them unchanged. For safety reasons, we recommend backing up (or at least renaming) the FS directories before updating.

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.

3.1 Integrating emFile

The default configuration of emFile contains a single storage device: a RAM disk. This should always be the first step to check if emFile functions properly on your target hardware. We assume that you are familiar with the tools you have selected for your development (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 on your target system (if you are using one). The SEGGER Embedded Studio IDE (https://www.segger.com/embedded-studio.html) is used in this document for all examples and screenshots, but every other ANSI C toolchain can also be used. It is also possible to use makefiles; in this case, when we say “add to the project”, this translates into “add to the makefile”.

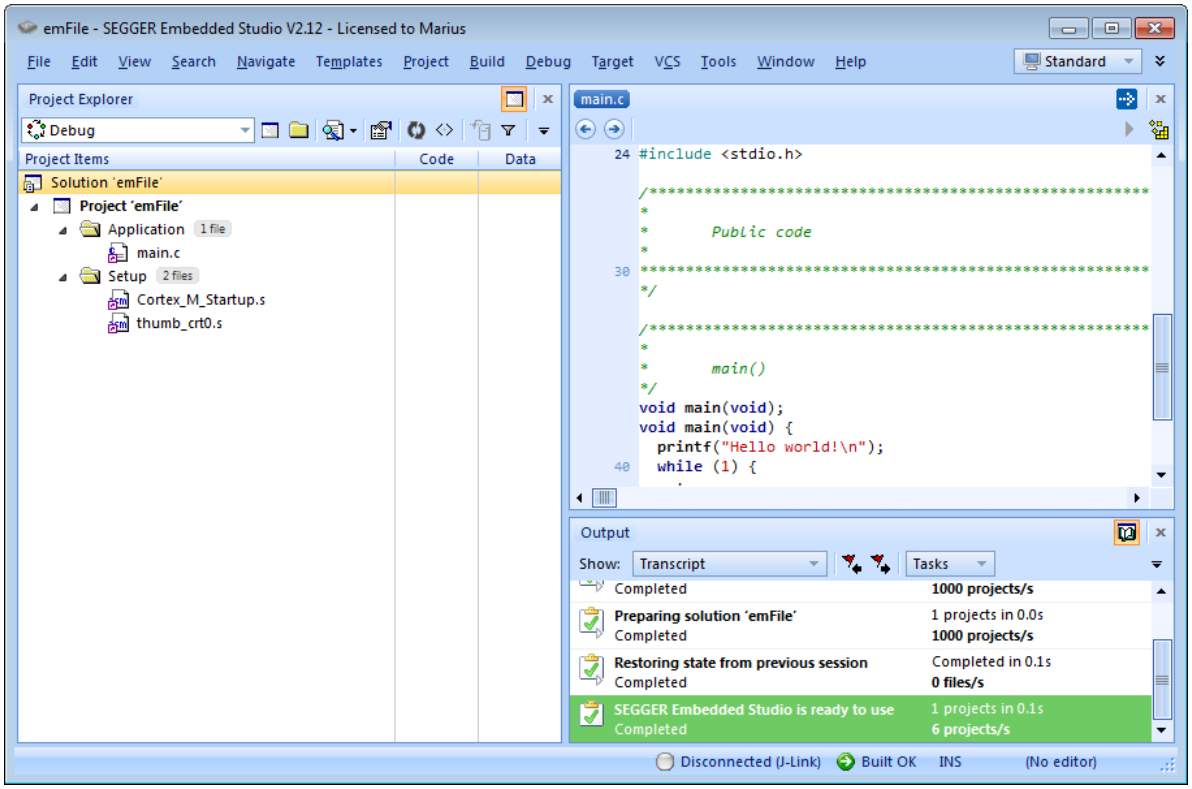
3.2 Procedure to follow

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.

3.3 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 the SEGGER embOS (https://www.segger.com/products/rtos/embos/), you can use the start project shipped with the embOS for this purpose.

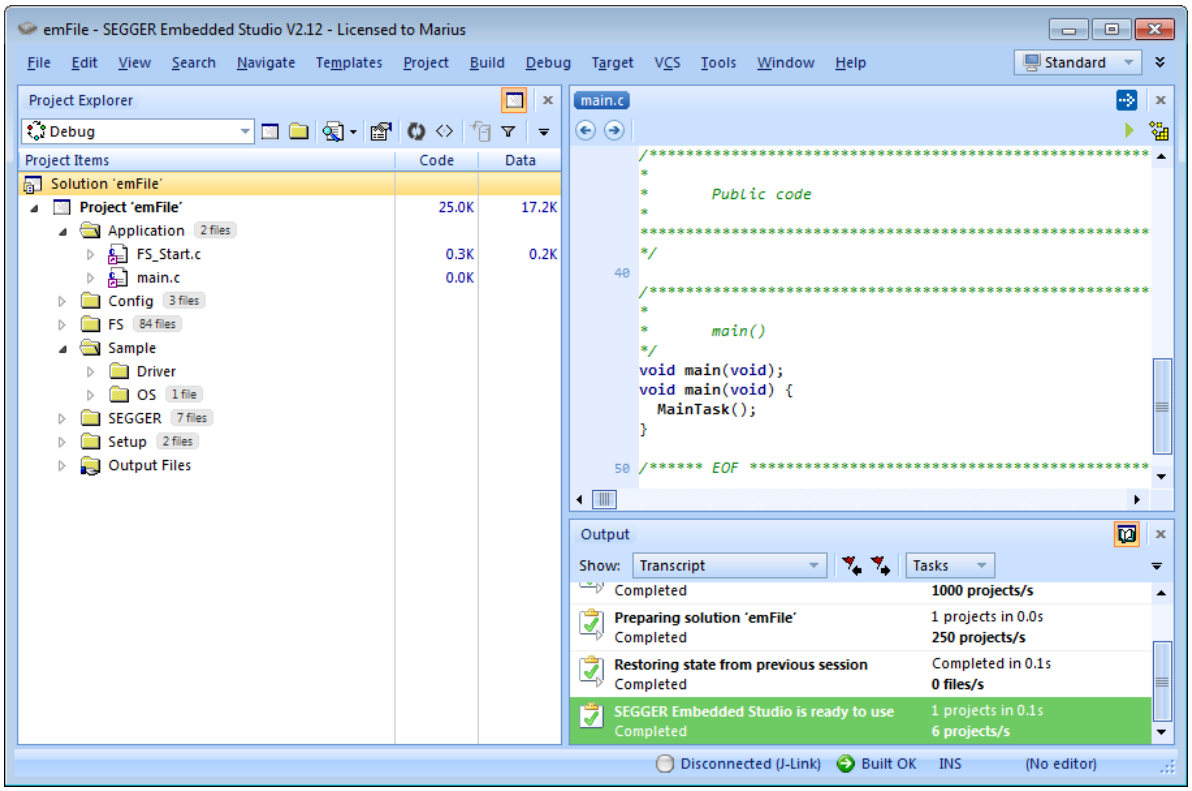


3.4 Step 2: Adding emFile to the start project

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

- `Application`
- `Config`
- `FS`
- `Sample/FS/Driver/RAM`
- `Sample/FS/OS` (Only one file located in this directory has to be included and only if the application uses an RTOS. The included file must be compatible with the RTOS used).
- `SEGGER` (The files `SEGGER_ARM_memcpy_*` are optional. If necessary, the project has to include only one of these files. The included file must be compatible with the IDE used to build the project).

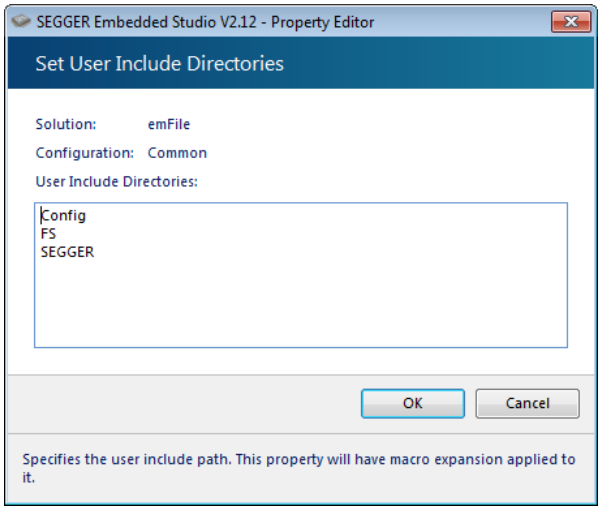
It is recommended to keep the provided folder structure.



3.4.1 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`
- `SEGGER`



3.4.2 Select the start application

For quick and easy testing of your emFile integration, start with the code found in the `Application` folder of the shipment. Exclude all files in the `Application` folder of your project except the supplied `main.c` and `FS_Start.c` files. The application performs the following steps:

- `main.c` calls `MainTask()`.
- `MainTask()` initializes and adds a device to file system.
- Checks if volume is low-level formatted and formats if required.
- Checks if volume is high-level formatted and formats if required.
- Outputs the volume name.
- Calls `FS_GetVolumeFreeSpace()` and outputs the return value -- the available total space of the RAM disk -- to console window.
- Creates and opens a file test with write access (`File.txt`) on the storage device
- Writes 4 bytes of data into the file and closes the file handle or outputs an error message.
- Calls `FS_GetVolumeFreeSpace()` and outputs the return value -- the available free space of the RAM disk -- again to console window.
- Outputs a quit message and runs into an endless loop.

3.4.3 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.

3.5 Step 3: Adding the device driver

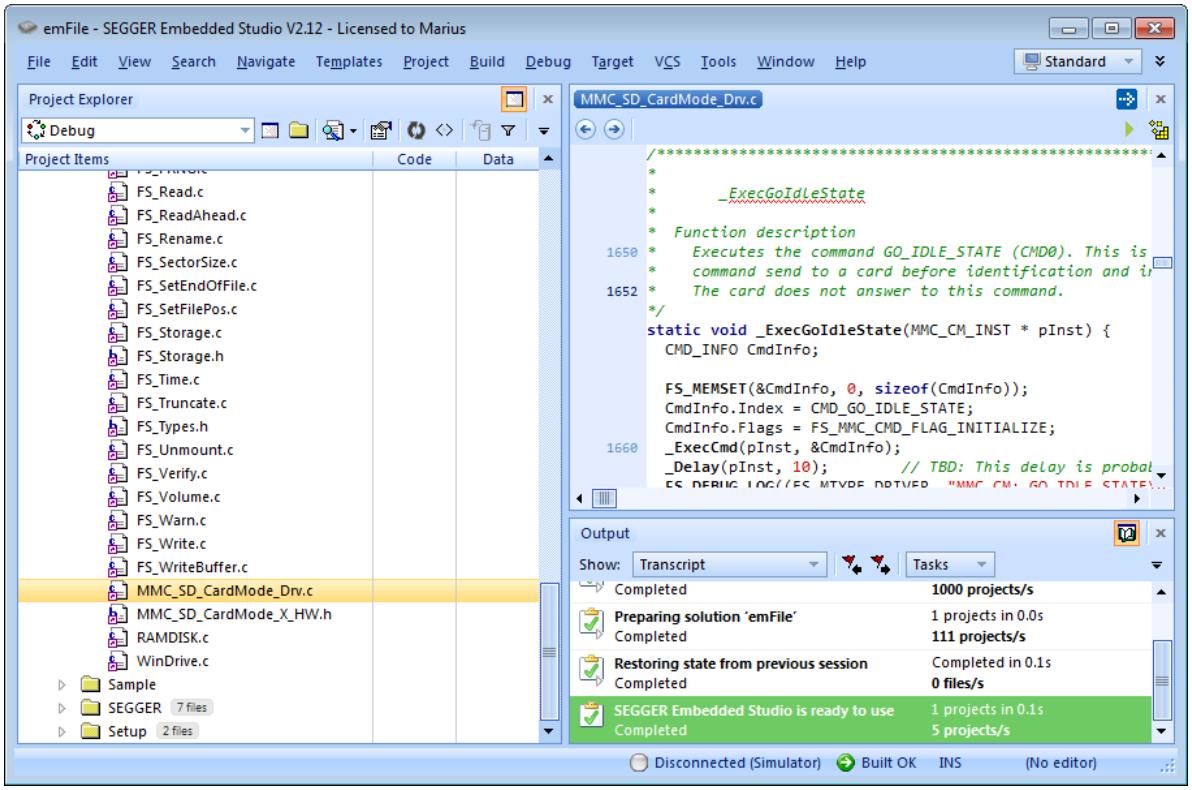
To configure emFile with a device driver, two modifications have to be performed:

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

Each step is explained in the following sections. For example, the implementation of the MMC/SD driver is shown, but all steps can easily be adapted to every other device driver implementation.

3.5.1 Adding the device driver source to project

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



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 flash
- NOR flash with serial devices
- MMC/SD cards
- Compact flash / IDE Drivers which do not require hardware routines are:
- NOR flash with CFI compliant devices
- RAM disk

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 the 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 53.

3.5.2 Adding hardware routines to project

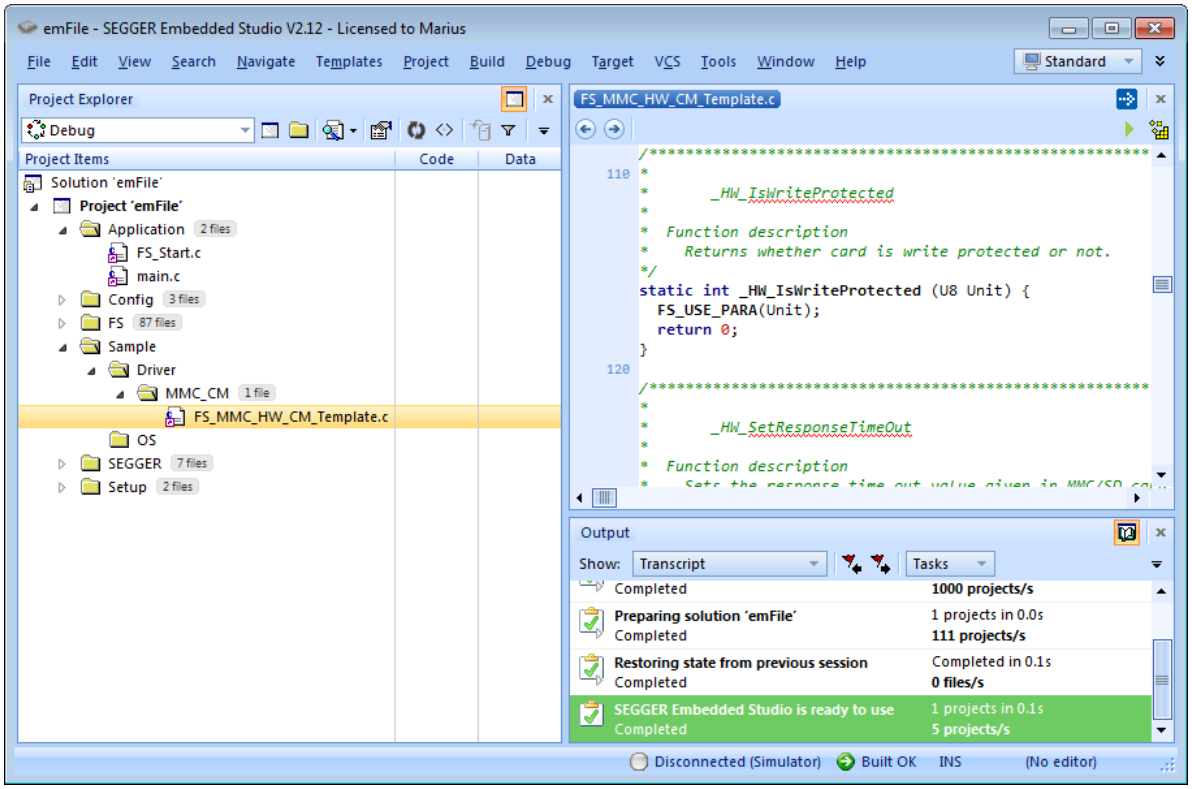
A template with empty function bodies and in most cases one ore more sample implementations are supplied for every driver that requires hardware routines. The easiest way to start is to use one of the ready-to-use samples. The ready-to-use samples can be found in the

subfolders `Sample/FS/Driver/<DRIVER_DIR>` of the emFile shipment where `DRIVER_DIR` is the type of the device driver. 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/FS/Driver/<DRIVER_DIR>` folder; 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 320 for detailed information about the implementation of the driver specific hardware functions, else refer to section *Step 4: Activating the driver* on page 53.

> **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 53 for more information about the activation of the driver with the configuration file.

3.6 Step 4: Activating the driver

After adding the driver source, and if required the hardware function implementation, copy the FS_Config<DRIVER_NAME>.c file (for example, FS_ConfigMMC_CardMode.c for the MMC/SD card driver using card mode) into the Config folder of your emFile work directory and add it to your project.



The configuration files contain, all the runtime configuration functions of the file system. 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 the driver, if required. A driver which does not require hardware routines has to be configured before it can be used.

3.6.1 Modifying the runtime configuration

The following example 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 on most systems.

```
/*********************************************************************
*
*       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 address of the NOR flash device
                                          // to be used as storage
#define FLASH0_START_ADDR  0x40000000     // Start address of the first sector
                                          // to be used as storage. If the entire
                                          // device is used for file system,
                                          // it is identical to
                                          // the base address.
#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 the NOR driver.
  //
  FS_AddDevice(&FS_NOR_Driver);
  //
  // Configure 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 326 for detailed information about the other runtime configurations of the file system. Before compiling and running the sample application with the added driver, you have to exclude `FS_ConfigRAMDisk.c` from the project.

> **Note**
>
> For the drivers which required hardware access 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.

3.7 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) require in general more RAM than the "simple" drivers such as for hard drives, compact flash or MMC/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.

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 too much RAM. Once your file system project is up and running, you can check the real RAM requirement of the driver via the public variable `FS_Global.MemManager.NumBytesAllocated`. Check the value of `FS_Global.MemManager.NumBytesAllocated` after the initialization of the file system and after a volume has been mounted. At this point `FS_Global.MemManager.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_Global.MemManager.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 990.

> **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

This chapter provides a detailed description of emFile API layer.

4.1 General information

Any functions or data structures that are not described in this chapter but are exposed through inclusion of the FS.h header file must be considered private and subject to change.

4.1.1 Volume, file and directory names

A volume, file or directory name is a 0-terminated string the application can use as a parameter to an API function (such as FS_FOpen()) to identify a volume, file or directory the API function has to operate on.

A file or directory name contains the following mandatory and optional elements where the optional elements are surrounded by []:

```
[VolumeName:[UnitNo:]][DirPath]FileName|DirName
```

- VolumeName is the name of the volume on which the file or directory is located. If not specified, the first configured volume is assumed.
- UnitNo is the index of the volume on which the file or directory is located. If not specified, the index 0 is assumed. It is not allowed to specify a unit number if volume name has not been specified.
- DirPath is the complete directory path to an already existing subdirectory. DirPath has to start and end with a directory delimiter character. The names of directories in the path are also separated by directory delimiter. The directory delimiter character can be configured at compile time via FS_DIRECTORY_DELIMITER configuration define. The default directory delimiter is the '\' character. The root directory is assumed if DirPath is not specified. emFile does not support relative file or directory names therefore the application has to always specify the full path a to a file or directory.
- FileName or DirName is the name of the file or directory that has to be accessed by the file system. If volume is formatted as FAT and the support for long file name is not enabled, all file and directory names directory names have to follow the standard FAT 8.3 naming convention. The same applies to the directory names in DirPath. EFS comes with native support for long file names. The length of a file or directory name is limited to 235 valid characters.

A volume name contains the following mandatory and optional elements where the optional elements are surrounded by []:

```
VolumeName:[UnitNo:]
```

VolumeName and UnitNo have the same meaning as described above.

Examples

The following examples specify the same file name assuming that the first driver added to file system is the NAND driver.

- "DirName\FileName.txt"
- "nand:\DirName\FileName.txt"
- "nand:0:\DirName\FileName.txt"

The following examples specify the same volume name assuming that the first driver added to file system is the NOR driver.

- ""
- "nor"
- "nor:0:"

4.2 API function overview

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

Function	Description
File system control functions	
FS_Init()	Starts the file system.
FS_DeInit()	Frees allocated resources.
FS_Mount()	Initializes a volume in default access mode.
FS_MountEx()	Initializes a volume in a specified access mode.
FS_SetAutoMount()	Sets the automatic mount behavior.
FS_Sync()	Saves cached information to storage.
FS_Unmount()	Synchronizes the data and marks the volume as not initialized.
FS_UnmountForced()	Marks the volume as not initialized.
File system configuration functions	
FS_AddDevice()	Adds a driver to file system.
FS_AddPhysDevice()	Adds a device to file system without assigning a volume to it.
FS_AssignMemory()	Assigns a memory pool to the file system.
FS_ConfigFileBufferDefault()	Configures the size and flags for the file buffer.
FS_LOGVOL_Create()	Creates a driver instance.
FS_LOGVOL_AddDevice()	Adds a storage device to a logical volume.
FS_SetFileBufferFlags()	Changes the operating mode of the file buffer.
FS_SetFileBufferFlagsEx()	Changes the operating mode of the file buffer.
FS_SetFileWriteMode()	Configures the file write mode.
FS_SetFileWriteModeEx()	Configures the write mode of a specified volume.
FS_SetMemHandler()	Configures functions for memory management.
FS_SetMaxSectorSize()	Configures the maximum size of a logical sector.
File access functions	
FS_FClose()	Closes an opened file.
FS_FOpen()	Opens an existing file or creates a new one.
FS_FOpenEx()	Opens an existing file or creates a new one.
FS_FRead()	Reads data from file.
FS_FSeek()	Sets the current position in file.
FS_FWrite()	Writes data to file.
FS_FTell()	Returns current position in file.
FS_GetFileSize()	Returns the size of a file.
FS_Read()	Reads data from a file.

Function	Description
FS_SetEndOfFile()	Sets the file size to current file position.
FS_SetFileSize()	Sets the file size to the specified number of bytes.
FS_SyncFile()	Synchronizes file to storage device.
FS_Truncate()	Changes the size of a file.
FS_Verify()	Verifies the file contents.
FS_Write()	Writes data to file.
Operations on files	
FS_CopyFile()	Copies a file.
FS_CopyFileEx()	Copies a file.
FS_GetFileAttributes()	Queries the attributes of a file or directory.
FS_GetFileInfo()	Returns information about a file or directory.
FS_GetFileTime()	Returns the creation time of a file or directory.
FS_GetFileTimeEx()	Gets the timestamp of a file or directory.
FS_ModifyFileAttributes()	Sets / clears the attributes of a file or directory.
FS_Move()	Moves a file or directory to another location.
FS_Remove()	Removes a file.
FS_Rename()	Changes the name of a file or directory.
FS_SetFileAttributes()	Modifies all the attributes of a file or directory.
FS_SetFileBuffer()	Assigns a file buffer to an opened file.
FS_SetFileTime()	Sets the creation time of a file or directory.
FS_SetFileTimeEx()	Sets the timestamp of a file or directory.
FS_WipeFile()	Overwrites the contents of a file with random data.
Directory functions	
FS_CreateDir()	Creates a directory including any missing directories from path.
FS_DeleteDir()	Removes a directory and its contents.
FS_FindClose()	Ends a directory scanning operation.
FS_FindFirstFile()	Initiates a directory scanning operation and returns information about the first file or directory.
FS_FindNextFile()	Returns information about the next file or directory in a directory scanning operation.
FS_FindNextFileEx()	Returns information about the next file or directory in a directory scanning operation.
FS_MkDir()	Creates a directory.
FS_RmDir()	Removes a directory.
Formatting a medium	
FS_Format()	Performs a high-level format.
FS_FormatLLIfRequired()	Performs a low-level format.

Function	Description
FS_FormatLow()	Performs a low-level format.
FS_IsHLFormatted()	Checks if a volume is high-level formatted or not.
FS_IsLLFormatted()	Returns whether a volume is low-level formatted or not.
File system structure checking	
FS_CheckAT()	Verifies the consistency of the allocation table.
FS_CheckDir()	Verifies the consistency of a single directory.
FS_CheckDisk()	Checks the file system for corruption.
FS_CheckDisk_ErrCode2Text()	Returns a human-readable text description of a disk checking error code.
FS_InitCheck()	Initializes a non-blocking disk checking operation.
File system extended functions	
FS_ConfigEOFErrorSuppression()	Enables / disables the reporting of end-of-file condition as error.
FS_ConfigPOSIXSupport()	Enables / disables support for the POSIX-like behavior.
FS_ConfigWriteVerification()	Enables / disables the verification of the written data.
FS_CreateMBR()	Updates the Master Boot Record (MBR) of a volume.
FS_FileTimeToTimeStamp()	Converts a broken-down date and time specification to a timestamp.
FS_FreeSectors()	Informs the device driver about unused sectors.
FS_GetFileId()	Calculates a value that uniquely identifies a file.
FS_GetFileWriteMode()	Returns the write mode.
FS_GetFileWriteModeEx()	Returns the write mode configured for a specified volume.
FS_GetFSType()	Returns the type of file system assigned to volume.
FS_GetMaxSectorSize()	Queries the maximum configured logical sector size.
FS_GetMemInfo()	Returns information about the memory management.
FS_GetMountType()	Returns information about how a volume is mounted.
FS_GetNumFilesOpen()	Queries the number of opened file handles.
FS_GetNumFilesOpenEx()	Queries the number of opened file handles on a volume.
FS_GetNumVolumes()	Queries the number of configured volumes.
FS_GetPartitionInfo()	Returns information about a MBR partition.
FS_GetVolumeFreeSpace()	Returns the free space available on a volume.

Function	Description
FS_GetVolumeFreeSpaceFirst()	Initiates the search for free space.
FS_GetVolumeFreeSpaceKB()	Returns the free space available on a volume.
FS_GetVolumeFreeSpaceNext()	Continues the search for free space.
FS_GetVolumeInfo()	Returns information about a volume.
FS_GetVolumeInfoEx()	Returns information about a volume.
FS_GetVolumeLabel()	Returns the label of the volume.
FS_GetVolumeName()	Returns the name of a volume.
FS_GetVolumeSize()	Returns the size of a volume.
FS_GetVolumeSizeKB()	Returns the size of a volume.
FS_GetVolumeStatus()	Returns the presence status of a volume.
FS_IsVolumeMounted()	Checks if a volume is mounted.
FS_Lock()	Claims exclusive access to file system.
FS_LockVolume()	Claims exclusive access to a volume.
FS_SetBusyLEDCallback()	Registers a callback for busy status changes of a volume.
FS_SetCharSetType()	Configures the character set that is used for the file and directory names.
FS_SetFSType()	Sets the type of file system a volume.
FS_SetMemCheckCallback()	Registers a callback for checking of 0-copy operations.
FS_SetTimeDateCallback()	Configures a function that the file system can use to get the current time and date.
FS_SetVolumeAlias()	Assigns an alternative name for a volume.
FS_SetVolumeLabel()	Modifies the label of a volume.
FS_TimeStampToFileTime()	Converts a timestamp to a broken-down date and time specification.
FS_Unlock()	Releases the exclusive access to file system.
FS_UnlockVolume()	Releases the exclusive access to a volume.
Storage layer functions	
FS_STORAGE_Clean()	Performs garbage collection on a volume.
FS_STORAGE_CleanOne()	Performs garbage collection on a volume.
FS_STORAGE_DeInit()	Frees the resources allocated by the storage layer.
FS_STORAGE_FreeSectors()	Informs the driver about unused sectors.
FS_STORAGE_GetCleanCnt()	Calculates the number of garbage collection sub-operations.
FS_STORAGE_GetCounters()	Returns the values of statistical counters.
FS_STORAGE_GetDeviceInfo()	Returns information about the storage device.
FS_STORAGE_GetSectorUsage()	Returns information about the usage of a logical sector.
FS_STORAGE_Init()	Initializes the storage layer.
FS_STORAGE_ReadSector()	Reads the data of one logical sector.

Function	Description
FS_STORAGE_ReadSectors()	Reads the data of one or more logical sectors.
FS_STORAGE_RefreshSectors()	Reads the contents of a logical sector and writes it back.
FS_STORAGE_ResetCounters()	Sets all statistical counters to 0.
FS_STORAGE_SetOnDeviceActivityCallback()	Registers a function to be called on any logical sector read or write operation.
FS_STORAGE_Sync()	Writes cached information to volume.
FS_STORAGE_SyncSectors()	Synchronize the contents of one or more logical sectors.
FS_STORAGE_Unmount()	Synchronizes a volume and marks it as not initialized.
FS_STORAGE_UnmountForced()	Marks a volume it as not initialized.
FS_STORAGE_WriteSector()	Modifies the data of a logical sector.
FS_STORAGE_WriteSectors()	Modifies the data of one or more logical sector.
FAT related functions	
FS_FAT_ConfigDirtyFlagUpdate()	Enables / disables the update of the flag that indicates if the volume has been unmounted correctly.
FS_FAT_ConfigFATCopyMaintenance()	Enables / disables the update of the second allocation table.
FS_FAT_ConfigFSInfoSectorUse()	Enables / disables the usage of information from the FSInfo sector.
FS_FAT_ConfigROFileMovePermission()	Enables / disables the permission to move (and rename) files and directories with the read-only file attribute set.
FS_FAT_DisableLFN()	Disables the support for long file names.
FS_FAT_FormatSD()	Formats the volume according to specification of SD Association.
FS_FAT_GrowRootDir()	Increases the size of the root directory.
FS_FAT_SetLFNConverter()	Configures how long file names are to be encoded and decoded.
FS_FAT_SupportLFN()	Enables the support for long file names.
EFS related functions	
FS_EFS_ConfigCaseSensitivity()	Configures how the file names are compared.
FS_EFS_ConfigStatusSectorSupport()	Enables or disables the usage of information from the status sector.
FS_EFS_SetFileNameConverter()	Configures how file names are to be encoded.
Error-handling functions	
FS_ClearErr()	Clears error status of a file handle.
FS_ErrorNo2Text()	Returns a human-readable text description of an API error code.
FS_FEof()	Returns if end of file has been reached.
FS_FError()	Return error status of a file handle.
Configuration checking functions	

Function	Description
FS_CONF_GetDebugLevel()	Returns the level of debug information configured for the file system.
FS_CONF_GetDirectoryDelimiter()	Returns the character that is configured as delimiter between the directory names in a file path.
FS_CONF_GetMaxPath()	Returns the configured maximum number of characters in a path to a file or directory.
FS_CONF_GetNumVolumes()	Returns the maximum number of volumes configured for the file system.
FS_CONF_GetOSLocking()	Returns the type of task locking configured for the file system.
FS_CONF_IsCacheSupported()	Checks if the file system is configured to support the sector cache.
FS_CONF_IsDeInitSupported()	Checks if the file system is configured to support deinitialization.
FS_CONF_IsEFSSupported()	Checks if the file system is configured to support the EFS file system.
FS_CONF_IsEncryptionSupported()	Checks if the file system is configured to support encryption.
FS_CONF_IsFATSupported()	Checks if the file system is configured to support the FAT file system.
FS_CONF_IsFreeSectorSupported()	Checks if the file system is configured to support the “free sector” command.
FS_CONF_IsJournalSupported()	Checks if the file system is configured to support journaling.
FS_CONF_IsTrialVersion()	Checks if the file system has been configured as a trial (limited) version.
FS_GetVersion()	Returns the version number of the file system.
Obsolete functions	
FS_AddOnExitHandler()	Registers a deinitialization callback.
FS_CloseDir()	Closes a directory.
FS_ConfigOnWriteDirUpdate()	Configures if the directory entry has be updated after writing to file.
FS_DirEnt2Attr()	Loads attributes of a directory entry.
FS_DirEnt2Name()	Loads the name of a directory entry.
FS_DirEnt2Size()	Loads the size of a directory entry.
FS_DirEnt2Time()	Loads the time stamp of a directory entry.
FS_GetNumFiles()	API function.
FS_OpenDir()	API function.
FS_ReadDir()	Reads next directory entry in directory.
FS_RewindDir()	Sets pointer for reading the next directory entry to the first entry in the directory.

4.3 File system control functions

4.3.1 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 the access of the storage device in a multi-tasking environment. This function has to be called before any other file system API function.

Example

```
#include "FS.h"

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


4.3.2 FS_DeInit()

Description

Frees allocated resources.

Prototype

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

Additional information

This function is optional. `FS_DeInit()` frees all resources that are allocated by the file system after initialization. Also, all static variables of all file system layers are reset in order to guarantee that the file system remains in a known state after deinitialization.The application can call this function only after it called `FS_Init()`.

This function has to be used when the file system is reset at runtime. For example this is the case if the system uses a software reboot which reinitializes the target application.

This function is available if the emFile sources are compiled with the `FS_SUPPORT_DEINIT` configuration define set to 1.

Example

```
#include "FS.h"

void SampleDeInit(void) {
  FS_Init();
  //
  // Access the file system...
  //
  FS_DeInit();
  //
  // The file system cannot be accessed anymore.
  //
}
```


4.3.3 FS_Mount()

Description

Initializes a volume in default access mode.

Prototype

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

Parameters

Parameter	Description
sVolumeName	The name of a volume. If the empty string is specified, the first device in the volume table is used.

Return value

= 0	Volume is not mounted.
= FS_MOUNT_RO	Volume is mounted read only.
= FS_MOUNT_RW	Volume is mounted read/write.
< 0	Error code indicating the failure reason. Refer to FS_ErrorNo2Text().

Additional information

The storage device has to be mounted before being accessed for the first time after file system initialization. The file system is configured by default to automatically mount the storage device at the first access in read / write mode. This function can be used to explicitly mount the storage device if the automatic mount behavior has been disabled via FS_SetAutoMount(). Refer to FS_SetAutoMount() for an overview of the different automatic mount types.

Example

```
#include "FS.h"

void SampleMount(void) {
  FS_Init();
  //
  // Mount default volume in read / write mode.
  //
  FS_Mount("");
  //
  // Access the data stored on the file system.
  //
}
```


4.3.4 FS_MountEx()

Description

Initializes a volume in a specified access mode.

Prototype

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

Parameters

Parameter	Description
sVolumeName	The name of the volume. If the empty string is specified, the first device in the volume table is used.
MountType	Indicates how the volume has to be mounted. • FS_MOUNT_RO Read only access. • FS_MOUNT_RW Read / write access.

Return value

= 0	Volume is not mounted.
= FS_MOUNT_RO	Volume is mounted read only.
= FS_MOUNT_RW	Volume is mounted read/write.
< 0	Error code indicating the failure reason. Refer to FS_ErrorNo2Text().

Additional information

Performs the same operation as FS_Mount() while it allows the application to specify how the storage device has to be mounted.

Example

```
#include "FS.h"

void SampleMountEx(void) {
  FS_Init();
  //
  // Mount default volume in read-only mode.
  //
  FS_MountEx("", FS_MOUNT_RO);
  //
  // Access the data stored on the file system.
  //
}
```


4.3.5 FS_SetAutoMount()

Description

Sets the automatic mount behavior.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Pointer to a string containing the name of the volume. If the empty string is specified, the first device in the volume table is used.
MountType	Indicates how the volume has to be mounted. • FS_MOUNT_RO Allows to automatically mount the volume in read only mode. • FS_MOUNT_RW Allows to automatically mount the volume in read / write mode. • 0 Disables the automatic mount operation for the volume.

Additional information

By default, the file system is configured to automatically mount all volumes in read / write mode and this function can be used to change the default automatic mount type or to disable the automatic mounting.

Example

```
#include "FS.h"

void SampleSetAutoMount(void) {
  FS_SetAutoMount("", FS_MOUNT_R);    // Mount default volume in read-only mode.
}
```


4.3.6 FS_Sync()

Description

Saves cached information to storage.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Pointer to a string containing the name of the volume to be synchronized. If the empty string is specified, the first configured volume is used.

Return value

= 0 OK, volume synchronized
≠ 0 Error code indicating the failure reason. Refer to FS_ErrorNo2Text() for more information.

Additional information

The function write the contents of write buffers and updates the management information of all opened file handles to storage device. All the file handles are left open. If configured, FS_Sync() also writes to storage the changes present in the write cache and in the journal. FS_Sync() can be called from the same task as the one writing data or from a different task.

Example

```
#include "FS.h"

void SampleSync(void) {
  FS_Sync("");      // Synchronize the default volume.
}
```


4.3.7 FS_Unmount()

Description

Synchronizes the data and marks the volume as not initialized.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Pointer to a string containing the name of the volume to be unmounted. If the empty string is specified, the first device in the volume table is used. Can not be NULL.

Additional information

This function closes all open files and synchronizes the volume, that is writes all cached data to storage device. FS_Unmount() hast to be called before a storage device is removed to make sure that all the information cached by the file system is updated to storage device. This function is also useful when shutting down a system.

The volume is initialized again at the next call to any other file system API function that requires access to storage device. The application can also explicitly initialize the volume via FS_Mount() or FS_MountEx().

Example

```
#include "FS.h"

void SampleUnmount(void) {
  FS_Unmount("");     // Unmount the default volume.
}
```


4.3.8 FS_UnmountForced()

Description

Marks the volume as not initialized.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Pointer to a string containing the name of the volume to be unmounted. If the empty string is specified, the first device in the volume table is used.

Additional information

This function performs the same operations as FS_Unmount(). FS_UnmountForced() has to be called if a storage device has been removed before it could be regularly unmounted. When using FS_UnmountForced() there is no guarantee that the information cached by the file system is updated to storage.

The volume is initialized again at the next call to any other file system API function that requires access to storage device. The application can also explicitly initialize the volume via FS_Mount() or FS_MountEx().

Opened file handles are only marked as invalid but they are not closed. The application has to close them explicitly by calling FS_FClose().

Example

```
#include "FS.h"
#include "FS_OS.h"

void SampleUnmountForced(void) {
  int IsPresentNew;
  int IsPresent;

  IsPresent = FS_GetVolumeStatus("");
  while (1) {
    IsPresentNew = FS_GetVolumeStatus("");
    //
    // Check if the presence status of the storage device has been changed.
    //
    if (IsPresentNew != IsPresent) {
      if (IsPresentNew == FS_MEDIA_IS_PRESENT) {
        FS_Mount("");
      } else  {
        FS_UnmountForced("");
      }
      IsPresent = IsPresentNew;
    }
    FS_X_OS_Delay(500);
  }
}
```


4.3.9 Volume mounting modes

Description

Modes for mounting a volume.

Definition

```
#define FS_MOUNT_RO    FS_MOUNT_R
#define FS_MOUNT_RW    (FS_MOUNT_R | FS_MOUNT_W)
```

Symbols

Definition	Description
FS_MOUNT_RO	Read-only. Data can only be read from storage device.
FS_MOUNT_RW	Read / Write. Data can be read form and written to storage device.

4.4 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()`.

4.4.1 FS_AddDevice()

Description

Adds a driver to file system.

Prototype

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

Parameters

Parameter	Description
pDevType	in Pointer to a function table identifying the driver that has to be added.

Return value

≠ 0 OK, driver added.
= 0 An error occurred.

Additional information

This function can be used to add a device or a logical driver to file system. The application has to add at least one driver to file system.

The function performs the following operations:

- Adds a physical device. This initializes the driver, allowing the driver to identify the storage device if required and to allocate memory for driver level management of the storage device. This makes sector operations possible.
- Assigns a logical volume to physical device. This makes it possible to mount the storage device, making it accessible for the file system and allowing file operations to be performed on it.

Example

```
#include "FS.h"

void FS_X_AddDevices(void) {
  //
  // Basic configuration of the file system...
  //

  //
  // Add and configure a device driver for NAND flash.
  //
  FS_AddDevice(&FS_NAND_UNI_Driver);

  //
  // Additional configuration of the file system...
  //
}
```


4.4.2 FS_AddPhysDevice()

Description

Adds a device to file system without assigning a volume to it.

Prototype

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

Parameters

Parameter	Description
pDevType	in Pointer to a function table identifying the driver that has to be added.

Return value

= 0 OK, storage device added.
≠ 0 An error occurred.

Additional information

This function can be used to add a device or a logical driver to file system. It works similarly to FS_AddDevice() with the difference that it does not assign a logical volume to storage device. This means that the storage device is not directly accessible by the application via the API functions of the file system. An additional logical driver is required to be added via FS_AddDevice() to make the storage device visible to application.

FS_AddPhysDevice() initializes 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.

4.4.3 FS_AssignMemory()

Description

Assigns a memory pool to the file system.

Prototype

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

Parameters

Parameter	Description
pData	in A pointer to the start of the memory region assigned to file system.
NumBytes	Size of the memory pool assigned.

Additional information

emFile comes with a simple semi-dynamic internal memory manager that is used to satisfy the runtime memory requirements of the file system. FS_AssignMemory() can be used to provide a memory pool to the internal memory manager of the file system. If not enough memory is assigned, the file system calls FS_X_Panic() in debug builds which by default halts the execution of the application. The actual number of bytes allocated is stored in the global variable FS_Global.MemManager.NumBytesAllocated. This variable can be used to fine-tune the size of the memory pool.

emFile supports also the use of an external memory manager (e.g. via malloc() and free() functions of the standard C library). The selection between the internal and the external memory management has to be done at compile time via the FS_SUPPORT_EXT_MEM_MANAGER define. The configuration of the memory management functions is done via FS_SetMemHandler().

This function has to be called in the initialization phase of the file system; typically in FS_X_AddDevices(). The support for internal memory management has to be enabled at compile time by setting the FS_SUPPORT_EXT_MEM_MANAGER define to 0. FS_AssignMemory() does nothing if the FS_SUPPORT_EXT_MEM_MANAGER define is set to 1.

Example

```
#include "FS.h"

#define ALLOC_SIZE    0x1000

static U32 _aMemBlock[ALLOC_SIZE / 4];

void FS_X_AddDevices(void) {
  FS_AssignMemory(_aMemBlock, sizeof(_aMemBlock));
  //
  // Perform additional file system configuration.
  //
}
```


4.4.4 FS_ConfigFileBufferDefault()

Description

Configures the size and flags for the file buffer.

Prototype

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

Parameters

Parameter	Description
BufferSize	Size of the file buffer in bytes.
Flags	File buffer operating mode. • 0 Read file buffer. • FS_FILE_BUFFER_WRITE Read / write file buffer. • FS_FILE_BUFFER_ALIGNED Logical sector boundary alignment.

Return value

= 0 OK, the file buffer has been configured.
≠ 0 Error code indicating the failure reason.

Additional information

The function has to be called only once, in FS_X_AddDevices(). If called after FS_Init() the function does nothing and generates a warning.

The file system allocates a file buffer of BufferSize bytes for each file the application opens. The operating mode of the file buffer can be changed at runtime via FS_SetFileBufferFlags(). If file buffers of different sizes are required FS_SetFileBuffer() should be used instead.

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

FS_SetFileBuffer() is available if the emFile sources are compiled with the FS_SUPPORT_FILE_BUFFER configuration define set to 1.

Example

```
#include "FS.h"

#define ALLOC_SIZE    0x1000

static U32 _aMemBlock[ALLOC_SIZE / 4];

void FS_X_AddDevices(void) {
  FS_AssignMemory(_aMemBlock, sizeof(_aMemBlock));
  //
  // Set the file buffer of 512 bytes for read and write operations.
  // The file buffer is allocated at runtime for each opened file
  // from _aMemBlock.
  //
#if (FS_USE_FILE_BUFFER || FS_SUPPORT_FILE_BUFFER)
  FS_ConfigFileBufferDefault(512, FS_FILE_BUFFER_WRITE);
#endif
}
```


4.4.5 FS_LOGVOL_Create()

Description

Creates a driver instance.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to create.

Return value

= 0 OK, volume has been created.
≠ 0 An error occurred.

Additional information

This function creates an instance of 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 from or writing to the logical volume. Logical volumes are typically used if multiple flash devices (NOR or NAND) are present, but they should be presented to the application in the same way as a single device with the combined capacity of both.

sVolumeName is the name that has to be assigned to the logical volume. This is the volume name that is passed to some of the FS API functions and that has to be used in a file path.

FS_LOGVOL_Create() does nothing if the module is configured to work in driver mode by setting the FS_LOGVOL_SUPPORT_DRIVER_MODE define option to 1. In this case a logical driver is created by adding it via FS_AddDevice() to file system.

Normally, all devices are added individually using FS_AddDevice(). This function adds the devices physically as well as logically to the file system. 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 performed: 1. The storage device has to be physically added to the file system using FS_AddPhysDevice(). 2. A logical volume has to be created using FS_LOGVOL_Create(). 3. The devices which are physically added to the file system have to be added to the logical volume using FS_LOGVOL_AddDevice().

4.4.6 FS_LOGVOL_AddDevice()

Description

Adds a storage device to a logical volume.

Prototype

```
int FS_LOGVOL_AddDevice(const char           * sVolumeName,
                        const FS_DEVICE_TYPE * pDeviceType,
                              U8               DeviceUnit,
                              U32              StartSector,
                              U32              NumSectors);
```

Parameters

Parameter	Description
sVolumeName	Name of the logical volume.
pDeviceType	Type of the storage device that has to be added.
DeviceUnit	Index of the storage device that has to be added (0-based).
StartSector	Index of the first sector that has to be used as storage (0-based).
NumSectors	Number of sectors that have to be used as storage.

Return value

= 0 OK, storage device added.
≠ 0 An error occurred.

Additional information

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

This function does nothing if FS_LOGVOL_SUPPORT_DRIVER_MODE is set to 1.

4.4.7 FS_SetFileBufferFlags()

Description

Changes the operating mode of the file buffer.

Prototype

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

Parameters

Parameter	Description
pFile	Handle to opened file.
Flags	File buffer operating mode. • 0 Read file buffer. • FS_FILE_BUFFER_WRITE Read / write file buffer. • FS_FILE_BUFFER_ALIGNED Logical sector boundary alignment.

Return value

= 0 OK, file buffer flags changed.
≠ 0 Error code indicating the failure reason.

Additional information

This function can only be called immediately after FS_FOpen(), to change the operating mode of the file buffer (read or read / write).

FS_SetFileBufferFlags() is available if the emFile sources are compiled with the FS_SUPPORT_FILE_BUFFER configuration define set to 1.

Example

```
#include "FS.h"

void SampleSetFileBufferFlags(void) {
  FS_FILE * pFile;
  U8        abData[16];

  pFile = FS_FOpen("Test.txt", "w");
  if (pFile != NULL) {
    memset(abData, 'a', sizeof(abData));
    FS_SetFileBufferFlags(pFile, FS_FILE_BUFFER_WRITE);
    FS_Write(pFile, abData, sizeof(abData));
    FS_FClose(pFile);
  }
}
```


4.4.8 FS_SetFileBufferFlagsEx()

Description

Changes the operating mode of the file buffer.

Prototype

```
int FS_SetFileBufferFlagsEx(const char * sVolumeName,
                            int    Flags);
```

Parameters

Parameter	Description
sVolumeName	Name of the volume for which to set the flags.
Flags	File buffer operating mode. • 0 Read file buffer. • FS_FILE_BUFFER_WRITE Read / write file buffer. • FS_FILE_BUFFER_ALIGNED Logical sector boundary alignment.

Return value

= 0 OK, operating mode changed.
≠ 0 Error code indicating the failure reason.

Additional information

This function can be used to change the operating mode of the file buffer for the files that are located on a specific volume.

FS_SetFileBufferFlagsEx() is available if the emFile sources are compiled with the FS_SUPPORT_FILE_BUFFER configuration define set to 1.

4.4.9 FS_SetFileWriteMode()

Description

Configures the file write mode.

Prototype

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

Parameters

Parameter	Description
WriteMode	Specifies how to write to file: • FS_WRITEMODE_SAFE Updates the allocation table and the directory entry at each write to file operation. • FS_WRITEMODE_MEDIUM Updates the allocation table at each write to file operation. • FS_WRITEMODE_FAST The allocation table and directory entry are updated when the file is closed.

Additional information

This function can be called to configure which mode the file system has to use when writing to a file. The file system uses by default FS_WRITEMODE_SAFE which allows the maximum fail-safe behavior, since the allocation table and the directory entry is updated on every write operation to file.

If FS_WRITEMODE_FAST is set, the update of the allocation table is performed 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 with the first one occupied by the file. This cluster chain is remembered, so that if the file grows and needs an additional cluster, the allocation doesn't have to be read again in order to find the next free cluster. The allocation table 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 large amounts of data, FS_WRITEMODE_FAST allows maximum performance, since usually the file system has to search for a free cluster in the allocation table and link it with the last one occupied by the file. In worst case, multiple sectors of the allocation table have to be read in order to find a free cluster.

Example

```
#include "FS.h"

void FS_X_AddDevices(void) {
  //
  // Configure the file system to write as fast as possible
  // to all files on all volumes.
  //
  FS_SetFileWriteMode(FS_WRITEMODE_FAST);
  //
  // Perform other file system configuration...
  //
}
```


4.4.10 FS_SetFileWriteModeEx()

Description

Configures the write mode of a specified volume.

Prototype

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

Parameters

Parameter	Description
WriteMode	Specifies how to write to file: • FS_WRITEMODE_SAFE Updates the allocation table and the directory entry at each write to file operation. • FS_WRITEMODE_MEDIUM Updates the allocation table at each write to file operation. • FS_WRITEMODE_FAST The allocation table and directory entry are updated when the file is closed.
sVolumeName	Identifies the volume for which the write mode has to be changed.

Additional information

When not explicitly set using this function the write mode of a volume is the write mode set via FS_SetFileWriteMode() or the default write mode (FS_WRITEMODE_SAFE). FS_SetFileWriteModeEx() is typically on file system configurations using multiple volumes that require different write modes. For example on a file system configured to use two volumes where one volume has to be configured for maximum write performance (FS_WRITEMODE_FAST) while on the other volume the write operation has to be fail-safe (FS_WRITEMODE_SAFE).

Refer to FS_SetFileWriteMode() for detailed information about the different write modes.

Example

```
#include "FS.h"

void FS_X_AddDevices(void) {
  //
  // Configure the file system to write as fast as possible
  // to all files on the "nand:0:" volume.
  //
  FS_SetFileWriteModeEx(FS_WRITEMODE_FAST, "nand:0");
  //
  // Perform other file system configuration...
  //
}
```


4.4.11 FS_SetMemHandler()

Description

Configures functions for memory management.

Prototype

```
void FS_SetMemHandler(FS_MEM_ALLOC_CALLBACK * pfAlloc,
                      FS_MEM_FREE_CALLBACK  * pfFree);
```

Parameters

Parameter	Description
pfAlloc	Pointer to a function that allocates memory (e.g. malloc()). It cannot be NULL.
pfFree	Pointer to a function that frees memory (e.g. free()). It cannot be NULL.

Additional information

The application can use this function to configure functions for the memory management. The file system calls pfAlloc to allocate memory and pfFree to release the allocated memory.

This function has to be called in the initialization phase of the file system; typically in FS_X_AddDevices(). The support for external memory management has to be enabled at compile time by setting the FS_SUPPORT_EXT_MEM_MANAGER define to 1. FS_SetMemHandler() does nothing if FS_SUPPORT_EXT_MEM_MANAGER is set to 0 (default).

Example

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

void FS_X_AddDevices(void) {
#if FS_SUPPORT_EXT_MEM_MANAGER
  //
  // Configure functions for dynamic memory allocation.
  //
  FS_SetMemHandler((FS_MEM_ALLOC_CALLBACK *)malloc, free);
#endif // FS_SUPPORT_EXT_MEM_MANAGER
  //
  // Configure the file system...
  //
}
```


4.4.12 FS_SetMaxSectorSize()

Description

Configures the maximum size of a logical sector.

Prototype

```
int FS_SetMaxSectorSize(unsigned MaxSectorSize);
```

Parameters

Parameter	Description
MaxSectorSize	Number of bytes in the logical sector.

Return value

= 0 OK, the sector size had been set.
≠ 0 Error code indicating the failure reason.

Additional information

The file system uses internal RAM buffers to store the data of logical sectors it accesses. The storage devices added to file system can have different logical sector sizes. Since the size of the logical sectors is not known at the time the internal RAM buffers are allocated the application has to call FS_SetMaxSectorSize() to specify the size of the largest logical sector used by the configured drivers.

The default value for the maximum size of a logical sector is 512 bytes. The size of the logical sector supported by a driver can be found in the section that describes the specific driver.

FS_SetMaxSectorSize() can be called only at file system initialization in FS_X_AddDevices().

Example

```
#include "FS.h"

void FS_X_AddDevices(void) {
  //
  // This function call is typically required when
  // a NAND flash with 2048 byte pages is used as
  // storage device.
  //
  FS_SetMaxSectorSize(2048);
}
```


4.4.13 FS_MEM_ALLOC_CALLBACK

Description

Type of function called by the file system to allocate memory dynamically.

Type definition

```
typedef void * FS_MEM_ALLOC_CALLBACK(U32 NumBytes);
```

Parameters

Parameter	Description
NumBytes	Number of bytes to be allocated.

Return value

≠ NULL OK, pointer to the allocated memory block.
= NULL Error, could not allocate memory.

Additional information

The callback function has the same signature as the malloc() standard C function. The application can register the callback function via FS_SetMemHandler().

4.4.14 FS_MEM_FREE_CALLBACK

Description

Type of function called by the file system to release dynamically allocated memory.

Type definition

```
typedef void FS_MEM_FREE_CALLBACK(void * pData);
```

Parameters

Parameter	Description
pData	Memory block to be released.

Additional information

The callback function has the same signature as the free() standard C function. The application can register the callback function via FS_SetMemHandler(). pMem points to a memory block that was allocated using a call to pfAlloc callback function registered via FS_SetMemHandler().

4.4.15 FS_WRITEMODE

Description

Modes of writing to file.

Type definition

```
typedef enum {
  FS_WRITEMODE_SAFE,
  FS_WRITEMODE_MEDIUM,
  FS_WRITEMODE_FAST,
  FS_WRITEMODE_UNKNOWN
} FS_WRITEMODE;
```

Enumeration constants

Constant	Description
FS_WRITEMODE_SAFE	Allows maximum fail-safe behavior. The allocation table and the directory entry are updated after each write access to file. This write mode provides the slowest performance.
FS_WRITEMODE_MEDIUM	Medium fail-safe. The allocation table is updated after each write access to file. The directory entry is updated only if file is synchronized that is when FS_Sync(), FS_FClose(), or FS_SyncFile() is called.
FS_WRITEMODE_FAST	This write mode provided the maximum performance. The directory entry is updated only if the file is synchronized that is when FS_Sync(), FS_FClose(), or FS_SyncFile() is called. The allocation table is modified only if necessary.
FS_WRITEMODE_UNKNOWN	End of enumeration. For internal use only.

4.5 File access functions

The functions in this section can be used to access the data stored in a file.

4.5.1 FS_FClose()

Description

Closes an opened file.

Prototype

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

Parameters

Parameter	Description
pFile	Handle to the file to be closed.

Return value

= 0 OK, file handle has been successfully closed.
≠ 0 Error code indicating the failure reason. Refer to `FS_ErrorNo2Text()`.

Example

```
#include "FS.h"

void SampleFClose(void) {
  FS_FILE * pFile;

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


4.5.2 FS_FOpen()

Description

Opens an existing file or creates a new one.

Prototype

```
FS_FILE *FS_FOpen(const char * sFileName,
                  const char * sMode);
```

Parameters

Parameter	Description
sFileName	Name of the file to be opened or created. It is a 0-terminated string that cannot be NULL.
sMode	Indicates how the file should be opened. It is a 0-terminated string that cannot be NULL.

Return value

≠ 0 OK, pointer to a file handle that identifies the opened file.
= 0 Error, unable to open the file.

Additional information

The sMode parameter can take one of the following values:

sMode	Description
"r" or "rb"	Open fies for reading.
"w" or "wb"	Truncates to zero length or creates file for writing.
"a" or "ab"	Appends; opens / creates file for writing at end of file.
"rb"	Appends; opens / creates file for writing at end of file.
"r+", "r+b" or "rb+"	Opens file for update (reading and writing).
"w+", "w+b" or "wb+"	Truncates to zero length or creates file for update.
"a+", "a+b" or "ab+"	Appends; opens / creates file for update, writing at end of file.

For more details about FS_FOpen(), refer to the ANSI C documentation of the fopen() function.

The file system does not distinguish between binary and text mode; the files are always accessed in binary mode.

In order to use long file names with FAT, the FS_FAT_SupportLFN() has to be called before after the file system is initialized.

FS_FOpen() accepts file names encoded in UTF-8 format. This feature is disabled by default. To enable it the file system has to be compiled with the FS_FAT_SUPPORT_UTF8 configuration define to 1. Additionally, the support for long file names has to be enabled for volumes formatted as FAT.

Example

```
#include "FS.h"

void SampleOpenFile(void) {
  FS_FILE * pFile;

  //
  // Opens for reading a file on the default volume.
  //
```



```
  pFile = FS_FOpen("Test.txt", "r");
  //
  // Opens for writing a file on the default volume.
  //
  pFile = FS_FOpen("Test.txt", "w");
  //
  // Opens for reading a file in folder "SubDir" on the default volume.
  // The directory delimiter is the '\' character.
  //
  pFile = FS_FOpen("\\SubDir\\Test.txt", "r");
  //
  // Opens for reading a file on the first RAM disk volume.
  //
  pFile = FS_FOpen("ram:Test.txt", "r");
  //
  // Opens for reading a file on the second RAM disk volume.
  //
  pFile = FS_FOpen("ram:1:Test.txt", "r");
  //
  // Opens for writing a file with a long name for writing.
  //
  FS_FAT_SupportLFN();
  pFile = FS_FOpen("Long file name.text", "w");
  //
  // Opens for writing a file with a name encoded in UTF-8 format.
  // The file system has to 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'
  //
  FS_FAT_SupportLFN();
  pFile = FS_FOpen("\xC3\xA4\xC3\xB6\xC3\x9F\xC3\xBC.txt", "w");
}
```


4.5.3 FS_FOpenEx()

Description

Opens an existing file or creates a new one.

Prototype

```
int FS_FOpenEx(const char     * sFileName,
               const char     * sMode,
                     FS_FILE ** ppFile);
```

Parameters

Parameter	Description
sFileName	Name of the file to be opened or created. It is a 0-terminated string that cannot be NULL.
sMode	Indicates how the file should be opened. It is a 0-terminated string that cannot be NULL.
ppFile	Pointer to a file handle pointer that receives the opened file handle.

Return value

= 0 OK, file opened.
≠ 0 Error code indicating the failure reason.

Additional information

For additional information about the sMode parameter refer to FS_FOpen().

Example

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

FS_FILE * pFile;

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

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


4.5.4 FS_FRead()

Description

Reads data from file.

Prototype

```
U32 FS_FRead(void    * pData,
             U32       ItemSize,
             U32       NumItems,
             FS_FILE * pFile);
```

Parameters

Parameter	Description
pData	Buffer that receives the data read from file.
ItemSize	Size of one item to be read from file (in bytes).
NumItems	Number of items to be read from the file.
pFile	Handle to opened file. It cannot be NULL.

Return value

Number of items read.

Additional information

The file has to be opened with read permissions. For more information about open modes refer to FS_FOpen().

The application has to check for possible errors using FS_FError() if the number of items actually read is different than the number of items requested to be read by the application.

The data is read from the current position in the file that is indicated by the file pointer. FS_FRead() moves the file pointer forward by the number of bytes successfully read.

Example

```
#include "FS.h"

char acBuffer[100];

void SampleFRead(void) {
  FS_FILE * pFile;
  U32       NumItems;
  int       ErrCode;

  pFile = FS_FOpen("Test.txt", "r");
  if (pFile) {
    NumItems = FS_FRead(acBuffer, 1, sizeof(acBuffer), pFile);
    if (NumItems != sizeof(acBuffer)) {
      ErrCode = FS_FError(pFile);
      if (ErrCode) {
        //
        // An error occurred during the read operation.
        //
      }
    }
    FS_FClose(pFile);
  }
}
```


4.5.5 FS_FSeek()

Description

Sets the current position in file.

Prototype

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

Parameters

Parameter	Description
pFile	Handle to opened file.
Offset	Byte offset for setting the file pointer position.
Origin	Indicates how the file pointer has to be moved.

Return value

= 0 OK, file pointer has been positioned according to the specified parameters.
≠ 0 Error code indicating the failure reason.

Additional information

FS_FSeek() moves the file pointer to a new location by a number of bytes relative to the position specified by the Origin parameter. The Origin parameter can take the values specified by the FS_SEEK_... defines

The file pointer can be repositioned anywhere in the file. It is also possible to reposition the file pointer beyond the end of the file. This feature is used together with FS_SetEndOfFile() to reserve space for a file (preallocate the file).

Alternative name

FS_SetFilePos

Example

```
#include "FS.h"

const char acText[]="Some text will be overwritten";

void SampleFTell(void) {
  FS_FILE * pFile;

  pFile = FS_FOpen("Test.txt", "w");
  if (pFile) {
    //
    // Write the data to file.
    //
    FS_FWrite(acText, 1, strlen(acText), pFile);
    //
    // Move the file pointer back 4 bytes from the current
    // position in file. The data stored in the file looks
    // like this: "Some text will be overwritten"
    //
    FS_FSeek(pFile, -4, FS_SEEK_CUR);
    //
    // The write operation overwrites the last
    // 4 characters that is "tten" of the data
    // written in the previous call to FS_FWrite().
    //
    FS_FWrite(acText, 1, strlen(acText), pFile);
    //
    // The data in the file looks now like this:
```



```
    // "Some text will be overwriSome text will be overwritten"
    //
    FS_FClose(pFile);
  }
}
```


4.5.6 FS_FWrite()

Description

Writes data to file.

Prototype

```
U32 FS_FWrite(const void    * pData,
                    U32       ItemSize,
                    U32       NumItems,
                    FS_FILE * pFile);
```

Parameters

Parameter	Description
pData	Data to be written to file.
ItemSize	Size of an item to be written to file (in bytes).
NumItems	Number of items to be written to file.
pFile	Handle to opened file. It cannot be NULL.

Return value

Number of elements written.

Additional information

The file has to be opened with write permissions. For more information about open modes refer to FS_FOpen().

The application has to check for possible errors using FS_FError() if the number of items actually written is different than the number of items requested to be written by the application.

The data is written at the current position in the file that is indicated by the file pointer. FS_FWrite() moves the file pointer forward by the number of bytes successfully written.

Example

```
#include "FS.h"

const char acText[]="Hello world\n";

void SampleFWrite(void) {
  FS_FILE * pFile;

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


4.5.7 FS_FTell()

Description

Returns current position in file.

Prototype

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

Parameters

Parameter	Description
pFile	Handle to opened file.

Return value

≠ 0xFFFFFFFF	OK, current position of the file pointer.
= 0xFFFFFFFF	An error occurred.

Additional information

The function returns the file position as a signed value for compatibility reasons. The return value has to be treated as a 32-bit unsigned with the value 0xFFFFFFFF indicating an error.

This function simply returns the file pointer member of the file handle structure pointed by pFile. 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(). Alternatively the FS_GetFileSize() function can be used.

Alternative name

FS_GetFilePos

Example

```
#include "FS.h"

const char acText[]="Hello world\n";

void SampleFTell(void) {
  FS_FILE * pFile;
  I32       FilePos;

  pFile = FS_FOpen("Test.txt", "w");
  if (pFile) {
    FS_FWrite(acText, 1, strlen(acText), pFile);
    FilePos = FS_FTell(pFile);
    FS_FClose(pFile);
  }
}
```


4.5.8 FS_GetFileSize()

Description

Returns the size of a file.

Prototype

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

Parameters

Parameter	Description
pFile	Handle to opened file.

Return value

≠ 0xFFFFFFFF	File size of the given file in bytes.
= 0xFFFFFFFF	An error occurred.

Additional information

The file has to be opened with read or write access.

Example

```
#include "FS.h"

void SampleGetFileSize(void) {
  FS_FILE * pFile;
  U32       FileSize;

  pFile = FS_FOpen("Test.txt", "r");
  if (pFile) {
    FileSize = FS_GetFileSize(pFile);
    FS_FClose(pFile);
  }
}
```


4.5.9 FS_Read()

Description

Reads data from a file.

Prototype

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

Parameters

Parameter	Description
pFile	Handle to an opened file. It cannot be NULL.
pData	Buffer to receive the read data.
NumBytes	Number of bytes to be read.

Return value

Number of bytes read.

Additional information

The file has to be opened with read permissions. For more information about open modes refer to FS_FOpen().

The application has to check for possible errors using FS_FError() if the number of bytes actually read is different than the number of bytes requested to be read by the application.

The data is read from the current position in the file that is indicated by the file pointer. FS_Read() moves the file pointer forward by the number of bytes successfully read.

Example

```
#include "FS.h"

U8 abBuffer[100];

void SampleRead(void) {
  FS_FILE * pFile;
  U32       NumBytesRead;

  pFile = FS_FOpen("Test.txt", "r");
  if (pFile) {
    NumBytesRead = FS_Read(pFile, abBuffer, sizeof(abBuffer));
    if (NumBytesRead != sizeof(abBuffer)) {
      //
      // An error occurred.
      //
    }
    FS_FClose(pFile);
  }
}
```


4.5.10 FS_SetEndOfFile()

Description

Sets the file size to current file position.

Prototype

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

Parameters

Parameter	Description
pFile	Handle to opened file.

Return value

= 0 OK, new file size has been set.
≠ 0 Error code indicating the failure reason.

Additional information

The file has to be opened with write permissions. Refer to FS_FOpen() for more information about the file open modes.

FS_SetEndOfFile() can be used to truncate as well as to extend a file. If the file is extended, the contents of the file between the old end-of-file and the new one are not defined. Extending a file (preallocation) can increase the write performance when the application writes large amounts of data to file as the file system is not required anymore to access the allocation table.

Example

```
#include "FS.h"

void SampleSetEndOfFile(void) {
  FS_FILE * pFile;

  //
  // Reserve space for a file of 20000 bytes.
  //
  pFile = FS_FOpen("Test.bin", "r+");
  if (pFile) {
    FS_FSeek(pFile, 20000, FS_SEEK_SET);
    FS_SetEndOfFile(pFile);
    FS_FClose(pFile);
  }
}
```


4.5.11 FS_SetFileSize()

Description

Sets the file size to the specified number of bytes.

Prototype

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

Parameters

Parameter	Description
pFile	Handle to an opened file.
NumBytes	New file size.

Return value

= 0 OK, new file size has been set.
≠ 0 Error code indicating the failure reason.

Additional information

The file has to be opened with write permissions. Refer to FS_FOpen() for more information about the file open modes. FS_SetFileSize() can be used to extend as well as truncate a file. The file position is preserved if the new file size is larger than or equal to the current file position. Else the file position is set to the end of the file.

Example

```
#include "FS.h"

void SampleSetFileSize(void) {
  FS_FILE * pFile;

  //
  // Create a file 20000 bytes large.
  //
  pFile = FS_FOpen("Test.bin", "r+");
  if (pFile) {
    FS_SetFileSize(pFile, 20000);
    FS_FClose(pFile);
  }
}
```


4.5.12 FS_SyncFile()

Description

Synchronizes file to storage device.

Prototype

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

Parameters

Parameter	Description
pFile	File pointer identifying the opened file. It can be NULL.

Return value

= 0 OK, file(s) synchronized.
≠ 0 Error code indicating the failure reason.

Additional information

The function synchronizes all the opened files if a NULL is passed as pFile parameter.

FS_SyncFile() cleans the write buffer if configured and updates the management information to storage device. The function performs basically the same operations as FS_FClose() with the only difference that it leaves the file open. FS_SyncFile() is used typically with fast or medium file write modes to make sure that the data cached by the file system for the file is written to storage medium.

The function can also be called from a different task than the one that writes to file if the support for multi-tasking is enabled in the file system. The support for multi-tasking can be enabled by compiling the file system sources with the FS_OS_LOCKING define is set to value different than 0.

Example

```
#include "FS.h"

void SampleFileSync(void) {
  FS_FILE * pFile;

  FS_SetFileWriteMode(FS_WRITEMODE_FAST);
  pFile = FS_FOpen("Test.txt", "w");
  if (pFile) {
    //
    // Write to file...
    //
    FS_SyncFile(pFile);
    //
    // Write to file...
    //
    FS_SyncFile(pFile);
    //
    // Write to file...
    //
    FS_FClose(pFile);
  }
}
```


4.5.13 FS_Truncate()

Description

Changes the size of a file.

Prototype

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

Parameters

Parameter	Description
pFile	Pointer to a valid opened file with write access.
NewFileSize	The new size of the file.

Return value

= 0 OK, file size has been truncated.
≠ 0 Error code indicating the failure reason.

Additional information

The file has to be opened with write permissions. For more information about open modes refer to FS_FOpen(). An error is returned if NewSize is larger than the actual file size.

Example

```
#include "FS.h"

void SampleTruncate(void) {
  FS_FILE * pFile;
  U32       FileSize;
  int       r;

  pFile = FS_FOpen("Test.bin", "r+");
  if (pFile) {
    FileSize = FS_GetFileSize(pFile);
    //
    // Reduce the size of the file by 200 bytes.
    //
    r = FS_Truncate(pFile, FileSize - 200);
    if (r) {
      //
      // An error occurred while truncating file.
      //
    }
  }
  FS_FClose(pFile);
}
```


4.5.14 FS_Verify()

Description

Verifies the file contents.

Prototype

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

Parameters

Parameter	Description
pFile	Handle to opened file.
pData	Data to be checked against.
NumBytes	Number of bytes to be checked.

Return value

= 0 Verification was successful.
≠ 0 Verification failed.

Additional information

The function starts checking at the current file position. That is the byte read from file position + 0 is checked against the byte at pData + 0, the byte read from file position + 1 is checked against the byte at pData + 1 and so on. FS_Verify() does not modify the file position.

Example

```
#include "FS.h"

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

void SampleVerify(void) {
  FS_FILE * pFile;
  int       r;

  //
  // Open file and write data into it.
  //
  pFile = FS_FOpen("Test.bin", "w+");
  if (pFile) {
    FS_Write(pFile, abVerifyData, sizeof(abVerifyData));
    //
    // Set file pointer to the start of the data that has to be verified.
    //
    FS_FSeek(pFile, 0, FS_SEEK_SET);
    //
    // Verify data.
    //
    r = FS_Verify(pFile, abVerifyData, sizeof(abVerifyData));
    if (r == 0) {
      FS_X_Log("Verification was successful.\n");
    } else {
      FS_X_Log("Verification failed.\n");
    }
    FS_FClose(pFile);
  }
}
```


4.5.15 FS_Write()

Description

Writes data to file.

Prototype

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

Parameters

Parameter	Description
pFile	Handle to opened file.
pData	The data to be written to file.
NumBytes	Number of bytes to be written to file.

Return value

Number of bytes written.

Additional information

The file has to be opened with write permissions. For more information about open modes refer to FS_FOpen().

The application has to check for possible errors using FS_FError() if the number of bytes actually written is different than the number of bytes requested to be written by the application.

The data is written at the current position in the file that is indicated by the file pointer. FS_FWrite() moves the file pointer forward by the number of bytes successfully written.

Example

```
#include "FS.h"

void SampleWrite(void) {
  FS_FILE    * pFile;
  const char   acText[] = "Hello world\n";

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


4.5.16 File positioning reference

Description

Reference point when changing the file position.

Definition

```
#define FS_SEEK_SET    0
#define FS_SEEK_CUR    1
#define FS_SEEK_END    2
```

Symbols

Definition	Description
FS_SEEK_SET	The reference is the beginning of the file.
FS_SEEK_CUR	The reference is the current position of the file pointer.
FS_SEEK_END	The reference is the end-of-file position.

4.6 Operations on files

The functions on this section can be used by an application to manipulate files.

4.6.1 FS_CopyFile()

Description

Copies a file.

Prototype

```
int FS_CopyFile(const char * sFileNameSrc,
                const char * sFileNameDest);
```

Parameters

Parameter	Description
sFileNameSrc	Name of the source file (fully qualified).
sFileNameDest	Name of the destination file (fully qualified).

Return value

= 0 OK, the file has been copied
≠ 0 Error code indicating the failure reason

Additional information

The copy process uses an internal temporary buffer of 512 bytes. FS_CopyFile() overwrites the destination file if it exists. The destination file has to be writable, that is the FS_ATTR_READ_ONLY flag is set to 0.

Example

```
#include "FS.h"

void SampleCopyFile(void) {
  FS_CopyFile("Src.txt", "ram:\\Dest.txt");
}
```


4.6.2 FS_CopyFileEx()

Description

Copies a file.

Prototype

```
int FS_CopyFileEx(const char * sFileNameSrc,
                  const char * sFileNameDest,
                        void * pBuffer,
                        U32    NumBytes);
```

Parameters

Parameter	Description
sFileNameSrc	Name of the source file (fully qualified).
sFileNameDest	Name of the destination file (fully qualified).
pBuffer	Buffer to temporary store the copied data. It cannot be NULL.
NumBytes	Size of the buffer size in bytes.

Return value

= 0 OK, the file has been copied.
≠ 0 Error code indicating the failure reason.

Additional information

The copy process uses an external buffer provided by the application. FS_CopyFile() overwrites the destination file if it exists. The destination file has to be writable, that is the FS_ATTR_READ_ONLY flag is set to 0.

The best performance is achieved when the copy buffer is a multiple of sector size. For example using a 7 Kbyte copy buffer to copy 512 byte sectors is more efficient than using a copy buffer of 7.2 Kbyte therefore the function rounds down the size of the copy buffer to a multiple of sector size. If the application specifies a copy buffer smaller than the sector size a warning is generated in debug builds indicating that the performance of the copy operation is not optimal.

Example

```
#include "FS.h"

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

void SampleCopyFileEx(void) {
  FS_CopyFileEx("Src.txt", "Dest.txt", _aBuffer, sizeof(_aBuffer));
}
```


4.6.3 FS_GetFileAttributes()

Description

Queries the attributes of a file or directory.

Prototype

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

Parameters

Parameter	Description
sName	Name of the file or directory to be queried.

Return value

= 0xFF An error occurred.
≠ 0xFF Bit mask containing the attributes of file or directory.

Additional information

The return value is an or-combination of the following attributes: FS_ATTR_READ_ONLY, FS_ATTR_HIDDEN, FS_ATTR_SYSTEM, FS_ATTR_ARCHIVE, or FS_ATTR_DIRECTORY.

Example

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

void SampleGetFileAttributes(void) {
  U8   Attr;
  char ac[100];

  Attr = FS_GetFileAttributes("Test.txt");
  if (Attr == 0xFF) {
    FS_X_Log("Could not get the file attributes.\n");
  } else {
    sprintf(ac,
            "File attributes: %c%c%c%c%c\n",
            (Attr & FS_ATTR_READ_ONLY) ? 'R' : '-',
            (Attr & FS_ATTR_HIDDEN)    ? 'H' : '-',
            (Attr & FS_ATTR_SYSTEM)    ? 'S' : '-',
            (Attr & FS_ATTR_ARCHIVE)   ? 'A' : '-',
            (Attr & FS_ATTR_DIRECTORY) ? 'D' : '-');
    FS_X_Log(ac);
  }
}
```


4.6.4 FS_GetFileInfo()

Description

Returns information about a file or directory.

Prototype

```
int FS_GetFileInfo(const char         * sName,
                   FS_FILE_INFO * pInfo);
```

Parameters

Parameter	Description
sName	Name of the file or directory.
pInfo	out Information about file or directory.

Return value

= 0 OK, information returned.
≠ 0 An error occurred.

Additional information

The function returns information about the attributes, size and time stamps of the specified file or directory. For more information refer to FS_FILE_INFO.

Example

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

void SampleGetFileInfo(void) {
  int          r;
  FS_FILE_INFO FileInfo;
  char         ac[100];
  FS_FILETIME  FileTime;

  r = FS_GetFileInfo("Test.txt", &FileInfo);
  if (r == 0) {
    FS_X_Log("File info:");
    sprintf(ac, "  Attributes:    0x%x", FileInfo.Attributes);
    FS_X_Log(ac);
    sprintf(ac, "  Size:          %d bytes", FileInfo.FileSize);
    FS_X_Log(ac);
    FS_TimeStampToFileTime(FileInfo.CreationTime, &FileTime);
    sprintf(ac, "  Creation time: %d-%.2d-%.2d %.2d:%.2d:%.2d",
            FileTime.Year, FileTime.Month,  FileTime.Day,
            FileTime.Hour, FileTime.Minute, FileTime.Second);
    FS_X_Log(ac);
    FS_TimeStampToFileTime(FileInfo.LastAccessTime, &FileTime);
    sprintf(ac, "  Access time:   %d-%.2d-%.2d %.2d:%.2d:%.2d",
            FileTime.Year, FileTime.Month,  FileTime.Day,
            FileTime.Hour, FileTime.Minute, FileTime.Second);
    FS_X_Log(ac);
    FS_TimeStampToFileTime(FileInfo.LastWriteTime, &FileTime);
    sprintf(ac, "  Write time:    %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_GetFileTime()

Description

Returns the creation time of a file or directory.

Prototype

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

Parameters

Parameter	Description
sName	File or directory name.
pTimeStamp	out Receives the timestamp value.

Return value

= 0 OK, timestamp returned.
≠ 0 Error code indicating the failure reason.

Additional information

The date and time encoded in the timestamp using the following format:

Bit field	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 (1--12, 1: January, 2: February, etc.)
25--31	Year (offset from 1980). Add 1980 to get the current year.

FS_TimeStampToFileTime() can be used to convert the timestamp to a FS_FILETIME structure that can be used to easily process the time information.

The last modification and the last access timestamps can be read via FS_GetFileTimeEx().

Example

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

void SampleGetFileTime(void) {
  char        ac[100];
  U32         TimeStamp;
  FS_FILETIME FileTime;
  int         r;

  r = FS_GetFileTime("Test.txt", &TimeStamp);
  if (r == 0) {
    FS_TimeStampToFileTime(TimeStamp, &FileTime);
    sprintf(ac, "Creation time: %d-%.2d-%.2d %.2d:%.2d:%.2d",
            FileTime.Year, FileTime.Month,  FileTime.Day,
            FileTime.Hour, FileTime.Minute, FileTime.Second);
    FS_X_Log(ac);
  }
}
```


4.6.6 FS_GetFileTimeEx()

Description

Gets the timestamp of a file or directory.

Prototype

```
int FS_GetFileTimeEx(const char * sName,
                           U32  * pTimeStamp,
                           int    TimeType);
```

Parameters

Parameter	Description
sName	File or directory name.
pTimeStamp	out Receives the timestamp value.
TimeType	Type of timestamp to return. It can take one of the following values: • FS_FILETIME_CREATE • FS_FILETIME_ACCESS • FS_FILETIME_MODIFY

Return value

= 0 OK, timestamp returned.
≠ 0 Error code indicating the failure reason.

Additional information

Refer to FS_GetFileTime() for a description of the timestamp format. FS_TimeStampToFileTime() can be used to convert the timestamp to a FS_FILETIME structure that can be used to easily process the time information.

EFS maintains only one filestamp that is updated when the file is created and updated therefore the same timestamp value is returned for all time types.

Example

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

void SampleGetFileTimeEx(void) {
  char        ac[100];
  U32         TimeStamp;
  FS_FILETIME FileTime;
  int         r;

  r = FS_GetFileTimeEx("Test.txt", &TimeStamp, FS_FILETIME_MODIFY);
  if (r == 0) {
    FS_TimeStampToFileTime(TimeStamp, &FileTime);
    sprintf(ac, "Modify time: %d-%.2d-%.2d %.2d:%.2d:%.2d",
            FileTime.Year, FileTime.Month,  FileTime.Day,
            FileTime.Hour, FileTime.Minute, FileTime.Second);
    FS_X_Log(ac);
  }
}
```


4.6.7 FS_ModifyFileAttributes()

Description

Sets / clears the attributes of a file or directory.

Prototype

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

Parameters

Parameter	Description
sName	Name of the file or directory.
SetMask	Bit mask of the attributes to be set.
ClrMask	Bit mask of the attributes to be cleared.

Return value

= 0xFF An error occurred.
≠ 0xFF Bit mask containing the old attributes of the file or directory.

Additional information

This function can be used to set and clear at the same time the attributes of a file or directory. The FS_ATTR_DIRECTORY attribute cannot be modified using this function.

The return value is an or-combination of the following attributes: FS_ATTR_READ_ONLY, FS_ATTR_HIDDEN, FS_ATTR_SYSTEM, FS_ATTR_ARCHIVE, or FS_ATTR_DIRECTORY.

The attributes that are specified in the SetMask are set to 1 while the attributes that are specified in the ClrMask are set to 0. SetMask and ClrMask values are an or-combination of the following attributes: FS_ATTR_READ_ONLY, FS_ATTR_HIDDEN, FS_ATTR_SYSTEM, or FS_ATTR_ARCHIVE. The attributes that are not specified nor in SetMask either in ClrMask are not modified.

Example

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

void SampleModifyFileAttributes(void) {
  U8   Attr;
  char ac[100];

  //
  // Set the read-only and the hidden flags. Clear archive flag.
  //
  Attr = FS_ModifyFileAttributes("Test.txt",
                                 FS_ATTR_READ_ONLY | FS_ATTR_HIDDEN,
                                 FS_ATTR_ARCHIVE);
  sprintf(ac,
          "Old file attributes: %c%c%c%c%c\n",
          (Attr & FS_ATTR_READ_ONLY) ? 'R' : '-',
          (Attr & FS_ATTR_HIDDEN)    ? 'H' : '-',
          (Attr & FS_ATTR_SYSTEM)    ? 'S' : '-',
          (Attr & FS_ATTR_ARCHIVE)   ? 'A' : '-',
          (Attr & FS_ATTR_DIRECTORY) ? 'D' : '-');
  FS_X_Log(ac);
  Attr = FS_GetFileAttributes("Test.txt");
  sprintf(ac,
          "New file attributes: %c%c%c%c%c\n",
          (Attr & FS_ATTR_READ_ONLY) ? 'R' : '-',
          (Attr & FS_ATTR_HIDDEN)    ? 'H' : '-',
          (Attr & FS_ATTR_SYSTEM)    ? 'S' : '-',
```



```
          (Attr & FS_ATTR_ARCHIVE)   ? 'A' : '-',
          (Attr & FS_ATTR_DIRECTORY) ? 'D' : '-');
  FS_X_Log(ac);
}
```


4.6.8 FS_Move()

Description

Moves a file or directory to another location.

Prototype

```
int FS_Move(const char * sNameSrc,
            const char * sNameDest);
```

Parameters

Parameter	Description
sNameSrc	Name of the source file or directory (partially qualified). Cannot be NULL.
sNameDest	Name of the destination file or directory (partially qualified). Cannot be NULL.

Return value

= 0 OK, file has been moved.
≠ 0 Error code indicating the failure reason.

Additional information

FS_Move() can be used to relocate a file to another location on the same or on a different volume either in the same in a different directory. If the source and the destination files are located on the same volume, the file is moved, otherwise the file is copied and the source is deleted.

The function is also able to move and entire directory trees when the source and destination are located on the same volume. Moving an entire directory tree on a different volume is not supported. This operation has to be performed in the application by iterating over the files and directories and by copying them one-by-one.

By default, the source files and directories that have the FS_ATTR_READ_ONLY attribute set and that are located on a volume formatted as FAT cannot be moved. This behavior can be changed by compiling the file system sources with FS_FAT_PERMIT_RO_FILE_MOVE configuration define to 1. FS_FAT_ConfigROFileMovePermission() can be used to change the behavior at runtime.

Source files and directories located on a EFS formatted volume can be moved even if they have the FS_ATTR_READ_ONLY attribute set.

The operation fails if the destination file or directory already exists.

Example

```
#include "FS.h"

void SampleMove(void) {
  //
  // File tree before the operation.
  //
  // <root>
  //   |
  //   +->SubDir2
  //        |
  //        +->SubDir3
  //
  FS_Move("SubDir1", "SubDir2\\SubDir3");
  //
  // File tree after the operation.
  //
  // <root>
  //   |
```



```
  //   +->SubDir1
  //   |
  //   +->SubDir2
  //
}
```


4.6.9 FS_Remove()

Description

Removes a file.

Prototype

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

Parameters

Parameter	Description
sFileName	Name of the file to be removed.

Return value

= 0 OK, file has been removed.
≠ 0 Error code indicating the failure reason.

Additional information

The function removes also files that have the FS_ATTR_READ_ONLY attribute set. The remove operation fails if the file to be deleted is open.

Example

```
#include "FS.h"

void SampleRemove(void) {
  FS_Remove("Test.txt");
}
```


4.6.10 FS_Rename()

Description

Changes the name of a file or directory.

Prototype

```
int FS_Rename(const char * sNameOld,
              const char * sNameNew);
```

Parameters

Parameter	Description
sNameOld	Old file or directory name (including the path).
sNameNew	New file or directory name (without path).

Return value

= 0 OK, file or directory has been renamed.
≠ 0 Error code indicating the failure reason.

Additional information

The function can rename either a file or a directory.

By default, the files and directories that have the FS_ATTR_READ_ONLY attribute set and that are located on a volume formatted as FAT cannot be renamed. This behavior can be changed by compiling the file system sources with the FS_FAT_PERMIT_RO_FILE_MOVE configuration define to 1. FS_FAT_ConfigROFileMovePermission() can be used to change the behavior at runtime.

Source files and directories located on a EFS formatted volume can be moved event if they have the FS_ATTR_READ_ONLY attribute set.

Example

```
#include "FS.h"

void SampleRename(void) {
  //
  // File tree before the operation.
  //
  // <root>
  //   |
  //   +->SubDir2
  //        |
  //        +->SubDir3
  //
  FS_Rename("SubDir2\\SubDir3", "SubDir1");
  //
  // File tree after the operation.
  //
  // <root>
  //   |
  //   +->SubDir1
  //        |
  //        +->SubDir3
  //
}
```


4.6.11 FS_SetFileAttributes()

Description

Modifies all the attributes of a file or directory.

Prototype

```
int FS_SetFileAttributes(const char * sName,
                         U8     AttrMask);
```

Parameters

Parameter	Description
sName	Pointer to a name of the file/directory.
AttrMask	Bit mask of the attributes to be set.

Return value

= 0 OK, attributes have been set.
≠ 0 Error code indicating the failure reason.

Additional information

The FS_ATTR_DIRECTORY attribute cannot be modified using this function. The value of AttrMask parameter is an or-combination of the following attributes: FS_ATTR_READ_ONLY, FS_ATTR_HIDDEN, FS_ATTR_SYSTEM, FS_ATTR_ARCHIVE, or FS_ATTR_DIRECTORY. The attributes that are not set in AttrMask are set to 0.

Example

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

void SampleFileAttributes(void) {
  U8   Attr;
  char ac[100];

  FS_SetFileAttributes("Test.txt", FS_ATTR_HIDDEN);
  Attr = FS_GetFileAttributes("Test.txt");
  sprintf(ac,
          "File attributes: %c%c%c%c%c\n",
          (Attr & FS_ATTR_READ_ONLY) ? 'R' : '-',
          (Attr & FS_ATTR_HIDDEN)    ? 'H' : '-',
          (Attr & FS_ATTR_SYSTEM)    ? 'S' : '-',
          (Attr & FS_ATTR_ARCHIVE)   ? 'A' : '-',
          (Attr & FS_ATTR_DIRECTORY) ? 'D' : '-');
  FS_X_Log(ac);
}
```


4.6.12 FS_SetFileBuffer()

Description

Assigns a file buffer to an opened file.

Prototype

```
int FS_SetFileBuffer(FS_FILE * pFile,
                     void    * pData,
                     I32       NumBytes,
                     int       Flags);
```

Parameters

Parameter	Description
pFile	Handle to opened file.
pData	Pointer to the to memory area which should be used as buffer.
NumBytes	Number of bytes in the buffer.
Flags	Specifies the operating mode of the file buffer. • 0 Read file buffer. • FS_FILE_BUFFER_WRITE Read / write file buffer. • FS_FILE_BUFFER_ALIGNED Logical sector boundary alignment.

Return value

= 0 OK, buffer assigned.
≠ 0 Error code indicating the failure reason.

Additional information

This function has to be called immediately after the file is opened and before any read or write operation is performed on the file. If the file buffer is configured in write mode the data of any operation that writes less bytes at once than the size of the file buffer is stored to file buffer. The contents of the file buffer is written to file in the following cases:

- when the file buffer is full.
- when place is required for new data read from file.
- when closing the file via FS_FClose().
- when synchronizing the file to storage via FS_SyncFile().
- when unmounting the file system via FS_Unmount() or FS_UnmountForced().
- when the file system is synchronized via FS_Sync().

In case of a read operation if the data is not present in the file buffer the file system fills the entire file buffer with the data from file.

FS_SetFileBuffer reports an error if the file system is configured to automatically allocate a file buffer for each file it opens via FS_ConfigFileBufferDefault().

The data required to manage the file buffer is allocated from pData. The FS_SIZEOF_FILE_BUFFER() define can be used to calculate the amount of RAM required to store a specified number of data bytes in the file buffer.

If the file is opened and closed in the same function the file buffer can be allocated locally on the stack. Otherwise the buffer has to be globally allocated. After the file is closed the memory allocated for the file buffer is no longer accessed by the file system and can be safely deallocated or used to store other data.

FS_SetFileBuffer() is available if the emFile sources are compiled with the FS_SUPPORT_FILE_BUFFER configuration define set to 1.

Example

```
#include "FS.h"

void SampleSetFileBuffer(void) {
  FS_FILE * pFile;
  U32       aBuffer[FS_SIZEOF_FILE_BUFFER(512) / 4];

  pFile = FS_FOpen("Test.txt", "w");
  if (pFile != NULL) {
    //
    // Set the file buffer for read and write operation.
    // The file buffer is allocated on the stack since the file
    // is closed before the function returns.
    //
    FS_SetFileBuffer(pFile, aBuffer, sizeof(aBuffer), 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);
  }
}
```


4.6.13 FS_SetFileTime()

Description

Sets the creation time of a file or directory.

Prototype

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

Parameters

Parameter	Description
sName	File or directory name.
TimeStamp	The value of the timestamp to be set.

Return value

= 0 OK, timestamp modified.
≠ 0 Error code indicating the failure reason.

Additional information

Refer to FS_GetFileTime() for a description of the timestamp format. FS_FileTimeToTimeStamp() can be used to convert a FS_FILETIME structure to a timestamp that can be used in a call to FS_SetFileTime().

This function is optional. The file system updates automatically the timestamps of file or directories.

Example

```
#include "FS.h"

void SampleSetFileTime(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);
}
```


4.6.14 FS_SetFileTimeEx()

Description

Sets the timestamp of a file or directory.

Prototype

```
int FS_SetFileTimeEx(const char * sName,
                     U32    TimeStamp,
                     int    TimeType);
```

Parameters

Parameter	Description
sName	File or directory name.
TimeStamp	The value of the timestamp to be set.
TimeType	Type of timestamp to be modified. It can take of the following values: • FS_FILETIME_CREATE • FS_FILETIME_ACCESS • FS_FILETIME_MODIFY

Return value

= 0 OK, timestamp modified.
≠ 0 Error code indicating the failure reason.

Additional information

Refer to FS_GetFileTime() for a description of the timestamp format. FS_FileTimeToTimeStamp() can be used to convert a FS_FILETIME structure to a timestamp that can be used in a call to FS_SetFileTimeEx().

EFS maintains only one filestamp therefore the TimeType parameter is ignored for files and directories stored on an EFS volume.

This function is optional. The file system updates automatically the timestamps of file or directories.

Example

```
#include "FS.h"

void SampleSetFileTime(void) {
  U32         TimeStamp;
  FS_FILETIME FileTime;

  FileTime.Year   = 2017;
  FileTime.Month  = 07;
  FileTime.Day    = 07;
  FileTime.Hour   = 14;
  FileTime.Minute = 48;
  FileTime.Second = 14;
  FS_FileTimeToTimeStamp(&FileTime, &TimeStamp);
  //
  // Set the access time of the file.
  //
  FS_SetFileTimeEx("Test.txt", TimeStamp, FS_FILETIME_ACCESS);
}
```


4.6.15 FS_WipeFile()

Description

Overwrites the contents of a file with random data.

Prototype

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

Parameters

Parameter	Description
sFileName	Name of the file to overwrite.

Return value

= 0 Contents of the file overwritten.
≠ 0 Error code indicating the failure reason.

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 it can be in theory restored by using a disk recovery tool. This can be a problem if the file stores sensitive data. Calling `FS_WipeFile()` before the file is removed makes the recovery of data impossible.

Example

```
#include "FS.h"

void SampleWipeFile(void) {
  FS_FILE * pFile;

  //
  // Create a file and write data to it.
  //
  pFile = FS_FOpen("Test.txt", "w");
  if (pFile) {
    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");
}
```


4.6.16 File attributes

Description

Attributes of files and directories.

Definition

```
#define FS_ATTR_READ_ONLY    0x01u
#define FS_ATTR_HIDDEN       0x02u
#define FS_ATTR_SYSTEM       0x04u
#define FS_ATTR_ARCHIVE      0x20u
#define FS_ATTR_DIRECTORY    0x10u
```

Symbols

Definition	Description
FS_ATTR_READ_ONLY	The file is read-only. Applications can read the file but cannot write to it.
FS_ATTR_HIDDEN	The file or directory is marked as hidden. Most of operating systems do not include these files in an ordinary directory listing. This flag is not evaluated by emFile.
FS_ATTR_SYSTEM	The file or directory is part of, or is used exclusively by, the operating system. This flag is not evaluated by emFile.
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. This flag is not evaluated by emFile.
FS_ATTR_DIRECTORY	The file is actually a directory.

4.6.17 FS_FILE_INFO

Description

Information about a file or directory.

Type definition

```
typedef struct {
  U8   Attributes;
  U32  CreationTime;
  U32  LastAccessTime;
  U32  LastWriteTime;
  U32  FileSize;
} FS_FILE_INFO;
```

Structure members

Member	Description
Attributes	File or directory attributes.
CreationTime	Date and time when the file was created.
LastAccessTime	Date and time when the file was accessed last.
LastWriteTime	Date and time when the file was written to last.
FileSize	Size of the file in bytes.

Additional information

The Attributes member is an or-combination of the following values: FS_ATTR_READ_ONLY, FS_ATTR_HIDDEN, FS_ATTR_SYSTEM, FS_ATTR_ARCHIVE, or FS_ATTR_DIRECTORY.

For directories the FileSize member is always 0.

4.6.18 File time types

Description

Types of timestamps available for a file or directory.

Definition

```
#define FS_FILETIME_CREATE    0
#define FS_FILETIME_ACCESS    1
#define FS_FILETIME_MODIFY    2
```

Symbols

Definition	Description
FS_FILETIME_CREATE	Date and time when the file or directory was created.
FS_FILETIME_ACCESS	Date and time of the last read access to file or directory.
FS_FILETIME_MODIFY	Date and time of the last write access to file or directory.

4.7 Directory functions

This section describes functions that operate on directories. API functions are provided for creating, deleting and listing directories.

4.7.1 FS_CreateDir()

Description

Creates a directory including any missing directories from path.

Prototype

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

Parameters

Parameter	Description
sDirName	Directory name.

Return value

= 0 Directory path has been created.
= 1 Directory path already exists.
< 0 Error code indicating the failure reason.

Additional information

The function creates automatically any subdirectories that are specified in the path but do not exist on the storage.

FS_CreateDir() uses a work buffer of FS_MAX_PATH bytes allocated on the stack for parsing the path to directory.

Example

```
#include "FS.h"

void SampleCreateDir(void) {
  //
  // File tree before the operation.
  //
  // <root>
  //   |
  //   +->SubDir1
  //
  FS_CreateDir("SubDir1\\SubDir2\\SubDir3");
  //
  // File tree after the operation.
  //
  // <root>
  //   |
  //   +->SubDir1
  //        |
  //        +->SubDir2
  //             |
  //             +->SubDir3
  //
}
```


4.7.2 FS_DeleteDir()

Description

Removes a directory and its contents.

Prototype

```
int FS_DeleteDir(const char * sDirName,
                 int    MaxRecursionLevel);
```

Parameters

Parameter	Description
sDirName	Directory name.
MaxRecursionLevel	Maximum depth of the directory tree.

Return value

= 0 OK, directory has been removed
≠ 0 Error code indicating the failure reason

Additional information

The function uses recursion to process the directory tree and it requires about 100 bytes of stack for each directory level it processes. The MaxRecursionLevel parameter can be used to prevent a stack overflow if the directory tree is too deep. For example:

- MaxRecursionLevel=0 Deletes the directory only if empty else an error is reported and the directory is not deleted.
- MaxRecursionLevel=1 Deletes the directory and all the files in it. If a subdirectory is present an error is reported and the directory is not deleted.
- MaxRecursionLevel=2 Deletes the directory and all the files and subdirectories in it. If the subdirectories contain other subdirectories an error is reported and the directory is not deleted.
- And so on…

Example

```
#include "FS.h"

void SampleDeleteDir(void) {
  //
  // File tree before the operation.
  //
  // <root>
  //   |
  //   +->SubDir1
  //   |
  //   +->SubDir2
  //        |
  //        +->File1.txt
  //        |
  //        +->File2.txt
  //
  FS_DeleteDir("SubDir2", 1);
  //
  // File tree after the operation.
  //
  // <root>
  //   |
  //   +->SubDir1
  //
}
```


4.7.3 FS_FindClose()

Description

Ends a directory scanning operation.

Prototype

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

Parameters

Parameter	Description
pFD	Context of the directory scanning operation.

Additional information

This function has to be called at the end of the directory scanning operation to clear the used context. After calling this function the same context can be used for a different directory scanning operation.

Example

```
#include "FS.h"

void SampleFindClose(void) {
  FS_FIND_DATA fd;
  int          r;
  char         acFileName[32];

  //
  // List the files from the root directory.
  //
  r = FS_FindFirstFile(&fd, "", acFileName, sizeof(acFileName));
  if (r == 0) {
    do {
      //
      // Process file or directory...
      //
    } while (FS_FindNextFile(&fd));
  }
  FS_FindClose(&fd);
  //
  // List the files from the "SubDir1" directory.
  //
  r = FS_FindFirstFile(&fd, "SubDir1", acFileName, sizeof(acFileName));
  if (r == 0) {
    do {
      //
      // Process file or directory...
      //
    } while (FS_FindNextFile(&fd));
  }
  FS_FindClose(&fd);
}
```


4.7.4 FS_FindFirstFile()

Description

Initiates a directory scanning operation and returns information about the first file or directory.

Prototype

```
int FS_FindFirstFile(      FS_FIND_DATA * pFD,
                     const char         * sDirName,
                           char         * sFileName,
                           int            SizeOfFileName);
```

Parameters

Parameter	Description
pFD	Context of the directory scanning operation.
sDirName	Name of the directory to search in.
sFileName	Buffer that receives the name of the file found.
SizeOfFileName	Size in bytes of the sFileName buffer.

Return value

= 0 O.K.
= 1 No files or directories available in directory.
< 0 Error code indicating the failure reason.

Additional information

This function returns information about the first file or directory found in the directory. This is typically the special directory name ".." when searching in a directory other than root. FS_FindFirstFile() and FS_FindNextFile() can be used to list all the files and directories in a directory.

The name of the file or directory is returned in sFileName. FS_FindFirstFile() stores at most SizeOfFileName - 1 characters in the buffer and it terminates the name of file or directory by a 0 character. The file or directory name is truncated if it contains more characters than the number of characters that can be stored in sFileName. For files or directories stored on a FAT volume with LFN support the name is truncated by returning the short version (8.3 format) of the file or directory name.

Alternatively, the file name can be accessed via the sFileName member of the pFD structure. Additional information about the file or directory such as size, attributes and timestamps are returned via the corresponding members of pFD structure. For more information refer to FS_FIND_DATA.

The returned file or directory does not necessarily be the first file or directory created in the root directory. The name of the first file or directory can change as files are deleted and created in the root directory.

Example

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

void SampleFindFirstFile(void) {
  FS_FIND_DATA fd;
  int          r;
  char         acFileName[32];
  char         ac[100];
  FS_FILETIME  FileTime;

  //
```



```
  // List the files from the "SubDir1" directory.
  //
  r = FS_FindFirstFile(&fd, "SubDir1", acFileName, sizeof(acFileName));
  if (r == 0) {
    do {
      //
      // Ignore "." and ".." entries.
      //
      if (fd.sFileName[0] == '.') {
        continue;
      }
      //
      // Show information about the file.
      //
      if (fd.Attributes & FS_ATTR_DIRECTORY) {
        sprintf(ac, "Directory name: %s\n", fd.sFileName);
      } else {
        sprintf(ac, "File name:      %s\n", fd.sFileName);
      }
      FS_X_Log(ac);
      sprintf(ac, "File size:      %lu\n", fd.FileSize);
      FS_X_Log(ac);
      sprintf(ac,
              "Attributes:     %c%c%c%c%c\n",
              (fd.Attributes & FS_ATTR_READ_ONLY) ? 'R' : '-',
              (fd.Attributes & FS_ATTR_HIDDEN)    ? 'H' : '-',
              (fd.Attributes & FS_ATTR_SYSTEM)    ? 'S' : '-',
              (fd.Attributes & FS_ATTR_ARCHIVE)   ? 'A' : '-',
              (fd.Attributes & FS_ATTR_DIRECTORY) ? 'D' : '-');
      FS_X_Log(ac);
      FS_TimeStampToFileTime(fd.CreationTime, &FileTime);
      sprintf(ac, "Creation time:  %d-%.2d-%.2d %.2d:%.2d:%.2d",
            FileTime.Year, FileTime.Month,  FileTime.Day,
            FileTime.Hour, FileTime.Minute, FileTime.Second);
      FS_X_Log(ac);
      FS_TimeStampToFileTime(fd.LastAccessTime, &FileTime);
      sprintf(ac, "Access time:    %d-%.2d-%.2d %.2d:%.2d:%.2d",
            FileTime.Year, FileTime.Month,  FileTime.Day,
            FileTime.Hour, FileTime.Minute, FileTime.Second);
      FS_X_Log(ac);
      FS_TimeStampToFileTime(fd.LastWriteTime, &FileTime);
      sprintf(ac, "Write time:     %d-%.2d-%.2d %.2d:%.2d:%.2d",
            FileTime.Year, FileTime.Month,  FileTime.Day,
            FileTime.Hour, FileTime.Minute, FileTime.Second);
      FS_X_Log(ac);
      FS_X_Log("--\n");
    } while (FS_FindNextFile(&fd));
  }
  FS_FindClose(&fd);
}
```


4.7.5 FS_FindNextFile()

Description

Returns information about the next file or directory in a directory scanning operation.

Prototype

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

Parameters

Parameter	Description
pFD	Context of the directory scanning operation.

Return value

1 O.K.
0 Error occurred or end of directory reached.

Additional information

FS_FindNextFile() returns information about the next file or directory located after the file or directory returned via a previous call to FS_FindFirstFile() or FS_FindNextFile() within the same listing context.

The name of the file or directory is returned in sFileName member of pFD. FS_FindNextFile() stores at most one less than the buffer size characters in the buffer and it terminates the name of the file or directory by a 0 character. The buffer for the file or directory name is specified in the call to FS_FindFirstFile(). The file or directory name is truncated if it contains more characters than the number of characters that can be stored in sFileName. For files or directories stored on a FAT volume with LFN support the name is truncated by returning the short version (8.3 format) of the file or directory name.

Additional information about the file or directory such as size, attributes and timestamps are returned via the corresponding members of pFD structure. For more information refer to FS_FIND_DATA.

FS_FindFirstFile() has to be called first to initialize the listing operation. It is allowed to perform operations on files or directories (creation, deletion, etc.) within the listing loop.

Example

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

void SampleFindFirstFile(void) {
  FS_FIND_DATA fd;
  int          r;
  char         acFileName[32];
  char         ac[100];
  FS_FILETIME  FileTime;

  //
  // List the files from the "SubDir1" directory.
  //
  r = FS_FindFirstFile(&fd, "SubDir1", acFileName, sizeof(acFileName));
  if (r == 0) {
    do {
      //
      // Ignore "." and ".." entries.
      //
      if (fd.sFileName[0] == '.') {
        continue;
      }
      //
      // Show information about the file.
      //
```



```
      if (fd.Attributes & FS_ATTR_DIRECTORY) {
        sprintf(ac, "Directory name: %s\n", fd.sFileName);
      } else {
        sprintf(ac, "File name:      %s\n", fd.sFileName);
      }
      FS_X_Log(ac);
      sprintf(ac, "File size:      %lu\n", fd.FileSize);
      FS_X_Log(ac);
      sprintf(ac,
              "Attributes:     %c%c%c%c%c\n",
              (fd.Attributes & FS_ATTR_READ_ONLY) ? 'R' : '-',
              (fd.Attributes & FS_ATTR_HIDDEN)    ? 'H' : '-',
              (fd.Attributes & FS_ATTR_SYSTEM)    ? 'S' : '-',
              (fd.Attributes & FS_ATTR_ARCHIVE)   ? 'A' : '-',
              (fd.Attributes & FS_ATTR_DIRECTORY) ? 'D' : '-');
      FS_X_Log(ac);
      FS_TimeStampToFileTime(fd.CreationTime, &FileTime);
      sprintf(ac, "Creation time:  %d-%.2d-%.2d %.2d:%.2d:%.2d",
            FileTime.Year, FileTime.Month,  FileTime.Day,
            FileTime.Hour, FileTime.Minute, FileTime.Second);
      FS_X_Log(ac);
      FS_TimeStampToFileTime(fd.LastAccessTime, &FileTime);
      sprintf(ac, "Access time:    %d-%.2d-%.2d %.2d:%.2d:%.2d",
            FileTime.Year, FileTime.Month,  FileTime.Day,
            FileTime.Hour, FileTime.Minute, FileTime.Second);
      FS_X_Log(ac);
      FS_TimeStampToFileTime(fd.LastWriteTime, &FileTime);
      sprintf(ac, "Write time:     %d-%.2d-%.2d %.2d:%.2d:%.2d",
            FileTime.Year, FileTime.Month,  FileTime.Day,
            FileTime.Hour, FileTime.Minute, FileTime.Second);
      FS_X_Log(ac);
      FS_X_Log("--\n");
    } while (FS_FindNextFile(&fd));
  }
  FS_FindClose(&fd);
}
```


4.7.6 FS_FindNextFileEx()

Description

Returns information about the next file or directory in a directory scanning operation.

Prototype

```
int FS_FindNextFileEx(FS_FIND_DATA * pFD);
```

Parameters

Parameter	Description
pFD	Context of the directory scanning operation.

Return value

= 1 No files or directories available in directory.
= 0 OK, information about the file or directory returned.
< 0 Error code indicating the failure reason.

Additional information

FS_FindNextFileEx() performs the same operation as FS_FindNextFile() with the difference that its return value is consistent with the return value of the other emFile API functions.

4.7.7 FS_MkDir()

Description

Creates a directory.

Prototype

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

Parameters

Parameter	Description
sDirName	Directory name.

Return value

= 0 OK, directory has been created
≠ 0 Error code indicating the failure reason

Additional information

The function fails if a directory with the same name already exists in the target directory. FS_MkDir() expects that all the directories in the path to the created directory exists. Otherwise the function returns with an error. FS_CreateDir() can be used to create a directory along with any missing directories in the path.

Example

```
#include "FS.h"

void SampleMkDir(void) {
  FS_MkDir("SubDir1");
}
```


4.7.8 FS_RmDir()

Description

Removes a directory.

Prototype

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

Parameters

Parameter	Description
sDirName	Directory name.

Return value

= 0 OK, directory has been removed
≠ 0 Error code indicating the failure reason

Additional information

The function fails if the directory is not empty. `FS_DeleteDir()` can be used to remove a directory and its contents.

Example

```
#include "FS.h"

void SampleRmDir(void) {
  FS_RmDir("SubDir1");
}
```


4.7.9 FS_FIND_DATA

Description

Information about a file or directory.

Type definition

```
typedef struct {
  U8      Attributes;
  U32     CreationTime;
  U32     LastAccessTime;
  U32     LastWriteTime;
  U32     FileSize;
  char  * sFileName;
  int     SizeofFileName;
  FS_DIR  Dir;
} FS_FIND_DATA;
```

Structure members

Member	Description
Attributes	Attributes of the file or directory.
CreationTime	Date and time when the file or directory was created.
LastAccessTime	Date and time when the file or directory was accessed last.
LastWriteTime	Date and time when the file or directory was modified last.
FileSize	Size of the file. Set to 0 for a directory.
sFileName	Name of the file or directory as 0-terminated string. It points to the buffer passed as argument to FS_FindFirstFile().
SizeofFileName	Internal. Do not use.
Dir	Internal. Do not use.

Additional information

This structure contains also the context for the file listing operation. These members are considered internal and should not be used by the application. FS_FIND_DATA is used as context by the FS_FindFirstFile() and FS_FindNextFile() pair of functions.

4.8 Formatting functions

A storage device has to be formatted before it can be used to write data to it. SD and MMC memory cards as well as USB drives are usually already preformatted. Normally, the NAND and NOR flash devices have to be reformatted. These storage devices require a low-level format first, followed by a high-level format. The low-level format is device-specific while the high-level format depends on the type of file system.

4.8.1 FS_Format()

Description

Performs a high-level format.

Prototype

```
int FS_Format(const char           * sVolumeName,
              const FS_FORMAT_INFO * pFormatInfo);
```

Parameters

Parameter	Description
sVolumeName	Volume name.
pFormatInfo	in Additional format information. Can be NULL.

Return value

= 0 O.K., format successful.
≠ 0 Error code indicating the failure reason.

Additional information

The high-level format operation has to be performed once before using the storage device for the first time. FS_Format() stores the management information required by the file system on the storage device. This means primarily the initialization of the allocation table and of the root directory, as well as of the BIOS Parameter Block (BPB) for a volume formatted as FAT and of the Information Sector for a volume formatted as EFS.

The type of file system can be selected at compile time via FS_SUPPORT_FAT and FS_SUPPORT_EFS defines. If both file systems are enabled at compile time the type of file system can be configured via FS_SetFSType().

There are many different ways to format a medium, even with one file system. If the pFormatInfo parameter is not specified, reasonable default values are used (auto-format). However, FS_Format() allows fine-tuning of the parameters used. For increased performance it is recommended to format the storage with clusters as large as possible. The larger the cluster the smaller gets the number of accesses to the allocation table the file system has to perform during a read or write operation. For more information about format parameters see FS_FORMAT_INFO.

Example

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

void SampleFormat(void) {
  FS_FORMAT_INFO FormatInfo;
  //
  // Format default volume using reasonable defaults.
  //
  FS_Format("", NULL);
  //
  // Format the "mmc:0:" volume using 32 Kbyte clusters
  // assuming that the size of the logical sector used
  // by the storage device is 512 bytes.
  //
  memset(&FormatInfo, 0, sizeof(FormatInfo));
  FormatInfo.SectorsPerCluster = 64;
  FS_Format("mmc:0:", &FormatInfo);
}
```


4.8.2 FS_FormatLLIfRequired()

Description

Performs a low-level format.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to be formatted.

Return value

= 0 O.K., low	level format successful.
= 1 Low	level format not required.
< 0	Error code indicating the failure reason.

Additional information

This function performs a low-level format of a storage device if it is not already low-level formatted. `FS_FormatLLIfRequired()` does nothing if the storage device is already low-level formatted. A storage device has to be low-level formatted once before the file system can perform any data access. All the data present on the storage device is lost after a low-level format.

The low-level format operation is required only for the storage devices that are managed by the file system such as NAND and NOR flash devices. SD cards and e.MMC devices do not require a low-level format.

Example

```
#include "FS.h"

void SampleFormatLLIfRequired(void) {
  FS_Init();
  //
  // Perform a low-level format if required.
  // Equivalent to the following sequence:
  //
  // if (FS_IsLLFormatted("nand:0:") == 0) {
  //   FS_FormatLow("nand:0:");
  // }
  //
  FS_FormatLLIfRequired("nand:0:");
}
```


4.8.3 FS_FormatLow()

Description

Performs a low-level format.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to be formatted.

Return value

= 0	O.K., Low level format successful.
≠ 0	Error code indicating the failure reason.

Additional information

This function prepares a storage device for the data access. The file system reports an error if an attempt is made to access a storage device that is not low-level formatted. All the data present on the storage device is lost after a low-level format.

The low-level format operation has to be performed only for the storage devices that are managed by the file system such as NAND and NOR flash devices. SD cards and e.MMC devices do not require a low-level format.

Example

```
#include "FS.h"

void SampleFormatLow(void) {
  FS_Init();
  //
  // Performs a low-level format of the NOR flash.
  //
  FS_FormatLow("nor:0:");
}
```


4.8.4 FS_IsHLFormatted()

Description

Checks if a volume is high-level formatted or not.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to be checked.

Return value

= 1	Volume is high level formatted.
= 0	Volume is not high level formatted.
< 0	Error code indicating the failure reason.

Additional information

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

Example

```
#include "FS.h"

void SampleIsHLFormatted(void) {
  FS_Init();
  //
  // Perform a high-level format if required.
  //
  if (FS_IsHLFormatted("") == 0) {
    FS_Format("", NULL);
  }
}
```


4.8.5 FS_IsLLFormatted()

Description

Returns whether a volume is low-level formatted or not.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to be checked. It cannot be NULL.

Return value

= 1	Volume is low level formatted
= 0	Volume is not low level formatted
< 0	Error code indicating the failure reason

Example

```
#include "FS.h"

void SampleIsLLFormatted(void) {
  FS_Init();
  //
  // Perform a low-level format if required.
  // Equivalent to the following sequence:
  //
  // FS_FormatLLIfRequired("nor:0:");
  //
  if (FS_IsLLFormatted("nor:0:") == 0) {
    FS_FormatLow("nor:0:");
  }
}
```


4.8.6 FS_FORMAT_INFO

Description

Parameters for the high-level format.

Type definition

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

Structure members

Member	Description
SectorsPerCluster	Number of sectors in a cluster.
NumRootDirEntries	Number of directory entries in the root directory.
pDevInfo	Information about the storage device.

Additional information

This structure is passed as parameter to FS_Format() to specify additional information about how the storage device has to be formatted.

A cluster is the minimal unit size a file system can handle. Sectors are combined together to form a cluster. SectorsPerCluster has to be a power of 2 value, for example 1, 2, 4, 8, 16, 32, 64. Larger values lead to a higher read / write performance when working with large files while low values (1) make more efficient use of disk space.

- Allowed values for FAT: 1--128
- Allowed values for EFS: 1--32768

NumRootDirEntries represents the number of directory entries in the root directory should have. This is only a proposed value. The actual value depends on the FAT type. This value is typically used for FAT12 or FAT16 formatted volume that have a fixed number of entries in the root directory. On FAT32 formatted volume the root directory can grow dynamically. The file system uses a default value of 256 if NumRootDirEntries is set to 0.

pDevInfo should be typically set to NULL unless some specific information about the storage device has to be passed to format function. The file system requests internally the information from storage device if pDevInfo is set to NULL.

4.9 File system structure checking

The functions in this section can be used by an application to check the consistency of the file system structure.

4.9.1 FS_CheckAT()

Description

Verifies the consistency of the allocation table.

Prototype

```
int FS_CheckAT(FS_CHECK_DATA * pCheckData);
```

Parameters

Parameter	Description
pCheckData	in Checking context. It cannot be NULL.

Return value

= FS_CHECKDISK_RETVAL_OK	No errors found or the callback returned FS_CHECKDISK_ACTION_DO_NOT_REPAIR.
= FS_CHECKDISK_RETVAL_RETRY	An error has been found. The error has been corrected since the callback function returned FS_CHECKDISK_ACTION_SAVE_CLUSTERS or FS_CHECKDISK_ACTION_DELETE_CLUSTERS. FS_CheckDir() has to be called again to check for the next error.
= FS_CHECKDISK_RETVAL_ABORT	The application requested the abort of disk checking operation through the callback returning FS_CHECKDISK_ACTION_ABORT.
= FS_CHECKDISK_RETVAL_CONTINUE	Indicates that not all the allocation table has been checked. The entire volume has to be checked again using FS_CheckDir().
< 0	Error code indicating the failure reason.

4.9.2 FS_CheckDir()

Description

Verifies the consistency of a single directory.

Prototype

```
int FS_CheckDir(      FS_CHECK_DATA * pCheckData,
                const char          * sPath);
```

Parameters

Parameter	Description
pCheckData	in Checking context. It cannot be NULL.
sPath	Path to the directory to be checked. It cannot be NULL.

Return value

= FS_CHECKDISK_RETVAL_OK	No errors found or the callback returned FS_CHECKDISK_ACTION_DO_NOT_REPAIR.
= FS_CHECKDISK_RETVAL_RETRY	An error has been found. The error has been corrected since the callback function returned FS_CHECKDISK_ACTION_SAVE_CLUSTERS or FS_CHECKDISK_ACTION_DELETE_CLUSTERS. FS_CheckDir() has to be called again to check for the next error.
= FS_CHECKDISK_RETVAL_ABORT	The application requested the abort of disk checking operation through the callback returning FS_CHECKDISK_ACTION_ABORT.
= FS_CHECKDISK_RETVAL_MAX_RECURSE	Maximum recursion level reached. The disk checking operation has been aborted.
< 0	Error code indicating the failure reason.

4.9.3 FS_CheckDisk()

Description

Checks the file system for corruption.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to be checked.
pBuffer	Work buffer to be used for checking the allocation table.
BufferSize	Size of the work buffer in bytes. It cannot be NULL.
MaxRecursionLevel	The maximum directory depth the function is allowed to checks.
pfOnError	Function that has to be called when an error is found. It cannot be NULL.

Return value

= FS_CHECKDISK_RETVAL_OK	No errors found or the callback returned FS_CHECKDISK_ACTION_DO_NOT_REPAIR.
= FS_CHECKDISK_RETVAL_RETRY	An error has been found. The error has been corrected since the callback function returned FS_CHECKDISK_ACTION_SAVE_CLUSTERS or FS_CHECKDISK_ACTION_DELETE_CLUSTERS. FS_CheckDisk() has to be called again to check for the next error.
= FS_CHECKDISK_RETVAL_ABORT	The application requested the abort of disk checking operation through the callback returning FS_CHECKDISK_ACTION_ABORT.
= FS_CHECKDISK_RETVAL_MAX_RECURSE	Maximum recursion level reached. The disk checking operation has been aborted.
< 0	Error code indicating the failure reason.

Additional information

This function can be used to check if any errors are present on a specific volume and, if necessary, to repair these errors. Ideally, the work buffer has to be large enough to store the usage information of all the clusters in the allocation table. FS_CheckDisk() uses one bit to store the usage state of a cluster. The typical size of the work buffer is about 2 KBytes. Additional iterations are performed if the work buffer is not large enough to check the whole allocation table in one step.

FS_CheckDisk() can detect and correct the following file system errors:

- Invalid directory entries.
- Lost clusters or cluster chains.
- Cross-linked clusters.
- Clusters are associated to a file with size of 0.
- Too few clusters are allocated to a file.
- Cluster is not marked as end-of-chain, although it should be.

The contents of a lost cluster chain is saved during the repair operation to files named FILE<FileIndex>.CHK that are stored in directories named FOUND.<DirIndex>. FileIndex is a 0-based 4-digit decimal number that is incremented by one for each cluster chain saved. DirIndex is a 0-based 3-digit decimal number that is incremented by one each time FS_CheckDisk() is called. For example the first created directory has the name FOUND.000, the second FOUND.001, and so on while the first file in the directory has the name FILE0000.CHK, the second FILE0001.CHK, and so on.

The callback function is used to notify the application about the errors found by FS_CheckDisk() during the disk checking operation. FS_CheckDisk() uses the return value of the callback function to decide if the error hast to be repaired or not. For more information refer to FS_CHECKDISK_ON_ERROR_CALLBACK.

FS_CheckDisk() closes all opened files before it starts the disk checking operation. The application is not allowed to access the storage device from a different task as long as the operation is in progress.

Example

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

static U32 _aBuffer[1024 / 4];
static int _NumErrors;

/*********************************************************************
*
*       _OnError
*/
int _OnError(int ErrCode, ...) {
  va_list      ParamList;
  const char * sFormat;
  char         c;
  char         ac[256];

  (void)memset(&ParamList, 0, sizeof(ParamList));
  sFormat = FS_CheckDisk_ErrCode2Text(ErrCode);
  if (sFormat) {
    va_start(ParamList, ErrCode);
    vsprintf(ac, sFormat, ParamList);
    FS_X_Log(ac);
    FS_X_Log("\n");
  }
  if (ErrCode != FS_CHECKDISK_ERRCODE_CLUSTER_UNUSED) {
    FS_X_Log("  Do you want to repair this? (y/n/a)");
  } else {
    FS_X_Log("  * Convert lost cluster chain into file (y)\n"
             "  * Delete cluster chain                 (d)\n"
             "  * Do not repair                        (n)\n"
             "  * Abort                                (a) ");
    FS_X_Log("\n");
  }
  _NumErrors++;
  c = getchar();
  FS_X_Log("\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 repair anything.
}

/*********************************************************************
*
*       SampleCheckDisk
*/
void SampleCheckDisk(void) {
  int r;
```



```
  while (1) {
    r = FS_CheckDisk("", _aBuffer, sizeof(_aBuffer), 5, _OnError);
    if (r != FS_CHECKDISK_RETVAL_RETRY) {
      break;
    }
  }
  if (_NumErrors == 0) {
    FS_X_Log("No errors were found.\n");
  } else {
    FS_X_Log("Errors were found.\n");
  }
}
```


4.9.4 FS_CheckDisk_ErrCode2Text()

Description

Returns a human-readable text description of a disk checking error code.

Prototype

```
char *FS_CheckDisk_ErrCode2Text(int ErrCode);
```

Parameters

Parameter	Description
ErrCode	Error code for which the text description has to be returned.

Return value

Text description as 0-terminated string.

Additional information

This function can be invoked inside the callback for FS_CheckDisk() to format the error information in human-readable format. The text description includes format specifiers for the printf() family of functions that can be used to show additional information about the file system error.

Refer to FS_CheckDisk() for more information about how to use this function.

4.9.5 FS_InitCheck()

Description

Initializes a non-blocking disk checking operation.

Prototype

```
int FS_InitCheck(      FS_CHECK_DATA                  * pCheckData,
                 const char                           * sVolumeName,
                       void                           * pBuffer,
                       U32                              BufferSize,
                       FS_CHECKDISK_ON_ERROR_CALLBACK * pfOnError);
```

Parameters

Parameter	Description
pCheckData	out Checking context. It cannot be NULL.
sVolumeName	Name of the volume on which the checking is performed. It cannot be NULL.
pBuffer	in Working buffer. It cannot be NULL.
BufferSize	Number of bytes in pBuffer.
pfOnError	Callback function to be invoked in case of a file system damage. It cannot be NULL.

Return value

= FS_CHECKDISK_RETVAL_OK	OK, disk checking has been initialized.
< 0	Error code indicating the failure reason.

4.10 File system extended functions

This section describes API functions that do not fall in any other category. It includes function that operate on volumes, utility functions as well as functions for locking the file system against concurrent access from different tasks.

4.10.1 FS_ConfigEOFErrorSuppression()

Description

Enables / disables the reporting of end-of-file condition as error.

Prototype

```
void FS_ConfigEOFErrorSuppression(int OnOff);
```

Parameters

Parameter	Description
OnOff	Activation status of the option. • 0 EOF is reported as error. • 1 EOF is not reported as error.

Additional information

The end-of-file indicator of a file handle is set to 1 as soon as the application tries to read more bytes than available in the file. This function controls if an error is reported via FS_FError() when the end-of-file indicator is set for a file handle. The default is to report the end-of-file condition as error. The configuration has effect on all the opened file handles.

4.10.2 FS_ConfigPOSIXSupport()

Description

Enables / disables support for the POSIX-like behavior.

Prototype

```
void FS_ConfigPOSIXSupport(int OnOff);
```

Parameters

Parameter	Description
OnOff	Activation status of the option. • 0 POSIX behavior is disabled. • 1 POSIX behavior is enabled.

Additional information

The end-of-file indicator of a file handle is set to 1 as soon as the application tries to read more bytes than available in the file. This function controls if an error is reported via FS_FError() when the end-of-file indicator is set for a file handle. The default is to report the end-of-file condition as error. The configuration has effect on all the opened file handles.

4.10.3 FS_ConfigWriteVerification()

Description

Enables / disables the verification of the written data.

Prototype

```
void FS_ConfigWriteVerification(int OnOff);
```

Parameters

Parameter	Description
OnOff	Activation status of the option. • 0 Verification is disabled. • 1 Verification is enabled.

Additional information

If enabled, this feature requests the file system to check that the data has been written correctly to storage device. This operation is performed by reading back and comparing the read with the written data. The verification is performed one logical sector at a time. A sector buffer is allocated from the memory pool for this operation.

4.10.4 FS_CreateMBR()

Description

Updates the Master Boot Record (MBR) of a volume.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Volume name for which the MBR has to be updated.
pPartInfo	in List of the partitions to be created.
NumPartitions	Number of partitions to be create.

Return value

= 0 OK, MBR created.
≠ 0 Error code indicating the failure reason.

Additional information

MBR (Master Boot Record) is a special sector that contains information about how the storage device is partitioned. This information is stored to the first sector of a storage device. The information stored to MBR can be queried via FS_GetPartitionInfo().

The function overwrites any information stored to the sector index 0 of the specified volume. The partition entries are stored in the order specified in the pPartInfo array: the information from pPartInfo[0] is stored to first partition entry, the information from pPartInfo[1] is stored to the second one, and so on.

If the Type field of the FS_PARTITION_INFO structure is set to 0 the function determines automatically the partition type and the CHS (Cylinder/Head/Sector) addresses (Type, StartAddr and EndAddr) based on the values stored in the StartSector and NumSector members of the same structure.

The DISKPART logical driver can be used to get access to the created partitions. pVolumeName has to be the name of a volume assigned to a device driver.

Example

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

void SampleCreateMBR(void) {
  FS_PARTITION_INFO aPartInfo[2];

  //
  // This example creates a MBR with 2 partitions.
  // The first partition is configured to be bootable.
  // All parameters are explicitly configured.
  // The second partition is configure not to be bootable
  // The partition type and CHS address are calculated
  // automaticall by FS_CreateMBR().
  //
  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);
}
```


4.10.5 FS_FileTimeToTimeStamp()

Description

Converts a broken-down date and time specification to a timestamp.

Prototype

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

Parameters

Parameter	Description
pFileTime	in Broken-down date and time to be converted. It cannot be NULL.
pTimeStamp	out Converted timestamp. It cannot be NULL.

Additional information

This function can be used to convert a broken-down date and time specification to a timestamp used by the file system. The converted timestamp can be directly passed to FS_SetFileTime() or FS_SetFileTimeEx() to change the timestamps of files and directories.

For a description of the timestamp format refer to FS_GetFileTime().

Example

```
#include "FS.h"

void SampleSetFileTime(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);
}
```


4.10.6 FS_FreeSectors()

Description

Informs the device driver about unused sectors.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume on which to perform the operation.

Return value

= 0 OK, sectors have been freed.
≠ 0 Error code indicating the failure reason.

Additional information

The function visits each entry of the allocation table and checks if the cluster is used to store data. If the cluster is free, 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 device drivers to optimize the internal wear-leveling process.

To use FS_FreeSectors() the support for "free sector" operation has to be enabled in the file system, that is FS_SUPPORT_FREE_SECTOR has to be set to 1. The function does nothing if FS_SUPPORT_FREE_SECTOR is set to 0.

This function is optional. The file system informs automatically the device drivers about unused sectors.

Example

```
#include "FS.h"

void SampleFreeSectors(void) {
  FS_FreeSectors("nand:0:");
}
```


4.10.7 FS_GetFileId()

Description

Calculates a value that uniquely identifies a file.

Prototype

```
int FS_GetFileId(const char * sFileName,
                 U8   * pId);
```

Parameters

Parameter	Description
sFileName	Name of the file for which to calculate the id.
pId	A 16-byte array where the id is stored.

Return value

= 0 Id returned OK.
≠ 0 Error code indicating the failure reason.

Additional information

The calculated value is a combination of the sector number that stores the directory entry assigned to file combined with the index of the directory index. Typically used by USB MTP component of SEGGER emUSB-Device to create an unique object id to the file.

4.10.8 FS_GetFileWriteMode()

Description

Returns the write mode.

Prototype

```
FS_WRITEMODE FS_GetFileWriteMode(void);
```

Return value

WriteMode Specifies how to write to file:

- `FS_WRITEMODE_SAFE` Updates the allocation table and the directory entry at each write to file operation.
- `FS_WRITEMODE_MEDIUM` Updates the allocation table at each write to file operation.
- `FS_WRITEMODE_FAST` The allocation table and directory entry are updated when the file is closed.
- `FS_WRITEMODE_UNKNOWN` An error occurred.

Additional information

This function can be used to query the write mode configured for the entire file system. The write mode for the entire file system can be configured via `FS_SetFileWriteMode()`. The write mode is set by default to `FS_WRITEMODE_SAFE` when the file system is initialized.

Refer to `FS_SetFileWriteMode()` for detailed information about the different write modes.

4.10.9 FS_GetFileWriteModeEx()

Description

Returns the write mode configured for a specified volume.

Prototype

```
FS_WRITEMODE FS_GetFileWriteModeEx(const char * sVolumeName);
```

Parameters

Parameter	Description
sVolumeName	Identifies the volume that have to be queried.

Return value

WriteMode Specifies how to write to file:

- FS_WRITEMODE_SAFE Updates the allocation table and the directory entry at each write to file operation.
- FS_WRITEMODE_MEDIUM Updates the allocation table at each write to file operation.
- FS_WRITEMODE_FAST The allocation table and directory entry are updated when the file is closed.
- FS_WRITEMODE_UNKNOWN An error occurred.

Additional information

This function can be used to query the write mode configured for the specified volume. The write mode of the volume can be configured via FS_SetFileWriteModeEx(). If the write mode is not explicitly configured by the application for the volume via FS_SetFileWrite-ModeEx() then the write mode configured for the entire file system is returned. The write mode for the entire file system can be configured via FS_SetFileWriteMode(). The write mode is set by default to FS_WRITEMODE_SAFE when the file system is initialized.

Refer to FS_SetFileWriteMode() for detailed information about the different write modes.

4.10.10 FS_GetFSType()

Description

Returns the type of file system assigned to volume.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to be queried. It cannot be NULL.

Return value

≥ 0 File system type (FS_FAT or FS_EFS)
< 0 Error code indicating the failure reason.

Additional information

This function is optional and available only when both the FAT and the EFS file system are enabled in the file system, that is the FS_SUPPORT_FAT and FS_SUPPORT_EFS configuration defines are both set to 1 (multiple-volume configuration).

Example

```
#include "FS.h"

void SampleGetFSType(void) {
  int FSType;

#if FS_SUPPORT_MULTIPLE_FS
  FSType = FS_GetFSType("");
#else
  FSType = FS_FAT;
#endif
  FS_X_Log("File system type: ");
  switch (FSType) {
  case FS_FAT:
    FS_X_Log("FAT");
    break;
  case FS_EFS:
    FS_X_Log("EFS");
    break;
  default:
    FS_X_Log("<unknown>");
    break;
  }
  FS_X_Log("\n");
}
```


4.10.11 FS_GetMaxSectorSize()

Description

Queries the maximum configured logical sector size.

Prototype

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

Return value

Maximum logical sector size in bytes.

Additional information

Default value of the maximum logical sector size is 512 bytes. Refer to `FS_SetMaxSectorSize()` for more information about the maximum logical sector size.

Example

```
#include "FS.h"

void SampleGetMaxSectorSize(void) {
  U32 MaxSectorSize;

  MaxSectorSize = FS_GetMaxSectorSize();
}
```


4.10.12 FS_GetMemInfo()

Description

Returns information about the memory management.

Prototype

```
int FS_GetMemInfo(FS_MEM_INFO * pMemInfo);
```

Parameters

Parameter	Description
pMemInfo	out Returned information.

Return value

= 0 OK, information returned.
≠ 0 Error code indicating the failure reason.

Additional information

The application can use this function to obtain information about the memory management such as number of bytes allocated by the file system, type of memory management used, etc.

4.10.13 FS_GetMountType()

Description

Returns information about how a volume is mounted.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to be queried.

Return value

= 0	Volume is not mounted.
= FS_MOUNT_RO	Volume is mounted read only.
= FS_MOUNT_RW	Volume is mounted read/write.
< 0	Error code indicating the failure reason. Refer to FS_ErrorNo2Text().

Additional information

sVolumeName is the name of a volume that already exists. If the volume name is not known to file system then an error is returned. Alternatively, the application can call FS_IsVolumeMounted() if the information about how the volume is actually mounted is not important. After the file system initialization all volumes are in unmounted state.

Example

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

void SampleGetMountType(void) {
  int          r;
  const char * sMountType;

  r = FS_GetMountType("");
  switch (r) {
  case 0:
    sMountType = "not mounted";
    break;
  case FS_MOUNT_RO:
    sMountType = "read only";
    break;
  case FS_MOUNT_RW:
    sMountType = "read/write";
    break;
  default:
    sMountType = FS_ErrorNo2Text(r);
    break;
  }
  printf("Mount type: %s\n");
}
```


4.10.14 FS_GetNumFilesOpen()

Description

Queries the number of opened file handles.

Prototype

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

Return value

Total number of file handles that are open.

Additional information

This function counts the number of file handles that were opened by the application via `FS_FOpen()` or `FS_FOpenEx()` and that were not closed yet via `FS_FClose()`.

The returned value is not the actual number of files opened because the same file can be opened by an application more than one time. For example, `FS_GetNumFilesOpen()` returns 2 if the same file is opened by the application twice.

Example

```
#include "FS.h"

void SampleGetNumFilesOpen(void) {
  FS_FILE * pFile1;
  FS_FILE * pFile2;
  int       NumFileHandles;

  FS_Init();
  //
  // The number of opened files is 0 at this stage.
  //
  NumFileHandles = FS_GetNumFilesOpen();
  //
  // Open the first file.
  //
  pFile1 = FS_FOpen("File1.txt", "w");
  //
  // The number of opened files is 1 at this stage.
  //
  NumFileHandles = FS_GetNumFilesOpen();
  //
  // Open the second file.
  //
  pFile2 = FS_FOpen("File2.txt", "w");
  //
  // The number of opened files is 2 at this stage.
  //
  NumFileHandles = FS_GetNumFilesOpen();
  //
  // Close first file.
  //
  FS_FClose(pFile1);
  //
  // The number of opened files is 1 at this stage.
  //
  NumFileHandles = FS_GetNumFilesOpen();
  //
  // Close second file.
  //
  FS_FClose(pFile2);
  //
  // The number of opened files is 0 at this stage.
  //
  NumFileHandles = FS_GetNumFilesOpen();
}
```


4.10.15 FS_GetNumFilesOpenEx()

Description

Queries the number of opened file handles on a volume.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume (0-terminated string). NULL is permitted.

Return value

≥ 0 Total number of file handles that are open on the volume.
< 0 Error code indicating the failure reason.

Additional information

This function counts the number of file handles that were opened by the application via FS_FOpen() or FS_FOpenEx() and that were not closed yet via FS_FClose() on a specified volume.

The returned value is not the actual number of files opened because the same file can be opened by an application more than one time. For example, FS_GetNumFilesOpenEx() returns 2 if the same file is opened by the application twice.

FS_GetNumFilesOpenEx() works in the same way as FS_GetNumFilesOpen() if sVolumeName is set to NULL.

4.10.16 FS_GetNumVolumes()

Description

Queries the number of configured volumes.

Prototype

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

Return value

Number of volumes.

Additional information

This function can be used to check how many volumes are configured in the file system. Each call to `FS_AddDevice()` creates a separate volume. Calling `FS_AddPhysDevice()` does not create a volume. The maximum number of volumes is limited only by available memory.

`FS_GetNumVolumes()` can be used together with `FS_GetVolumeName()` to list the names of all configured volumes.

Example

```
#include "FS.h"

void SampleGetVolumeName(void) {
  int  NumVolumes;
  int  iVolume;
  int  BufferSize;
  int  NumBytesStored;
  char acVolumeName[32];

  BufferSize = sizeof(acVolumeName);
  NumVolumes = FS_GetNumVolumes();
  FS_X_Log("Available volumes:\n");
  for (iVolume = 0; iVolume < NumVolumes; iVolume++) {
    NumBytesStored = FS_GetVolumeName(iVolume, acVolumeName, BufferSize);
    if (NumBytesStored < BufferSize) {
      FS_X_Log("  ");
      FS_X_Log(acVolumeName);
      FS_X_Log("\n");
    }
  }
}
```


4.10.17 FS_GetPartitionInfo()

Description

Returns information about a MBR partition.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume on which the MBR is located.
pPartInfo	out Information about the partition.
PartIndex	Index of the partition to query.

Return value

= 0 OK, partition information read.
≠ 0 Error code indicating the failure reason.

Additional information

The function reads the information from the Master Boot Record (MBR) that is stored on the first sector (the sector with the index 0) of the specified volume. An error is returned if no MBR information is present on the volume. If the Type member of the FS_PARTITION_INFO structure is 0, the partition entry is not valid. FS_NUM_PARTITIONS specifies the maximum number of partitions in MBR.

Example

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

void SampleGetPartitionInfo(void) {
  int               iPart;
  FS_PARTITION_INFO PartInfo;
  char              ac[100];

  //
  // Show the contents of the partition list stored in the Master
  // Boot Record. Only the valid entries are displayed.
  //
  for (iPart = 0; iPart < FS_NUM_PARTITIONS; ++iPart) {
    FS_GetPartitionInfo("", &PartInfo, iPart);
    if (PartInfo.Type) {
      sprintf(ac, "  Index:         %u\n", iPart);
      FS_X_Log(ac);
      sprintf(ac, "  StartSector:   %lu\n", PartInfo.StartSector);
      FS_X_Log(ac);
      sprintf(ac, "  NumSectors:    %lu\n", PartInfo.NumSectors);
      FS_X_Log(ac);
      sprintf(ac, "  Type:          %u\n", PartInfo.Type);
      FS_X_Log(ac);
      sprintf(ac, "  IsActive:      %u\n", PartInfo.IsActive);
      FS_X_Log(ac);
      sprintf(ac, "  FirstCylinder: %u\n", PartInfo.StartAddr.Cylinder);
      FS_X_Log(ac);
      sprintf(ac, "  FirstHead:     %u\n", PartInfo.StartAddr.Head);
      FS_X_Log(ac);
      sprintf(ac, "  FirstSector:   %u\n", PartInfo.StartAddr.Sector);
      FS_X_Log(ac);
      sprintf(ac, "  LastCylinder:  %u\n", PartInfo.EndAddr.Cylinder);
      FS_X_Log(ac);
      sprintf(ac, "  LastHead:      %u\n", PartInfo.EndAddr.Head);
```



```
      FS_X_Log(ac);
      sprintf(ac, "  LastSector:    %u\n", PartInfo.EndAddr.Sector);
      FS_X_Log(ac);
      FS_X_Log("\n");
    }
  }
}
```


4.10.18 FS_GetVolumeFreeSpace()

Description

Returns the free space available on a volume.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to be checked.

Return value

≠ 0 Number of bytes available on the volume.
= 0 An error occurred.

Additional information

This function returns the free space available to application to store files and directories.

A free space larger than four Gbytes is reported as 0xFFFFFFFF because this is the maximum value that can be represented in an unsigned 32-bit integer. The function FS_GetVolumeFreeSpaceKB() can be used instead if the available free space is larger than four Gbytes.

FS_GetVolumeFreeSpace() can sill be reliably used if the application does not need to know if there is more than four GBytes of free space available.

Example

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

void SampleGetVolumeFreeSpace(void) {
  U32  NumBytes;
  char ac[100];

  NumBytes = FS_GetVolumeFreeSpace("");
  if (NumBytes < 0x8000) {
    sprintf(ac, "Free space: %lu Kbytes\n", NumBytes);
  } else {
    NumBytes >>= 10;
    sprintf(ac, "Free space: %lu Mbytes\n", NumBytes);
  }
  FS_X_Log(ac);
}
```


4.10.19 FS_GetVolumeFreeSpaceFirst()

Description

Initiates the search for free space.

Prototype

```
int FS_GetVolumeFreeSpaceFirst(      FS_FREE_SPACE_DATA * pFSD,
                               const char               * sVolumeName,
                                     void               * pBuffer,
                                     int                  SizeOfBuffer);
```

Parameters

Parameter	Description
pFSD	Context of the free space search.
sVolumeName	Name of the volume to search on.
pBuffer	Work buffer.
SizeOfBuffer	Size of the work buffer in bytes.

Return value

= 1 OK, the entire allocation table has been searched.
= 0 OK, search is not completed.
< 0 Error code indicating the failure reason.

Additional information

FS_GetVolumeFreeSpaceFirst() together with FS_GetVolumeFreeSpaceNext() can be used to calculate the amount of available free space on a volume. This pair of functions implement the same functionality as FS_GetVolumeFreeSpace() with the difference that they block the access to the file system for a very short time.

Typically, FS_GetVolumeFreeSpace() uses the information stored to the FSInfo sector of a FAT file system or to the Status sector of an EFS file system to get the number of free clusters on the volume. If this information is not available then FS_GetVolumeFreeSpace() calculates the free space by reading and evaluating the entire allocation table. The size of the allocation table depends on the size of the volume with sizes of a few tens of Mbytes for a volume pf 4 Gbytes or larger. Reading such amount of data can take a relatively long time to complete during which the application is not able to access the file system. FS_GetVolumeFreeSpaceFirst() and FS_GetVolumeFreeSpaceNext() can be used in such cases by calculating the available free space while the application is performing other file system accesses.

FS_GetVolumeFreeSpaceFirst() is used by the application to initiate the search process followed by one or more calls to FS_GetVolumeFreeSpaceNext(). The free space is returned in the NumClustersFree member of pFSD. The cluster size can be determined by calling FS_GetVolumeInfoEx() with the Flags parameter set to 0.

Example

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

void SampleGetVolumeFreeSpace(void) {
  char               ac[100];
  FS_FREE_SPACE_DATA fsd;
  int                r;
  FS_DISK_INFO       VolumeInfo;
  U32                BytesPerCluster;
  U32                NumKBytesFree;

  r = FS_GetVolumeFreeSpaceFirst(&fsd, "", NULL, 0);
```



```
  if (r < 0) {
    sprintf("Could not initiate searching (%s)\n", FS_ErrorNo2Text(r));
    FS_X_Log(ac);
  } else {
    if (r == 0) {
      for (;;) {
        r = FS_GetVolumeFreeSpaceNext(&fsd);
        if (r < 0) {
          sprintf("Could not continue searching (%s)\n", FS_ErrorNo2Text(r));
          FS_X_Log(ac);
          break;
        }
        if (r == 1) {
          // End of allocation table reached.
          break;
        }
      }
    }
  }
  FS_GetVolumeInfoEx("", &VolumeInfo, 0);
  BytesPerCluster = (VolumeInfo.BytesPerSector * VolumeInfo.SectorsPerCluster);
  NumKBytesFree   = fsd.NumClustersFree * (BytesPerCluster >> 10);
  sprintf(ac, "Free space: %lu Kbytes\n", NumKBytesFree);
  FS_X_Log(ac);
}
```


4.10.20 FS_GetVolumeFreeSpaceKB()

Description

Returns the free space available on a volume.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to be checked.

Return value

≠ 0 The space available on the volume in Kbytes.
= 0 An error occurred.

Additional information

This function returns the free space available to application to store files and directories.

Example

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

void SampleGetVolumeFreeSpaceKB(void) {
  U32  NumKBytes;
  char ac[100];

  NumKBytes = FS_GetVolumeFreeSpaceKB("");
  sprintf(ac, "Free space: %lu Kbytes\n", NumKBytes);
  FS_X_Log(ac);
}
```


4.10.21 FS_GetVolumeFreeSpaceNext()

Description

Continues the search for free space.

Prototype

```
int FS_GetVolumeFreeSpaceNext(FS_FREE_SPACE_DATA * pFSD);
```

Parameters

Parameter	Description
pFSD	Context of the free space search.

Return value

= 1 OK, the entire allocation table has been searched.
= 0 OK, search is not completed.
< 0 Error code indicating the failure reason.

Additional information

pFSD has to be initialized via a call to FS_GetVolumeFreeSpaceFirst(). One or more calls to FS_GetVolumeFreeSpaceNext() are required in order to determine the available free space. For more information refer to FS_GetVolumeFreeSpaceFirst().

Example

For a sample usage refer to FS_GetVolumeFreeSpaceFirst()

4.10.22 FS_GetVolumeInfo()

Description

Returns information about a volume.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to query. It cannot be NULL.
pInfo	out Volume information. It cannot be NULL.

Return value

= 0 OK, information returned.
≠ 0 Error code indicating the failure reason.

Additional information

This function collects the information about the volume such as volume size, available free space, format type, etc.

Example

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

void SampleGetVolumeInfo(void) {
  FS_DISK_INFO   VolumeInfo;
  int            r;
  char           ac[100];
  const char   * sFSType;

  memset(&VolumeInfo, 0, sizeof(VolumeInfo));
  r = FS_GetVolumeInfo("", &VolumeInfo);
  if (r == 0) {
    sprintf(ac, "Number of total clusters:         %lu\n", VolumeInfo.NumTotalClusters);
    FS_X_Log(ac);
    sprintf(ac, "Number of free clusters:          %lu\n", VolumeInfo.NumFreeClusters);
    FS_X_Log(ac);
    sprintf(ac, "Sectors per cluster:              %d\n", VolumeInfo.SectorsPerCluster);
    FS_X_Log(ac);
    sprintf(ac, "Bytes per sector:                 %d\n", VolumeInfo.BytesPerSector);
    FS_X_Log(ac);
    sprintf(ac, "Number of root directory entries: %d\n", VolumeInfo.NumRootDirEntries);
    FS_X_Log(ac);
    switch (VolumeInfo.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;
    }
```



```
    sprintf(ac, "File system type:                %s\n", sFSType);
    FS_X_Log(ac);
    sprintf(ac, "Formatted acc. to SD spec.:      %s\n", VolumeInfo.IsSDFormatted
                                                           ? "Yes" : "No");
    FS_X_Log(ac);
    sprintf(ac, "Has been modified:               %s\n", VolumeInfo.IsDirty
                                                           ? "Yes" : "No");
    FS_X_Log(ac);
  }
}
```


4.10.23 FS_GetVolumeInfoEx()

Description

Returns information about a volume. Identical to FS_GetVolumeInfo except it gives the application more control about which type of information should be returned.

Prototype

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

Parameters

Parameter	Description
sVolumeName	The volume name.
pInfo	out Volume information.
Flags	Bit mask that controls what type of information the function returns. It is an or-combination of FS_DISKINFO_FLAG_... defines.

Return value

= 0 OK, information returned.
≠ 0 Error code indicating the failure reason.

Additional information

This function collects the information about the volume such as volume size, available free space, format type, etc. The kind of information is actually returned can be controlled via the Flags parameter.

Example

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

void SampleGetVolumeInfoEx(void) {
  FS_DISK_INFO   VolumeInfo;
  int            r;
  char           ac[100];
  const char   * sFSType;

  memset(&VolumeInfo, 0, sizeof(VolumeInfo));
  //
  // Return also the available free space.
  //
  r = FS_GetVolumeInfoEx("", &VolumeInfo, FS_DISKINFO_FLAG_FREE_SPACE);
  if (r == 0) {
    sprintf(ac, "Number of total clusters:         %lu\n", VolumeInfo.NumTotalClusters);
    FS_X_Log(ac);
    sprintf(ac, "Number of free clusters:          %lu\n", VolumeInfo.NumFreeClusters);
    FS_X_Log(ac);
    sprintf(ac, "Sectors per cluster:              %d\n", VolumeInfo.SectorsPerCluster);
    FS_X_Log(ac);
    sprintf(ac, "Bytes per sector:                 %d\n", VolumeInfo.BytesPerSector);
    FS_X_Log(ac);
    sprintf(ac, "Number of root directory entries: %d\n", VolumeInfo.NumRootDirEntries);
    FS_X_Log(ac);
    switch (VolumeInfo.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;
    }
    sprintf(ac, "File system type:                %s\n", sFSType);
    FS_X_Log(ac);
    sprintf(ac, "Formatted acc. to SD spec.:      %s\n", VolumeInfo.IsSDFormatted
                                                         ? "Yes" : "No");
    FS_X_Log(ac);
    sprintf(ac, "Has been modified:               %s\n", VolumeInfo.IsDirty
                                                         ? "Yes" : "No");
    FS_X_Log(ac);
  }
}
```


4.10.24 FS_GetVolumeLabel()

Description

Returns the label of the volume.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Identifies the volume to be queried. It cannot be NULL.
sVolumeLabel	Buffer that receives the volume label. It cannot be NULL.
VolumeLabelSize	Number of characters in pVolumeName buffer.

Return value

= 0 OK, volume label set.
≠ 0 Error code indicating the failure reason.

Additional information

The function stores at most VolumeLabelSize - 1 bytes to pVolumeLabel buffer. The label string is terminated by a 0. The volume label is truncated if it contains more characters than the number of characters that can be stored to pVolumeLabel.

EFS does not have a volume label. FS_GetVolumeLabel() returns with an error if the application tries to read the volume label of a volume formatted as EFS.

Example

```
#include "FS.h"

void SampleGetVolumeName(void) {
  char acVolumeLabel[16];
  int  r;

  r = FS_GetVolumeLabel("", acVolumeLabel, sizeof(acVolumeLabel));
  if (r == 0) {
    FS_X_Log("Volume label: ");
    FS_X_Log(acVolumeLabel);
    FS_X_Log("\n");
  }
}
```


4.10.25 FS_GetVolumeName()

Description

Returns the name of a volume.

Prototype

```
int FS_GetVolumeName(int    VolumeIndex,
                     char * sVolumeName,
                     int    VolumeNameSize);
```

Parameters

Parameter	Description
VolumeIndex	0-based index of the volume to be queried.
sVolumeName	out Receives the name of the volume as 0-terminated string. It cannot be NULL.
VolumeNameSize	Number of bytes in sVolumeName.

Return value

> 0 Number of bytes required to store the volume name.
< 0 An error occurred.

Additional information

If the function succeeds, the return value is the length of the string copied to sVolumeName in bytes, excluding the 0-terminating character. FS_GetVolumeName() stores at most VolumeNameSize - 1 characters to sVolumeName and it terminates the string by a 0.

VolumeIndex specifies the position of the volume in the internal volume list of the file system. The first volume added to file system in FS_X_AddDevices() via FS_AddDevice() is stored at index 0 in the volume list, the second volume added via FS_AddDevice() is store at the index 1 in the volume list and so on. The total number of volumes can be queried via FS_GetNumVolumes().

If the sVolumeName is too small to hold the entire volume name, the return value is the size of the buffer required to hold the volume name plus the terminating 0 character. Therefore, if the return value is greater than VolumeNameSize, the application has to call FS_GetVolumeName() again with a buffer that is large enough to hold the volume name.

Example

```
#include "FS.h"

void SampleGetVolumeName(void) {
  int  NumVolumes;
  int  iVolume;
  int  BufferSize;
  int  NumBytesStored;
  char acVolumeName[32];

  BufferSize = sizeof(acVolumeName);
  NumVolumes = FS_GetNumVolumes();
  FS_X_Log("Available volumes:\n");
  for (iVolume = 0; iVolume < NumVolumes; iVolume++) {
    NumBytesStored = FS_GetVolumeName(iVolume, acVolumeName, BufferSize);
    if (NumBytesStored < BufferSize) {
      FS_X_Log("  ");
      FS_X_Log(acVolumeName);
      FS_X_Log("\n");
    }
  }
}
```


4.10.26 FS_GetVolumeSize()

Description

Returns the size of a volume.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to be queried.

Return value

≠ 0 Total number of bytes available to file system.
= 0 An error occurred.

Additional information

This function returns the total number of bytes available to file system as data storage. The actual number of bytes available to application for files and directories is smaller than the value returned by `FS_GetVolumeSize()` since the file system requires some space for the boot sector and the allocation table.

A size larger than four Gbytes is reported as `0xFFFFFFFF` because this is the maximum value that can be represented in an unsigned 32-bit integer. The function `FS_GetVolumeSizeKB()` can be used instead if the volume size is larger than four Gbytes.

Example

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

void SampleGetVolumeSize(void) {
  U32  NumBytes;
  char ac[100];

  NumBytes = FS_GetVolumeSize("");
  if (NumBytes < 0x8000) {
    sprintf(ac, "Volume size: %lu Kbytes\n", NumBytes);
  } else {
    NumBytes >>= 10;
    sprintf(ac, "Volume size: %lu Mbytes\n", NumBytes);
  }
  FS_X_Log(ac);
}
```


4.10.27 FS_GetVolumeSizeKB()

Description

Returns the size of a volume.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to be queried.

Return value

≠ 0 Storage available to file system in Kbytes.
= 0 An error occurred.

Additional information

This function returns the total number of bytes available to file system as data storage. The actual number of bytes available to application for files and directories is smaller than the value returned by FS_GetVolumeSize() since the file system requires some space for the boot sector and the allocation table.

Example

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

void SampleGetVolumeSizeKB(void) {
  U32  NumKBytes;
  char ac[100];

  NumKBytes = FS_GetVolumeSizeKB("");
  sprintf(ac, "Volume size: %lu Kbytes\n", NumKBytes);
  FS_X_Log(ac);
}
```


4.10.28 FS_GetVolumeStatus()

Description

Returns the presence status of a volume.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume. It cannot be NULL.

Return value

FS_MEDIA_NOT_PRESENT	Storage device is not present.
FS_MEDIA_IS_PRESENT	Storage device is present.
FS_MEDIA_STATE_UNKNOWN	Presence status is unknown.

Additional information

This function can be used to check if a removable storage device that is assigned to a volume is present or not. FS_GetVolumeStatus() is typically called periodically from a separate task to handle the insertion and removal of a removable storage device.

Example

```
#include "FS.h"
#include "FS_OS.h"

void SampleUnmountForced(void) {
  int IsPresentNew;
  int IsPresent;

  IsPresent = FS_GetVolumeStatus("");
  while (1) {
    IsPresentNew = FS_GetVolumeStatus("");
    //
    // Check if the presence status of the storage device has been changed.
    //
    if (IsPresentNew != IsPresent) {
      if (IsPresentNew == FS_MEDIA_IS_PRESENT) {
        FS_Mount("");
      } else  {
        FS_UnmountForced("");
      }
      IsPresent = IsPresentNew;
    }
    FS_X_OS_Delay(500);
  }
}
```


4.10.29 FS_IsVolumeMounted()

Description

Checks if a volume is mounted.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to be queried.

Return value

1 Volume is mounted.
0 Volume is not mounted or does not exist.

Additional information

The function returns 1 if the volume is mounted either in read-only mode or in read / write mode.

Example

```
#include "FS.h"

void SampleIsVolumeMounted(void) {
  int IsMounted;

  IsMounted = FS_IsVolumeMounted("");
  if (IsMounted) {
    FS_X_Log("Volume is mounted.\n");
  } else {
    FS_X_Log("Volume is not mounted.\n");
  }
}
```


4.10.30 FS_Lock()

Description

Claims 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 file system grants it exclusive access. After the task gets exclusive access to file system the other tasks that try to perform file system operations are blocked until the task calls `FS_Unlock()`.

`FS_Lock()` is typically used by applications that call device driver functions from different tasks. These functions are usually not protected against concurrent accesses. Additionally, `FS_Lock()` can be used to protect a group of file system operations against concurrent access.

`FS_Lock()` is available if `FS_OS_LOCKING` set to 1. The function does nothing if `FS_OS_LOCKING` set to 0 or 2. The calls to `FS_Lock()` / `FS_Unlock()` cannot be nested.

The API functions of the file system are multitasking safe. It is not required to explicitly lock these function calls via `FS_Lock()` / `FS_Unlock()`. All API functions call internal versions of `FS_Lock()` and `FS_Unlock()` on function entry and exit respectively.

Example

```
#include "FS.h"

void SampleLock(void) {
  FS_FILE           * pFile;
  FS_NAND_DISK_INFO   DiskInfo;

  //
  // The device driver functions are not protected against concurrent
  // access from different tasks. Make sure that only one task can access
  // the file system.
  //
#if (FS_OS_LOCKING == 1)
  FS_Lock();
#endif // FS_OS_LOCKING == 1
  FS_NAND_UNI_GetDiskInfo(0, &DiskInfo);
#if (FS_OS_LOCKING == 1)
  FS_Unlock();
#endif // FS_OS_LOCKING == 1
  //
  // Make sure that the write operation is not interrupted
  // by a different task.
  //
#if (FS_OS_LOCKING == 1)
  FS_Lock();
#endif // FS_OS_LOCKING == 1
  pFile = FS_FOpen("Test.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Test", 4);
    FS_FClose(pFile);
  }
#if (FS_OS_LOCKING == 1)
  FS_Unlock();
#endif // FS_OS_LOCKING == 1
}
```


4.10.31 FS_LockVolume()

Description

Claims exclusive access to a volume.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume that has to be locked.

Additional information

The execution of the task that calls this function is suspended until the file system grants it exclusive access to the specified volume. After the task gets exclusive access to volume the other tasks that try to perform file system operations on that volume are blocked until the task calls FS_UnlockVolume() with the same volume name as in the call to FS_LockVolume().

FS_LockVolume() is typically used by applications that call device driver functions from different tasks. These functions are usually not protected against concurrent accesses. Additionally, FS_LockVolume() can be used to protect a group of file system operations against concurrent access.

FS_LockVolume() is available if FS_OS_LOCKING set to 2. The function does nothing if FS_OS_LOCKING set to 0 or 1. The calls to FS_LockVolume() / FS_UnlockVolume() cannot be nested.

The API functions of the file system are multitasking safe. It is not required to explicitly lock these function calls via FS_LockVolume() / FS_UnlockVolume(). All API functions call internal versions of FS_LockVolume() and FS_UnlockVolume() on function entry and exit respectively.

Example

```
#include "FS.h"

void SampleLockVolume(void) {
  FS_FILE           * pFile;
  FS_NAND_DISK_INFO   DiskInfo;

  //
  // The device driver functions are not protected against concurrent
  // access from different tasks. Make sure that only one task can access
  // the file system.
  //
#if (FS_OS_LOCKING == 2)
  FS_LockVolume("nand:0:");
#endif // FS_OS_LOCKING == 2
  FS_NAND_UNI_GetDiskInfo(0, &DiskInfo);
#if (FS_OS_LOCKING == 2)
  FS_UnlockVolume("nand:0:");
#endif // FS_OS_LOCKING == 2
  //
  // Make sure that the write operation is not interrupted
  // by a different task.
  //
#if (FS_OS_LOCKING == 2)
  FS_LockVolume("nand:0:");
#endif // FS_OS_LOCKING == 2
  pFile = FS_FOpen("nand:0:\\Test.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Test", 4);
    FS_FClose(pFile);
  }
```



```
#if (FS_OS_LOCKING == 2)
  FS_UnlockVolume("nand:0:");
#endif // FS_OS_LOCKING == 2
}
```


4.10.32 FS_SetBusyLEDCallback()

Description

Registers a callback for busy status changes of a volume.

Prototype

```
int FS_SetBusyLEDCallback(const char                 * sVolumeName,
                          FS_BUSY_LED_CALLBACK * pfBusyLED);
```

Parameters

Parameter	Description
sVolumeName	Name of the volume for which the callback has to be registered. It cannot be NULL.
pfBusyLED	Function to be called when the busy status changes. It cannot be NULL.

Return value

= 0 OK, callback registered.
≠ 0 Error code indicating the failure reason.

Additional information

The application can use this FS_SetBusyLEDCallback() to register a function that is called by the file system each time the busy status of a volume changes. The volume becomes busy when it starts an access to storage device. When the access to storage device ends the volume becomes ready. The busy status of a volume can change several times during a single file system operation.

FS_SetBusyLEDCallback() is available if the FS_SUPPORT_BUSY_LED configuration define is set to 1. The function does nothing if FS_SUPPORT_BUSY_LED is set to 0.

Example

```
#include "FS.h"

static void _cbBusyLED(U8 OnOff) {
  if (OnOff) {
    //
    // Set LED on.
    //
  } else {
    //
    // Set LED off.
    //
  }
}

void SampleSetBusyLEDCallback(void) {
  FS_FILE * pFile;

  FS_SetBusyLEDCallback("nand:0:", _cbBusyLED);
  pFile = FS_FOpen("nand:0:\\File.txt", "w");
  FS_FClose(pFile);
}
```


4.10.33 FS_SetCharSetType()

Description

Configures the character set that is used for the file and directory names.

Prototype

```
void FS_SetCharSetType(const FS_CHARSET_TYPE * pCharSetType);
```

Parameters

Parameter	Description
pCharSetType	Type of character set used.

Additional information

Permitted values of pCharSetType are:

Identifier	Description
FS_CHARSET_CP437	Latin characters
FS_CHARSET_CP932	Japanese characters (encoded as Shift JIS)

4.10.34 FS_SetFSType()

Description

Sets the type of file system a volume.

Prototype

```
int FS_SetFSType(const char * sVolumeName,
                 int    FSType);
```

Parameters

Parameter	Description
sVolumeName	Name of the volume for which the type of the file system has to be set. It cannot be NULL.
FSType	Type of file system. It take be one these values: • FS_FAT FAT file system. • FS_EFS SEGGER's Embedded File System.

Return value

= 0 OK, file system type set.
≠ 0 Error code indicating the failure reason.

Additional information

This function is optional and available only when both the FAT and the EFS file system are enabled in the file system, that is the FS_SUPPORT_FAT and FS_SUPPORT_EFS configuration defines are both set to 1 (multiple-volume configuration).

In a multiple-volume configuration the application has to call FS_SetFSType() before formatting a volume that has not been formatted before or when the volume has been formatted using a different file system type.

Example

```
#include "FS.h"

void SampleSetFSType(void) {
  //
  // Format the "nand:0:" volume as EFS.
  //
#if FS_SUPPORT_MULTIPLE_FS
  FS_SetFSType("nand:0:", FS_EFS);
#endif // FS_SUPPORT_MULTIPLE_FS
  FS_Format("nand:0:", NULL);
  //
  // Format the "nor:0:" volume as FAT.
  //
#if FS_SUPPORT_MULTIPLE_FS
  FS_SetFSType("nor:0:", FS_FAT);
#endif // FS_SUPPORT_MULTIPLE_FS
  FS_Format("nor:0:", NULL);
}
```


4.10.35 FS_SetMemCheckCallback()

Description

Registers a callback for checking of 0-copy operations.

Prototype

```
int FS_SetMemCheckCallback(const char                  * sVolumeName,
                           FS_MEM_CHECK_CALLBACK * pfMemCheck);
```

Parameters

Parameter	Description
sVolumeName	Name of the volume for which the callback has to be registered. It cannot be NULL.
pfMemCheck	Function to be called before a 0-copy operation is executed. It cannot be NULL.

Return value

= 0 OK, callback registered.
≠ 0 Error code indicating the failure reason.

Additional information

FS_SetMemCheckCallback() can be used by an application to register a function that is called by the file system before any read or write operation to check if a data buffer can be used in 0-copy operation. In a 0-copy operation, a pointer to the data is passed directly to the device driver instead of the data being copied first in an internal buffer and then being passed it to device driver.

Example

```
#include "FS.h"

#if FS_SUPPORT_CHECK_MEMORY

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

#endif // FS_SUPPORT_CHECK_MEMORY

void SampleSetMemCheckCallback(void) {
  FS_FILE * pFile;

#if FS_SUPPORT_CHECK_MEMORY
  FS_SetMemCheckCallback("", _cbMemCheck);
#endif // FS_SUPPORT_CHECK_MEMORY
  pFile = FS_FOpen("File.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Test\n", 5);
  }
  FS_FClose(pFile);
}
```


4.10.36 FS_SetTimeDateCallback()

Description

Configures a function that the file system can use to get the current time and date.

Prototype

```
void FS_SetTimeDateCallback(FS_TIME_DATE_CALLBACK * pfTimeDate);
```

Parameters

Parameter	Description
pfTimeDate	Function to be invoked.

Additional information

During the initialization of the file system via `FS_Init()` the callback function is initialized to point to `FS_X_GetTimeDate()`.

4.10.37 FS_SetVolumeAlias()

Description

Assigns an alternative name for a volume.

Prototype

```
int FS_SetVolumeAlias(const char * sVolumeName,
                      const char * sVolumeAlias);
```

Parameters

Parameter	Description
sVolumeName	in Name of the volume to which the alternative name has to be assigned. It cannot be set to NULL.
sVolumeAlias	in Alternative name as 0-terminated string. Can be set to NULL.

Return value

= 0 OK, the alternative name has been assigned.
< 0 Error code indicating the failure reason.

Additional information

The assigned alias can be used as replacement in any path to a file or directory that contains a volume name and it can be passed instead of a volume name to any API function that requires one. The alias replaces the volume and the unit number. When used as a volume name the volume separator character (':') has to be added to the end of the alias otherwise it is not recognized as a valid volume name by the file system.

Valid characters in an alias are ASCII capital and small letters, digits and the underscore character. The comparison applied to alias is case sensitive and is performed after the file system checks the real volume names.

The alias name is copied to the internal instance of the volume. The function fails with an error if the alias is longer that the space available in the internal buffer. The size of the internal buffer for the alias can be configured at compile time via FS_MAX_LEN_VOLUME_ALIAS. The alias can be removed by either passing a NULL or an empty string as sVolumeAlias parameter.

This function is active only when FS_MAX_LEN_VOLUME_ALIAS is set to a value larger than 0.

The alias of a volume can be queried either via FS_GetVolumeInfo() or FS_GetVolumeInfoEx(). The sAlias member of the FS_DISK_INFO structure stores the configured alias.

4.10.38 FS_SetVolumeLabel()

Description

Modifies the label of a volume.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume for which the label has to be modified. It cannot be NULL.
sVolumeLabel	Volume label as 0-terminated ASCII string. The volume label is deleted if set to NULL.

Return value

= 0 OK, volume label set.
≠ 0 Error code indicating the failure reason.

Additional information

The volume label is a ASCII string that can be assigned to a volume. It is not evaluated by the file system and it cannot be used as volume name. A Windows PC shows the volume label of a removable storage device in the "Computer" window when it is mounted via USB MSD. The volume label can be read via FS_GetVolumeLabel().

The volume label of a FAT-formatted volume can contain at most 11 characters. The following characters are not allowed in a volume name: '"', '&', '*', '+', '-', ',', '.', '/', ':', ';', '<', '=', '>', '?', '[', ']', '\'.

EFS does not have a volume label. FS_GetVolumeLabel() returns with an error if the application tries to read the volume label of a volume formatted as EFS.

Example

```
#include "FS.h"

void SampleSetVolumeName(void) {
  FS_SetVolumeLabel("", "SEGGER");
}
```


4.10.39 FS_TimeStampToFileTime()

Description

Converts a timestamp to a broken-down date and time specification.

Prototype

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

Parameters

Parameter	Description
TimeStamp	Timestamp to be converted.
pFileTime	out Converted broken-down date and time. It cannot be NULL.

Additional information

This function can be used to convert a timestamp as used by the file system a broken-down date and time specification.

For a description of the timestamp format refer to FS_GetFileTime().

Example

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

void SampleGetFileTime(void) {
  char        ac[100];
  U32         TimeStamp;
  FS_FILETIME FileTime;
  int         r;

  r = FS_GetFileTime("Test.txt", &TimeStamp);
  if (r == 0) {
    FS_TimeStampToFileTime(TimeStamp, &FileTime);
    sprintf(ac, "Creation time: %d-%.2d-%.2d %.2d:%.2d:%.2d",
            FileTime.Year, FileTime.Month,  FileTime.Day,
            FileTime.Hour, FileTime.Minute, FileTime.Second);
    FS_X_Log(ac);
  }
}
```


4.10.40 FS_Unlock()

Description

Releases the exclusive access to file system.

Prototype

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

Additional information

This function has to be called in pair with `FS_Lock()` to allow other tasks to access the file system. For each call to `FS_Lock()` the application has to call `FS_Unlock()`.

`FS_Unlock()` is available if `FS_OS_LOCKING` set to 1. The function does nothing if `FS_OS_LOCKING` set to 0 or 2. The calls to `FS_Lock()` / `FS_Unlock()` cannot be nested.

Example

For sample usage refer to `FS_Lock()`.

4.10.41 FS_UnlockVolume()

Description

Releases the exclusive access to a volume.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume that has to be locked.

Additional information

This function has to be called in pair with FS_LockVolume() to allow other tasks to access the volume. For each call to FS_LockVolume() the application has to call FS_UnlockVolume() with the same volume name as parameter.

FS_UnlockVolume() is available if FS_OS_LOCKING set to 2. The function does nothing if FS_OS_LOCKING set to 0 or 1. The calls to FS_LockVolume() / FS_UnlockVolume() cannot be nested.

Example

For sample usage refer to FS_LockVolume().

4.10.42 Disk checking error codes

Description

Error codes reported by `FS_CheckDisk()` during operation.

Definition

```
#define FS_CHECKDISK_ERRCODE_0FILE                      0x10
#define FS_CHECKDISK_ERRCODE_SHORTEN_CLUSTER            0x11
#define FS_CHECKDISK_ERRCODE_CROSSLINKED_CLUSTER        0x12
#define FS_CHECKDISK_ERRCODE_FEW_CLUSTER                0x13
#define FS_CHECKDISK_ERRCODE_CLUSTER_UNUSED             0x14
#define FS_CHECKDISK_ERRCODE_CLUSTER_NOT_EOC            0x15
#define FS_CHECKDISK_ERRCODE_INVALID_CLUSTER            0x16
#define FS_CHECKDISK_ERRCODE_INVALID_DIRECTORY_ENTRY    0x17
#define FS_CHECKDISK_ERRCODE_SECTOR_NOT_IN_USE          0x18
```

Symbols

Definition	Description
FS_CHECKDISK_ERRCODE_0FILE	A cluster chain is assigned to a file that do not contain data (file size is 0)
FS_CHECKDISK_ERRCODE_SHORTEN_CLUSTER	The cluster chain assigned to a file is longer than the size of the file.
FS_CHECKDISK_ERRCODE_CROSSLINKED_CLUSTER	A cluster is assigned to more than one file or directory.
FS_CHECKDISK_ERRCODE_FEW_CLUSTER	The size of the file is larger than the cluster chain assigned to file.
FS_CHECKDISK_ERRCODE_CLUSTER_UNUSED	A cluster is marked as in use, but not assigned to any file or directory.
FS_CHECKDISK_ERRCODE_CLUSTER_NOT_EOC	A cluster is does not have end-of-chain marker.
FS_CHECKDISK_ERRCODE_INVALID_CLUSTER	Invalid cluster id.
FS_CHECKDISK_ERRCODE_INVALID_DIRECTORY_ENTRY	Invalid directory entry.
FS_CHECKDISK_ERRCODE_SECTOR_NOT_IN_USE	A logical sector that stores data is not marked as in use in the device driver.

Additional information

The error codes are reported via the ErrCode parameter of the callback function.

Example

For a sample usage refer to `FS_CheckDisk()`.

4.10.43 Disk checking return values

Description

Error codes returned by `FS_CheckDisk()`.

Definition

```
#define FS_CHECKDISK_RETVAL_OK              0
#define FS_CHECKDISK_RETVAL_RETRY           1
#define FS_CHECKDISK_RETVAL_ABORT           2
#define FS_CHECKDISK_RETVAL_MAX_RECURSE     3
#define FS_CHECKDISK_RETVAL_CONTINUE        4
#define FS_CHECKDISK_RETVAL_SKIP            5
```

Symbols

Definition	Description
FS_CHECKDISK_RETVAL_OK	OK, file system not in corrupted state
FS_CHECKDISK_RETVAL_RETRY	An error has be found and repaired, retry is required.
FS_CHECKDISK_RETVAL_ABORT	User aborted operation via callback or API call
FS_CHECKDISK_RETVAL_MAX_RECURSE	Maximum recursion level reached, operation aborted
FS_CHECKDISK_RETVAL_CONTINUE	`FS_CheckAT()` returns this value to indicate that the allocation table has not been entirely checked.
FS_CHECKDISK_RETVAL_SKIP	`FS_CheckDir()` returns this value to indicate that the directory entry does not have to be checked.

Example

For a sample usage refer to `FS_CheckDisk()`.

4.10.44 Disk checking action codes

Description

Values returned by the FS_CheckDisk() callback function.

Definition

```
#define FS_CHECKDISK_ACTION_DO_NOT_REPAIR      0
#define FS_CHECKDISK_ACTION_SAVE_CLUSTERS      1
#define FS_CHECKDISK_ACTION_ABORT              2
#define FS_CHECKDISK_ACTION_DELETE_CLUSTERS    3
```

Symbols

Definition	Description
FS_CHECKDISK_ACTION_DO_NOT_REPAIR	The error does not have to be repaired.
FS_CHECKDISK_ACTION_SAVE_CLUSTERS	The data stored in the clusters of a faulty cluster chain has to be saved to a file.
FS_CHECKDISK_ACTION_ABORT	The disk checking operation has to be aborted.
FS_CHECKDISK_ACTION_DELETE_CLUSTERS	The data stored in the clusters of a faulty cluster chain has to be deleted.

Additional information

These values indicate FS_CheckDisk() how to handle a file system error.

Example

For a sample usage refer to FS_CheckDisk().

4.10.45 File system types

Description

Types of file systems supported by emFile.

Definition

```
#define FS_FAT                0
#define FS_EFS                1
#define FS_FAT_TYPE_UNKONWN   0u
#define FS_FAT_TYPE_FAT12     12u
#define FS_FAT_TYPE_FAT16     16u
#define FS_FAT_TYPE_FAT32     32u
```

Symbols

Definition	Description
FS_FAT	File Allocation Table (FAT).
FS_EFS	SEGGER's Embedded File System (EFS).
FS_FAT_TYPE_UNKONWN	Internal use.
FS_FAT_TYPE_FAT12	Internal use.
FS_FAT_TYPE_FAT16	Internal use.
FS_FAT_TYPE_FAT32	Internal use.

4.10.46 Format types

Description

Types of format supported by the emFile.

Definition

```
#define FS_TYPE_FAT12    0x000C
#define FS_TYPE_FAT16    0x0010
#define FS_TYPE_FAT32    0x0020
#define FS_TYPE_EFS      0x0120
```

Symbols

Definition	Description
FS_TYPE_FAT12	FAT with 12-bit allocation table entries.
FS_TYPE_FAT16	FAT with 16-bit allocation table entries.
FS_TYPE_FAT32	FAT with 32-bit allocation table entries.
FS_TYPE_EFS	SEGGER’s Embedded File System (EFS).

4.10.47 FS_BUSY_LED_CALLBACK

Description

Type of function called by the file system to report a change in the busy status of a volume.

Type definition

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

Parameters

Parameter	Description
OnOff	Indicates if the volume is busy or not. • 1 The volume is busy. • 0 The volume is ready.

Additional information

A function of this type can be registered with the file system via FS_SetBusyLEDCallback(). FS_BUSY_LED_CALLBACK() is called by the file system each time the volume changes the busy status from busy to ready and reverse. Therefore, an application can use FS_BUSY_LED_CALLBACK()to indicate the busy / ready status of a volume via LED or by other means.

The file system calls FS_BUSY_LED_CALLBACK() with OnOff set to 1 when the volume goes busy. When the volume becomes ready FS_BUSY_LED_CALLBACK() is called again with OnOff set to 0.

Example

For a sample usage refer to FS_SetBusyLEDCallback().

4.10.48 FS_CHECKDISK_ON_ERROR_CALLBACK

Description

Type of function called by FS_CheckDisk() to report an error.

Type definition

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

Parameters

Parameter	Description
ErrCode	Code of the reported error.

Return value

Value that indicates FS_CheckDisk() how the error should be handled.

Additional information

ErrCode is one of the FS_CHECKDISK_ERRCODE_... defines. The value returned by FS_CHECKDISK_ON_ERROR_CALLBACK() can be one of the FS_CHECKDISK_ACTION_... defines.

In addition to ErrCode FS_CheckDisk() can pass other parameters FS_CHECKDISK_ON_ERROR_CALLBACK() providing information about the reported error. These parameters can be directly passed to a printf()-style function along with the format returned by FS_CheckDisk_ErrCode2Text() to create a text description of the error in human-readable format.

Example

For a sample usage refer to FS_CheckDisk().

4.10.49 FS_CHS_ADDR

Description

Address of physical block on a mechanical drive.

Type definition

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

Structure members

Member	Description
Cylinder	Cylinder number (0-based)
Head	Read / write head (0-based)
Sector	Index of the sector on a cylinder.

4.10.50 FS_DISK_INFO

Description

Information about a volume.

Type definition

```
typedef struct {
  U32          NumTotalClusters;
  U32          NumFreeClusters;
  U16          SectorsPerCluster;
  U16          BytesPerSector;
  U16          NumRootDirEntries;
  U16          FSType;
  U8           IsSDFormatted;
  U8           IsDirty;
  const char * sAlias;
} FS_DISK_INFO;
```

Structure members

Member	Description
NumTotalClusters	Total number of clusters on the storage device.
NumFreeClusters	Number of clusters that are not in use.
SectorsPerCluster	Number of sectors in a cluster.
BytesPerSector	Size of the logical sector in bytes.
NumRootDirEntries	Number of directory entries in the root directory. This member is valid only for volumes formatted as FS_TYPE_FAT12 or FS_TYPE_FAT16.
FSType	Type of file system. One of FS_TYPE_... defines.
IsSDFormatted	Set to 1 if the volume has been formatted according to SD specification. This member is valid only for volumes formatted as FAT.
IsDirty	Set to 1 if the volume was not unmounted correctly or the file system modified the storage. This member is valid only for volumes formatted as FAT.
sAlias	Alternative name of the volume (0-terminated). Set to NULL if the volume alias feature is disabled.

Additional information

IsDirty can be used to check if the volume formatted as FAT has been correctly unmounted before a system reset. IsDirty is set to 1 at file system initialization if the file system was not properly unmounted.

Example

For a sample usage refer to FS_GetVolumeInfo().

4.10.51 FS_FILETIME

Description

Time and date representation.

Type definition

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

Structure members

Member	Description
Year	Year (The value has to be greater than 1980.)
Month	Month (1--12, 1: January, 2: February, etc.)
Day	Day of the month (1--31)
Hour	Hour (0--23)
Minute	Minute (0--59)
Second	Second (0--59)

Additional information

FS_FILETIME represents a timestamp using individual members for the month, day, year, weekday, hour, minute, and second values. This can be useful for getting or setting a timestamp of a file or directory. The conversion between timestamp and FS_FILETIME can be done using FS_FileTimeToTimeStamp() and FS_TimeStampToFileTime()

4.10.52 FS_FREE_SPACE_DATA

Description

Information the number of free space available on a volume.

Type definition

```
typedef struct {
  U32         NumClustersFree;
  int         SizeOfBuffer;
  void      * pBuffer;
  FS_VOLUME * pVolume;
  U32         FirstClusterId;
} FS_FREE_SPACE_DATA;
```

Structure members

Member	Description
NumClustersFree	Number of unallocated clusters found.
SizeOfBuffer	Internal. Do not use. Number of bytes in the work buffer.
pBuffer	Internal. Do not use. Work buffer.
pVolume	Internal. Do not use. Volume information.
FirstClusterId	Internal. Do not use. Id of the first cluster to be checked.

Additional information

This structure stores the result and context of the operation that calculates the amount of free space available on a volume. The amount of free space is returned as a number of clusters via the NumClustersFree member. The members of the search context are considered internal and should not be used by the application. FS_FREE_SPACE_DATA is used by the FS_GetVolumeFreeSpaceFirst() FS_GetVolumeFreeSpaceNext() pair of API functions.

4.10.53 FS_MEM_CHECK_CALLBACK

Description

Type of function called by the file system to check if a memory region can be used in a 0-copy operation.

Type definition

```
typedef int FS_MEM_CHECK_CALLBACK(void * pMem,
                                  U32    NumBytes);
```

Parameters

Parameter	Description
pMem	Points to the first byte in the memory area to be checked.
NumBytes	Size of the memory area in bytes.

Return value

≠ 0 The memory region can be used in a 0 copy operation.
= 0 The memory region cannot be used in a 0 copy operation.

Additional information

A function of this type is called by the file system before any read or write operation performed by the application. The function has to check if the memory region defined by pMem and NumBytes can be passed directly to the device driver during a 0-copy operation.

The callback function is typically required on a system where the device driver uses DMA to transfer data to and from the storage device and where not all the memory regions can be accessed by the DMA. If the memory region cannot be accessed by DMA the callback function has to return 0. The file system copies then the data to an internal buffer that is accessible to DMA and performs the data transfer. The callback has to return a value different than 0 if the DMA can access the specified memory region. In this case the memory region is passed directly to device driver to perform the data transfer and the internal copy operation of the file system is skipped.

4.10.54 FS_MEM_INFO

Description

Information about the memory management.

Type definition

```
typedef struct {
  U8   IsExternal;
  U32  NumBytesTotal;
  U32  NumBytesAllocated;
} FS_MEM_INFO;
```

Structure members

Member	Description
IsExternal	Memory allocator type.
NumBytesTotal	Size of the memory pool in bytes.
NumBytesAllocated	Number of bytes allocated from the memory pool.

Additional information

The information can be queried via FS_GetMemInfo().

IsExternal contains the value of the FS_SUPPORT_EXT_MEM_MANAGER configuration define.

If the file system is configured to use the internal memory allocator (FS_SUPPORT_EXT_MEM_MANAGER set to 0) then NumBytesTotal stores the value passed as second parameter to FS_AssignMemory(). NumBytesTotal is set to 0 if the file system is configured to use an external memory allocator (FS_SUPPORT_EXT_MEM_MANAGER set to 1) because this value is not known to the file system.

The value of NumBytesAllocated stores the number of bytes allocated by the file system at the time FS_GetMemInfo() is called. In emFile versions older than 5.xx this information was stored in the global variable FS_NumBytesAllocated.

4.10.55 FS_PARTITION_INFO

Description

Information about a MBR partition.

Type definition

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

Structure members

Member	Description
NumSectors	Total number of sectors in the partition.
StartSector	Index of the first sector in the partition relative to the beginning of the storage device.
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.

Example

For a sample usage refer to FS_GetPartitionInfo().

4.10.56 FS_TIME_DATE_CALLBACK

Description

Type of function called by the file system to get the actual time an date.

Type definition

```
typedef U32 FS_TIME_DATE_CALLBACK(void);
```

Return value

Current time and date in a format suitable for the file system.

Additional information

The time and date have to be encoded 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: Count of years from 1980 (0-127) The callback function can be registered via `FS_SetTimeDateCallback()`. `FS_X_GetTimeDate()` is set as default callback function at the file system initialization. The application has to provide the implementation of `FS_X_GetTimeDate()`.

Example

Refer to `FS_SetTimeDateCallback()` for a sample usage.

4.10.57 Volume information flags

Description

Flags that control the information returned by `FS_GetVolumeInfoEx()`.

Definition

```
#define FS_DISKINFO_FLAG_FREE_SPACE    0x01
```

Symbols

Definition	Description
FS_DISKIN-FO_FLAG_FREE_SPACE	Returns the available free space on the storage medium

4.11 Storage layer functions

The functions described in this section can be used to access and to manage the data at the logical sector level effectively bypassing the file system. These functions are typically used when the application comes with its own file system or when the storage device has to be mounted as mass storage device via USB.

4.11.1 FS_STORAGE_Clean()

Description

Performs garbage collection on a volume.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume on which to perform the garbage collection.

Return value

= 0 OK, volume cleaned.
≠ 0 Error code indicating the failure reason.

Additional information

The application can call this function to convert storage blocks that contain invalid data to free space that can be used to store new data. This operation is supported only by storage devices that are managed by the file system such as NAND and NOR flash.

FS_STORAGE_Clean() is optional since the device drivers perform the garbage collection operation automatically. The function can be used to increase the write performance by preparing the storage device in advance of the write operation.

The actions executed by FS_STORAGE_Clean() are device-driver-dependent. For example, the sector map NOR driver converts all logical blocks that contain invalid data into free (writable) logical blocks. As a consequence, the following write operation will run faster since no physical sector is required to be erased.

FS_STORAGE_Clean() can potentially take a long time to complete, preventing the access of other tasks to the file system. How long the execution takes depends on the type of storage device and on the number of storage blocks that contain invalid data. The file system provides an alternative function FS_STORAGE_CleanOne() that completes in a shorter period of time than FS_STORAGE_Clean(). This is realized by executing only one sub-operation of the entire garbage collection operation at a time. For more information refer to FS_STORAGE_CleanOne().

Additional information about the garbage collection operation can be found in the “Garbage collection” section of the NAND and NOR drivers of the emFile manual.

Example

```
#include "FS.h"

void SampleStorageClean(void) {
  FS_FILE * pFile;

  //
  // Perform garbage collection on the storage device.
  //
  FS_STORAGE_Clean("nor:0:");
  //
  // The write to file is fast since the NOR driver does not have to perform
  // any garbage collection during the write operation.
  //
  pFile = FS_FOpen("nor:0:\\File.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Test", 4);
    FS_FClose(pFile);
  }
```



```
}
```


4.11.2 FS_STORAGE_CleanOne()

Description

Performs garbage collection on a volume.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the storage volume on which to perform the garbage collection.
pMoreToClean	out Indicates if the storage device has been cleaned completely or not. It can be NULL. • ≠0 Not cleaned completely. • =0 Completely clean.

Return value

= 0 OK, clean operation executed.
≠ 0 Error code indicating the failure reason.

Additional information

This function performs the same operation as FS_STORAGE_Clean() with the difference that it executes only one sub-operation of the garbage collection operation at a time. This is practical if other tasks of the application require access to the file system before the garbage collection operation is completed. The completion of the garbage collection operation is indicated via pMoreToClean. FS_STORAGE_CleanOne() returns a value different than 0 via pMoreToClean if the operation is not completed.

Additional information about the garbage collection operation can be found in the "Garbage collection" section of the NAND and NOR drivers.

Example

```
#include "FS.h"

void SampleStorageCleanOne(void) {
  FS_FILE * pFile;
  int       MoreToClean;

  //
  // Perform garbage collection on the storage device.
  //
  MoreToClean = 0;
  do {
    FS_STORAGE_CleanOne("nand:0:", &MoreToClean);
  } while (MoreToClean);
  //
  // The write to file is fast since the NAND driver does not have to perform
  // any garbage collection during the write operation.
  //
  pFile = FS_FOpen("nand:0:\\File.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Test", 4);
    FS_FClose(pFile);
  }
}
```


4.11.3 FS_STORAGE_DeInit()

Description

Frees the resources allocated by the storage layer.

Prototype

```
void FS_STORAGE_DeInit(void);
```

Additional information

This function is optional. `FS_STORAGE_DeInit()` frees all resources that are allocated by the storage layer after initialization. The application can call this function only after it called `FS_STORAGE_Init()`.

This function is available if the emFile sources are compiled with the `FS_SUPPORT_DEINIT` configuration define set to 1.

Example

```
#include "FS.h"

void SampleStorageDeInit(void) {
  FS_STORAGE_Init();
#if FS_SUPPORT_DEINIT
  //
  // Access the storage layer...
  //
  FS_STORAGE_DeInit();
#endif // FS_SUPPORT_DEINIT
  //
  // The storage layer cannot be accessed anymore.
  //
}
```


4.11.4 FS_STORAGE_FreeSectors()

Description

Informs the driver about unused sectors.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume on which to perform the operation.
FirstSector	Index of the first logical sector to be marked as invalid (0-based).
NumSectors	Number of sectors to be marked as invalid starting from FirstSector.

Return value

= 0 OK, sectors freed.
≠ 0 Error code indicating the failure reason

Additional information

Typically, this function is called by the application to inform the driver which logical sectors are no longer used for data storage. The NAND and NOR drivers can use this information to optimize the internal relocation of data blocks during the wear-leveling operation. The data of the logical sectors marked as not in use is not copied anymore that can typically lead to an improvement of the write performance.

FS_STORAGE_FreeSectors() performs a similar operation as the trim command of SSDs (Solid-State Drives). The data stored to a logical sector is no longer available to an application after the logical sector has been freed via FS_STORAGE_FreeSectors(),

Example

```
#include "FS.h"

void SampleStorageFreeSectors(void) {
  //
  // Inform the NAND driver that the logical sectors
  // with the indexes 2, 3, and 4 are no longer in use.
  //
  FS_STORAGE_FreeSectors("nand:0:", 2, 3);
}
```


4.11.5 FS_STORAGE_GetCleanCnt()

Description

Calculates the number of garbage collection sub-operations.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to be queried.
pCleanCnt	out Number of sub-operations left.

Return value

= 0 OK, clean count returned.
≠ 0 Error code indicating the failure reason.

Additional information

FS_STORAGE_GetCleanCnt() calculates and returns the number of sub-operations that the application has to perform until the garbage collection operation is completed. The value returned via pCleanCnt is the number of times FS_STORAGE_CleanOne() has to be called to complete the garbage collection. FS_STORAGE_GetCleanCnt() is supported only for volumes mounted on a storage device managed by emFile such as NAND or NOR flash.

Example

```
#include "FS.h"

void SampleStorageGetCleanCnt(void) {
  U32 CleanCnt;

  CleanCnt = 0;
  FS_STORAGE_GetCleanCnt("nor:0:", &CleanCnt);
  if (CleanCnt) {
    do {
      FS_STORAGE_CleanOne("nor:0:", NULL);
    } while (--CleanCnt);
  }
}
```


4.11.6 FS_STORAGE_GetCounters()

Description

Returns the values of statistical counters.

Prototype

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

Parameters

Parameter	Description
pStat	out Current values of statistical counters.

Additional information

This function returns the values of the counters that indicate how many operations the storage layer executed since the file system initialization or the last call to FS_STORAGE_ResetCounters().

The statistical counters are updated only on debug builds if the file system sources are compiled with FS_DEBUG_LEVEL greater then or equal to FS_DEBUG_LEVEL_CHECK_PARA or FS_STORAGE_ENABLE_STAT_COUNTERS set to 1.

Example

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

void SampleStorageGetCounters(void) {
  FS_STORAGE_COUNTERS StatCnt;
  char                ac[100];

  FS_STORAGE_GetCounters(&StatCnt);
  sprintf(ac, "Num. read operations:                %lu\n", StatCnt.ReadOperationCnt);
  FS_X_Log(ac);
  sprintf(ac, "Num. sectors read:                   %lu\n", StatCnt.ReadSectorCnt);
  FS_X_Log(ac);
  sprintf(ac, "Num. sectors read (from cache):      %lu\n", StatCnt.ReadSectorCachedCnt);
  FS_X_Log(ac);
  sprintf(ac, "Num. sectors read (alloc. table):    %lu\n", StatCnt.ReadSectorCntMan);
  FS_X_Log(ac);
  sprintf(ac, "Num. sectors read (dir. entry):      %lu\n", StatCnt.ReadSectorCntDir);
  FS_X_Log(ac);
  sprintf(ac, "Num. write operations:               %lu\n", StatCnt.WriteOperationCnt);
  FS_X_Log(ac);
  sprintf(ac, "Num. sectors written:                %lu\n", StatCnt.WriteSectorCnt);
  FS_X_Log(ac);
  sprintf(ac, "Num. sectors written (by cache):     %lu\n", StatCnt.WriteSectorCntCleaned);
  FS_X_Log(ac);
  sprintf(ac, "Num. sectors written (alloc. table): %lu\n", StatCnt.WriteSectorCntMan);
  FS_X_Log(ac);
  sprintf(ac, "Num. sectors written (dir. entry):   %lu\n", StatCnt.WriteSectorCntDir);
  FS_X_Log(ac);
}
```


4.11.7 FS_STORAGE_GetDeviceInfo()

Description

Returns information about the storage device.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to be queried. It cannot be NULL.
pDeviceInfo	out Information about the storage device. It cannot be NULL.

Return value

= 0 O.K., information returned.
≠ 0 Error code indicating the failure reason.

Additional information

This function returns information about the logical organization of the storage device such as the number of logical sectors and the size of the logical sector supported. FS_STORAGE_GetDeviceInfo() requests the information directly from the device driver.

Obsolete name

FS_GetDeviceInfo

Example

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

void SampleStorageGetDeviceInfo(void) {
  FS_DEV_INFO DeviceInfo;
  char        ac[100];

  FS_STORAGE_GetDeviceInfo("", &DeviceInfo);
  sprintf(ac, "Total num. sectors:      %lu\n", DeviceInfo.NumSectors);
  FS_X_Log(ac);
  sprintf(ac, "Num. bytes per sector:   %d\n", DeviceInfo.BytesPerSector);
  FS_X_Log(ac);
  sprintf(ac, "Num. read / write heads: %d\n", DeviceInfo.NumHeads);
  FS_X_Log(ac);
  sprintf(ac, "Num. sectors per track:  %d\n", DeviceInfo.SectorsPerTrack);
  FS_X_Log(ac);
}
```


4.11.8 FS_STORAGE_GetSectorUsage()

Description

Returns information about the usage of a logical sector.

Prototype

```
int FS_STORAGE_GetSectorUsage(const char * sVolumeName,
                              U32    SectorIndex);
```

Parameters

Parameter	Description
sVolumeName	Name of the volume.
SectorIndex	Index of the sector to be queried.

Return value

= FS_SECTOR_IN_USE	The sector contains valid data.
= FS_SECTOR_NOT_USED	The sector contains invalid data.
= FS_SECTOR_USAGE_UNKNOWN	The usage of the sector is unknown or the operation is not supported.
< 0	Error code indicating the failure reason.

Additional information

After a low-level format all the logical sectors contain invalid information. The data of a logical sector becomes valid after the application performs a write operation to that sector. The sector data can be explicitly invalidated by calling FS_STORAGE_FreeSectors().

Example

```
#include "FS.h"

void SampleStorageGetSectorUsage(void) {
  int SectorUsage;

  SectorUsage = FS_STORAGE_GetSectorUsage("", 7);
  switch (SectorUsage) {
  case FS_SECTOR_IN_USE:
    FS_X_Log("The sector is used to store data.\n");
    break;
  case FS_SECTOR_NOT_USED:
    FS_X_Log("The sector is not used to store data.\n");
    break;
  case FS_SECTOR_USAGE_UNKNOWN:
    FS_X_Log("The sector usage is unknown.\n");
    break;
  default:
    FS_X_Log("An error occurred during the operation.\n");
  }
}
```


4.11.9 FS_STORAGE_Init()

Description

Initializes the storage layer.

Prototype

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

Return value

Number of OS synchronization objects required to protect the file system against concurrent access from different tasks.

Additional information

This function initializes the only drivers and if necessary the OS layer. It has to be called before any other function of the storage layer (`FS_STORAGE_…`) The storage layer allows an application to access the file system at logical sector level. The storage device is presented as an array of logical sector that can be accessed via a 0-based index. This can be useful when using the file system as USB mass storage client driver.

`FS_STORAGE_Init()` is called internally at the initialization of the file system. The return value of this function is used by `FS_Init()` to calculate the number of internal buffers the file system has to allocate for the read and write operations. The application is not required to call `FS_STORAGE_Init()` if it already calls `FS_Init()`.

`FS_STORAGE_DeInit()` is the counterpart of `FS_STORAGE_Init()` that can be used to free the resources allocated by the drivers and if enabled of the OS layer.

Obsolete name

`FS_InitStorage`

Example

```
#include "FS.h"

void SampleStorageInit(void) {
  FS_STORAGE_Init();
  //
  // Access logical sectors...
  //
}
```


4.11.10 FS_STORAGE_ReadSector()

Description

Reads the data of one logical sector.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to read from. It cannot be NULL.
pData	out Buffer that receives the read sector data. It cannot be NULL.
SectorIndex	Index of the sector to read from.

Return value

= 0 O.K., sector data read.
≠ 0 Error code indicating the failure reason.

Additional information

pData has to point to a memory area large enough to store the contents of one logical sector. The size of the logical sector is driver-dependent and typically 512 bytes in size. SectorIndex is a 0-based index that specifies the logical sector to be read. The size of the logical sector and the number of logical sectors in a storage device can be determined via FS_STORAGE_GetDeviceInfo().

FS_STORAGE_ReadSector() reports an error and does not store any data to pData if SectorIndex is out of bounds.

The application can call FS_STORAGE_ReadSectors() instead of calling FS_STORAGE_ReadSector() multiple times if it has to read consecutive logical sectors at once.

Obsolete name

FS_ReadSector

Example

```
#include "FS.h"

void SampleStorageReadSector(void) {
  U32 aSectorData[512 / 4];

  //
  // Read the data of the logical sector with the index 10.
  //
  FS_STORAGE_ReadSector("", aSectorData, 10);
}
```


4.11.11 FS_STORAGE_ReadSectors()

Description

Reads the data of one or more logical sectors.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to read from. It cannot be NULL.
pData	out Buffer that receives the read sector data. It cannot be NULL.
FirstSector	Index of the first sector to read from.
NumSectors	Number of sectors to be read.

Return value

= 0 O.K., sector data read.
≠ 0 Error code indicating the failure reason.

Additional information

This function can be used to read the contents of multiple consecutive logical sectors. pData has to point to a memory area large enough to store the contents of all the logical sectors read. The size of the logical sector is driver-dependent typically 512 bytes in size. FirstSector is a 0-based index that specifies the index of the first logical sector to be read. The size of the logical sector and the number of logical sectors in a storage device can be determined via FS_STORAGE_GetDeviceInfo().

FS_STORAGE_ReadSectors() reports an error and does not store any data to pData if any of the indexes of the specified logical sectors is out of bounds.

Example

```
#include "FS.h"

void SampleStorageReadSectors(void) {
  U32 aSectorData[512 * 2 / 4];

  //
  // Read the data of the logical sectors with the indexes 10 and 11.
  //
  FS_STORAGE_ReadSectors("", aSectorData, 10, 2);
}
```


4.11.12 FS_STORAGE_RefreshSectors()

Description

Reads the contents of a logical sector and writes it back.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume on which to perform the operation. It cannot be NULL.
FirstSector	Index of the first sector to refresh (0-based).
NumSectors	Number of sectors to refresh starting from FirstSector.
pBuffer	Temporary storage for the sector data. Must be at least one sector large. It cannot be NULL.
NumBytes	Number of bytes in pBuffer.

Return value

= 0 OK, sectors refreshed.
≠ 0 Error code indicating the failure reason

Additional information

This function reads the contents of each specified logical sector to pBuffer and then it writes the same data to it. FS_STORAGE_RefreshSectors() can read and write more than one logical sector at once if the size of pBuffer allows it. The larger pBuffer the faster runs the refresh operation.

FS_STORAGE_RefreshSectors() function can be used on volumes mounted on a NAND flash device to prevent the accumulation of bit errors due to excessive read operations (read disturb effect). The function can also be used to prevent data loses caused by the data reaching the retention limit. Reading and then writing back the contents of a logical sector causes the NAND driver to relocate the data on the NAND flash device that in turn eliminates bit errors. Typically, the refresh operation has to be performed periodically at large time intervals (weeks). The NAND flash may wear out too soon if the refresh operation is performed too often.

Example

```
#include "FS.h"

static U32 _aBuffer[2048 * 2 / 4];

void SampleStorageRefreshSectors(void) {

  //
  // Refresh the logical sectors with the indexes 10-99
  // stored on the NAND flash.
  //
  FS_STORAGE_RefreshSectors("nand:0:", 10, 100, _aBuffer, sizeof(_aBuffer));
}
```


4.11.13 FS_STORAGE_ResetCounters()

Description

Sets all statistical counters to 0.

Prototype

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

Additional information

This function can be used to set to 0 all the statistical counters maintained by the storage layer. This can be useful for example in finding out how many sector operations are performed during a specific file system operation. The application calls `FS_STORAGE_ResetCounters()` before the file system operation and then `FS_STORAGE_GetCounters()` at the end.

The statistical counters are available only on debug builds if the file system sources are compiled with `FS_DEBUG_LEVEL` greater then or equal to `FS_DEBUG_LEVEL_CHECK_PARA` or `FS_STORAGE_ENABLE_STAT_COUNTERS` set to 1.

Example

```
#include "FS.h"

void SampleStorageResetCounters(void) {
  FS_STORAGE_COUNTERS StatCnt;

  //
  // Set all statistical counters to 0.
  //
  FS_STORAGE_ResetCounters();

  //
  // Perform the file system operation...
  //

  //
  // Check the statistical counters.
  //
  FS_STORAGE_GetCounters(&StatCnt);
}
```


4.11.14 FS_STORAGE_SetOnDeviceActivityCallback()

Description

Registers a function to be called on any logical sector read or write operation.

Prototype

```
void FS_STORAGE_SetOnDeviceActivityCallback
                      (const char                          * sVolumeName,
                             FS_ON_DEVICE_ACTIVITY_CALLBACK * pfOnDeviceActivity);
```

Parameters

Parameter	Description
sVolumeName	Name of the volume for which the callback function is registered. It cannot be NULL.
pfOnDeviceActivity	Function to be invoked.

Additional information

This function is optional. It is available only when the file system is built with FS_DEBUG_LEVEL set to a value grater than or equal to FS_DEBUG_LEVEL_CHECK_PARA or with FS_STORAGE_SUPPORT_DEVICE_ACTIVITY set to 1.

Example

Refer to the sample application FS_DeviceActivity.c located in the Sample/FS/Application folder of the emFile shipment.

4.11.15 FS_STORAGE_Sync()

Description

Writes cached information to volume.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to be synchronized.

Additional information

This function updates all the information present only in sector cache (if enabled) to storage device. It is also requests the driver to perform a synchronization operation. The operations performed during the synchronization are driver-dependent.

Typically, FS_STORAGE_Sync() has to be called if a write-back sector cache is configured for the volume to reduce the chance of a data loss in case of an unexpected reset.

Obsolete name

FS_CleanVolume

Example

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

void SampleStorageSync(void) {
  U32 aSectorData[512 / 4];

  memset(aSectorData, 'a', sizeof(aSectorData));
  //
  // Write some data to sector with the index 0.
  //
  FS_STORAGE_WriteSector("", aSectorData, 0);
  //
  // Make sure that the data is written to storage device.
  //
  FS_STORAGE_Sync("");
}
```


4.11.16 FS_STORAGE_SyncSectors()

Description

Synchronize the contents of one or more logical sectors.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume.
FirstSector	Index of the first sector to be synchronized (0-based).
NumSectors	Number of sectors to be synchronized starting from FirstSector.

Return value

= 0 OK, sectors synchronized.
≠ 0 Error code indicating the failure reason.

Additional information

This operation is driver-dependent and is currently supported only by the RAID1 logical driver. The operation updates the contents of the specified logical sectors that are located on the secondary storage (mirrored) with the contents of the corresponding logical sectors from the primary storage (master). RAID1 logical driver updates the logical sectors only if the contents is different.

Example

```
#include "FS.h"

void SampleStorageSyncSectors(void) {
  //
  // Synchronizes the logical sectors with the indexes 7, 8 and 9.
  //
  FS_STORAGE_SyncSectors("", 7, 3);
}
```


4.11.17 FS_STORAGE_Unmount()

Description

Synchronizes a volume and marks it as not initialized.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to be unmounted. It cannot be NULL.

Additional information

The function sends an unmount command to the driver and marks the volume as unmounted and uninitialized. If a write sector cache is enabled, FS_STORAGE_Unmount() also stores any modified data from sector cache to storage device. This function has to be called before the device is shutdown to prevent a data loss.

The file system mounts automatically the volume at the call to an API function of the storage layer.

Obsolete name

FS_UnmountLL

Example

```
#include "FS.h"

void SampleStorageUnmount(void) {
  //
  // Synchronize the sector cache and mark the volume as not initialized.
  //
  FS_STORAGE_Unmount("");
  //
  // The device can be shutdown here without loosing any data.
  //
}
```


4.11.18 FS_STORAGE_UnmountForced()

Description

Marks a volume it as not initialized.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to be unmounted. It cannot be NULL.

Additional information

This function performs the same operations as FS_STORAGE_Unmount(). FS_STORAGE_UnmountForced() has to be called if a storage device has been removed before being regularly unmounted. When using FS_STORAGE_UnmountForced() there is no guarantee that the information cached by the file system is updated to storage.

The file system mounts automatically the volume at the call to an API function of the storage layer.

Example

```
#include "FS.h"

void SampleStorageUnmountForced(void) {
  if (FS_GetVolumeStatus("") == FS_MEDIA_IS_PRESENT) {
    //
    // Synchronize the sector cache and mark the volume as not initialized.
    //
    FS_STORAGE_Unmount("");
    //
    // The device can be shutdown here without loosing any data.
    //
  } else {
    //
    // Storage device has been removed. Mark the volume as not initialized.
    // Data loss is possible.
    //
    FS_STORAGE_UnmountForced("");
  }
}
```


4.11.19 FS_STORAGE_WriteSector()

Description

Modifies the data of a logical sector.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to write to. It cannot be NULL.
pData	in Buffer that contains the sector data to be written. It cannot be NULL.
SectorIndex	Index of the sector to write to.

Return value

= 0 O.K., sector data modified.
≠ 0 Error code indicating the failure reason.

Additional information

pData has to point to a memory area that stores the contents of one logical sector. The size of the logical sector is driver-dependent and typically 512 bytes in size. SectorIndex is a 0-based index that specifies the logical sector to be written. The size of the logical sector and the number of logical sectors in a storage device can be determined via FS_STORAGE_GetDeviceInfo().

FS_STORAGE_WriteSector() reports an error and does modify the contents of the logical sector if SectorIndex is out of bounds.

The application can call FS_STORAGE_WriteSectors() instead of calling FS_STORAGE_WriteSector() multiple times if it has to write consecutive logical sectors at once.

Obsolete name

FS_WriteSector

Example

```
#include <string.h>
#include "FS_Storage.h"

void SampleStorageWriteSector(void) {
  U32 aSectorData[512 / 4];

  memset(aSectorData, 'a', sizeof(aSectorData));
  //
  // Write the data of the logical sector with the index 7.
  //
  FS_STORAGE_WriteSector("", aSectorData, 7);
}
```


4.11.20 FS_STORAGE_WriteSectors()

Description

Modifies the data of one or more logical sector.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to write to. It cannot be NULL.
pData	in Buffer that contains the sector data to be written. It cannot be NULL.
FirstSector	Index of the first sector to read from.
NumSectors	Number of sectors to be read.

Return value

= 0 O.K., sector data modified.
≠ 0 Error code indicating the failure reason.

Additional information

This function can be used to write the contents of multiple consecutive logical sectors. pData has to point to a memory area that store the contents of all the logical sectors to be written. The size of the logical sector is driver-dependent and typically 512 bytes in size. FirstSector is a 0-based index that specifies the index of the first logical sector to be written. The size of the logical sector and the number of logical sectors in a storage device can be determined via FS_STORAGE_GetDeviceInfo().

FS_STORAGE_WriteSectors() reports an error and does not modify the contents of the logical sectors if any of the indexes of the specified logical sectors is out of bounds.

Example

```
#include "FS.h"

void SampleStorageWriteSectors(void) {
  U32 aSectorData[512 / 4 * 2];

  //
  // Write the data of the logical sectors with the indexes 7 and 8.
  //
  FS_STORAGE_WriteSectors("", aSectorData, 7, 2);
}
```


4.11.21 FS_DEV_INFO

Description

Information about the storage device.

Type definition

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

Structure members

Member	Description
NumHeads	Number of read / write heads.
SectorsPerTrack	Number of sectors stored on a track.
NumSectors	Total number of sectors on the storage device.
BytesPerSector	Size of a logical sector in bytes.

Additional information

NumHeads and SectorsPerTrack are relevant only for mechanical drives. The application can access the information about the storage device by calling FS_STORAGE_GetDeviceInfo().

4.11.22 FS_ON_DEVICE_ACTIVITY_CALLBACK

Description

The type of the callback function invoked by the file system on a logical sector read or write operation.

Type definition

```
typedef void (FS_ON_DEVICE_ACTIVITY_CALLBACK)(FS_DEVICE * pDevice,
                                               unsigned    Operation,
                                               U32         StartSector,
                                               U32         NumSectors,
                                               int         Sectortype);
```

Parameters

Parameter	Description
pDevice	Storage device that performed the operation.
Operation	Type of the operation performed. Can be one of the values listed in *Transfer direction* on page 249
StartSector	Index of the first logical sector transfered (0-based)
NumSectors	Number of logical sectors transfered in the current operation.
SectorType	Type of the data stored in the logical sector. It can be one of the values listed in *Sector data type* on page 248

Additional information

This is the type of function that can be registered via `FS_STORAGE_SetOnDeviceActivityCallback()`

4.11.23 FS_STORAGE_COUNTERS

Description

Statistical counters.

Type definition

```
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;
```

Structure members

Member	Description
ReadOperationCnt	Number of “Read sector operation” calls.
ReadSectorCnt	Number of sectors read (before cache).
ReadSectorCachedCnt	Number of sectors read from cache
WriteOperationCnt	Number of “Write sector operation” calls
WriteSectorCnt	Number of sectors written (before cache).
WriteSectorCntCleaned	Number of sectors written by the cache 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).

Additional information

ReadSectorCnt can be (and typically is) higher than ReadOperationCnt, since one read operation can request multiple sectors (in a burst). The same applies to write operations: WriteSectorCnt can be (and typically is) higher than WriteOperationCnt, since one read operation can request multiple sectors (in a burst).

The statistical counters can be read via FS_STORAGE_GetCounters(). They are set to 0 when the file system is initialized. Additionally, the application can explicitly set them to 0 via FS_STORAGE_ResetCounters().

4.11.24 Sector data type

Description

Type of data stored in a logical sector

Definition

```
#define FS_SECTOR_TYPE_DATA    0u
#define FS_SECTOR_TYPE_DIR     1u
#define FS_SECTOR_TYPE_MAN     2u
```

Symbols

Definition	Description
FS_SECTOR_TYPE_DATA	Sector that stores file data.
FS_SECTOR_TYPE_DIR	Sector that stores directory entries.
FS_SECTOR_TYPE_MAN	Sector that stores entries of the allocation table.

4.11.25 Transfer direction

Description

Direction of the transfered data.

Definition

```
#define FS_OPERATION_READ     0
#define FS_OPERATION_WRITE    1
```

Symbols

Definition	Description
FS_OPERATION_READ	Data is transferred from storage device to MCU.
FS_OPERATION_WRITE	Data is transferred from MCU to storage device.

Additional information

One of these values is passed by the file system via Operation parameter to the callback function registered via FS_STORAGE_SetOnDeviceActivityCallback()

4.12 FAT related functions

The functions described in this section can be used only on volumes formatted as FAT.

4.12.1 FS_FAT_ConfigDirtyFlagUpdate()

Description

Enables / disables the update of the flag that indicates if the volume has been unmounted correctly.

Prototype

```
void FS_FAT_ConfigDirtyFlagUpdate(int OnOff);
```

Parameters

Parameter	Description
OnOff	Activation status of the option. • 0 Disable the option. • 1 Enable the option.

Additional information

If enabled, the file system updates an internal dirty flag that is set to 1 each time data is written to storage device. The dirty flag is set to 0 when the application unmounts the file system. The value of the dirty flag is updated to storage device and can be used to check if the storage device has been properly unmounted before the system reset. FS_GetVolumeInfo() can be used to get the value of this dirty flag (IsDirty member of FS_DISK_INFO).

The update of the dirty flag is enabled by default if the compile-time option FS_FAT_UPDATE_DIRTY_FLAG is set to 1. FS_FAT_ConfigDirtyFlagUpdate() is available only if the compile-time option FS_FAT_UPDATE_DIRTY_FLAG is set to 1.

Example

```
#include "FS.h"

void SampleFATConfigDirtyFlagUpdate(void) {
  FS_FILE      * pFile;
  FS_DISK_INFO   VolumeInfo;

  FS_Init();
#if FS_FAT_UPDATE_DIRTY_FLAG
  FS_FAT_ConfigDirtyFlagUpdate(1);
#endif
  pFile = FS_FOpen("Test.txt", "w");
  if (pFile) {
    FS_FClose(pFile);
  }
  FS_GetVolumeInfo("", &VolumeInfo);
  if (VolumeInfo.IsDirty) {
    FS_X_Log("Volume has been modified.\n");
  } else {
    FS_X_Log("Volume has not been modified.\n");
  }
}
```


4.12.2 FS_FAT_ConfigFATCopyMaintenance()

Description

Enables / disables the update of the second allocation table.

Prototype

```
void FS_FAT_ConfigFATCopyMaintenance(int OnOff);
```

Parameters

Parameter	Description
OnOff	Activation status of the option. • 0 Disable the option. • 1 Enable the option.

Additional information

The FAT file system has support for a second (redundant) allocation table. FS_FAT_ConfigFATCopyMaintenance() can be used to enable or disable the update of the second allocation table. The data in the second allocation table is not used by the file system but it may be required to be present by some PC file system checking utilities. Enabling this option can possible reduce the write performance of the file system.

The update of the second allocation table is enabled by default if the compile-time option FS_MAINTAIN_FAT_COPY is set to 1. FS_FAT_ConfigFATCopyMaintenance() is available only if the compile-time option FS_MAINTAIN_FAT_COPY is set to 1.

Example

```
#include "FS.h"

void SampleFATConfigFATCopyMaintenance(void) {
  FS_Init();
#if FS_MAINTAIN_FAT_COPY
  FS_FAT_ConfigFATCopyMaintenance(1);
#endif
  //
  // At each write access to the allocation table
  // the second allocation table is also updated.
  //
}
```


4.12.3 FS_FAT_ConfigFSInfoSectorUse()

Description

Enables / disables the usage of information from the FSInfo sector.

Prototype

```
void FS_FAT_ConfigFSInfoSectorUse(int OnOff);
```

Parameters

Parameter	Description
OnOff	Activation status of the option. • 0 Disable the option. • 1 Enable the option.

Additional information

FSInfo sector is a management sector present on FAT32-formatted volumes that stores information about the number of free clusters and the id of the first free cluster. This information, when available and valid, can be used to increase the performance of the operation that calculates the available free space on a volume such as FS_GetVolumeFreeSpace(), FS_GetVolumeFreeSpaceKB(), FS_GetVolumeInfo(), or FS_GetVolumeInfoEx(). If the information in the FSInfo sector is missing or invalid, the file system has to scan the entire allocation to calculate the available free space an operation that can take a long time to complete on storage devices with a large capacity (few Gbytes.)

The file system invalidates the information in the FSInfo sector on the first operation that allocates or frees a cluster. The FSInfo sector is updated when the volume is unmounted via FS_Unmount() or synchronized via FS_Sync().

FSInfo sector is evaluated and updated by default if the compile-time option FS_FAT_USE_FSINFO_SECTOR is set to 1. FS_FAT_ConfigFSInfoSectorUse() is available only if the compile-time option FS_FAT_USE_FSINFO_SECTOR is set to 1.

Example

```
#include "FS.h"

void SampleFATConfigFSInfoSectorUse(void) {
  U32 NumBytes;

  FS_Init();
#if FS_FAT_USE_FSINFO_SECTOR
  FS_FAT_ConfigFSInfoSectorUse(1);
#endif
  NumBytes = FS_GetVolumeFreeSpace("");
}
```


4.12.4 FS_FAT_ConfigROFileMovePermission()

Description

Enables / disables the permission to move (and rename) files and directories with the read-only file attribute set.

Prototype

```
void FS_FAT_ConfigROFileMovePermission(int OnOff);
```

Parameters

Parameter	Description
OnOff	Activation status of the option. • 0 Disable the option. • 1 Enable the option.

Additional information

The application is per default allowed to move or rename via FS_Move() and FS_Rename() respectively files and directories that have the read-only file attribute (FS_ATTR_READ_ONLY) set. FS_FAT_ConfigROFileMovePermission() can be used to disable this option and thus to prevent an application to perform move or rename operations on files and directories marked as read-only.

FS_FAT_ConfigROFileMovePermission() is available only if the compile-time option FS_FAT_PERMIT_RO_FILE_MOVE is set to 1.

Example

```
#include "FS.h"

void SampleFATConfigROFileMovePermission(void) {
  FS_FILE * pFile;
  int       r;

#if FS_FAT_PERMIT_RO_FILE_MOVE
  FS_FAT_ConfigROFileMovePermission(0);
#endif
  pFile = FS_FOpen("Test.txt", "w");
  if (pFile != NULL) {
    FS_FClose(pFile);
  }
  FS_SetFileAttributes("Test.txt", FS_ATTR_READ_ONLY);
  r = FS_Move("Test.txt", "TestOld.txt");
  if (r != 0) {
    //
    // A file marked as read-only cannot be moved to another location.
    //
  }
}
```


4.12.5 FS_FAT_DisableLFN()

Description

Disables the support for long file names.

Prototype

```
void FS_FAT_DisableLFN(void);
```

Additional information

After calling this function the file system accepts only file and directory names in 8.3 format. Files and directories created with support for long file names enabled are still accessible since each long file name has an associated name in 8.3 format. The short name is automatically generated by the file system based on the first characters of the long name and a sequential index. The support for long file names can be activated via `FS_FAT_SupportLFN()`.

This function applies only to volumes formatted as FAT. EFS-formatted volumes have native support for long file names.

Example

```
#include "FS.h"

void SampleFATDisableLFN(void) {
  FS_FILE * pFile;

  FS_Init();
#if FS_SUPPORT_FAT
  FS_FAT_DisableLFN();
#endif
  pFile = FS_FOpen("FileName.txt", "w");
  if (pFile) {
    FS_FClose(pFile);
  }
}
```


4.12.6 FS_FAT_FormatSD()

Description

Formats the volume according to specification of SD Association.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume to be formatted.

Return value

= 0 O.K., format successful.
≠ 0 Error code indicating the failure reason.

Additional information

The SD Association defines the layout of the information that has to be stored to an SD, SDHC or SDXC card during the FAT format operation to ensure the best read and write performance by taking advantage of the physical structure of the storage device. FS_FAT_FormatSD() implements this recommended layout and it shall be used to format SD and MMC storage devices but it can be used for oder storage devices as well. It typically reserves more space for the file system as FS_Format() and as a consequence less space is available for the application to store files and directories.

FS_FAT_FormatSD() performs the following steps:

- Writes partition entry into the MBR.
- Formats the storage device as FAT.

The function is available only if the file system sources are compiled with the FS_SUPPORT_FAT option define set to 1.

Obsolete name

FS_FormatSD

Example

```
#include "FS.h"

void SampleFATFormatSD(void) {
  FS_Init();
#if FS_SUPPORT_FAT
  FS_FAT_FormatSD("");
#endif
}
```


4.12.7 FS_FAT_GrowRootDir()

Description

Increases the size of the root directory.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume for which to increase the root directory.
NumAddEntries	Number of directory entries to be added.

Return value

> 0	Number of entries added.
= 0	Clusters after root directory are not free. The number of entries in the root directory has not been changed.
= 0xFFFFFFFF	An error occurred.

Additional information

The formatting function allocates per default one cluster for the root directory of a FAT32 formatted volume. The file system increases automatically the size of the root directory as more files are added to it. This operation has a certain overhead that depends on the size of the allocation table and on the available free space. This overhead can be eliminated by calling FS_FAT_GrowRootDir() to increase the size of the root directory to the number of files and directories the application is expected to store in it.

This function increases the size of the root directory on a FAT32 formatted volume by the number of entries specified in NumAddEntries. The file system allocates one directory entry for each file or directory if the support for long file names is not enabled. With the support for long file names enabled the number of directory entries allocated to a file or directory depends on the number of characters in the name of the created file or directory.

This function shall be called after formatting the volume. The function fails with an error if:

- is not called after format operation.
- the specified volume is formatted as FAT12 or FAT16.
- the required number of clusters cannot be allocated immediately after the cluster already allocated to the root directory.

FS_FAT_GrowRootDir() is available only if the compile-time option FS_SUPPORT_FAT set to 1.

Example

```
#include "FS.h"

void SampleFATGrowRootDir(void) {
  FS_Format("", NULL);
#if FS_SUPPORT_FAT
  FS_FAT_GrowRootDir("", 64);
#endif
}
```


4.12.8 FS_FAT_SetLFNConverter()

Description

Configures how long file names are to be encoded and decoded.

Prototype

```
void FS_FAT_SetLFNConverter(const FS_UNICODE_CONV * pUnicodeConv);
```

Parameters

Parameter	Description
pUnicodeConv	File name converter.

Additional information

This function is available only if FS_SUPPORT_FILE_NAME_ENCODING is set to 1 which is the default.

Permitted values for pUnicodeConv are:

Identifier	Description
FS_UNICODE_CONV_CP437	Unicode <-> CP437 (DOS Latin US) converter
FS_UNICODE_CONV_CP932	Unicode <-> CP932 (Shift JIS) converter
FS_UNICODE_CONV_UTF8	Unicode <-> UTF-8 converter

4.12.9 FS_FAT_SupportLFN()

Description

Enables the support for long file names.

Prototype

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

Additional information

The file system accepts per default only file and directory names in 8.3 format, that is maximum 8 characters in the base name of a file, an optional period character, and an optional extension of maximum 3 characters. The application can call `FS_FAT_SupportLFN()` to enable the file system to work with names for files and directories longer that in the 8.3 format.

This function applies only to volumes formatted as FAT. EFS-formatted volumes have native support for long file names.

Example

```
#include "FS.h"

void SampleFATSupportLFN(void) {
  FS_FILE * pFile;

  FS_Init();
#if FS_SUPPORT_FAT
  FS_FAT_SupportLFN();
#endif
  pFile = FS_FOpen("This is a unusually long file name", "w");
  if (pFile) {
    FS_FClose(pFile);
  }
}
```


4.13 EFS related functions

The functions described in this section can be used only on volumes formatted as EFS.

4.13.1 FS_EFS_ConfigCaseSensitivity()

Description

Configures how the file names are compared.

Prototype

```
void FS_EFS_ConfigCaseSensitivity(int OnOff);
```

Parameters

Parameter	Description
OnOff	Specifies if the case of a character is important or not. • 1 Character case is important. • 0 Character case is ignored.

Additional information

This function is optional and is active only when the file system is compiled with FS_EFS_CASE_SENSITIVE set to 1. By default the EFS file system layer ignores the case of a character in file names when it compares two file names. That is, “testfile.txt”, “TestFile.txt” and “TESTFILE.TXT” represent the same file name. The case of a file name is preserved when the file is created via FS_FOpen() or FS_FOpenEx() or in case of a directory when FS_MkDir() and FS_CreateDir() is called to create one.

If the case sensitivity is enabled by calling FS_EFS_ConfigCaseSensitivity() with the OnOff parameter set to 1 than “testfile.txt”, “TestFile.txt” and “TESTFILE.TXT” are three different file names.

4.13.2 FS_EFS_ConfigStatusSectorSupport()

Description

Enables or disables the usage of information from the status sector.

Prototype

```
void FS_EFS_ConfigStatusSectorSupport(int OnOff);
```

Parameters

Parameter	Description
OnOff	Specifies if the status sector information has to be used or not. • 1 Status sector information is used. • 0 Status sector information is not used.

Additional information

This function is optional an is active only when the file system is compiled with FS_EFS_SUPPORT_STATUS_SECTOR set to 1. The status sector stores information about the free space available on the storage which helps the file system improve the performance of the operations that return the available free space such as FS_GetVolumeInfo().

4.13.3 FS_EFS_SetFileNameConverter()

Description

Configures how file names are to be encoded.

Prototype

```
void FS_EFS_SetFileNameConverter(const FS_UNICODE_CONV * pUnicodeConv);
```

Parameters

Parameter	Description
pUnicodeConv	File name converter.

Additional information

This function is available only if FS_SUPPORT_FILE_NAME_ENCODING is set to 1. By default, FS_SUPPORT_FILE_NAME_ENCODING is set to 0 in order to reduce the RAM and ROM usage.

Permitted values for pUnicodeConv are:

Identifier	Description
FS_UNICODE_CONV_CP437	Unicode <-> CP437 (DOS Latin US) converter
FS_UNICODE_CONV_CP932	Unicode <-> CP932 (Shift JIS) converter
FS_UNICODE_CONV_UTF8	Unicode <-> UTF-8 converter

4.14 Error-handling functions

An application can use the following functions to check and clear the error status of a file handle.

4.14.1 FS_ClearErr()

Description

Clears error status of a file handle.

Prototype

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

Parameters

Parameter	Description
pFile	Handle to opened file.

Additional information

The function sets the error status of a file handle to FS_ERRCODE_OK. FS_ClearErr() is the only function that clears the error status of a file handle. The other API functions modify the error status of a file handle only if it is cleared. The application has to call this function after it detects that an error occurred during a file system operation on an opened file handle and before it starts a new file system operation if it wants to get the correct error status in case of a failure.

Example

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

void SampleClearErr(void) {
  FS_FILE * pFile;
  U32       aBuffer[32 / 4];
  int       ErrCode;
  char      ac[100];
  U32       NumBytesRead;

  pFile = FS_FOpen("Test.txt", "r");
  if (pFile) {
    NumBytesRead = FS_Read(pFile, aBuffer, sizeof(aBuffer));
    ErrCode = FS_FError(pFile);
    if (ErrCode == FS_ERRCODE_OK) {
      sprintf(ac, "First read operation failed with: %d (%s)\n",
        ErrCode, FS_ErrorNo2Text(ErrCode));
      FS_X_Log(ac);
      FS_ClearErr(pFile);
    } else {
      sprintf(ac, "The first operation read %d bytes from file.\n", NumBytesRead);
      FS_X_Log(ac);
    }
    NumBytesRead = FS_Read(pFile, aBuffer, sizeof(aBuffer));
    ErrCode = FS_FError(pFile);
    if (ErrCode != FS_ERRCODE_OK) {
      sprintf(ac, "Second read operation failed with: %d (%s)\n",
        ErrCode, FS_ErrorNo2Text(ErrCode));
      FS_X_Log(ac);
      FS_ClearErr(pFile);
    } else {
      sprintf(ac, "The second operation read %d bytes from file.\n", NumBytesRead);
      FS_X_Log(ac);
    }
    FS_FClose(pFile);
  }
}
```


4.14.2 FS_ErrorNo2Text()

Description

Returns a human-readable text description of an API error code.

Prototype

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

Parameters

Parameter	Description
ErrCode	Error code for which the text description has to be returned.

Return value

The text description as 0-terminated ASCII string.

Additional information

For a list of supported error codes refer to *Error codes* on page 269. If the error code is not known FS_ErrorNo2Text() returns the "Unknown error" string. The error status of an opened file handle can be queried via FS_FError(). Most of the API functions of the file system also return one of the defined codes in case of an error. This error code can be passed to FS_ErrorNo2Text() to get a human-readable text description.

Example

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

void SampleErrorNo2Text(void) {
  FS_FILE * pFile;
  int       r;
  char      ac[100];

  r = FS_FOpenEx("Test.txt", "r", &pFile);
  if (r) {
    sprintf(ac, "The file open operation failed with: %d (%s)\n", r, FS_ErrorNo2Text(r));
    FS_X_Log(ac);
  } else {
    //
    // Access the file contents...
    //
    FS_FClose(pFile);
  }
}
```


4.14.3 FS_FEof()

Description

Returns if end of file has been reached.

Prototype

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

Parameters

Parameter	Description
pFile	Handle to opened file.

Return value

= 1 An attempt was made to read beyond the end of the file.
= 0 The end of file has not been reached.
< 0 An error occurred.

Additional information

The end-of-file flag of the file handle is set by the file system when the application tries to read more bytes than available in the file. This is not an error condition but just an indication that the file pointer is positioned beyond the last byte in the file and that by trying to read from this position no bytes are returned.

Example

```
#include "FS.h"

void SampleFEof(void) {
  FS_FILE * pFile;
  U32       aBuffer[32 / 4];
  int       r;
  int       ErrCode;

  pFile = FS_FOpen("Test.txt", "r");
  if (pFile) {
    while (1) {
      FS_Read(pFile, aBuffer, sizeof(aBuffer));
      if (FS_FEof(pFile)) {
        break;                  // End of file reached.
      }
      ErrCode = FS_FError(pFile);
      if (ErrCode) {
        break;                  // An error occurred while reading from file.
      }
    }
    FS_FClose(pFile);
  }
}
```


4.14.4 FS_FError()

Description

Return error status of a file handle.

Prototype

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

Parameters

Parameter	Description
pFile	Handle to opened file.

Return value

= FS_ERRCODE_OK No error present.
≠ FS_ERRCODE_OK An error has occurred.

Additional information

The application can use this function to check for example what kind of error caused the call to FS_Read(), FS_FRead(), FS_Write() or FS_FWrite() to fail. These functions do not return an error code but the number of byte read or written. The error status remains set until the application calls FS_ClearErr().

FS_ErrorNo2Text() can be used to return a human-readable description of the error as a 0-terminated string.

Example

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

void SampleFError(void) {
  FS_FILE * pFile;
  U32       aBuffer[32 / 4];
  int       ErrCode;
  char      ac[100];
  U32       NumBytesRead;

  pFile = FS_FOpen("Test.txt", "r");
  if (pFile) {
    NumBytesRead = FS_Read(pFile, aBuffer, sizeof(aBuffer));
    ErrCode = FS_FError(pFile);
    if (ErrCode == FS_ERRCODE_OK) {
      sprintf(ac, "The read operation failed with: %d (%s)\n",
        ErrCode, FS_ErrorNo2Text(ErrCode));
      FS_X_Log(ac);
      FS_ClearErr(pFile);
    } else {
      sprintf(ac, "The operation read %d bytes from file.\n", NumBytesRead);
      FS_X_Log(ac);
    }
    FS_FClose(pFile);
  }
}
```


4.14.5 Error codes

Description

Values returned by the API functions to indicate the reason of an error.

Definition

```
#define FS_ERRCODE_OK                          0
#define FS_ERRCODE_EOF                         (-1)
#define FS_ERRCODE_PATH_TOO_LONG               (-2)
#define FS_ERRCODE_INVALID_PARA                (-3)
#define FS_ERRCODE_WRITE_ONLY_FILE             (-5)
#define FS_ERRCODE_READ_ONLY_FILE              (-6)
#define FS_ERRCODE_READ_FAILURE                (-7)
#define FS_ERRCODE_WRITE_FAILURE               (-8)
#define FS_ERRCODE_FILE_IS_OPEN                (-9)
#define FS_ERRCODE_PATH_NOT_FOUND              (-10)
#define FS_ERRCODE_FILE_DIR_EXISTS             (-11)
#define FS_ERRCODE_NOT_A_FILE                  (-12)
#define FS_ERRCODE_TOO_MANY_FILES_OPEN         (-13)
#define FS_ERRCODE_INVALID_FILE_HANDLE         (-14)
#define FS_ERRCODE_VOLUME_NOT_FOUND            (-15)
#define FS_ERRCODE_READ_ONLY_VOLUME            (-16)
#define FS_ERRCODE_VOLUME_NOT_MOUNTED          (-17)
#define FS_ERRCODE_NOT_A_DIR                   (-18)
#define FS_ERRCODE_FILE_DIR_NOT_FOUND          (-19)
#define FS_ERRCODE_NOT_SUPPORTED               (-20)
#define FS_ERRCODE_CLUSTER_NOT_FREE            (-21)
#define FS_ERRCODE_INVALID_CLUSTER_CHAIN       (-22)
#define FS_ERRCODE_STORAGE_NOT_PRESENT         (-23)
#define FS_ERRCODE_BUFFER_NOT_AVAILABLE        (-24)
#define FS_ERRCODE_STORAGE_TOO_SMALL           (-25)
#define FS_ERRCODE_STORAGE_NOT_READY           (-26)
#define FS_ERRCODE_BUFFER_TOO_SMALL            (-27)
#define FS_ERRCODE_INVALID_FS_FORMAT           (-28)
#define FS_ERRCODE_INVALID_FS_TYPE             (-29)
#define FS_ERRCODE_FILENAME_TOO_LONG           (-30)
#define FS_ERRCODE_VERIFY_FAILURE              (-31)
#define FS_ERRCODE_VOLUME_FULL                 (-32)
#define FS_ERRCODE_DIR_NOT_EMPTY               (-33)
#define FS_ERRCODE_IOCTL_FAILURE               (-34)
#define FS_ERRCODE_INVALID_MBR                 (-35)
#define FS_ERRCODE_OUT_OF_MEMORY               (-36)
#define FS_ERRCODE_UNKNOWN_DEVICE              (-37)
#define FS_ERRCODE_ASSERT_FAILURE              (-38)
#define FS_ERRCODE_TOO_MANY_TRANSACTIONS_OPEN  (-39)
#define FS_ERRCODE_NO_OPEN_TRANSACTION         (-40)
#define FS_ERRCODE_INIT_FAILURE                (-41)
#define FS_ERRCODE_FILE_TOO_LARGE              (-42)
#define FS_ERRCODE_HW_LAYER_NOT_SET            (-43)
#define FS_ERRCODE_INVALID_USAGE               (-44)
#define FS_ERRCODE_TOO_MANY_INSTANCES          (-45)
#define FS_ERRCODE_TRANSACTION_ABORTED         (-46)
#define FS_ERRCODE_INVALID_CHAR                (-47)
```

Symbols

Definition	Description
FS_ERRCODE_OK	No error
FS_ERRCODE_EOF	End of file reached
FS_ERRCODE_PATH_TOO_LONG	Path to file or directory is too long

Definition	Description
FS_ERRCODE_INVALID_PARA	Invalid parameter passed
FS_ERRCODE_WRITE_ONLY_FILE	File can only be written
FS_ERRCODE_READ_ONLY_FILE	File can not be written
FS_ERRCODE_READ_FAILURE	Error while reading from storage medium
FS_ERRCODE_WRITE_FAILURE	Error while writing to storage medium
FS_ERRCODE_FILE_IS_OPEN	Trying to delete a file which is open
FS_ERRCODE_PATH_NOT_FOUND	Path to file or directory not found
FS_ERRCODE_FILE_DIR_EXISTS	File or directory already exists
FS_ERRCODE_NOT_A_FILE	Trying to open a directory instead of a file
FS_ERRCODE_TOO_MANY_FILES_OPEN	Exceeded number of files opened at once (trial version)
FS_ERRCODE_INVALID_FILE_HANDLE	The file handle has been invalidated
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	Trying to open a file instead of a directory
FS_ERRCODE_FILE_DIR_NOT_FOUND	File or directory not found
FS_ERRCODE_NOT_SUPPORTED	Functionality not supported
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

Definition	Description
FS_ERRCODE_INVALID_FS_TYPE	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_VOLUME_FULL	No more space on storage medium
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
FS_ERRCODE_OUT_OF_MEMORY	Could not allocate memory
FS_ERRCODE_UNKNOWN_DEVICE	Storage device is not configured
FS_ERRCODE_ASSERT_FAILURE	FS_DEBUG_ASSERT() macro failed
FS_ERRCODE_TOO_MANY_TRANSACTIONS_OPEN	Maximum number of opened journal transactions exceeded
FS_ERRCODE_NO_OPEN_TRANSACTION	Trying to close a journal transaction which is not opened
FS_ERRCODE_INIT_FAILURE	Error while initializing the storage medium
FS_ERRCODE_FILE_TOO_LARGE	Trying to write to a file which is larger than 4 Gbytes
FS_ERRCODE_HW_LAYER_NOT_SET	The HW layer is not configured
FS_ERRCODE_INVALID_USAGE	Trying to call a function in an invalid state
FS_ERRCODE_TOO_MANY_INSTANCES	Trying to create one instance more than maximum configured
FS_ERRCODE_TRANSACTION_ABORTED	A journal transaction has been stopped by the application
FS_ERRCODE_INVALID_CHAR	Invalid character in the name of a file

Additional information

The last error code of an file handle can be checked using FS_FError().

4.15 Configuration checking functions

The functions described in this section are useful to check at runtime how the file system has been configured at compile time.

4.15.1 FS_CONF_GetDebugLevel()

Description

Returns the level of debug information configured for the file system.

Prototype

```
int FS_CONF_GetDebugLevel(void);
```

Return value

Value of FS_DEBUG_LEVEL configuration define.

4.15.2 FS_CONF_GetDirectoryDelimiter()

Description

Returns the character that is configured as delimiter between the directory names in a file path.

Prototype

```
char FS_CONF_GetDirectoryDelimiter(void);
```

Return value

Value of FS_DIRECTORY_DELIMITER configuration define.

Example

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

void SampleCONFGetDirectoryDelimiter(void) {
  char ac[100];
  char Delim;

  Delim = FS_CONF_GetDirectoryDelimiter();
  sprintf(ac, "The '%c' character is used to delimit directory names in a path\n", Delim);
  FS_X_Log(ac);
}
```


4.15.3 FS_CONF_GetMaxPath()

Description

Returns the configured maximum number of characters in a path to a file or directory.

Prototype

```
int FS_CONF_GetMaxPath(void);
```

Return value

Value of FS_MAX_PATH configuration define.

Example

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

void SampleCONFGetMaxPath(void) {
  char ac[100];
  int  NumChars;

  NumChars = FS_CONF_GetMaxPath();
  sprintf(ac, "The maximum number of characters in the path is: %d\n", NumChars);
  FS_X_Log(ac);
}
```


4.15.4 FS_CONF_GetNumVolumes()

Description

Returns the maximum number of volumes configured for the file system.

Prototype

```
int FS_CONF_GetNumVolumes(void);
```

Return value

Returns the value of FS_NUM_VOLUMES configuration define.

Example

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

void SampleCONFGetNumVolumes(void) {
  char ac[100];
  int  NumVolumes;

  NumVolumes = FS_CONF_GetNumVolumes();
  sprintf(ac, "The maximum number of supported by Journal: %d\n", NumVolumes);
  FS_X_Log(ac);
}
```


4.15.5 FS_CONF_GetOSLocking()

Description

Returns the type of task locking configured for the file system.

Prototype

```
int FS_CONF_GetOSLocking(void);
```

Return value

Value of FS_OS_LOCKING configuration define.

Example

```
#include "FS.h"

void SampleCONFGetOSLocking(void) {
  int LockingType;

  LockingType = FS_CONF_GetOSLocking();
  switch (LockingType) {
  case FS_OS_LOCKING_NONE:
    FS_X_Log("The file system is not locked against concurrent access.\n");
    break;
  case FS_OS_LOCKING_API:
    FS_X_Log("The file system is locked at API function level.\n");
    break;
  case FS_OS_LOCKING_DRIVER:
    FS_X_Log("The file system is locked at device driver level.\n");
    break;
  default:
    FS_X_Log("The type of locking is unknown.\n");
    break;
  }
}
```


4.15.6 FS_CONF_IsCacheSupported()

Description

Checks if the file system is configured to support the sector cache.

Prototype

```
int FS_CONF_IsCacheSupported(void);
```

Return value

Value of FS_SUPPORT_CACHE configuration define.

Additional information

This function does not check if the sector cache is actually active. It only indicates if the file system has been compiled with support for sector cache. The sector cache has to be activated via FS_AssignCache().

Example

```
#include "FS.h"

void SampleCONFIsCacheSupported(void) {
  int IsSupported;

  IsSupported = FS_CONF_IsCacheSupported();
  if (IsSupported) {
    FS_X_Log("The sector cache is supported.\n");
  } else {
    FS_X_Log("The sector cache is not supported.\n");
  }
}
```


4.15.7 FS_CONF_IsDeInitSupported()

Description

Checks if the file system is configured to support deinitialization.

Prototype

```
int FS_CONF_IsDeInitSupported(void);
```

Return value

Value of FS_SUPPORT_DEINIT configuration define.

Example

```
#include "FS.h"

void SampleCONFIsDeInitSupported(void) {
  int IsSupported;

  IsSupported = FS_CONF_IsDeInitSupported();
  if (IsSupported) {
    FS_X_Log("File system deinitialization is supported.\n");
  } else {
    FS_X_Log("File system deinitialization is not supported.\n");
  }
}
```


4.15.8 FS_CONF_IsEFSSupported()

Description

Checks if the file system is configured to support the EFS file system.

Prototype

```
int FS_CONF_IsEFSSupported(void);
```

Return value

Value of FS_SUPPORT_EFS configuration define.

Example

```
#include "FS.h"

void SampleCONFIsEFSSupported(void) {
  int IsSupported;

  IsSupported = FS_CONF_IsEFSSupported();
  if (IsSupported) {
    FS_X_Log("The file system supports EFS.\n");
  } else {
    FS_X_Log("The file system does not support EFS.\n");
  }
}
```


4.15.9 FS_CONF_IsEncryptionSupported()

Description

Checks if the file system is configured to support encryption.

Prototype

```
int FS_CONF_IsEncryptionSupported(void);
```

Return value

Value of `FS_SUPPORT_ENCRYPTION` configuration define.

Example

```
#include "FS.h"

void SampleCONFIsEncryptionSupported(void) {
  int IsSupported;

  IsSupported = FS_CONF_IsEncryptionSupported();
  if (IsSupported) {
    FS_X_Log("The file system supports encryption.\n");
  } else {
    FS_X_Log("The file system does not support encryption.\n");
  }
}
```


4.15.10 FS_CONF_IsFATSupported()

Description

Checks if the file system is configured to support the FAT file system.

Prototype

```
int FS_CONF_IsFATSupported(void);
```

Return value

Value of FS_SUPPORT_FAT configuration define.

Example

```
#include "FS.h"

void SampleCONFIsFATSupported(void) {
  int IsSupported;

  IsSupported = FS_CONF_IsFATSupported();
  if (IsSupported) {
    FS_X_Log("The file system supports FAT.\n");
  } else {
    FS_X_Log("The file system does not support FAT.\n");
  }
}
```


4.15.11 FS_CONF_IsFreeSectorSupported()

Description

Checks if the file system is configured to support the “free sector” command.

Prototype

```
int FS_CONF_IsFreeSectorSupported(void);
```

Return value

Value of FS_SUPPORT_FREE_SECTOR configuration define.

Example

```
#include "FS.h"

void SampleCONFIsFreeSectorSupported(void) {
  int IsSupported;

  IsSupported = FS_CONF_IsFreeSectorSupported();
  if (IsSupported) {
    FS_X_Log("The file system supports \"free sector\" operation.\n");
  } else {
    FS_X_Log("The file system does not support \"free sector\" operation.\n");
  }
}
```


4.15.12 FS_CONF_IsJournalSupported()

Description

Checks if the file system is configured to support journaling.

Prototype

```
int FS_CONF_IsJournalSupported(void);
```

Return value

Value of FS_SUPPORT_JOURNAL configuration define.

Example

```
#include "FS.h"

void SampleCONFIsJournalSupported(void) {
  int IsSupported;

  IsSupported = FS_CONF_IsJournalSupported();
  if (IsSupported) {
    FS_X_Log("The file system supports journaling.\n");
  } else {
    FS_X_Log("The file system does not support journaling.\n");
  }
}
```


4.15.13 FS_CONF_IsTrialVersion()

Description

Checks if the file system has been configured as a trial (limited) version.

Prototype

```
int FS_CONF_IsTrialVersion(void);
```

Return value

= 0	Full version.
≠ 0	Trial version.

Example

```
#include "FS.h"

void SampleCONFIsTrialVersion(void) {
  int IsTrial;

  IsTrial = FS_CONF_IsTrialVersion();
  if (IsTrial) {
    FS_X_Log("This is a trial version of emFile.\n");
  } else {
    FS_X_Log("This is a full-featured version of emFile.\n");
  }
}
```


4.15.14 FS_GetVersion()

Description

Returns the version number of the file system.

Prototype

```
U16 FS_GetVersion(void);
```

Return value

Version number.

Additional information

The version is formatted as follows: `Mmmrr`

where:

- `M` the major version number
- `mm` the minor version number
- `rr` the revision number

For example 40201 represents the version 4.02a (major version: 4, minor version: 2, revision: a)

Example

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

void SampleGetVersion(void) {
  int  Version;
  int  MajorVersion;
  int  MinorVersion;
  char PatchVersion;
  char ac[100];

  Version = FS_GetVersion();
  MajorVersion = Version / 10000;
  MinorVersion = Version / 100 % 100;
  PatchVersion = Version % 100;
  sprintf(ac, "This is the version %d.%d.%d of emFile.\n",
    MajorVersion, MinorVersion, PatchVersion);
  FS_X_Log(ac);
}
```


4.16 Obsolete functions

This section describes API functions that should no longer be used in new applications. These functions will be removed in a future version of emFile.

4.16.1 FS_AddOnExitHandler()

Description

Registers a deinitialization callback.

Prototype

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

Parameters

Parameter	Description
pCB	in Structure holding the callback information.
pfOnExit	Pointer to the callback function to be invoked.

Additional information

The pCB memory location is used internally by the file system and it should remain valid from the time the handler is registered until the FS_DeInit() function is called. The FS_DeInit() function invokes all the registered callback functions 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 set to 1.

4.16.2 FS_CloseDir()

Description

Closes a directory.

Prototype

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

Parameters

Parameter	Description
pDir	Handle to an opened directory.

Return value

= 0 Directory has been closed.
≠ 0 An error occurred.

4.16.3 FS_ConfigOnWriteDirUpdate()

Description

Configures if the directory entry has be updated after writing to file.

Prototype

```
void FS_ConfigOnWriteDirUpdate(char OnOff);
```

Parameters

Parameter	Description
OnOff	Specifies if the feature has to be enabled or disabled. • 1 Enable update directory after write. • 0 Do not update directory. FS_FClose() updates the directory entry.

4.16.4 FS_DirEnt2Attr()

Description

Loads attributes of a directory entry.

Prototype

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

Parameters

Parameter	Description
pDirEnt	in Data of directory entry.
pAttr	out Pointer to a memory location that receives the attributes.

4.16.5 FS_DirEnt2Name()

Description

Loads the name of a directory entry.

Prototype

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

Parameters

Parameter	Description
pDirEnt	in Data of directory entry.
pBuffer	out Pointer to a memory location that receives the name.

4.16.6 FS_DirEnt2Size()

Description

Loads the size of a directory entry.

Prototype

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

Parameters

Parameter	Description
pDirEnt	in Data of directory entry.

Return value

Size of the file or directory.

4.16.7 FS_DirEnt2Time()

Description

Loads the time stamp of a directory entry.

Prototype

```
U32 FS_DirEnt2Time(FS_DIRENT * pDirEnt);
```

Parameters

Parameter	Description
pDirEnt	in Data of directory entry.

Return value

Time stamp of the file or directory.

4.16.8 FS_GetNumFiles()

Description

API function. Returns the number of files in a directory.

Prototype

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

Parameters

Parameter	Description
pDir	Pointer to an opened directory handle.

Return value

0xFFFFFFFF Indicates failure. 0 - 0xFFFFFFFE File size of the given file.

4.16.9 FS_OpenDir()

Description

API function. Open an existing directory for reading.

Prototype

```
FS_DIR *FS_OpenDir(const char * sDirName);
```

Parameters

Parameter	Description
sDirName	Fully qualified directory name.

Return value

= 0 Unable to open the directory.
≠ 0 Address of an FS_DIR data structure.

4.16.10 FS_ReadDir()

Description

Reads next directory entry in directory.

Prototype

```
FS_DIRENT *FS_ReadDir(FS_DIR * pDir);
```

Parameters

Parameter	Description
pDir	Handle to an opened directory.

Return value

≠ 0 Pointer to a directory entry.
= 0 No more directory entries or error.

4.16.11 FS_RewindDir()

Description

Sets pointer for reading the next directory entry to the first entry in the directory.

Prototype

```
void FS_RewindDir(FS_DIR * pDir);
```

Parameters

Parameter	Description
pDir	Handle to an opened directory.

Chapter 5

Caching and buffering

This chapter gives an introduction into cache handling of emFile. Furthermore, it contains the function description and an example.

5.1 Sector cache

The sector cache is a memory area where frequently used sector data can be stored for fast access. In many cases, this can enhance the average access time thus increasing the performance. With applications which do not use a cache, data will always be read from the storage medium even if it has been used before. The sector cache stores accessed and processed sector data. If the sector data has to be processed again, it will be copied out of the cache instead of reading it from the storage device. This condition is called "hit". When the sector data is not present in the sector cache and it has to be read from the storage device. This condition is called "miss". The sector cache works efficiently when the number of hit conditions is greater than the number of miss conditions. The number of hit and miss conditions can be queried using the `FS_STORAGE_GetCounters()` function.

5.1.1 Write cache and journaling

The write back caching has to be disabled if the journaling is employed to make the file system fail safe. The journaling will not work properly if the write back cache (and generally any form of write cache) is enabled. More detailed information can be found in the section *Journaling and write caching* on page 1005.

5.1.2 Types of caches

emFile supports the usage of different cache modules as listed in the following table:

Cache module	Description
`FS_CACHE_ALL`	A pure read cache.
`FS_CACHE_MAN`	A pure read cache that caches only the management sectors.
`FS_CACHE_RW`	A read / write cache module.
`FS_CACHE_RW_QUOTA`	A read / write cache module with configurable capacity per sector type.
`FS_CACHE_MULTI_WAY`	A read / write cache module with configurable associativity level.

5.1.2.1 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. The caching is enabled right after calling `FS_AssignCache()`.

5.1.2.2 FS_CACHE_MAN

This module is also a pure read cache. In contrast to the `FS_CACHE_ALL`, this module only caches the management sectors of the file system (for example, the sectors of the allocation table). The caching is enabled right after calling `FS_AssignCache()`. This type of cache is useful if the application is creating and deleting a large number of files.

5.1.2.3 FS_CACHE_RW

This is a configurable cache module. It can be either used as read, write or as read / write cache. Additionally, the type of sectors that has to be cached are also configurable via `FS_CACHE_SetMode()`.

5.1.2.4 FS_CACHE_RW_QUOTA

This is a configurable cache module. It can be either used as read, write or as read / write cache. Additionally, the type of sectors and the maximum number of sectors of a given type that can be cached at the same time are configurable via `FS_CACHE_SetMode()` and `FS_CACHE_SetQuota()` respectively. It is mandatory to call `FS_CACHE_SetMode()` and `FS_CACHE_SetQuota()`, otherwise the functionality of the cache module is disabled.

5.1.2.5 FS_CACHE_MULTI_WAY

This is configurable cache module that can be used as read, write or read / write cache. The type of sectors that should be cached and the number of places in the cache where the data of a sector can be mapped (associativity level) is configurable via `FS_CACHE_SetMode()` and `FS_CACHE_SetQuota()` respectively. Per default, this cache works with an associativity level of two.

5.1.3 Cache API functions

The following functions can be used to enable, configure and control the emFile sector cache modules.

Function	Description
FS_AssignCache()	Enables / disables a sector cache for a specific volume.
FS_CACHE_Clean()	Writes modified sector data to storage device.
FS_CACHE_GetNumSectors()	Queries the size of the sector cache.
FS_CACHE_Invalidate()	Removes all sector data from cache.
FS_CACHE_SetAssocLevel()	Modifies the associativity level of multi-way cache.
FS_CACHE_SetMode()	Sets the operating mode of sector cache.
FS_CACHE_SetQuota()	Sets the quotas of a specific sector cache.

5.1.3.1 FS_AssignCache()

Description

Enables / disables a sector cache for a specific volume.

Prototype

```
U32 FS_AssignCache(const char          * sVolumeName,
                   void                * pData,
                   I32                   NumBytes,
                   FS_INIT_CACHE       * pfInit);
```

Parameters

Parameter	Description
sVolumeName	Name of the volume on which the sector cache should be enabled / disabled. Cannot be NULL.
pData	Pointer to a memory area that should be used as cache.
NumBytes	Number of bytes in the memory area pointed by pData.
pfInit	Pointer to the initialization function of the cache module that has to be used. It can take one of the following values: • FS_CACHE_ALL • FS_CACHE_MAN • FS_CACHE_RW • FS_CACHE_RW_QUOTA • FS_CACHE_MULTI_WAY.

Return value

> 0	Number of sectors which fit in cache.
= 0	An error occurred. The memory area cannot be used as sector cache.

Additional information

The first configured volume is used if the empty string is specified as sVolumeName.

To disable the cache for a specific device, call FS_AssignCache() with NumBytes set to 0. In this case the function returns 0.

A portion of the memory area assigned to the cache is used to store management data. The following defines can help an application allocate a memory area large enough to store a specified number of sectors: FS_SIZEOF_CACHE_ALL(), FS_SIZEOF_CACHE_MAN(), FS_SIZEOF_CACHE_RW(), FS_SIZEOF_CACHE_RW_QUOTA(), or FS_SIZEOF_CACHE_MULTI_WAY().

Example

The following example demonstrates how to enable and then disable a sector cache of type FS_CACHE_ALL.

```
#include "FS.h"

#define CACHE_SIZE FS_SIZEOF_CACHE_ALL(200, 512) // 200 sectors of 512 bytes.

static U32 _aCache[CACHE_SIZE / 4]; // Allocate RAM for the cache buffer.

void SampleCacheAssign(void) {
  //
  // Assign a read cache to the first configured storage device.
  //
  FS_AssignCache("", _aCache, sizeof(_aCache), FS_CACHE_ALL);

  //
  // Access the file system with sector cache.
```



```
  //

  //
  // Disable the read cache.
  //
  FS_AssignCache("", 0, 0, 0);
}
```


5.1.3.2 FS_CACHE_Clean()

Description

Writes modified sector data to storage device.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume for which the cache should be cleaned. If not specified, the first volume will be used.

Additional information

This function can be used to make sure that modifications made to cached data are also committed to storage device.

Because only write or read / write caches need to be cleaned, this function can only be called for volumes where FS_CACHE_RW, FS_CACHE_RW_QUOTA, or FS_CACHE_MULTI_WAY module is assigned. The other cache modules ignore the cache clean operation.

The cleaning of the cache is also performed when the volume is unmounted via FS_Unmount() or when the cache is disabled or reassigned via FS_AssignCache().

Example

```
#include "FS.h"

void SampleCacheClean(void) {
  FS_CACHE_Clean("");
}
```


5.1.3.3 FS_CACHE_GetNumSectors()

Description

Queries the size of the sector cache.

Prototype

```
int FS_CACHE_GetNumSectors(const char * sVolumeName,
                           U32  * pNumSectors);
```

Parameters

Parameter	Description
sVolumeName	Name of the volume for which the cache should be cleaned. If not specified, the first volume will be used.
pNumSectors	out Number of sectors that can be stored in the cache at the same time. It cannot be NULL.

Return value

= 0 OK, number of sectors returned.
≠ 0 An error occurred.

Additional information

This function returns the number of sectors that can be stored in the cache at the same time.

Example

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

void SampleCacheGetNumSectors(void) {
  U32 NumSectors;

  FS_CACHE_GetNumSectors("", &NumSectors);
  printf("Number of sectors in cache: %lu\n", NumSectors);
}
```


5.1.3.4 FS_CACHE_Invalidate()

Description

Removes all sector data from cache.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Name of the volume for which the cache should be cleaned. If not specified, the first volume will be used.

Return value

= 0 Success, the cache has been emptied.
≠ 0 An error occurred.

Additional information

This function can be called to remove all the sectors from the cache. FS_CACHE_Invalidate() does not write to storage modified sector data. After calling FS_CACHE_Invalidate() the contents of modified sector data is lost. FS_CACHE_Clean() has to be called first to prevent a data loss.

Example

```
#include "FS.h"

void SampleCacheInvalidate(void) {
  FS_CACHE_Invalidate("");
}
```


5.1.3.5 FS_CACHE_SetAssocLevel()

Description

Modifies the associativity level of multi-way cache.

Prototype

```
int FS_CACHE_SetAssocLevel(const char * sVolumeName,
                           int    AssocLevel);
```

Parameters

Parameter	Description
sVolumeName	Name of the volume for which the cache should be cleaned. If not specified, the first volume will be used.
AssocLevel	Number of entries in the cache. It has to be a power of two value.

Return value

= 0 OK, associativity level set.
≠ 0 An error occurred.

Additional information

This function is supported only by the FS_CACHE_MULTI_WAY cache module. An error is returned if the function is used with any other cache module.

The associativity level specifies on how many different places in the cache the data of the same sector can be stored. The cache replacement policy uses this information to decide where to store the contents of a sector in the cache. Caches with higher associativity levels tend to have higher hit rates. The default associativity level is two.

Example

The following example shows how to configure the FS_CACHE_MULTI_WAY cache module to store the data of the same sector at 4 different places on the cache.

```
#include "FS.h"

void SampleCacheSetAssocLevel(void) {
  FS_CACHE_SetAssocLevel("", 4);
}
```


5.1.3.6 FS_CACHE_SetMode()

Description

Sets the operating mode of sector cache.

Prototype

```
int FS_CACHE_SetMode(const char * sVolumeName,
                     int    TypeMask,
                     int    ModeMask);
```

Parameters

Parameter	Description
sVolumeName	Name of the volume for which the cache should be cleaned. If not specified, the first volume will be used.
TypeMask	Specifies the types of the sectors that have to be cached. This parameter can be an OR-combination of the Sector type masks (FS_SECTOR_TYPE_MASK_...)
ModeMask	Specifies the operating mode of the cache module. This parameter can be one of the values defined as Sector cache modes (FS_CACHE_MODE_...)

Return value

= 0 OK, mode set.
≠ 0 An error occurred.

Additional information

This function is supported by the following cache types: FS_CACHE_RW, FS_CACHE_RW_QUOTA, and FS_CACHE_MULTI_WAY. These cache modules have to be configured using this function otherwise, neither read nor write operations are cached.

When configured in FS_CACHE_MODE_WB mode the cache module writes the sector data automatically to storage device if free space is required for new sector data. The application can call the FS_CACHE_Clean() function at any time to write all the cache sector data to storage device.

Example

The following example shows how to configure the FS_CACHE_MULTI_WAY cache module to use a read cache for the data sectors, a write back cache for the management and directory sectors.

```
#include "FS.h"

#define CACHE_SIZE FS_SIZEOF_CACHE_MULTI_WAY(200, 512)

static U32 _acCache[CACHE_SIZE / 4];  // Allocate RAM for the cache buffer.

void SampleCacheSetMode(void) {
  //
  // Assign a cache to the first available storage device.
  //
  FS_AssignCache("", _acCache, sizeof(_acCache), FS_CACHE_MULTI_WAY);
  //
  // Configure a read cache for the sectors that store file data.
  //
  FS_CACHE_SetMode("", FS_SECTOR_TYPE_MASK_DATA, FS_CACHE_MODE_R);
  //
  // Configure a write back cache for the sectors that store directory
  // and allocation table entries.
  //
  FS_CACHE_SetMode("",
```



```
                    FS_SECTOR_TYPE_MASK_DIR | FS_SECTOR_TYPE_MASK_MAN,
                    FS_CACHE_MODE_WB);
  //
  // Access the file system with sector cache.
  //

  //
  // Disable the cache.
  //
  FS_AssignCache("", 0, 0, FS_CACHE_NONE);
}
```


5.1.3.7 FS_CACHE_SetQuota()

Description

Sets the quotas of a specific sector cache.

Prototype

```
int FS_CACHE_SetQuota(const char * sVolumeName,
                      int    TypeMask,
                      U32    NumSectors);
```

Parameters

Parameter	Description
sVolumeName	Name of the volume for which the cache should be cleaned. If not specified, the first volume will be used.
TypeMask	Specifies the types of the sectors that have to be cached. This parameter can be an OR-combination of the Sector type masks (FS_SECTOR_TYPE_MASK_...)
NumSectors	Specifies the maximum number of sectors to be stored for each sector type specified in TypeMask.

Return value

= 0 OK, maximum number of sectors set.
≠ 0 An error occurred.

Additional information

This function is currently only usable with the FS_CACHE_RW_QUOTA cache module. After the FS_CACHE_RW_QUOTA cache module has been assigned to a volume and the cache mode has been set, the quotas for the different sector types have to be configured using this function. Otherwise, neither read nor write operations are cached.

Example

The following example shows how to configure the FS_CACHE_RW_QUOTA cache module to limit the number of directory and data sector types to a maximum of 10 each.

```
#include "FS.h"

#define CACHE_SIZE   FS_SIZEOF_CACHE_RW_QUOTA(200, 512)

static U32 _acCache[CACHE_SIZE / 4];  // Allocate RAM for the cache buffer.

void SampleCacheSetQuota(void) {
  //
  // Assign a cache to the first available storage device.
  //
  FS_AssignCache("", _acCache, sizeof(_acCache), FS_CACHE_RW_QUOTA);
  //
  // Configure the cache module to cache all sectors. The sector data is cached
  // for read and write. Write operation to storage device are delayed.
  //
  FS_CACHE_SetMode("", FS_SECTOR_TYPE_MASK_ALL, FS_CACHE_MODE_WB);
  //
  // Set the maximum number of sectors to cache for directory
  // and data sector types 10 sectors each.
  //
  FS_CACHE_SetQuota("", FS_SECTOR_TYPE_MASK_DATA | FS_SECTOR_TYPE_MASK_DIR, 10);

  //
  // Access the file system with sector cache.
  //

  //
```



```
  // Make sure that the data is written to storage device.
  //
  FS_CACHE_Clean("");

  //
  // Perform other accesses to file system with sector cache.
  //

  //
  // Disable the cache.
  //
  FS_AssignCache("", 0, 0, FS_CACHE_NONE);
}
```


5.1.3.8 Sector cache types

Description

Types of sector caches.

Definition

```
#define FS_CACHE_NONE          NULL
#define FS_CACHE_ALL           FS_CacheAll_Init
#define FS_CACHE_MAN           FS_CacheMan_Init
#define FS_CACHE_RW            FS_CacheRW_Init
#define FS_CACHE_RW_QUOTA      FS_CacheRWQuota_Init
#define FS_CACHE_MULTI_WAY     FS_CacheMultiWay_Init
```

Symbols

Definition	Description
FS_CACHE_NONE	Pseudo-type that can be used to disable the sector cache.
FS_CACHE_ALL	A pure read cache.
FS_CACHE_MAN	A pure read cache that caches only the management sectors.
FS_CACHE_RW	A read / write cache module.
FS_CACHE_RW_QUOTA	A read / write cache module with configurable capacity per sector type.
FS_CACHE_MULTI_WAY	A read / write cache module with configurable associativity level.

Additional information

The type of a cache module can be configured via `FS_AssignCache()` function when the sector cache is enabled for a specified volume.

5.1.3.9 Sector cache modes

Description

Operating modes of sector caches.

Definition

```
#define FS_CACHE_MODE_R     0x01u
#define FS_CACHE_MODE_WT    (FS_CACHE_MODE_R | FS_CACHE_MODE_W)
#define FS_CACHE_MODE_WB    (FS_CACHE_MODE_R | FS_CACHE_MODE_W | FS_CACHE_MODE_D)
```

Symbols

Definition	Description
FS_CACHE_MODE_R	Pure read cache.
FS_CACHE_MODE_WT	Write-through cache.
FS_CACHE_MODE_WB	Write-back cache.

Additional information

The operating mode of a cache module can be configured via the FS_CACHE_SetMode() function separately for each sector type.

5.1.3.10 Sector cache size

Description

Calculates the cache size.

Definition

```
#define FS_SIZEOF_CACHE_ALL(NumSectors,          SectorSize)
  (FS_SIZEOF_CACHE_ALL_DATA +               \
   (FS_SIZEOF_CACHE_ALL_BLOCK_INFO +       \
   (SectorSize)) * (NumSectors))
#define FS_SIZEOF_CACHE_MAN(NumSectors,          SectorSize)
  (FS_SIZEOF_CACHE_MAN_DATA +               \
   (FS_SIZEOF_CACHE_MAN_BLOCK_INFO +       \
   (SectorSize)) * (NumSectors))
#define FS_SIZEOF_CACHE_RW(NumSectors,           SectorSize)
  (FS_SIZEOF_CACHE_RW_DATA +                \
   (FS_SIZEOF_CACHE_RW_BLOCK_INFO +        \
   (SectorSize)) * (NumSectors))
#define FS_SIZEOF_CACHE_RW_QUOTA(NumSectors,     SectorSize)
  (FS_SIZEOF_CACHE_RW_QUOTA_DATA +          \
   (FS_SIZEOF_CACHE_RW_QUOTA_BLOCK_INFO +  \
   (SectorSize)) * (NumSectors))
#define FS_SIZEOF_CACHE_MULTI_WAY(NumSectors,    SectorSize)
  (FS_SIZEOF_CACHE_MULTI_WAY_DATA +         \
   (FS_SIZEOF_CACHE_MULTI_WAY_BLOCK_INFO + \
   (SectorSize)) * (NumSectors))
#define FS_SIZEOF_CACHE_ANY(NumSectors,          SectorSize)
  FS_SIZEOF_CACHE_RW_QUOTA(NumSectors, SectorSize)
```

Symbols

Definition	Description
FS_SIZEOF_CACHE_ALL(NumSectors,	Calculates the cache size of a FS_CACHE_ALL cache module.
FS_SIZEOF_CACHE_MAN(NumSectors,	Calculates the cache size of a FS_CACHE_MAN cache module.
FS_SIZEOF_CACHE_RW(NumSectors,	Calculates the cache size of a FS_CACHE_RW cache module.
FS_SIZEOF_CACHE_RW_QUOTA(NumSectors,	Calculates the cache size of a FS_CACHE_RW_QUOTA cache module.
FS_SIZEOF_CACHE_MULTI_WAY(NumSectors,	Calculates the cache size of a FS_CACHE_MULTI_WAY cache module.
FS_SIZEOF_CACHE_ANY(NumSectors,	Calculates the size of cache that works with any cache module.

Additional information

These defines can be used to calculate the size of the memory area to be assigned to a cache module based on the size of a sector (SectorSize) and the number of sectors to be cached (NumSectors). The sector size of a specific volume can be queried via FS_GetVolumeInfo().

5.1.4 Performance and resource usage

5.1.4.1 Performance

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 have been measured on a SEGGER emPower evaluation board https://www.segger.com/evaluate-our-software/segger/empower/ using an SD card as storage device. The test application has been compiled using SEGGER Embedded Studio IDE https://www.segger.com/products/development-tools/embedded-studio/ but any other IDE can be used as well. The test application is provided in source code and is located in the `Application` folder of the emFile shipment.

Description of the test

- Initialize the file system.
- Perform a high-level format if required.
- Create 50 files without a cache.
- Write the time which was required for creation in the terminal I/O window.
- Enable a read and write cache.
- Create 50 files with the enabled read and write cache.
- Write the time which was required for creation in the terminal I/O window.
- Flush the cache.
- Write the time which was required for flushing in the terminal I/O window.
- Disable the cache.
- Create again 50 files without a cache.
- Write the time which was required for creation in the terminal I/O window.

Terminal output

```
Start
High-level formatting
Cache disabled
Creation of 50 files took: 16 ms
Cache enabled
Creation of 50 files took: 2 ms
Cache flush took: 0 ms
Cache disabled
Creation of 50 files took: 57 ms
Finished
```

Source code

```
/*********************************************************************
*                   (c) SEGGER Microcontroller GmbH                  *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File    : FS_SectorCache.c
Purpose : Demonstrates how to configure and use the sector cache.

Additional information:
  Preparations:
    Works out-of-the-box with any storage device.

  Expected behavior:
    Measures the time it takes to create a number of files with
    and without the sector cache enabled. The results are output
    on the debug console.

  Sample output:
    Start
    Cache disabled
```



```
    Creation of 50 files took: 60 ms
    Cache enabled
    Creation of 50 files took: 18 ms
    Cache flush took: 0 ms
    Cache disabled
    Creation of 50 files took: 114 ms
    Finished
*/

/*********************************************************************
*
*       #include Section
*
**********************************************************************
*/
#include <string.h>
#include "FS.h"
#include "FS_OS.h"
#include "SEGGER.h"

/*********************************************************************
*
*       Defines, configurable
*
**********************************************************************
*/
#define VOLUME_NAME   ""
#define NUM_FILES     32

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/
static U32  _aCache[0x400];
static char _aacFileName[NUM_FILES][13];
static char _ac[128];

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       _CreateFiles
*/
static void _CreateFiles(void) {
  int       i;
  U32       Time;
  FS_FILE * pFile[NUM_FILES];

  Time = FS_X_OS_GetTime();
  for (i = 0; i < NUM_FILES; i++) {
    pFile[i] = FS_FOpen(&_aacFileName[i][0], "w");
  }
  Time = FS_X_OS_GetTime() - Time;
  SEGGER_snprintf(_ac, (int)sizeof(_ac), "Creation of %d files took: %d ms
\n", NUM_FILES, Time);
  FS_X_Log(_ac);
  for (i = 0; i < NUM_FILES; i++) {
    (void)FS_FClose(pFile[i]);
  }
}

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
```



```
*
*       MainTask
*/
#ifdef __cplusplus
extern "C" {     /* Make sure we have C-declarations in C++ programs */
#endif
void MainTask(void);
#ifdef __cplusplus
}
#endif
void MainTask(void) {
  int i;
  U32 Time;

  FS_X_Log("Start\n");
  //
  // Initialize file system
  //
  FS_Init();
  //
  // Check if low-level format is required
  //
  (void)FS_FormatLLIfRequired(VOLUME_NAME);
  //
  // Check if volume needs to be high level formatted.
  //
  if (FS_IsHLFormatted(VOLUME_NAME) == 0) {
    FS_X_Log("High-level formatting\n");
    (void)FS_Format(VOLUME_NAME, NULL);
  }
  //
  // Prepare file names in advance.
  //
  for (i = 0; i < NUM_FILES; i++) {
    SEGGER_snprintf(_aacFileName[i], (int)sizeof(_aacFileName[0]), "file%.2d.txt", i);
  }
  //
  // Create and measure the time used to create the files with the cache enabled.
  //
  FS_X_Log("Cache disabled\n");
  _CreateFiles();
  //
  // Create and measure the time used to create the files.
  //
  (void)FS_AssignCache(VOLUME_NAME, _aCache, (int)sizeof(_aCache), FS_CACHE_RW);
  (void)FS_CACHE_SetMode(VOLUME_NAME, (int)FS_SECTOR_TYPE_MASK_ALL, (int)FS_CACHE_MODE_WB);
  FS_X_Log("Cache enabled\n");
  _CreateFiles();
  Time = FS_X_OS_GetTime();
  FS_CACHE_Clean(VOLUME_NAME);
  Time = FS_X_OS_GetTime() - Time;
  SEGGER_snprintf(_ac, (int)sizeof(_ac), "Cache flush took: %d ms\n", Time);
  FS_X_Log(_ac);
  //
  // Create and measure the time used to create the files with the cache disabled.
  //
  FS_X_Log("Cache disabled\n");
  (void)FS_AssignCache(VOLUME_NAME, NULL, 0, FS_CACHE_NONE);
  _CreateFiles();
  FS_X_Log("Finished\n");
  for(;;) {
    ;
  }
}

/************************** End of file ***************************/
```


5.2 File buffer

emFile comes with support for file buffer. A file buffer is a small memory area assigned to an opened file where the file system can store frequently accessed file data. Using a file buffer can greatly increase the performance especially when the application has to access small amounts of data.

The file buffer can be configured in different access modes. In read mode, that is only the data that is read by the application from the file is stored to file buffer or in read / write mode where the data read as well as the data written by the application is stored to file buffer. The access modes can be changed at runtime each time the file is opened via `FS_SetFileBufferFlags()` or `FS_SetFileBuffer()`.

The size of the file buffer is configurable either globally for all opened files via `FS_ConfigFileBufferDefault()` or individually for each opened file via `FS_SetFileBuffer()`.

Chapter 6

Device drivers

This chapter describes the file system components that provide access to a storage device.

6.1 General information

emFile has been designed to operate with any kind of storage device. To use a specific type of storage device with emFile, a device driver for that specific storage device is required. A device driver is a file system component that provides block access to a storage device. The device driver consists of basic I/O functions for accessing the storage device and a global table that holds pointers to these functions.

The following table lists the device drivers available in emFile.

Driver name	Acronym	Identifier	Volume name
CompactFlash card and IDE driver	IDE	`FS_IDE_Driver`	`“ide:”`
SPI MMC/SD driver	MMC_SPI	`FS_MMC_SPI_Driver`	`“mmc:”`
Card Mode MMC/SD driver	MMC_CM	`FS_MMC_CM_Driver`	`“mmc:”`
SLC1 NAND driver	NAND	`FS_NAND_Driver`	`“nand:”`
Universal NAND driver	NAND_UNI	`FS_NAND_UNI_Driver`	`“nand:”`
Sector Map NOR driver	NOR	`FS_NOR_Driver`	`“nor:”`
Block Map NOR driver	NOR_BM	`FS_NOR_BM_Driver`	`“nor:”`
RAM Disk driver	RAMDISK	`FS_RAMDISK_Driver`	`“ram:”`
WinDrive driver	WINDRIVE	`FS_WINDRIVE_Driver`	`“win:”`

By default, the file system is not configured to access any particular storage device. A device driver has to be added by the application by calling `FS_AddDevice()` in `FS_X_AddDevices()`. The file system calls `FS_X_AddDevices()` during the initialization to allow the application to configure the file system. `FS_AddDevice()` takes as parameter a pointer to a the function table of the particular device driver as specified in the “Identifier” column in the table above. It is possible to add more than one device driver to the file system if more than one storage device is present in the system or if the system uses one storage device that is partitioned.

The file system assigns a volume to each device driver added by the application. A volume is an internal structure that contains information about organization of the storage device and how to access it. The application uses the volume name as specified in the “Volume name” column in the table above to select a particular storage device. The names of the volumes are fixed and cannot be changed by the application.

If the same device driver is added to file system more than once, an unit number is used to differentiate between them. The unit number is an optional part of the volume name and it can be used by the application to access a specific storage device. Most device driver functions as well as most of the underlying hardware functions receive the unit number as the first parameter. If there are for example tow NAND flash drivers which operate as two different NAND flash 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 number parameter can be ignored by the underlying hardware functions.

6.1.1 Hardware layer

Most of the device drivers require functions to access the hardware. This set of functions is called hardware layer. The implementation of the hardware layer is user responsibility. emFile comes with template and sample hardware layers that can be used as a starting point in developing a new one.

The hardware layer can be implemented in two different ways:

- Polled mode
- Interrupt-driven mode

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 device driver in this way. The following example to demonstrate the fundamentals of an polled-driven hardware layer.

```
#include "FS.h"

/*********************************************************************
*
*       _HW_Write
*
*  Function description
*    FS hardware layer function. Writes a specified number of bytes
*    to storage device via DMA.
*/
static int _HW_Write(U8 Unit, const U8 * pData, int NumBytes) {
  int r;

  //
  // Start transmission using DMA.
  //
  _StartDMAWrite(Unit, pData, NumBytes);
  //
  // Wait for the DMA operation to complete.
  //
  while (1) {
    if (_IsDMAOperationCompleted()) {
      r = 0;          // OK, success.
      break;
    }
    if (_IsDMAError()) {
      r = 1;          // Error, could not write data via DMA.
      break;
    }
  }
  return r;
}
```

Interrupt-driven mode

In the interrupt-driven mode the completion of an I/O operation is signaled through an interrupt. This operating mode requires the support of an operating system. The following example demonstrates the fundamentals of an interrupt-driven hardware layer.

```
#include "FS.h"
#include "FS_OS.h"

/*********************************************************
*
*       _IRQ_Handler
*
*  Function description
*    Handles the hardware interrupt. Has to be registered
*    as interrupt service routine.
*/
static void _IRQ_Handler(void) {
  //
  // Disable further interrupts.
  //
  _DisableInterrupts();
  //
  // Signal (wake) the waiting task.
  //
  FS_X_OS_Signal();
}
```



```
/*********************************************************************
*
*       _HW_Write
*
*  Function description
*    FS hardware layer function. Writes a specified number of bytes
*    to storage device via DMA.
*/
static int _HW_Write(U8 Unit, const U8 * pData, int NumBytes) {
  //
  // Start transmission using DMA.
  //
  _StartDMAWrite(Unit, pData, NumBytes);
  //
  // Wait for the DMA operation to complete.
  //
  while (1) {
    if (_IsDMAOperationCompleted()) {
      r = 0;          // OK, success.
      break;
    }
    if (_IsDMAError()) {
      r = 1;          // Error, could not write data via DMA.
      break;
    }
    //
    // Wait for IRQ to signal the event and wake up the task.
    //
    r = FS_X_OS_Wait(TIMEOUT);
    if (r) {
      break;          // Error, timeout expired.
    }
  }
}
```


6.2 RAM Disk driver

This device driver uses a region of the system memory as storage. It is very helpful in getting started using emFile since it does not require any hardware access functions. The RAM Disk driver can also be used for testing the file system on a PC.

6.2.1 Supported hardware

The RAM Disk driver can be used with every target that has sufficient memory. The size of the storage is defined as the number of sectors reserved for the file system.

6.2.2 Theory of operation

The RAM Disk driver stores the data of the logical sectors it receives from file system without modification to the memory region configured by the application via `FS_RAMDISK_Configure()`. The byte offset at which the data is stored is calculated by multiplying the index of the logical sector by the size of a logical sector. For example, the data of the logical sector with the index 7 is stored at the byte offset 7 * 512 = 3584 assuming that the size of the logical sector is 512 bytes.

6.2.3 Fail-safe operation

When power is lost, all the data is typically lost, except for systems with battery backup. For this reason, the fail-safety is relevant only for systems that provide such battery backup.

Unexpected reset

In case of an unexpected reset the data will be preserved. However, if the unexpected reset interrupts a write operation, the data of the sector may contain partially invalid data.

Power failure

A 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 Formatting

The storage used by RAM Disk driver is unformatted after each startup. Exceptions to this rule are systems, that use a battery to back up the system memory. The application has to format the storage via `FS_Format()` before it can store data on it. If the application adds only one RAM Disk driver to file system, `FS_Format()` can be called with an empty string as volume name. For example, `FS_Format("", NULL)`. If more than one RAM Disk driver is added to file system, a the volume name has to be specified. For example, `FS_Format("ram:0:", NULL)` for the first storage device and `FS_Format("ram:1:", NULL)` for the second.

6.2.6 Configuring the driver

The application has to add the RAM Disk driver to the file system before using it. This is done by calling `FS_AddDevice()` in `FS_X_AddDevices()` with the parameter set to the function table of the RAM Disk driver `FS_RAMDISK_Driver`.

In addition, the application has to configure via `FS_RAMDISK_Configure()` the memory region the RAM Disk driver has to use as storage. The memory region can be located on any system memory that is accessible to CPU.

The maximum number of RAM Disk driver instances that can be added to file system can be configured a compile-time using the FS_RAMDISK_MAX_NUM_UNITS configuration define. By default, a maximum number of four RAM Disk drivers can be added to file system.

Example

```
#include <FS.h>

/*********************************************************************
*
*       Defines, configurable
*
**********************************************************************
*/
#define ALLOC_SIZE          0x1100    // Memory pool for the file system in bytes
#define NUM_SECTORS         16        // Number of sectors on the RAM storage
#define BYTES_PER_SECTOR    512       // Size of a sector in bytes

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/
//
// Memory pool used for semi-dynamic allocation.
//
static U32 _aMemBlock[ALLOC_SIZE / 4];
//
// Memory to be used as storage.
//
static U32 _aRAMDisk[(NUM_SECTORS * BYTES_PER_SECTOR) / 4];

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       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 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) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add and configure the driver.
  //
  FS_AddDevice(&FS_RAMDISK_Driver);
  FS_RAMDISK_Configure(0, _aRAMDisk, BYTES_PER_SECTOR, NUM_SECTORS);
#if FS_SUPPORT_FILE_BUFFER
  //
  // Enable the file buffer to increase the performance
  // when reading/writing a small number of bytes.
  //
  FS_ConfigFileBufferDefault(BYTES_PER_SECTOR, FS_FILE_BUFFER_WRITE);
#endif
}
```

6.2.6.1 Compile-time configuration

The compile time configuration is realized via preprocessor defines that have to be added to the FS_Conf.h file which is the main configuration file of emFile. For detailed information about the configuration of emFile and of the configuration define types, refer to *Configu-*

ration of emFile on page 927 The WinDrive driver does not require any hardware access functions. The following table lists the configuration defines supported by the journaling component.

Type	Define	Default value	Description
N	FS_RAMDISK_NUM_UNITS	4	Maximum number of driver instances.

6.2.6.1.1 FS_RAMDISK_NUM_UNITS

This define specifies the maximum number of driver instances that can be created by the application. The memory for each driver instance is allocated statically.

6.2.6.2 Runtime configuration

This section describes the functions that can be used to configure an instance of the RAM Disk driver.

Function	Description
FS_RAMDISK_Configure()	Configures an instance of RAM disk driver.

6.2.6.2.1 FS_RAMDISK_Configure()

Description

Configures an instance of RAM disk driver.

Prototype

```
void FS_RAMDISK_Configure(U8     Unit,
                          void * pData,
                          U16    BytesPerSector,
                          U32    NumSectors);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
pData	Memory area to be used as storage. It cannot be NULL.
BytesPerSector	Number of bytes in a logical sector.
NumSectors	Number of logical sectors in the storage.

Additional information

The application has to call this function for each instance of the RAM disk driver it adds to file system. Unit identifies the instance of the RAM disk driver. The instance of the RAM disk driver added first to file system has the index 0, the second instance has the index 1, and so on.

pData has to point to a memory region that is at least BytesPerSector * NumSectors bytes large. The memory region can by located on any system memory that is accessible by CPU. BytesPerSector has to be a power of 2 value.

Example

```
#include <FS.h>

#define NUM_SECTORS         16        // Number of sectors on the RAM storage
#define BYTES_PER_SECTOR    512       // Size of a sector in bytes

static U32 _aRAMDisk[(NUM_SECTORS * BYTES_PER_SECTOR) / 4]; // Memory to be used as storage.

/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*/
void FS_X_AddDevices(void) {
  //
  // Basic file system configuration...
  //

  //
  // Add and configure the driver.
  //
  FS_AddDevice(&FS_RAMDISK_Driver);
  FS_RAMDISK_Configure(0, _aRAMDisk, BYTES_PER_SECTOR, NUM_SECTORS);

  //
  // Additional file system configuration...
  //
}
```


6.2.7 Performance and resource usage

6.2.7.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the RAM Disk driver has been measured using the SEGGER Embedded Studio IDE V4.20 configured to generate code for a Cortex-M4 CPU in Thumb mode and with the size optimization enabled.

Usage: 500 bytes

6.2.7.2 Static RAM usage

Static RAM usage refers to the amount of RAM required by the RAM Disk driver internally for all the driver instances. The number of bytes can be seen in the compiler list file of the `RAMDISK.c` file.

Usage: 40 bytes

6.2.7.3 Dynamic RAM usage

The RAM Disk driver does not allocate any dynamic memory.

6.2.7.4 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 have been performed as described in *Performance* on page 992. All values are given in Mbytes/sec.

CPU type	Write speed	Read speed
Atmel AT91SAM9261 (200 MHz)	128.0	128.0
LogicPD LH79520 (51 MHz)	20.0	20.0

6.3 NAND flash driver

6.3.1 General information

emFile supports the use of raw NAND flash devices as data storage. Two drivers for NAND flash devices also known as flash translation layers are available:

- *SLC1 NAND driver* - Supports SLC NAND flash devices that require 1-bit error correction. In addition, it also supports the Microchip / Atmel and Adesto DataFlash devices.
- *Universal NAND driver* - Supports all modern SLC and MLC NAND flash devices. It can make use of the hardware ECC engine integrated into NAND flash devices to correct bit errors.

6.3.1.1 Selecting the correct NAND driver

The first factor to be considered is the type of storage device used. Microchip / Atmel and Adesto DataFlash devices are currently supported only by the SLC1 NAND driver while NAND flash devices are typically supported by both NAND drivers. The bit error correction requirement of the NAND flash device is the next factor. It indicates how many bit errors the error correcting code (ECC) have to be able to detect and correct. If the NAND flash devices requires only 1-bit correction capability then the SLC1 NAND driver can be used. The SLC1 NAND driver performs the bit error detection and correction in the software. For an correction capability larger than 1 bit the Universal NAND driver has to be used instead. The Universal NAND driver is able to calculated the ECC either in the hardware using the ECC integrated into the NAND flash device or MCU or in the software using an ECC library such as SEGGER emLib-ECC (https://www.segger.com/products/security-iot/emlib/variations/ecc/)

You can always get in contact with us if you are not sure what type of NAND driver to choose.

6.3.1.2 Multiple driver configurations

Both NAND drivers store management information to spare area of a page. The layout and the content of this information is different for each NAND driver which means that data written using one NAND driver cannot be accessed with the other one and vice versa. For example, if an application uses the SLC1 NAND driver to store data to a NAND flash device than it cannot use the Universal NAND driver to access it. The Universal NAND driver reports in this case that the NAND flash device is not properly formatted. For applications that have to support different types of NAND flash devices that require the usage of both NAND drivers it is possible to select the correct NAND driver by checking the device id of the connected NAND flash device. The description of the `FS_NAND_PHY_ReadDeviceId()` function provides an example showing how to a NAND flash device can be identified at the configuration of the file system.

6.3.1.3 Software structure

The NAND driver is organized into different layers and contains from top to bottom:

- NAND driver layer
- *NAND physical layer*
- *NAND hardware layer*

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 hard- ware access.

6.3.1.4 Bad block management

Bad block management is supported in the driver. Blocks marked as defective are skipped and are not used for data storage. The block is recognized as defective when the first byte

in the spare area of the first or second page is different than 0xFF. The driver marks blocks as defective in the following cases:

- when the NAND flash reports an error after a write operation.
- when the NAND flash reports an error after an erase operation.
- when an uncorrectable ECC error is detected on the data read from NAND flash.

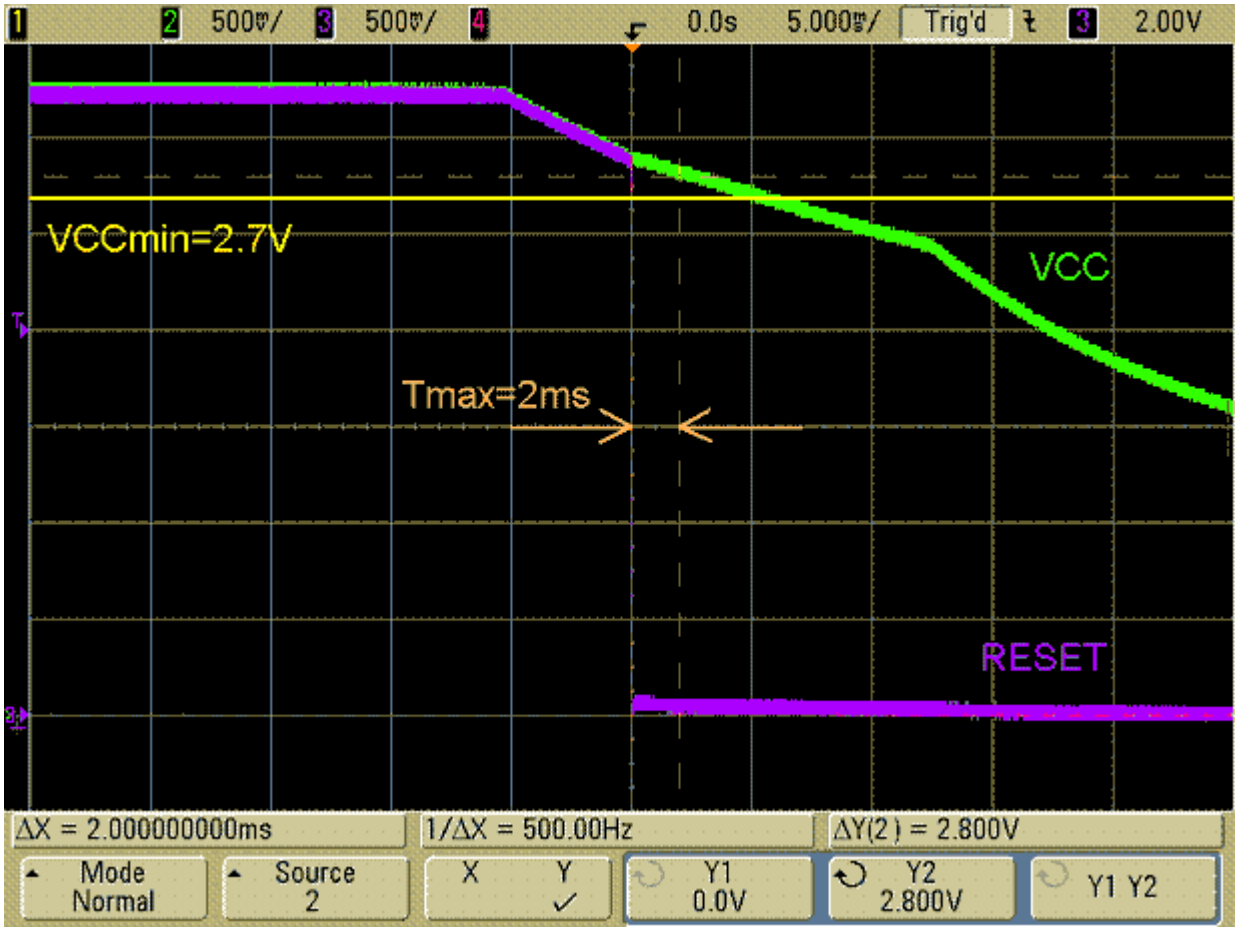
6.3.1.5 Garbage collection

The driver performs the garbage collection automatically during the write operations. If no empty NAND blocks are available to store the data, new empty NAND blocks are created by erasing the data of the NAND blocks marked as invalid. This involves a NAND block erase operation which, depending on the characteristics of the NAND flash, can potentially take a long time to complete, and therefor reduces the write throughput. For applications which require maximum write throughput, the garbage collection can be done in the application. Typically, this 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 NAND blocks are converted to free NAND blocks. A write operation following the call to this function runs at maximum speed. The other function, FS_STORAGE_CleanOne(), converts a single invalid NAND block. Depending on the number of invalid NAND blocks, several calls to this function are required to clean up the entire NAND flash.

6.3.1.6 Fail-safe operation

The NAND drivers are fail-safe which means that the drivers make only atomic operations and guarantee that the file system data is always valid. If an unexpected power loss or reset interrupts a a write operation, then the old data is kept while the new corrupted data is discarded. The fail-safe operation can only be guaranteed if the NAND flash device is able to fully complete the last command it received from the MCU. The picture below is an oscilloscope capture which shows how a power down sequence should look like in order to meet the requirements needed for a fail-safe operation of a NAND driver. The labels in the picture have the following meaning:

- 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.



6.3.1.7 Wear leveling

Wear leveling is supported by the NAND drivers. The procedure makes sure that the number of times a NAND block is erased remains approximately equal for all the NAND blocks. This is realized by maintaining an erase count for each NAND block. The maximum allowed erase count difference is configurable at compile time via `FS_NAND_MAX_ERASE_CNT_DIFF` as well as at runtime via `FS_NAND_SetMaxEraseCntDiff()` for the SLC1 NAND driver and `FS_NAND_UNI_SetMaxEraseCntDiff()` for the Universal NAND driver.

6.3.1.8 Read disturb errors

Read disturb errors are bit errors that occur when a large number of read operations (a few hundred thousand to one million) are preformed on a NAND flash block without an erase operation taking place in between. These bit errors can be avoided by simply rewriting the sector data. This can be realized by calling in the application the `FS_STORAGE_RefreshSectors()` storage layer API function that is able to refresh multiple sectors in a single call.

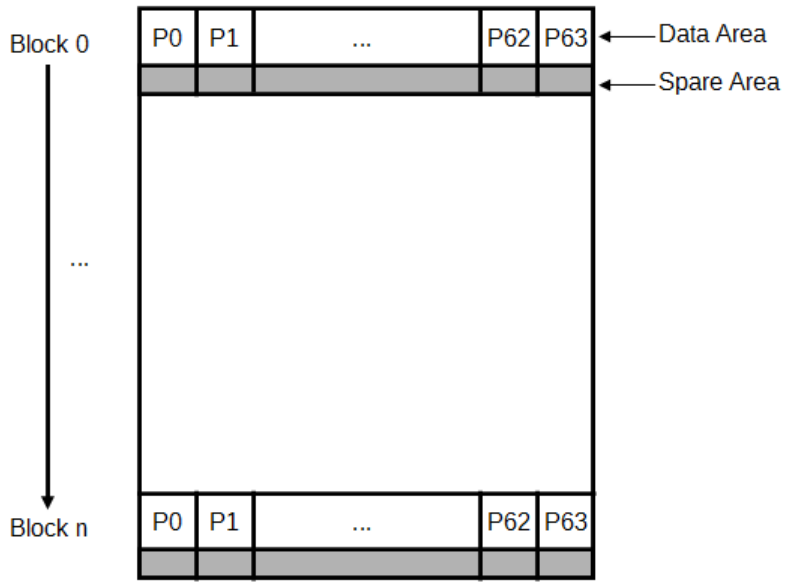
6.3.1.9 Low-level format

Before using a NAND flash device for the first time, the application has to initialize it by performing a low-level format operation on it. Refer to `FS_FormatLow()` and `FS_FormatLLIfRequired()` for detailed information about this.

6.3.1.10 NAND flash organization

A NAND flash device is a serial-type memory device that uses the I/O pins for both address and data transfer as well as for command inputs. A NAND flash device consist of a number of individual storage 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 changed from 1 to 0. Only entire blocks (all pages in the block) can be erased. Erasing

means bringing all memory bits in all pages of the block to logical 1. The erase and program operations are executed automatically by the NAND flash device.



Small NAND flashes (up to 256 Mbytes) have a page size of 528 bytes, that is 512 bytes for storing file system data and 16 bytes, called the spare area, for storing management information (ECC, etc.).

Large NAND devices (256 Mbytes or more) have a page size of 2112 bytes with 2048 bytes used for storing file system data data and with 64 bytes of spare area.

6.3.1.11 Pin description - parallel NAND flash device

This type of devices uses 8 or 16 I/O signals to exchange the data with the MCU an additional set of signals that are used to control the data transfer.

Signal name	Description
CE	Chip Enable - The CE signal enables the device. This signal is active low. If the signal is inactive, device is in standby mode.
WE	Write Enable - The WE signal controls the transfer of data from MCU to NAND flash device. This signal is active low. Commands, address and data are latched on the rising edge of the WE pulse.
RE	Read Enable - The RE signal controls the transfer of data from NAND flash device to MCU. This signal is active low. When active the NAND flash device outputs data.
CLE	Command Latch Enable - The CLE signal controls the activating path for commands sent to NAND flash device. This signal is active high. When active, command data sent via the I/O signals is latched by the NAND flash device into the internal command register on the rising edge of the WE signal.
ALE	Address Latch Enable - The ALE signal controls the activating path for address to the internal address registers. This signal is active high. Address data is latched on the rising edge of WE with ALE high.
WP	Write Protect - The WP signal controls if data can be stored to NAND flash device. This signal is active low and is typically connected to VCC (recommended), but may also be connected to port pin.
R/B	Ready/Busy - The R/B signal indicates the status of the NAND flash device operation to MCU. When low, it indicates that a program, erase or read operation is in process. The signal returns to high when the operation is completed. It is an open drain output that should be connected to a port

Signal name	Description
	pin with pull-up. If available, a port pin which can trigger an interrupt can be connected to this signal.
I/O0-I/O7	Data Input/Output - The data I/O signals are used to input command, address and data, and to output data during read operations.
I/O8-I/O15	Data Input/Output - Additional data I/O signals.

6.3.1.12 Pin description - DataFlash device

DataFlash devices are commonly used when low pin count and easy data transfer are required. DataFlash devices use the following pins:

Signal name	Description
CS	Chip Select - This signal selects the DataFlash device. The device is selected, when CS pin is driven low.
SCLK	Serial Clock - The SCLK signal 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 signal is used to transfer data to the DataFlash device. It is used for all data input including opcodes and address sequences.
SO	Serial Data Out - The SO signal is used to transfer data serially out of the DataFlash device.

Additionally, the host system has to comply with the following requirements:

- The data transfer width is 8 bit.
- The Chip Select (CS) signal sets the device active at low-level and inactive at high level.
- The clock signal has to be generated by the target system. The serial flash devices are always in slave mode.
- The bit order requires most significant bit (MSB) to be sent out first.

6.3.2 SLC1 NAND driver

SLC1 NAND driver is an extremely efficient and low footprint driver that can be used to access the data stored on a raw NAND flash device. The driver is designed to support one or multiple Single Level Cell (SLC) NAND flash devices that require 1-bit error correction capability. The SLC1 NAND flash driver can also be used to access the data stored on Microchip / Atmel and Adesto DataFlash devices. Different sector sizes are supported to help reduce the RAM usage of the file system.

6.3.2.1 Supported hardware

the SLC1 NAND driver supports almost all SLC NAND flash devices. This includes NAND flash devices with page sizes of 512 + 16 and 2048 + 64 bytes.

6.3.2.1.1 Tested and compatible NAND flash devices

The table below lists the NAND flash devices that have been tested or are compatible with a tested device:

Device	Page size (bytes)	Capacity (bits)
Hynix		
HY27xS08281A	512+16	16Mx8
HY27xS08561M	512+16	32Mx8
HY27xS08121M	512+16	64Mx8
HY27xA081G1M	512+16	128Mx8
HY27UF082G2M	2048+64	256Mx8
HY27UF084G2M	2048+64	512Mx8
HY27UG084G2M	2048+64	512Mx8
HY27UG084GDM	2048+64	512Mx8
Micron		
MT29F2G08AAB	2048+64	256Mx8
MT29F2G08ABD	2048+64	256Mx8
MT29F4G08AAA	2048+64	512Mx8
MT29F4G08BAB	2048+64	512Mx8
MT29F2G16AAD	2048+64	128Mx16
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

Device	Page size (bytes)	Capacity (bits)
K9K1G08U0M	512+16	128Mx8
K9T1GJ8U0M	512+16	128Mx8
K9F1G08x0A	2048+64	256Mx8
K9F2G08U0M	2048+64	256Mx8
K9K2G08R0A	2048+64	256Mx8
K9K2G08U0M	2048+64	256Mx8
K9F4G08U0M	2048+64	512Mx8
K9F8G08U0M	2048+64	1024Mx8
STM		
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
NAND01GR3B	2048+64	128Mx8
NAND01GW3B	2048+64	128Mx8
NAND02GR3B	2048+64	256Mx8
NAND02GW3B	2048+64	256Mx8
NAND04GW3	2048+64	512Mx8
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

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.

6.3.2.1.2 Tested and compatible DataFlash devices

The NAND flash driver fully supports the Microchip / Atmel and Adesto DataFlash devices with capacities up to 128 MBit. Currently, the following devices are supported:

Device	Page size (bytes)	Capacity (bits)
Microchip / Atmel		
AT45DB011B	256+8	1Mx1
AT45DB021B	256+8	2Mx1
AT45DB041B	256+8	4Mx1

Device	Page size (bytes)	Capacity (bits)
AT45DB081B	256+8	8Mx1
AT45DB161B	512+16	16Mx1
AT45DB321C	512+16	32Mx1
AT45BR3214B	512+16	32Mx1
AT45DCB002	512+16	16Mx1
AT45DCB004	512+16	32Mx1
AT45DCB008	512+16	64Mx1
AT45DB642D	1024+32	64Mx1
AT45DB1282	1024+32	128Mx1
Adesto		
AT45DB321E	512+16	32Mx1

Note

DataFlash devices with a page size that is a power of 2 are not supported by the SLC1 NAND driver.

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. 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-Correcting code{c}) 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.

6.3.2.3 Error correction using ECC

The SLC1 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 calculated has 2 or more bit errors, the data cannot be corrected. Standard memory controllers calculate an ECC for the complete block size (512 / 2048 bytes). The ECC routines of the SLC1 NAND driver calculates the ECC for data chunks of 256 bytes (e.g. a page with 2048 bytes is divided into 8 parts of 256 bytes), therefore 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 SLC1 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 SLC1 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.

6.3.2.4 Partial write operations

Most 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 extensive 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 four. Exceeding the maximum number of partial writes does not lead automatically to the corruption of data in that page but it can increase the probability of bit errors. The driver is be able to correct single bit errors using ECC. For some combinations of logical sector size and NAND page size the driver might exceed the limit for the number of partial write operations. The table below summarizes the maximum number of partial writes performed by the SLC1 NAND driver for different logical sector sizes:

NAND page size (bytes)	Logical sector size (bytes)	Maximum number of partial write operations
512	512	4
2048	2048	4
2048	1024	5
2048	512	8

6.3.2.5 Configuring the driver

This section describes how to configure the file system to make use of the SLC1 NAND driver.

6.3.2.5.1 Runtime configuration

The driver must be added to the file system by calling `FS_AddDevice()` with the driver identifier set to the address of `FS_NAND_Driver`. This function call together with other function calls that configure the driver operation have to be added to `FS_X_AddDevices()` as demonstrated in the following example.

Example

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

#define ALLOC_SIZE      0x9000          // Memory pool for the file system in bytes

static U32 _aMemBlock[ALLOC_SIZE / 4];  // Memory pool used for semi-dynamic allocation.

void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add the driver to file system.
  //
  FS_AddDevice(&FS_NAND_Driver);
  //
  // Set the physical interface of the NAND flash.
  //
  FS_NAND_SetPhyType(0, &FS_NAND_PHY_x8);
  //
  // Skip 2 blocks (256 Kbytes in case of 2K device)
  // The size of the configured area is 128 blocks.
  // For 2K devices, this means 2 Kbytes * 64 * 128 = 16 Mbytes
  //
  FS_NAND_SetBlockRange(0, 2, 128);
  //
  // Configure the HW access routines.
  //
  FS_NAND_x8_SetHWType(0, &FS_NAND_HW_Template);
#if FS_SUPPORT_FILE_BUFFER
  //
```



```
  // Enable the file buffer to increase the performance
  // when reading/writing a small number of bytes.
  //
  FS_ConfigFileBufferDefault(512, FS_FILE_BUFFER_WRITE);
#endif // FS_SUPPORT_FILE_BUFFER
}
```

The API functions listed in the next table can be used by the application to configure the behavior of the SLC1 NAND driver. The application can call them only at the file system initialization in FS_X_AddDevices().

Function	Description
FS_NAND_SetBlockRange()	Specifies which NAND flash blocks can used as data storage.
FS_NAND_SetCleanThreshold()	Specifies the minimum number sectors that the driver should keep available for fast write operations.
FS_NAND_SetEraseVerification()	Enables or disables the checking of the block erase operation.
FS_NAND_SetMaxEraseCntDiff()	Configures the threshold of the wear leveling procedure.
FS_NAND_SetNumWorkBlocks()	Sets number of work blocks the SLC1 NAND driver uses for write operations.
FS_NAND_SetOnFatalErrorCallback()	Registers a function to be called by the driver when a fatal error occurs.
FS_NAND_SetPhyType()	Configures NAND flash access functions.
FS_NAND_SetWriteVerification()	Enables or disables the checking of each page write operation.

6.3.2.5.1.1 FS_NAND_SetBlockRange()

Description

Specifies which NAND flash blocks can used as data storage.

Prototype

```
void FS_NAND_SetBlockRange(U8  Unit,
                           U16 FirstBlock,
                           U16 MaxNumBlocks);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
FirstBlock	Index of the first NAND flash block to be used as storage (0-based).
MaxNumBlocks	Maximum number of NAND flash blocks to be used as storage.

Additional information

This function is optional. By default, the SLC1 NAND driver uses all blocks of the NAND flash as data storage. FS_NAND_SetBlockRange() is useful when a part of the NAND flash has to be used for another purpose, for example to store the application program used by a boot loader, and therefore it cannot be managed by the SLC1 NAND driver. Limiting the number of blocks used by the SLC1 NAND driver can also help reduce the RAM usage.

FirstBlock is the index of the first physical NAND block were 0 is the index of the first block of the NAND flash device. MaxNumBlocks can be larger that the actual number of available NAND blocks in which case the SCL1 NAND driver silently truncates the value to reflect the actual number of NAND blocks available.

The SLC1 NAND driver uses the first NAND block in the range to store management information at low-level format. If the first NAND block happens to be marked as defective, then the next usable NAND block is used.

The read optimization of the FS_NAND_PHY_2048x8 physical layer has to be disabled when this function is used to partition the NAND flash device in order to ensure data consistency. The read cache can be disabled at runtime using FS_NAND_2048x8_DisableReadCache().

If the FS_NAND_SetBlockRange() is used to subdivide the same physical NAND flash device into two or more partitions than the application has to make sure that they do not overlap.

Example

Sample configuration showing how to reserve the first 16 blocks of the NAND flash.

```
#include <FS.h>

void FS_X_AddDevices(void) {
  //
  // Basic file system configuration...
  //

  //
  // Add and configure the SLC1 NAND driver.
  // Only the NAND blocks with the physical
  // indexes 16-143 are used for storage.
  //
  FS_AddDevice(&FS_NAND_Driver);
  FS_NAND_SetPhyType(0, &FS_NAND_PHY_2048x8);
  FS_NAND_SetBlockRange(0, 16, 128);

  //
  // Additional file system configuration...
```



```
  //
}
```

Sample configuration showing how to partition the same NAND flash device.

```
#include <FS.h>

void FS_X_AddDevices(void) {
  //
  // Basic file system configuration...
  //

  //
  // Add and configure the first SLC1 NAND driver.
  // The blocks 16 blocks are used for storage,
  // that is the NAND blocks with the physical
  // indexes 0-15.
  //
  FS_AddDevice(&FS_NAND_Driver);
  FS_NAND_SetPhyType(0, &FS_NAND_PHY_2048x8);
  FS_NAND_SetBlockRange(0, 0, 16);

  //
  // Add and configure the second SLC1 NAND driver.
  // The blocks 64 blocks are used for storage,
  // that is the NAND blocks with the physical
  // indexes 16-79.
  //
  FS_AddDevice(&FS_NAND_Driver);
  FS_NAND_SetPhyType(0, &FS_NAND_PHY_2048x8);
  FS_NAND_SetBlockRange(0, 16, 64);

  //
  // Additional file system configuration...
  //
}
```


6.3.2.5.1.2 FS_NAND_SetCleanThreshold()

Description

Specifies the minimum number sectors that the driver should keep available for fast write operations.

Prototype

```
int FS_NAND_SetCleanThreshold(U8       Unit,
                              unsigned NumBlocksFree,
                              unsigned NumSectorsFree);
```

Parameters

Parameter	Description
Unit	Unit number of NAND flash device.
NumBlocksFree	Number of blocks to be kept free.
NumSectorsFree	Number of sectors to be kept free on each block.

Return value

= 0 OK, threshold set.
≠ 0 An error occurred.

Additional information

Typically, used for allowing the NAND flash to write data fast to a file on a sudden reset. At the startup, the application reserves free space, by calling the function with NumBlocksFree and NumSectorsFree set to a value different than 0. The number of free sectors depends on the number of bytes written and on how the file system is configured. When the unexpected reset occurs, the application tells the driver that it can write to the free sectors, by calling the function with NumBlocksFree and NumSectorsFree set to 0. Then, the application writes the data to file and the NAND driver stores it to the free space. Since no erase or copy operation is required, the data is written as fastest as the NAND flash device permits it.

The NAND flash device will wear out faster than normal if sectors are reserved in a work block (NumSectors > 0).

6.3.2.5.1.3 FS_NAND_SetEraseVerification()

Description

Enables or disables the checking of the block erase operation.

Prototype

```
void FS_NAND_SetEraseVerification(U8 Unit,
                                  U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
OnOff	Specifies if the feature has to be enabled or disabled • =0 The erase operation is not checked. • ≠0 The erase operation is checked.

Additional information

This function is optional. The result of a block erase operation is normally checked by evaluating the error bits maintained by the NAND flash device in a internal status register. FS_NAND_SetEraseVerification() can be used to enable additional verification of the block erase operation that is realized by reading back the contents of the entire erased physical block and by checking that all the bytes in it are set to 0xFF. Enabling this feature can negatively impact the write performance of SLC1 NAND driver driver.

The block erase verification feature is active only when the SLC1 NAND driver is compiled with the FS_NAND_VERIFY_ERASE configuration define is set to 1 (default is 0) or when the FS_DEBUG_LEVEL configuration define is set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL.

6.3.2.5.1.4 FS_NAND_SetMaxEraseCntDiff()

Description

Configures the threshold of the wear leveling procedure.

Prototype

```
void FS_NAND_SetMaxEraseCntDiff(U8  Unit,
                                U32 EraseCntDiff);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
EraseCntDiff	Maximum allowed difference between the erase counts of any two NAND blocks.

Additional information

This function is optional. It can be used to control how the SLC1 NAND driver performs the wear leveling. The wear leveling procedure makes sure that the NAND blocks are equally erased to meet the life expectancy of the storage device by keeping track of the number of times a NAND block has been erased (erase count). The SLC1 NAND driver executes this procedure when a new empty NAND block is required for data storage. The wear leveling procedure works by first choosing the next available NAND block. Then the difference between the erase count of the chosen block and the NAND block with lowest erase count is computed. If this value is greater than EraseCntDiff the NAND block with the lowest erase count is freed and made available for use.

The same threshold can also be configured at compile time via the FS_NAND_MAX_ERASE_CNT_DIFF configuration define.

Example

```
#include <FS.h>

void FS_X_AddDevices(void) {
  //
  // Basic file system configuration...
  //

  //
  // Add and configure the SLC1 NAND driver.
  //
  FS_AddDevice(&FS_NAND_Driver);
  FS_NAND_SetPhyType(0, &FS_NAND_PHY_2048x8);
  FS_NAND_SetMaxEraseCntDiff(0, 5000);

  //
  // Additional file system configuration...
  //
}
```


6.3.2.5.1.5 FS_NAND_SetNumWorkBlocks()

Description

Sets number of work blocks the SLC1 NAND driver uses for write operations.

Prototype

```
void FS_NAND_SetNumWorkBlocks(U8  Unit,
                              U32 NumWorkBlocks);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
NumWorkBlocks	Number of work blocks.

Additional information

This function is optional. It can be used to change the default number of work blocks according to the requirements of the application. Work blocks are physical NAND blocks that the SLC1 NAND driver uses to temporarily store the data written to NAND flash device. The SLC1 NAND driver calculates at low-level format the number of work blocks based on the total number of blocks available on the NAND flash device.

By default, the NAND driver allocates 10% of the total number of NAND blocks used as storage, but no more than 10 NAND blocks. The minimum number of work blocks allocated by default depends on whether journaling is used or not. If the journal is active 4 work blocks are allocated, else SLC1 NAND driver allocates 3 work blocks. The currently allocated number of work blocks can be checked via FS_NAND_GetDiskInfo(). The value is returned in the NumWorkBlocks member of the FS_NAND_DISK_INFO structure.

Increasing the number of work blocks can help increase the write performance of the SLC1 NAND driver. At the same time the RAM usage of the SLC1 NAND driver increases since each configured work block requires a certain amount of RAM for the data management. This is a trade-off between write performance and RAM usage.

The new value take effect after the NAND flash device is low-level formatted via FS_FormatLow() or FS_NAND_FormatLow() API functions.

Example

```
#include <FS.h>

void FS_X_AddDevices(void) {
  //
  // Basic file system configuration...
  //

  //
  // Add and configure the SLC1 NAND driver.
  //
  FS_AddDevice(&FS_NAND_Driver);
  FS_NAND_SetPhyType(0, &FS_NAND_PHY_2048x8);
  FS_NAND_SetNumWorkBlocks(0, 5);

  //
  // Additional file system configuration...
  //
}
```


6.3.2.5.1.6 FS_NAND_SetOnFatalErrorCallback()

Description

Registers a function to be called by the driver when a fatal error occurs.

Prototype

```
void FS_NAND_SetOnFatalErrorCallback
                                (FS_NAND_ON_FATAL_ERROR_CALLBACK * pfOnFatalError);
```

Parameters

Parameter	Description
pfOnFatalError	Pointer to callback function.

Additional information

This function is optional. If no callback function is registered the SLC1 NAND driver behaves as if the callback function returned 1. This means that the NAND flash remains writable after the occurrence of the fatal error. emFile versions previous to 4.04b behave differently and mark the NAND flash as read-only in this case. For additional information refer to FS_NAND_ON_FATAL_ERROR_CALLBACK.

Typically, the SLC1 NAND driver reports a fatal error when an uncorrectable bit error occurs, that is when the ECC is not able to correct the bit errors. A fatal error can also be reported on other events such the failure to erase a NAND block. The type of error is indicated to the callback function via the ErrorType member of the FS_NAND_FATAL_ERROR_INFO structure.

All instances of the SLC1 NAND driver share the same callback function. The Unit member of the FS_NAND_FATAL_ERROR_INFO structure passed as parameter to the pfOnFatalError callback function can be used to determine which driver instance triggered the fatal error.

6.3.2.5.1.7 FS_NAND_SetPhyType()

Description

Configures NAND flash access functions.

Prototype

```
void FS_NAND_SetPhyType(      U8                  Unit,
                        const FS_NAND_PHY_TYPE * pPhyType);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based)
pPhyType	in Physical layer to be used to access the NAND flash device.

Additional information

This function is mandatory and it has to be called in FS_X_AddDevices() once for each instance of the SLC1 NAND driver. The driver instance is identified by the Unit parameter. First SLC1 NAND driver instance added to the file system via a FS_AddDevice(&FS_NAND_Driver) call has the unit number 0, the SLC1 NAND driver added by a second call to FS_AddDevice() has the unit number 1 and so on.

6.3.2.5.1.8 FS_NAND_SetWriteVerification()

Description

Enables or disables the checking of each page write operation.

Prototype

```
void FS_NAND_SetWriteVerification(U8 Unit,
                                  U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
OnOff	Specifies if the feature has to be enabled or disabled • =0 The write operation is not checked. • ≠0 The write operation is checked.

Additional information

This function is optional. The result of a page write operation is normally checked by evaluating the error bits maintained by the NAND flash device in a internal status register. FS_NAND_SetWriteVerification() can be used to enable additional verification of the page write operation that is realized by reading back the contents of the written page and by checking that all the bytes are matching the data requested to be written. Enabling this feature can negatively impact the write performance of SLC1 NAND driver.

The page write verification feature is active only when the SLC1 NAND driver is compiled with the FS_NAND_VERIFY_WRITE configuration define is set to 1 (default is 0) or when the FS_DEBUG_LEVEL configuration define is set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL.

6.3.2.5.1.9 FS_NAND_ON_FATAL_ERROR_CALLBACK

Description

The type of the callback function invoked by the NAND driver when a fatal error occurs.

Type definition

```
typedef int FS_NAND_ON_FATAL_ERROR_CALLBACK(FS_NAND_FATAL_ERROR_INFO * pFatalErrorInfo);
```

Parameters

Parameter	Description
pFatalErrorInfo	Information about the fatal error.

Return value

= 0 The NAND driver has to mark the NAND flash device as read only.
≠ 0 The NAND flash device has to remain writable.

Additional information

If the callback function returns a 0 the NAND driver marks the NAND flash device as read-only and it remains in this state until the NAND flash device is low-level formated. In this state all further write operations are rejected with an error by the NAND driver.

The application is responsible to handle the fatal error by for example checking the consistency of the file system via FS_CheckDisk(). The callback function is not allowed to invoke any other FS API functions therefore the handling of the error has to be done after the FS API function that triggered the error returns.

6.3.2.5.1.10 FS_NAND_FATAL_ERROR_INFO

Description

Information passed to callback function when a fatal error occurs.

Type definition

```
typedef struct {
  U8   Unit;
  U8   ErrorType;
  U32  ErrorSectorIndex;
} FS_NAND_FATAL_ERROR_INFO;
```

Structure members

Member	Description
Unit	Instance of the NAND driver that generated the fatal error.
ErrorType	Type of fatal error.
ErrorSectorIndex	Index of the physical sector where the error occurred. Not applicable for all type of errors.

6.3.2.6 Additional driver functions

The following functions are optional and can be used by the application to perform operations directly on the NAND flash device.

Function	Description
FS_NAND_Clean()	Makes sectors available for fast write operations.
FS_NAND_EraseBlock()	Sets all the bytes in a NAND block to 0xFF.
FS_NAND_EraseFlash()	Erases the entire NAND partition.
FS_NAND_FormatLow()	Performs a low-level format of the NAND flash device.
FS_NAND_GetBlockInfo()	Returns information about a specified NAND block.
FS_NAND_GetDiskInfo()	Returns information about the NAND partition.
FS_NAND_GetStatCounters()	Returns the actual values of statistical counters.
FS_NAND_IsBlockBad()	Checks if a NAND block is marked as defective.
FS_NAND_IsLLFormatted()	Checks if the NAND flash is low-level formatted.
FS_NAND_ReadPageRaw()	Reads data from a page without ECC.
FS_NAND_ReadPhySector()	This function reads a physical sector from NAND flash.
FS_NAND_ResetStatCounters()	Sets the values of statistical counters to 0.
FS_NAND_TestBlock()	Fills all the pages in a block (including the spare area) with the specified pattern and verifies if the data was written correctly.
FS_NAND_WritePage()	Stores data to a page of a NAND flash with ECC.
FS_NAND_WritePageRaw()	Stores data to a page of a NAND flash without ECC.

6.3.2.6.1 FS_NAND_Clean()

Description

Makes sectors available for fast write operations.

Prototype

```
int FS_NAND_Clean(U8       Unit,
                  unsigned NumBlocksFree,
                  unsigned NumSectorsFree);
```

Parameters

Parameter	Description
Unit	Unit number of NAND flash device.
NumBlocksFree	Number of blocks to be kept free.
NumSectorsFree	Number of sectors to be kept free on each block.

Return value

= 0 OK
≠ 0 An error occurred

Additional information

This function is optional. It can be used to free space on the NAND flash device for data that the application has to write as fast as possible. FS_NAND_UNI_Clean() performs two internal operations: (1) Converts all work blocks that have less free sectors than NumSectorsFree into data blocks. (2) If required, convert work blocks until at least NumBlocksFree are available.

6.3.2.6.2 FS_NAND_EraseBlock()

Description

Sets all the bytes in a NAND block to `0xFF`.

Prototype

```
int FS_NAND_EraseBlock(U8       Unit,
                       unsigned BlockIndex);
```

Parameters

Parameter	Description
`Unit`	Index of the driver instance (0-based).
`BlockIndex`	Index of the NAND flash block to be erased.

Return value

= 0 OK, block erased.
≠ 0 An error occurred.

Additional information

This function is optional. `FS_NAND_EraseBlock()` function does not check if the block is marked as defective before erasing it.

6.3.2.6.3 FS_NAND_EraseFlash()

Description

Erases the entire NAND partition.

Prototype

```
int FS_NAND_EraseFlash(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).

Return value

≥ 0 Number of blocks which failed to erase.
< 0 An error occurred.

Additional information

This function is optional. After the call to FS_NAND_EraseFlash() all the bytes in the NAND partition are set to 0xFF.

This function has to be used with care, since it also erases blocks marked as defective and therefore the information about the block status will be lost. FS_NAND_EraseFlash() can be used without this side effect on storage devices that are guaranteed to not have any bad blocks, such as DataFlash devices.

6.3.2.6.4 FS_NAND_FormatLow()

Description

Performs a low-level format of the NAND flash device.

Prototype

```
int FS_NAND_FormatLow(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).

Return value

= 0 OK, NAND flash device is low-level formatted.
≠ 0 An error occurred.

Additional information

This function is optional. It is recommended that application use FS_FormatLow() to initialize the NAND flash device instead of FS_NAND_FormatLow().

After the low-level format operation all data that was stored on the NAND flash device is lost. A low-level format has to be performed only once before using the NAND flash device for the first time. The application can check if the NAND flash device is low-level formatted by calling FS_IsLLFormated() or alternatively FS_NAND_IsLLFormated().

6.3.2.6.5 FS_NAND_GetBlockInfo()

Description

Returns information about a specified NAND block.

Prototype

```
int FS_NAND_GetBlockInfo(U8                   Unit,
                         U32                  PhyBlockIndex,
                         FS_NAND_BLOCK_INFO * pBlockInfo);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
PhyBlockIndex	Index of the physical block to get information about.
pBlockInfo	out Information about the NAND block.

Return value

= 0 OK, information returned.
≠ 0 An error occurred.

Additional information

This function is not required for the functionality of the SLC1 NAND driver and is typically not linked in in production builds.

Example

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

void SampleNANDGetBlockInfo(void) {
  FS_NAND_BLOCK_INFO BlockInfo;

  printf("Get information about block 1 of the first SLC1 NAND driver instance\n");
  FS_NAND_GetBlockInfo(0, 1, &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);
}
```


6.3.2.6.6 FS_NAND_GetDiskInfo()

Description

Returns information about the NAND partition.

Prototype

```
int FS_NAND_GetDiskInfo(U8                  Unit,
                        FS_NAND_DISK_INFO * pDiskInfo);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
pDiskInfo	out Information about the NAND partition.

Return value

= 0 OK, information returned.
≠ 0 An error occurred.

Additional information

This function is not required for the functionality of the SLC1 NAND driver and is typically not linked in production builds.

Example

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

void SampleNANDGetDiskInfo(void) {
  FS_NAND_DISK_INFO DiskInfo;

  memset(&DiskInfo, 0, sizeof(DiskInfo));
  printf("Get information about the first SLC1 NAND driver instance\n");
  FS_NAND_GetDiskInfo(0, &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);
}
```


6.3.2.6.7 FS_NAND_GetStatCounters()

Description

Returns the actual values of statistical counters.

Prototype

```
void FS_NAND_GetStatCounters(U8                       Unit,
                             FS_NAND_STAT_COUNTERS * pStat);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based)
pStat	out Values of statistical counters.

Additional information

This function is optional. It is active only when the file system is compiled with FS_DEBUG_LEVEL set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL or FS_NAND_ENABLE_STATS set to 1.

The statistical counters can be cleared via FS_NAND_ResetStatCounters().

6.3.2.6.8 FS_NAND_IsBlockBad()

Description

Checks if a NAND block is marked as defective.

Prototype

```
int FS_NAND_IsBlockBad(U8       Unit,
                       unsigned BlockIndex);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
BlockIndex	Index of the NAND flash block to be checked.

Return value

1	Block is defective
0	Block is not defective

Additional information

This function is optional.

6.3.2.6.9 FS_NAND_IsLLFormatted()

Description

Checks if the NAND flash is low-level formatted.

Prototype

```
int FS_NAND_IsLLFormatted(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).

Return value

= 1 NAND flash device is low-level formatted.
= 0 NAND flash device is not low-level formatted.
< 0 An error occurred.

Additional information

This function is optional.

6.3.2.6.10 FS_NAND_ReadPageRaw()

Description

Reads data from a page without ECC.

Prototype

```
int FS_NAND_ReadPageRaw(U8         Unit,
                        U32        PageIndex,
                        void     * pData,
                        unsigned   NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
PageIndex	Index of the page to be read.
pData	out Data to be written.
NumBytes	Number of bytes to be read.

Return value

= 0 OK, data read.
≠ 0 An error occurred.

Additional information

This function is optional.

The data is read beginning from byte offset 0 in the page. If more data is requested than the page + spare area size, typ. 2 Kbytes + 64 bytes, the function does not modify the remaining bytes in pData.

6.3.2.6.11 FS_NAND_ReadPhySector()

Description

This function reads a physical sector from NAND flash.

Prototype

```
int FS_NAND_ReadPhySector(U8         Unit,
                          U32        PhySectorIndex,
                          void     * pData,
                          unsigned * pNumBytesData,
                          void     * pSpare,
                          unsigned * pNumBytesSpare);
```

Parameters

Parameter	Description
Unit	Unit/Index number of the NAND flash to read from.
PhySectorIndex	Physical sector index.
pData	Pointer to a buffer to store read data.
pNumBytesData	in Pointer to variable storing the size of the data buffer. out The number of bytes that were stored in the data buffer.
pSpare	Pointer to a buffer to store read spare data.
pNumBytesSpare	in Pointer to variable storing the size of the spare data buffer. out The number of bytes that were stored in the spare data buffer.

Return value

< 0	OK, all bytes are set to 0xFF in the physical sector.
= 0	OK, no bit errors.
= 1	OK, one bit error found and corrected.
= 2	OK, one bit error found in the ECC.
= 3	Error, more than 1 bit errors occurred.
= 4	Error, could not read form NAND flash device.
= 5	Error, internal operation.

Additional information

This function is optional.

6.3.2.6.12 FS_NAND_ResetStatCounters()

Description

Sets the values of statistical counters to 0.

Prototype

```
void FS_NAND_ResetStatCounters(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based)

Additional information

This function is optional. It is active only when the file system is compiled with FS_DEBUG_LEVEL set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL or FS_NAND_ENABLE_STATS set to 1.

The statistical counters can be queried via FS_NAND_GetStatCounters().

6.3.2.6.13 FS_NAND_TestBlock()

Description

Fills all the pages in a block (including the spare area) with the specified pattern and verifies if the data was written correctly.

Prototype

```
int FS_NAND_TestBlock(U8                   Unit,
                      unsigned             BlockIndex,
                      U32                  Pattern,
                      FS_NAND_TEST_INFO *  pInfo);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
BlockIndex	Index of the NAND block to be tested.
Pattern	Data pattern to be written during the test.
pInfo	out Optional information about the test result.

Return value

FS_NAND_TEST_RETVAL_OK	OK, no bit errors.
FS_NAND_TEST_RETVAL_CORRECTABLE_ERROR	OK, correctable bit errors found. The number of bit errors is returned in NumErrorsCorrectable of pResult.
FS_NAND_TEST_RETVAL_FATAL_ERROR	Fatal error, uncorrectable bit error found. The page index is returned in PageIndexFatalError of pResult.
FS_NAND_TEST_RETVAL_BAD_BLOCK	Bad block, skipped.
FS_NAND_TEST_RETVAL_ERASE_FAILURE	Erase operation failed. The block has been marked as defective.
FS_NAND_TEST_RETVAL_WRITE_FAILURE	Write operation failed. The block has been marked as defective.
FS_NAND_TEST_RETVAL_READ_FAILURE	Read operation failed.
FS_NAND_TEST_RETVAL_INTERNAL_ERROR	NAND flash access error.

Additional information

This function is optional. It can be used by the application to test the data reliability of a NAND block. BlockIndex is relative to the beginning of the NAND partition where the first block has the index 0.

Example

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

void _SampleNANDTestBlock(void) {
  int               r;
  U32               NumPhyBlocks;
  U32               iBlock;
  FS_NAND_DISK_INFO DiskInfo;
  FS_NAND_TEST_INFO TestInfo;

  memset(&DiskInfo, 0, sizeof(DiskInfo));
  memset(&TestInfo, 0, sizeof(TestInfo));
  FS_NAND_GetDiskInfo(0, &DiskInfo);
  NumPhyBlocks = DiskInfo.NumPhyBlocks;
  for (iBlock = 0; iBlock < NumPhyBlocks; ++iBlock) {
    r = FS_NAND_TestBlock(0, iBlock, 0xAA5500FFuL, &TestInfo);
    switch (r) {
    case FS_NAND_TEST_RETVAL_OK:
```



```
      printf("Block %lu: OK.\n", iBlock);
      break;
    case FS_NAND_TEST_RETVAL_CORRECTABLE_ERROR:
      printf("Block %lu: %lu correctable error(s).\n", iBlock, TestInfo.BitErrorCnt);
      break;
    case FS_NAND_TEST_RETVAL_FATAL_ERROR:
      printf("Block %lu: %lu correctable error(s), fatal error on page %lu.
\n", iBlock, TestInfo.BitErrorCnt, TestInfo.PageIndex);
      break;
    case FS_NAND_TEST_RETVAL_BAD_BLOCK:
      printf("Block %lu: Bad. Skipped.\n", iBlock);
      break;
    case FS_NAND_TEST_RETVAL_ERASE_FAILURE:
      printf("Block %lu: Erase failure. Marked as bad.\n", iBlock);
      break;
    case FS_NAND_TEST_RETVAL_WRITE_FAILURE:
      printf("Block %lu: Write failure on page %lu. Marked as bad.
\n", iBlock, TestInfo.PageIndex);
      break;
    case FS_NAND_TEST_RETVAL_READ_FAILURE:
      printf("Block %lu: Read failure on page %lu.\n", iBlock, TestInfo.PageIndex);
      break;
    default:
      printf("Block %lu: Internal error.\n", iBlock);
      break;
    }
  }
}
```


6.3.2.6.14 FS_NAND_WritePage()

Description

Stores data to a page of a NAND flash with ECC.

Prototype

```
int FS_NAND_WritePage(      U8         Unit,
                            U32        PageIndex,
                      const void     * pData,
                            unsigned   NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
PageIndex	Index of the page to be written.
pData	in Data to be written.
NumBytes	Number of bytes to be written.

Return value

= 0 OK, data written.
≠ 0 An error occurred.

Additional information

This function is optional.

The data is written beginning with the byte offset 0 in the page. If more data is written than the size of the page, typically 2 KB + 64 bytes, the excess bytes are discarded. Data in the area reserved for ECC cannot be written using this function and it will be overwritten.

6.3.2.6.15 FS_NAND_WritePageRaw()

Description

Stores data to a page of a NAND flash without ECC.

Prototype

```
int FS_NAND_WritePageRaw(      U8         Unit,
                               U32        PageIndex,
                         const void     * pData,
                               unsigned   NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
PageIndex	Index of the page to be written.
pData	in Data to be written.
NumBytes	Number of bytes to be written.

Return value

= 0 OK, data written.
≠ 0 An error occurred.

Additional information

This function is optional.

The data is written beginning at the byte offset 0 in the page. If more data is written than the size of the page + spare area, typically 2 Kbytes + 64 bytes, the excess bytes are ignored.

6.3.2.6.16 FS_NAND_DISK_INFO

Description

Information about the NAND partition.

Type definition

```
typedef struct {
  U32  NumPhyBlocks;
  U32  NumLogBlocks;
  U32  NumUsedPhyBlocks;
  U32  NumBadPhyBlocks;
  U32  NumPagesPerBlock;
  U32  NumSectorsPerBlock;
  U32  BytesPerPage;
  U32  BytesPerSpareArea;
  U32  BytesPerSector;
  U32  EraseCntMin;
  U32  EraseCntMax;
  U32  EraseCntAvg;
  U8   BadBlockMarkingType;
  U8   IsWriteProtected;
  U8   HasFatalError;
  U8   ErrorType;
  U32  ErrorSectorIndex;
  U16  BlocksPerGroup;
  U32  NumWorkBlocks;
  U8   IsFormatted;
} FS_NAND_DISK_INFO;
```

Structure members

Member	Description
NumPhyBlocks	Total number of blocks in the NAND partition.
NumLogBlocks	Total number of NAND blocks that can be used to store data.
NumUsedPhyBlocks	Number of NAND blocks that store valid data.
NumBadPhyBlocks	Number of NAND blocks that are marked as defective.
NumPagesPerBlock	Number of pages in a NAND block.
NumSectorsPerBlock	Number of logical sectors stored in a NAND block.
BytesPerPage	Number of bytes in the main area of a NAND page.
BytesPerSpareArea	Number of bytes in the spare area of a NAND page.
BytesPerSector	Number of bytes is a logical sector.
EraseCntMin	Minimum value of the erase counts in the NAND partition.
EraseCntMax	Maximum value of the erase counts in the NAND partition.
EraseCntAvg	Average value of the erase counts of all the NAND blocks in the NAND partition.
BadBlockMarkingType	Type of the bad block marking.
IsWriteProtected	Set to 1 if the NAND flash device cannot be written.
HasFatalError	Set to 1 if the SLC1 NAND driver reported a fatal error.
ErrorType	Type of fatal error that has occurred.
ErrorSectorIndex	Sector index on which a fatal error occurred.
BlocksPerGroup	Number of NAND blocks in a group.
NumWorkBlocks	Number of work blocks used by the SLC1 NAND driver.
IsFormatted	Set to 1 if the NAND partition has been low-level formatted.

Additional information

Refer to *Bad block marking types* on page 414 for a list of permitted values for `BadBlock-MarkingType`.

6.3.2.6.17 FS_NAND_BLOCK_INFO

Description

Information about a NAND block.

Type definition

```
typedef struct {
  U32          EraseCnt;
  U32          lbi;
  U16          NumSectorsBlank;
  U16          NumSectorsValid;
  U16          NumSectorsInvalid;
  U16          NumSectorsECCError;
  U16          NumSectorsECCCorrectable;
  U16          NumSectorsErrorInECC;
  U8           IsDriverBadBlock;
  U8           BadBlockErrorType;
  U16          BadBlockErrorBRSI;
  U8           Type;
  const char * sType;
} FS_NAND_BLOCK_INFO;
```

Structure members

Member	Description
EraseCnt	Number of times the block has been erased
lbi	Logical block index assigned to queried physical block.
NumSectorsBlank	Sectors are not used yet.
NumSectorsValid	Sectors contain correct data.
NumSectorsInvalid	Sectors have been invalidated.
NumSectorsECCError	Sectors have incorrect ECC.
NumSectorsECCCorrectable	Sectors have correctable ECC error.
NumSectorsErrorInECC	Sectors have error in ECC.
IsDriverBadBlock	Set to 1 if the block has been marked as bad by the driver.
BadBlockErrorType	Type of the error that caused the block to be marked as defective.
BadBlockErrorBRSI	Block-relative sector index on which the fatal error occurred that caused the block to be marked as defective.
Type	Type of data stored in the block.
sType	Zero-terminated string holding the block type.

Additional information

Refer to *NAND block types* on page 419 for a list of permitted values for Type.

IsDriverBadBlock is valid only when Type is set to FS_NAND_BLOCK_TYPE_BAD.

6.3.2.7 Performance and resource usage

This section provides information about the ROM and RAM usage as well as the performance of the SLC1 NAND driver. Each SLC1 NAND driver instance requires one instance of one NAND physical layer in order to operate. The resource usage of the used NAND physical layer has to be taken into account when calculating the total resource usage of the SLC1 NAND driver.

6.3.2.7.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the SLC1 NAND driver was measured using the SEGGER Embedded Studio IDE V4.20 configured to generate code for a Cortex-M4 CPU in Thumb mode and with the size optimization enabled.

Usage: 4.5 Kbytes

6.3.2.7.2 Static RAM usage

Static RAM usage refers to the amount of RAM required by the SLC1 NAND driver internally for all the driver instances. The number of bytes can be seen in the compiler list file of the {FS_NAND_Drv.c} file.

Usage: 32 bytes

6.3.2.7.3 Dynamic RAM usage

Dynamic RAM usage is the amount of RAM allocated by the driver at runtime. The amount RAM required depends on the runtime configuration and on the characteristics of the used NAND flash device. The approximate RAM usage of the SLC1 NAND driver can be calculated as follows:

```
MemAllocated = 160 + 2 * NumberOfUsedBlocks + 4 * SectorsPerBlock
               + 1.04 * MaxSectorSize
```

Parameter	Description
MemAllocated	Number of bytes allocated for one instance of the NAND driver
NumberOfUsedBlocks	Number of NAND blocks used for data storage
SectorsPerBlock	Number of logical sectors that can be stored in a NAND block
MaxSectorSize	Size of a logical sector in bytes

Example 1

2 GBit NAND flash with 2K pages, 2048 blocks used, 512-byte sectors, one NAND 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

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           = 6642 bytes
```

Example 3

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
```

6.3.2.7.4 Performance

The following performance measurements are in no way complete, but they give a good approximation of time required for common operations on various target hardware. The tests were performed as described in *Performance* on page 992. All values are given in Mbytes/sec.

CPU type	NAND flash device	Write speed	Read speed
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**
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.	**5.0**	**5.9**
Atmel AT91SAM3U (96 MHz)	NAND flash with 2048 bytes per page and a sector size of 512	**2.3**	**4.7**

CPU type	NAND flash device	Write speed	Read speed
	bytes using the built-in NAND controller/external bus-interface.		
Atmel AT91SAM9261 (200 MHz)	AT45DB64 DataFlash with a sector size of 1024 bytes using the SPI interface.	**0.16**	**0.67**

6.3.3 Universal NAND driver

Universal NAND driver is a very efficient device driver that was designed to support NAND flash devices that are using Single-Level Cell (SLC) as well Multi-Level Cell (MLC) technology. It requires a relatively small amount of RAM to operate and it can correct multiple bit errors by using the internal ECC of a NAND flash device or by calling ECC calculation routines provided by the application. The ECC protects the sector data and the driver management data stored in the spare area of a page which provides better data reliability. The Universal NAND driver uses a logical sector size that is equal to page size of the used NAND flash device. The NAND flash device must have a page of least 2048 bytes. Smaller logical sector sizes can be used with the help of an additional file system layer such as *Sector Size Adapter driver* on page 897.

6.3.3.1 Supported hardware

In general, the Universal NAND driver supports almost all popular Single-Level Cell NAND flashes (SLC) with a page size larger than 2048+64 bytes. The table below shows the NAND flash devices that have been tested or are compatible with a tested device:

Device	Page size (bytes)	Capacity (bits)	HW ECC
Hynix			
HY27UF082G2M	2048+64	256Mx8	no
HY27UF084G2M	2048+64	512Mx8	no
HY27UG084G2M	2048+64	512Mx8	no
HY27UG084GDM	2048+64	512Mx8	no
ISSI			
IS37SML01G1	2048+64	128Mx1	yes
IS38SML01G1	2048+64	128Mx1	yes
Macronix			
MX30LF1GE8AB	2048+64	128Mx8	yes
MX30LF2GE8AB	2048+64	256Mx8	yes
MX30LF4GE8AB	2048+64	512Mx8	yes
MX35LF1GE4AB	2048+64	128Mx1	yes
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	256Mx8	yes
MT29F8G08ABABA	4069+224	512Mx8	no
MT29F1G01AAADD	2048+64	1Gx1	yes
MT29F2G01AAAED	2048+64	2Gx1	yes
MT29F4G01AAADD	2048+64	4Gx1	yes
MT29F1G08ABAFA	2048+128	1Gx1	yes
MT29F1G01ABAFD	2048+128	1Gx1	yes
MT29F2G01ABAGD	2048+128	1Gx1	yes
MT29F8G08ABACA	4069+224	8Gx8	no
Samsung			

Device	Page size (bytes)	Capacity (bits)	HW ECC
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
Cypress (Spansion)			
S34ML01G1	2048+64	128Mx8	no
S34ML02G1	2048+64	256Mx8	no
S34ML04G1	2048+64	512Mx8	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
Kioxia (Toshiba)			
TC58BVG0S3HTAI0	2048+64	128Mx8	yes
TC58CVG1S3HxAIx	2048+64	2Gx1	yes
TC58CYG2S0HRAIG	4096+128/256	4Gx1	yes
Winbond			
W25N01GVxxIG/IT	2048+64	1Gx1	yes

Support for devices not in the 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.

6.3.3.2 Theory of operation

NAND flash devices are divided into physical blocks and physical pages. One physical block is the smallest erasable unit while 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 Universal NAND driver reserves the first page of a block for management data such as erase count. The Universal NAND driver uses the spare bytes for the following purposes:

1. To check if the block is valid.
   If a block is valid then the driver uses it for data storage. When the driver detects a uncorrectable bit error in a page, the entire block containing that page 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 and load an ECC (Error Correction Code) for data reliability.
   When reading a page, the driver also reads the ECC stored in the spare area of that page, 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 of that page.

6.3.3.3 Support for custom hardware

It is possible to use the Universal NAND driver with a 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 and normally no changes to the physical layer are required. If the NAND driver has to 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 all the hardware accesses.

6.3.3.4 Partial write operations

The Universal NAND driver writes only once in any page of the NAND flash between two erase operations of the NAND block that contain that page. Therefore, the number of partial write operation is one which makes the driver compatible with any SLC and MLC NAND flash device.

6.3.3.5 High-reliability operation

The Universal NAND driver supports a feature that can be used to increase the reliability of the data stored to a NAND flash device. It works by checking the number of bit errors corrected by the ECC during each page read operation. If the number of bit errors is equal to or greater than a configured threshold the Universal NAND driver copies the contents of the NAND block containing the read page to an other location. This prevents the accumulation of bit errors that can cause ECC correction errors. An ECC correction error occurs when the number of bit errors is larger than the number of bit errors the ECC is able to correct. Typically, a correction error leads to a data loss.

This feature requires support for the `FS_NAND_PHY_TYPE_GET_ECC_RESULT` function in the NAND physical layer. By default, the feature is disabled for performance reasons and has to be explicitly enabled at compile time by setting `FS_NAND_MAX_BIT_ERROR_CNT` to a value greater than 0. This value can be modified at runtime by calling `FS_NAND_UNI_SetMaxBitErrorCnt()`.

> **Note**
>
> Enabling the high-reliability operation can reduce the specified lifetime of the NAND flash due to the increased number of erase cycles.

6.3.3.6 Configuring the driver

This section describes how to configure the file system to make use of the SLC1 NAND driver.

6.3.3.6.1 Runtime configuration

The driver must be added to the file system by calling `FS_AddDevice()` with the driver identifier set to the address of `FS_NAND_UNI_Driver`. This function call together with other function calls that configure the driver operation have to be added to `FS_X_AddDevices()` as demonstrated in the following example.

Example

```
#include "FS.h"
#include "FS_NAND_HW_Template.h"

#define ALLOC_SIZE      0x9000          // Memory pool for the file system in bytes

static U32 _aMemBlock[ALLOC_SIZE / 4];  // Memory pool used for semi-dynamic allocation.

void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add the NAND flash driver to file system. Volume name: "nand:0:".
```



```
  //
  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);
  //
  // Configure the HW access routines.
  //
  FS_NAND_ONFI_SetHWType(0, &FS_NAND_HW_Template);
#if FS_SUPPORT_FILE_BUFFER
  //
  // Enable the file buffer to increase the performance
  // when reading/writing a small number of bytes.
  //
  FS_ConfigFileBufferDefault(512, FS_FILE_BUFFER_WRITE);
#endif // FS_SUPPORT_FILE_BUFFER
}
```

The API functions listed in the next table can be used by the application to configure the behavior of the Universal NAND driver. The application can call them only at the file system initialization in FS_X_AddDevices().

Function	Description
FS_NAND_UNI_AllowBlankUnusedSectors()	Configures if the data of unused sectors has to be initialized.
FS_NAND_UNI_AllowReadErrorBadBlocks()	Configures if a block is marked as defective on read fatal error.
FS_NAND_UNI_SetBlockRange()	Specifies which NAND blocks the driver can use to store the data.
FS_NAND_UNI_SetBlockReserve()	Configures the number of NAND flash blocks to be reserved as replacement for the NAND flash blocks that become defective.
FS_NAND_UNI_SetCleanThreshold()	Specifies the minimum number sectors that the driver should keep available for fast write operations.
FS_NAND_UNI_SetDriverBadBlockReclamation()	Configures if the bad blocks marked as defective by the driver have to be erased at low-level format or not.
FS_NAND_UNI_SetECCHook()	Configures the ECC algorithm to be used for the correction of bit errors.
FS_NAND_UNI_SetEraseVerification()	Enables or disables the checking of the block erase operation.
FS_NAND_UNI_SetMaxBitErrorCnt()	Configures the number of bit errors that trigger the relocation of the data stored in a NAND block.
FS_NAND_UNI_SetMaxEraseCntDiff()	Configures the threshold of the wear leveling procedure.
FS_NAND_UNI_SetNumBlocksPerGroup()	Specifies the number of physical NAND blocks in a virtual block.
FS_NAND_UNI_SetNumWorkBlocks()	Sets number of work blocks the Universal NAND driver uses for write operations.

Function	Description
FS_NAND_UNI_SetOnFatalErrorCallback()	Registers a function to be called by the driver when a fatal error occurs.
FS_NAND_UNI_SetPhyType()	Configures NAND flash access functions.
FS_NAND_UNI_SetWriteDisturbHandling()	Configures if the bit errors caused by write operations are handled or not.
FS_NAND_UNI_SetWriteVerification()	Enables or disables the checking of each page write operation.

6.3.3.6.1.1 FS_NAND_UNI_AllowBlankUnusedSectors()

Description

Configures if the data of unused sectors has to be initialized.

Prototype

```
void FS_NAND_UNI_AllowBlankUnusedSectors(U8 Unit,
                                         U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of driver instance (0-based).
OnOff	Specifies if the feature has to be enabled or disabled • =0 Sector data is filled with 0s. • ≠0 Sector data is not initialized (all bytes remain set to 0xFF).

Additional information

This function is optional. The default behavior of the Universal NAND driver is to fill the unused sectors with 0s. This done in order to reduce the chance of a bit error caused by an unwanted transition from 1 to 0 of the value stored to memory cell.

Some NAND flash devices may wear out faster than expected if excessive number of bytes are set to 0. This limitation is typically documented in the data sheet of the corresponding NAND flash. For such devices it is recommended configure the Universal NAND driver to do not initialize the unused sectors. This can be realized by calling FS_NAND_UNI_Allow-BlankUnusedSectors() with the OnOff parameter set to 1 in FS_X_AddDevices().

6.3.3.6.1.2 FS_NAND_UNI_AllowReadErrorBadBlocks()

Description

Configures if a block is marked as defective on read fatal error.

Prototype

```
void FS_NAND_UNI_AllowReadErrorBadBlocks(U8 Unit,
                                         U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of driver instance (0-based).
OnOff	Specifies if the feature has to be enabled or disabled • =0 The block is not marked as defective. • ≠0 The block is marked as defective.

Additional information

This function is optional. The default behavior of the Universal NAND driver is to mark a block as defective if a read error or and uncorrectable bit error occurs while reading data from a page of that block.

6.3.3.6.1.3 FS_NAND_UNI_SetBlockRange()

Description

Specifies which NAND blocks the driver can use to store the data.

Prototype

```
void FS_NAND_UNI_SetBlockRange(U8  Unit,
                               U16 FirstBlock,
                               U16 MaxNumBlocks);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
FirstBlock	Index of the first NAND flash block to be used as storage (0-based).
MaxNumBlocks	Maximum number of NAND flash blocks to be used as storage.

Additional information

This function is optional. By default, the Universal NAND driver uses all blocks of the NAND flash as data storage. FS_UNI_NAND_SetBlockRange() is useful when a part of the NAND flash has to be used for another purpose, for example to store the application program used by a boot loader, and therefore it cannot be managed by the Universal NAND driver. Limiting the number of blocks used by the Universal NAND driver can also help reduce the RAM usage.

FirstBlock is the index of the first physical NAND block were 0 is the index of the first block of the NAND flash device. MaxNumBlocks can be larger that the actual number of available NAND blocks in which case the Universal NAND driver silently truncates the value to reflect the actual number of NAND blocks available.

The Universal NAND driver uses the first NAND block in the range to store management information at low-level format. If the first NAND block happens to be marked as defective, then the next usable NAND block is used.

If the FS_UNI_NAND_SetBlockRange() is used to subdivide the same physical NAND flash device into two or more partitions than the application has to make sure that the created partitions do not overlap.

The read optimization of the FS_NAND_PHY_2048x8 physical layer has to be disabled when this function is used to partition the NAND flash device in order to ensure data consistency. The read cache can be disabled at runtime via FS_NAND_2048x8_DisableReadCache().

6.3.3.6.1.4 FS_NAND_UNI_SetBlockReserve()

Description

Configures the number of NAND flash blocks to be reserved as replacement for the NAND flash blocks that become defective.

Prototype

```
void FS_NAND_UNI_SetBlockReserve(U8       Unit,
                                 unsigned pctOfBlocks);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
pctOfBlocks	Number of NAND flash blocks to be reserved as percentage of the total number of NAND flash blocks.

Additional information

This function is optional. The Universal NAND driver reserves by default about 3% of the total number of NAND flash blocks which is sufficient for most applications. Reserving more NAND flash blocks can increase the lifetime of the NAND flash device. The NAND flash device has to be low-level formatted after changing the number of reserved blocks.

6.3.3.6.1.5 FS_NAND_UNI_SetCleanThreshold()

Description

Specifies the minimum number sectors that the driver should keep available for fast write operations.

Prototype

```
int FS_NAND_UNI_SetCleanThreshold(U8       Unit,
                                  unsigned NumBlocksFree,
                                  unsigned NumSectorsFree);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
NumBlocksFree	Number of blocks to be kept free.
NumSectorsFree	Number of sectors to be kept free on each block.

Return value

= 0 OK, threshold has been set.
≠ 0 An error occurred.

Additional information

This function is optional. It can be used by the application to prepare the NAND flash device to write data as fast as possible once when an event occurs such an unexpected reset. At the startup, the application reserves free space, by calling the function with NumBlocksFree and NumSectorsFree set to a value different than 0. The number of free sectors depends on the number of bytes written and on how the file system is configured. When the unexpected reset occurs, the application tells the driver that it can write to the free sectors, by calling the function with NumBlocksFree and NumSectorsFree set to 0. Then, the application writes the data to file and the NAND driver stores it to the free space. Since no erase or copy operation is required, the data is written as fastest as the NAND flash device permits it.

The NAND flash device will wear out faster than normal if sectors are reserved in a work block (NumSectors > 0).

Example

The following sample code demonstrates how to determine the threshold parameters.

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

void SampleNAND_UNISetCleanThreshold(void) {
  unsigned             NumPhyBlocks;
  unsigned             NumWorkBlocks;
  FS_FILE            * pFile;
  FS_NAND_DISK_INFO    DiskInfo;
  FS_NAND_BLOCK_INFO   BlockInfo;
  U32                  BlockIndex;
  unsigned             MaxNumSectors;
  unsigned             NumSectorsValid;
  //
  // Get information about the storage.
  //
  memset(&DiskInfo, 0, sizeof(DiskInfo));
  FS_NAND_UNI_GetDiskInfo(0, &DiskInfo);
  NumPhyBlocks  = DiskInfo.NumPhyBlocks;
  NumWorkBlocks = DiskInfo.NumWorkBlocks;
  //
  // Make sure that all work blocks are converted to data blocks.
  //
  FS_NAND_UNI_Clean(0, NumWorkBlocks, 0);
```



```
  //
  // Create a test file, write to file the data
  // to be saved on sudden reset then close it.
  //
  pFile = FS_FOpen("Test.txt", "w");
  FS_Write(pFile, "Reset", 5);
  FS_FClose(pFile);
  //
  // Calculate how many work blocks were written
  // and the maximum number of sectors written in a work block.
  //
  NumWorkBlocks = 0;
  BlockIndex    = 0;
  MaxNumSectors = 0;
  do {
    memset(&BlockInfo, 0, sizeof(BlockInfo));
    FS_NAND_UNI_GetBlockInfo(0, BlockIndex++, &BlockInfo);
    if (BlockInfo.Type == FS_NAND_BLOCK_TYPE_WORK) {
      ++NumWorkBlocks;
      NumSectorsValid = BlockInfo.NumSectorsValid;
      if (MaxNumSectors < NumSectorsValid) {
        MaxNumSectors = NumSectorsValid;
      }
    }
  } while (--NumPhyBlocks);
  //
  // Clean up: remove the test file.
  //
  FS_Remove("Test.txt");
  //
  // Show the calculated values.
  // NumWorkBlocks and MaxNumSectors can be passed as 2nd and 3rd
  // argument in the call to FS_NAND_UNI_SetWorkBlockReserve().
  //
  printf("Number of work blocks used:  %u\n", NumWorkBlocks);
  printf("Max. number of used sectors: %u\n", MaxNumSectors);
}
```

Sample output

```
Number of work blocks used:  1
Max. number of used sectors: 5
```


6.3.3.6.1.6 FS_NAND_UNI_SetDriverBadBlockReclamation()

Description

Configures if the bad blocks marked as defective by the driver have to be erased at low-level format or not.

Prototype

```
void FS_NAND_UNI_SetDriverBadBlockReclamation(U8 Unit,
                                              U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
OnOff	Configures if the feature has to be enable or disabled. • =1 Defective blocks marked as such by the driver are erased at low-level format. • =0 Defective blocks marked as such by the driver are not erased at low-level format.

Additional information

This function is active only when the option FS_NAND_RECLAIM_DRIVER_BAD_BLOCKS is set to 1. The default behavior is to erase the blocks marked as defective by the driver.

6.3.3.6.1.7 FS_NAND_UNI_SetECCHook()

Description

Configures the ECC algorithm to be used for the correction of bit errors.

Prototype

```
void FS_NAND_UNI_SetECCHook(      U8                  Unit,
                            const FS_NAND_ECC_HOOK * pECCHook);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based)
pECCHook	in ECC algorithm.

Additional information

This function is optional. By default, the Universal NAND driver uses an software algorithm that is capable of correcting 1 bit errors and of detecting 2 bit errors. If the NAND flash device requires a better error correction, the application has to specify an different ECC algorithm via FS_NAND_UNI_SetECCHook().

The ECC algorithms can be either implemented in the software or the calculation routines can take advantage of any dedicated ECC HW available on the target system.

The following ECC algorithms are supported:

Type	Description
FS_NAND_ECC_HW_NULL	This is a pseudo ECC algorithm that requests the Universal NAND driver to use the internal HW ECC of a NAND flash device.
FS_NAND_ECC_HW_4BIT	This is a pseudo ECC algorithm that requests the Universal NAND driver to use HW 4 bit error correction. It can be used for example with dedicated a NAND flash controller that comes with a configurable HW ECC.
FS_NAND_ECC_HW_8BIT	This is a pseudo ECC algorithm that requests the Universal NAND driver to use HW 8 bit error correction. It can be used for example with dedicated a NAND flash controller that comes with a configurable HW ECC.
FS_NAND_ECC_SW_1BIT	This is an software algorithm that is able to correct 1 bit errors and detect 2 bit errors. More specifically it can correct a 1 bit error per 256 bytes of data and a 1 bit error per 4 bytes of spare area.

Additional software algorithms with error correction capabilities greater than 1 are supported via the SEGGER emLib ECC component (www.segger.com/products/security-iot/emlib/variations/ecc/).

6.3.3.6.1.8 FS_NAND_UNI_SetEraseVerification()

Description

Enables or disables the checking of the block erase operation.

Prototype

```
void FS_NAND_UNI_SetEraseVerification(U8 Unit,
                                      U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
OnOff	Specifies if the feature has to be enabled or disabled • =0 The erase operation is not checked. • ≠0 The erase operation is checked.

Additional information

This function is optional. The result of a block erase operation is normally checked by evaluating the error bits maintained by the NAND flash device in a internal status register. FS_NAND_UNI_SetEraseVerification() can be used to enable additional verification of the block erase operation that is realized by reading back the contents of the entire erased physical block and by checking that all the bytes in it are set to 0xFF. Enabling this feature can negatively impact the write performance of Universal NAND driver.

The block erase verification feature is active only when the Universal NAND driver is compiled with the FS_NAND_VERIFY_ERASE configuration define is set to 1 (default is 0) or when the FS_DEBUG_LEVEL configuration define is set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL.

6.3.3.6.1.9 FS_NAND_UNI_SetMaxBitErrorCnt()

Description

Configures the number of bit errors that trigger the relocation of the data stored in a NAND block.

Prototype

```
void FS_NAND_UNI_SetMaxBitErrorCnt(U8       Unit,
                                   unsigned BitErrorCnt);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
BitErrorCnt	Number of physical bit errors.

Additional information

This function is optional and is active only when the file system is compiled with FS_NAND_MAX_BIT_ERROR_CNT set to a value greater than 0.

FS_NAND_UNI_SetMaxBitErrorCnt() can be used to configure when the Universal NAND Driver has to copy the contents of a NAND block to another location in order to prevent the accumulation of bit errors. BitErrorCnt has to be smaller than or equal to the bit error correction requirement of the NAND flash device. The feature can be disabled by calling FS_NAND_UNI_SetMaxBitErrorCnt() with BitErrorCnt set to 0.

The lifetime of the NAND flash device can be negatively affected if this feature is enabled due to the increased number of block erase operations.

6.3.3.6.1.10 FS_NAND_UNI_SetMaxEraseCntDiff()

Description

Configures the threshold of the wear leveling procedure.

Prototype

```
void FS_NAND_UNI_SetMaxEraseCntDiff(U8  Unit,
                                    U32 EraseCntDiff);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
EraseCntDiff	Maximum allowed difference between the erase counts of any two NAND blocks.

Additional information

This function is optional. It can be used to control how the Universal NAND driver performs the wear leveling. The wear leveling procedure makes sure that the NAND blocks are equally erased to meet the life expectancy of the storage device by keeping track of the number of times a NAND block has been erased (erase count). The Universal NAND driver executes this procedure when a new empty NAND block is required for data storage. The wear leveling procedure works by first choosing the next available NAND block. Then the difference between the erase count of the chosen block and the NAND block with lowest erase count is computed. If this value is greater than EraseCntDiff the NAND block with the lowest erase count is freed and made available for use.

The same threshold can also be configured at compile time via the FS_NAND_MAX_ERASE_CNT_DIFF configuration define.

6.3.3.6.1.11 FS_NAND_UNI_SetNumBlocksPerGroup()

Description

Specifies the number of physical NAND blocks in a virtual block.

Prototype

```
int FS_NAND_UNI_SetNumBlocksPerGroup(U8       Unit,
                                     unsigned BlocksPerGroup);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
BlocksPerGroup	Number of physical blocks in a virtual block. The value must be a power of 2.

Return value

= 0 OK, value configured.
≠ 0 Error code indicating the failure reason.

Additional information

This function is optional. It can be used to specify how many physical NAND blocks are grouped together to form a virtual block. Grouping physical blocks helps reduce the RAM usage of the driver when the NAND flash device contains a large number of physical blocks. For example, the dynamic RAM usage of the Universal NAND driver using a NAND flash device with 8196 blocks and a page size of 2048 bytes is about 20 KB. The dynamic RAM usage is reduced to about 7 KB if 4 physical blocks are grouped together.

The FS_NAND_SUPPORT_BLOCK_GROUPING configuration define has to be set to 1 in order to enable this function. The FS_NAND_SUPPORT_BLOCK_GROUPING configuration define is set to 1 by default. When set to 0 the function returns an error if called in the application.

FS_NAND_UNI_SetNumBlocksPerGroup() is optional and may be called only from FS_X_AddDevices(). Changing the block grouping requires a low-level format of the NAND flash device.

6.3.3.6.1.12 FS_NAND_UNI_SetNumWorkBlocks()

Description

Sets number of work blocks the Universal NAND driver uses for write operations.

Prototype

```
void FS_NAND_UNI_SetNumWorkBlocks(U8  Unit,
                                  U32 NumWorkBlocks);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
NumWorkBlocks	Number of work blocks.

Additional information

This function is optional. It can be used to change the default number of work blocks according to the requirements of the application. Work blocks are physical NAND blocks that the Universal NAND driver uses to temporarily store the data written to NAND flash device. The Universal NAND driver calculates at low-level format the number of work blocks based on the total number of blocks available on the NAND flash device.

By default, the NAND driver allocates 10% of the total number of NAND blocks used as storage, but no more than 10 NAND blocks. The minimum number of work blocks allocated by default depends on whether journaling is used or not. If the journal is active 4 work blocks are allocated, else Universal NAND driver allocates 3 work blocks. The currently allocated number of work blocks can be checked via FS_NAND_UNI_GetDiskInfo(). The value is returned in the NumWorkBlocks member of the FS_NAND_DISK_INFO structure.

Increasing the number of work blocks can help increase the write performance of the Universal NAND driver. At the same time the RAM usage of the Universal NAND driver increases since each configured work block requires a certain amount of RAM for the data management. This is a trade-off between write performance and RAM usage.

The new value take effect after the NAND flash device is low-level formatted via the FS_FormatLow() API function.

6.3.3.6.1.13 FS_NAND_UNI_SetOnFatalErrorCallback()

Description

Registers a function to be called by the driver when a fatal error occurs.

Prototype

```
void FS_NAND_UNI_SetOnFatalErrorCallback
                                (FS_NAND_ON_FATAL_ERROR_CALLBACK * pfOnFatalError);
```

Parameters

Parameter	Description
pfOnFatalError	Function to be called when a fatal error occurs.

Additional information

This function is optional.

6.3.3.6.1.14 FS_NAND_UNI_SetPhyType()

Description

Configures NAND flash access functions.

Prototype

```
void FS_NAND_UNI_SetPhyType(      U8                  Unit,
                            const FS_NAND_PHY_TYPE * pPhyType);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based)
pPhyType	in Physical layer to be used to access the NAND flash device.

Additional information

This function is mandatory and it has to be called in FS_X_AddDevices() once for each instance of the Universal NAND driver. The driver instance is identified by the Unit parameter. First Universal NAND driver instance added to the file system via a FS_AddDevice(&FS_NAND_UNI_Driver) call has the unit number 0, the Universal NAND driver added by a second call to FS_AddDevice() has the unit number 1 and so on.

6.3.3.6.1.15 FS_NAND_UNI_SetWriteDisturbHandling()

Description

Configures if the bit errors caused by write operations are handled or not.

Prototype

```
void FS_NAND_UNI_SetWriteDisturbHandling(U8 Unit,
                                         U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
OnOff	Activation status of the feature. • =0 Write disturb errors are not handled (default). • ≠0 Write disturb errors are handled.

Additional information

This function is optional and is active only when the file system is compiled with FS_NAND_MAX_BIT_ERROR_CNT set to a value greater than 0.

A write operation can cause bit errors in the pages located on the same NAND block with the page being currently written. Normally, these bit errors are corrected later when the data of the NAND block is copied internally by the Universal NAND driver to another location on the NAND flash device during a wear leveling, a garbage collection or a write operation. This is the default behavior of the Universal NAND driver.

The Universal NAND driver is also able to check for and correct any bit errors right a after the write operation in order to reduce the accumulation of bit errors that can lead to a data loss. This error handling mode can be enabled by calling FS_NAND_UNI_SetWriteDisturb-Handling() with the OnOff parameter set to 1. In this error handling mode if the number of bit errors in any page in the checked NAND block is greater than or equal to the value configured via FS_NAND_UNI_SetMaxBitErrorCnt() then the NAND block is copied to another location on the NAND flash device in order to correct the bit errors.

The lifetime of the NAND flash device can be negatively affected if this feature is enabled due to the increased number of block erase operations. The write performance can be negatively affected as well.

6.3.3.6.1.16 FS_NAND_UNI_SetWriteVerification()

Description

Enables or disables the checking of each page write operation.

Prototype

```
void FS_NAND_UNI_SetWriteVerification(U8 Unit,
                                      U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
OnOff	Specifies if the feature has to be enabled or disabled • =0 The write operation is not checked. • ≠0 The write operation is checked.

Additional information

This function is optional. The result of a page write operation is normally checked by evaluating the error bits maintained by the NAND flash device in a internal status register. FS_NAND_UNI_SetWriteVerification() can be used to enable additional verification of the page write operation that is realized by reading back the contents of the written page and by checking that all the bytes are matching the data requested to be written. Enabling this feature can negatively impact the write performance of Universal NAND driver.

The page write verification feature is active only when the Universal NAND driver is compiled with the FS_NAND_VERIFY_WRITE configuration define is set to 1 (default is 0) or when the FS_DEBUG_LEVEL configuration define is set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL.

6.3.3.6.1.17 FS_NAND_ECC_HOOK

Description

ECC calculation algorithm.

Type definition

```
typedef struct {
  FS_NAND_ECC_HOOK_CALC  * pfCalc;
  FS_NAND_ECC_HOOK_APPLY * pfApply;
  U8                       NumBitsCorrectable;
  U8                       ldBytesPerBlock;
} FS_NAND_ECC_HOOK;
```

Structure members

Member	Description
pfCalc	Calculates the ECC of specified data.
pfApply	Verifies and corrects bit errors using ECC.
NumBitsCorrectable	Number of bits the ECC algorithm is able to correct in the data block and 4 bytes of spare area.
ldBytesPerBlock	Number of bytes in the data block given as power of 2 value.

Additional information

This structure contains pointers to API functions and attributes related to an ECC calculation algorithm.

ldBytesPerBlock is typically set to 9 which indicates a block size of 512 bytes. The size of the ECC block is imposed by the specifications of the NAND flash device. If the value is set to 0 the Universal NAND driver assumes that the ECC block is 512 bytes large.

NumBitsCorrectable is used by the Universal NAND driver to check if the ECC algorithm is capable of correcting the number of bit errors required by the used NAND flash device.

6.3.3.6.1.18 FS_NAND_ECC_HOOK_CALC

Description

Calculates the parity bits of the specified data using ECC.

Type definition

```
typedef void FS_NAND_ECC_HOOK_CALC(const U32 * pData,
                                   U8  * pSpare);
```

Parameters

Parameter	Description
pData	in Data to be written to the main area of the page.
pSpare	in Data to be written to the spare area of the page. out The calculated parity bits.

Additional information

FS_NAND_ECC_HOOK_CALC is called by the Universal NAND driver before it writes the data to the NAND flash device to calculate the parity check bits. The parity check bits are stored together with the data to NAND flash device. They are used by FS_NAND_ECC_HOOK_CALC to correct eventual bit errors.

This function has to calculate the parity bits of pData and of four bytes of pSpare. That is the calculated parity bits are used to protect the data stored in pData as well as pSpare. The number of bytes in pData to be protected by ECC is specified via ldBytesPerBlock in FS_NAND_ECC_HOOK. The data in pSpare to be protected by ECC is located at byte offset four. The calculated parity bits must be stored to pSpare at byte offset eight. The byte order of the stored parity bits is not relevant for the Universal NAND driver. pSpare is organized as follows:

Byte range	Description
0-3	Not protected by ECC
4-7	Data to be protected by ECC
8-N	Parity check bits calculated via the ECC algorithm

FS_NAND_ECC_HOOK_CALC is not allowed to store more than N - 8 + 1 parity check bytes at byte offset eight to pSpare. N depends on the size of the main and spare are of the NAND flash and on ldBytesPerBlock and can be calculate using this formula:

```
N = ((BytesPerSpareArea / (BytesPerPage >> ldBytesPerBlock)) - 8) - 1
```

Parameter	Description
BytesPerSpareArea	Number of bytes in the spare area of the NAND flash device
BytesPerPage	Number of bytes in the spare area of the NAND flash device
ldBytesPerBlock	Value specified in FS_NAND_ECC_HOOK

6.3.3.6.1.19 FS_NAND_ECC_HOOK_APPLY

Description

Checks and corrects bit errors in the specified data using ECC.

Type definition

```
typedef int FS_NAND_ECC_HOOK_APPLY(U32 * pData,
                                   U8  * pSpare);
```

Parameters

Parameter	Description
pData	in Data read from the main area of the page. out Corrected main area data.
pSpare	in Data read from the spare area of the page. out Corrected spare area data.

Return value

< 0 Uncorrectable bit errors detected.
≥ 0 Number of bit errors detected and corrected.

Additional information

FS_NAND_ECC_HOOK_APPLY is called by the Universal NAND driver after it reads data with from the NAND flash device to verify that the data is not corrupted. If the function detects bit errors then it uses the parity check bits to correct them. The correction has to be performed in place that is directly in pData and pSpare. The parity check bits are located at byte offset eight in pSpare. They protect all the bytes in pData and four bytes of pSpare. Refer to FS_NAND_ECC_HOOK_CALC for a description of pSpare data layout. FS_NAND_ECC_HOOK_APPLY function has to also correct the bit errors that occurred in the parity check.

For backwards compatibility the return value is interpreted differently by the Universal NAND driver if ldBytesPerBlock is set to 0 as follows:

Value	Description
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.

6.3.3.7 Additional driver functions

The following functions are optional and can be used by the application to perform operations directly on the NAND flash device.

Function	Description
FS_NAND_UNI_Clean()	Makes storage space available for fast write operations.
FS_NAND_UNI_EraseBlock()	Sets all the bytes in a NAND block to 0xFF.
FS_NAND_UNI_EraseFlash()	Erases the entire NAND partition.
FS_NAND_UNI_GetBlockInfo()	Returns information about the specified NAND block.
FS_NAND_UNI_GetDiskInfo()	Returns information about the NAND partition.
FS_NAND_UNI_GetStatCounters()	Returns the actual values of statistical counters.
FS_NAND_UNI_IsBlockBad()	Checks if a NAND block is marked as defective.
FS_NAND_UNI_ReadLogSectorPartial()	Reads a specified number of bytes from a logical sector.
FS_NAND_UNI_ReadPageRaw()	Reads data from a page without ECC.
FS_NAND_UNI_ReadPhySector()	This function reads a physical sector from NAND flash.
FS_NAND_UNI_ResetStatCounters()	Sets the values of statistical counters to 0.
FS_NAND_UNI_TestBlock()	Fills all the pages in a block (including the spare area) with the specified pattern and verifies if the data was written correctly.
FS_NAND_UNI_WritePage()	Stores data to a page of a NAND flash with ECC.
FS_NAND_UNI_WritePageRaw()	Stores data to a page of a NAND flash without ECC.

6.3.3.7.1 FS_NAND_UNI_Clean()

Description

Makes storage space available for fast write operations.

Prototype

```
int FS_NAND_UNI_Clean(U8       Unit,
                      unsigned NumBlocksFree,
                      unsigned NumSectorsFree);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
NumBlocksFree	Number of blocks to be kept free.
NumSectorsFree	Number of sectors to be kept free on each block.

Return value

= 0 OK, space has been made available.
≠ 0 An error occurred.

Additional information

This function is optional. It can be used to free space on the NAND flash device for data that the application has to write as fast as possible. FS_NAND_UNI_Clean() performs two internal operations: (1) Converts all work blocks that have less free sectors than NumSectorsFree into data blocks. (2) If required, convert work blocks until at least NumBlocksFree are available.

6.3.3.7.2 FS_NAND_UNI_EraseBlock()

Description

Sets all the bytes in a NAND block to `0xFF`.

Prototype

```
int FS_NAND_UNI_EraseBlock(U8       Unit,
                           unsigned BlockIndex);
```

Parameters

Parameter	Description
`Unit`	Index of the driver instance (0-based).
`BlockIndex`	Index of the NAND flash block to be erased.

Return value

= 0 OK, block erased
≠ 0 An error occurred

Additional information

This function is optional. `FS_NAND_UNI_EraseBlock()` function does not check if the block is marked as defective before erasing it.

6.3.3.7.3 FS_NAND_UNI_EraseFlash()

Description

Erases the entire NAND partition.

Prototype

```
int FS_NAND_UNI_EraseFlash(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).

Return value

≥ 0 . Number of blocks which failed to erase.
< 0 An error occurred.

Additional information

This function is optional. After the call to this function all the bytes in the NAND partition are set to 0xFF.

This function has to be used with care, since it also erases blocks marked as defective and therefore the information about the block status will be lost. FS_NAND_EraseFlash() can be used without this side effect on storage devices that are guaranteed to not have any bad blocks, such as DataFlash devices.

6.3.3.7.4 FS_NAND_UNI_GetBlockInfo()

Description

Returns information about the specified NAND block.

Prototype

```
int FS_NAND_UNI_GetBlockInfo(U8                   Unit,
                             U32                  BlockIndex,
                             FS_NAND_BLOCK_INFO * pBlockInfo);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
BlockIndex	Index of the physical block to get information about.
pBlockInfo	out Information about the NAND block.

Return value

= 0 OK, information returned.
≠ 0 An error occurred.

Additional information

This function is not required for the functionality of the driver and will typically not be linked in production builds.

6.3.3.7.5 FS_NAND_UNI_GetDiskInfo()

Description

Returns information about the NAND partition.

Prototype

```
int FS_NAND_UNI_GetDiskInfo(U8                  Unit,
                            FS_NAND_DISK_INFO * pDiskInfo);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
pDiskInfo	out Information about the NAND partition.

Return value

= 0 OK, information returned.
≠ 0 An error occurred.

Additional information

This function is not required for the functionality of the Universal NAND driver and is typically not linked in production builds.

6.3.3.7.6 FS_NAND_UNI_GetStatCounters()

Description

Returns the actual values of statistical counters.

Prototype

```
void FS_NAND_UNI_GetStatCounters(U8                       Unit,
                                 FS_NAND_STAT_COUNTERS * pStat);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based)
pStat	out Values of statistical counters.

Additional information

This function is optional. It is active only when the file system is compiled with FS_DEBUG_LEVEL set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL or FS_NAND_ENABLE_STATS set to 1.

The statistical counters can be cleared via FS_NAND_UNI_ResetStatCounters().

6.3.3.7.7 FS_NAND_UNI_IsBlockBad()

Description

Checks if a NAND block is marked as defective.

Prototype

```
int FS_NAND_UNI_IsBlockBad(U8       Unit,
                           unsigned BlockIndex);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
BlockIndex	Index of the NAND flash block to be checked.

Return value

1 Block is defective
0 Block is not defective

Additional information

This function is optional.

6.3.3.7.8 FS_NAND_UNI_ReadLogSectorPartial()

Description

Reads a specified number of bytes from a logical sector.

Prototype

```
int FS_NAND_UNI_ReadLogSectorPartial(U8         Unit,
                                     U32        LogSectorIndex,
                                     void     * pData,
                                     unsigned   Off,
                                     unsigned   NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
LogSectorIndex	Index of the logical sector to read from.
pData	out Data read from NAND flash.
Off	Byte offset to read from (relative to beginning of the sector).
NumBytes	Number of bytes to be read.

Return value

≠ 0 Number of bytes read.
= 0 An error occurred.

Additional information

This function is optional.

For NAND flash devices with internal HW ECC only the specified number of bytes is transferred and not the entire sector. Typ. used by the applications that access the NAND flash directly (that is without a file system) to increase the read performance.

6.3.3.7.9 FS_NAND_UNI_ReadPageRaw()

Description

Reads data from a page without ECC.

Prototype

```
int FS_NAND_UNI_ReadPageRaw(U8         Unit,
                            U32        PageIndex,
                            void     * pData,
                            unsigned   NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
PageIndex	Index of the page to be read.
pData	out Data to be written.
NumBytes	Number of bytes to be read.

Return value

= 0 OK, data read.
≠ 0 An error occurred.

Additional information

This function is optional.

The data is read beginning from byte offset 0 in the page. If more data is requested than the page + spare area size, typ. 2 Kbytes + 64 bytes, the function does not modify the remaining bytes in pData.

FS_NAND_UNI_ReadPageRaw() does not work correctly on NAND flash devices with HW ECC that cannot be disabled.

6.3.3.7.10 FS_NAND_UNI_ReadPhySector()

Description

This function reads a physical sector from NAND flash.

Prototype

```
int FS_NAND_UNI_ReadPhySector(U8         Unit,
                              U32        PhySectorIndex,
                              void     * pData,
                              unsigned * pNumBytesData,
                              void     * pSpare,
                              unsigned * pNumBytesSpare);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
PhySectorIndex	Physical sector index.
pData	Pointer to a buffer to store read data.
pNumBytesData	in Pointer to variable storing the size of the data buffer. out The number of bytes that were stored in the data buffer.
pSpare	Pointer to a buffer to store read spare data.
pNumBytesSpare	in Pointer to variable storing the size of the spare data buffer. out The number of bytes that were stored in the spare data buffer.

Return value

≥ 0 OK, sector data read.
< 0 An error occurred.

Additional information

This function is optional.

6.3.3.7.11 FS_NAND_UNI_ResetStatCounters()

Description

Sets the values of statistical counters to 0.

Prototype

```
void FS_NAND_UNI_ResetStatCounters(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based)

Additional information

This function is optional. It is active only when the file system is compiled with FS_DEBUG_LEVEL set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL or FS_NAND_ENABLE_STATS set to 1.

The statistical counters can be queried via FS_NAND_UNI_GetStatCounters().

6.3.3.7.12 FS_NAND_UNI_TestBlock()

Description

Fills all the pages in a block (including the spare area) with the specified pattern and verifies if the data was written correctly.

Prototype

```
int FS_NAND_UNI_TestBlock(U8                   Unit,
                          unsigned             BlockIndex,
                          U32                  Pattern,
                          FS_NAND_TEST_INFO *  pInfo);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
BlockIndex	Index of the NAND block to be tested.
Pattern	Data pattern to be written during the test.
pInfo	Additional parameters and information about the test.

Return value

FS_NAND_TEST_RETVAL_OK	OK, no bit errors.
FS_NAND_TEST_RETVAL_CORRECTABLE_ERROR	OK, correctable bit errors found. The number of bit errors is returned in NumErrorsCorrectable of pResult.
FS_NAND_TEST_RETVAL_FATAL_ERROR	Fatal error, uncorrectable bit error found. The page index is returned in PageIndexFatalError of pResult.
FS_NAND_TEST_RETVAL_BAD_BLOCK	Bad block, skipped.
FS_NAND_TEST_RETVAL_ERASE_FAILURE	Erase operation failed. The block has been marked as defective.
FS_NAND_TEST_RETVAL_WRITE_FAILURE	Write operation failed. The block has been marked as defective.
FS_NAND_TEST_RETVAL_READ_FAILURE	Read operation failed.
FS_NAND_TEST_RETVAL_INTERNAL_ERROR	NAND flash access error.

Additional information

This function is optional. It can be used by the application to test the data reliability of a NAND block. BlockIndex is relative to the beginning of the NAND partition where the first block has the index 0.

Example

The following sample code demonstrates how FS_NAND_UNI_TestBlock() can be used to check all the blocks of a NAND flash device. The function does not work correctly on NAND flash devices with HW ECC that cannot be disabled.

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

void SampleNAND_UNITestBlocks(void) {
  int               r;
  U32               NumPhyBlocks;
  U32               iBlock;
  FS_NAND_DISK_INFO DiskInfo;
  FS_NAND_TEST_INFO TestInfo;

  memset(&DiskInfo, 0, sizeof(DiskInfo));
  memset(&TestInfo, 0, sizeof(TestInfo));
  FS_NAND_UNI_GetDiskInfo(0, &DiskInfo);
```



```
  NumPhyBlocks = DiskInfo.NumPhyBlocks;
  TestInfo.NumBitsCorrectable = 1;
  TestInfo.BytesPerSpare      = (U16)DiskInfo.BytesPerSpareArea;
  for (iBlock = 0; iBlock < NumPhyBlocks; ++iBlock) {
    r = FS_NAND_UNI_TestBlock(0, iBlock, 0xAA5500FFuL, &TestInfo);
    switch (r) {
    case FS_NAND_TEST_RETVAL_OK:
      printf("Block %lu: OK.\n", iBlock);
      break;
    case FS_NAND_TEST_RETVAL_CORRECTABLE_ERROR:
      printf("Block %lu: %lu correctable error(s).\n", iBlock, TestInfo.BitErrorCnt);
      break;
    case FS_NAND_TEST_RETVAL_FATAL_ERROR:
      printf("Block %lu: %lu correctable error(s), fatal error on page %lu.\n",
             iBlock, TestInfo.BitErrorCnt, TestInfo.PageIndex);
      break;
    case FS_NAND_TEST_RETVAL_BAD_BLOCK:
      printf("Block %lu: Bad. Skipped.\n", iBlock);
      break;
    case FS_NAND_TEST_RETVAL_ERASE_FAILURE:
      printf("Block %lu: Erase failure. Marked as bad.\n", iBlock);
      break;
    case FS_NAND_TEST_RETVAL_WRITE_FAILURE:
      printf("Block %lu: Write failure on page %lu. Marked as bad.\n",
             iBlock, TestInfo.PageIndex);
      break;
    case FS_NAND_TEST_RETVAL_READ_FAILURE:
      printf("Block %lu: Read failure on page %lu.\n", iBlock, TestInfo.PageIndex);
      break;
    default:
      printf("Block %lu: Internal error.\n", iBlock);
      break;
    }
  }
}
```


6.3.3.7.13 FS_NAND_UNI_WritePage()

Description

Stores data to a page of a NAND flash with ECC.

Prototype

```
int FS_NAND_UNI_WritePage(      U8         Unit,
                                U32        PageIndex,
                          const void     * pData,
                                unsigned   NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
PageIndex	Index of the page to be written.
pData	in Data to be written.
NumBytes	Number of bytes to be written.

Return value

= 0 OK, data written.
≠ 0 An error occurred.

Additional information

This function is optional.

The data is written beginning with the byte offset 0 in the page. If more data is written than the size of the page, typ 2 KB + 64 bytes, the excess bytes are discarded. Data in the area reserved for ECC cannot be written using this function and it will be overwritten.

6.3.3.7.14 FS_NAND_UNI_WritePageRaw()

Description

Stores data to a page of a NAND flash without ECC.

Prototype

```
int FS_NAND_UNI_WritePageRaw(      U8         Unit,
                                   U32        PageIndex,
                             const void     * pData,
                                   unsigned   NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
PageIndex	Index of the page to be written.
pData	in Data to be written.
NumBytes	Number of bytes to be written.

Return value

= 0 OK, data written.
≠ 0 An error occurred.

Additional information

This function is optional.

The data is written beginning at the byte offset 0 in the page. If more data is written than the size of the page + spare area, typ. 2 Kbytes + 64 bytes, the excess bytes are ignored.

FS_NAND_UNI_WritePageRaw() does not work correctly on NAND flash devices with HW ECC that cannot be disabled.

6.3.3.7.15 Bad block marking types

Description

Methods to mark a block as defective.

Definition

```
#define FS_NAND_BAD_BLOCK_MARKING_TYPE_UNKNOWN    0
#define FS_NAND_BAD_BLOCK_MARKING_TYPE_ONFI       1
#define FS_NAND_BAD_BLOCK_MARKING_TYPE_LEGACY     2
```

Symbols

Definition	Description
FS_NAND_BAD_BLOCK_MARKING_TYPE_UNKNOWN	Not known.
FS_NAND_BAD_BLOCK_MARKING_TYPE_ONFI	The block is marked as bad in the first page of the block.
FS_NAND_BAD_BLOCK_MARKING_TYPE_LEGACY	The block is marked as bad in the first and second page of the block.

6.3.3.7.16 ECC correction status

Description

Result of the bit error correction.

Definition

```
#define FS_NAND_CORR_NOT_APPLIED    0u
#define FS_NAND_CORR_APPLIED        1u
#define FS_NAND_CORR_FAILURE        2u
```

Symbols

Definition	Description
FS_NAND_CORR_NOT_APPLIED	No bit errors detected.
FS_NAND_CORR_APPLIED	Bit errors were detected and corrected.
FS_NAND_CORR_FAILURE	Bit errors were detected but not corrected.

6.3.3.7.17 FS_NAND_STAT_COUNTERS

Description

Statistical counters of NAND flash driver.

Type definition

```
typedef struct {
  U32  NumFreeBlocks;
  U32  NumBadBlocks;
  U32  EraseCnt;
  U32  ReadDataCnt;
  U32  ReadSpareCnt;
  U32  ReadSectorCnt;
  U32  NumReadRetries;
  U32  WriteDataCnt;
  U32  WriteSpareCnt;
  U32  WriteSectorCnt;
  U32  NumWriteRetries;
  U32  ConvertViaCopyCnt;
  U32  ConvertInPlaceCnt;
  U32  NumValidSectors;
  U32  CopySectorCnt;
  U32  BlockRelocationCnt;
  U32  ReadByteCnt;
  U32  WriteByteCnt;
  U32  BitErrorCnt;
  U32  aBitErrorCnt[];
} FS_NAND_STAT_COUNTERS;
```

Structure members

Member	Description
NumFreeBlocks	Number of NAND blocks not used for data.
NumBadBlocks	Number of NAND blocks marked as defective.
EraseCnt	Number of block erase operation performed.
ReadDataCnt	Number of times the NAND driver read from the main area of a page.
ReadSpareCnt	Number of times the NAND driver read from the spare area of a page.
ReadSectorCnt	Number of logical sectors read from the NAND flash.
NumReadRetries	Number of times a read operation has been retried because of an error.
WriteDataCnt	Number of times the NAND driver wrote to the main area of a page.
WriteSpareCnt	Number of times the NAND driver wrote to the spare area of a page.
WriteSectorCnt	Number of logical sectors wrote to the NAND flash.
NumWriteRetries	Number of times a write operation has been retried because of an error.
ConvertViaCopyCnt	Number of block conversions via copy.
ConvertInPlaceCnt	Number of block conversions in place.
NumValidSectors	Number of logical sectors that contain valid data.
CopySectorCnt	Number of times the NAND driver copied a logical sector to another location.
BlockRelocationCnt	Number of times the NAND driver relocated a NAND block due to errors.

Member	Description
ReadByteCnt	Number of bytes read from NAND flash.
WriteByteCnt	Number of bytes written to NAND flash.
BitErrorCnt	Number of bit errors detected and corrected.
aBitErrorCnt	Number of times a specific number of bit errors occurred.

Additional information

This structure can be used to get statistical information about the operation of the Universal as well as SLC1 NAND driver via the function FS_NAND_UNI_GetStatCounters() and FS_NAND_GetStatCounters() respectively.

aBitErrorCnt[0] stores the number of 1 bit error occurrences, aBitErrorCnt[1] stores the number of 2 bit error occurrences, and so on.

Example

Refer to FS_NAND_UNI_GetStatCounters() and FS_NAND_GetStatCounters() for a sample usage.

6.3.3.7.18 FS_NAND_TEST_INFO

Description

Additional information passed to test routine.

Type definition

```
typedef struct {
  U8   NumBitsCorrectable;
  U8   OffSpareECCProt;
  U8   NumBytesSpareECCProt;
  U16  BytesPerSpare;
  U32  BitErrorCnt;
  U32  PageIndex;
} FS_NAND_TEST_INFO;
```

Structure members

Member	Description
NumBitsCorrectable	Number of bits the ECC can correct in the data and spare area (typ. 4)
OffSpareECCProt	Offset in the spare area of the first byte protected by ECC (typ. 4).
NumBytesSpareECCProt	Number of bytes in the spare area protected by ECC (typ. 4 bytes)
BytesPerSpare	Total number of bytes in the spare area. When set to 0 the default value of 1/32 of page size is used.
BitErrorCnt	Number of bit errors detected and corrected.
PageIndex	Index of the physical page where the error happened.

Additional information

The test routine returns information about what went wrong during a test via FS_NAND_UNI_TestBlock() and FS_NAND_TestBlock().

Example

Refer to FS_NAND_UNI_TestBlock() and FS_NAND_TestBlock() for a sample usage.

6.3.3.7.19 NAND block types

Description

Type of data stored to a NAND block.

Definition

```
#define FS_NAND_BLOCK_TYPE_UNKNOWN    0
#define FS_NAND_BLOCK_TYPE_BAD        1
#define FS_NAND_BLOCK_TYPE_EMPTY      2
#define FS_NAND_BLOCK_TYPE_WORK       3
#define FS_NAND_BLOCK_TYPE_DATA       4
```

Symbols

Definition	Description
FS_NAND_BLOCK_TYPE_UNKNOWN	The type of the block cannot be determined.
FS_NAND_BLOCK_TYPE_BAD	The block marked as defective.
FS_NAND_BLOCK_TYPE_EMPTY	The block does not store any data.
FS_NAND_BLOCK_TYPE_WORK	The block that stores data temporarily.
FS_NAND_BLOCK_TYPE_DATA	The block that stores data permanently.

6.3.3.7.20 NAND test return values

Description

Return values of the NAND block test functions.

Definition

```
#define FS_NAND_TEST_RETVAL_OK                  0
#define FS_NAND_TEST_RETVAL_CORRECTABLE_ERROR   1
#define FS_NAND_TEST_RETVAL_FATAL_ERROR         2
#define FS_NAND_TEST_RETVAL_BAD_BLOCK           3
#define FS_NAND_TEST_RETVAL_ERASE_FAILURE       4
#define FS_NAND_TEST_RETVAL_WRITE_FAILURE       5
#define FS_NAND_TEST_RETVAL_READ_FAILURE        6
#define FS_NAND_TEST_RETVAL_INTERNAL_ERROR      (-1)
```

Symbols

Definition	Description
FS_NAND_TEST_RET-VAL_OK	The test was successful.
FS_NAND_TEST_RET-VAL_CORRECTABLE_ERROR	Bit errors occurred that were corrected via ECC.
FS_NAND_TEST_RET-VAL_FATAL_ERROR	Bit errors occurred that the ECC was not able to correct.
FS_NAND_TEST_RET-VAL_BAD_BLOCK	The tested block was marked as defective.
FS_NAND_TEST_RET-VAL_ERASE_FAILURE	An error occurred during the block erase operation.
FS_NAND_TEST_RET-VAL_WRITE_FAILURE	An error occurred while writing the data to the NAND block.
FS_NAND_TEST_RET-VAL_READ_FAILURE	An error occurred while reading the data from the NAND block.
FS_NAND_TEST_RET-VAL_INTERNAL_ERROR	An internal processing error occurred.

6.3.3.8 Performance and resource usage

This section provides information about the ROM and RAM usage as well as the performance of the Universal NAND driver. Please note that a Universal NAND driver instance requires one instance of one NAND physical layer in order to operate. The resource usage of the used NAND physical layer has to be taken into account when calculating the total resource usage of the Universal NAND driver.

6.3.3.8.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the Universal NAND driver was measured using the SEGGER Embedded Studio IDE V4.20 configured to generate code for a Cortex-M4 CPU in Thumb mode and with the size optimization enabled.

Usage: 7.9 Kbytes

6.3.3.8.2 Static RAM usage

Static RAM usage refers to the amount of RAM required by the Universal NAND driver internally for all the driver instances. The number of bytes can be seen in the compiler list file of the {FS_NAND_UNI_Drv.c} file.

Usage: 32 bytes

6.3.3.8.3 Dynamic RAM usage

Dynamic RAM usage is the amount of RAM allocated by the driver at runtime. The amount required depends on the runtime configuration and the characteristics of the used NAND flash device. The approximate RAM usage of the Universal NAND driver can be calculated as follows:

```
MemAllocated = 148 + 2 * NumBlocks
             + ((PagesPerBlock - 1) + 18) * NumWorkBlocks
             + 1.04 * PageSize
```

Parameter	Description
MemAllocated	Number of bytes allocated for one instance of the NAND driver.
NumBlocks	Number of blocks in the NAND flash device.
NumWorkBlocks	Number of blocks the driver reserves as temporary storage for the written data. By default, 3 blocks are reserved. The number of work blocks can be specified at runtime via FS_NAND_UNI_SetNumWorkBlocks().
PageSize	Number of bytes in a page.

Example

This example uses a 2 GBit NAND flash device with 2Kbyte pages and 2048 blocks. One block consists of 64 pages and 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
```


6.3.3.8.4 Performance

The following performance measurements are in no way complete, but they give a good approximation of time required for common operations on various target hardware. The tests were performed as described in Performance. All values are given in Mbytes/sec.

CPU type	NAND flash device	Write speed	Read speed
NXP LPC4322 (180 MHz)	Serial NAND flash with 2048 bytes per page connected to a quad SPI controller via 4 data lines transferring data at 60 MHz.	**4.1**	**12.2**
NXP LPC4322 (180 MHz)	Serial NAND flash with 2048 bytes per page connected to a standard SPI controller transferring data at 60 MHz.	**3.9**	**7.7**
ST STM32MP157 (650 MHz)	Parallel NAND flash with 4096 bytes per page connected to external memory controller via an 8-bit data bus. The 8-bit ECC is calculated using SEGGER emFile-ECC.	**10.8**	**22.2**
Atmel AT91SAM3U (96 MHz)	Parallel NAND flash 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.	**2.6**	**7.5**

6.3.4 NAND physical layer

6.3.4.1 General information

The NAND physical layer provides the basic functionality for accessing a NAND flash device such as device identification, block erase operation, page read and write operations, etc. Every instance of the Universal or SLC1 NAND driver requires an instance of a NAND physical layer in order to be able to operate. A NAND physical layer instance is automatically allocated either statically or dynamically at the first call to one of its API functions. Each instance is identified by a unit number that is identical with the unit number of the NAND driver that uses that instance of the NAND physical layer. The type of the NAND physical layer assigned to an instance of a Universal or SLC1 NAND driver is configured via `FS_NAND_UNI_SetPhyType()` and `FS_NAND_SetPhyType()` respectively.

The following sections provide information about the usage and the implementation of a NAND physical layer.

6.3.4.2 Available physical layers

The table below lists the NAND physical layers that ship with emFile. Refer to *Configuring the driver* on page 337 and *Configuring the driver* on page 375 for detailed information about how to add and configure a physical layer in an application.

Physical layer identifier	Works with Universal NAND driver?	Works with SLC1 NAND driver?	Hardware layer type
`FS_NAND_PHY_512x8`	no	yes	`FS_NAND_HW_TYPE`
`FS_NAND_PHY_2048x8`	yes	yes	`FS_NAND_HW_TYPE`
`FS_NAND_PHY_2048x8_Small`	yes	yes	`FS_NAND_HW_TYPE`
`FS_NAND_PHY_2048x16`	yes	yes	`FS_NAND_HW_TYPE`
`FS_NAND_PHY_4096x8`	yes	yes	`FS_NAND_HW_TYPE`
`FS_NAND_PHY_DataFlash`	no	yes	`FS_NAND_HW_TYPE_DF`
`FS_NAND_PHY_ONFI`	yes	yes	`FS_NAND_HW_TYPE`
`FS_NAND_PHY_ONFI_RO`	yes	yes	`FS_NAND_HW_TYPE`
`FS_NAND_PHY_ONFI_Small`	yes	yes	`FS_NAND_HW_TYPE`
`FS_NAND_PHY_QSPI`	yes	no	`FS_NAND_HW_TYPE_QSPI`
`FS_NAND_PHY_SPI`	yes	no	`FS_NAND_HW_TYPE_SPI`
`FS_NAND_PHY_x`	yes	yes	`FS_NAND_HW_TYPE`
`FS_NAND_PHY_x8`	yes	yes	`FS_NAND_HW_TYPE`

6.3.4.2.1 512x8 physical layer

This NAND physical layer supports any NAND flash device with a page size of 512 bytes and a spare area of 16 bytes that is connected to MCU via an 8-bit data bus. It works only with the SLC1 NAND driver because the size of the spare area is not sufficiently large for the amount of management data required by the Universal NAND driver. The instances of this physical layer are allocated statically. The maximum number of instances can be configure at build time via `FS_NAND_NUM_UNITS`.

The table below summarizes the specific functions of the physical layer followed by a detailed description of each function.

Function	Description
`FS_NAND_512x8_SetHWType()`	Configures the hardware access routines for a NAND physical layer of type `FS_NAND_PHY_512x8`.

6.3.4.2.1.1 FS_NAND_512x8_SetHWType()

Description

Configures the hardware access routines for a NAND physical layer of type FS_NAND_PHY_512x8.

Prototype

```
void FS_NAND_512x8_SetHWType(      U8                Unit,
                             const FS_NAND_HW_TYPE * pHWType);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)
pHWType	Type of the hardware layer to use. Cannot be NULL.

Additional information

This function is mandatory and has to be called once in FS_X_AddDevices() for every instance of a NAND physical layer of type FS_NAND_PHY_512x8.

6.3.4.2.2 2048x8 physical layer

This NAND physical layer supports any NAND flash device with a page size of 2048 bytes and a spare area larger than or equal to 64 bytes that is connected to MCU via an 8-bit data bus. The instances of this physical layer are allocated statically. The maximum number of instances can be configure at build time via FS_NAND_NUM_UNITS.

The table below summarizes the specific functions of the physical layer followed by a detailed description of each function.

Function	Description
FS_NAND_2048x8_DisableReadCache()	Deactivates the page read optimization.
FS_NAND_2048x8_EnableReadCache()	Activates the page read optimization.
FS_NAND_2048x8_SetHWType()	Configures the hardware access routines for a NAND physical layer of type FS_NAND_PHY_2048x8.

6.3.4.2.2.1 FS_NAND_2048x8_DisableReadCache()

Description

Deactivates the page read optimization.

Prototype

```
void FS_NAND_2048x8_DisableReadCache(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)

Additional information

This function is optional and is available only when the file system is build with FS_NAND_SUPPORT_READ_CACHE set to 1 which is the default. The optimization can be enabled at runtime via FS_NAND_2048x8_EnableReadCache().

Refer to FS_NAND_2048x8_EnableReadCache() for more information about how the page read optimization works

6.3.4.2.2.2 FS_NAND_2048x8_EnableReadCache()

Description

Activates the page read optimization.

Prototype

```
void FS_NAND_2048x8_EnableReadCache(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)

Additional information

This function is optional and is available only when the file system is build with FS_NAND_SUPPORT_READ_CACHE set to 1 which is the default. Activating the read cache can increase the overall performance of the NAND driver especially when using the SLC1 NAND driver with a logical sector size smaller than the page of the used NAND flash device.

The optimization takes advantage of how the NAND flash device implements the read page operation. A NAND page read operation consists of two steps. In the first step, the page data is read from the memory array to internal page register of the NAND flash device. In the second step, the data is transferred from the internal page register of NAND flash device to MCU. With the optimization enabled the first step is skipped whenever possible.

The optimization is enabled by default and has to be disabled if two or more instances of the NAND driver are configured to access the same physical NAND flash device. At runtime, the optimization can be disabled via FS_NAND_2048x8_DisableReadCache().

6.3.4.2.2.3 FS_NAND_2048x8_SetHWType()

Description

Configures the hardware access routines for a NAND physical layer of type FS_NAND_PHY_2048x8.

Prototype

```
void FS_NAND_2048x8_SetHWType(      U8                Unit,
                              const FS_NAND_HW_TYPE * pHWType);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)
pHWType	Type of the hardware layer to use. Cannot be NULL.

Additional information

This function is mandatory and has to be called once in FS_X_AddDevices() for every instance of a NAND physical layer of type FS_NAND_PHY_2048x8.

6.3.4.2.3 Small 2048x8 physical layer

This physical layer is a variant of FS_NAND_PHY_2048x8 with reduced ROM usage. FS_NAND_PHY_2048x8_Small supports the same NAND flash devices as FS_NAND_PHY_2048x8 but it does not provide support for the NAND internal page copy operation and for the reading the ECC correction result. FS_NAND_PHY_2048x8_Small provides read as well as write access to NAND flash device. The instances of this physical layer are allocated statically.

FS_NAND_PHY_2048x8_Small provides the same specific functions as FS_NAND_PHY_2048x8.

6.3.4.2.4 2048x16 physical layer

This physical layer supports NAND flash devices with page size of 2048 bytes and a spare area larger than or equal to 64 bytes that is connected to MCU via an 8-bit data bus. The instances of this physical layer are allocated statically. The maximum number of instances can be configure at build time via `FS_NAND_NUM_UNITS`.

The table below summarizes the specific functions of the physical layer followed by a detailed description of each function.

Function	Description
`FS_NAND_2048x16_SetHWType()`	Configures the hardware access routines for a NAND physical layer of type `FS_NAND_PHY_2048x16`.

6.3.4.2.4.1 FS_NAND_2048x16_SetHWType()

Description

Configures the hardware access routines for a NAND physical layer of type FS_NAND_PHY_2048x16.

Prototype

```
void FS_NAND_2048x16_SetHWType(      U8                Unit,
                               const FS_NAND_HW_TYPE * pHWType);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)
pHWType	Type of the hardware layer to use. Cannot be NULL.

Additional information

This function is mandatory and has to be called once in FS_X_AddDevices() for every instance of a NAND physical layer of type FS_NAND_PHY_2048x16.

6.3.4.2.5 4096x8 physical layer

This NAND physical layer supports any NAND flash device with a page size of 4096 bytes and a spare area larger than or equal to 128 bytes that is connected to MCU via an 8-bit data bus. The instances of this physical layer are allocated statically. The maximum number of instances can be configure at build time via `FS_NAND_NUM_UNITS`.

The table below summarizes the specific functions of the physical layer followed by a detailed description of each function.

Function	Description
`FS_NAND_4096x8_SetHWType()`	Configures the hardware access routines for a NAND physical layer of type `FS_NAND_PHY_4096x8`.

6.3.4.2.5.1 FS_NAND_4096x8_SetHWType()

Description

Configures the hardware access routines for a NAND physical layer of type FS_NAND_PHY_4096x8.

Prototype

```
void FS_NAND_4096x8_SetHWType(      U8                Unit,
                              const FS_NAND_HW_TYPE * pHWType);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)
pHWType	Type of the hardware layer to use. Cannot be NULL.

Additional information

This function is mandatory and has to be called once in FS_X_AddDevices() for every instance of a NAND physical layer of type FS_NAND_PHY_4096x8.

6.3.4.2.6 DataFlash physical layer

This physical layer supports Microchip / Atmel and Adesto DataFlash storage devices. The devices are accessed via SPI using a hardware layer of type FS_NAND_HW_TYPE_DF. FS_NAND_PHY_DataFlash works only with the SLC1 NAND driver because the size of the spare area is not sufficiently large for the amount of management data required by the Universal NAND driver. The instances of this physical layer are allocated statically.

The table below summarizes the specific functions of the physical layer followed by a detailed description of each function.

Function	Description
FS_NAND_DF_EraseChip()	Erases the entire device.
FS_NAND_DF_SetMinPageSize()	Configures the required minimum page size.
FS_NAND_DF_SetHWType()	Configures the hardware access routines.

6.3.4.2.6.1 FS_NAND_DF_EraseChip()

Description

Erases the entire device.

Prototype

```
void FS_NAND_DF_EraseChip(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based)

Additional information

This function is optional. It sets all the bits of the DataFlash memory to 1. All the data stored on the DataFlash memory is lost.

6.3.4.2.6.2 FS_NAND_DF_SetMinPageSize()

Description

Configures the required minimum page size.

Prototype

```
void FS_NAND_DF_SetMinPageSize(U8  Unit,
                               U32 NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based)
NumBytes	Page size in bytes.

Additional information

This function is optional. The application can use it to request a minimum page size to work with. If the size of the physical page is smaller than the specified value then then adjacent physical pages are grouped together into one virtual page that is presented as a single page to the SLC1 NAND driver. This is required when the size of a physical page is smaller than 512 bytes which is the minimum sector size the SLC1 NAND driver can work with. NumBytes has to be a power of 2 value.

6.3.4.2.6.3 FS_NAND_DF_SetHWType()

Description

Configures the hardware access routines.

Prototype

```
void FS_NAND_DF_SetHWType(      U8                    Unit,
                          const FS_NAND_HW_TYPE_DF * pHWType);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based)
pHWType	Table of hardware routines.

Additional information

This function is mandatory and it has to be called once for each used instance of the physical layer.

6.3.4.2.7 ONFI physical layer

This physical layer supports NAND flash devices that are compliant to ONFI specification. The geometry of the NAND flash device such as page size and number of blocks is determined automatically by evaluating the parameters stored in the NAND flash device according to ONFI specification. The instances of this physical layer are allocated dynamically.

The table below summarizes the specific functions of the physical layer followed by a detailed description of each function.

Function	Description
FS_NAND_ONFI_SetHWType()	Configures the hardware access routines for a NAND physical layer of type FS_NAND_PHY_ONFI.

6.3.4.2.7.1 FS_NAND_ONFI_SetHWType()

Description

Configures the hardware access routines for a NAND physical layer of type FS_NAND_PHY_ONFI.

Prototype

```
void FS_NAND_ONFI_SetHWType(      U8                Unit,
                            const FS_NAND_HW_TYPE * pHWType);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)
pHWType	Type of the hardware layer to use. Cannot be NULL.

Additional information

This function is mandatory and has to be called once in FS_X_AddDevices() for every instance of a NAND physical layer of type FS_NAND_PHY_ONFI.

6.3.4.2.8 Read-only ONFI physical layer

This physical layer is a variant of FS_NAND_PHY_ONFI with read-only access. FS_NAND_PHY_ONFI_RO provides the same functionality as FS_NAND_PHY_ONFI except that it cannot modify any data stored on the NAND flash device only to read it. The erase and write functions of FS_NAND_PHY_ONFI_RO do nothing and return and error to NAND driver when called.

FS_NAND_PHY_ONFI_RO provides the same specific functions as FS_NAND_PHY_ONFI.

6.3.4.2.9 Small ONFI physical layer

This physical layer is a variant of FS_NAND_PHY_ONFI with reduced ROM usage. FS_NAND_PHY_ONFI_Small supports the same NAND flash devices as FS_NAND_PHY_ONFI but it does not provide support for the internal page copy operation and for the reading the ECC correction result. FS_NAND_PHY_ONFI_Small provides read as well as write access to NAND flash device.

FS_NAND_PHY_ONFI_Small provides the same specific functions as FS_NAND_PHY_ONFI.

6.3.4.2.10 Quad-SPI physical layer

This physical layer supports NAND flash devices interfaced via quad or dual SPI. The geometry of the NAND flash device such as page size and number of blocks is determined automatically by evaluating the parameters stored in the NAND flash device. The instances of this physical layer are allocated dynamically.

The table below summarizes the specific functions of the physical layer followed by a detailed description of each function.

Function	Description
FS_NAND_QSPI_Allow2bitMode()	Specifies if the physical layer can exchange data via 2 data lines.
FS_NAND_QSPI_Allow4bitMode()	Specifies if the physical layer can exchange data via 4 data lines.
FS_NAND_QSPI_DisableReadCache()	Deactivates the page read optimization.
FS_NAND_QSPI_EnableReadCache()	Activates the page read optimization.
FS_NAND_QSPI_SetHWType()	Configures the hardware access routines for a NAND physical layer of type FS_NAND_PHY_QSPI.

6.3.4.2.10.1 FS_NAND_QSPI_Allow2bitMode()

Description

Specifies if the physical layer can exchange data via 2 data lines.

Prototype

```
void FS_NAND_QSPI_Allow2bitMode(U8 Unit,
                                U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based)
OnOff	Activation status of the option. • 0 Data is exchanged via 1 data line. • 1 Data is exchanged via 2 data lines.

Additional information

This function is optional. By default the data is exchanged via 1 data line (standard SPI mode).

6.3.4.2.10.2 FS_NAND_QSPI_Allow4bitMode()

Description

Specifies if the physical layer can exchange data via 4 data lines.

Prototype

```
void FS_NAND_QSPI_Allow4bitMode(U8 Unit,
                                U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based)
OnOff	Activation status of the option. • 0 Data is exchanged via 1 data line or 2 data lines. • 1 Data is exchanged via 4 data lines.

Additional information

This function is optional. By default the data is exchanged via 1 data line (standard SPI mode).

6.3.4.2.10.3 FS_NAND_QSPI_DisableReadCache()

Description

Deactivates the page read optimization

Prototype

```
void FS_NAND_QSPI_DisableReadCache(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)

Additional information

This function is optional and is available only when the file system is build with FS_NAND_SUPPORT_READ_CACHE set to 1 which is the default. The optimization can be enabled at runtime via FS_NAND_QSPI_EnableReadCache().

Refer to FS_NAND_QSPI_EnableReadCache() for more information about how the page read optimization works

6.3.4.2.10.4 FS_NAND_QSPI_EnableReadCache()

Description

Activates the page read optimization

Prototype

```
void FS_NAND_QSPI_EnableReadCache(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)

Additional information

This function is optional and is available only when the file system is build with FS_NAND_SUPPORT_READ_CACHE set to 1 which is the default. Activating the read cache can increase the overall performance of the NAND driver.

The optimization takes advantage of how the NAND flash device implements the read page operation. A NAND page read operation consists of two steps. In the first step, the page data is read from the memory array to internal page register of the NAND flash device. In the second step, the data is transferred from the internal page register of NAND flash device to MCU. With the optimization enabled the first step is skipped whenever possible.

The optimization is enabled by default and has to be disabled if two or more instances of the NAND driver are configured to access the same physical NAND flash device. At runtime, the optimization can be disabled via FS_NAND_QSPI_DisableReadCache().

6.3.4.2.10.5 FS_NAND_QSPI_SetHWType()

Description

Configures the hardware access routines for a NAND physical layer of type FS_NAND_PHY_QSPI.

Prototype

```
void FS_NAND_QSPI_SetHWType(      U8                       Unit,
                            const FS_NAND_HW_TYPE_QSPI * pHWType);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)
pHWType	Type of the hardware layer to use. Cannot be NULL.

Additional information

This function is mandatory and has to be called once in FS_X_AddDevices() for every instance of a NAND physical layer of type FS_NAND_PHY_QSPI.

6.3.4.2.11 SPI physical layer

This physical layer supports NAND flash devices interfaced via standard SPI. The geometry of the NAND flash device such as page size and number of blocks is determined automatically by evaluating the parameters stored in the NAND flash device. The instances of this physical layer are allocated dynamically.

The table below summarizes the specific functions of the physical layer followed by a detailed description of each function.

Function	Description
FS_NAND_SPI_DisableReadCache()	Deactivates the page read optimization.
FS_NAND_SPI_EnableReadCache()	Activates the page read optimization.
FS_NAND_SPI_SetDeviceList()	Specifies the list of enabled serial NAND flash devices.
FS_NAND_SPI_SetHWType()	Configures the hardware access routines for a NAND physical layer of type FS_NAND_PHY_SPI.

6.3.4.2.11.1 FS_NAND_SPI_DisableReadCache()

Description

Deactivates the page read optimization

Prototype

```
void FS_NAND_SPI_DisableReadCache(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)

Additional information

This function is optional and is available only when the file system is build with FS_NAND_SUPPORT_READ_CACHE set to 1 which is the default. The optimization can be enabled at runtime via FS_NAND_SPI_EnableReadCache().

Refer to FS_NAND_SPI_EnableReadCache() for more information about how the page read optimization works

6.3.4.2.11.2 FS_NAND_SPI_EnableReadCache()

Description

Activates the page read optimization

Prototype

```
void FS_NAND_SPI_EnableReadCache(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)

Additional information

This function is optional and is available only when the file system is build with FS_NAND_SUPPORT_READ_CACHE set to 1 which is the default. Activating the read cache can increase the overall performance of the NAND driver.

The optimization takes advantage of how the NAND flash device implements the read page operation. A NAND page read operation consists of two steps. In the first step, the page data is read from the memory array to internal page register of the NAND flash device. In the second step, the data is transferred from the internal page register of NAND flash device to MCU. With the optimization enabled the first step is skipped whenever possible.

The optimization is enabled by default and has to be disabled if two or more instances of the NAND driver are configured to access the same physical NAND flash device. At runtime, the optimization can be disabled via FS_NAND_SPI_DisableReadCache().

6.3.4.2.11.3 FS_NAND_SPI_SetDeviceList()

Description

Specifies the list of enabled serial NAND flash devices.

Prototype

```
void FS_NAND_SPI_SetDeviceList(      U8                          Unit,
                               const FS_NAND_SPI_DEVICE_LIST * pDeviceList);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based)
pDeviceList	in List of serial NAND flash devices.

Additional information

All supported serial NAND flash devices are enabled by default. Serial NAND flash devices that are not on the list are not recognized by the file system.

Permitted values for the pDeviceList parameter are:

Identifier	Description
FS_NAND_SPI_DeviceList_All	Enables handling of serial NAND flash devices from all manufacturers.
FS_NAND_SPI_DeviceList_Default	Enables handling of NAND flash devices for any other manufacturer.
FS_NAND_SPI_DeviceList_ISSI	Enables handling of ISSI serial NAND flash devices.
FS_NAND_SPI_DeviceList_Macronix	Enables handling of Macronix serial NAND flash devices.
FS_NAND_SPI_DeviceList_Micron	Enables handling of Micron serial NAND flash devices.
FS_NAND_SPI_DeviceList_Toshiba	Enables handling of Toshiba serial NAND flash devices.
FS_NAND_SPI_DeviceList_Winbond	Enables handling of Winbond serial NAND flash devices.

6.3.4.2.11.4 FS_NAND_SPI_SetHWType()

Description

Configures the hardware access routines for a NAND physical layer of type FS_NAND_PHY_SPI.

Prototype

```
void FS_NAND_SPI_SetHWType(      U8                     Unit,
                           const FS_NAND_HW_TYPE_SPI * pHWType);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)
pHWType	Type of the hardware layer to use. Cannot be NULL.

Additional information

This function is mandatory and has to be called once in FS_X_AddDevices() for every instance of a NAND physical layer of type FS_NAND_PHY_SPI.

6.3.4.2.12 8/16-bit data bus physical layer

This physical layer supports all the NAND flash devices supported by the following NAND physical layers:

- FS_NAND_PHY_512x8
- FS_NAND_PHY_2048x8
- FS_NAND_PHY_2048x16
- FS_NAND_PHY_4096x8
- FS_NAND_PHY_ONFI

The table below summarizes the specific functions of the physical layer followed by a detailed description of each function.

Function	Description
FS_NAND_x_Configure()	Configures the parameters of the NAND flash device for a NAND physical layer of type FS_NAND_PHY_x.
FS_NAND_x_SetHWType()	Configures the hardware access routines for a NAND physical layer of type FS_NAND_PHY_x.

6.3.4.2.12.1 FS_NAND_x_Configure()

Description

Configures the parameters of the NAND flash device for a NAND physical layer of type FS_NAND_PHY_x.

Prototype

```
void FS_NAND_x_Configure(U8       Unit,
                         unsigned NumBlocks,
                         unsigned PagesPerBlock,
                         unsigned BytesPerPage,
                         unsigned BytesPerSpareArea,
                         unsigned DataBusWidth);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)
NumBlocks	Total number of blocks in the NAND flash device.
PagesPerBlock	Total number of pages in a NAND block.
BytesPerPage	Number of bytes in a page without the spare area.
BytesPerSpareArea	Number of bytes in the spare area of a NAND page.
DataBusWidth	Number of data lines used for data exchange.

Additional information

This function is mandatory only when the file system is built with FS_NAND_SUPPORT_AUTO_DETECTION set to 0 which is not the default. FS_NAND_x_Configure() has to be called once in FS_X_AddDevices() for each instance of the FS_NAND_PHY_x physical layer. FS_NAND_x_Configure() is not available if FS_NAND_SUPPORT_AUTO_DETECTION is set to 0.

By default, the FS_NAND_PHY_x physical layer identifies the parameters of the NAND flash device by evaluating the first and second byte of the reply returned by the NAND flash device to the READ ID (0x90) command. The identification operation is disabled if FS_NAND_SUPPORT_AUTO_DETECTION set to 0 and the application must specify the NAND flash parameters via this function.

6.3.4.2.12.2 FS_NAND_x_SetHWType()

Description

Configures the hardware access routines for a NAND physical layer of type FS_NAND_PHY_x.

Prototype

```
void FS_NAND_x_SetHWType(      U8                Unit,
                         const FS_NAND_HW_TYPE * pHWType);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)
pHWType	Type of the hardware layer to use. Cannot be NULL.

Additional information

This function is mandatory and has to be called once in FS_X_AddDevices() for every instance of a NAND physical layer of type FS_NAND_PHY_x.

6.3.4.2.13 8-bit data bus physical layer

This physical layer supports all the NAND flash devices supported by the following NAND physical layers:

- FS_NAND_PHY_512x8
- FS_NAND_PHY_2048x8
- FS_NAND_PHY_4096x8
- FS_NAND_PHY_ONFI

The table below summarizes the specific functions of the physical layer followed by a detailed description of each function.

Function	Description
FS_NAND_x8_Configure()	Configures the parameters of the NAND flash device for a NAND physical layer of type FS_NAND_PHY_x8.
FS_NAND_x8_SetHWType()	Configures the hardware access routines for a NAND physical layer of type FS_NAND_PHY_x8.

6.3.4.2.13.1 FS_NAND_x8_Configure()

Description

Configures the parameters of the NAND flash device for a NAND physical layer of type FS_NAND_PHY_x8.

Prototype

```
void FS_NAND_x8_Configure(U8       Unit,
                          unsigned NumBlocks,
                          unsigned PagesPerBlock,
                          unsigned BytesPerPage,
                          unsigned BytesPerSpareArea);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)
NumBlocks	Total number of blocks in the NAND flash device.
PagesPerBlock	Total number of pages in a NAND block.
BytesPerPage	Number of bytes in a page without the spare area.
BytesPerSpareArea	Number of bytes in the spare area of a NAND page.

Additional information

This function is mandatory only when the file system is built with FS_NAND_SUPPORT_AUTO_DETECTION set to 0 which is not the default. FS_NAND_x_Configure() has to be called once in FS_X_AddDevices() for each instance of the FS_NAND_PHY_x8 physical layer. FS_NAND_x_Configure() is not available if FS_NAND_SUPPORT_AUTO_DETECTION is set to 0.

By default, the FS_NAND_PHY_x8 physical layer identifies the parameters of the NAND flash device by evaluating the first and second byte of the reply returned by the NAND flash device to the READ ID (0x90) command. The identification operation is disabled if FS_NAND_SUPPORT_AUTO_DETECTION set to 0 and the application must specify the NAND flash parameters via this function.

6.3.4.2.13.2 FS_NAND_x8_SetHWType()

Description

Configures the hardware access routines for a NAND physical layer of type FS_NAND_PHY_x8.

Prototype

```
void FS_NAND_x8_SetHWType(      U8                Unit,
                          const FS_NAND_HW_TYPE * pHWType);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)
pHWType	Type of the hardware layer to use. Cannot be NULL.

Additional information

This function has to be called once in FS_X_AddDevices() for every instance of a NAND physical layer of type FS_NAND_PHY_x8.

6.3.4.3 Physical layer API

The physical layers that come with emFile provide support most of the popular NAND flash device and target MCU types. Therefore, there is no need to modify any of the provided NAND physical layers. Typically, only the NAND hardware layer has to be adapted to a specific target hardware. However, when none of the provided NAND physical layers are compatible with the target hardware a new NAND physical layer implementation is required. This section provides information about the API of the NAND physical layer that helps to create a new physical layer from scratch or to modify an existing one.

The API of the physical layer is implemented as a structure of type FS_NAND_PHY_TYPE that contains pointers to functions. The following sections describe these functions in detail together with the data structure passed to these functions as parameters.

6.3.4.3.1 FS_NAND_PHY_TYPE

Description

NAND physical layer API.

Type definition

```
typedef struct {
  FS_NAND_PHY_TYPE_ERASE_BLOCK          * pfEraseBlock;
  FS_NAND_PHY_TYPE_INIT_GET_DEVICE_INFO * pfInitGetDeviceInfo;
  FS_NAND_PHY_TYPE_IS_WP                * pfIsWP;
  FS_NAND_PHY_TYPE_READ                 * pfRead;
  FS_NAND_PHY_TYPE_READ_EX              * pfReadEx;
  FS_NAND_PHY_TYPE_WRITE                * pfWrite;
  FS_NAND_PHY_TYPE_WRITE_EX             * pfWriteEx;
  FS_NAND_PHY_TYPE_ENABLE_ECC           * pfEnableECC;
  FS_NAND_PHY_TYPE_DISABLE_ECC          * pfDisableECC;
  FS_NAND_PHY_TYPE_CONFIGURE_ECC        * pfConfigureECC;
  FS_NAND_PHY_TYPE_COPY_PAGE            * pfCopyPage;
  FS_NAND_PHY_TYPE_GET_ECC_RESULT       * pfGetECCResult;
  FS_NAND_PHY_TYPE_DEINIT               * pfDeInit;
  FS_NAND_PHY_TYPE_SET_RAW_MODE         * pfSetRawMode;
} FS_NAND_PHY_TYPE;
```

Structure members

Member	Description
pfEraseBlock	Erases a NAND block.
pfInitGetDeviceInfo	Initializes the physical layer.
pfIsWP	Checks the write protection status of NAND flash device.
pfRead	Reads data from a NAND page.
pfReadEx	Reads data from a NAND page.
pfWrite	Writes data to a NAND page.
pfWriteEx	Writes data to a NAND page.
pfEnableECC	Enables the hardware ECC.
pfDisableECC	Disables the hardware ECC.
pfConfigureECC	Configures the hardware ECC.
pfCopyPage	Copies a NAND page.
pfGetECCResult	Returns the result of bit correction via ECC.
pfDeInit	Frees allocated resources.
pfSetRawMode	Enables or disables the raw operation mode.

6.3.4.3.2 FS_NAND_PHY_TYPE_ERASE_BLOCK

Description

Erases a NAND block.

Type definition

```
typedef int FS_NAND_PHY_TYPE_ERASE_BLOCK(U8  Unit,
                                         U32 PageIndex);
```

Parameters

Parameter	Description
Unit	Index of the NAND physical layer instance (0-based)
PageIndex	Index of the first page in the NAND block to be erased (0-based)

Return value

= 0 OK, the NAND block has been erased.
≠ 0 An error occurred.

Additional information

This function is a member of the NAND physical layer API and is mandatory to be implemented by each NAND physical layer. It is called by the NAND driver to set to 1 all the bits of a NAND block. A NAND block is the smallest erasable unit of a NAND flash device.

The index of the actual NAND block to be erased depends on the number of pages stored in a NAND block. For example if the NAND block contains 64 pages, then the PageIndex parameter passed by the NAND driver to the function has to be be interpreted as follows:

- 0 - NAND block 0
- 64 - NAND block 1
- 128 - NAND block 2
- etc.

6.3.4.3.3 FS_NAND_PHY_TYPE_INIT_GET_DEVICE_INFO

Description

Initializes the physical layer.

Type definition

```
typedef int FS_NAND_PHY_TYPE_INIT_GET_DEVICE_INFO(U8                   Unit,
                                                  FS_NAND_DEVICE_INFO * pDevInfo);
```

Parameters

Parameter	Description
Unit	Index of the NAND physical layer instance (0-based)
pDevInfo	out Information about the NAND flash device.

Return value

= 0 OK, physical layer has been initialized.
≠ 0 An error occurred.

Additional information

This function is a member of the NAND physical layer API and it is mandatory to be implemented by each NAND physical layer. It is the first function of the physical layer API that is called by the NAND driver when the NAND flash device is mounted.

This function initializes hardware layer, resets and tries to identify the NAND flash device. If the NAND flash device can be handled, the pDevInfo is filled with information about the organization and the ECC requirements of the NAND flash device.

6.3.4.3.4 FS_NAND_PHY_TYPE_IS_WP

Description

Checks if the NAND flash device is write protected.

Type definition

```
typedef int FS_NAND_PHY_TYPE_IS_WP(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND physical layer instance (0-based)

Return value

= 0 Data stored on the NAND flash device can be modified.
≠ 0 Data stored on the NAND flash device cannot be modified.

Additional information

This function is a member of the NAND physical layer API and it is mandatory to be implemented by each NAND physical layer.

The write protection status is checked by evaluating the bit 7 of the NAND status register. Typical reason for write protection is that either the supply voltage is too low or the /WP-pin is connected to ground.

6.3.4.3.5 FS_NAND_PHY_TYPE_READ

Description

Reads data from a NAND page.

Type definition

```
typedef int FS_NAND_PHY_TYPE_READ(U8       Unit,
                                  U32      PageIndex,
                                  void   * pData,
                                  unsigned Off,
                                  unsigned NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the NAND physical layer instance (0-based)
PageIndex	Index of the NAND page to read from (0-based).
pData	out Data read from NAND page.
Off	Byte offset to read from.
NumBytes	Number of bytes to be read.

Return value

= 0 OK, data read.
≠ 0 An error occurred.

Additional information

This function is a member of the NAND physical layer API and it is mandatory to be implemented by each NAND physical layer. The NAND driver uses FS_NAND_PHY_TYPE_READ to read data from the main as well as from the spare area of a page.

6.3.4.3.6 FS_NAND_PHY_TYPE_READ_EX

Description

Reads data from a NAND page.

Type definition

```
typedef int FS_NAND_PHY_TYPE_READ_EX(U8       Unit,
                                     U32      PageIndex,
                                     void   * pData0,
                                     unsigned Off0,
                                     unsigned NumBytes0,
                                     void   * pData1,
                                     unsigned Off1,
                                     unsigned NumBytes1);
```

Parameters

Parameter	Description
Unit	Index of the NAND physical layer instance (0-based)
PageIndex	Index of the NAND page to read from (0-based).
pData0	out Data read from Off0 of the NAND page.
Off0	Byte offset to read from for pData0.
NumBytes0	Number of bytes to be read.
pData1	out Data read from Off1 of a NAND page.
Off1	Byte offset to read from for pData1.
NumBytes1	Number of bytes to be read from Off1.

Return value

= 0 OK, data read.
≠ 0 An error occurred.

Additional information

This function is a member of the NAND physical layer API and it is mandatory to be implemented by each NAND physical layer. FS_NAND_PHY_TYPE_READ_EX is typically used by the NAND driver to read the data from main and spare area of a page at the same time.

6.3.4.3.7 FS_NAND_PHY_TYPE_WRITE

Description

Writes data to a NAND page.

Type definition

```
typedef int FS_NAND_PHY_TYPE_WRITE(      U8       Unit,
                                         U32      PageIndex,
                                   const void   * pData,
                                         unsigned Off,
                                         unsigned NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the NAND physical layer instance (0-based)
PageIndex	Index of the NAND page to write to (0-based).
pData	in Data to be written to the NAND page.
Off	Byte offset to write to.
NumBytes	Number of bytes to be written.

Return value

= 0 OK, data written.
≠ 0 An error occurred.

Additional information

This function is a member of the NAND physical layer API and it is mandatory to be implemented only by NAND physical layer that are working with the SLC1 NAND driver. The Universal NAND driver does not call this function. FS_NAND_PHY_TYPE_WRITE is used by the SLC1 NAND driver to write data to the main as well as to the spare area of a page.

6.3.4.3.8 FS_NAND_PHY_TYPE_WRITE_EX

Description

Writes data to a NAND page.

Type definition

```
typedef int FS_NAND_PHY_TYPE_WRITE_EX(      U8       Unit,
                                            U32      PageIndex,
                                      const void   * pData,
                                            unsigned Off,
                                            unsigned NumBytes,
                                      const void   * pSpare,
                                            unsigned OffSpare,
                                            unsigned NumBytesSpare);
```

Parameters

Parameter	Description
Unit	Index of the NAND physical layer instance (0-based)
PageIndex	Index of the NAND page to write to (0-based).
pData0	in Data to be written to the NAND page at Off0.
Off0	Byte offset to write to for pData0.
NumBytes0	Number of bytes to be written at Off0.
pData1	in Data to be written to the NAND page at Off1.
Off1	Byte offset to write to for pData1.
NumBytes1	Number of bytes to be written at Off1.

Return value

= 0 OK, data written.
≠ 0 An error occurred.

Additional information

This function is a member of the NAND physical layer API and it is mandatory to be implemented by each NAND physical layer. The NAND driver uses FS_NAND_PHY_TYPE_WRITE_EX to write data to the main and spare area of a page at the same time.

6.3.4.3.9 FS_NAND_PHY_TYPE_ENABLE_ECC

Description

Enables the hardware ECC.

Type definition

```
typedef int FS_NAND_PHY_TYPE_ENABLE_ECC(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND physical layer instance (0-based)

Return value

= 0 OK, hardware ECC activated.
≠ 0 An error occurred.

Additional information

This function is a member of the NAND physical layer API. It has to be implemented by the NAND physical layers that provide hardware support for bit error correction either on MCU via a dedicated NAND flash controller or via on-die ECC of the NAND flash device. FS_NAND_PHY_TYPE_ENABLE_ECC is called only by the Universal NAND driver.

After the call to this function the Universal NAND driver expects that FS_NAND_PHY_TYPE_READ and FS_NAND_PHY_TYPE_READ_EX return corrected data that is without bit errors. In addition, the Universal NAND driver expects that FS_NAND_PHY_TYPE_WRITE and FS_NAND_PHY_TYPE_WRITE_EX calculate and store the ECC to NAND flash device.

6.3.4.3.10 FS_NAND_PHY_TYPE_DISABLE_ECC

Description

Deactivates the hardware ECC.

Type definition

```
typedef int FS_NAND_PHY_TYPE_DISABLE_ECC(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND physical layer instance (0-based)

Return value

= 0 OK, hardware ECC deactivated.
≠ 0 An error occurred.

Additional information

This function is a member of the NAND physical layer API. It has to be implemented by the NAND physical layers that provide hardware support for bit error correction. FS_NAND_PHY_TYPE_DISABLE_ECC is called only by the Universal NAND driver.

After the call to this function the Universal NAND driver expects that FS_NAND_PHY_TYPE_READ and FS_NAND_PHY_TYPE_READ_EX return data that might contain bit errors. In addition, the Universal NADN driver expects that FS_NAND_PHY_TYPE_WRITE and FS_NAND_PHY_TYPE_WRITE_EX store the data without calculating the ECC.

6.3.4.3.11 FS_NAND_PHY_TYPE_CONFIGURE_ECC

Description

Configures the hardware ECC.

Type definition

```
typedef int FS_NAND_PHY_TYPE_CONFIGURE_ECC(U8  Unit,
                                           U8  NumBitsCorrectable,
                                           U16 BytesPerECCBlock);
```

Parameters

Parameter	Description
Unit	Index of the NAND physical layer instance (0-based)
NumBitsCorrectable	Maximum number of bit errors the hardware ECC has to be able to correct.
BytesPerECCBlock	Number of consecutive data bytes protected by a single ECC.

Return value

= 0 OK, hardware ECC configured.
≠ 0 An error occurred.

Additional information

This function is a member of the NAND physical layer API. It has to be implemented by the NAND physical layers that provide hardware support for bit error correction with configurable ECC strength. FS_NAND_PHY_TYPE_CONFIGURE_ECC is called only by the Universal NAND driver.

6.3.4.3.12 FS_NAND_PHY_TYPE_COPY_PAGE

Description

Copies the contents of an entire page to another page.

Type definition

```
typedef int FS_NAND_PHY_TYPE_COPY_PAGE(U8  Unit,
                                       U32 PageIndexSrc,
                                       U32 PageIndexDest);
```

Parameters

Parameter	Description
Unit	Index of the NAND physical layer instance (0-based)
PageIndexSrc	Index of the page to copy from.
PageIndexDest	Index of the page to copy to.

Return value

= 0 OK, page copied.
≠ 0 An error occurred or operation not supported.

Additional information

This function is a member of the NAND physical layer API. It may be be implemented by the NAND physical layers that can provide a faster method of copying the contents of a page than by first reading the source page contents to MCU and then by writing the contents to the destination page. One such method is the internal page copy operation supported by some NAND flash device. FS_NAND_PHY_TYPE_COPY_PAGE is called only by the Universal NAND driver.

6.3.4.3.13 FS_NAND_PHY_TYPE_GET_ECC_RESULT

Description

Returns the error correction status of the last read page.

Type definition

```
typedef int FS_NAND_PHY_TYPE_GET_ECC_RESULT(U8                    Unit,
                                            FS_NAND_ECC_RESULT * pResult);
```

Parameters

Parameter	Description
Unit	Index of the NAND physical layer instance (0-based)
pResult	out Information about the correction status.

Return value

= 0 OK, information returned.
≠ 0 An error occurred or operation not supported.

Additional information

This function is a member of the NAND physical layer API. It may be implemented by the NAND physical layers that can provide information about the status of the bit correction operation and about the number of bit errors corrected. FS_NAND_PHY_TYPE_GET_ECC_RESULT is called only by the Universal NAND driver.

The information returned by FS_NAND_PHY_TYPE_GET_ECC_RESULT is used by the Universal NAND driver to decide when a NAND block has to be relocated in order to prevent bit correction errors. A bit correction error occurs when the number of bit errors is greater than the number of bit errors the ECC is able to correct. Typically, a bit correction error causes a data loss.

6.3.4.3.14 FS_NAND_PHY_TYPE_DEINIT

Description

Releases the allocated resources.

Type definition

```
typedef void FS_NAND_PHY_TYPE_DEINIT(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND physical layer instance (0-based)

Additional information

This function is a member of the NAND physical layer API. It has to be implemented by the NAND physical layers that allocate resources dynamically during the initialization such as memory for the instance.

The NAND driver calls this function when the file system is unmounted.

6.3.4.3.15 FS_NAND_PHY_TYPE_SET_RAW_MODE

Description

Enables or disables the data translation.

Type definition

```
typedef int FS_NAND_PHY_TYPE_SET_RAW_MODE(U8 Unit,
                                          U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the NAND physical layer instance (0-based)
OnOff	Activation status of the feature.

Return value

= 0 OK, status changed.
≠ 0 An error occurred or operation not supported.

Additional information

This function is a member of the NAND physical layer API. It may be implemented by the NAND physical layers that store the data to NAND flash device using a different layout than the NAND driver. FS_NAND_PHY_TYPE_SET_RAW_MODE is not called by the NAND driver during the normal operation. It is called only by specific functions of the Universal NAND driver such as FS_NAND_UNI_WritePageRaw() and FS_NAND_UNI_ReadPageRaw().

6.3.4.3.16 FS_NAND_DEVICE_INFO

Description

Information about the NAND flash device.

Type definition

```
typedef struct {
  U8                BPP_Shift;
  U8                PPB_Shift;
  U16               NumBlocks;
  U16               BytesPerSpareArea;
  FS_NAND_ECC_INFO  ECC_Info;
  U8                DataBusWidth;
  U8                BadBlockMarkingType;
} FS_NAND_DEVICE_INFO;
```

Structure members

Member	Description
BPP_Shift	Bytes per page as a power of two value.
PPB_Shift	Pages per block as a power of two value.
NumBlocks	Total number of blocks in the NAND flash device.
BytesPerSpareArea	Number of bytes in the spare area.
ECC_Info	Information about the ECC capability required by the NAND flash device.
DataBusWidth	Number of lines used for exchanging the data with the NAND flash device.
BadBlockMarkingType	Specifies how the blocks are marked as defective.

Additional information

The initialization function of the physical layer FS_NAND_PHY_TYPE_INIT_GET_DEVICE_INFO uses this structure to return information about the NAND flash device to the NAND driver.

Typical values for BPP_Shift are 9 and 11 for NAND flash devices with a page size (without the spare area) of 512 and 2048 bytes respectively.

BytesPerSpareArea is typically 1/32 of the page size (2^BPP_Shift) but some NAND flash devices have a spare area larger than this. For example Micron MT29F8G08ABABA has a spare area of 224 bytes for a page size of 4096 bytes.

DataBusWith can take the following values:

- 0 - unknown
- 1 - SPI
- 8 - parallel 8-bit
- 16 - parallel 16-bit

Refer to *Bad block marking types* on page 414 for a list of permitted values for BadBlock-MarkingType.

6.3.4.3.17 FS_NAND_ECC_INFO

Description

Information about the ECC used to protect data stored on the NAND flash.

Type definition

```
typedef struct {
  U8  NumBitsCorrectable;
  U8  ldBytesPerBlock;
  U8  HasHW_ECC;
} FS_NAND_ECC_INFO;
```

Structure members

Member	Description
NumBitsCorrectable	Number of bits the ECC should be able to correct.
ldBytesPerBlock	Number of bytes the ECC should protect.
HasHW_ECC	Set to 1 if the NAND flash has HW internal ECC.

6.3.4.3.18 FS_NAND_ECC_RESULT

Description

Information about the ECC number of bits corrected in an ECC block.

Type definition

```
typedef struct {
  U8  CorrectionStatus;
  U8  MaxNumBitsCorrected;
} FS_NAND_ECC_RESULT;
```

Structure members

Member	Description
CorrectionStatus	Indicates if the correction succeeded or failed.
MaxNumBitsCorrected	Maximum number of bit errors detected and corrected in any ECC block of a page.

Additional information

This structure is filled by the FS_NAND_PHY_TYPE_GET_ECC_RESULT function of the NAND physical layer to return the result of the bit error correction operation.

Refer to *ECC correction status* on page 415 for a list of permitted values for CorrectionStatus.

An ECC block is the number of bytes protected by a single ECC. Typically, the ECC block is 512 bytes large, therefor the number of ECC blocks in a 2 KByte page is 4. Most of the NAND flash devices report the number of bits corrected in each of the ECC blocks. The value stored to MaxNumBitsCorrected must be the maximum of these values.

6.3.4.4 Additional physical layer functions

The following functions are optional. They can be called to get information about the NAND flash device and to control additional functionality of physical layers.

Function	Description
FS_NAND_PHY_ReadDeviceId()	Returns the id information stored in a NAND flash device.
FS_NAND_PHY_ReadONFIPara()	Reads the ONFI parameters from a NAND flash.
FS_NAND_PHY_SetHWType()	Configures the hardware access routines for FS_NAND_PHY_ReadDeviceId() and FS_NAND_PHY_ReadONFIPara().

6.3.4.4.1 FS_NAND_PHY_ReadDeviceId()

Description

Returns the id information stored in a NAND flash device.

Prototype

```
int FS_NAND_PHY_ReadDeviceId(U8    Unit,
                             U8  * pId,
                             U32   NumBytes);
```

Parameters

Parameter	Description
Unit	Index of HW layer connected to NAND flash.
pId	out Identification data read from NAND flash.
NumBytes	Number of bytes to read.

Return value

= 0 OK, id information read.
≠ 0 An error occurred.

Additional information

FS_NAND_PHY_ReadDeviceId() executes the READ ID command to read the id information from the NAND flash device. NumBytes specifies the number of bytes to be read. Refer to the data sheet of the NAND flash device for additional information about the meaning of the data returned by the NAND flash device. Typically, the first byte stores the manufactured id while the second byte provides information about the organization of the NAND flash device.

It is permitted to call FS_NAND_PHY_ReadONFIPara()from FS_X_AddDevices() since it does not require for the file system to be fully initialized and it invokes only functions of the NAND hardware layer. No instance of NAND driver is required to invoke this function.

Typical usage is to determine at runtime the type of NAND driver to be used for the connected NAND flash device.

Example

The following example shows how an application can select at runtime a different NAND drivers based on the type of the used NAND flash.

```
#include "FS.h"

#define ALLOC_SIZE            0x4000    // Memory pool for the file system in bytes

static U32 _aMemBlock[ALLOC_SIZE / 4];  // Memory pool used for semi-dynamic allocation.

/*********************************************************************
*
*       FS_X_AddDevices
*/
void FS_X_AddDevices(void) {
  U8 Id;

  FS_AssignMemory(_aMemBlock, sizeof(_aMemBlock));
  //
  // Read the first byte of the identification array.
  // This byte stores the manufacturer type.
  //
  FS_NAND_PHY_SetHWType(0, &FS_NAND_HW_Default);
  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);
    FS_NAND_2048x8_SetHWType(0, &FS_NAND_HW_Default);
  } 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);
    FS_NAND_ONFI_SetHWType(0, &FS_NAND_HW_Default);
  } else {
    //
    // NAND flash from another manufacturer, use auto-identification.
    //
    FS_AddDevice(&FS_NAND_Driver);
    FS_NAND_SetPhyType(0, &FS_NAND_PHY_x8);
    FS_NAND_x8_SetHWType(0, &FS_NAND_HW_Default);
  }
}
```


6.3.4.4.2 FS_NAND_PHY_ReadONFIPara()

Description

Reads the ONFI parameters from a NAND flash.

Prototype

```
int FS_NAND_PHY_ReadONFIPara(U8     Unit,
                             void * pPara);
```

Parameters

Parameter	Description
Unit	Index of HW layer connected to NAND flash.
pPara	out Data of ONFI parameter page read from NAND flash. This parameter can be set to NULL.

Return value

= 0 ONFI parameters read.
≠ 0 ONFI is not supported by the NAND flash.

Additional information

Refer to the data sheet of the NAND flash device for a description of the data layout of the returned ONFI parameters.

This function can be used to read the ONFI parameter stored in a NAND flash. It is permitted to call FS_NAND_PHY_ReadONFIPara()from FS_X_AddDevices() since it does not require for the file system to be fully initialized and it invokes only functions of the NAND hardware layer. No instance of NAND driver is required to invoke this function.

FS_NAND_PHY_ReadONFIPara() can also be used to check if the NAND flash device is ONFI compliant by setting pPara to NULL.

The size of the buffer passed via pParam must be at least 256 bytes large.

Example

This example demonstrates how an application can configure at runtime different NAND driver based on the type of the NAND flash.

```
#include "FS.h"

#define ALLOC_SIZE            0x4000    // Memory pool for the file system in bytes

static U32 _aMemBlock[ALLOC_SIZE / 4];  // Memory pool used for semi-dynamic allocation.

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

  FS_AssignMemory(_aMemBlock, sizeof(_aMemBlock));
  //
  // Check whether the NAND flash supports ONFI.
  //
  FS_NAND_PHY_SetHWType(0, &FS_NAND_HW_Default);
  r = FS_NAND_PHY_ReadONFIPara(0, NULL);
  if (r != 0) {
    //
    // 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);
  }
}
```


6.3.4.4.3 FS_NAND_PHY_SetHWType()

Description

Configures the hardware access routines for FS_NAND_PHY_ReadDeviceId() and FS_NAND_PHY_ReadONFIPara().

Prototype

```
void FS_NAND_PHY_SetHWType(      U8                  Unit,
                           const FS_NAND_HW_TYPE * pHWType);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)
pHWType	Type of the hardware layer to use. Cannot be NULL.

Additional information

This function is mandatory if the application calls either FS_NAND_PHY_ReadDeviceId() or FS_NAND_PHY_ReadONFIPara(). FS_NAND_PHY_SetHWType() has to be called once in FS_X_AddDevices() for every different Unit number passed to FS_NAND_PHY_ReadDeviceId() or FS_NAND_PHY_ReadONFIPara().

6.3.4.5 Resource usage

This section describes the ROM and RAM usage of the NAND physical layers.

6.3.4.5.1 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 was measured using the SEGGER Embedded Studio IDE V4.20 configured to generate code for a Cortex-M4 CPU in Thumb mode and with the size optimization enabled. The following table lists the ROM usage of all the available NOR physical layers.

Name	ROM usage(Kbytes)
512x8	
2048x8	
Small 2048x8	
2048x16	
4096x8	
DataFlash	
ONFI	
ONFI_RO	
Small ONFI	
QSPI	
SPI	
8/16-bit data bus	
8-bit data	

Physical layer	Description	ROM [Kbytes]
FS_NAND_PHY_512x8	Physical layer for small NAND devices with an 8-bit interface.	1.1

Physical layer	Description	ROM [Kbytes]
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

6.3.4.5.2 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. The next table lists the static RAM usage of all the available NOR physical layers.

Name	RAM usage(bytes)
512x8	
2048x8	
Small 2048x8	
2048x16	
4096x8	
DataFlash	
ONFI	
ONFI_RO	
Small ONFI	
QSPI	
SPI	
8/16-bit data bus	
8-bit data	

6.3.4.5.3 Dynamic RAM usage

Dynamic RAM usage is the amount of RAM allocated by the physical layer at runtime. The amount of RAM required depends on the compile time and runtime configuration. The following table lists the the dynamic RAM usage of the available NOR physical layers.

Name	RAM usage(bytes)
512x8	
2048x8	
Small 2048x8	
2048x16	
4096x8	
DataFlash	
ONFI	
ONFI_RO	
Small ONFI	
QSPI	

Name	RAM usage(bytes)
SPI	
8/16-bit data bus	
8-bit data	

6.3.5 NAND hardware layer

6.3.5.1 General information

The NAND hardware layer provides functionality for accessing a NAND flash device via the target hardware such as external memory controller, GPIO, SPI, etc. The functions of the NAND hardware layer are called by the NAND physical layer to exchange commands and data with a NAND flash device. Since these functions are hardware dependent, they have to be implemented by the user. emFile comes with template hardware layers and sample implementations for popular evaluation boards that can be used as starting point for implementing new hardware layers. The relevant files are located in the `/Sample/FS/Driver/NAND` folder of the emFile shipment.

6.3.5.2 Hardware layer types

The functions of the NAND hardware layer are organized in a function table implemented a C structure. Different hardware layer types are provided to support different ways of interfacing a NAND flash device. The type of hardware layer an application has to use depends on the type NAND physical layer configured. The following table shows what hardware layer is required by each physical layer.

NAND hardware layer	NAND physical layer
FS_NAND_HW_TYPE	FS_NAND_PHY_512x8 FS_NAND_PHY_2048x8 FS_NAND_PHY_2048x16 FS_NAND_PHY_4096x8 FS_NAND_PHY_x FS_NAND_PHY_x8 FS_NAND_PHY_ONFI
FS_NAND_HW_TYPE_DF	FS_NAND_PHY_DataFlash
FS_NAND_HW_TYPE_QSPI	FS_NAND_PHY_QSPI
FS_NAND_HW_TYPE_SPI	FS_NAND_PHY_SPI

6.3.5.3 Hardware layer API - FS_NAND_HW_TYPE

This NAND hardware layer supports NAND flash devices that can be accessed via an 8- or 16-bit data bus. The functions of this hardware layer are grouped in a structure of type FS_NAND_HW_TYPE. The following sections describe these functions in detail.

6.3.5.3.1 FS_NAND_HW_TYPE

Description

NAND hardware layer API for NAND flash devices connected via parallel I/O.

Type definition

```
typedef struct {
  FS_NAND_HW_TYPE_INIT_X8         * pfInit_x8;
  FS_NAND_HW_TYPE_INIT_X16        * pfInit_x16;
  FS_NAND_HW_TYPE_DISABLE_CE      * pfDisableCE;
  FS_NAND_HW_TYPE_ENABLE_CE       * pfEnableCE;
  FS_NAND_HW_TYPE_SET_ADDR_MODE   * pfSetAddrMode;
  FS_NAND_HW_TYPE_SET_CMD_MODE    * pfSetCmdMode;
  FS_NAND_HW_TYPE_SET_DATA_MODE   * pfSetDataMode;
  FS_NAND_HW_TYPE_WAIT_WHILE_BUSY * pfWaitWhileBusy;
  FS_NAND_HW_TYPE_READ_X8         * pfRead_x8;
  FS_NAND_HW_TYPE_WRITE_X8        * pfWrite_x8;
  FS_NAND_HW_TYPE_READ_X16        * pfRead_x16;
  FS_NAND_HW_TYPE_WRITE_X16       * pfWrite_x16;
} FS_NAND_HW_TYPE;
```

Structure members

Member	Description
pfInit_x8	Initializes a NAND flash device with an 8-bit data interface.
pfInit_x16	Initializes a NAND flash device with a 16-bit data interface.
pfDisableCE	Disables the NAND flash device.
pfEnableCE	Enables the NAND flash device.
pfSetAddrMode	Initiates the transfer of an address.
pfSetCmdMode	Initiates the transfer of a command.
pfSetDataMode	Initiates the transfer of data.
pfWaitWhileBusy	Waits for the NAND flash device to become ready.
pfRead_x8	Reads a specified number of bytes from NAND flash device via the 8-bit data interface.
pfWrite_x8	Writes a specified number of bytes to NAND flash device via the 8-bit data interface.
pfRead_x16	Reads a specified number of bytes from NAND flash device via the 16-bit data interface.
pfWrite_x16	Writes a specified number of bytes to NAND flash device via the 16-bit data interface.

6.3.5.3.2 FS_NAND_HW_TYPE_INIT_X8

Description

Initializes the hardware for 8-bit mode access.

Type definition

```
typedef void FS_NAND_HW_TYPE_INIT_X8(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)

Additional information

This function is a member of the FS_NAND_PHY_TYPE NAND hardware layer API and it has to be implemented by any NAND hardware layer that accesses a NAND flash device via an 8-bit data bus. FS_NAND_HW_TYPE_INIT_X8 is the first function of the hardware layer API that is called by a NAND physical layer during the mounting of the file system.

This function has to perform any initialization of the MCU hardware required to access the NAND flash device such as clocks, port pins, memory controllers, etc.

6.3.5.3.3 FS_NAND_HW_TYPE_INIT_X16

Description

Initializes the hardware for 16-bit mode access.

Type definition

```
typedef void FS_NAND_HW_TYPE_INIT_X16(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)

Additional information

This function is a member of the FS_NAND_PHY_TYPE NAND hardware layer API. The implementation of this function is optional for systems that interface to the NAND flash device via an 8-bit data bus. FS_NAND_HW_TYPE_INIT_X16 must be implemented for systems that have the NAND flash device connected to MCU via an 16-bit data bus.

FS_NAND_HW_TYPE_INIT_X16 is the first function of the hardware layer API that is called by the NAND physical layer when the NAND flash device is mounted.

6.3.5.3.4 FS_NAND_HW_TYPE_DISABLE_CE

Description

Disables the NAND flash device.

Type definition

```
typedef void FS_NAND_HW_TYPE_DISABLE_CE(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)

Additional information

This function is a member of the FS_NAND_PHY_TYPE NAND hardware layer API and it is mandatory to be implemented. The implementation of this function can be left empty if the hardware is driving the Chip Enable (CE) signal. Typically, the NAND flash device is disabled by driving the CE signal to a logic-low level.

6.3.5.3.5 FS_NAND_HW_TYPE_ENABLE_CE

Description

Enables the NAND flash device.

Type definition

```
typedef void FS_NAND_HW_TYPE_ENABLE_CE(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)

Additional information

This function is a member of the FS_NAND_PHY_TYPE NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer. The implementation of this function can be left empty if the hardware is driving the Chip Enable (CE) signal. Typically, the NAND flash device is enabled by driving the CE signal to a logic-high level.

6.3.5.3.6 FS_NAND_HW_TYPE_SET_ADDR_MODE

Description

Changes the data access to address mode.

Type definition

```
typedef void FS_NAND_HW_TYPE_SET_ADDR_MODE(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)

Additional information

This function is a member of the FS_NAND_PHY_TYPE NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer.

After the call to this function the NAND hardware layer has to make sure that any data sent to NAND flash device is interpreted as address information. This can be achieved by setting the Address Latch Enable signal (ALE) signal to logic-high and the Command Latch Enable (CLE) signal to logic-low.

6.3.5.3.7 FS_NAND_HW_TYPE_SET_CMD_MODE

Description

Changes the data access to command mode.

Type definition

```
typedef void FS_NAND_HW_TYPE_SET_CMD_MODE(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)

Additional information

This function is a member of the FS_NAND_PHY_TYPE NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer.

After the call to this function the NAND hardware layer has to make sure that any data sent to NAND flash device is interpreted as command information. This can be achieved by setting the Command Latch Enable (CLE) signal to logic-high and the Address Latch Enable signal (ALE) signal to logic-low.

6.3.5.3.8 FS_NAND_HW_TYPE_SET_DATA_MODE

Description

Changes the data access to data mode.

Type definition

```
typedef void FS_NAND_HW_TYPE_SET_DATA_MODE(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)

Additional information

This function is a member of the FS_NAND_PHY_TYPE NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer.

After the call to this function the NAND hardware layer has to make sure that any data sent to NAND flash device is interpreted as data information. This can be achieved by setting the Command Latch Enable (CLE) and Address Latch Enable signals to logic-low.

6.3.5.3.9 FS_NAND_HW_TYPE_WAIT_WHILE_BUSY

Description

Waits for the NAND flash device to become ready.

Type definition

```
typedef int FS_NAND_HW_TYPE_WAIT_WHILE_BUSY(U8       Unit,
                                            unsigned us);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)
us	Maximum time to wait in microseconds.

Return value

= 0 The NAND flash device is ready.
≠ 0 The NAND flash device is busy or the operation is not supported.

Additional information

This function is a member of the FS_NAND_PHY_TYPE NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer.

A NAND hardware layer calls this function every time it checks the status of the NAND flash device. A typical implementation uses the status of the Ready/Busy (R/B) signal that is set to logic-low by the NAND flash device as long as it is busy and it cannot accept and other commands from MCU.

FS_NAND_HW_TYPE_WAIT_WHILE_BUSY must return 1 if the status of the NAND flash device cannot be queried via R/B signal. In this case, the NAND hardware layer checks the status via a read status command.

Typically, a NAND flash device does not set R/B signal to logic-low immediately after it accepts a command from MCU but only after a time interval labeled as tWB in the data sheet. This means that FS_NAND_HW_TYPE_WAIT_WHILE_BUSY has to wait for tWB time interval to elapse before it samples the R/B signal for the first time in order to make sure that it returns correct status information to the NAND physical layer.

6.3.5.3.10 FS_NAND_HW_TYPE_READ_X8

Description

Reads data from NAND flash device via 8-bit data bus.

Type definition

```
typedef void FS_NAND_HW_TYPE_READ_X8(U8       Unit,
                                     void   * pData,
                                     unsigned NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)
pData	out Data read from NAND flash. It cannot be NULL.
NumBytes	Number of bytes to be read. It cannot be 0.

Additional information

This function is a member of the FS_NAND_PHY_TYPE NAND hardware layer API. It has to be implemented by a NAND hardware layer that exchanges the data with the NAND flash device via an 8-bit data bus.

FS_NAND_HW_TYPE_READ_X8 is called by a NAND physical layer in data mode to transfer data from NAND flash device to MCU. It is not called in address and command data access modes.

The transfer of the data is controlled by the MCU using the Read Enable (RE) signal. If the NAND hardware layer exchanges the data via GPIO it has to make sure that the timing of the RE signal meets the specifications of the NAND flash device.

6.3.5.3.11 FS_NAND_HW_TYPE_WRITE_X8

Description

Writes data to NAND flash device via 8-bit data bus.

Type definition

```
typedef void FS_NAND_HW_TYPE_WRITE_X8(      U8       Unit,
                                      const void   * pData,
                                            unsigned NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)
pData	in Data to be written to NAND flash. It cannot be NULL.
NumBytes	Number of bytes to be read. It cannot be 0.

Additional information

This function is a member of the FS_NAND_PHY_TYPE NAND hardware layer API. It has to be implemented by a NAND hardware layer that exchanges the data with the NAND flash device via an 8-bit data bus.

FS_NAND_HW_TYPE_WRITE_X8 is called by a NAND physical layer in all data access modes to transfer data from MCU to NAND flash device.

The transfer of the data is controlled by the MCU using the Write Enable (WE) signal. If the NAND hardware layer exchanges the data via GPIO it has to make sure that the timing of the WE signal meets the specifications of the NAND flash device.

6.3.5.3.12 FS_NAND_HW_TYPE_READ_X16

Description

Reads data from NAND flash device via 16-bit data bus.

Type definition

```
typedef void FS_NAND_HW_TYPE_READ_X16(U8       Unit,
                                      void   * pData,
                                      unsigned NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)
pData	out Data read from NAND flash. It cannot be NULL.
NumBytes	Number of bytes to be read. It cannot be 0.

Additional information

This function is a member of the FS_NAND_PHY_TYPE NAND hardware layer API. It has to be implemented by a NAND hardware layer that exchanges the data with the NAND flash device via an 16-bit data bus.

FS_NAND_HW_TYPE_READ_X16 is called by a NAND physical layer in data mode to transfer data from NAND flash device to MCU. It is not called in address and command data access modes.

The transfer of the data is controlled by the MCU using the Read Enable (RE) signal. If the NAND hardware layer exchanges the data via GPIO it has to make sure that the timing of the RE signal meets the specifications of the NAND flash device.

pData is aligned to a half-word (2-byte) boundary.

6.3.5.3.13 FS_NAND_HW_TYPE_WRITE_X16

Description

Writes data to NAND flash device via 16-bit data bus.

Type definition

```
typedef void FS_NAND_HW_TYPE_WRITE_X16(      U8       Unit,
                                       const void   * pdata,
                                             unsigned NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)
pData	in Data to be written to NAND flash. It cannot be NULL.
NumBytes	Number of bytes to be read. It cannot be 0.

Additional information

This function is a member of the FS_NAND_PHY_TYPE NAND hardware layer API. It has to be implemented by a NAND hardware layer that exchanges the data with the NAND flash device via an 16-bit data bus.

FS_NAND_HW_TYPE_WRITE_X16 is called by a NAND physical layer in all data access modes to transfer data from MCU to NAND flash device.

The transfer of the data is controlled by the MCU using the Write Enable (WE) signal. If the NAND hardware layer exchanges the data via GPIO it has to make sure that the timing of the WE signal meets the specifications of the NAND flash device.

pData is aligned to a half-word (2-byte) boundary.

6.3.5.3.14 Sample implementation

The following sample implementation uses the GPIO ports of an NXP K66 MCU to interface with a NAND flash device. This NAND hardware layer was tested on the SGGER emPower board (https://www.segger.com/evaluate-our-software/segger/empower/)

```
/*********************************************************************
*                   (c) SEGGER Microcontroller GmbH                  *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File    : FS_NAND_HW_K66_SEGGER_emPower.c
Purpose : NAND flash hardware layer header file for the SEGGER emPower
          V2 evaluation board.
Literature:
  [1] K66 Sub-Family Reference Manual
      (\\FILESERVER\Techinfo\Company\Freescale\MCU\Kinetis_K-series
\K66P144M180SF5RMV2_Rev2_1505.pdf)
  [2] emPower Evaluation and prototyping platform for SEGGER software User Guide & Reference
 Manual
      (\\FILESERVER\Product\Doc4Review\UM06001_emPower.pdf)
*/

/*********************************************************************
*
*       #include Section
*
**********************************************************************
*/
#include "FS.h"
#include "FS_NAND_HW_K66_SEGGER_emPower.h"
#include "MK66F18.h"

/*********************************************************************
*
*       Defines, configurable
*
**********************************************************************
*/
#ifndef   FS_NAND_HW_USE_OS
  #define FS_NAND_HW_USE_OS       0         // Enables / disables the
                                            // event-driven operation.
#endif

/*********************************************************************
*
*       #include section, conditional
*
**********************************************************************
*/
#if FS_NAND_HW_USE_OS
  #include "RTOS.h"
  #include "FS_OS.h"
#endif // FS_NAND_HW_USE_OS

/*********************************************************************
*
*        Defines, fixed
*
**********************************************************************
*/

/*********************************************************************
*
*        Port pins
*/
#define NAND_CS_PIN               0
#define NAND_WAIT_PIN             1
#define NAND_ALE_PIN              4
#define NAND_CLE_PIN              5
#define NAND_nOE_PIN              6
#define NAND_nWE_PIN              7
```



```
#define NAND_DATA_MASK            0xff
#define NAND_D0_SHIFT             8
#define NAND_D0_PIN               8
#define NAND_D1_PIN               9
#define NAND_D2_PIN               10
#define NAND_D3_PIN               11
#define NAND_D4_PIN               12
#define NAND_D5_PIN               13
#define NAND_D6_PIN               14
#define NAND_D7_PIN               15

/*********************************************************************
*
*       Macros for pin configuration
*/
#define SET_DATA2INPUT()          GPIOD_PDDR &= ~(NAND_DATA_MASK << NAND_D0_SHIFT)
#define SET_DATA2OUTPUT()         GPIOD_PDDR |=  (NAND_DATA_MASK << NAND_D0_SHIFT)

/*********************************************************************
*
*       Macros for pin control
*/
#define NAND_GET_DATA(Data)       Data = *((U8 *)&GPIOD_PDIR + 0x1);
#define NAND_SET_DATA(Data)       *((U8 *)((U8 *)&GPIOD_PDOR + 0x1)) = Data;
#define NAND_SET_ALE()            GPIOD_PSOR = (1 << NAND_ALE_PIN);
#define NAND_CLR_ALE()            GPIOD_PCOR = (1 << NAND_ALE_PIN);
#define NAND_SET_CLE()            GPIOD_PSOR = (1 << NAND_CLE_PIN);
#define NAND_CLR_CLE()            GPIOD_PCOR = (1 << NAND_CLE_PIN);
#define NAND_SET_CE()             GPIOD_PSOR = (1 << NAND_CS_PIN);
#define NAND_CLR_CE()             GPIOD_PCOR = (1 << NAND_CS_PIN);
#define NAND_SET_RE()             GPIOD_PSOR = (1 << NAND_nOE_PIN);
#define NAND_CLR_RE()             GPIOD_PCOR = (1 << NAND_nOE_PIN); _Delay(2);
#define NAND_SET_WE()             GPIOD_PSOR = (1 << NAND_nWE_PIN);
#define NAND_CLR_WE()             GPIOD_PCOR = (1 << NAND_nWE_PIN);
#define NAND_GET_BUSY()           (GPIOD_PDIR & (1 << NAND_WAIT_PIN))

/*********************************************************************
*
*       Misc. defines
*/
#define PORTD_IRQ_PRIO            15
#define WAIT_TIMEOUT_MS           1000      // Maximum time to wait for the NAND
                                            // flash device to become ready.

/*********************************************************************
*
*       Local functions
*
**********************************************************************
*/

/*********************************************************************
*
*       _Delay
*/
static void _Delay(register volatile int NumCycles) {
  do {
    __asm("nop");
  }  while (--NumCycles);
}

/*********************************************************************
*
*       Local functions (public through callback)
*
**********************************************************************
*/

/*********************************************************************
*
*       _HW_EnableCE
*/
static void _HW_EnableCE(U8 Unit) {
  FS_USE_PARA(Unit);
  NAND_CLR_CE();
}
```



```
/*********************************************************************
*
*       _HW_DisableCE
*/
static void _HW_DisableCE(U8 Unit) {
  FS_USE_PARA(Unit);
  NAND_SET_CE();
}

/*********************************************************************
*
*       _HW_SetDataMode
*/
static void _HW_SetDataMode(U8 Unit) {
  FS_USE_PARA(Unit);
  //
  // nCE low, CLE low, ALE low
  //
  NAND_CLR_CLE();
  NAND_CLR_ALE();
}

/*********************************************************************
*
*       _HW_SetCmdMode
*/
static void _HW_SetCmdMode(U8 Unit) {
  FS_USE_PARA(Unit);
  //
  // nCE low, CLE high, ALE low
  //
  NAND_SET_CLE();
  NAND_CLR_ALE();
}

/*********************************************************************
*
*       _HW_SetAddr
*/
static void _HW_SetAddrMode(U8 Unit) {
  FS_USE_PARA(Unit);
  //
  // nCE low, CLE low, ALE high
  //
  NAND_CLR_CLE();
  NAND_SET_ALE();
}

/*********************************************************************
*
*       _HW_Read_x8
*/
static void _HW_Read_x8(U8 Unit, void * p, unsigned NumBytes) {
  U8 * pData;

  FS_USE_PARA(Unit);
  pData = (U8 *)p;
  SET_DATA2INPUT();
  do {
    NAND_CLR_RE();        // RE is active low
    NAND_GET_DATA(*pData);
    pData++;
    NAND_SET_RE();        // disable RE
  } while (--NumBytes);
}

/*********************************************************************
*
*       _HW_Write_x8
*/
static void _HW_Write_x8(U8 Unit, const void * p, unsigned NumBytes) {
  const U8 * pData;

  FS_USE_PARA(Unit);
  pData = (const U8 *)p;
```



```
  SET_DATA2OUTPUT();
  do {
    NAND_CLR_WE();        // WE is active low
    NAND_SET_DATA(*pData);
    pData++;
    NAND_SET_WE();        // disable WE
  } while (--NumBytes);
}

/*********************************************************************
*
*       _HW_Init_x8
*/
static void _HW_Init_x8(U8 Unit) {
  FS_USE_PARA(Unit);
#if FS_NAND_HW_USE_OS
  NVIC_DisableIRQ(PORTD_IRQn);
  NVIC_SetPriority(PORTD_IRQn, PORTD_IRQ_PRIO);
#endif // FS_NAND_HW_USE_OS
  //
  // Add here the initialization of your NAND hardware
  //
  SIM_SCGC5   |= SIM_SCGC5_PORTD_MASK;               // Enable clock for Port D
  PORTD_PCR0   =  PORT_PCR_MUX(0x01);                // Configure as GPIO
  PORTD_PCR1   =  PORT_PCR_MUX(0x01)                 // Configure as GPIO
               |  PORT_PCR_PS_MASK                   // Enable the pull-up.
               |  PORT_PCR_PE_MASK
               ;
  PORTD_PCR4   =  PORT_PCR_MUX(0x01);                // Configure as GPIO
  PORTD_PCR5   =  PORT_PCR_MUX(0x01);                // Configure as GPIO
  PORTD_PCR6   =  PORT_PCR_MUX(0x01);                // Configure as GPIO
  PORTD_PCR7   =  PORT_PCR_MUX(0x01);                // Configure as GPIO
  PORTD_PCR8   =  PORT_PCR_MUX(0x01);                // Configure as GPIO
  PORTD_PCR9   =  PORT_PCR_MUX(0x01);                // Configure as GPIO
  PORTD_PCR10  =  PORT_PCR_MUX(0x01);                // Configure as GPIO
  PORTD_PCR11  =  PORT_PCR_MUX(0x01);                // Configure as GPIO
  PORTD_PCR12  =  PORT_PCR_MUX(0x01);                // Configure as GPIO
  PORTD_PCR13  =  PORT_PCR_MUX(0x01);                // Configure as GPIO
  PORTD_PCR14  =  PORT_PCR_MUX(0x01);                // Configure as GPIO
  PORTD_PCR15  =  PORT_PCR_MUX(0x01);                // Configure as GPIO
  GPIOD_PDDR  |=  (1u << NAND_CS_PIN)
              |   (1u << NAND_ALE_PIN)
              |   (1u << NAND_CLE_PIN)
              |   (1u << NAND_nOE_PIN)
              |   (1u << NAND_nWE_PIN)
              ;
  GPIOD_PDDR  &=  ~(1 << NAND_WAIT_PIN);
#if FS_NAND_HW_USE_OS
  NVIC_EnableIRQ(PORTD_IRQn);
#endif
}

/*********************************************************************
*
*       _HW_WaitWhileBusy
*/
static int _HW_WaitWhileBusy(U8 Unit, unsigned us) {
  FS_USE_PARA(Unit);
  FS_USE_PARA(us);

  //
  // Make sure that we do not sample the busy signal too early.
  // In order to do so we have to wait here at least the time
  // specified at tWB in the data sheet of the NAND flash device.
  // Typically this time is about 100 ns. Assuming that the CPU
  // is running at 168 MHz then we have to wait here 100 / 5.9 = 16.9
  // that is about 17 cycles.
  //
  _Delay(17);
  while (1) {
    if (NAND_GET_BUSY() != 0) {
      break;
    }
#if FS_NAND_HW_USE_OS
    PORTD_PCR1 |= PORT_PCR_IRQC(0xC);       // Generate an interrupt when the
                                            // the busy signal goes to HIGH.
```



```
    (void)FS_X_OS_Wait(WAIT_TIMEOUT_MS);
#else
    __asm("nop");
#endif // FS_NAND_HW_USE_OS
  }
  return 0;
}

/*********************************************************************
*
*      Public code
*
**********************************************************************
*/

#if FS_NAND_HW_USE_OS

/*********************************************************************
*
*       PORTD_IRQHandler
*/
void PORTD_IRQHandler(void);
void PORTD_IRQHandler(void) {
  OS_EnterNestableInterrupt();
  //
  // Disable the interrupt.
  //
  PORTD_PCR1 &= ~PORT_PCR_IRQC_MASK;          // Disable the interrupt.
  PORTD_PCR1 |=  PORT_PCR_ISF_MASK;
  FS_X_OS_Signal();
  OS_LeaveNestableInterrupt();
}

#endif // FS_NAND_HW_USE_OS

/*********************************************************************
*
*       Public data
*
**********************************************************************
*/

/*********************************************************************
*
*       FS_NAND_HW_K66_SEGGER_emPower
*/
const FS_NAND_HW_TYPE FS_NAND_HW_K66_SEGGER_emPower = {
  _HW_Init_x8,
  NULL,
  _HW_DisableCE,
  _HW_EnableCE,
  _HW_SetAddrMode,
  _HW_SetCmdMode,
  _HW_SetDataMode,
  _HW_WaitWhileBusy,
  _HW_Read_x8,
  _HW_Write_x8,
  NULL,
  NULL
};

/************************** End of file ****************************/
```


6.3.5.4 Hardware layer API - FS_NAND_HW_TYPE_DF

This hardware layer supports Microchip / Atmel / Adesto DataFlash devices and is used by the `FS_NAND_PHY_DataFlash` physical layer to exchange data with a DataFlash device via a standard SPI interface. The functions of this hardware layer are grouped in the structure of type `FS_NAND_HW_TYPE_DF`. The following sections describe these functions in detail.

6.3.5.4.1 FS_NAND_HW_TYPE_DF

Description

NAND hardware layer API for DataFlash devices.

Type definition

```
typedef struct {
  FS_NAND_HW_TYPE_DF_INIT       * pfInit;
  FS_NAND_HW_TYPE_DF_ENABLE_CS  * pfEnableCS;
  FS_NAND_HW_TYPE_DF_DISABLE_CS * pfDisableCS;
  FS_NAND_HW_TYPE_DF_READ       * pfRead;
  FS_NAND_HW_TYPE_DF_WRITE      * pfWrite;
} FS_NAND_HW_TYPE_DF;
```

Structure members

Member	Description
pfInit	Initializes the hardware.
pfEnableCS	Enables the DataFlash device.
pfDisableCS	Disables the DataFlash device.
pfRead	Transfers a specified number of bytes from DataFlash device to MCU.
pfWrite	Transfers a specified number of bytes from MCU to DataFlash device.

6.3.5.4.2 FS_NAND_HW_TYPE_DF_INIT

Description

Initializes the hardware.

Type definition

```
typedef int FS_NAND_HW_TYPE_DF_INIT(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)

Return value

= 0 OK, initialization was successful.
≠ 0 An error occurred.

Additional information

This function is a member of the FS_NAND_HW_TYPE_DF NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer of this type. FS_NAND_HW_TYPE_DF_INIT is the first function of the hardware layer API that is called by a NAND physical layer during the mounting of the file system.

This function has to perform any initialization of the MCU hardware required to access the DataFlash device such as clocks, port pins, memory controllers, etc.

A DataFlash requires that the SPI communication protocol meets the following requirements:

- 8-bit data length.
- The most significant bit hast to be sent out first.
- Chip Select (CS) signal should be initially high to disable the DataFlash device.
- The SPI clock frequency does not have to exceed the maximum clock frequency that is specified by the DataFlash device (Usually: 20MHz).

6.3.5.4.3 FS_NAND_HW_TYPE_DF_ENABLE_CS

Description

Enables the DataFlash device.

Type definition

```
typedef void FS_NAND_HW_TYPE_DF_ENABLE_CS(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)

Additional information

This function is a member of the FS_NAND_HW_TYPE_DF NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer of this type.

The DataFlash device is enabled by driving the Chip Select (CS) signal to logic-low.

6.3.5.4.4 FS_NAND_HW_TYPE_DF_DISABLE_CS

Description

Disables the DataFlash device.

Type definition

```
typedef void FS_NAND_HW_TYPE_DF_DISABLE_CS(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)

Additional information

This function is a member of the FS_NAND_HW_TYPE_DF NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer of this type.

The DataFlash device is disabled by driving the Chip Select (CS) signal to logic-high.

6.3.5.4.5 FS_NAND_HW_TYPE_DF_READ

Description

Transfers a specified number of bytes from DataFlash device to MCU.

Type definition

```
typedef void FS_NAND_HW_TYPE_DF_READ(U8   Unit,
                                     U8 * pData,
                                     int  NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)
pData	out Data read from DataFlash device.
NumBytes	Number of bytes to be read.

Additional information

This function is a member of the FS_NAND_HW_TYPE_DF NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer of this type.

6.3.5.4.6 FS_NAND_HW_TYPE_DF_WRITE

Description

Transfers a specified number of bytes from MCU to DataFlash device.

Type definition

```
typedef void FS_NAND_HW_TYPE_DF_WRITE(      U8   Unit,
                                      const U8 * pData,
                                            int  NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)
pData	in Data to be written to DataFlash device.
NumBytes	Number of bytes to be written.

Additional information

This function is a member of the FS_NAND_HW_TYPE_DF NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer of this type.

6.3.5.4.7 Sample implementation

The following sample implementation uses the SPI controller of a NXP LPC4322 MCU to interface with a DataFlash device. This NAND hardware layer was tested on a SGGER internal test board.

```
/*********************************************************************
*                   (c) SEGGER Microcontroller GmbH                  *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File    : FS_NAND_HW_DF_LPC4322_SEGGER_QSPIFI_Test_Board.c
Purpose : DataFlash hardware layer for NXP LPC4322.
Literature:
  [1] UM10503 LPC43xx ARM Cortex-M4/M0 multi-core microcontroller
      (\\fileserver\Techinfo\Company\NXP\MCU\LPC43xx
\UserManual_LPC43xx_UM10503_Rev1.90_150218.pdf)
------------  END-OF-HEADER  ----------------------------------------
*/

#include "FS_Int.h"

/*********************************************************************
*
*      Defines, configurable
*
**********************************************************************
*/
#define   PER_CLK_HZ            180000000uL
  // Frequency of the clock supplied to SPI unit
#ifndef   SPI_CLK_HZ
  #define SPI_CLK_HZ            25000000uL
  // Frequency of the clock supplied to DataFlash device
#endif
#define   RESET_DELAY_LOOPS     100000
  // Number of software loops to wait for DataFlash to reset

/*********************************************************************
*
*       Defines, non-configurable
*
**********************************************************************
*/

/*********************************************************************
*
*       SPI unit
*/
#define SPI_BASE_ADDR         0x40100000
#define SPI_CR                (*(volatile U32*)(SPI_BASE_ADDR + 0x00))
  // SPI Control Register
#define SPI_SR                (*(volatile U32*)(SPI_BASE_ADDR + 0x04))
  // SPI Status Register
#define SPI_DR                (*(volatile U32*)(SPI_BASE_ADDR + 0x08))
  // SPI Data Register
#define SPI_CCR               (*(volatile U32*)(SPI_BASE_ADDR + 0x0C))
  // SPI Clock Counter Register
#define SPI_TCR               (*(volatile U32*)(SPI_BASE_ADDR + 0x10))
  // SPI Test Control register
#define SPI_TSR               (*(volatile U32*)(SPI_BASE_ADDR + 0x14))
  // SPI Test Status register
#define SPI_INT               (*(volatile U32*)(SPI_BASE_ADDR + 0x1C))
  // SPI Interrupt Flag

/*********************************************************************
*
*       Clock generation unit
*/
#define CGU_BASE_ADDR         0x40050000
#define BASE_SPI_CLK          (*(volatile U32*)(CGU_BASE_ADDR + 0x0074))

/*********************************************************************
```



```
*
*       Clock control unit 1
*/
#define CCU1_BASE_ADDR        0x40051000
#define CLK_SPI_CFG           (*(volatile U32*)(CCU1_BASE_ADDR + 0x0A00))
#define CLK_SPI_STAT          (*(volatile U32*)(CCU1_BASE_ADDR + 0x0A04))

/*********************************************************************
*
*       System control unit
*/
#define SCU_BASE_ADDR         0x40086000
#define SFSP3_2               (*(volatile U32*)(SCU_BASE_ADDR + 0x188))
  // Pin configuration register for pin P3_2
#define SFSP3_3               (*(volatile U32*)(SCU_BASE_ADDR + 0x18C))
  // Pin configuration register for pin P3_3
#define SFSP3_6               (*(volatile U32*)(SCU_BASE_ADDR + 0x198))
  // Pin configuration register for pin P3_6
#define SFSP3_7               (*(volatile U32*)(SCU_BASE_ADDR + 0x19C))
  // Pin configuration register for pin P3_7
#define SFSP3_8               (*(volatile U32*)(SCU_BASE_ADDR + 0x1A0))
  // Pin configuration register for pin P3_8

/*********************************************************************
*
*       GPIO unit
*/
#define GPIO_BASE_ADDR        0x400F4000
#define GPIO_DIR5             (*(volatile U32*)(GPIO_BASE_ADDR + 0x2014))
  // Direction registers port 5
#define GPIO_SET5             (*(volatile U32*)(GPIO_BASE_ADDR + 0x2214))
  // Set register for port 5
#define GPIO_CLR5             (*(volatile U32*)(GPIO_BASE_ADDR + 0x2294))
  // Clear register for port 5

/*********************************************************************
*
*       Misc. defines
*/
#define CGU_IDIV_BIT            2
#define CGU_AUTOBLOCK_BIT       11
#define CGU_CLK_SEL_BIT         24
#define SFS_MODE_BIT            0
#define SFS_EPUN_BIT            4
#define SFS_EHS_BIT             5
#define SFS_EZI_BIT             6
#define SFS_ZIF_BIT             7
#define CLK_STAT_RUN_BIT        0
#define CLK_CFG_RUN_BIT         0
#define NOR_CS_BIT              11
#define CR_MSTR_BIT             5
#define SR_SPIF                 7
#define CCR_COUNTER_MAX         0xFE        // The actual maximum value is
 0xFF but it has to be a multiple of 2
#define NOR_RESET_BIT           9

/*********************************************************************
*
*      Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       _SetSpeed
*
*  Function description
*    Configures the speed of the clock supplied NOR flash device.
*
*  Return value
*    Frequency of the SPI clock in Hz.
*/
static U32 _SetSpeed(unsigned MaxFreq_Hz) {
  U32 Div;
  U32 Freq_Hz;
```



```
  //
  // According to [1] the clock divisor has to be >= 8 and a multiple of 2.
  //
  Div = 8;
  while (1) {
    Freq_Hz = PER_CLK_HZ / Div;
    if (Freq_Hz <= MaxFreq_Hz) {
      break;
    }
    Div += 2;
    if (Div >= CCR_COUNTER_MAX) {
      Div = CCR_COUNTER_MAX;
      Freq_Hz = PER_CLK_HZ / Div;
      break;
    }
  }
  SPI_CCR = Div;
  return Freq_Hz;
}

/*********************************************************************
*
*       _TransferByte
*
*  Function description
*    Exchanges 8 bits of data with the NOR flash.
*
*  Parameters
*    Data       Data to be sent to NOR flash.
*
*  Return value
*    The data received from NOR flash
*/
static U8 _Transfer8Bits(U8 Data) {
  SPI_DR = Data;
  while (1) {
    if (SPI_SR & (1uL << SR_SPIF)) {
      break;
    }
  }
  Data = SPI_DR;
  return Data;
}

/*********************************************************************
*
*      Public code (via callback)
*
**********************************************************************
*/

/*********************************************************************
*
*       _HW_Init
*
*  Function description
*    Initialize the hardware before accessing the device.
*
*  Parameters
*    Unit     Device index
*
*  Return Value
*    ==0      Hardware successfully initialized
*    !=0      An error occurred
*/
static int _HW_Init(U8 Unit) {
  unsigned NumLoops;

  FS_USE_PARA(Unit);    // This device has only one HW unit.
  BASE_SPI_CLK = 0
                 | 9uL  << CGU_CLK_SEL_BIT      // Use PLL1 as clock generator
                 | 1uL  << CGU_AUTOBLOCK_BIT
                 ;
  //
  // Enable the SPI clock if required.
```



```
  //
  if ((CLK_SPI_STAT & (1uL << CLK_STAT_RUN_BIT)) == 0) {
    CLK_SPI_CFG |= (1uL << CLK_CFG_RUN_BIT);
    while (1) {
      if (CLK_SPI_STAT & (1uL << CLK_CFG_RUN_BIT)) {
        ;
      }
    }
  }
  //
  // Clock pin (SPI_CLK).
  //
  SFSP3_3 = 0
          | (1uL << SFS_MODE_BIT)       // Controlled by SPI unit.
          | (1uL << SFS_EPUN_BIT)
          | (1uL << SFS_EHS_BIT)
          | (1uL << SFS_EZI_BIT)
          ;
  //
  // Master In Slave Out (SPI_MISO).
  //
  SFSP3_6 = 0
          | (1uL << SFS_MODE_BIT)       // Controlled by SPI unit.
          | (1uL << SFS_EPUN_BIT)
          | (1uL << SFS_EZI_BIT)
          ;
  //
  // Master Out Slave In (SPI_MOSI).
  //
  SFSP3_7 = 0
          | (1uL << SFS_MODE_BIT)       // Controlled by SPI unit.
          | (1uL << SFS_EPUN_BIT)
          | (1uL << SFS_EZI_BIT)
          ;
  //
  // Chip select pin (SPI_CS).
  //
  SFSP3_8 = 0
          | (4uL << SFS_MODE_BIT)       // GPIO controlled by the this hardware layer.
          | (1uL << SFS_EPUN_BIT)
          ;
  GPIO_DIR5 |= 1uL << NOR_CS_BIT;
  GPIO_SET5  = 1uL << NOR_CS_BIT;       // Set to idle state.
  //
  // Reset pin. Present only on 16-pin devices.
  //
  SFSP3_2 = 0
          | (4uL << SFS_MODE_BIT)                   // Controlled by this HW layer.
          | (1uL << SFS_EPUN_BIT)
          ;
  GPIO_DIR5 |= 1uL << NOR_RESET_BIT;
  //
  // Reset the NOR device.
  //
  GPIO_CLR5 = 1uL << NOR_RESET_BIT;                 // Assert reset signal (active low).
  NumLoops = RESET_DELAY_LOOPS;
  do {
    ;
  } while (--NumLoops);
  GPIO_SET5 = 1uL << NOR_RESET_BIT;                 // De-assert reset signal (active low).
  //
  // Configure the SPI unit.
  //
  SPI_CR = 0
         | (1uL << CR_MSTR_BIT)
         ;
  (void)_SetSpeed(SPI_CLK_HZ);
  return 0;
}

/*********************************************************************
*
*       _HW_EnableCS
*
*  Function description
*    Sets the device active using the chip select (CS) line.
```



```
*
*  Parameters
*    Unit     Device index
*/
static void _HW_EnableCS(U8 Unit) {
  FS_USE_PARA(Unit);
  GPIO_CLR5 = 1uL << NOR_CS_BIT;      // The CS signal is active low.
}

/*********************************************************************
*
*       _HW_DisableCS
*
*  Function description
*    Sets the device inactive using the chip select (CS) line.
*
*  Parameters
*    Unit     Device index
*/
static void _HW_DisableCS(U8 Unit) {
  FS_USE_PARA(Unit);
  GPIO_SET5 = 1uL << NOR_CS_BIT;      // The CS signal is active low.
}

/*********************************************************************
*
*       _HW_Read
*
*  Function description
*    Reads a specified number of bytes from device.
*
*  Parameters
*    Unit       Device index
*    pData      Pointer to a data buffer
*    NumBytes   Number of bytes
*/
static void _HW_Read(U8 Unit, U8 * pData, int NumBytes) {
  FS_USE_PARA(Unit);
  do {
    *pData++ = _Transfer8Bits(0xFF);
  } while (--NumBytes);
}

/*********************************************************************
*
*       _HW_Write
*
*  Function description
*    Writes a specified number of bytes from data buffer to device.
*
*  Parameters
*    Unit         Device index
*    pData        Pointer to a data buffer
*    NumBytes     Number of bytes
*/
static void _HW_Write(U8 Unit, const U8 * pData, int NumBytes) {
  FS_USE_PARA(Unit);
  do {
    (void)_Transfer8Bits(*pData++);
  } while (--NumBytes);
}

/*********************************************************************
*
*       Public data
*
**********************************************************************
*/
const FS_NAND_HW_TYPE_DF FS_NAND_HW_DF_LPC4322_SEGGER_QSPIFI_Test_Board = {
  _HW_Init,
  _HW_EnableCS,
  _HW_DisableCS,
  _HW_Read,
  _HW_Write
};
```



```
/*************************** End of file ****************************/
```


6.3.5.5 Hardware layer API - FS_NAND_HW_TYPE_QSPI

This hardware layer supports any serial NAND flash device with a standard, dual or quad SPI interface. It is used by the FS_NAND_PHY_QSPI physical layer to exchange data with a NAND flash device that supports such an interface. Typically, the implementation of FS_NAND_HW_TYPE_QSPI hardware layer makes use of a SPI controller that is able to exchange the data via two or four data lines. This kind of SPI controllers are found on most of the popular MCUs and they are typically used to execute code in-place from an external NOR flash device but they can be equally well used to exchange data with a serial NAND flash device.

The functions of this hardware layer are grouped in the structure of type FS_NAND_HW_TYPE_QSPI. The following sections describe these functions in detail.

6.3.5.5.1 FS_NAND_HW_TYPE_QSPI

Description

NAND hardware layer API for NAND flash devices connected via quad and dual SPI.

Type definition

```
typedef struct {
  FS_NAND_HW_TYPE_QSPI_INIT       * pfInit;
  FS_NAND_HW_TYPE_QSPI_EXEC_CMD   * pfExecCmd;
  FS_NAND_HW_TYPE_QSPI_READ_DATA  * pfReadData;
  FS_NAND_HW_TYPE_QSPI_WRITE_DATA * pfWriteData;
  FS_NAND_HW_TYPE_QSPI_POLL       * pfPoll;
  FS_NAND_HW_TYPE_QSPI_DELAY      * pfDelay;
  FS_NAND_HW_TYPE_QSPI_LOCK       * pfLock;
  FS_NAND_HW_TYPE_QSPI_UNLOCK     * pfUnlock;
} FS_NAND_HW_TYPE_QSPI;
```

Structure members

Member	Description
pfInit	Initializes the hardware.
pfExecCmd	Executes a command without data transfer.
pfReadData	Transfers data from NAND flash device to MCU.
pfWriteData	Transfers data from MCU to NAND flash device.
pfPoll	Sends a command repeatedly and checks the response data for a specified condition.
pfDelay	Blocks the execution for the specified number of milliseconds.
pfLock	Requests exclusive access to SPI bus.
pfUnlock	Releases the exclusive access of SPI bus.

6.3.5.5.2 FS_NAND_HW_TYPE_QSPI_INIT

Description

Initializes the hardware.

Type definition

```
typedef int FS_NAND_HW_TYPE_QSPI_INIT(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)

Return value

Frequency of the QSPI clock in kHz.

Additional information

This function is a member of the FS_NAND_HW_TYPE_QSPI NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer of this type.

This function has to configure anything necessary for the data transfer via QSPI such as clocks, port pins, QSPI peripheral, DMA, etc.

The frequency value returned by the function has to be greater than or equal to the actual clock frequency used to transfer the data via QSPI. The FS_NAND_PHY_QSPI physical layer uses this value to calculate the number of software cycles it has to wait for NAND flash device to finish an operation before a timeout error is reported. FS_NAND_HW_TYPE_QSPI_INIT must return 0 if the clock frequency is unknown. In this case, the FS_NAND_PHY_QSPI physical layer waits indefinitely for the end of a NAND flash operation and it never reports a timeout error.

6.3.5.5.3 FS_NAND_HW_TYPE_QSPI_EXEC_CMD

Description

Executes a command without data transfer.

Type definition

```
typedef int FS_NAND_HW_TYPE_QSPI_EXEC_CMD(U8 Unit,
                                          U8 Cmd,
                                          U8 BusWidth);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)
Cmd	Code of the command to be sent.
BusWidth	Number of data lines to be used for sending the command code.

Return value

= 0 OK, command sent.
≠ 0 An error occurred.

Additional information

This function is a member of the FS_NAND_HW_TYPE_QSPI NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer of this type.

FS_NAND_HW_TYPE_QSPI_EXEC_CMD must to send a 8 bits to NAND flash device containing the value specified via Cmd.

BusWidth specifies the number of data lines to be used when sending the command code. Permitted values are:

- 1 for standard SPI
- 2 for quad SPI (2 data lines)
- 4 for quad SPI (4 data lines)

Please note that this is not an encoded value such as the value passed via BusWidth in a call to FS_NAND_HW_TYPE_QSPI_READ_DATA, FS_NAND_HW_TYPE_QSPI_WRITE_DATA or FS_NAND_HW_TYPE_QSPI_POLL.

6.3.5.5.4 FS_NAND_HW_TYPE_QSPI_READ_DATA

Description

Transfers data from NAND flash device to MCU.

Type definition

```
typedef int FS_NAND_HW_TYPE_QSPI_READ_DATA(      U8       Unit,
                                                 U8       Cmd,
                                           const U8     * pPara,
                                                 unsigned NumBytesPara,
                                                 unsigned NumBytesAddr,
                                                 U8     * pData,
                                                 unsigned NumBytesData,
                                                 U16      BusWidth);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)
Cmd	Code of the command to be sent.
pPara	Additional command parameters (can be NULL).
NumBytesPara	Number of additional bytes to be sent after command. Can be 0 if pPara is NULL.
NumBytesAddr	Number of address bytes to be sent. Can be 0 if pPara is NULL.
pData	Data received from the NAND flash device.
NumBytesData	Number of bytes to be received from the NAND flash device.
BusWidth	Specifies the number of data lines to be used during the data transfer.

Return value

= 0 OK, data transferred.
≠ 0 An error occurred.

Additional information

This function is a member of the FS_NAND_HW_TYPE_QSPI NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer of this type.

The data has to be sent in this order: Cmd (1 byte) and pPara (NumBytesPara bytes). The first NumBytesAddr bytes in pPara represent the address bytes while the rest until NumBytesPara bytes are dummy bytes. These values are useful for QSPI hardware that can differentiate between address and dummy bytes. If NumBytesPara and NumBytesAddr are equal, no dummy bytes have to be sent. NumBytesPara is never be smaller than NumBytesAddr. The data received from NAND flash has to be stored to pData (NumBytesData bytes).

The number of data lines to be used during the data transfer is specified via BusWidth. The value is encoded with each nibble (4 bits) specifying the number of data lines for one part of the data transfer. The macros *SPI bus width decoding* on page 529 can be used to decode the number of data lines of each part of the request (command and address) and of the response (data). The dummy bytes have to be sent using the number of data lines returned by FS_BUSWIDTH_GET_ADDR().

6.3.5.5.5 FS_NAND_HW_TYPE_QSPI_WRITE_DATA

Description

Transfers data from MCU to NAND flash device.

Type definition

```
typedef int FS_NAND_HW_TYPE_QSPI_WRITE_DATA(      U8       Unit,
                                                  U8       Cmd,
                                            const U8     * pPara,
                                                  unsigned NumBytesPara,
                                                  unsigned NumBytesAddr,
                                            const U8     * pData,
                                                  unsigned NumBytesData,
                                                  U16      BusWidth);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)
Cmd	Code of the command to be sent.
pPara	Additional command parameters (can be NULL).
NumBytesPara	Number of additional bytes to be sent after command. Can be 0 if pPara is NULL.
NumBytesAddr	Number of address bytes to be sent. Can be 0 if pPara is NULL.
pData	Data to be sent to NAND flash device.
NumBytesData	Number of data bytes to be sent to NAND flash device.
BusWidth	Specifies the number of data lines to be used during the data transfer.

Additional information

This function is a member of the FS_NAND_HW_TYPE_QSPI NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer of this type.

The data has to be sent in this order: Cmd (1 byte), pPara (NumBytesPara bytes), pData (NumBytesData bytes). The first NumBytesAddr bytes in pPara represent the address bytes while the rest until NumBytesPara bytes are dummy bytes. These values are useful for QSPI hardware that can differentiate between address and dummy bytes. If NumBytesPara and NumBytesAddr are equal, no dummy bytes have to be sent. NumBytesPara will never be smaller than NumBytesAddr.

The number of data lines to be used during the data transfer is specified via BusWidth. The value is encoded with each nibble specifying the number of data lines for one part of the data transfer. The macros *SPI bus width decoding* on page 529 can be used to decode the number of data lines of each part of the request (command, address and data). The dummy bytes have to be sent using the number of data lines returned by FS_BUSWIDTH_GET_ADDR().

6.3.5.5.6 FS_NAND_HW_TYPE_QSPI_POLL

Description

Sends a command repeatedly and checks the response data for a specified condition.

Type definition

```
typedef int FS_NAND_HW_TYPE_QSPI_POLL(      U8       Unit,
                                            U8       Cmd,
                                      const U8     * pPara,
                                            unsigned NumBytesPara,
                                            U8       BitPos,
                                            U8       BitValue,
                                            U32      Delay,
                                            U32      TimeOut_ms,
                                            U16      BusWidth);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)
Cmd	Code of the command to be sent.
pPara	Additional command parameters (can be NULL).
NumBytesPara	Number of additional bytes to be sent after command. Can be 0 if pPara is NULL.
BitPos	Position of the bit inside response data that has to be checked (0-based, with 0 being LSB)
BitValue	Bit value to wait for.
Delay_ms	Time between two command executions.
TimeOut_ms	Maximum time to wait for the bit at BitPos to be set to BitValue.
BusWidth	Specifies how many data lines have to be used for sending the command and parameters and for reading the data.

Return value

> 0	Error, timeout occurred.
= 0	OK, bit set to specified value.
< 0	Error, feature not supported.

Additional information

This function is a member of the FS_NAND_HW_TYPE_QSPI NAND hardware layer API. The implementation of this function is optional. FS_NAND_HW_TYPE_QSPI_POLL is called by the QSPI NAND physical layer when it has to wait for the NAND flash device to reach a certain status.

FS_NAND_HW_TYPE_QSPI_POLL has to read periodically a byte of data from the NAND flash device until the the value of the bit at BitPos in the in the returned data byte is set to a value specified by BitValue. The data is read by sending a command with a code specified by Cmd followed by an optional number of additional bytes specified by pPara and NumBytesPara. Then 8 bits of data are transferred from NAND flash device to MCU and this the value stores the bit to be checked.

The number of data lines to be used during the data transfer is specified via BusWidth. The value is encoded with each nibble specifying the number of data lines for one part of the data transfer. The macros *SPI bus width decoding* on page 529 can be used to decode the number of data lines of each part of the request (command, parameters and data). The pPara data has to be sent using the number of data lines returned by FS_BUSWIDTH_GET_ADDR().

The time FS_NAND_HW_TYPE_QSPI_POLL waits for the condition to be true shall not exceed the number of milliseconds specified by TimeOut_ms. Delay_ms specifies the number of milliseconds that can elapse between the execution of two consecutive commands.

6.3.5.5.7 FS_NAND_HW_TYPE_QSPI_DELAY

Description

Blocks the execution for the specified number of milliseconds.

Type definition

```
typedef void FS_NAND_HW_TYPE_QSPI_DELAY(U8  Unit,
                                        int ms);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)
ms	Number of milliseconds to wait.

Additional information

This function is a member of the FS_NAND_HW_TYPE_QSPI NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer of this type.

The time FS_NAND_HW_TYPE_QSPI_DELAY blocks does not have to be accurate. That is the function can block the execution longer that the number of specified milliseconds but no less than that.

6.3.5.5.8 FS_NAND_HW_TYPE_QSPI_LOCK

Description

Requests exclusive access to SPI bus.

Type definition

```
typedef void FS_NAND_HW_TYPE_QSPI_LOCK(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)

Additional information

This function is a member of the FS_NAND_HW_TYPE_QSPI NAND hardware layer API. The implementation of this function is optional.

The FS_NAND_PHY_QSPI physical layer calls this function to indicate that it needs exclusive to access the NAND flash device via the SPI bus. It is guaranteed that the FS_NAND_PHY_QSPI physical layer does not attempt to communicate via the SPI bus before calling this function first. It is also guaranteed that FS_NAND_HW_TYPE_QSPI_LOCK and FS_NAND_HW_TYPE_QSPI_UNLOCK are called in pairs.

FS_NAND_HW_TYPE_QSPI_UNLOCK and FS_NAND_HW_TYPE_QSPI_LOCK can be used to synchronize the access to the SPI bus when other devices than the NAND flash are connected to it. A possible implementation would make use of an OS semaphore that is acquired in this function and released in FS_NAND_HW_TYPE_QSPI_UNLOCK.

6.3.5.5.9 FS_NAND_HW_TYPE_QSPI_UNLOCK

Description

Releases the exclusive access of SPI bus.

Type definition

```
typedef void FS_NAND_HW_TYPE_QSPI_UNLOCK(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)

Additional information

This function is a member of the FS_NAND_HW_TYPE_QSPI NAND hardware layer API. The implementation of this function is optional.

The FS_NAND_PHY_QSPI physical layer calls this function when it no longer needs to access the NAND flash device via the SPI bus. It is guaranteed that the FS_NAND_PHY_QSPI physical layer does not attempt to communicate via the SPI bus before calling FS_NAND_HW_TYPE_QSPI_LOCK. It is also guaranteed that FS_NAND_HW_TYPE_QSPI_UNLOCK and FS_NAND_HW_TYPE_QSPI_LOCK are called in pairs.

FS_NAND_HW_TYPE_QSPI_UNLOCK and FS_NAND_HW_TYPE_QSPI_LOCK can be used to synchronize the access to the SPI bus when other devices than the NAND flash are connected to it. A possible implementation would make use of an OS semaphore that is acquired in this function and released in FS_NAND_HW_TYPE_QSPI_UNLOCK.

6.3.5.5.10 SPI bus width decoding

Description

Macros for the handling of SPI bus width.

Definition

```
#define FS_BUSWIDTH_GET_CMD(BusWidth)     (((BusWidth) >> FS_BUSWIDTH_CMD_SHIFT)
  & FS_BUSWIDTH_MASK)
#define FS_BUSWIDTH_GET_ADDR(BusWidth)
  (((BusWidth) >> FS_BUSWIDTH_ADDR_SHIFT) & FS_BUSWIDTH_MASK)
#define FS_BUSWIDTH_GET_DATA(BusWidth)
  (((BusWidth) >> FS_BUSWIDTH_DATA_SHIFT) & FS_BUSWIDTH_MASK)
```

Symbols

Definition	Description
FS_BUSWIDTH_GET_CMD(BusWidth)	Returns the number of data lines to be used for sending the command code.
FS_BUSWIDTH_GET_ADDR(BusWidth)	Returns the number of data lines to be used for sending the data address.
FS_BUSWIDTH_GET_DATA(BusWidth)	Returns the number of data lines to be used for sending and receiving of the user data.

Additional information

The functions of the FS_NOR_HW_TYPE_SPIFI NOR hardware layer and of the FS_NAND_HW_TYPE_QSPI NAND hardware layer that exchange data with the storage device take a parameter that specifies how many data lines have to be used to send different parts of a request and to receive the answer. The value of this parameter is encoded as a 16-bit value and the macros in this section can help extract the bus width value for a specific part of the request and response.

The value is encoded as follows:

Bit range	Description
0-3	Number of data lines to be used to transfer the data.
4-7	Number of data lines to be used to transfer the address.
8-11	Number of data lines to be used to transfer the command.

6.3.5.5.11 Sample implementation

The sample implementation listed below uses the QUADSPI controller of an ST STM32H743 MCU to interface with a serial NAND flash device. This NAND hardware layer was tested on a SGGER internal test board.

```
/*********************************************************************
*                   (c) SEGGER Microcontroller GmbH                  *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File    : FS_NAND_HW_QSPI_STM32H743_SEGGER_QSPI_Flash_Evaluator.c
Purpose : HW layer for quad and dual serial NAND flash for ST STM32H743.
Literature:
  [1] RM0433 Reference manual STM32H7x3 advanced ARM-based 32-bit MCUs
    (\\FILESERVER\Techinfo\Company\ST\MCU\STM32\STM32H7\STM32H7x3_RM0433_Rev3.pdf)
  [2] STM32H753xI Errata sheet STM32H753xI rev Y device limitations
    (\\FILESERVER\Techinfo\Company\ST\MCU\STM32\STM32H7\STM32H753_Errata_Rev2.pdf)
*/

/*********************************************************************
*
*       #include section
*
**********************************************************************
*/
#include <stddef.h>
#include "FS.h"

/*********************************************************************
*
*       Defines, configurable
*
**********************************************************************
*/
#define PER_CLK_HZ              240000000         // Clock of Quad-SPI unit
#define NAND_CLK_HZ             48000000          // Frequency of the clock
                                                  // supplied to NAND flash device
#define NUM_CYCLES_PER_MS       10000             // Number of software loops
                                                  // create a 1ms delay

/*********************************************************************
*
*       Defines, fixed
*
**********************************************************************
*/

/*********************************************************************
*
*       QSPI registers
*/
#define QUADSPI_BASE_ADDR       0x52005000uL
#define QUADSPI_CR              (*(volatile U32 *)(QUADSPI_BASE_ADDR + 0x00))
#define QUADSPI_DCR             (*(volatile U32 *)(QUADSPI_BASE_ADDR + 0x04))
#define QUADSPI_SR              (*(volatile U32 *)(QUADSPI_BASE_ADDR + 0x08))
#define QUADSPI_FCR             (*(volatile U32 *)(QUADSPI_BASE_ADDR + 0x0C))
#define QUADSPI_DLR             (*(volatile U32 *)(QUADSPI_BASE_ADDR + 0x10))
#define QUADSPI_CCR             (*(volatile U32 *)(QUADSPI_BASE_ADDR + 0x14))
#define QUADSPI_AR              (*(volatile U32 *)(QUADSPI_BASE_ADDR + 0x18))
#define QUADSPI_ABR             (*(volatile U32 *)(QUADSPI_BASE_ADDR + 0x1C))
#define QUADSPI_DR              (*(volatile U32 *)(QUADSPI_BASE_ADDR + 0x20))
#define QUADSPI_DR_BYTE         (*(volatile U8  *)(QUADSPI_BASE_ADDR + 0x20))
#define QUADSPI_PSMKR           (*(volatile U32 *)(QUADSPI_BASE_ADDR + 0x24))
#define QUADSPI_PSMAR           (*(volatile U32 *)(QUADSPI_BASE_ADDR + 0x28))
#define QUADSPI_PIR             (*(volatile U32 *)(QUADSPI_BASE_ADDR + 0x2C))
#define QUADSPI_LPTR            (*(volatile U32 *)(QUADSPI_BASE_ADDR + 0x30))

/*********************************************************************
*
*       Port B registers
*/
```



```
#define GPIOB_BASE_ADDR         0x58020400uL
#define GPIOB_MODER             (*(volatile U32 *)(GPIOB_BASE_ADDR + 0x00))
#define GPIOB_OSPEEDR           (*(volatile U32 *)(GPIOB_BASE_ADDR + 0x08))
#define GPIOB_PUPDR             (*(volatile U32 *)(GPIOB_BASE_ADDR + 0x0C))
#define GPIOB_ODR               (*(volatile U32 *)(GPIOB_BASE_ADDR + 0x14))
#define GPIOB_AFRL              (*(volatile U32 *)(GPIOB_BASE_ADDR + 0x20))

/*********************************************************************
*
*       Port F registers
*/
#define GPIOF_BASE_ADDR         0x58021400uL
#define GPIOF_MODER             (*(volatile U32 *)(GPIOF_BASE_ADDR + 0x00))
#define GPIOF_OSPEEDR           (*(volatile U32 *)(GPIOF_BASE_ADDR + 0x08))
#define GPIOF_PUPDR             (*(volatile U32 *)(GPIOF_BASE_ADDR + 0x0C))
#define GPIOF_ODR               (*(volatile U32 *)(GPIOF_BASE_ADDR + 0x14))
#define GPIOF_AFRL              (*(volatile U32 *)(GPIOF_BASE_ADDR + 0x20))
#define GPIOF_AFRH              (*(volatile U32 *)(GPIOF_BASE_ADDR + 0x24))

/*********************************************************************
*
*       Reset and clock control
*/
#define RCC_BASE_ADDR           0x58024400uL
#define RCC_AHB3RSTR            (*(volatile U32*)(RCC_BASE_ADDR + 0x7C))
#define RCC_AHB3ENR             (*(volatile U32*)(RCC_BASE_ADDR + 0xD4))
#define RCC_AHB4ENR             (*(volatile U32*)(RCC_BASE_ADDR + 0xE0))

/*********************************************************************
*
*       Masks for the peripheral enable bits
*/
#define AHB1ENR_GPIOBEN         1
#define AHB1ENR_GPIOFEN         5
#define AHB3ENR_QSPIEN          14

/*********************************************************************
*
*       GPIO bit positions of SPI signals
*/
#define NOR_BK1_NCS_BIT         6   // Port B
#define NOR_CLK_BIT             10  // Port F
#define NOR_BK1_D0_BIT          8   // Port F
#define NOR_BK1_D1_BIT          9   // Port F
#define NOR_BK1_D2_BIT          7   // Port F
#define NOR_BK1_D3_BIT          6   // Port F

/*********************************************************************
*
*       GPIO bits and masks
*/
#define OSPEEDR_HIGH            2uL
#define OSPEEDR_MASK            0x3uL
#define MODER_MASK              0x3uL
#define MODER_ALT               2uL
#define AFR_MASK                0x3uL

/*********************************************************************
*
*       Quad-SPI control register
*/
#define CR_EN_BIT               0
#define CR_ABORT_BIT            1
#define CR_PRESCALER_BIT        24
#define CR_PRESCALER_MAX        255

/*********************************************************************
*
*       Quad-SPI device configuration register
*/
#define DCR_CKMODE_BIT          0
#define DCR_FSIZE_BIT           16
#define DCR_FSIZE_MAX           0x1FuL

/*********************************************************************
*
```



```
*       Quad-SPI status register
*/
#define SR_TEF_BIT              0
#define SR_TCF_BIT              1
#define SR_SMF_BIT              3
#define SR_TOF_BIT              4
#define SR_BUSY_BIT             5
#define SR_FLEVEL_BIT           8
#define SR_FLEVEL_MASK          0x3FuL

/*********************************************************************
*
*       Quad-SPI communication configuration register
*/
#define CCR_INTRUCTION_BIT      0
#define CCR_IMODE_BIT           8
#define CCR_MODE_NONE           0
#define CCR_MODE_SINGLE         1uL
#define CCR_MODE_DUAL           2uL
#define CCR_MODE_QUAD           3uL
#define CCR_ADMODE_BIT          10
#define CCR_ADSIZE_BIT          12
#define CCR_ADSIZE_MASK         0x03uL
#define CCR_ABMODE_BIT          14
#define CCR_ABSIZE_BIT          16
#define CCR_ABSIZE_MASK         0x03uL
#define CCR_DCYC_BIT            18
#define CCR_DMODE_BIT           24
#define CCR_FMODE_BIT           26
#define CCR_FMODE_WRITE         0x0uL
#define CCR_FMODE_READ          0x1uL
#define CCR_FMODE_MMAP          0x3uL
#define CCR_FMODE_MASK          0x3uL

/*********************************************************************
*
*       Misc. defines
*/
#define NUM_BYTES_FIFO          32

/*********************************************************************
*
*       ASSERT_IS_LOCKED
*/
#if FS_SUPPORT_TEST
  #define ASSERT_IS_LOCKED()                  \
    if (_LockCnt != 1) {                      \
      FS_X_PANIC(FS_ERRCODE_INVALID_USAGE);   \
    }
#else
  #define ASSERT_IS_LOCKED()
#endif

/*********************************************************************
*
*       IF_TEST
*/
#if FS_SUPPORT_TEST
  #define IF_TEST(Expr)       Expr
#else
  #define IF_TEST(Expr)
#endif

/*********************************************************************
*
*      Static data
*
**********************************************************************
*/
#if FS_SUPPORT_TEST
  static U8 _LockCnt = 0;      // Just to test if the Lock()/Unlock
                               // function are called correctly.
#endif

/*********************************************************************
*
```



```
*       Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       _CalcClockDivider
*/
static U8 _CalcClockDivider(U32 * pFreq_Hz) {
  U8  Div;
  U32 Freq_Hz;
  U32 MaxFreq_Hz;

  Div        = 0;
  MaxFreq_Hz = *pFreq_Hz;
  while (1) {
    Freq_Hz = PER_CLK_HZ / (Div + 1);
    if (Freq_Hz <= MaxFreq_Hz) {
      break;
    }
    ++Div;
    if (Div == CR_PRESCALER_MAX) {
      break;
    }
  }
  *pFreq_Hz = Freq_Hz;
  return Div;
}

/*********************************************************************
*
*       _GetMode
*/
static U32 _GetMode(unsigned BusWidth, U32 NumBytes) {
  U32 Mode;

  Mode = CCR_MODE_NONE;
  if (NumBytes) {
    switch (BusWidth) {
    case 1:
      Mode = CCR_MODE_SINGLE;
      break;
    case 2:
      Mode = CCR_MODE_DUAL;
      break;
    case 4:
      Mode = CCR_MODE_QUAD;
      break;
    default:
      break;
    }
  }
  return Mode;
}

/*********************************************************************
*
*       _ClearFlags
*/
static void _ClearFlags(void) {
  QUADSPI_FCR = 0
              | (1uL << SR_TEF_BIT)
              | (1uL << SR_TCF_BIT)
              | (1uL << SR_SMF_BIT)
              | (1uL << SR_TOF_BIT)
              ;
  //
  // Wait for the flags to be cleared.
  //
  while (1) {
    if ((QUADSPI_SR & ((1uL << SR_TEF_BIT) |
                       (1uL << SR_TCF_BIT) |
                       (1uL << SR_SMF_BIT) |
                       (1uL << SR_TOF_BIT))) == 0) {
      break;
```



```
    }
  }
}

/*********************************************************************
*
*       Public code (via callback)
*
**********************************************************************
*/

/*********************************************************************
*
*       _HW_Init
*
*  Function description
*    Initializes the QSPI hardware.
*
*  Parameters
*    Unit           Device index (0-based)
*
*  Return value
*    Frequency of the QSPI clock in kHz.
*
*  Additional information
*    _HW_Init() is mandatory and it has to be implemented by any
*    hardware layer. The QSPI NAND physical layer calls this function
*    first before any other function of the hardware layer. This function
*    has to configure anything necessary for the data transfer via QSPI
*    such as clocks, port pins, QSPI peripheral, DMA, etc.
*/
static int _HW_Init(U8 Unit) {
  U32 Div;
  U32 Freq_Hz;

  FS_USE_PARA(Unit);
  //
  // Enable the clocks of peripherals and reset them.
  //
  RCC_AHB4ENR  |= 0
               | (1uL << AHB1ENR_GPIOBEN)
               | (1uL << AHB1ENR_GPIOFEN)
               ;
  RCC_AHB3ENR  |=   1uL << AHB3ENR_QSPIEN;
  RCC_AHB3RSTR |=   1uL << AHB3ENR_QSPIEN;
  RCC_AHB3RSTR &= ~(1uL << AHB3ENR_QSPIEN);
  //
  // Wait for the unit to exit reset.
  //
  while (1) {
    if ((RCC_AHB3RSTR & (1uL << AHB3ENR_QSPIEN)) == 0) {
      break;
    }
  }
  //
  // NCS of bank 1 is an output signal and is controlled by the QUADSPI unit.
  //
  GPIOB_MODER   &= ~(MODER_MASK   << (NOR_BK1_NCS_BIT << 1));
  GPIOB_MODER   |=   MODER_ALT    << (NOR_BK1_NCS_BIT << 1);
  GPIOB_AFRL    &= ~(AFR_MASK     << (NOR_BK1_NCS_BIT << 2));
  GPIOB_AFRL    |=   0xAuL        << (NOR_BK1_NCS_BIT << 2);
  GPIOB_OSPEEDR &= ~(OSPEEDR_MASK << (NOR_BK1_NCS_BIT << 1));
  GPIOB_OSPEEDR |=   OSPEEDR_HIGH << (NOR_BK1_NCS_BIT << 1);
  //
  // CLK is an output signal controlled by the QUADSPI unit.
  //
  GPIOF_MODER   &= ~(MODER_MASK   << (NOR_CLK_BIT << 1));
  GPIOF_MODER   |=   MODER_ALT    << (NOR_CLK_BIT << 1);
  GPIOF_AFRH    &= ~(AFR_MASK     << ((NOR_CLK_BIT - 8) << 2));
  GPIOF_AFRH    |=   0x9uL        << ((NOR_CLK_BIT - 8) << 2);
  GPIOF_OSPEEDR &= ~(OSPEEDR_MASK << (NOR_CLK_BIT << 1));
  GPIOF_OSPEEDR |=   OSPEEDR_HIGH << (NOR_CLK_BIT << 1);
  //
  // D0 of bank 1 is an input/output signal controlled by the QUADSPI unit.
  //
  GPIOF_MODER   &= ~(MODER_MASK   << (NOR_BK1_D0_BIT << 1));
```



```
  GPIOF_MODER   |=   MODER_ALT    << (NOR_BK1_D0_BIT << 1);
  GPIOF_AFRH    &= ~(AFR_MASK     << ((NOR_BK1_D0_BIT - 8) << 2));
  GPIOF_AFRH    |=   0xAuL        << ((NOR_BK1_D0_BIT - 8) << 2);
  GPIOF_OSPEEDR &= ~(OSPEEDR_MASK << (NOR_BK1_D0_BIT << 1));
  GPIOF_OSPEEDR |=   OSPEEDR_HIGH << (NOR_BK1_D0_BIT << 1);
  //
  // D1 of bank 1 is an input/output signal controlled by the QUADSPI unit.
  //
  GPIOF_MODER   &= ~(MODER_MASK   << (NOR_BK1_D1_BIT << 1));
  GPIOF_MODER   |=   MODER_ALT    << (NOR_BK1_D1_BIT << 1);
  GPIOF_AFRH    &= ~(AFR_MASK     << ((NOR_BK1_D1_BIT - 8) << 2));
  GPIOF_AFRH    |=   0xAuL        << ((NOR_BK1_D1_BIT - 8) << 2);
  GPIOF_OSPEEDR &= ~(OSPEEDR_MASK << (NOR_BK1_D1_BIT << 1));
  GPIOF_OSPEEDR |=   OSPEEDR_HIGH << (NOR_BK1_D1_BIT << 1);
  //
  // D2 of bank 1 is an input/output signal and is controlled by the QUADSPI unit.
  //
  GPIOF_MODER   &= ~(MODER_MASK   << (NOR_BK1_D2_BIT << 1));
  GPIOF_MODER   |=   MODER_ALT    << (NOR_BK1_D2_BIT << 1);
  GPIOF_AFRL    &= ~(AFR_MASK     << (NOR_BK1_D2_BIT << 2));
  GPIOF_AFRL    |=   0x9uL        << (NOR_BK1_D2_BIT << 2);
  GPIOF_OSPEEDR &= ~(OSPEEDR_MASK << (NOR_BK1_D2_BIT << 1));
  GPIOF_OSPEEDR |=   OSPEEDR_HIGH << (NOR_BK1_D2_BIT << 1);
  //
  // D3 of bank 1 is an output signal and is controlled by the QUADSPI unit.
  //
  GPIOF_MODER   &= ~(MODER_MASK   << (NOR_BK1_D3_BIT << 1));
  GPIOF_MODER   |=   MODER_ALT    << (NOR_BK1_D3_BIT << 1);
  GPIOF_AFRL    &= ~(AFR_MASK     << (NOR_BK1_D3_BIT << 2));
  GPIOF_AFRL    |=   0x9uL        << (NOR_BK1_D3_BIT << 2);
  GPIOF_OSPEEDR &= ~(OSPEEDR_MASK << (NOR_BK1_D3_BIT << 1));
  GPIOF_OSPEEDR |=   OSPEEDR_HIGH << (NOR_BK1_D3_BIT << 1);
  //
  // Initialize the Quad-SPI controller.
  //
  Freq_Hz = NAND_CLK_HZ;
  Div = (U32)_CalcClockDivider(&Freq_Hz);
  QUADSPI_CR  = 0
              | (1uL << CR_EN_BIT)                  // Enable the Quad-SPI unit.
              | (Div << CR_PRESCALER_BIT)
              ;
  QUADSPI_DCR = 0
              | (1uL           << DCR_CKMODE_BIT)   // CLK signal stays HIGH
                                                    // when the NAND flash is not selected.
              | (DCR_FSIZE_MAX << DCR_FSIZE_BIT)    // We set the NAND flash size to maximum
                                                    // since the actual value is not known
                                                    // at this stage.
              ;
  return (int)Freq_Hz / 1000;
}

/*********************************************************************
*
*       _HW_ExecCmd
*
*  Function description
*    Executes a NAND flash command without data transfer.
*
*  Parameters
*    Unit           Device index (0-based)
*    Cmd            Code of the command to be sent.
*    BusWidth       Number of data lines to be used for sending
*                   the command code.
*
*  Return value
*    ==0    OK, command sent.
*    !=0    An error occurred.
*
*  Additional information
*    This function is mandatory and it has to be implemented by any
*    hardware layer.
*/
static int _HW_ExecCmd(U8 Unit, U8 Cmd, U8 BusWidth) {
  U32 CfgReg;
  U32 CmdMode;
```



```
  ASSERT_IS_LOCKED();
  FS_USE_PARA(Unit);    // This device has only one HW unit.
  //
  // Fill local variables.
  //
  CmdMode = _GetMode(BusWidth, sizeof(Cmd));
  CfgReg  = 0
          | (CCR_FMODE_WRITE << CCR_FMODE_BIT)
          | (CmdMode         << CCR_IMODE_BIT)
          | (Cmd             << CCR_INTRUCTION_BIT)
          ;
  //
  // Wait until the unit is ready for the new command.
  //
  while (1) {
    if ((QUADSPI_SR & (1uL << SR_BUSY_BIT)) == 0) {
      break;
    }
  }
  //
  // Execute the command.
  //
  QUADSPI_DLR = 0;
  QUADSPI_ABR = 0;
  QUADSPI_CCR = CfgReg;
  //
  // Wait until the command has been completed.
  //
  while (1) {
    if ((QUADSPI_SR & (1uL << SR_BUSY_BIT)) == 0) {
      break;
    }
  }
  return 0;
}

/*********************************************************************
*
*       _HW_ReadData
*
*  Function description
*    Transfers data from NAND flash device to MCU.
*
*  Parameters
*    Unit           Device index (0-based)
*    Cmd            Code of the command to be sent.
*    pPara          Additional command parameters (can be NULL).
*    NumBytesPara   Number of additional bytes to be sent after command.
*                   Can be 0 if pPara is NULL.
*    NumBytesAddr   Number of address bytes to be sent.
*                   Can be 0 if pPara is NULL.
*    pData          Data received from the NAND flash device.
*    NumBytesData   Number of bytes to be received from the NAND flash device.
*    BusWidth       Specifies the number of data lines to be used
*                   during the data transfer.
*
*  Return value
*    ==0    OK, data transferred.
*    !=0    An error occurred.
*
*  Additional information
*    This function is mandatory and it has to be implemented by any
*    hardware layer.
*/
static int _HW_ReadData(U8 Unit, U8 Cmd, const U8 * pPara, unsigned NumBytesPara,
                        unsigned NumBytesAddr, U8 * pData, unsigned NumBytesData,
                        U16 BusWidth) {
  U32 AddrReg;
  U32 AltReg;
  U32 CfgReg;
  U32 DataMode;
  U32 AddrMode;
  U32 AltMode;
  U32 CmdMode;
  U32 DataReg;
  U32 AltSize;
```



```
  U32 AddrSize;
  U32 NumBytesAvail;
  U32 NumBytes;
  U32 NumBytesAlt;

  ASSERT_IS_LOCKED();
  FS_USE_PARA(Unit);    // This device has only one HW unit.
  //
  // Fill local variables.
  //
  AddrReg     = 0;
  AltReg      = 0;
  NumBytesAlt = NumBytesPara - NumBytesAddr;
  CmdMode     = _GetMode(FS_BUSWIDTH_GET_CMD(BusWidth), sizeof(Cmd));
  AddrMode    = _GetMode(FS_BUSWIDTH_GET_ADDR(BusWidth), NumBytesAddr);
  AltMode     = _GetMode(FS_BUSWIDTH_GET_ADDR(BusWidth), NumBytesAlt);
  DataMode    = _GetMode(FS_BUSWIDTH_GET_DATA(BusWidth), NumBytesData);
  //
  // Encode the address.
  //
  if (NumBytesAddr) {
    NumBytes = NumBytesAddr;
    do {
      AddrReg <<= 8;
      AddrReg  |= (U32)(*pPara++);
    } while (--NumBytes);
  }
  //
  // Encode the dummy and mode bytes.
  //
  if (NumBytesPara > NumBytesAddr) {
    NumBytes = NumBytesAlt;
    do {
      AltReg <<= 8;
      AltReg  |= (U32)(*pPara++);
    } while (--NumBytes);
  }
  AddrSize = 0;
  if (NumBytesAddr) {
    AddrSize = (NumBytesAddr - 1) & CCR_ADSIZE_MASK;
  }
  AltSize = 0;
  if (NumBytesAlt) {
    AltSize = (NumBytesAlt - 1) & CCR_ABSIZE_MASK;
  }
  CfgReg = 0
         | (CCR_FMODE_READ << CCR_FMODE_BIT)
         | (DataMode       << CCR_DMODE_BIT)
         | (AltSize        << CCR_ABSIZE_BIT)
         | (AltMode        << CCR_ABMODE_BIT)
         | (AddrSize       << CCR_ADSIZE_BIT)
         | (AddrMode       << CCR_ADMODE_BIT)
         | (CmdMode        << CCR_IMODE_BIT)
         | (Cmd            << CCR_INTRUCTION_BIT)
         ;
  //
  // Wait until the unit is ready for the new command.
  //
  while (1) {
    if ((QUADSPI_SR & (1uL << SR_BUSY_BIT)) == 0) {
      break;
    }
  }
  //
  // Execute the command.
  //
  _ClearFlags();
  if (NumBytesData) {
    QUADSPI_DLR = NumBytesData - 1;       // 0 means "read 1 byte".
  }
  QUADSPI_ABR = AltReg;
  QUADSPI_CCR = CfgReg;
  if (NumBytesAddr) {
    QUADSPI_AR = AddrReg;
  }
  //
```



```
  // Read data from NOR flash.
  //
  if (NumBytesData) {
    do {
      //
      // Wait for the data to be received.
      //
      while (1) {
        NumBytesAvail = (QUADSPI_SR >> SR_FLEVEL_BIT) & SR_FLEVEL_MASK;
        if ((NumBytesAvail >= 4) || (NumBytesAvail >= NumBytesData)) {
          break;
        }
      }
      //
      // Read data and store it to destination buffer.
      //
      if (NumBytesData < 4) {
        //
        // Read single bytes.
        //
        do {
          *pData++ = QUADSPI_DR_BYTE;
        } while (--NumBytesData);
      } else {
        //
        // Read 4 bytes at a time.
        //
        DataReg = QUADSPI_DR;
        *pData++ = (U8)DataReg;
        *pData++ = (U8)(DataReg >> 8);
        *pData++ = (U8)(DataReg >> 16);
        *pData++ = (U8)(DataReg >> 24);
        NumBytesData -= 4;
      }
    } while (NumBytesData);
  }
  //
  // Wait until the data transfer has been completed.
  //
  while (1) {
    if (QUADSPI_SR & (1uL << SR_TCF_BIT)) {
      break;
    }
  }
  return 0;
}

/*********************************************************************
*
*       _HW_WriteData
*
*  Function description
*    Transfers data from MCU to NAND flash device.
*
*  Parameters
*    Unit           Device index (0-based)
*    Cmd            Code of the command to be sent.
*    pPara          Additional command parameters (can be NULL).
*    NumBytesPara   Number of additional bytes to be sent after command.
*                   Can be 0 if pPara is NULL.
*    NumBytesAddr   Number of address bytes to be sent.
*                   Can be 0 if pPara is NULL.
*    pData          Data to be sent to NAND flash device.
*    NumBytesData   Number of data bytes to be sent to NAND flash device.
*    BusWidth       Specifies the number of data lines to be used
*                   during the data transfer.
*
*  Return value
*    ==0    OK, data transferred.
*    !=0    An error occurred.
*
*  Additional information
*    This function is mandatory and it has to be implemented by any
*    hardware layer.
*/
static int _HW_WriteData(U8 Unit, U8 Cmd, const U8 * pPara, unsigned NumBytesPara,
```



```
                        unsigned NumBytesAddr, const U8 * pData, unsigned NumBytesData,
                        U16 BusWidth) {
  U32 AddrReg;
  U32 AltReg;
  U32 CfgReg;
  U32 DataMode;
  U32 AddrMode;
  U32 AltMode;
  U32 CmdMode;
  U32 DataReg;
  U32 AddrSize;
  U32 AltSize;
  U32 NumBytes;
  U32 NumBytesAlt;
  U32 NumBytesFree;

  ASSERT_IS_LOCKED();
  FS_USE_PARA(Unit);    // This device has only one HW unit.
  //
  // Fill local variables.
  //
  AddrReg     = 0;
  AltReg      = 0;
  NumBytesAlt = NumBytesPara - NumBytesAddr;
  CmdMode     = _GetMode(FS_BUSWIDTH_GET_CMD(BusWidth), sizeof(Cmd));
  AddrMode    = _GetMode(FS_BUSWIDTH_GET_ADDR(BusWidth), NumBytesAddr);
  AltMode     = _GetMode(FS_BUSWIDTH_GET_ADDR(BusWidth), NumBytesAlt);
  DataMode    = _GetMode(FS_BUSWIDTH_GET_DATA(BusWidth), NumBytesData);
  //
  // Encode the address.
  //
  if (NumBytesAddr) {
    NumBytes = NumBytesAddr;
    do {
      AddrReg <<= 8;
      AddrReg  |= (U32)(*pPara++);
    } while (--NumBytes);
  }
  //
  // Encode the dummy and mode bytes.
  //
  if (NumBytesPara > NumBytesAddr) {
    NumBytes = NumBytesAlt;
    do {
      AltReg <<= 8;
      AltReg  |= (U32)(*pPara++);
    } while (--NumBytes);
  }
  AddrSize = 0;
  if (NumBytesAddr) {
    AddrSize = NumBytesAddr - 1;
  }
  AltSize = 0;
  if (NumBytesAlt) {
    AltSize = NumBytesAlt - 1;
  }
  CfgReg = 0
         | (CCR_FMODE_WRITE << CCR_FMODE_BIT)
         | (DataMode        << CCR_DMODE_BIT)
         | (AltSize         << CCR_ABSIZE_BIT)
         | (AltMode         << CCR_ABMODE_BIT)
         | (AddrSize        << CCR_ADSIZE_BIT)
         | (AddrMode        << CCR_ADMODE_BIT)
         | (CmdMode         << CCR_IMODE_BIT)
         | (Cmd             << CCR_INTRUCTION_BIT)
         ;
  //
  // Wait until the unit is ready for the new command.
  //
  while (1) {
    if ((QUADSPI_SR & (1uL << SR_BUSY_BIT)) == 0) {
      break;
    }
  }
  //
  // Execute the command.
```



```
  //
  _ClearFlags();
  if (NumBytesData) {
    QUADSPI_DLR = NumBytesData - 1;       // 0 means "read 1 byte".
  }
  QUADSPI_ABR = AltReg;
  QUADSPI_CCR = CfgReg;
  if (NumBytesAddr) {
    QUADSPI_AR = AddrReg;
  }
  //
  // write data to NOR flash.
  //
  if (NumBytesData) {
    do {
      //
      // Wait for free space in FIFO.
      //
      while (1) {
        NumBytesFree = (QUADSPI_SR >> SR_FLEVEL_BIT) & SR_FLEVEL_MASK;
        NumBytesFree = NUM_BYTES_FIFO - NumBytesFree;
        if ((NumBytesFree >= 4) || (NumBytesFree >= NumBytesData)) {
          break;
        }
      }
      //
      // Get the data from source buffer and write it.
      //
      if (NumBytesData < 4) {
        //
        // Write single bytes.
        //
        do {
          QUADSPI_DR_BYTE = *pData++;
        } while (--NumBytesData);
      } else {
        //
        // Write 4 bytes at a time if possible.
        //
        DataReg  = (U32)*pData++;
        DataReg |= (U32)*pData++ << 8;
        DataReg |= (U32)*pData++ << 16;
        DataReg |= (U32)*pData++ << 24;
        NumBytesData -= 4;
        QUADSPI_DR = DataReg;
      }
    } while (NumBytesData);
  }
  //
  // Wait until the data transfer has been completed.
  //
  while (1) {
    if (QUADSPI_SR & (1uL << SR_TCF_BIT)) {
      break;
    }
  }
  return 0;
}

/*********************************************************************
*
*       _HW_Delay
*
*  Function description
*    Blocks the execution for the specified number of milliseconds.
*
*  Parameters
*    Unit           Device index (0-based)
*    ms             Number of milliseconds to wait.
*
*  Additional information
*    This function is mandatory and it has to be implemented by any
*    hardware layer.
*/
static void _HW_Delay(U8 Unit, int ms) {
  FS_USE_PARA(Unit);
```



```
  if (ms) {
    volatile U32 NumCycles;

    do {
      NumCycles = NUM_CYCLES_PER_MS;
      do {
        ;
      } while (--NumCycles);
    } while (--ms);
  }
}

/*********************************************************************
*
*       _HW_Poll
*
*  Function description
*    Sends a command repeatedly and checks the response data
*    for a specified condition.
*
*  Parameters
*    Unit           Device index (0-based)
*    Cmd            Code of the command to be sent.
*    pPara          Additional command parameters (can be NULL).
*    NumBytesPara   Number of additional bytes to be sent after command.
*                   Can be 0 if pPara is NULL.
*    BitPos         Position of the bit inside response data that has
*                   to be checked (0-based, with 0 being LSB)
*    BitValue       Bit value to wait for.
*    Delay_ms       Time between two command executions.
*    TimeOut_ms     Maximum time to wait for the bit at BitPos to be set to BitValue.
*    BusWidth       Specifies how many data lines have to be used for sending
*                   the command and parameters and for reading the data.
*
*  Return value
*    > 0    Error, timeout occurred.
*    ==0    OK, bit set to specified value.
*    < 0    Error, feature not supported.
*
*  Additional information
*    This function is optional and it can be left unimplemented
*    or set to NULL in the structure of the hardware layer below.
*/
static int _HW_Poll(U8 Unit, U8 Cmd, const U8 * pPara, unsigned NumBytesPara,
                    U8 BitPos, U8 BitValue, U32 Delay_ms, U32 TimeOut_ms, U16 BusWidth) {
  FS_USE_PARA(Unit);
  FS_USE_PARA(Cmd);
  FS_USE_PARA(pPara);
  FS_USE_PARA(NumBytesPara);
  FS_USE_PARA(BitPos);
  FS_USE_PARA(BitValue);
  FS_USE_PARA(Delay_ms);
  FS_USE_PARA(TimeOut_ms);
  FS_USE_PARA(BusWidth);
  return -1;
}

/*********************************************************************
*
*       _HW_Lock
*
*  Function description
*    Requests exclusive access to SPI bus.
*
*  Parameters
*    Unit     Device index (0-based)
*
*  Additional information
*    This function is optional and it can be left unimplemented
*    or set to NULL in the structure of the hardware layer below.
*/
static void _HW_Lock(U8 Unit) {
  FS_USE_PARA(Unit);
  IF_TEST(_LockCnt++);
  ASSERT_IS_LOCKED();
}
```



```
/*********************************************************************
*
*       _HW_Unlock
*
*  Function description
*    Releases the exclusive access of SPI bus.
*
*  Parameters
*    Unit     Device index (0-based)
*
*  Additional information
*    This function is optional and it can be left unimplemented
*    or set to NULL in the structure of the hardware layer below.
*/
static void _HW_Unlock(U8 Unit) {
  FS_USE_PARA(Unit);
  ASSERT_IS_LOCKED();
  IF_TEST(_LockCnt--);
}

/*********************************************************************
*
*       Public const
*
**********************************************************************
*/

/*********************************************************************
*
*       FS_NAND_HW_QSPI_STM32H743_SEGGER_QSPI_Flash_Evaluator
*/
const FS_NAND_HW_TYPE_QSPI FS_NAND_HW_QSPI_STM32H743_SEGGER_QSPI_Flash_Evaluator = {
  _HW_Init,
  _HW_ExecCmd,
  _HW_ReadData,
  _HW_WriteData,
  _HW_Poll,
  _HW_Delay,
  _HW_Lock,
  _HW_Unlock
};

/************************** End of file ****************************/
```


6.3.5.6 Hardware layer API - FS_NAND_HW_TYPE_SPI

This hardware layer supports any serial NAND flash device with a standard SPI. It is used by the FS_NAND_PHY_SPI physical layer to exchange data with a NAND flash device that supports such an interface. Typically, the implementation of FS_NAND_HW_TYPE_SPI hardware layer makes use of a SPI controller. It is also possible to implement the data exchange via GPIO pins.

The functions of this hardware layer are grouped in the structure of type FS_NAND_HW_TYPE_SPI. The following sections describe these functions in detail.

6.3.5.6.1 FS_NAND_HW_TYPE_SPI

Description

NAND hardware layer API for NAND flash devices connected via SPI.

Type definition

```
typedef struct {
  FS_NAND_HW_TYPE_SPI_INIT       * pfInit;
  FS_NAND_HW_TYPE_SPI_DISABLE_CS * pfDisableCS;
  FS_NAND_HW_TYPE_SPI_ENABLE_CS  * pfEnableCS;
  FS_NAND_HW_TYPE_SPI_DELAY      * pfDelay;
  FS_NAND_HW_TYPE_SPI_READ       * pfRead;
  FS_NAND_HW_TYPE_SPI_WRITE      * pfWrite;
  FS_NAND_HW_TYPE_SPI_LOCK       * pfLock;
  FS_NAND_HW_TYPE_SPI_UNLOCK     * pfUnlock;
} FS_NAND_HW_TYPE_SPI;
```

Structure members

Member	Description
pfInit	Initializes the hardware.
pfDisableCS	Disables the NAND flash device.
pfEnableCS	Enables the NAND flash device.
pfDelay	Block the execution for a specified number of milliseconds.
pfRead	Transfers a specified number of bytes from NAND flash device to MCU.
pfWrite	Transfers a specified number of bytes from MCU to NAND flash device.
pfLock	Request exclusive access to SPI bus.
pfUnlock	Releases the exclusive access to SPI bus.

6.3.5.6.2 FS_NAND_HW_TYPE_SPI_INIT

Description

Initializes the hardware.

Type definition

```
typedef int FS_NAND_HW_TYPE_SPI_INIT(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)

Return value

≠ 0 OK, frequency of the SPI clock in kHz.
= 0 An error occurred.

Additional information

This function is a member of the FS_NAND_HW_TYPE_SPI NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer of this type. FS_NAND_HW_TYPE_SPI_INIT is the first function of the hardware layer API that is called by a NAND physical layer during the mounting of the file system.

This function has to perform any initialization of the MCU hardware required to access the NAND flash device such as clocks, port pins, SPI controllers, etc.

A serial NAND flash requires that the SPI communication protocol meets the following requirements:

- 8-bit data length.
- The most significant bit hast to be sent out first.
- Chip Select (CS) signal should be initially high to disable the serial NAND flash device.
- The SPI clock frequency does not have to exceed the maximum clock frequency that is specified by the serial NAND flash device.

Typically, two SPI modes are supported:

- Mode 0 - CPOL = 0, CPHA = 0, SPI clock remains low in idle state.
- Mode 3 - CPOL = 1, CPHA = 1, SPI clock remains high in idle state.

6.3.5.6.3 FS_NAND_HW_TYPE_SPI_DISABLE_CS

Description

Disables the NAND flash device.

Type definition

```
typedef void FS_NAND_HW_TYPE_SPI_DISABLE_CS(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)

Additional information

This function is a member of the FS_NAND_HW_TYPE_SPI NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer of this type.

The NAND flash device is disabled by driving the Chip Select (CS) signal to logic-high. The NAND flash device ignores any command or data sent with the CS signal set to logic-high.

6.3.5.6.4 FS_NAND_HW_TYPE_SPI_ENABLE_CS

Description

Enables the NAND flash device.

Type definition

```
typedef void FS_NAND_HW_TYPE_SPI_ENABLE_CS(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)

Additional information

This function is a member of the FS_NAND_HW_TYPE_SPI NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer of this type.

The NAND flash device is enabled by driving the Chip Select (CS) signal to logic-low. Typically, the NAND flash device is enabled at the beginning of command and data sequence and disabled at the end of it.

6.3.5.6.5 FS_NAND_HW_TYPE_SPI_DELAY

Description

Blocks the execution for the specified number of milliseconds.

Type definition

```
typedef void FS_NAND_HW_TYPE_SPI_DELAY(U8  Unit,
                                       int ms);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)
ms	Number of milliseconds to wait.

Additional information

This function is a member of the FS_NAND_HW_TYPE_SPI NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer of this type.

The time FS_NAND_HW_TYPE_SPI_DELAY blocks does not have to be accurate. That is the function can block the execution longer that the number of specified milliseconds but no less than that.

6.3.5.6.6 FS_NAND_HW_TYPE_SPI_READ

Description

Transfers a specified number of bytes from NAND flash device to MCU.

Type definition

```
typedef int FS_NAND_HW_TYPE_SPI_READ(U8       Unit,
                                     void   * pData,
                                     unsigned NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)
pData	Data read from NAND flash device.
NumBytes	Number of bytes to be read.

Additional information

This function is a member of the FS_NAND_HW_TYPE_SPI NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer of this type.

The function has to sample the data on the falling edge of the SPI clock.

6.3.5.6.7 FS_NAND_HW_TYPE_SPI_WRITE

Description

Transfers a specified number of bytes from MCU to NAND flash device.

Type definition

```
typedef int FS_NAND_HW_TYPE_SPI_WRITE(      U8       Unit,
                                      const void   * pData,
                                            unsigned NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)
pData	in Data to be written to NAND flash device.
NumBytes	Number of bytes to be written.

Additional information

This function is a member of the FS_NAND_HW_TYPE_SPI NAND hardware layer API and it is mandatory to be implemented by any NAND hardware layer of this type.

The NAND flash device samples the data on the rising edge of the SPI clock.

6.3.5.6.8 FS_NAND_HW_TYPE_SPI_LOCK

Description

Requests exclusive access to SPI bus.

Type definition

```
typedef void FS_NAND_HW_TYPE_SPI_LOCK(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)

Additional information

This function is a member of the FS_NAND_HW_TYPE_SPI NAND hardware layer API. The implementation of this function is optional.

The FS_NAND_PHY_SPI physical layer calls this function to indicate that it needs exclusive to access the serial NAND flash device via the SPI bus. It is guaranteed that the FS_NAND_PHY_SPI physical layer does not attempt to exchange any data with the serial NAND flash device via the SPI bus before calling this function first. It is also guaranteed that FS_NAND_HW_TYPE_SPI_LOCK and FS_NAND_HW_TYPE_SPI_UNLOCK are called in pairs. Typically, this function is used for synchronizing the access to SPI bus when the SPI bus is shared between the serial NAND flash and other SPI devices.

A possible implementation would make use of an OS semaphore that is acquired in FS_NAND_HW_TYPE_SPI_LOCK and released in FS_NAND_HW_TYPE_SPI_UNLOCK.

6.3.5.6.9 FS_NAND_HW_TYPE_SPI_UNLOCK

Description

Releases the exclusive access of SPI bus.

Type definition

```
typedef void FS_NAND_HW_TYPE_SPI_UNLOCK(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the NAND hardware layer instance (0-based)

Additional information

This function is a member of the FS_NAND_HW_TYPE_SPI NAND hardware layer API. The implementation of this function is optional.

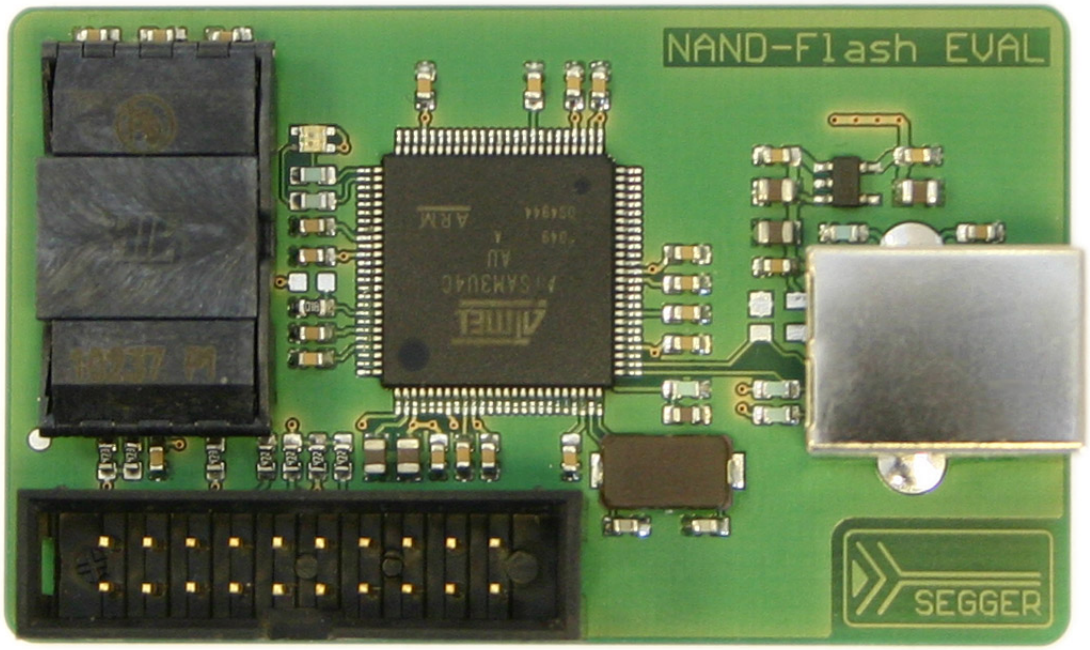
The FS_NAND_PHY_SPI physical layer calls this function after it no longer needs to access the serial NAND flash device via the SPI bus. It is guaranteed that the FS_NAND_PHY_SPI physical layer does not attempt to exchange any data with the serial NAND flash device via the SPI bus before calling FS_NAND_HW_TYPE_SPI_LOCK. It is also guaranteed that FS_NAND_HW_TYPE_SPI_UNLOCK and FS_NAND_HW_TYPE_SPI_LOCK are called in pairs.

FS_NAND_HW_TYPE_SPI_UNLOCK and FS_NAND_HW_TYPE_SPI_LOCK can be used to synchronize the access to the SPI bus when other devices than the serial NAND flash are connected to it. A possible implementation would make use of an OS semaphore that is acquired FS_NAND_HW_TYPE_SPI_LOCK and released in FS_NAND_HW_TYPE_SPI_UNLOCK.

6.3.6 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 drive). 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 thus be performed without the need of an expensive development environment. All current operating systems will recognize the board out of the box.

The NAND-Flash EVAL board comes with a ready to use USB-MSD application in binary form. emFile is provided in object code{c} 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

6.3.7 FAQs

Q: Are Multi-Level Cell NAND flashes (MLCs) supported?
A: Yes, the Universal NAND driver supports MLCs.
Q: Are NAND flash devices with 4 Kbyte pages supported?
A: Yes, they are supported via the `FS_NAND_PHY_4096x8` and `FS_NAND_PHY_ONFI` physical layers.

6.4 NOR flash driver

6.4.1 General information

emFile supports the use of raw NOR flash devices as data storage. Two drivers for NOR flash devices also known as flash translation layers are available:

- Sector Map NOR driver - Supports almost any parallel or serial NOR flash device. This driver was optimized for increase write performance.
- Block Map NOR driver - Supports almost any parallel or serial NOR flash device. This driver was optimized for reduced RAM usage. The difference between the two NOR drivers consists in the way they are managing the mapping of file system logical sectors to the NOR flash device. The Sector map driver was designed with the goal to access the data fast at a time when NOR flash devices had a relatively small capacity (a few Mbytes). To achieve this, the driver maintains a mapping table at logical sector granularity. This approach has proven to be very efficient, but for modern NOR flash devices with capacities larger than 8 Mbytes the RAM usage of the Sector Map NOR driver becomes increasingly high. This is the reason why the Block Map NOR driver was developed. The design goal of the Block Map NOR driver was to use as few RAM as possible. This is realized, by mapping blocks of file system logical sectors to NOR flash device.

6.4.1.1 Software structure

The NOR drivers are organized into different layers and contain from top to bottom:

- NOR driver layer
- *NOR physical layer*
- *NOR hardware layer*

A hardware layer is required only for serial NOR flash devices. Typically, CFI compliant NOR flash device is mapped into the system memory of the MCU and can be accessed directly without a hardware layer. Normally no changes to the physical layer are required. If the physical layer needs to be adapted, a template is available.

6.4.1.2 Garbage collection

The driver performs the garbage collection automatically during the write operations. If no empty logical sectors are available to store the data, new empty logical sectors are created by erasing the data of the logical sectors marked as invalid. This involves a NOR physical sector erase operation which, depending on the characteristics of the NOR flash, can potentially take a long time to complete, and therefor reduces the write performance. For applications which require maximum write throughput, the garbage collection can be done in the application. Typically, this 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 to 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 entire storage device.

6.4.1.3 Fail-safe operation

The operation of emFile NOR driver is fail-safe. This means that the driver performs only atomic operation in order to make sure that the stored data is always valid after a unexpected power loss that interrupts a write operation. If the power loss interrupts the write operation, then the old data is preserved and is not corrupted.

The fail-safe operation is only guaranteed if the NOR flash device is able to fully complete the last write operation it receives from the CPU before the unexpected reset. In other words, the supply voltage of the NOR flash device has to remain valid for a few hundreds

of microseconds after the CPU enters reset. The exact timing can be taken from the data sheet of the NOR flash device.

The number of bytes written by the NOR driver in one operation varies between the value configured via FS_NOR_LINE_SIZE and the page size of the NOR flash device. Not all the data the NOR driver writes to the storage is critical. Non-critical write operations can be interrupted by an unexpected reset without negatively affecting the fail-safe operation. The critical data is always stored as a write operation containing FS_NOR_LINE_SIZE bytes.

The erase operation of a NOR flash sector can be interrupted by an unexpected reset because the NOR driver is able determine at low-level mount which NOR flash sector was not completely erased. The affected NOR flash sector will be erased again.

> **Note**
>
> The requirement about the write operation does not apply for configurations that use the Block Map NOR driver with the CRC verification enabled because in this configuration the file system is able to determine and discard incomplete write operations at low-level mount.

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.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 Low-level format

Before using a NOR flash device for the first time, the application has to initialize it by performing a low-level format operation on it. Refer to FS_FormatLow() and FS_FormatLLIfRequired() for detailed information about this.

6.4.1.6 Supported hardware

The NOR flash drivers can be used with almost any NOR flash device. This includes NOR flash device with 8-bit and 16-bit parallel interfaces, as well as two 16-bit interfaces in parallel. Serial NOR flash devices are also supported. A NOR flash device has to meet the following requirements:

- The NOR flash device has to contain a minimum of two physical sectors. The physical sectors do not need to be of equal size but at least two physical sectors of each size must exist. For example a NOR flash device 11 physical sectors organized as 8 * 8 Kbytes + 3 * 64 Kbytes is supported.
- All the physical sectors need to be at least 2048 bytes large.
- The erase operation must set all bits in a physical sector to one.

In order to provide the best write performance, the NOR drivers make use of a NOR flash device feature that permits the same location to be modified multiple times without an erase operation in between, as long as only bits set to 1 are changed to 0. For NOR flash devices that do not support this feature, the NOR drivers have to be compiled with the configuration define FS_NOR_CAN_REWRITE set to 0.

In general, the NOR drivers support almost all serial and parallel NOR flashes which meet the listed requirements. In addition, the NOR drivers can use the internal flash memory of a MCU as storage device. The table below shows the serial NOR flash devices that have been tested or are compatible with a tested device:

Device	Capacity (bits)
Cypress (Spansion)	
S25FL032P	32 Mbits (4 Mbytes x 1)
S29GL064N	64 Mbits (8 Mbytes x 16)

Device	Capacity (bits)
S25FL128S	128 Mbits (16 Mbytes x 1)
S25FL127S	128 Mbits (16 Mbytes x 1)
S25FL256L	256 Mbits (32 Mbytes x 1)
GigaDevice	
GD25Q16B	16 Mbits (2 Mbytes x 1)
GD25Q32C	32 Mbits (4 Mbytes x 1)
GD25Q64C	64 Mbits (8 Mbytes x 1)
Intel	
28FxxxP30	64 Mbits - 1 Gbits
28FxxxP33	64 Mbits - 512 Mbits
ISSI	
IS25WP064A	64 Mbits (8 Mbytes x 1)
IS25LP064A	64 Mbits (8 Mbytes x 1)
IS25LP128	128 Mbits (16 Mbytes x 1)
Macronix	
MX25V1635F	16 Mbits (2 Mbytes x 1)
MX25R3235F	32 Mbits (4 Mbytes x 1)
MX25L3233	32 Mbits (4 Mbytes x 1)
MX25L6433F	64 Mbits (8 Mbytes x 1)
MX25R64	64 Mbits (8 Mbytes x 1)
MX25L128	128 Mbits (16 Mbytes x 1)
MX25L256	256 Mbits (32 Mbytes x 1)
MX25L512	512 Mbits (64 Mbytes x 1)
MX66L51235F	512 Mbits (64 Mbytes x 1)
Microchip	
SST26VF064B	64 Mbits (8 Mbytes x 1)
Micron	
M25PX16	16 Mbits (2 Mbytes x 1)
MT28F320	32 Mbits
MT28F640	64 Mbits
N25Q064	64 Mbits (8 Mbytes x 1)
M25P64	64 Mbits (8 Mbytes x 1)
MT28F256	256 Mbits
MT28F128	128 Mbits
N25Q128A	128 Mbits (16 Mbytes x 1)
N25Q256	256 Mbits (32 Mbytes x 1)
MT25QL02GC	2 Gbits
ST-Microelectronics	
M25P40	4 Mbits (512 Kbytes x 1)
M25P80	8 Mbits (1 Mbyte x 1)
M29F080	8 Mbits (1 Mbyte x 8)
M28W160	16 Mbits (1 Mbytes x 16)
M29W160	16 Mbits (2 Mbytes x 8 or 1 Mbytes x 16)

Device	Capacity (bits)
M25P16	16 Mbits (2 Mbytes x 1)
M28W320	32 Mbits (2 Mbytes x 16)
M29W320	32 Mbits (4 Mbytes x 8 or 2 Mbytes x 16)
M25P32	32 Mbits (4 Mbytes x 1)
M28W640	64 Mbits (4 Mbytes x 16)
M29W640	64 Mbits (8 Mbytes x 8 or 4 Mbytes x 16)
M58LW064	64 Mbits (8 Mbytes x 8, 4 Mbytes x 16)
M25P128	128 Mbits (16 Mbytes x 1)
Winbond	
W25Q64	64 Mbits (8 Mbytes x 1)

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 easily be done. Get in touch with us if you have questions about support for devices not in this list.

6.4.1.6.1 Using the same NOR flash device for code and data

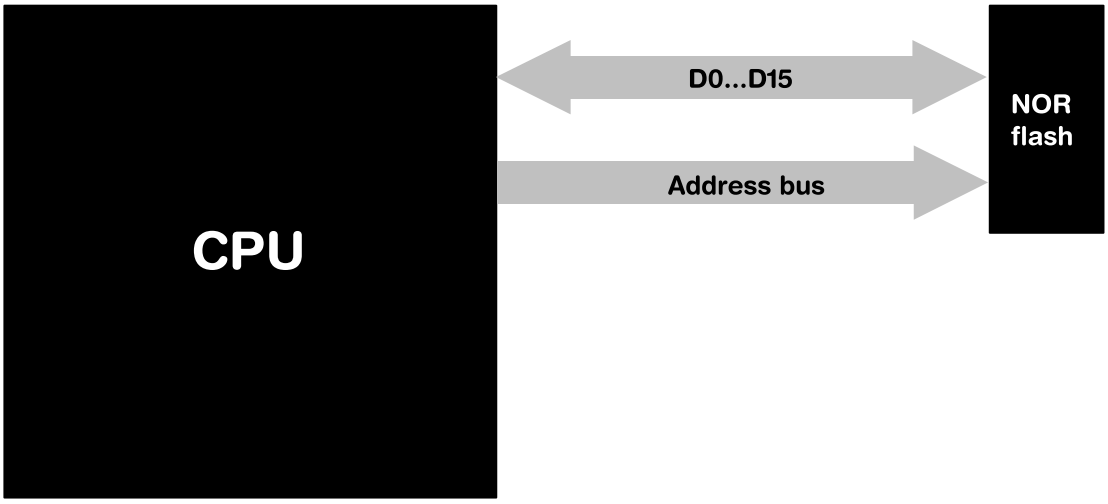
Most NOR flash device cannot be read out during a program or erase operation. This means that the CPU cannot execute code from the NOR flash device while the NOR flash device is busy performing a program or erase operation. A program crash is almost certain if the CPU tries to execute code in this case.

There are multiple options to solve this problem:

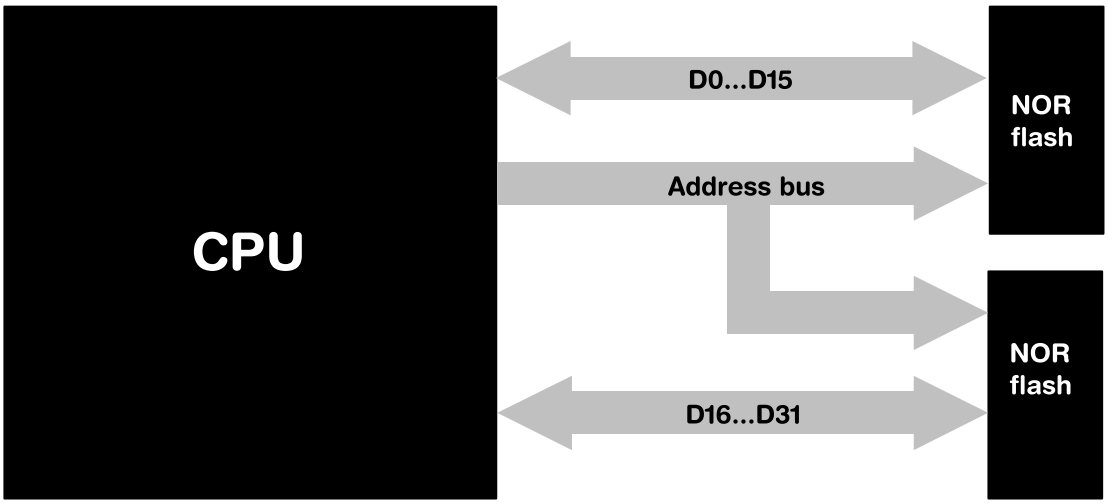
1. Multiple NOR flash devices can be used with code and data being stored on different devices.
2. A NOR flash device that supports multiple flash banks can be used. Typically, this type of NOR flash device allows the code execution from one back while performing an erase or program operation on a different one.
3. The hardware routines that program, erase or identify the NOR flash device are located in RAM and the interrupts are disabled.

6.4.1.7 Interfacing with a NOR flash device

The most common setup is a CFI compliant NOR flash device with a 16-bit interface. The picture below shows how this can be realized.



In addition to this, emFile supports the use of two CFI compliant NOR flash devices with a 16-bit interface which are connected to the same address bus as illustrated in the picture below.



6.4.1.8 Common flash interface (CFI)

The NOR flash drivers can be used with any CFI-compliant 16-bit device. 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 device, you are able to use one driver for different flash products by reading identifying information out of the flash device 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 device. For more technical details about CFI (Common Flash Interface) and SFDP (Serial Flash Discoverable Parameters), check the documents and specifications that are available free of charge at https://www.jedec.org

6.4.2 Sector Map NOR driver

This section describes the NOR driver that was optimized for increased write performance. The Sector Map NOR driver works by mapping single logical sectors to locations on the NOR flash device.

6.4.2.1 Theory of operation

Differentiating between logical sectors or blocks and physical sectors is essential to understand this section. A logical sector/block s the smallest readable and writable unit of any file system and its usual size is 512 bytes. A physical sector is an array of bytes on the NOR flash device that are erased together (typically between 2 Kbytes - 128 Kbytes). The Sector Map NOR 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, a direct relation between the sector number and its physical location cannot exist. The flash driver manages the logical sector numbers by writing it into special headers. To the upper layer, it does not matter were the logical sector is stored or how much flash memory is used as a buffer. All logical sectors (starting with sector 0) always exist and are always available for user access.

6.4.2.2 Configuring the driver

This section describes how to configure the file system to make use of the Sector Map NOR driver.

6.4.2.2.1 Runtime configuration

The driver must be added to the file system by calling `FS_AddDevice()` with the driver identifier set to the address of `FS_NOR_Driver`. This function call together with other function calls that configure the driver operation have to be added to `FS_X_AddDevices()` as demonstrated in the following example. This example shows how to configure the file system to access a NOR flash device connected via SPI.

```
#include "FS.h"

#define ALLOC_SIZE        0x4000        // Size defined in bytes
#define BYTES_PER_SECTOR  2048

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().
*/
void FS_X_AddDevices(void) {
  //
  // Give the file system memory to work with.
  //
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add and configure the NOR flash driver. The first 65536 bytes of the
  // serial NOR flash device are not used by the file system as data storage
  // and can be used by the application for other purposes. The file system
  // uses a total of 1 Mbyte from the serial NOR flash device as storage.
  //
  FS_AddDevice(&FS_NOR_Driver);
  FS_NOR_SetPhyType(0, &FS_NOR_PHY_ST_M25);
  FS_NOR_Configure(0, 0x00000000, 0x00010000, 0x00100000);
  FS_NOR_SetSectorSize(0, BYTES_PER_SECTOR);
  //
  // Set a larger logical sector size as the default
  // in order to reduce the RAM usage of the NOR driver.
  //
```



```
  FS_SetMaxSectorSize(BYTES_PER_SECTOR);
}
```

The API functions listed in the next table can be used by the application to configure the behavior of the Sector Map NOR driver. The application can call them only at the file system initialization in FS_X_AddDevices().

Function	Description
FS_NOR_Configure()	Configures an instance of the sector map NOR driver.
FS_NOR_ConfigureReserve()	Configures the number of logical sectors to be reserved.
FS_NOR_SetBlankSectorSkip()	Configures if the physical sectors which are already blank should be erased during the low-level format operation.
FS_NOR_SetDeviceLineSize()	Configures the minimum number of bytes that can be written to NOR flash.
FS_NOR_SetDeviceRewriteSupport()	Specifies if the NOR flash device can rewrite the same data if 0s are preserved.
FS_NOR_SetDirtyCheckOptimization()	Enables or disables the blank checking of a logical sector before write.
FS_NOR_SetEraseVerification()	Enables or disables the checking of the sector erase operation.
FS_NOR_SetPhyType()	Configures the type of NOR physical layer.
FS_NOR_SetSectorSize()	Configures the number of bytes in a logical sector.
FS_NOR_SetWriteVerification()	Enables or disables the checking of the page write operation.

6.4.2.2.1.1 FS_NOR_Configure()

Description

Configures an instance of the sector map NOR driver

Prototype

```
void FS_NOR_Configure(U8  Unit,
                      U32 BaseAddr,
                      U32 StartAddr,
                      U32 NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the sector map NOR driver (0-based).
BaseAddr	Address of the first byte in NOR flash.
StartAddr	Address of the first byte the NOR driver is permitted to use as storage.
NumBytes	Number of bytes starting from StartAddr available to be used by the NOR driver as storage.

Additional information

This function is mandatory and it has to be called once in FS_X_AddDevices() for each instance of the sector map NOR driver created by the application. Different instances of the NOR driver are identified by the Unit parameter.

BaseAddr is used only for NOR flash devices that are memory mapped. For serial NOR flash devices that are not memory mapped BaseAddr has to be set to 0.

StartAddr has to be greater than or equal to BaseAddr and smaller than the total number of bytes in the NOR flash device. The sector map NOR driver rounds up StartAddr to the start address of the next physical sector in the NOR flash device.

NumBytes is rounded up to a physical sector boundary if the memory range defined by StartAddr and NumBytes is smaller than the capacity of the NOR flash device. If the memory range defined by StartAddr and NumBytes is larger than the capacity of the NOR flash device than NumBytes is rounded down so that the memory range fits into the NOR flash device.

The sector map NOR driver can work with physical sectors of different size. It is required that at least two physical sectors of each sector size are available.

Example

The following sample demonstrates how to configure the file system to work with a single CFI compliant NOR flash device connected via a 16-bit data bus.

```
#include "FS.h"

#define ALLOC_SIZE 0x2000               // Size defined 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().
*/
void FS_X_AddDevices(void) {
  //
  // Give the file system memory to work with.
```



```
  //
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add the NOR driver.
  //
  FS_AddDevice(&FS_NOR_Driver);
  //
  // Set the physical layer type for single CFI compliant
  // NOR flash device with 16-bit data bus interface.
  //
  FS_NOR_SetPhyType(0, &FS_NOR_PHY_CFI_1x16);
  //
  // Configure the range of NOR flash device to be used as storage (2 Mbytes)
  //
  FS_NOR_Configure(0, 0x1000000, 0x1000000, 0x00200000);
}
```

The next sample demonstrates how to configure the file system to work with two CFI compliant NOR flash devices with each device being connected via a separate 16-bit data bus.

```
#include "FS.h"

#define ALLOC_SIZE 0x4000               // Size defined 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().
*/
void FS_X_AddDevices(void) {
  //
  // Give the file system memory to work with.
  //
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add the NOR driver.
  //
  FS_AddDevice(&FS_NOR_Driver);
  //
  // Set the physical layer type for two CFI compliant
  // NOR flash devices with 16-bit data bus interface.
  //
  FS_NOR_SetPhyType(0, &FS_NOR_PHY_CFI_2x16);
  //
  // Configure the range of NOR flash device to be used as storage (2 Mbytes each)
  //
  FS_NOR_Configure(0, 0x1000000, 0x1000000, 0x00400000);
}
```

The file system is able to handle two or more NOR flash devices that are not connected in parallel. The next sample shows how this can be realized.

```
#include "FS.h"

#define ALLOC_SIZE 0x4000               // Size defined 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().
*/
void FS_X_AddDevices(void) {
```



```
  //
  // Give the file system memory to work with.
  //
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add and configure the first NOR driver. The storage can be accessed
  // using the volume name "nor:0:". The physical layer is set to single
  // CFI compliant NOR flash device with 16-bit interface. The NOR flash
  // device is mapped at address 0x10000000 in the system memory and
  // 2 Mbytes of it are used by the file system as storage.
  //
  FS_AddDevice(&FS_NOR_Driver);
  FS_NOR_SetPhyType(0, &FS_NOR_PHY_CFI_1x16);
  FS_NOR_Configure(0, 0x1000000, 0x1000000, 0x00200000);
  //
  // Add and configure the second NOR driver. The storage can be accessed
  // using the volume name "nor:1:". The physical layer is set to single
  // CFI compliant NOR flash device with 16-bit interface. The NOR flash
  // device is mapped at address 0x40000000 in the system memory and
  // 4 Mbytes of it are used by the file system as storage.
  //
  FS_AddDevice(&FS_NOR_Driver);
  FS_NOR_SetPhyType(1, &FS_NOR_PHY_CFI_1x16);
  FS_NOR_Configure(1, 0x4000000, 0x4000000, 0x00400000);
}
```


6.4.2.2.1.2 FS_NOR_ConfigureReserve()

Description

Configures the number of logical sectors to be reserved.

Prototype

```
void FS_NOR_ConfigureReserve(U8 Unit,
                             U8 pctToReserve);
```

Parameters

Parameter	Description
Unit	Index of the sector map NOR driver (0-based).
pctToReserve	Percent of the total number of logical sectors to reserve.

Additional information

This function is optional. By default, the sector map NOR driver reserves about 10% of the total number of logical sector for future improvements and extensions. FS_NOR_ConfigureReserve() can be used in an application to modify this value. If set to 0 the sector map NOR driver uses all the available logical sectors to store file system data.

The application has to reformat the NOR flash device in order for the modified value to take effect.

6.4.2.2.1.3 FS_NOR_SetBlankSectorSkip()

Description

Configures if the physical sectors which are already blank should be erased during the low-level format operation.

Prototype

```
void FS_NOR_SetBlankSectorSkip(U8 Unit,
                               U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the sector map NOR driver (0-based).
OnOff	Specifies if the feature has to be enabled or disabled • =0 All the physical sectors are erased. • ≠0 Physical sectors are not erased if they are blank (default).

Additional information

This function is optional. The blank checking feature is disabled by default and has to be explicitly enabled at compile time by setting FS_NOR_SKIP_BLANK_SECTORS to 1. The feature can then be enabled or disabled at runtime using FS_NOR_BM_SetBlankSectorSkip().

Activating this feature can improve the speed of the low-level format operation when most of the physical sectors of the NOR flash device are already blank which is typically the case with NOR flash devices that ship from factory.

6.4.2.2.1.4 FS_NOR_SetDeviceLineSize()

Description

Configures the minimum number of bytes that can be written to NOR flash.

Prototype

```
void FS_NOR_SetDeviceLineSize(U8 Unit,
                              U8 ldBytesPerLine);
```

Parameters

Parameter	Description
Unit	Index of the sector map NOR driver (0-based).
ldBytesPerLine	Line size in bytes as power of 2 value.

Additional information

This function is optional. Typically, the NOR flash have lines smaller than 4 bytes which is the fixed default value configured at compile time. The FS_NOR_SUPPORT_VARIABLE_LINE_SIZE configuration define has to be set to a value different than 0 in order to enable this function.

6.4.2.2.1.5 FS_NOR_SetDeviceRewriteSupport()

Description

Specifies if the NOR flash device can rewrite the same data if 0s are preserved.

Prototype

```
void FS_NOR_SetDeviceRewriteSupport(U8 Unit,
                                    U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the sector map NOR driver (0-based).
OnOff	Specifies if the feature has to be enabled or disabled • =0 Rewrite operations are not performed. • ≠0 Rewrite operation are performed (default).

Additional information

This function is optional. Typically, the NOR flash devices are able to rewrite the same data and by default this feature is disabled at compile time. The FS_NOR_SUPPORT_VARIABLE_LINE_SIZE configuration define has to be set to a value different than 0 in order to enable this function, otherwise the function does nothing.

6.4.2.2.1.6 FS_NOR_SetDirtyCheckOptimization()

Description

Enables or disables the blank checking of a logical sector before write.

Prototype

```
void FS_NOR_SetDirtyCheckOptimization(U8 Unit,
                                      U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the sector map NOR driver (0-based).
OnOff	Specifies if the feature has to be enabled or disabled • =0 Dirty check is disabled (default). • ≠0 Dirty check is enabled.

Additional information

This function is optional. Per default, the NOR driver checks if the data of the logical sector is blank (all bytes 0xFF) before it writes the data. This is necessary in order to make sure that the NOR driver does not write to partially written logical sectors. Writing to a partially written logical sector can cause a data loss since the write operation can change the value of a bit only from 1 to 0. A partially written logical sector occurs when the write operation is interrupted by an unexpected reset. In this case the status of the logical sector indicates that the logical sector is blank which is not correct. Therefore the logical sector cannot be used for storage and it is marked by the NOR driver as invalid.

Typically, the blank checking runs fast but on some targets it may reduce the write performance. In this cases, this option can be used to skip the blank checking which helps improve the performance. When the optimization is enabled, the blank checking is not longer performed on the logical sectors located on physical sectors that have been erased at least once since the last mount operation. The NOR driver can skip the blank checking for these physical sectors since it knows that they do not contain any partially written logical sectors. The application can remove any partially written logical sectors by performing a clean operation of the entire storage via the FS_STORAGE_Clean() or FS_STORAGE_CleanOne() API functions. The NOR driver requires one bit of RAM storage for each physical sector used as storage.

The FS_NOR_OPTIMIZE_DIRTY_CHECK configuration define has to be set to a value different than 0 in order to enable this function, otherwise the function does nothing.

6.4.2.2.1.7 FS_NOR_SetEraseVerification()

Description

Enables or disables the checking of the sector erase operation.

Prototype

```
void FS_NOR_SetEraseVerification(U8 Unit,
                                 U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the sector map NOR driver (0-based).
OnOff	Specifies if the feature has to be enabled or disabled • =0 The erase operation is not checked. • ≠0 The erase operation is checked.

Additional information

This function is optional. The result of a sector erase operation is normally checked by evaluating the error bits maintained by the NOR flash device in a internal status register. FS_NOR_SetEraseVerification() can be used to enable additional verification of the sector erase operation that is realized by reading back the contents of the entire erased physical sector and by checking that all the bytes in it are set to 0xFF. Enabling this feature can negatively impact the write performance of sector map NOR driver.

The sector erase verification feature is active only when the sector map NOR driver is compiled with the FS_NOR_VERIFY_ERASE configuration define is set to 1 (default is 0) or when the FS_DEBUG_LEVEL configuration define is set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL.

6.4.2.2.1.8 FS_NOR_SetPhyType()

Description

Configures the type of NOR physical layer.

Prototype

```
void FS_NOR_SetPhyType(      U8                Unit,
                       const FS_NOR_PHY_TYPE * pPhyType);
```

Parameters

Parameter	Description
Unit	Index of the sector map NOR driver (0-based).
pPhyType	in NOR physical layer.

Additional information

This function is mandatory and it has to be called once in FS_X_AddDevices() for each instance of the sector map NOR driver created by the application. Different instances of the sector map NOR driver are identified by the Unit parameter.

Permitted values for the pPhyType parameter are:

Identifier	Description
FS_NOR_PHY_CFI_1x16	One CFI compliant NOR flash device with 16-bit interface.
FS_NOR_PHY_CFI_2x16	Two CFI compliant NOR flash device with 16-bit interfaces.
FS_NOR_PHY_DSPI	This a pseudo physical layer that uses the physical layers FS_NOR_PHY_ST_M25 and FS_NOR_PHY_SFDP
FS_NOR_PHY_SFDP	Serial NOR flash devices that support Serial Flash Discoverable Parameters (SFDP)
FS_NOR_PHY_SPIFI	Memory mapped serial quad NOR flash devices.
FS_NOR_PHY_ST_M25	Serial NOR flash devices compatible to ST ST25Pxx.

6.4.2.2.1.9 FS_NOR_SetSectorSize()

Description

Configures the number of bytes in a logical sector.

Prototype

```
void FS_NOR_SetSectorSize(U8  Unit,
                          U16 SectorSize);
```

Parameters

Parameter	Description
Unit	Index of the sector map NOR driver (0-based).
SectorSize	Number of bytes in a logical sector.

Additional information

This function is optional. It can be used to modify the size of the logical sector used by the sector map NOR driver. By default the sector map NOR driver uses the logical sector size configured a file system level that is set to 512 bytes at the file system initialization and can be later changed via FS_SetMaxSectorSize(). The NOR flash device has to be reformatted in order for the new logical sector size to take effect.

SectorSize has to be a power of 2 value.

6.4.2.2.1.10 FS_NOR_SetWriteVerification()

Description

Enables or disables the checking of the page write operation.

Prototype

```
void FS_NOR_SetWriteVerification(U8 Unit,
                                 U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the sector map NOR driver (0-based).
OnOff	Specifies if the feature has to be enabled or disabled • =0 The write operation is not checked. • ≠0 The write operation is checked.

Additional information

This function is optional. The result of a page write operation is normally checked by evaluating the error bits maintained by the NOR flash device in a internal status register. FS_NOR_SetWriteVerification() can be used to enable additional verification of the page write operation that is realized by reading back the contents of the written page and by checking that all the bytes are matching the data requested to be written. Enabling this feature can negatively impact the write performance of sector map NOR driver.

The page write verification feature is active only when the sector map NOR driver is compiled with the FS_NOR_VERIFY_WRITE configuration define is set to 1 (default is 0) or when the FS_DEBUG_LEVEL configuration define is set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL.

6.4.2.2.1.11 FS_NOR_STAT_COUNTERS

Description

Statistical counters maintained by the sector map NOR driver.

Type definition

```
typedef struct {
  U32  EraseCnt;
  U32  ReadSectorCnt;
  U32  WriteSectorCnt;
  U32  CopySectorCnt;
} FS_NOR_STAT_COUNTERS;
```

Structure members

Member	Description
EraseCnt	Number of sector erase operations.
ReadSectorCnt	Number of logical sectors read by the file system.
WriteSectorCnt	Number of logical sectors written by the file system.
CopySectorCnt	Number of logical sectors copied internally by the driver.

6.4.2.2.2 Additional sample configurations

The sample configurations below show how to create multiple volumes, logical volumes etc., on a NOR flash device. All configuration steps have to be performed inside the FS_X_AddDevices() function that is called during the initialization of the file system.

6.4.2.2.2.1 Multiple volumes on a single NOR flash device

The following example illustrates how to create multiple volumes on a single NOR flash device. The sample creates two volumes on one NOR flash device.

```
#include "FS.h"

#define ALLOC_SIZE 0x2000               // Size defined in bytes
//
// 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

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().
*/
void FS_X_AddDevices(void) {
  //
  // Give the file system memory to work with.
  //
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Volume name: "nor: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 name: "nor: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);
}
```

6.4.2.2.2.2 Multiple volumes on different NOR flash devices

The following example illustrates how to create multiple volumes on multiple NOR flash devices. This sample creates two volume, each one located on different NOR flash device.

```
#include "FS.h"

#define ALLOC_SIZE 0x2000               // Size defined in bytes
//
// Config:  2 NOR flash devices, 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
```



```
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().
*/
void FS_X_AddDevices(void) {
  //
  // Give the file system memory to work with.
  //
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Volume name: "nor: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 name: "nor: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);
}
```

6.4.2.2.2.3 Volume that stretches over multiple NOR flash devices

The following example illustrates how to create a volume that stretches over two NOR flash devices. This is realized by using the logical volume functionality.

```
#include "FS.h"

#define ALLOC_SIZE 0x2000               // Size defined in bytes
//
// Config:  2 NOR flash devices, 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

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().
*/
void FS_X_AddDevices(void) {
  //
  // Give the file system memory to work with.
  //
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // 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);
  //
  // 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);
```



```
  //
  // Now create a logical volume, containing the physical devices. Volume name: "LogVol"
  //
  FS_LOGVOL_Create("LogVol");
  FS_LOGVOL_AddDevice("LogVol", &FS_NOR_Driver, 0, 0, 0);
  FS_LOGVOL_AddDevice("LogVol", &FS_NOR_Driver, 1, 0, 0);
}
```


6.4.2.3 Additional driver functions

These functions are optional can be called to get information about the status of the Sector Map NOR driver or to directly access the data stored the NOR flash device.

Function	Description
FS_NOR_EraseDevice()	Erases all the physical sectors configured as storage.
FS_NOR_FormatLow()	Performs a low-level format of NOR flash device.
FS_NOR_GetDiskInfo()	Returns information about the organization and the management of the NOR flash device.
FS_NOR_GetSectorInfo()	Returns information about a specified physical sector.
FS_NOR_GetStatCounters()	Returns the values of the statistical counters.
FS_NOR_IsLLFormatted()	Checks it the NOR flash is low-level formatted.
FS_NOR_LogSector2PhySectorAddr()	Returns the address in memory of a specified logical sector.
FS_NOR_ReadOff()	Reads a range of bytes from the NOR flash device.
FS_NOR_ResetStatCounters()	Sets the value of the statistical counters to 0.

6.4.2.3.1 FS_NOR_EraseDevice()

Description

Erases all the physical sectors configured as storage.

Prototype

```
int FS_NOR_EraseDevice(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the sector map NOR driver (0-based).

Return value

= 0 Physical sectors erased.
≠ 0 An error occurred.

Additional information

This function is optional. After the call to this function all the bytes in area of the NOR flash device configured as storage are set to 0xFF.

6.4.2.3.2 FS_NOR_FormatLow()

Description

Performs a low-level format of NOR flash device.

Prototype

```
int FS_NOR_FormatLow(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the sector map NOR driver (0-based).

Return value

= 0 OK, NOR flash device has been successfully low-level formated.
≠ 0 An error occurred.

Additional information

This function is optional. FS_NOR_FormatLow() erases the first physical sector and stores the format information in it. The other physical sectors are either erased or invalidated. Per default the physical sectors are invalidated in order to reduce the time it takes for the operation to complete.

6.4.2.3.3 FS_NOR_GetDiskInfo()

Description

Returns information about the organization and the management of the NOR flash device.

Prototype

```
int FS_NOR_GetDiskInfo(U8                 Unit,
                       FS_NOR_DISK_INFO * pDiskInfo);
```

Parameters

Parameter	Description
Unit	Index of the sector map NOR driver (0-based).
pDiskInfo	out Requested information.

Return value

= 0 OK, information returned.
≠ 0 An error occurred.

Additional information

This function is not required for the functionality of the sector map NOR driver and will typically not be linked in production builds.

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

void SampleNORGetDiskInfo(void) {
  FS_NOR_DISK_INFO DiskInfo;

  printf("Get information about the first sector map NOR driver instance\n");
  FS_NOR_GetDiskInfo(0, &DiskInfo);
  printf(" Physical sectors: %d\n"
         " Logical sectors : %d\n"
         " Used sectors:     %d\n", DiskInfo.NumPhysSectors,
                                    DiskInfo.NumLogSectors,
                                    DiskInfo.NumUsedSectors);
}
```


6.4.2.3.4 FS_NOR_GetSectorInfo()

Description

Returns information about a specified physical sector.

Prototype

```
void FS_NOR_GetSectorInfo(U8                   Unit,
                          U32                  PhySectorIndex,
                          FS_NOR_SECTOR_INFO * pSectorInfo);
```

Parameters

Parameter	Description
Unit	Index of the sector map NOR driver (0-based).
PhySectorIndex	Index of the physical sector to be queried (0-based).
pSectorInfo	out Information related to the specified physical sector.

Additional information

This function is optional. The application can use it to get information about the usage of a particular physical sector.

PhySectorIndex is relative to the beginning of the region configured as storage via FS_NOR_Configure().

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

void SampleNORGetDiskInfo(void) {
  FS_NOR_SECTOR_INFO SectorInfo;

  printf("Get information about the physical sector 0 of the first sector map NOR driver
 instance\n");
  FS_NOR_GetSectorInfo(0, 0, &SectorInfo);
  printf(" Offset:                   %d\n"
         " Size:                     %d bytes\n"
         " Erase Count:              %d\n"
         " Used logical sectors:     %d\n"
         " Free logical sectors:     %d\n"
         " Erasable logical sectors: %d\n", SectorInfo.Off,
                                            SectorInfo.Size,
                                            SectorInfo.EraseCnt,
                                            SectorInfo.NumUsedSectors,
                                            SectorInfo.NumFreeSectors,
                                            SectorInfo.NumEraseableSectors);
}
```


6.4.2.3.5 FS_NOR_GetStatCounters()

Description

Returns the values of the statistical counters.

Prototype

```
void FS_NOR_GetStatCounters(U8                    Unit,
                            FS_NOR_STAT_COUNTERS * pStat);
```

Parameters

Parameter	Description
Unit	Index of the sector map NOR driver (0-based).
pStat	out Statistical counter values.

Additional information

This function is optional. The application can use it to get the actual values of the statistical counters maintained by the sector map NOR driver. The statistical counters provide information about the number of internal operations performed by the sector map NOR driver such as sector read and write. All statistical counters are set to 0 when the NOR flash device is low-level mounted. The application can explicitly set them to 0 by using FS_NOR_ResetStatCounters(). A separate set of statistical counters is maintained for each instance of the sector map NOR driver.

The statistical counters are available only when the sector map NOR driver is compiled with the FS_DEBUG_LEVEL configuration define set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL or with the FS_NOR_ENABLE_STATS configuration define set to 1.

6.4.2.3.6 FS_NOR_IsLLFormatted()

Description

Checks it the NOR flash is low-level formatted.

Prototype

```
int FS_NOR_IsLLFormatted(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the sector map NOR driver (0-based).

Return value

≠ 0 The NOR flash device is low-level formatted.
= 0 The NOR flash device is not low-level formatted or an error has occurred.

Additional information

This function is optional. An application should use `FS_IsLLFormatted()` instead.

6.4.2.3.7 FS_NOR_LogSector2PhySectorAddr()

Description

Returns the address in memory of a specified logical sector.

Prototype

```
void *FS_NOR_LogSector2PhySectorAddr(U8  Unit,
                                     U32 LogSectorIndex);
```

Parameters

Parameter	Description
Unit	Index of the sector map NOR driver (0-based).
LogSectorIndex	Index of the logical sector for which the address has to be calculated.

Return value

≠ NULL OK, address of the first byte in the logical sector.
= NULL An error occurred.

Additional information

This function is optional. It can be used only with NOR flash devices that are memory mapped. FS_NOR_LogSector2PhySectorAddr() returns the address in the system memory of the first byte in the specified logical sector.

6.4.2.3.8 FS_NOR_ReadOff()

Description

Reads a range of bytes from the NOR flash device.

Prototype

```
int FS_NOR_ReadOff(U8     Unit,
                   U32    Off,
                   void * pData,
                   U32    NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the sector map NOR driver (0-based).
Off	Offset of the first byte to be read.
pData	out Read data.
NumBytes	Number of bytes to be read.

Return value

= 0 OK, data read.
≠ 0 An error occurred.

Additional information

This function is not required for the functionality of the driver and will typically not be linked in production builds.

Off has to be specified in bytes and is relative to the beginning of the NOR flash area configured via FS_NOR_Configure().

6.4.2.3.9 FS_NOR_ResetStatCounters()

Description

Sets the value of the statistical counters to 0.

Prototype

```
void FS_NOR_ResetStatCounters(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the sector map NOR driver (0-based).

Additional information

This function is optional. The application can use it to set the statistical counters maintained by the sector map NOR driver to 0. The statistical counters can be read via `FS_NOR_GetStatCounters()`

`FS_NOR_ResetStatCounters()` is available only when the sector map NOR driver is compiled with the `FS_DEBUG_LEVEL` configuration define set to a value greater than or equal to `FS_DEBUG_LEVEL_CHECK_ALL` or with the `FS_NOR_ENABLE_STATS` configuration define set to 1.

6.4.2.3.10 FS_NOR_DISK_INFO

Description

Management information maintained by the sector map NOR driver.

Type definition

```
typedef struct {
  U32  NumPhysSectors;
  U32  NumLogSectors;
  U32  NumUsedSectors;
} FS_NOR_DISK_INFO;
```

Structure members

Member	Description
NumPhysSectors	Number of physical sectors available for data storage.
NumLogSectors	Number of available logical sectors.
NumUsedSectors	Number of logical sectors that store valid data.

6.4.2.3.11 FS_NOR_SECTOR_INFO

Description

Information about a physical sector maintained by the sector map NOR driver.

Type definition

```
typedef struct {
  U32  Off;
  U32  Size;
  U32  EraseCnt;
  U16  NumUsedSectors;
  U16  NumFreeSectors;
  U16  NumEraseableSectors;
  U8   Type;
} FS_NOR_SECTOR_INFO;
```

Structure members

Member	Description
Off	Position of the physical sector relative to the first byte of NOR flash device
Size	Size of physical sector in bytes
EraseCnt	Number of times the physical sector has been erased
NumUsedSectors	Number of logical sectors that contain valid data
NumFreeSectors	Number of logical sectors that are empty (i.e. blank)
NumEraseableSectors	Number of logical sectors that contain old (i.e. invalid) data.
Type	Type of data stored in the physical sector (see FS_NOR_SECTOR_TYPE_...)

6.4.2.4 Performance and resource usage

This section provides information about the ROM and RAM usage as well as the performance of the Sector Map NOR driver. Each driver instance requires one instance of one NOR physical layer in order to operate. The resource usage of the used NOR physical layer has to be taken into account when calculating the total resource usage of the Sector Map NOR driver. Refer to the section *Resource usage* on page 685 for information about the resource usage of the available NOR physical layers.

6.4.2.4.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the Sector Map NOR driver was measured using the SEGGER Embedded Studio IDE V4.20 configured to generate code for a Cortex-M4 CPU in Thumb mode and with the size optimization enabled.

Usage: 4.0 Kbytes

6.4.2.4.2 Static RAM usage

Static RAM usage refers to the amount of RAM required by the Sector Map NOR driver internally for all the driver instances. The number of bytes can be seen in the compiler list file of the `FS_NOR_Drv.c` file.

Usage: 20 bytes

6.4.2.4.3 Dynamic RAM usage

Dynamic RAM usage is the amount of RAM allocated by the driver at runtime. The amount of RAM required depends on the runtime configuration and on the used NOR flash device. The approximate RAM usage of the Sector Map NOR driver can be calculated as follows:

```
MemAllocated = 500 + (BitsPerEntry * FlashSize / 8) / LogSectorSize
```

Parameter	Description
`MemAllocated`	Number of bytes allocated.
`BitsPerEntry`	Number of bits required to store the offset of the last byte on the storage.
`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()`.

The following table lists the approximate amount of RAM required for different combinations of NOR flash size and logical sector size:

Flash size (Mbytes)	512 byte sectors	1 Kbyte sectors	2 Kbyte sectors
1	4.6 KBytes	2.5 KBytes	1.5 KBytes
2	8.7 KBytes	4.6 KBytes	2.5 KBytes
4	16.8 KBytes	8.7 KBytes	4.6 KBytes
8	33.2 KBytes	16.8 KBytes	8.7 KBytes

> **Note**
>
> Please note that by increasing the logical sector size the RAM usage of the file system increases too because more memory is required for the sector buffers.

6.4.2.4.4 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. All values are given in Kbytes/second

CPU type	NOR flash device	Write speed	Read speed
ST STR912 (96 MHz)	Winbond W25Q32BV (SPI)	**75**	**2625**
NXP LPC2478 (57.6 MHz)	SST SST39VF201 CFI NOR flash device with 16-bit interface and without support for write buffer	**53.5**	**2560**
ST STM32F103 (72 MHz)	ST M29W128 CFI NOR flash device 16-bit interface and with support for write buffer	**60.4**	**8000**
ST STM32F103 (72 MHz)	ST M25P64 SPI NOR flash device	**62.8**	**1125**
NXP LPC4357 (204 MHz)	Spansion S25FL032P serial NOR flash device interfaced via SPIFI.	**185**	**26947**

6.4.2.5 FAQs

Q: How many physical sectors are reserved by the driver?
A: The driver reserves 2 physical sectors for its internal use.

6.4.3 Block Map NOR driver

This section describes the NOR driver which is optimized for reduced RAM usage. It works by mapping blocks of logical sectors to physical sectors of the NOR flash device.

6.4.3.1 Theory of operation

Differentiating between logical sectors or blocks and physical sectors is essential to understand this section. A logical sector or block is the smallest readable and writable unit of any file system and its usual size is 512 bytes. A physical sector is an array of bytes on the NOR flash device that are erased together (typically between 2 Kbytes - 128 Kbytes). The NOR flash driver is an abstraction layer between these two types of sectors.

The Block Map NOR driver maintains a table that maps ranges of logical sectors, called logical blocks, to physical sectors on the NOR flash device. 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 updated, its content is written to a special physical sector called work block. A work block is a kind of temporary storage for the modified data of a logical sector that is later converted into a data block when a new empty work block needs to be allocated.

6.4.3.2 Configuring the driver

The Block Map NOR driver has to be configured at runtime and optionally at compile time. The following sections described how this can be realized in an application.

6.4.3.2.1 Compile time configuration

The Block Map NOR driver can optionally be configured at compile time. Typically, this step can be omitted because reasonable default values are provided that work with most of the applications. The compile time configuration is realized via preprocessor defines that have to be added to the `FS_Conf.h` file which is the main configuration file of emFile. For detailed information about the configuration of emFile and of the configuration define types, refer to *Configuration of emFile* on page 927 The following table lists the configuration defines supported by the Block Map NOR driver.

Define	Default value	Type	Description
FS_NOR_CAN_REWRITE	1	B	Indicates if modifying the same byte on the NOR flash device more than one time is permitted.
FS_NOR_ENABLE_STATS	0 or 1	B	Enables or disables the support for statistical counters.
FS_NOR_LINE_SIZE	4	N	Number of bytes in a write unit.
FS_NOR_NUM_READ_RETRIES	10	N	Configures the number of retries in case of a read error.
FS_NOR_NUM_WRITE_RETRIES	10	N	Configures the number of retries in case of a write error.
FS_NOR_NUM_UNITS	4	N	Maximum number of driver instances.
FS_NOR_VERIFY_WRITE	0	B	Reads back and compares the sector data to check if the write operation was successful.
FS_NOR_VERIFY_ERASE	0	B	Reads back and compares the sector data to check it the erase operation was successful.

6.4.3.2.1.1 FS_NOR_CAN_REWRITE

For the majority of NOR flash device the same byte can be modified more than once without a erase operation in between while preserving the bits set to 0. That is a byte can be written

up to eight times with each write operation setting a different bit in that byte to 0. The Block Map NOR driver makes use of this feature to make efficient use of the storage space and to improve the performance. FS_NOR_CAN_REWRITE has to be set to 0 for NOR flash devices that permit only one write operation to a byte or a range of bytes between two erase operations of that storage block.

6.4.3.2.1.2 FS_NOR_ENABLE_STATS

This define can be used to enable the support for statistical counters. The statistical counters provide information about the number of operations performed internally by the Block Map NOR driver that can be useful for debugging. The statistical counters can be queried via FS_NOR_BM_GetStatCounters(). By default FS_NOR_ENABLE_STATS is set to 1 if FS_DEBUG_LEVEL is set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL.

6.4.3.2.1.3 FS_NOR_LINE_SIZE

FS_NOR_LINE_SIZE specifies the minimum number of bytes the Block Map NOR driver writes at once. The amount of data written is always a multiple of this value. In addition, FS_NOR_LINE_SIZE specifies the alignment of the written data. That is the offset at which the Block Map NOR driver writes the data is always a multiple of this value. FS_NOR_LINE_SIZE has to be a power of two value.

6.4.3.2.1.4 FS_NOR_NUM_READ_RETRIES

FS_NOR_NUM_READ_RETRIES specifies the maximum number of times the Block Map NOR driver repeats a read operation in case of an error. A read retry is performed each time the read function of the NOR physical layer reports an error or if the CRC feature is activated when the CRC verification fails. NumReadRetries of FS_NOR_BM_STAT_COUNTERS is incremented by 1 on each retry.

6.4.3.2.1.5 FS_NOR_NUM_WRITE_RETRIES

FS_NOR_NUM_WRITE_RETRIES specifies the maximum number of times the Block Map NOR driver repeats a write operation in case of an error. A write retry is performed each time the write function of the NOR physical layer reports an error.

6.4.3.2.1.6 FS_NOR_NUM_UNITS

This define specifies the maximum number of driver instances of the Block Map NOR driver the application is allowed create. Four bytes of static RAM are reserved for each instance. If the maximum number of driver instances is smaller than the default then FS_NOR_NUM_UNITS can be set to the to that value in order to reduce the RAM usage.

6.4.3.2.1.7 FS_NOR_VERIFY_WRITE

This define can be used to activate the write verification of the Block Map NOR driver. The write verification is performed after each write operation by reading back the modified data from storage and by comparing it with the data requested to be written. An error is reported if a difference is detected. This feature has to be enabled at runtime by calling FS_NOR_BM_SetWriteVerification().

> **Note**
>
> Enabling this feature can negatively affect the write performance.

6.4.3.2.1.8 FS_NOR_VERIFY_ERASE

This define can be used to activate the erase verification of the Block Map NOR driver. The erase verification is performed after each erase operation by reading back the entire data of the erase physical sector from storage and by comparing it with 0xFF. An error is reported if a difference is detected. This feature has to be enabled at runtime by calling FS_NOR_BM_SetWriteVerification().

> **Note**
>
> Enabling this feature can negatively affect the write performance.

6.4.3.2.2 Runtime configuration

The driver must be added to the file system by calling `FS_AddDevice()` with the driver identifier set to the address of `FS_NOR_BM_Driver`. This function call together with other function calls that configure the driver operation have to be added to `FS_X_AddDevices()` as demonstrated in the following example. This example shows how to configure the file system to access a NOR flash device connected via SPI.

```
#include "FS.h"

#define ALLOC_SIZE 0x4000               // Size defined 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().
*/
void FS_X_AddDevices(void) {
  //
  // Give the file system memory to work with.
  //
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add and configure the NOR flash driver. The first 65536 bytes of the
  // serial NOR flash device are not used by the file system as data storage
  // and can be used by the application for other purposes. The file system
  // uses a total of 1 Mbyte from the serial NOR flash device as storage.
  //
  FS_AddDevice(&FS_NOR_BM_Driver);
  FS_NOR_BM_SetPhyType(0, &FS_NOR_PHY_ST_M25);
  FS_NOR_BM_Configure(0, 0x00000000, 0x00010000, 0x00100000);
}
```

The API functions listed in the next table can be used by the application to configure the behavior of the Block Map NOR driver. The application can call them only at the file system initialization in `FS_X_AddDevices()`.

Function	Description
`FS_NOR_BM_Configure()`	Configures an instance of the block map NOR driver.
`FS_NOR_BM_DisableCRC()`	Disables the data integrity check.
`FS_NOR_BM_EnableCRC()`	Enables the data integrity check.
`FS_NOR_BM_SetBlankSectorSkip()`	Configures if the physical sectors which are already blank should be erased during the low-level format operation.
`FS_NOR_BM_SetByteOrderBE()`	Sets the byte order of multi byte management data to big-endian (default).
`FS_NOR_BM_SetByteOrderLE()`	Sets the byte order of multi-byte management data to little-endian format.
`FS_NOR_BM_SetCRCHook()`	Configures the calculation routines for CRC verification.
`FS_NOR_BM_SetDeviceLineSize()`	Configures the minimum number of bytes that can be written to NOR flash.

Function	Description
FS_NOR_BM_SetDeviceRewriteSupport()	Specifies if the NOR flash device can rewrite the same data if 0s are preserved.
FS_NOR_BM_SetEraseVerification()	Enables or disables the checking of the sector erase operation.
FS_NOR_BM_SetFailSafeErase()	Configures the fail-safe mode of the sector erase operation.
FS_NOR_BM_SetInvalidSectorError()	Configures if an error is reported when an invalid sector is read.
FS_NOR_BM_SetMaxEraseCntDiff()	Configure the threshold for the active wear leveling operation.
FS_NOR_BM_SetNumWorkBlocks()	Configures the number of work blocks.
FS_NOR_BM_SetOnFatalErrorCallback()	Registers a function to be called by the driver encounters a fatal error.
FS_NOR_BM_SetPhyType()	Configures the type of NOR physical layer.
FS_NOR_BM_SetSectorSize()	Configures the number of bytes in a logical sector.
FS_NOR_BM_SetUsedSectorsErasure()	Configures if the physical sectors have to be erased at low-level format.
FS_NOR_BM_SetWriteVerification()	Enables or disables the checking of the page write operation.

6.4.3.2.2.1 FS_NOR_BM_Configure()

Description

Configures an instance of the block map NOR driver

Prototype

```
void FS_NOR_BM_Configure(U8  Unit,
                         U32 BaseAddr,
                         U32 StartAddr,
                         U32 NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).
BaseAddr	Address of the first byte in NOR flash.
StartAddr	Address of the first byte the NOR driver is permitted to use as storage.
NumBytes	Number of bytes starting from StartAddr available to be used by the NOR driver as storage.

Additional information

This function is mandatory and it has to be called once in FS_X_AddDevices() for each instance of the block map NOR driver created by the application. Different instances of the NOR driver are identified by the Unit parameter.

BaseAddr is used only for NOR flash devices that are memory mapped. For serial NOR flash devices that are not memory mapped BaseAddr has to be set to 0.

StartAddr has to be greater than or equal to BaseAddr and smaller than the total number of bytes in the NOR flash device. The block map NOR driver rounds up StartAddr to the start address of the next physical sector in the NOR flash device.

NumBytes is rounded up to a physical sector boundary if the memory range defined by StartAddr and NumBytes is smaller than the capacity of the NOR flash device. If the memory range defined by StartAddr and NumBytes is larger than the capacity of the NOR flash device than NumBytes is rounded down so that the memory range fits into the NOR flash device.

The block map NOR driver can work only with physical sectors of the same size. The longest continuous range of physical sectors with the same size is chosen that fits in the range defined by StartAddr and NumBytes parameters.

Example

```
#include "FS.h"
#include "FS_NOR_HW_SPIFI_Template.h"

#define ALLOC_SIZE 0x4000               // Size defined 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().
*/
void FS_X_AddDevices(void) {
  //
  // Give file system memory to work with.
```



```
  //
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Configure the size of the logical sector and activate the file buffering.
  //
  FS_SetMaxSectorSize(512);
  FS_ConfigFileBufferDefault(512, FS_FILE_BUFFER_WRITE);
  //
  // Add and configure the NOR driver.
  //
  FS_AddDevice(&FS_NOR_BM_Driver);
  FS_NOR_BM_SetPhyType(0, &FS_NOR_PHY_SPIFI);
  FS_NOR_BM_Configure(0, 0x80000000, 0x80000000, 0x01000000);
  FS_NOR_BM_SetSectorSize(0, 512);
  //
  // Configure the NOR physical layer.
  //
  FS_NOR_SPIFI_Allow2bitMode(0, 1);
  FS_NOR_SPIFI_Allow4bitMode(0, 1);
  FS_NOR_SPIFI_SetHWType(0, &FS_NOR_HW_SPIFI_Template);
}
```


6.4.3.2.2.2 FS_NOR_BM_DisableCRC()

Description

Disables the data integrity check.

Prototype

```
int FS_NOR_BM_DisableCRC(void);
```

Return value

= 0	Data integrity check deactivated.
≠ 0	An error occurred.

Additional information

The support for data integrity check is not available by default and it has to be included at compile time via the `FS_NOR_SUPPORT_CRC` define. This function disables the data integrity check for all instances of the NOR driver.

6.4.3.2.2.3 FS_NOR_BM_EnableCRC()

Description

Enables the data integrity check.

Prototype

```
int FS_NOR_BM_EnableCRC(void);
```

Return value

= 0	Data integrity check activated.
≠ 0	An error occurred.

Additional information

The support for data integrity check is not available by default and it has to be included at compile time via the FS_NOR_SUPPORT_CRC define. This function enables the data integrity check for all instances of the NOR driver.

6.4.3.2.2.4 FS_NOR_BM_SetBlankSectorSkip()

Description

Configures if the physical sectors which are already blank should be erased during the low-level format operation.

Prototype

```
void FS_NOR_BM_SetBlankSectorSkip(U8 Unit,
                                  U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).
OnOff	Specifies if the feature has to be enabled or disabled • =0 All the physical sectors are erased. • ≠0 Physical sectors are not erased if they are blank (default).

Additional information

This function is optional. The blank checking feature is disabled by default and has to be explicitly enabled at compile time by setting FS_NOR_SKIP_BLANK_SECTORS to 1. The feature can then be enabled or disabled at runtime using FS_NOR_BM_SetBlankSectorSkip().

Activating this feature can improve the speed of the low-level format operation when most of the physical sectors of the NOR flash device are already blank which is typically the case with NOR flash devices that ship from factory.

6.4.3.2.2.5 FS_NOR_BM_SetByteOrderBE()

Description

Sets the byte order of multi byte management data to big-endian (default).

Prototype

```
int FS_NOR_BM_SetByteOrderBE(void);
```

Return value

= 0	Multi-byte management data is stored in big-endian format.
≠ 0	An error occurred.

Additional information

This function is optional. Multi-byte management data is stored in the byte order of the CPU. Typically, `FS_NOR_BM_SetByteOrderBE()` is used by the NOR Image Creator utility to create NOR images with a byte order of the target CPU when the byte order of the host CPU is different. The byte order can be set to little-endian using `FS_NOR_BM_SetByteOrderLE()`. By default the byte order is set to little-endian.

The support for configurable byte order is disabled by default and has to be enabled by compiling the block map NOR driver with the `FS_NOR_SUPPORT_VARIABLE_BYTE_ORDER` configuration define set to 1 otherwise `FS_NOR_BM_SetByteOrderBE()` does nothing.

6.4.3.2.2.6 FS_NOR_BM_SetByteOrderLE()

Description

Sets the byte order of multi-byte management data to little-endian format.

Prototype

```
int FS_NOR_BM_SetByteOrderLE(void);
```

Return value

= 0	Multi-byte management data is stored in little-endian format.
≠ 0	An error occurred.

Additional information

This function is optional. Multi-byte management data is stored in the byte order of the CPU. Typically, `FS_NOR_BM_SetByteOrderLE()` is used by the NOR Image Creator utility to create NOR images with a byte order of the target CPU when the byte order of the host CPU is different. The byte order can be set to big-endian using `FS_NOR_BM_SetByteOrderBE()`. By default the byte order is set to little-endian.

The support for configurable byte order is disabled by default and has to be enabled by compiling the block map NOR driver with the `FS_NOR_SUPPORT_VARIABLE_BYTE_ORDER` configuration define set to 1 otherwise `FS_NOR_BM_SetByteOrderLE()` does nothing.

6.4.3.2.2.7 FS_NOR_BM_SetCRCHook()

Description

Configures the calculation routines for CRC verification.

Prototype

```
int FS_NOR_BM_SetCRCHook(const FS_NOR_CRC_HOOK * pCRCHook);
```

Parameters

Parameter	Description
pCRCHook	CRC calculation routines.

Return value

= 0 OK, CRC calculation routines set.
≠ 0 Error code indicating the failure reason.

Additional information

This function is optional. The driver uses by default the internal software routines of the file system for the CRC calculation. FS_NOR_BM_SetCRCHook() can be used by the application to specify different CRC calculation functions. This function is available only when the FS_NOR_SUPPORT_CRC configuration define is set to 1. Otherwise FS_NOR_BM_SetCRCHook() does nothing and returns an error. In order to save ROM space, the internal CRC software routines can be disabled a runtime by setting the FS_NOR_CRC_HOOK_DEFAULT configuration define to NULL.

6.4.3.2.2.8 FS_NOR_BM_SetDeviceLineSize()

Description

Configures the minimum number of bytes that can be written to NOR flash.

Prototype

```
int FS_NOR_BM_SetDeviceLineSize(U8 Unit,
                                U8 ldBytesPerLine);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).
ldBytesPerLine	Line size in bytes as power of 2 value.

Return value

= 0 OK, line size changed.
≠ 0 Error code indicating the failure reason.

Additional information

This function is optional. Typically, the NOR flash have lines smaller than 4 bytes which is the fixed default value configured at compile time. The FS_NOR_SUPPORT_VARIABLE_LINE_SIZE configuration define has to be set to a value different than 0 in order to enable this function.

6.4.3.2.2.9 FS_NOR_BM_SetDeviceRewriteSupport()

Description

Specifies if the NOR flash device can rewrite the same data if 0s are preserved.

Prototype

```
int FS_NOR_BM_SetDeviceRewriteSupport(U8 Unit,
                                      U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).
OnOff	Enables / disables the support for rewrite. • =0 Rewrite support is disabled. • ≠0 Rewrite support is enabled.

Return value

= 0 OK, support for rewrite changed.
≠ 0 Error code indicating the failure reason.

Additional information

This function is optional. Typically, the NOR flash devices are able to rewrite the same data and by default this feature is disabled at compile time. The FS_NOR_SUPPORT_VARIABLE_LINE_SIZE configuration define has to be set to a value different than 0 in order to enable this function, otherwise the function does nothing.

6.4.3.2.2.10 FS_NOR_BM_SetEraseVerification()

Description

Enables or disables the checking of the sector erase operation.

Prototype

```
void FS_NOR_BM_SetEraseVerification(U8 Unit,
                                    U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).
OnOff	Specifies if the feature has to be enabled or disabled • =0 The erase operation is not checked. • ≠0 The erase operation is checked.

Additional information

This function is optional. The result of a sector erase operation is normally checked by evaluating the error bits maintained by the NOR flash device in a internal status register. FS_NOR_BM_SetEraseVerification() can be used to enable additional verification of the sector erase operation that is realized by reading back the contents of the entire erased physical sector and by checking that all the bytes in it are set to 0xFF. Enabling this feature can negatively impact the write performance of block map NOR driver.

The sector erase verification feature is active only when the block map NOR driver is compiled with the FS_NOR_VERIFY_ERASE configuration define is set to 1 (default is 0) or when the FS_DEBUG_LEVEL configuration define is set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL.

6.4.3.2.2.11 FS_NOR_BM_SetFailSafeErase()

Description

Configures the fail-safe mode of the sector erase operation.

Prototype

```
int FS_NOR_BM_SetFailSafeErase(U8 Unit,
                               U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).
OnOff	Specifies if the feature has to be enabled or disabled • =0 Sector erase operation is not fail-safe. • ≠0 Sector erase operation is fail-safe.

Return value

= 0 OK, fail	safe mode configured.
≠ 0	Error code indicating the failure reason.

Additional information

This function is optional. It can be used by the application to enable or disable the fail-safe sector erase feature of the block map NOR driver. The fail-safe sector erase feature helps the driver identify a sector erase operation that did not completed successfully due to an unexpected power failure. The block map NOR driver enables the fail-safe sector erase feature by default except when the NOR flash device does not support modifying the same data more than once (rewrite) or when the application activates the data integrity check feature via CRC. It is not possible to use the fail-safe sector erase feature with a NOR flash device that cannot rewrite because the current implementation requires the overwriting of an already existing data. `FS_NOR_BM_SetFailSafeErase()` can be used to enable the fail-safe sector erase feature for configurations that enable the data integrity check via CRC. This has the potential of improving the performance of the clean operation executed via `FS_STORAGE_Clean()` and `FS_STORAGE_CleanOne()` at the expense of loosing the integrity check for the erase count stored in the header of each physical sector.

The value configured via `FS_NOR_BM_SetFailSafeErase()` is evaluated only during the low-level format operation. That is, after changing the activation status of the fail-safe sector erase feature a low-level format operation has to be performed via `FS_FormatLow()` in order for the setting to take effect.

6.4.3.2.2.12 FS_NOR_BM_SetInvalidSectorError()

Description

Configures if an error is reported when an invalid sector is read.

Prototype

```
void FS_NOR_BM_SetInvalidSectorError(U8 Unit,
                                     U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).
OnOff	Specifies if the feature has to be enabled or disabled • =0 No error is reported. • ≠0 An error is reported.

Additional information

This function is optional. An invalid sector is a logical sector that does not store any data. All logical sectors are invalid after a low-level format operation. A logical sector becomes valid as soon as the file system writes some data to it. The file system does not read invalid sectors with the exception of a high-level format operation.

By default the Block Map NOR driver does not report any error when the file system reads an invalid sector. The contents of an invalid sector is filled with the byte pattern specified via FS_NOR_READ_BUFFER_FILL_PATTERN.

6.4.3.2.2.13 FS_NOR_BM_SetMaxEraseCntDiff()

Description

Configure the threshold for the active wear leveling operation.

Prototype

```
void FS_NOR_BM_SetMaxEraseCntDiff(U8  Unit,
                                  U32 EraseCntDiff);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).
EraseCntDiff	Difference between the erase counts.

Additional information

This function is optional. The application can use FS_NOR_BM_SetMaxEraseCntDiff() to modify the value of the threshold that is used to decide if a data block has to be relocated to make sure that is equally erased. The block map NOR driver compares the difference between the erase counts of two blocks with the configured value. If the calculated difference is larger than EraseCntDiff then the block with the smaller erase count is copied to the block with the larger erase count then the block with the smaller erase count is used to store new data to it.

6.4.3.2.2.14 FS_NOR_BM_SetNumWorkBlocks()

Description

Configures the number of work blocks.

Prototype

```
void FS_NOR_BM_SetNumWorkBlocks(U8       Unit,
                                unsigned NumWorkBlocks);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).
NumWorkBlocks	Number of work blocks to configure.

Additional information

This function is optional. Work blocks are physical sectors that are used by the block map NOR driver to temporarily store the data written to NOR flash device by the file system layer. An application can use this function to modify the number of work blocks used by the block map NOR driver according to its requirements. By default, the block map NOR driver allocates 1% from the total number of physical sectors configured for storage but no more than 10 work blocks. The minimum number of work blocks allocated by default depends on whether journaling is activated or not in the application. If the journal is active the 4 work blocks are allocated otherwise 3.

The write performance of the block map NOR driver can be improved by increasing the number work blocks which at the same time increases the RAM usage.

The NOR flash device has to be reformatted in order for the new number of work blocks to take effect.

6.4.3.2.2.15 FS_NOR_BM_SetOnFatalErrorCallback()

Description

Registers a function to be called by the driver encounters a fatal error.

Prototype

```
void FS_NOR_BM_SetOnFatalErrorCallback
                                  (FS_NOR_ON_FATAL_ERROR_CALLBACK * pfOnFatalError);
```

Parameters

Parameter	Description
pfOnFatalError	Address to the callback function.

Additional information

Typically, the NOR driver reports a fatal error when data integrity check (CRC) fails.

All instances of the NOR driver share the same callback function. The Unit member of the FS_NOR_FATAL_ERROR_INFO structure passed as parameter to the pfOnFatalError callback function indicates which driver instance triggered the fatal error.

6.4.3.2.2.16 FS_NOR_BM_SetPhyType()

Description

Configures the type of NOR physical layer.

Prototype

```
void FS_NOR_BM_SetPhyType(      U8                Unit,
                          const FS_NOR_PHY_TYPE * pPhyType);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).
pPhyType	in NOR physical layer.

Additional information

This function is mandatory and it has to be called once in FS_X_AddDevices() for each instance of the block map NOR driver created by the application. Different instances of the block map NOR driver are identified by the Unit parameter.

Permitted values for the pPhyType parameter are:

Identifier	Description
FS_NOR_PHY_CFI_1x16	One CFI compliant NOR flash device with 16-bit interface.
FS_NOR_PHY_CFI_2x16	Two CFI compliant NOR flash device with 16-bit interfaces.
FS_NOR_PHY_DSPI	This a pseudo physical layer that uses the physical layers FS_NOR_PHY_ST_M25 and FS_NOR_PHY_SFDP
FS_NOR_PHY_SFDP	Serial NOR flash devices that support Serial Flash Discoverable Parameters (SFDP)
FS_NOR_PHY_SPIFI	Memory mapped serial quad NOR flash devices.
FS_NOR_PHY_ST_M25	Serial NOR flash devices compatible to ST ST25Pxx.

6.4.3.2.2.17 FS_NOR_BM_SetSectorSize()

Description

Configures the number of bytes in a logical sector.

Prototype

```
void FS_NOR_BM_SetSectorSize(U8       Unit,
                             unsigned SectorSize);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).
SectorSize	Number of bytes in a logical sector.

Additional information

This function is optional. It can be used to modify the size of the logical sector used by the block map NOR driver. By default the block map NOR driver uses the logical sector size configured a file system level that is set to 512 bytes at the file system initialization and can be later changed via FS_SetMaxSectorSize(). The NOR flash device has to be reformatted in order for the new logical sector size to take effect.

SectorSize has to be a power of 2 value.

6.4.3.2.2.18 FS_NOR_BM_SetUsedSectorsErasure()

Description

Configures if the physical sectors have to be erased at low-level format.

Prototype

```
void FS_NOR_BM_SetUsedSectorsErasure(U8 Unit,
                                     U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).
OnOff	Specifies if the feature has to be enabled or disabled • =0 Physical sectors are invalidated (default). • ≠0 Physical sectors are erased.

Additional information

This function is optional. The default behavior of the block map NOR driver is to invalidate the physical sectors at low-level format which makes the format operation faster.

6.4.3.2.2.19 FS_NOR_BM_SetWriteVerification()

Description

Enables or disables the checking of the page write operation.

Prototype

```
void FS_NOR_BM_SetWriteVerification(U8 Unit,
                                    U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).
OnOff	Specifies if the feature has to be enabled or disabled • =0 The write operation is not checked. • ≠0 The write operation is checked.

Additional information

This function is optional. The result of a page write operation is normally checked by evaluating the error bits maintained by the NOR flash device in a internal status register. FS_NOR_BM_SetWriteVerification() can be used to enable additional verification of the page write operation that is realized by reading back the contents of the written page and by checking that all the bytes are matching the data requested to be written. Enabling this feature can negatively impact the write performance of block map NOR driver.

The page write verification feature is active only when the block map NOR driver is compiled with the FS_NOR_VERIFY_WRITE configuration define is set to 1 (default is 0) or when the FS_DEBUG_LEVEL configuration define is set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL.

6.4.3.2.2.20 FS_NOR_CRC_HOOK

Description

Callback functions for the CRC calculation.

Type definition

```
typedef struct {
  FS_NOR_CRC_HOOK_CALC_CRC8  * pfCalcCRC8;
  FS_NOR_CRC_HOOK_CALC_CRC16 * pfCalcCRC16;
} FS_NOR_CRC_HOOK;
```

Structure members

Member	Description
pfCalcCRC8	Calculates an 8-bit CRC.
pfCalcCRC16	Calculates a 16-bit CRC.

Additional information

By default the Block Map NOR driver uses internal functions of the file system for the CRC calculation. This structure can be used to configure external CRC calculation callback functions that for example use hardware CRC calculation units for increase performance. The application can register the calculation callback functions via FS_NOR_BM_SetCRCHook()

6.4.3.2.2.21 FS_NOR_CRC_HOOK_CALC_CRC8

Description

The type of the callback function invoked by the NOR driver to calculate an 8-bit CRC.

Type definition

```
typedef U8 FS_NOR_CRC_HOOK_CALC_CRC8(const U8     * pData,
                                      unsigned NumBytes,
                                      U8       crc);
```

Parameters

Parameter	Description
pData	in Data to be protected by CRC.
NumBytes	Number of bytes to be protected by CRC.
crc	Initial CRC value.

Return value

Calculated 8-bit CRC.

Additional information

This function calculates the CRC of the NumBytes of data pointed to by pData. The type of polynomial is not relevant.

6.4.3.2.2.22 FS_NOR_CRC_HOOK_CALC_CRC16

Description

The type of the callback function invoked by the NOR driver to calculate a 16-bit CRC.

Type definition

```
typedef U16 FS_NOR_CRC_HOOK_CALC_CRC16(const U8     * pData,
                                       unsigned NumBytes,
                                       U16      crc);
```

Parameters

Parameter	Description
pData	in Data to be protected by CRC.
NumBytes	Number of bytes to be protected by CRC.
crc	Initial CRC value.

Return value

Calculated 16-bit CRC.

Additional information

This function calculates the CRC of the NumBytes of data pointed to by pData. The type of polynomial is not relevant. NumBytes is always an even number. pData is always aligned to a 16-bit boundary.

6.4.3.2.2.23 FS_NOR_FATAL_ERROR_INFO

Description

Information passed to callback function when a fatal error occurs.

Type definition

```
typedef struct {
  U8   Unit;
  U8   ErrorType;
  U32  ErrorPSI;
} FS_NOR_FATAL_ERROR_INFO;
```

Structure members

Member	Description
Unit	Index of the driver that triggered the fatal error.
ErrorType	Type of the error that occurred.
ErrorPSI	Index of the physical sector in which the error occurred.

6.4.3.2.2.24 FS_NOR_ON_FATAL_ERROR_CALLBACK

Description

The type of the callback function invoked by the NOR driver when a fatal error occurs.

Type definition

```
typedef int FS_NOR_ON_FATAL_ERROR_CALLBACK(FS_NOR_FATAL_ERROR_INFO * pFatalErrorInfo);
```

Parameters

Parameter	Description
pFatalErrorInfo	Information about the fatal error.

Return value

= 0 The NOR driver has to mark the NOR flash device as read only.
≠ 0 The NOR flash device has to remain writable.

Additional information

If the callback function returns a 0 the NOR driver marks the NOR flash device as read-only and it remains in this state until the NOR flash device is low-level formated. In this state, all further write operations are rejected with an error by the NOR driver.

The application is responsible to handle the fatal error by for example checking the consistency of the file system via FS_CheckDisk(). The callback function is not allowed to invoke any other FS API functions therefore the handling of the error has to be done after the FS API function that triggered the error returns.

6.4.3.3 Additional driver functions

These functions are optional. They can be used to get information about the sector map NOR driver and to directly access the data stored on the NOR flash device.

Function	Description
FS_NOR_BM_EraseDevice()	Erases all the physical sectors configured as storage.
FS_NOR_BM_ErasePhySector()	Sets all the bits in a physical sector to 1.
FS_NOR_BM_FormatLow()	Performs a low-level format of NOR flash device.
FS_NOR_BM_GetDiskInfo()	Returns information about the organization and the management of the NOR flash device.
FS_NOR_BM_GetSectorInfo()	Returns information about a specified physical sector.
FS_NOR_BM_GetStatCounters()	Returns the values of the statistical counters.
FS_NOR_BM_IsCRCEnabled()	Checks if the data integrity checking is enabled.
FS_NOR_BM_IsLLFormatted()	Checks it the NOR flash is low-level formatted.
FS_NOR_BM_ReadOff()	Reads a range of bytes from the NOR flash device.
FS_NOR_BM_ResetStatCounters()	Sets the value of the statistical counters to 0.
FS_NOR_BM_SuspendWearLeveling()	Disables temporarily the wear leveling process.
FS_NOR_BM_WriteOff()	Writes data to NOR flash memory.

6.4.3.3.1 FS_NOR_BM_EraseDevice()

Description

Erases all the physical sectors configured as storage.

Prototype

```
int FS_NOR_BM_EraseDevice(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).

Return value

= 0 Physical sectors erased.
≠ 0 An error occurred.

Additional information

This function is optional. After the call to this function all the bytes in area of the NOR flash device configured as storage are set to 0xFF.

6.4.3.3.2 FS_NOR_BM_ErasePhySector()

Description

Sets all the bits in a physical sector to 1.

Prototype

```
int FS_NOR_BM_ErasePhySector(U8  Unit,
                             U32 PhySectorIndex);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).
PhySectorIndex	Index of the physical sector to be erased.

Return value

= 0 OK, physical sector erased successfully.
≠ 0 An error occurred.

Additional information

PhySectorIndex is 0-based and is relative to the beginning of the NOR flash area configured via FS_NOR_BM_Configure(). The number of bytes actually erased depends on the size of the physical sector supported by the NOR flash device. Information about a physical sector can be obtained via FS_NOR_BM_GetSectorInfo().

6.4.3.3.3 FS_NOR_BM_FormatLow()

Description

Performs a low-level format of NOR flash device.

Prototype

```
int FS_NOR_BM_FormatLow(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).

Return value

= 0 OK, NOR flash device has been successfully low-level formated.
≠ 0 An error occurred.

Additional information

This function is optional. FS_NOR_BM_FormatLow() erases the first physical sector and stores the format information in it. The other physical sectors are either erased or invalidated. Per default the physical sectors are invalidated in order to reduce the time it takes for the operation to complete. The application can request the block map NOR driver to erase the physical sectors instead of invalidating them by calling FS_NOR_BM_SetUsedSectorsErasure() with the Off parameter set to 1.

6.4.3.3.4 FS_NOR_BM_GetDiskInfo()

Description

Returns information about the organization and the management of the NOR flash device.

Prototype

```
int FS_NOR_BM_GetDiskInfo(U8                    Unit,
                          FS_NOR_BM_DISK_INFO * pDiskInfo);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).
pDiskInfo	out Requested information.

Return value

= 0 OK, information returned.
≠ 0 An error occurred.

Additional information

This function is not required for the functionality of the block map NOR driver and will typically not be linked in production builds.

6.4.3.3.5 FS_NOR_BM_GetSectorInfo()

Description

Returns information about a specified physical sector.

Prototype

```
int FS_NOR_BM_GetSectorInfo(U8                      Unit,
                            U32                     PhySectorIndex,
                            FS_NOR_BM_SECTOR_INFO * pSectorInfo);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).
PhySectorIndex	Index of the physical sector to be queried (0-based).
pSectorInfo	out Information related to the specified physical sector.

Return value

= 0 OK, information returned.
≠ 0 An error occurred.

Additional information

This function is optional. The application can use it to get information about the usage of a particular physical sector.

PhySectorIndex is relative to the beginning of the region configured as storage via FS_NOR_BM_Configure().

6.4.3.3.6 FS_NOR_BM_GetStatCounters()

Description

Returns the values of the statistical counters.

Prototype

```
void FS_NOR_BM_GetStatCounters(U8                        Unit,
                               FS_NOR_BM_STAT_COUNTERS * pStat);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).
pStat	out Statistical counter values.

Additional information

This function is optional. The application can use it to get the actual values of the statistical counters maintained by the block map NOR driver. The statistical counters provide information about the number of internal operations performed by the block map NOR driver such as sector read and write. All statistical counters are set to 0 when the NOR flash device is low-level mounted. The application can explicitly set them to 0 by using FS_NOR_BM_ResetStatCounters(). A separate set of statistical counters is maintained for each instance of the block map NOR driver.

The statistical counters are available only when the block map NOR driver is compiled with the FS_DEBUG_LEVEL configuration define set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL or with the FS_NOR_ENABLE_STATS configuration define set to 1.

6.4.3.3.7 FS_NOR_BM_IsCRCEnabled()

Description

Checks if the data integrity checking is enabled.

Prototype

```
int FS_NOR_BM_IsCRCEnabled(void);
```

Return value

= 0	Data integrity check is deactivated.
≠ 0	Data integrity check is activated.

Additional information

This function is available only the file system sources are compiled with `FS_NOR_SUPPORT_CRC` set to 1.

6.4.3.3.8 FS_NOR_BM_IsLLFormatted()

Description

Checks it the NOR flash is low-level formatted.

Prototype

```
int FS_NOR_BM_IsLLFormatted(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).

Return value

≠ 0 The NOR flash device is low-level formatted.
= 0 The NOR flash device is not low-level formatted or an error has occurred.

Additional information

This function is optional. An application should use `FS_IsLLFormatted()` instead.

6.4.3.3.9 FS_NOR_BM_ReadOff()

Description

Reads a range of bytes from the NOR flash device.

Prototype

```
int FS_NOR_BM_ReadOff(U8     Unit,
                      void * pData,
                      U32    Off,
                      U32    NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).
pData	out Read data.
Off	Offset of the first byte to be read.
NumBytes	Number of bytes to be read.

Return value

= 0 OK, data read.
≠ 0 An error occurred.

Additional information

This function is not required for the functionality of the driver and will typically not be linked in production builds.

Off has to be specified in bytes and is relative to the beginning of the NOR flash area configured via FS_NOR_BM_Configure().

6.4.3.3.10 FS_NOR_BM_ResetStatCounters()

Description

Sets the value of the statistical counters to 0.

Prototype

```
void FS_NOR_BM_ResetStatCounters(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).

Additional information

This function is optional. The application can use it to set the statistical counters maintained by the block map NOR driver to 0. The statistical counters can be read via FS_NOR_BM_GetStatCounters()

FS_NOR_BM_ResetStatCounters() is available only when the block map NOR driver is compiled with the FS_DEBUG_LEVEL configuration define set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL or with the FS_NOR_ENABLE_STATS configuration define set to 1.

6.4.3.3.11 FS_NOR_BM_SuspendWearLeveling()

Description

Disables temporarily the wear leveling process.

Prototype

```
void FS_NOR_BM_SuspendWearLeveling(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).

Additional information

This function is optional. Disabling the wear leveling can help reduce the write latencies that occur when the block map NOR driver has to make space for the new data by erasing storage blocks that contain invalid data. With the wear leveling disabled the block map NOR driver searches for an already empty block (i.e. a storage block that does not have to be erased before use) when it needs more space. Empty blocks are created when the application performs a clean operation via FS_STORAGE_Clean() or FS_STORAGE_CleanOne(). The wear leveling is automatically re-enabled by the block map NOR driver when no more empty storage blocks are available and an erase operation is required to create one.

FS_NOR_BM_SuspendWearLeveling() disables only the wear leveling operation performed when the file system writes data to NOR flash device. The wear leveling is still performed during a clean operation.

The activation status of the wear leveling is returned via the IsWLSuspended member of the FS_NOR_BM_DISK_INFO structure that can be queried via FS_NOR_BM_GetDiskInfo()

6.4.3.3.12 FS_NOR_BM_WriteOff()

Description

Writes data to NOR flash memory.

Prototype

```
int FS_NOR_BM_WriteOff(      U8     Unit,
                       const void * pData,
                             U32    Off,
                             U32    NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the block map NOR driver (0-based).
pData	in Data to be written.
Off	Location where to write the data.
NumBytes	Number of bytes to be written.

Return value

= 0 OK, data written successfully.
≠ 0 An error occurred.

Additional information

Off has to be specified in bytes and is relative to the beginning of the NOR flash area configured via FS_NOR_BM_Configure().

FS_NOR_BM_WriteOff() is able to write across page and physical sector boundaries. This function can only change bit values from 1 to 0. The bits can be set to 1 block-wise via FS_NOR_BM_ErasePhySector().

The function takes care of the alignment required when writing to NOR flash devices with line size larger than 1.

6.4.3.3.13 FS_NOR_BM_DISK_INFO

Description

Management information maintained by the block map NOR driver.

Type definition

```
typedef struct {
  U16  NumPhySectors;
  U16  NumLogBlocks;
  U16  NumUsedPhySectors;
  U16  LSectorsPerPSector;
  U16  BytesPerSector;
  U32  EraseCntMax;
  U32  EraseCntMin;
  U32  EraseCntAvg;
  U8   HasFatalError;
  U8   ErrorType;
  U32  ErrorPSI;
  U8   IsWriteProtected;
  U8   IsWLSuspended;
  U32  MaxEraseCntDiff;
} FS_NOR_BM_DISK_INFO;
```

Structure members

Member	Description
NumPhySectors	Number of physical sectors available for data storage.
NumLogBlocks	Number blocks available for the storage of file system data.
NumUsedPhySectors	Number of physical sectors currently in use.
LSectorsPerPSector	Number of logical sectors that are mapped in a physical sector.
BytesPerSector	Size of the logical sector used by the driver in bytes.
EraseCntMax	Maximum value of an erase count.
EraseCntMin	Minimum value of an erase count.
EraseCntAvg	Average value of an erase count.
HasFatalError	Indicates if a fatal error has occurred during the operation (0 - no fatal error, 1 - a fatal error occurred).
ErrorType	Code indicating the type of fatal error that occurred.
ErrorPSI	Index of the physical sector where the fatal error occurred.
IsWriteProtected	Indicates if the NOR flash device has been switched permanently to read-only mode as a result of a fatal error (0 - NOR flash device is writable, 1 - NOR flash device is write protected)
IsWLSuspended	Indicates if the wear leveling process is temporarily suspended (0 - active, 1 - suspended)
MaxEraseCntDiff	Difference between the erase counts of two physical sectors that trigger an active wear leveling operation.

6.4.3.3.14 FS_NOR_BM_SECTOR_INFO

Description

Information about a physical sector maintained by the block map NOR driver.

Type definition

```
typedef struct {
  U32  Off;
  U32  Size;
  U32  EraseCnt;
  U16  lbi;
  U8   Type;
} FS_NOR_BM_SECTOR_INFO;
```

Structure members

Member	Description
Off	Position of the physical sector relative to the first byte of NOR flash device
Size	Size of 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 (see FS_NOR_BLOCK_TYPE_...)

6.4.3.3.14.1 FS_NOR_BM_STAT_COUNTERS

Description

Statistical counters maintained by the block map NOR driver.

Type definition

```
typedef struct {
  U32  NumFreeBlocks;
  U32  EraseCnt;
  U32  ReadSectorCnt;
  U32  WriteSectorCnt;
  U32  ConvertViaCopyCnt;
  U32  ConvertInPlaceCnt;
  U32  NumValidSectors;
  U32  CopySectorCnt;
  U32  NumReadRetries;
  U32  ReadPSHCnt;
  U32  WritePSHCnt;
  U32  ReadLSHCnt;
  U32  WriteLSHCnt;
  U32  ReadCnt;
  U32  ReadByteCnt;
  U32  WriteCnt;
  U32  WriteByteCnt;
} FS_NOR_BM_STAT_COUNTERS;
```

Structure members

Member	Description
NumFreeBlocks	Number of blocks that are not used for data storage.
EraseCnt	Number of sector erase operations.
ReadSectorCnt	Number of logical sectors read by the file system.
WriteSectorCnt	Number of logical sectors written by the file system.
ConvertViaCopyCnt	Number of times a work block has been converted via a copy operation.
ConvertInPlaceCnt	Number of times a work block has been converted in-place (i.e. by changing the block type from work to data).
NumValidSectors	Number of valid logical sectors stored on the NOR flash device.
CopySectorCnt	Number of logical sectors copied internally by the driver.
NumReadRetries	Number of times the driver retried a read operation due to an error.
ReadPSHCnt	Number of times the driver read from the header of a physical sector.
WritePSHCnt	Number of times the driver wrote to the header of a physical sector.
ReadLSHCnt	Number of times the driver read from the header of a logical sector.
WriteLSHCnt	Number of times the driver wrote to the header of a logical sector.
ReadCnt	Number of times the driver read data from the NOR flash device.
ReadByteCnt	Number of bytes read by the driver from the NOR flash device.

Member	Description
WriteCnt	Number of times the driver wrote data to the NOR flash device.
WriteByteCnt	Number of bytes written by the driver to the NOR flash device.

Additional information

The statistical counters can be queried via `FS_NOR_BM_GetStatCounters()` and can be set to 0 via `FS_NOR_BM_ResetStatCounters()`.

6.4.3.4 Performance and resource usage

6.4.3.4.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the Sector Map NOR driver was measured using the SEGGER Embedded Studio IDE V4.20 configured to generate code for a Cortex-M4 CPU in Thumb mode and with the size optimization enabled.

Usage: 5.5 Kbytes

In addition, one of the physical layers listed in the section ROM usage on page 432 is required.

6.4.3.4.2 Static RAM usage

Static RAM usage refers to the amount of RAM required by the Block Map NOR driver internally for all the driver instances. The number of bytes can be seen in the compiler list file of the `FS_NOR_BM_Drv.c` file.

Usage: 72 bytes

6.4.3.4.3 Dynamic RAM usage

Dynamic RAM usage is the amount of RAM allocated by the driver at runtime. The amount of RAM required depends on the runtime configuration and on the used NOR flash device. The approximate RAM usage of the Block Map NOR driver can be calculated as follows:

```
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.
`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.

6.4.3.4.4 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. All values are given in Kbytes/second

CPU type	NOR flash device	Write speed	Read speed
NXP LPC2478 (57.6 MHz)	SST SST39VF201 CFI NOR flash device with 16-bit interface and without support for write buffer	**45.5**	**2064**
ST STM32F103 (72 MHz)	ST M25P64 serial NOR flash device	**59.3**	**1110**
NXP LPC4357 (204 MHz)	Two Spansion S29GL064N CFI NOR flash devices with 16-bit interface and with support for write buffer.	**179**	**25924**

6.4.4 NOR physical layer

6.4.4.1 General information

The NOR physical layer provides the basic functionality for accessing a NOR flash device such as device identification, block erase operation, read and write operations, etc. Every instance of the Sector map or Block Map NOR driver requires an instance of a NOR physical layer in order to be able to operate. A NOR physical layer instance is automatically allocated either statically or dynamically at the first call to one of its API functions. Each instance is identified by a unit number that is identical with the unit number of the NOR driver that uses that instance of the NOR physical layer. The type of the NOR physical layer assigned to an instance of a Sector map or Block Map NOR driver is configured via `FS_NOR_SetPhyType()` and `FS_NOR_BM_SetPhyType()` respectively.

The following sections provide information about the usage and the implementation of a NOR physical layer.

6.4.4.2 Available physical layers

The table below lists the NOR physical layers that ship with emFile. Refer to *Configuring the driver* on page 560 and *Configuring the driver* on page 593 for detailed information about how to add and configure a physical layer in an application.

Physical layer identifier	Hardware layer type
`FS_NOR_PHY_CFI_1x16`	Not required.
`FS_NOR_PHY_CFI_2x16`	Not required.
`FS_NOR_PHY_DSPI`	`FS_NOR_HW_TYPE_SPI`
`FS_NOR_PHY_SFDP`	`FS_NOR_HW_TYPE_SPI`
`FS_NOR_PHY_SPIFI`	`FS_NOR_HW_TYPE_SPIFI`
`FS_NOR_PHY_ST_M25`	`FS_NOR_HW_TYPE_SPI`

6.4.4.2.1 CFI 1x16 physical layer

This physical layer supports any CFI compliant NOR flash device connected via a 16-bit data bus.

The table below summarizes the specific functions of the physical layer followed by a detailed description of each function.

Function	Description
FS_NOR_CFI_SetAddrGap()	Configures a memory access gap.
FS_NOR_CFI_SetReadCFICallback()	Registers a read function for the CFI parameters.

6.4.4.2.1.1 FS_NOR_CFI_SetAddrGap()

Description

Configures a memory access gap.

Prototype

```
void FS_NOR_CFI_SetAddrGap(U8  Unit,
                           U32 StartAddr,
                           U32 NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based)
StartAddr	Address of the first byte in the gap.
NumBytes	Number of bytes in the gap.

Additional information

This function is optional. The application can use FS_NOR_CFI_SetAddrGap() to specify a range in the memory region where the contents of the NOR flash device is mapped that is not assigned to the NOR flash device. Any access to an address equal to or greater than StartAddr is translated by NumBytes. StartAddr and NumBytes have to be aligned to a physical sector boundary of the used NOR flash device.

The application is permitted to call FS_NOR_CFI_SetAddrGap() only during the file system initialization in FS_X_AddDevices().

Example

```
#include "FS.h"

#define ALLOC_SIZE            0x4000
#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

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().
*/
void FS_X_AddDevices(void) {
  //
  // Give the file system memory to work with.
  //
  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);
}
```


6.4.4.2.1.2 FS_NOR_CFI_SetReadCFICallback()

Description

Registers a read function for the CFI parameters.

Prototype

```
void FS_NOR_CFI_SetReadCFICallback(U8                          Unit,
                                   FS_NOR_READ_CFI_CALLBACK * pReadCFI);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based)
pReadCFI	Function to be registered.

Additional information

This function is optional. It can be used to specify a different function for the reading of CFI parameters than the default function used by the physical layer. This is typically required when the CFI parameters do not fully comply with the CFI specification.

The application is permitted to call `FS_NOR_CFI_SetReadCFICallback()` only during the file system initialization in `FS_X_AddDevices()`.

Example

```
#include "FS.h"

#define ALLOC_SIZE                0x4000      // Size defined in bytes
#define BYTES_PER_SECTOR          2048

#define CFI_READCONFIG(BaseAddr)              // TBD
#define CFI_RESET(BaseAddr)                   // TBD
#define CFI_READCONFIG_NON_CONFORM(BaseAddr)  // TBD

static U32 _aMemBlock[ALLOC_SIZE / 4];        // Memory pool used for
                                              // semi-dynamic allocation.
static int _IsInited       = 0;
static int _IsCFICompliant = 0;

/*********************************************************************
*
*       _cbReadCFI
*/
static void _cbReadCFI(U8 Unit, U32 BaseAddr, U32 Off, U8 * pData, unsigned NumItems) {
  volatile U16 FS_NOR_FAR * pAddr;
  U8                        aData[3];

  FS_USE_PARA(Unit);
  //
  // We initially need to check whether the flash is fully CFI compliant.
  //
  if (_IsInited == 0) {
    FS_MEMSET(aData, 0, sizeof(aData));
    pAddr = (volatile U16 FS_NOR_FAR *)(BaseAddr + (0x10 << 1));
    CFI_READCONFIG(BaseAddr);
    aData[0] = (U8)*pAddr++;
    aData[1] = (U8)*pAddr++;
    aData[2] = (U8)*pAddr++;
    CFI_RESET(BaseAddr);
    if (   (aData[0] == 'Q')
        && (aData[1] == 'R')
        && (aData[2] == 'Y')) {
        _IsCFICompliant = 1;
    }
    _IsInited = 1;
  }
  pAddr = (volatile U16 FS_NOR_FAR *)(BaseAddr + (Off << 1));
```



```
  //
  // Write the correct CFI-query sequence
  //
  if (_IsCFICompliant) {
    CFI_READCONFIG(BaseAddr);
  } else {
    CFI_READCONFIG_NON_CONFORM(BaseAddr);
  }
  //
  // Read the data
  //
  do {
    *pData++ = (U8)*pAddr++;  // Only the low byte of the CFI data is relevant
  } while(--NumItems);
  //
  // Perform a reset, which means, return from CFI mode to normal mode
  //
  CFI_RESET(BaseAddr);
}

/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*/
void FS_X_AddDevices(void) {
  //
  // Give the file system memory to work with.
  //
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add and configure the NOR flash driver.
  //
  FS_AddDevice(&FS_NOR_Driver);
  FS_NOR_SetPhyType(0, &FS_NOR_PHY_CFI_1x16);
  //
  // Configure a different function for reading CFI information.
  //
  FS_NOR_CFI_SetReadCFICallback(0, _cbReadCFI);
  FS_NOR_Configure(0, 0x10000000, 0x10000000, 0xFFFFFFFF);
  FS_NOR_SetSectorSize(0, BYTES_PER_SECTOR);
  //
  // Set a larger logical sector size as the default
  // in order to reduce the RAM usage of the NOR driver.
  //
  FS_SetMaxSectorSize(BYTES_PER_SECTOR);
}
```


6.4.4.2.1.3 FS_NOR_READ_CFI_CALLBACK

Description

Reads CFI information from a NOR flash device.

Type definition

```
typedef void (FS_NOR_READ_CFI_CALLBACK)(U8       Unit,
                                        U32      BaseAddr,
                                        U32      Off,
                                        U8     * pData,
                                        unsigned NumItems);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based).
BaseAddr	Address in system memory where the NOR flash device is mapped.
Off	Byte offset to read from.
pData	out CFI information read
NumItems	Number of 16-bit values to be read.

Additional information

This is the type of the callback function invoked by CFI NOR physical layers to read CFI information from NOR flash device. The CFI NOR physical layers already have an internal function for reading the CFI information. FS_NOR_CFI_SetReadCFICallback() can be used to register a callback function that replaces the functionality of this internal function.

Example

Refer to FS_NOR_CFI_SetReadCFICallback() for a sample usage.

6.4.4.2.2 CFI 2x16 physical layer

This physical layer supports any setup with two CFI compliant NOR flash devices each one being connected via a 16-bit data bus.

6.4.4.2.3 DSPI physical layer

This a pseudo physical layer that uses the physical layers FS_NOR_PHY_ST_M25 and FS_NOR_PHY_SFDP to access a serial NOR flash device by exchanging the data via 1 SPI data line.

The table below summarizes the specific functions of the physical layer followed by a detailed description of each function.

Function	Description
FS_NOR_DSPI_SetHWType()	Configures the HW access routines.

6.4.4.2.3.1 FS_NOR_DSPI_SetHWType()

Description

Configures the HW access routines.

Prototype

```
void FS_NOR_DSPI_SetHWType(      U8                    Unit,
                           const FS_NOR_HW_TYPE_SPI * pHWType);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based).
pHWType	Hardware access routines. Cannot be NULL.

Additional information

This function is mandatory and it has to be called once for each instance of the physical layer.

6.4.4.2.4 SFDP physical layer

This physical layer support any serial NOR flash device that complies with the JEDEC JESD216 specification. This specification standardizes the method of describing the capabilities of a serial NOR flash device. This information is stored internally by any compatible serial NOR flash device and is used by the SFDP physical layer to get information about the capacity of the NOR flash device, the codes of specific read commands and so on.

The table below summarizes the specific functions of the physical layer followed by a detailed description of each function.

Function	Description
FS_NOR_SFDP_Allow2bitMode()	Specifies if the physical layer is permitted to exchange data via two data lines.
FS_NOR_SFDP_Allow4bitMode()	Specifies if the physical layer is permitted to exchange data via four data lines.
FS_NOR_SFDP_SetDeviceList()	Configures the type of handled serial NOR flash devices.
FS_NOR_SFDP_SetDeviceParaList()	Configures parameters of serial NOR flash devices.
FS_NOR_SFDP_SetHWType()	Configures the HW access routines.
FS_NOR_SFDP_SetSectorSize()	Configures the size of the physical sector to be used by the driver.

6.4.4.2.4.1 FS_NOR_SFDP_Allow2bitMode()

Description

Specifies if the physical layer is permitted to exchange data via two data lines.

Prototype

```
void FS_NOR_SFDP_Allow2bitMode(U8 Unit,
                               U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based).
OnOff	Activation status of the option. • 0 Disable the option. • 1 Enable the option.

Additional information

This function is optional. By default the data is exchanged via one data line (standard SPI). The data transfer via two data lines is used only if this type of data transfer is supported by the serial NOR flash device. In dual mode two bits of data are transferred with each clock period which helps improve the performance. If the serial NOR flash device does not support the dual mode then the data is transferred in standard mode (one data bit per clock period).

The application is permitted to call this function only at the file system initialization in FS_X_AddDevices().

6.4.4.2.4.2 FS_NOR_SFDP_Allow4bitMode()

Description

Specifies if the physical layer is permitted to exchange data via four data lines.

Prototype

```
void FS_NOR_SFDP_Allow4bitMode(U8 Unit,
                               U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based).
OnOff	Activation status of the option. • 0 Disable the option. • 1 Enable the option.

Additional information

This function is optional. By default the data is exchanged via one data line (standard SPI). The data transfer via four data lines is used only if this type of data transfer is supported by the serial NOR flash device. In quad mode four bits of data are transferred on each clock period which helps improve the performance. If the serial NOR flash device does not support the quad mode then the data is transferred in dual mode if enabled and supported or in standard mode (one bit per clock period).

The application is permitted to call this function only at the file system initialization in FS_X_AddDevices().

6.4.4.2.4.3 FS_NOR_SFDP_SetDeviceList()

Description

Configures the type of handled serial NOR flash devices.

Prototype

```
void FS_NOR_SFDP_SetDeviceList(      U8                         Unit,
                               const FS_NOR_SPI_DEVICE_LIST * pDeviceList);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based).
pDeviceList	List of NOR flash devices the physical layer can handle.

Additional information

This function is optional. This function can be used to enable handling for vendor specific features of serial NOR flash device such as error handling and data protection. By default the physical layer is configured to handle only Micron serial NOR flash devices. Handling for serial NOR flash devices from other manufacturers has to be explicitly enabled via this function.

Permitted values for the pDeviceList parameter are:

Identifier	Description
FS_NOR_SPI_DeviceList_All	Enables handling of serial NOR flash devices from all manufacturers.
FS_NOR_SPI_DeviceList_Cypress	Enables handling of Cypress serial NOR flash devices.
FS_NOR_SPI_DeviceList_Default	Enables handling of Micron and of SFDP compatible serial NOR flash devices.
FS_NOR_SPI_DeviceList_GigaDevice	Enables handling of GigaDevice serial NOR flash devices.
FS_NOR_SPI_DeviceList_ISSI	Enables handling of ISSI serial NOR flash devices.
FS_NOR_SPI_DeviceList_Macronix	Enables handling of Macronix serial NOR flash devices.
FS_NOR_SPI_DeviceList_Micron	Enables handling of Micron serial NOR flash devices.
FS_NOR_SPI_DeviceList_Micron_x	Enables handling of Micron serial NOR flash devices in single and dual chip setups.
FS_NOR_SPI_DeviceList_Micron_x2	Enables handling of Micron serial NOR flash devices in dual chip setups.
FS_NOR_SPI_DeviceList_Microchip	Enables handling of Microchip serial NOR flash devices.
FS_NOR_SPI_DeviceList_Spansion	Enables handling of Spansion serial NOR flash devices.
FS_NOR_SPI_DeviceList_Winbond	Enables handling of Winbond serial NOR flash devices.

The application can save ROM space by setting FS_NOR_DEVICE_LIST_DEFAULT to NULL at compile time and by calling at runtime FS_NOR_SFDP_SetDeviceList() with the actual list of serial NOR flash devices that have to be handled.

The application is permitted to call this function only at the file system initialization in `FS_X_AddDevices()`.

6.4.4.2.4.4 FS_NOR_SFDP_SetDeviceParaList()

Description

Configures parameters of serial NOR flash devices.

Prototype

```
void FS_NOR_SFDP_SetDeviceParaList
                            (      U8                              Unit,
                             const FS_NOR_SPI_DEVICE_PARA_LIST * pDeviceParaList);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based).
pDeviceParaList	List of device parameters.

Additional information

This function is optional. By default, the parameters of the used serial NOR flash device are determined by evaluating the SFDP tables stored in it. However, the information about the commands that can be used to write the data via two and four data lines is not stored to this parameters. `FS_NOR_SFDP_SetDeviceParaList()` can be used to specify this information. The parameters are matched by comparing the first byte (manufacturer id) and the third byte (device id) of the information returned by the READ ID (`0x9F`) function with the MfgId and Id members of the `FS_NOR_SPI_DEVICE_PARA` structure.

The application is permitted to call this function only at the file system initialization in `FS_X_AddDevices()`.

6.4.4.2.4.5 FS_NOR_SFDP_SetHWType()

Description

Configures the HW access routines.

Prototype

```
void FS_NOR_SFDP_SetHWType(      U8                    Unit,
                           const FS_NOR_HW_TYPE_SPI * pHWType);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based).
pHWType	Hardware access routines. Cannot be NULL.

Additional information

It is mandatory to call this function during the file system initialization in FS_X_AddDevices() once for each instance of a physical layer.

6.4.4.2.4.6 FS_NOR_SFDP_SetSectorSize()

Description

Configures the size of the physical sector to be used by the driver.

Prototype

```
void FS_NOR_SFDP_SetSectorSize(U8  Unit,
                               U32 BytesPerSector);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based).
BytesPerSector	Sector size to be used.

Additional information

Typically, a serial NOR flash device supports erase commands that can be used to erase sectors of different sizes (4 KB, 32 KB, etc.) For performance reasons the physical layer chooses always the erase command corresponding to the largest physical sector. This function can be used to request the physical layer to use a different (smaller) physical sector size. The mount operation fails if the serial NOR flash device does not support the specified physical sector size.

The application is permitted to call this function only at the file system initialization in FS_X_AddDevices().

6.4.4.2.4.7 FS_NOR_SPI_DEVICE_PARA_LIST

Description

Defines a list of serial NOR flash device parameters.

Type definition

```
typedef struct {
  U8                              NumParas;
  const FS_NOR_SPI_DEVICE_PARA * pPara;
} FS_NOR_SPI_DEVICE_PARA_LIST;
```

Structure members

Member	Description
NumParas	Number of parameters
pPara	List of parameters.

6.4.4.2.5 SPIFI physical layer

This physical layer supports serial NOR flash devices that are mapped to the memory of the target system.

The table below summarizes the specific functions of the physical layer followed by a detailed description of each function.

Function	Description
FS_NOR_SPIFI_Allow2bitMode()	Specifies if the physical layer is permitted to exchange data via two data lines.
FS_NOR_SPIFI_Allow4bitMode()	Specifies if the physical layer is permitted to exchange data via four data lines.
FS_NOR_SPIFI_SetDeviceList()	Configures the type of handled serial NOR flash devices.
FS_NOR_SPIFI_SetDeviceParaList()	Configures parameters of serial NOR flash devices.
FS_NOR_SPIFI_SetHWType()	Configures the HW access routines.
FS_NOR_SPIFI_SetSectorSize()	Configures the size of the physical sector to be used by the driver.

6.4.4.2.5.1 FS_NOR_SPIFI_Allow2bitMode()

Description

Specifies if the physical layer is permitted to exchange data via two data lines.

Prototype

```
void FS_NOR_SPIFI_Allow2bitMode(U8 Unit,
                                U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based).
OnOff	Activation status of the option. • 0 Disable the option. • 1 Enable the option.

Additional information

This function is optional. By default the data is exchanged via one data line (standard SPI). The data transfer via two data lines is used only if this type of data transfer is supported by the serial NOR flash device. In dual mode two bits of data are transferred with each clock period which helps improve the performance. If the serial NOR flash device does not support the dual mode then the data is transferred in standard mode (one data bit per clock period).

The application is permitted to call this function only at the file system initialization in FS_X_AddDevices().

6.4.4.2.5.2 FS_NOR_SPIFI_Allow4bitMode()

Description

Specifies if the physical layer is permitted to exchange data via four data lines.

Prototype

```
void FS_NOR_SPIFI_Allow4bitMode(U8 Unit,
                                U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based).
OnOff	Activation status of the option. • 0 Disable the option. • 1 Enable the option.

Additional information

This function is optional. By default the data is exchanged via one data line (standard SPI). The data transfer via four data lines is used only if this type of data transfer is supported by the serial NOR flash device. In quad mode four bits of data are transferred on each clock period which helps improve the performance. If the serial NOR flash device does not support the quad mode then the data is transferred in dual mode if enabled and supported or in standard mode (one bit per clock period).

The application is permitted to call this function only at the file system initialization in FS_X_AddDevices().

6.4.4.2.5.3 FS_NOR_SPIFI_SetDeviceList()

Description

Configures the type of handled serial NOR flash devices.

Prototype

```
void FS_NOR_SPIFI_SetDeviceList(      U8                         Unit,
                                const FS_NOR_SPI_DEVICE_LIST * pDeviceList);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based).
pDeviceList	List of NOR flash devices the physical layer can handle.

Additional information

This function is optional. By default the physical layer is configured to handle only Micron serial NOR flash devices. Handling for serial NOR flash devices from other manufacturers has to be explicitly enabled via this function.

Permitted values for the pDeviceList parameter are:

Identifier	Description
FS_NOR_SPI_DeviceList_All	Enables handling of serial NOR flash devices from all manufacturers.
FS_NOR_SPI_DeviceList_Cypress	Enables handling of Cypress serial NOR flash devices.
FS_NOR_SPI_DeviceList_Default	Enables handling of Micron and of SFDP compatible serial NOR flash devices.
FS_NOR_SPI_DeviceList_GigaDevice	Enables handling of GigaDevice serial NOR flash devices.
FS_NOR_SPI_DeviceList_ISSI	Enables handling of ISSI serial NOR flash devices.
FS_NOR_SPI_DeviceList_Macronix	Enables handling of Macronix serial NOR flash devices.
FS_NOR_SPI_DeviceList_Micron	Enables handling of Micron serial NOR flash devices.
FS_NOR_SPI_DeviceList_Micron_x	Enables handling of Micron serial NOR flash devices in single and dual chip setups.
FS_NOR_SPI_DeviceList_Micron_x2	Enables handling of Micron serial NOR flash devices in dual chip setups.
FS_NOR_SPI_DeviceList_Microchip	Enables handling of Microchip serial NOR flash devices.
FS_NOR_SPI_DeviceList_Spansion	Enables handling of Spansion serial NOR flash devices.
FS_NOR_SPI_DeviceList_Winbond	Enables handling of Winbond serial NOR flash devices.

The application can save ROM space by setting FS_NOR_DEVICE_LIST_DEFAULT to NULL at compile time and by calling at runtime FS_NOR_SFDP_SetDeviceList() with the actual list of serial NOR flash devices that have to be handled.

The application is permitted to call this function only at the file system initialization in FS_X_AddDevices().

6.4.4.2.5.4 FS_NOR_SPIFI_SetDeviceParaList()

Description

Configures parameters of serial NOR flash devices.

Prototype

```
void FS_NOR_SPIFI_SetDeviceParaList
                            (      U8                              Unit,
                             const FS_NOR_SPI_DEVICE_PARA_LIST * pDeviceParaList);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based).
pDeviceParaList	List of device parameters.

Additional information

This function is optional. This function can be used to enable handling for vendor specific features of serial NOR flash device such as error handling and data protection. By default, the parameters of the used serial NOR flash device are determined by evaluating the SFDP tables stored in it. However, the information about the commands that can be used to write the data via two and four data lines is not stored to this parameters. FS_NOR_SFDP_SetDeviceParaList() can be used to specify this information. The parameters are matched by comparing the first byte (manufacturer id) and the third byte (device id) of the information returned by the READ ID (0x9F) function with the MfgId and Id members of the FS_NOR_SPI_DEVICE_PARA structure.

The application is permitted to call this function only at the file system initialization in FS_X_AddDevices().

6.4.4.2.5.5 FS_NOR_SPIFI_SetHWType()

Description

Configures the HW access routines.

Prototype

```
void FS_NOR_SPIFI_SetHWType(      U8                       Unit,
                            const FS_NOR_HW_TYPE_SPIFI * pHWType);
```

Parameters

Parameter	Description
Unit	Index of the physical layer.
pHWType	Hardware layer to be used for data exchanged.

Additional information

It is mandatory to call this function during the file system initialization in `FS_X_AddDevices()` once for each instance of a physical layer.

6.4.4.2.5.6 FS_NOR_SPIFI_SetSectorSize()

Description

Configures the size of the physical sector to be used by the driver.

Prototype

```
void FS_NOR_SPIFI_SetSectorSize(U8  Unit,
                                U32 BytesPerSector);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based).
BytesPerSector	Sector size to be used.

Additional information

Typically, a serial NOR flash device supports erase commands that can be used to erase sectors of different sizes (4 KB, 32 KB, etc.) For performance reasons the physical layer chooses always the erase command corresponding to the largest physical sector. This function can be used to request the physical layer to use a different (smaller) physical sector size. The mount operation fails if the serial NOR flash device does not support the specified physical sector size.

The application is permitted to call this function only at the file system initialization in FS_X_AddDevices().

6.4.4.2.6 ST M25 physical layer

This physical layer supports legacy standard serial NOR flash devices.

The table below summarizes the specific functions of the physical layer followed by a detailed description of each function.

Function	Description
FS_NOR_SPI_AddDevice()	Specifies parameters for a NOR flash device that has to be supported.
FS_NOR_SPI_Configure()	Configures the parameters of the NOR flash device.
FS_NOR_SPI_ReadDeviceId()	Reads device identification information from NOR flash device.
FS_NOR_SPI_SetHWType()	Configures the HW access routines.
FS_NOR_SPI_SetPageSize()	Specifies the number of bytes in page.

6.4.4.2.6.1 FS_NOR_SPI_AddDevice()

Description

Specifies parameters for a NOR flash device that has to be supported.

Prototype

```
void FS_NOR_SPI_AddDevice(const FS_NOR_SPI_DEVICE_PARA * pDevicePara);
```

Parameters

Parameter	Description
pDevicePara	Device parameters.

Additional information

This function is optional. It allows an application to define the parameters of a NOR flash device that is not yet supported by the physical layer. The maximum number of NOR flash devices that can be added to the list can be specified via FS_NOR_MAX_NUM_DEVICES define. By default this feature is disabled, that is FS_NOR_MAX_NUM_DEVICES is set to 0. The data pointed to by pDevicePara should remain valid until FS_DeInit() is called, because the function saves internally only the pointer value.

This function is available only when the file system is compiled with FS_NOR_MAX_NUM_DEVICES set to a value larger than 0.

6.4.4.2.6.2 FS_NOR_SPI_Configure()

Description

Configures the parameters of the NOR flash device.

Prototype

```
void FS_NOR_SPI_Configure(U8  Unit,
                          U32 SectorSize,
                          U16 NumSectors);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based).
SectorSize	The size of a physical sector in bytes.
NumSectors	The number of physical sectors available.

Additional information

This function is optional. By default the physical layer identifies the parameters of the NOR flash device automatically using the information returned by the READ ID (0x9F) command. This method does not work for some older ST M25 NOR flash devices. In this case the application can use FS_NOR_SPI_Configure() to specify the parameters of the NOR flash device. SPI NOR (M25 series) flash devices have uniform sectors, which means only one sector size is used for the entire device.

The capacity of the serial NOR flash device is determined as follows:

Value of 3rd byte	Capacity in Mbits
0x11	1
0x12	2
0x13	4
0x14	8
0x15	16
0x16	32
0x17	64
0x18	128

The application required to call FS_NOR_SPI_Configure(), only if the serial NOR flash device does not identify itself with one of the values specified in the table above.

SectorSize must be set to the size of the size of the storage area erased via the Block Erase (0xD8) command. NumSectors is the device capacity in bytes divided by SectorSize.

The application is permitted to call this function only at the file system initialization in FS_X_AddDevices().

Example

```
#include "FS.h"

#define ALLOC_SIZE        0x4000        // Size defined 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().
*/
void FS_X_AddDevices(void) {
  //
  // Give the file system memory to work with.
  //
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add the NOR driver driver.
  //
  FS_AddDevice(&FS_NOR_Driver);
  //
  // Configure manually a 16 Mbit 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);
}
```


6.4.4.2.6.3 FS_NOR_SPI_ReadDeviceId()

Description

Reads device identification information from NOR flash device.

Prototype

```
void FS_NOR_SPI_ReadDeviceId(U8         Unit,
                             U8       * pId,
                             unsigned   NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based).
pId	out Information read from NOR flash.
NumBytes	Number of bytes to read from NOR flash.

Additional information

The data returned by this function is the response to the READ ID (0x9F) command.

6.4.4.2.6.4 FS_NOR_SPI_SetHWType()

Description

Configures the HW access routines.

Prototype

```
void FS_NOR_SPI_SetHWType(      U8                    Unit,
                          const FS_NOR_HW_TYPE_SPI * pHWType);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based).
pHWType	Hardware access routines. Cannot be NULL.

Additional information

This function is mandatory and it has to be called once for each instance of the physical layer.

6.4.4.2.6.5 FS_NOR_SPI_SetPageSize()

Description

Specifies the number of bytes in page.

Prototype

```
void FS_NOR_SPI_SetPageSize(U8  Unit,
                            U16 BytesPerPage);
```

Parameters

Parameter	Description
Unit	Index of the physical layer (0-based).
BytesPerPage	Number of bytes in a page.

Additional information

This function is optional. A page is the largest amount of bytes that can be written at once to a serial NOR flash device. By default the physical layer uses a page size of 256 bytes a value that is supported by the majority of serial NOR flash devices.

The size of a page cannot be automatically detected by the physical layer at runtime. Therefore, if the used serial NOR flash device has a page size different than 256 bytes, FS_NOR_SPI_SetPageSize() has to be used to configure the page size to the actual value. The write operation fails if the page size used by the physical layer is larger than the page size used by the serial NOR flash device. The write operation works if the application specifies a smaller page size than the actual page size of the serial NOR flash device but the write performance will be worst.

BytesPerPage has to be a power of 2 value.

6.4.4.2.6.6 FS_NOR_SPI_DEVICE_PARA

Description

Defines the parameters of a serial NOR flash device.

Type definition

```
typedef struct {
  U8   Id;
  U8   ldBytesPerSector;
  U8   ldBytesPerPage;
  U8   NumBytesAddr;
  U16  NumSectors;
  U8   Flags;
  U8   MfgId;
  U8   CmdWrite112;
  U8   CmdWrite122;
  U8   CmdWrite114;
  U8   CmdWrite144;
} FS_NOR_SPI_DEVICE_PARA;
```

Structure members

Member	Description
Id	Value to identify the serial NOR flash device. This is the 3rd byte in the response to READ ID (0x9F) command.
ldBytesPerSector	Number of bytes in a physical sector as power of 2.
ldBytesPerPage	Number of bytes in a page as a power of 2.
NumBytesAddr	Number of address bytes. 4 for NOR flash devices with a capacity larger than 128 Mbit (16 Mbyte).
NumSectors	Total number of physical sectors in the device.
Flags	Additional functionality supported by device that requires special processing.
MfgId	Id of device manufacturer. This is the 1st byte in the response to READ ID (0x9F) command.
CmdWrite112	Code of the command used to write the data to NOR flash via 2 data lines. The command itself and the address are transferred via data 1 line.
CmdWrite122	Code of the command used to write the data to NOR flash via 2 data lines. The command itself is transferred via data 1 line while the address via 2 data lines.
CmdWrite114	Code of the command used to write the data to NOR flash via 4 data lines. The command itself and the address are transferred via data 1 line.
CmdWrite144	Code of the command used to write the data to NOR flash via 4 data lines. The command itself is transferred via data 1 line while the address via 4 data lines.

Additional information

Flags is an bitwise-OR combination of *Device operation flags* on page 673.

6.4.4.2.6.7 Device operation flags

Description

Values to be used for the Flags member of FS_NOR_SPI_DEVICE_PARA

Definition

```
#define FS_NOR_SPI_DEVICE_FLAG_ERROR_STATUS     (1u << 0)
#define FS_NOR_SPI_DEVICE_FLAG_WEL_ADDR_MODE    (1u << 1)
```

Symbols

Definition	Description
FS_NOR_SPI_DEVICE_FLAG_ERROR_STATUS	The device reports errors via Flag Status Register.
FS_NOR_SPI_DEVICE_FLAG_WEL_ADDR_MODE	The write enable latch has to be set before changing the address mode.

6.4.4.3 Physical layer API

The physical layers that come with emFile provide support most of the popular NOR flash device and target MCU types. Therefore, there is typically no need to modify any of the provided NOR physical layers. Typically, only the NOR hardware layer has to be adapted to a specific target hardware. However, when none of the provided NOR physical layers are compatible with the target hardware a new NOR physical layer implementation is required. This section provides information about the API of the NOR physical layer that helps to create a new physical layer from scratch or to modify an existing one.

The API of the physical layer is implemented as a structure of type FS_NOR_PHY_TYPE that contains pointers to functions. The following sections describe these functions in detail together with the data structure passed to these functions as parameters.

6.4.4.3.1 FS_NOR_PHY_TYPE

Description

NOR physical layer API.

Type definition

```
typedef struct {
  FS_NOR_PHY_TYPE_WRITE_OFF       * pfWriteOff;
  FS_NOR_PHY_TYPE_READ_OFF        * pfReadOff;
  FS_NOR_PHY_TYPE_ERASE_SECTOR    * pfEraseSector;
  FS_NOR_PHY_TYPE_GET_SECTOR_INFO * pfGetSectorInfo;
  FS_NOR_PHY_TYPE_GET_NUM_SECTORS * pfGetNumSectors;
  FS_NOR_PHY_TYPE_CONFIGURE       * pfConfigure;
  FS_NOR_PHY_TYPE_ON_SELECT_PHY   * pfOnSelectPhy;
  FS_NOR_PHY_TYPE_DE_INIT         * pfDeInit;
  FS_NOR_PHY_TYPE_IS_SECTOR_BLANK * pfIsSectorBlank;
  FS_NOR_PHY_TYPE_INIT            * pfInit;
} FS_NOR_PHY_TYPE;
```

Structure members

Member	Description
pfWriteOff	Writes data to the NOR flash device.
pfReadOff	Reads data from the NOR flash device.
pfEraseSector	Erases a NOR physical sector.
pfGetSectorInfo	Returns information about a NOR physical sector.
pfGetNumSectors	Returns the number of NOR physical sectors.
pfConfigure	Configures an instance of the NOR physical layer.
pfOnSelectPhy	Prepares the access to the NOR flash device.
pfDeInit	Frees the resources allocated by the NOR physical layer instance.
pfIsSectorBlank	Verifies if a NOR physical sector is blank.
pfInit	Initializes the instance of the NOR physical layer.

6.4.4.3.2 FS_NOR_PHY_TYPE_WRITE_OFF

Description

Writes data to the NOR flash device.

Type definition

```
typedef int FS_NOR_PHY_TYPE_WRITE_OFF(      U8     Unit,
                                            U32    Off,
                                      const void * pData,
                                            U32    NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)
Off	Byte offset to write to.
pData	in Data to be written to NOR flash device.
NumBytes	Number of bytes to be written.

Return value

= 0 OK, data written.
≠ 0 An error occurred.

Additional information

This function is a member of the NOR physical layer API and is mandatory to be implemented by each NOR physical layer. The NOR driver calls this function when it writes data to the NOR flash device. It is guaranteed that the NOR driver tries to modify memory regions of the NOR flash device that are erased. This means that the function does not have to check if the memory region it has to modify are already erased. FS_NOR_PHY_TYPE_WRITE_OFF has to be able to handle write requests that stretch over multiple physical sectors.

Off specifies a number of bytes relative to the StartAddr value specified in the call to FS_NOR_PHY_TYPE_CONFIGURE.

6.4.4.3.3 FS_NOR_PHY_TYPE_READ_OFF

Description

Reads data from the NOR flash device.

Type definition

```
typedef int FS_NOR_PHY_TYPE_READ_OFF(U8     Unit,
                                     void * pData,
                                     U32    Off,
                                     U32    NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)
pData	out Data read from NOR flash device.
Off	Byte offset to read from.
NumBytes	Number of bytes to be read.

Return value

= 0 OK, data read.
≠ 0 An error occurred.

Additional information

This function is a member of the NOR physical layer API and is mandatory to be implemented by each NOR physical layer. The NOR driver calls this function when it reads data from the NOR flash device. FS_NOR_PHY_TYPE_READ_OFF has to be able to handle read request that stretch over multiple physical sectors.

Off specifies a number of bytes relative to the StartAddr value specified in the call to FS_NOR_PHY_TYPE_CONFIGURE.

6.4.4.3.4 FS_NOR_PHY_TYPE_ERASE_SECTOR

Description

Erases a NOR physical sector.

Type definition

```
typedef int FS_NOR_PHY_TYPE_ERASE_SECTOR(U8           Unit,
                                         unsigned int SectorIndex);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)
SectorIndex	Index of the NOR physical sector to be erased.

Return value

= 0 OK, physical sector is not blank.
≠ 0 An error occurred.

Additional information

This function is a member of the NOR physical layer API and is mandatory to be implemented by each NOR physical layer. The NOR driver calls this function when it erases a physical sector NOR flash device. The erase operation must set to 1 all the bits in the specified physical sector.

SectorIndex is relative to StartAddr value specified in the call to FS_NOR_PHY_TYPE_CONFIGURE. The physical sector located at StartAddr has the index 0.

6.4.4.3.5 FS_NOR_PHY_TYPE_GET_SECTOR_INFO

Description

Returns information about a NOR physical sector.

Type definition

```
typedef void FS_NOR_PHY_TYPE_GET_SECTOR_INFO(U8           Unit,
                                             unsigned int SectorIndex,
                                             U32        * pOff,
                                             U32        * pNumBytes);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)
SectorIndex	Index of the NOR physical sector to be queried.
pOff	out Byte offset of the NOR physical sector. Can be NULL.
pNumBytes	out Number of bytes in the NOR physical sector. Can be NULL.

Additional information

This function is a member of the NOR physical layer API and is mandatory to be implemented by each NOR physical layer. The NOR driver calls this function when it tries to determine the position and the size of a physical sector.

The value returned via pOff and SectorIndex are relative to StartAddr value specified in the call to FS_NOR_PHY_TYPE_CONFIGURE.

6.4.4.3.6 FS_NOR_PHY_TYPE_GET_NUM_SECTORS

Description

Returns the number of NOR physical sectors.

Type definition

```
typedef int FS_NOR_PHY_TYPE_GET_NUM_SECTORS(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)

Return value

≠ 0 OK, the number of NOR physical sectors.
= 0 An error occurred.

Additional information

This function is a member of the NOR physical layer API and is mandatory to be implemented by each NOR physical layer. The value returned by this function is the number of physical sectors located in the memory region specified by StartAddr and NumBytes values specified in the call to FS_NOR_PHY_TYPE_CONFIGURE.

6.4.4.3.7 FS_NOR_PHY_TYPE_CONFIGURE

Description

Configures an instance of the NOR physical layer.

Type definition

```
typedef void FS_NOR_PHY_TYPE_CONFIGURE(U8  Unit,
                                       U32 BaseAddr,
                                       U32 StartAddr,
                                       U32 NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)
BaseAddr	Address in system memory where the first byte of the NOR flash device is mapped.
StartAddr	Address of the first byte of the NOR flash device to be used for data storage.
NumBytes	Number of bytes to be used for data storage.

Additional information

This function is a member of the NOR physical layer API and is mandatory to be implemented by each NOR physical layer. FS_NOR_PHY_TYPE_CONFIGURE is called by the NOR driver during the initialization of the file system.

For more information about the parameters refer to FS_NOR_Configure() and FS_NOR_BM_Configure().

6.4.4.3.8 FS_NOR_PHY_TYPE_ON_SELECT_PHY

Description

Prepares the access to the NOR flash device.

Type definition

```
typedef void FS_NOR_PHY_TYPE_ON_SELECT_PHY(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)

Additional information

This function is a member of the NOR physical layer API and is mandatory to be implemented by each NOR physical layer. The NOR driver calls this function during the file system initialization when a NOR physical layer is assigned to a driver instance. Typically, FS_NOR_PHY_TYPE_ON_SELECT_PHY allocates the memory requires for the instance of the NOR physical layer and set up everything requires for the access to NOR flash device.

6.4.4.3.9 FS_NOR_PHY_TYPE_DE_INIT

Description

Frees the resources allocated by the NOR physical layer instance.

Type definition

```
typedef void FS_NOR_PHY_TYPE_DE_INIT(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)

Additional information

This function is a member of the NOR physical layer API. It is mandatory to be implemented only by the NOR physical layers that allocate dynamic memory. `FS_NOR_PHY_TYPE_DE_INIT` is called by the NOR driver at the file system deinitialization when the application calls `FS_DeInit()`. It has to free any dynamic memory allocated for the instance of specified the NOR physical layer.

6.4.4.3.10 FS_NOR_PHY_TYPE_IS_SECTOR_BLANK

Description

Verifies if a NOR physical sector is blank.

Type definition

```
typedef int FS_NOR_PHY_TYPE_IS_SECTOR_BLANK(U8           Unit,
                                            unsigned int SectorIndex);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)
SectorIndex	Index of the NOR physical sector to be checked.

Return value

≠ 0 NOR physical sector is blank.
= 0 NOR physical sector is not blank.

Additional information

This function is a member of the NOR physical layer API. The implementation of this function is optional. By default the NOR driver checks if a physical sector is blank by reading the entire contents of the physical sector and by checking that all bits are set to 1. If the hardware provides a faster method to check if a physical sector is blank then this function can implemented to use this method for increasing the write performance.

6.4.4.3.11 FS_NOR_PHY_TYPE_INIT

Description

Initializes the instance of the NOR physical layer.

Type definition

```
typedef int FS_NOR_PHY_TYPE_INIT(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the physical layer instance (0-based)

Return value

= 0 OK, physical layer initialized.
≠ 0 An error occurred.

Additional information

This function is a member of the NOR physical layer API. The implementation of this function is optional. If implemented, FS_NOR_PHY_TYPE_INIT is called by the NOR driver before the first access to the NOR flash device to identify the device, read the device parameters, etc.

6.4.4.4 Resource usage

This section describes the ROM and RAM usage of the NOR physical layers.

6.4.4.4.1 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 was measured using the SEGGER Embedded Studio IDE V4.20 configured to generate code for a Cortex-M4 CPU in Thumb mode and with the size optimization enabled. The following table lists the ROM usage of all the available NOR physical layers.

Name	ROM usage(Kbytes)
CFI 1x16	**2.1**
CFI 2x16	**1.5**
DSPI	**0.5**
SFDP	**1.7**
SPIFI	**1.6**
ST M25	**1.1**

6.4.4.4.2 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. The next table lists the static RAM usage of all the available NOR physical layers.

Name	RAM usage (bytes)
CFI 1x16	**20**
CFI 2x16	**20**
DSPI	**20**
SFDP	**20**
SPIFI	**20**
ST M25	**20**

6.4.4.4.3 Dynamic RAM usage

Dynamic RAM usage is the amount of RAM allocated by the physical layer at runtime. The amount of RAM required depends on the compile time and runtime configuration. The following table lists the the dynamic RAM usage of the available NOR physical layers.

Name	RAM usage (bytes)
CFI 1x16	**64**
CFI 2x16	**64**
DSPI	**0**
SFDP	**116**
SPIFI	**120**
ST M25	**38**

6.4.5 NOR hardware layer

6.4.5.1 General information

The NOR hardware layer provides functionality for accessing a NOR flash device via the target hardware such as external memory controller, GPIO, SPI, etc. The functions of the NOR hardware layer are called by the NOR physical layer to exchange commands and data with a NOR flash device. Since these functions are hardware dependent, they have to be implemented by the user. emFile comes with template hardware layers and sample implementations for popular evaluation boards that can be used as starting point for implementing new hardware layers. The relevant files are located in the `/Sample/FS/Driver/NOR` folder of the emFile shipment.

6.4.5.2 Hardware layer types

The functions of the NOR hardware layer are organized in a function table implemented a C structure. Different hardware layer types are provided to support different ways of interfacing a NOR flash device. The type of hardware layer an application has to use depends on the type NOR physical layer configured. The following table shows what hardware layer is required by each physical layer.

NOR hardware layer	NOR physical layer

`FS_NOR_HW_TYPE_SPI`	`FS_NOR_PHY_DSPI`
	`FS_NOR_PHY_SFDP`
	`FS_NOR_PHY_ST_M25`

`FS_NOR_HW_TYPE_SPIFI`	`FS_NOR_PHY_SPIFI`

6.4.5.3 Hardware layer API - FS_NOR_HW_TYPE_SPI

This hardware layer supports serial NOR flash devices that are interfaced via SPI. That is this hardware layer is able to transfer the data via one, two or four data lines. The functions of this hardware layer are grouped in a structure of type `FS_NOR_HW_TYPE_SPI`. The following sections describe these functions in detail.

6.4.5.3.1 FS_NOR_HW_TYPE_SPI

Description

Hardware layer for serial NOR flash devices.

Type definition

```
typedef struct {
  FS_NOR_HW_TYPE_SPI_INIT       * pfInit;
  FS_NOR_HW_TYPE_SPI_ENABLE_CS  * pfEnableCS;
  FS_NOR_HW_TYPE_SPI_DISABLE_CS * pfDisableCS;
  FS_NOR_HW_TYPE_SPI_READ       * pfRead;
  FS_NOR_HW_TYPE_SPI_WRITE      * pfWrite;
  FS_NOR_HW_TYPE_SPI_READ_X2    * pfRead_x2;
  FS_NOR_HW_TYPE_SPI_WRITE_X2   * pfWrite_x2;
  FS_NOR_HW_TYPE_SPI_READ_X4    * pfRead_x4;
  FS_NOR_HW_TYPE_SPI_WRITE_X4   * pfWrite_x4;
  FS_NOR_HW_TYPE_SPI_DELAY      * pfDelay;
  FS_NOR_HW_TYPE_SPI_LOCK       * pfLock;
  FS_NOR_HW_TYPE_SPI_UNLOCK     * pfUnlock;
} FS_NOR_HW_TYPE_SPI;
```

Structure members

Member	Description
pfInit	Initializes the SPI hardware.
pfEnableCS	Enables the serial NOR flash device.
pfDisableCS	Disables the serial NOR flash device.
pfRead	Transfers data from serial NOR flash device to MCU via 1 data line.
pfWrite	Transfers data from MCU to serial NOR flash device via 1 data line.
pfRead_x2	Transfers data from serial NOR flash device to MCU via 2 data lines.
pfWrite_x2	Transfers data from MCU to serial NOR flash device via 2 data lines.
pfRead_x4	Transfers data from serial NOR flash device to MCU via 4 data lines.
pfWrite_x4	Transfers data from MCU to serial NOR flash device via 4 data lines.
pfDelay	Blocks the execution for the specified time.
pfLock	Requests exclusive access to SPI bus.
pfUnlock	Releases the exclusive access to SPI bus.

6.4.5.3.2 FS_NOR_HW_TYPE_SPI_INIT

Description

Initializes the SPI hardware.

Type definition

```
typedef int FS_NOR_HW_TYPE_SPI_INIT(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).

Return value

> 0 OK, frequency of the SPI clock in kHz.
= 0 An error occurred.

Additional information

This function is a member of the SPI NOR hardware layer API and it has to be implemented by any hardware layer.

This function is called by the NOR physical layer once and before any other function of the hardware layer each time the file system mounts the serial NOR flash.

6.4.5.3.3 FS_NOR_HW_TYPE_SPI_ENABLE_CS

Description

Enables the serial NOR flash device.

Type definition

```
typedef void FS_NOR_HW_TYPE_SPI_ENABLE_CS(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).

Additional information

This function is a member of the SPI NOR hardware layer API and it has to be implemented by any hardware layer.

The Chip Select (CS) signal is used to address a specific serial NOR flash device connected via SPI. Enabling is equal to setting the CS signal to a logic low level.

6.4.5.3.4 FS_NOR_HW_TYPE_SPI_DISABLE_CS

Description

Disables the serial NOR flash device.

Type definition

```
typedef void FS_NOR_HW_TYPE_SPI_DISABLE_CS(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).

Additional information

This function is a member of the SPI NOR hardware layer API and it has to be implemented by any hardware layer.

The Chip Select (CS) signal is used to address a specific serial NOR flash device connected via SPI. Disabling is equal to setting the CS signal to a logic high level.

6.4.5.3.5 FS_NOR_HW_TYPE_SPI_READ

Description

Transfers data from serial NOR flash device to MCU via 1 data line.

Type definition

```
typedef void FS_NOR_HW_TYPE_SPI_READ(U8   Unit,
                                     U8 * pData,
                                     int  NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).
pData	out Data transfered from the serial NOR flash device.
NumBytes	Number of bytes to be transferred.

Additional information

This function is a member of the SPI NOR hardware layer API and it has to be implemented by any hardware layer.

6.4.5.3.6 FS_NOR_HW_TYPE_SPI_WRITE

Description

Transfers data from MCU to serial NOR flash device via 1 data line.

Type definition

```
typedef void FS_NOR_HW_TYPE_SPI_WRITE(      U8   Unit,
                                      const U8 * pData,
                                            int  NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).
pData	in Data transfered to the serial NOR flash device.
NumBytes	Number of bytes to be transferred.

Additional information

This function is a member of the SPI NOR hardware layer API and it has to be implemented by any hardware layer.

6.4.5.3.7 FS_NOR_HW_TYPE_SPI_READ_X2

Description

Transfers data from serial NOR flash device to MCU via 2 data lines.

Type definition

```
typedef void FS_NOR_HW_TYPE_SPI_READ_X2(U8   Unit,
                                        U8 * pData,
                                        int  NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).
pData	out Data transfered from the serial NOR flash device.
NumBytes	Number of bytes to be transferred.

Additional information

This function is a member of the SPI NOR hardware layer API. The implementation of this function is optional.

FS_NOR_HW_TYPE_SPI_READ_X2 transfers 2 bits of data on each clock period. Typically, the data is transferred via the DataOut (DO) and DataIn (DI) signals of the serial NOR flash where the even numbered bits of a byte are sent via the DI signal while the odd-numbered bits via the DO signal.

FS_NOR_HW_TYPE_SPI_READ_X2 is used only by the SFDP NOR physical layer.

6.4.5.3.8 FS_NOR_HW_TYPE_SPI_WRITE_X2

Description

Transfers data from MCU to serial NOR flash device via 2 data lines.

Type definition

```
typedef void FS_NOR_HW_TYPE_SPI_WRITE_X2(      U8   Unit,
                                         const U8 * pData,
                                               int  NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).
pData	in Data transfered to the serial NOR flash device.
NumBytes	Number of bytes to be transferred.

Additional information

This function is a member of the SPI NOR hardware layer API. The implementation of this function is optional.

FS_NOR_HW_TYPE_SPI_WRITE_X2 transfers 2 bits of data on each clock period. Typically, the data is transferred via the DataOut (DO) and DataIn (DI) signals of the serial NOR flash where the even numbered bits of a byte are sent via the DI signal while the odd-numbered bits via the DO signal.

FS_NOR_HW_TYPE_SPI_WRITE_X2 is used only by the SFDP NOR physical layer.

6.4.5.3.9 FS_NOR_HW_TYPE_SPI_READ_X4

Description

Transfers data from serial NOR flash device to MCU via 4 data lines.

Type definition

```
typedef void FS_NOR_HW_TYPE_SPI_READ_X4(U8   Unit,
                                        U8 * pData,
                                        int  NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).
pData	out Data transfered from the serial NOR flash device.
NumBytes	Number of bytes to be transferred.

Additional information

This function is a member of the SPI NOR hardware layer API. The implementation of this function is optional.

FS_NOR_HW_TYPE_SPI_READ_X4 exchanges 4 bits of data on each clock period. Typically, the data is transferred via the DataOut (DO), DataIn (DO), Write Protect (WP), and Hold (H) signals of the serial NOR flash device. A byte is transferred as follows:

- bits 0 and 4 via the DI signal
- bits 1 and 5 via the DO signal
- bits 2 and 6 via the WP signal
- bits 3 and 7 via the H signal.

FS_NOR_HW_TYPE_SPI_READ_X4 is used only by the SFDP NOR physical layer.

6.4.5.3.10 FS_NOR_HW_TYPE_SPI_WRITE_X4

Description

Transfers data from MCU to serial NOR flash device via 4 data lines.

Type definition

```
typedef void FS_NOR_HW_TYPE_SPI_WRITE_X4(      U8   Unit,
                                         const U8 * pData,
                                               int  NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).
pData	in Data transfered to the serial NOR flash device.
NumBytes	Number of bytes to be transferred.

Additional information

This function is a member of the SPI NOR hardware layer API. The implementation of this function is optional.

FS_NOR_HW_TYPE_SPI_WRITE_X4 exchanges 4 bits of data on each clock period. Typically, the data is transferred via the DataOut (DO), DataIn (DO), Write Protect (WP), and Hold (H) signals of the serial NOR flash device. A byte is transferred as follows:

- bits 0 and 4 via the DI signal
- bits 1 and 5 via the DO signal
- bits 2 and 6 via the WP signal
- bits 3 and 7 via the H signal.

FS_NOR_HW_TYPE_SPI_WRITE_X4 is used only by the SFDP NOR physical layer.

6.4.5.3.11 FS_NOR_HW_TYPE_SPI_DELAY

Description

Blocks the execution for the specified time.

Type definition

```
typedef int FS_NOR_HW_TYPE_SPI_DELAY(U8  Unit,
                                     U32 ms);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).
ms	Number of milliseconds to block the execution.

Return value

= 0 OK, delay executed.
< 0 Functionality not supported.

Additional information

This function is a member of the SPI NOR hardware layer API. The implementation of this function is optional. FS_NOR_HW_TYPE_SPI_DELAY can block the execution for longer than the number of milliseconds specified but not less than that. Typically, the function is implemented using a delay function of the used RTOS if any. If the hardware layer chooses not to implement FS_NOR_HW_TYPE_SPI_DELAY then the pfDelay member of FS_NOR_HW_TYPE_SPI can be set to NULL or to the address of a function that returns a negative value. If the function is not implemented by the hardware layer then the NOR physical layer blocks the execution by using a software loop. The number of software loops is calculated based in the SPI clock frequency returned by FS_NOR_HW_TYPE_SPI_INIT.

6.4.5.3.12 FS_NOR_HW_TYPE_SPI_LOCK

Description

Requests exclusive access to SPI bus.

Type definition

```
typedef void FS_NOR_HW_TYPE_SPI_LOCK(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).

Additional information

This function is a member of the SPI NOR hardware layer API. The implementation of this function is optional.

The NOR physical layer calls this function to indicate that it needs exclusive to access the NOR flash device via the SPI bus. It is guaranteed that the NOR physical layer does not attempt to exchange any data with the serial NOR flash device via the SPI bus before calling this function first. It is also guaranteed that FS_NOR_HW_TYPE_SPI_LOCK and FS_NOR_HW_TYPE_SPI_UNLOCK are called in pairs. Typically, this function is used for synchronizing the access to SPI bus when the SPI bus is shared between the serial NOR flash and other SPI devices.

A possible implementation would make use of an OS semaphore that is acquired in FS_NOR_HW_TYPE_SPI_LOCK and released in FS_NOR_HW_TYPE_SPI_UNLOCK.

6.4.5.3.13 FS_NOR_HW_TYPE_SPI_UNLOCK

Description

Requests exclusive access to SPI bus.

Type definition

```
typedef void FS_NOR_HW_TYPE_SPI_UNLOCK(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).

Additional information

This function is a member of the SPI NOR hardware layer API. The implementation of this function is optional.

The NOR physical layer calls this function after it no longer needs to access the serial NOR flash device via the SPI bus. It is guaranteed that the NOR physical layer does not attempt to exchange any data with the serial NOR flash device via the SPI bus before calling FS_NOR_HW_TYPE_SPI_LOCK. It is also guaranteed that FS_NOR_HW_TYPE_SPI_UNLOCK and FS_NOR_HW_TYPE_SPI_LOCK are called in pairs.

FS_NOR_HW_TYPE_SPI_UNLOCK and FS_NOR_HW_TYPE_SPI_LOCK can be used to synchronize the access to the SPI bus when other devices than the serial NAND flash are connected to it. A possible implementation would make use of an OS semaphore that is acquired FS_NOR_HW_TYPE_SPI_LOCK and released in FS_NOR_HW_TYPE_SPI_UNLOCK.

6.4.5.3.14 Sample implementation

The following sample implementation uses the SPI controller of an NXP K66 MCU to interface with a serial NOR flash device. This NOR hardware layer was tested on the SGGER emPower board (https://www.segger.com/evaluate-our-software/segger/empower/)

```
/*********************************************************************
*                  (c) SEGGER Microcontroller GmbH                   *
*                       The Embedded Experts                         *
*                          www.segger.com                            *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File    : FS_NOR_HW_SPI_K66_SEGGER_emPower.c
Purpose : NOR SPI hardware layer for the SEGGER emPower board
Literature:
  [1] K66 Sub-Family Reference Manual
      (\\FILESERVER\Techinfo\Company\Freescale\MCU\Kinetis_K-series
\K66P144M180SF5RMV2_Rev2_1505.pdf)
  [2] emPower Evaluation and prototyping platform for SEGGER software User Guide & Reference
 Manual
      (\\FILESERVER\Product\Doc4Review\UM06001_emPower.pdf)
*/

/*********************************************************************
*
*       #include Section
*
**********************************************************************
*/
#include <stddef.h>
#include "FS.h"
#include "FS_Int.h"

/*********************************************************************
*
*       Defines, configurable
*
**********************************************************************
*/
#ifndef   FS_NOR_HW_SPI_BUS_CLK_HZ
  #define FS_NOR_HW_SPI_BUS_CLK_HZ          (168000000uL / 3)   // SPI module clock
#endif

#ifndef   FS_NOR_HW_SPI_NOR_CLK_HZ
  #define FS_NOR_HW_SPI_NOR_CLK_HZ          28000000uL          // Clock of NOR flash device
#endif

#ifndef   FS_NOR_HW_SPI_INIT_WAIT_LOOPS
  #define FS_NOR_HW_SPI_INIT_WAIT_LOOPS     20000               // Waits about
 1ms for the SPI module to start/stop
#endif

#ifndef   FS_NOR_HW_SPI_UNIT_NO
  #define FS_NOR_HW_SPI_UNIT_NO             0
  // Selects the number of SPI unit to be used for data transfer (Permitted values: 0 and 2)
#endif

#ifndef   FS_NOR_HW_SPI_USE_OS
  #define FS_NOR_HW_SPI_USE_OS              0
  // Enables / disables the event-driven operation.
#endif

/*********************************************************************
*
*       #include section, conditional
*
**********************************************************************
*/
#if FS_NOR_HW_SPI_USE_OS
  #include "RTOS.h"
  #include "MK66F18.h"
#endif // FS_NOR_HW_SPI_USE_OS
```



```
/*********************************************************************
*
*       Defines, fixed
*
**********************************************************************
*/

/*********************************************************************
*
*       SPI module
*/
#if (FS_NOR_HW_SPI_UNIT_NO == 2)
  #define SPI_BASE_ADDR     0x400AC000uL
#else
  #define SPI_BASE_ADDR     0x4002C000uL
#endif
#define SPI_MCR             (*(volatile U32 *)(SPI_BASE_ADDR + 0x00))
  // Module Configuration Register
#define SPI_TCR             (*(volatile U32 *)(SPI_BASE_ADDR + 0x08))
  // Transfer Count Register
#define SPI_CTAR0           (*(volatile U32 *)(SPI_BASE_ADDR + 0x0C))
  // Clock and Transfer Attributes Register (In Master Mode)
#define SPI_CTAR1           (*(volatile U32 *)(SPI_BASE_ADDR + 0x10))
  // Clock and Transfer Attributes Register (In Master Mode)
#define SPI_SR              (*(volatile U32 *)(SPI_BASE_ADDR + 0x2C))     // Status Register
#define SPI_RSER            (*(volatile U32 *)(SPI_BASE_ADDR + 0x30))
  // DMA/Interrupt Request Select and Enable Register
#define SPI_PUSHR           (*(volatile U32 *)(SPI_BASE_ADDR + 0x34))
  // PUSH TX FIFO Register In Master Mode
#define SPI_POPR            (*(volatile U32 *)(SPI_BASE_ADDR + 0x38))
  // POP RX FIFO Register
#define SPI_TXFR0           (*(volatile U32 *)(SPI_BASE_ADDR + 0x3C))
  // Transmit FIFO Registers
#define SPI_TXFR1           (*(volatile U32 *)(SPI_BASE_ADDR + 0x40))
  // Transmit FIFO Registers
#define SPI_TXFR2           (*(volatile U32 *)(SPI_BASE_ADDR + 0x44))
  // Transmit FIFO Registers
#define SPI_TXFR3           (*(volatile U32 *)(SPI_BASE_ADDR + 0x48))
  // Transmit FIFO Registers
#define SPI_RXFR0           (*(volatile U32 *)(SPI_BASE_ADDR + 0x7C))
  // Receive FIFO Registers
#define SPI_RXFR1           (*(volatile U32 *)(SPI_BASE_ADDR + 0x80))
  // Receive FIFO Registers
#define SPI_RXFR2           (*(volatile U32 *)(SPI_BASE_ADDR + 0x84))
  // Receive FIFO Registers
#define SPI_RXFR3           (*(volatile U32 *)(SPI_BASE_ADDR + 0x88))
  // Receive FIFO Registers

#if (FS_NOR_HW_SPI_USE_OS == 0)

/*********************************************************************
*
*       System integration module
*/
#define SIM_BASE_ADDR       0x40047000uL
#define SIM_SCGC3           (*(volatile U32 *)(SIM_BASE_ADDR + 0x1030))
  // System Clock Gating Control Register 3
#define SIM_SCGC5           (*(volatile U32 *)(SIM_BASE_ADDR + 0x1038))
  // System Clock Gating Control Register 5
#define SIM_SCGC6           (*(volatile U32 *)(SIM_BASE_ADDR + 0x103C))
  // System Clock Gating Control Register 6

/*********************************************************************
*
*       Port B
*/
#define PORTB_BASE_ADDR     0x4004A000uL
#define PORTB_PCR20         (*(volatile U32 *)(PORTB_BASE_ADDR + 0x50))
  // Port Control Register
#define PORTB_PCR21         (*(volatile U32 *)(PORTB_BASE_ADDR + 0x54))
  // Port Control Register
#define PORTB_PCR22         (*(volatile U32 *)(PORTB_BASE_ADDR + 0x58))
  // Port Control Register
#define PORTB_PCR23         (*(volatile U32 *)(PORTB_BASE_ADDR + 0x5C))
  // Port Control Register
```



```
#define PORTB_DFER          (*(volatile U32 *)(PORTB_BASE_ADDR + 0xC0))
  // Digital filter enable register

/*********************************************************************
*
*       Port C
*/
#define PORTC_BASE_ADDR     0x4004B000uL
#define PORTC_PCR4          (*(volatile U32 *)(PORTC_BASE_ADDR + 0x10))
  // Port Control Register
#define PORTC_PCR5          (*(volatile U32 *)(PORTC_BASE_ADDR + 0x14))
  // Port Control Register
#define PORTC_PCR6          (*(volatile U32 *)(PORTC_BASE_ADDR + 0x18))
  // Port Control Register
#define PORTC_PCR7          (*(volatile U32 *)(PORTC_BASE_ADDR + 0x1C))
  // Port Control Register
#define PORTC_DFER          (*(volatile U32* )(PORTC_BASE_ADDR + 0xC0))
  // Digital filter enable register

/*********************************************************************
*
*       GPIO B
*/
#define GPIOB_BASE_ADDR     0x400FF040uL
#define GPIOB_PDOR          (*(volatile U32 *)(GPIOB_BASE_ADDR + 0x00))
  // Port Data Output Register
#define GPIOB_PSOR          (*(volatile U32 *)(GPIOB_BASE_ADDR + 0x04))
  // Port Set Output Register
#define GPIOB_PCOR          (*(volatile U32 *)(GPIOB_BASE_ADDR + 0x08))
  // Port Clear Output Register
#define GPIOB_PTOR          (*(volatile U32 *)(GPIOB_BASE_ADDR + 0x0C))
  // Port Toggle Output Register
#define GPIOB_PDIR          (*(volatile U32 *)(GPIOB_BASE_ADDR + 0x10))
  // Port Data Input Register
#define GPIOB_PDDR          (*(volatile U32 *)(GPIOB_BASE_ADDR + 0x14))
  // Port Data Direction Register

/*********************************************************************
*
*       GPIO C
*/
#define GPIOC_BASE_ADDR     0x400FF080uL
#define GPIOC_PDOR          (*(volatile U32 *)(GPIOC_BASE_ADDR + 0x00))
  // Port Data Output Register
#define GPIOC_PSOR          (*(volatile U32 *)(GPIOC_BASE_ADDR + 0x04))
  // Port Set Output Register
#define GPIOC_PCOR          (*(volatile U32 *)(GPIOC_BASE_ADDR + 0x08))
  // Port Clear Output Register
#define GPIOC_PTOR          (*(volatile U32 *)(GPIOC_BASE_ADDR + 0x0C))
  // Port Toggle Output Register
#define GPIOC_PDIR          (*(volatile U32 *)(GPIOC_BASE_ADDR + 0x10))
  // Port Data Input Register
#define GPIOC_PDDR          (*(volatile U32 *)(GPIOC_BASE_ADDR + 0x14))
  // Port Data Direction Register

#else

/*********************************************************************
*
*       Port C
*/
#define PORTC_BASE_ADDR     0x4004B000uL
#define PORTC_DFER          (*(volatile U32* )(PORTC_BASE_ADDR + 0xC0))
  // Digital filter enable register

#endif // FS_NOR_HW_SPI_USE_OS

/*********************************************************************
*
*       Port Control Register
*/
#define PCR_MUX_BIT         8     // Alternate port setting
#define PCR_MUX_GPIO        1uL   // Controlled directly by the HW layer
#define PCR_MUX_SPI0        2uL   // Controlled by SPI module
#define PCR_MUX_SPI2        2uL   // Controlled by SPI module
#define PCR_MUX_MASK        0x7uL
```



```
#define PCR_DSE_BIT         6     // Driver strength enable
#define PCR_PE_BIT          1     // Pull enable
#define PCR_PS_BIT          0     // Pull-up enable

/*********************************************************************
*
*       Clock enable bits
*/
#define SCGC3_SPI2_BIT      12  // SPI2
#define SCGC5_PORTB_BIT     10  // GPIO B
#define SCGC5_PORTC_BIT     11  // GPIO C
#define SCGC6_SPI0_BIT      12  // SPI0

/*********************************************************************
*
*       SPI Module Configuration Register
*/
#define MCR_MSTR_BIT        31
#define MCR_FRZ_BIT         27
#define MCR_CLR_TXF_BIT     11
#define MCR_CLR_RXF_BIT     10
#define MCR_HALT_BIT        0

/*********************************************************************
*
*       SPI Status Register
*/
#define SR_TCF_BIT          31
#define SR_TXRXS_BIT        30
#define SR_EOQF_BIT         28
#define SR_TFUF_BIT         27
#define SR_TFFF_BIT         25
#define SR_RFOF_BIT         19
#define SR_RFDF_BIT         17
#define SR_RXCTR_BIT        4
#define SR_RXCTR_MASK       0xFuL

/*********************************************************************
*
*       SPI Clock and Transfer Attributes Register
*/
#define CTAR_DBR_BIT        31
#define CTAR_FMSZ_BIT       27
#define CTAR_CPOL_BIT       26
#define CTAR_CPHA_BIT       25
#define CTAR_BR_BIT         0
#define CTAR_BR_MAX         0xFuL

/*********************************************************************
*
*       SPI misc. defines
*/
#define PUSHR_CTAS_BIT      28
#define RSER_RFDF_RE_BIT    17
#define SPI_IRQ_PRIO        15
#define WAIT_TIMEOUT_MS     1000

/*********************************************************************
*
*       NOR flash pin assignments
*/
#if (FS_NOR_HW_SPI_UNIT_NO == 2)
  #define NOR_CS_PIN        20  // Port B
  #define NOR_CLK_PIN       21  // Port B
  #define NOR_MOSI_PIN      22  // Port B
  #define NOR_MISO_PIN      23  // Port B
#else
  #define NOR_CS_PIN        4   // Port C
  #define NOR_CLK_PIN       5   // Port C
  #define NOR_MOSI_PIN      6   // Port C
  #define NOR_MISO_PIN      7   // Port C
#endif

/*********************************************************************
*
*       Static code
```



```
*
**********************************************************************
*/

/*********************************************************************
*
*       _Transfer8Bit
*/
static U8 _Transfer8Bit(U8 Data) {
  U32 NumBytes;

  SPI_PUSHR = Data;
  //
  // Wait to receive 1 byte.
  //
  while (1) {
    NumBytes = (SPI_SR >> SR_RXCTR_BIT) & SR_RXCTR_MASK;
    if (NumBytes == 1) {
      break;
    }
#if FS_NOR_HW_SPI_USE_OS
    {
      int r;

      SPI_RSER |= 1uL << RSER_RFDF_RE_BIT;     // Enable the interrupt.
      r = FS_X_OS_Wait(WAIT_TIMEOUT_MS);
      if (r != 0) {
        break;
      }
    }
#endif // FS_NOR_HW_SPI_USE_OS
  }
  Data = (U8)SPI_POPR;
  return Data;
}

#if (FS_NOR_HW_SPI_UNIT_NO == 0)

/*********************************************************************
*
*       _Read64Bits
*
*  Function description
*    Transfers 64 bits of data from NOR device to MCU.
*/
static void _Read64Bits(U8 * pData) {
  U16 Data16;
  U32 NumItems;
  U32 Dummy;

  //
  // Send dummy data.
  //
  Dummy = 0
        | 0xFFFFuL
        | (1uL << PUSHR_CTAS_BIT)
  // Use the second set of parameters which is configured for 16-bit transfers.
        ;
  SPI_PUSHR = Dummy;
  SPI_PUSHR = Dummy;
  SPI_PUSHR = Dummy;
  SPI_PUSHR = Dummy;
  //
  // Wait to receive 4 half-words (4 is the FIFO size)
  //
  while (1) {
    NumItems = (SPI_SR >> SR_RXCTR_BIT) & SR_RXCTR_MASK;
    if (NumItems == 4) {
      break;
    }
#if FS_NOR_HW_SPI_USE_OS
    {
      int r;

      SPI_RSER |= 1uL << RSER_RFDF_RE_BIT;     // Enable the interrupt.
      r = FS_X_OS_Wait(WAIT_TIMEOUT_MS);
```



```
      if (r != 0) {
        break;
      }
    }
#endif // FS_NOR_HW_SPI_USE_OS
  }
  //
  // Read data from FIFO.
  //
  Data16 = (U16)SPI_POPR;
  FS_StoreU16BE(pData, Data16);
  Data16 = (U16)SPI_POPR;
  FS_StoreU16BE(pData + 2, Data16);
  Data16 = (U16)SPI_POPR;
  FS_StoreU16BE(pData + 4, Data16);
  Data16 = (U16)SPI_POPR;
  FS_StoreU16BE(pData + 6, Data16);
}

/*********************************************************************
*
*       _Write64Bits
*
*  Function description
*    Transfers 64 bits of data from MCU to NOR device.
*/
static void _Write64Bits(const U8 * pData) {
  U16 Data16;
  U32 NumItems;

  //
  // Write data to NOR device. Use the second set of parameters which is configured for
 16-bit transfers.
  //
  Data16 = FS_LoadU16BE(pData);
  SPI_PUSHR = (U32)Data16 | (1uL << PUSHR_CTAS_BIT);
  Data16 = FS_LoadU16BE(pData + 2);
  SPI_PUSHR = (U32)Data16 | (1uL << PUSHR_CTAS_BIT);
  Data16 = FS_LoadU16BE(pData + 4);
  SPI_PUSHR = (U32)Data16 | (1uL << PUSHR_CTAS_BIT);
  Data16 = FS_LoadU16BE(pData + 6);
  SPI_PUSHR = (U32)Data16 | (1uL << PUSHR_CTAS_BIT);
  //
  // Wait to receive 4 half-words (4 is the FIFO size)
  //
  while (1) {
    NumItems = (SPI_SR >> SR_RXCTR_BIT) & SR_RXCTR_MASK;
    if (NumItems == 4) {
      break;
    }
#if FS_NOR_HW_SPI_USE_OS
    {
      int r;

      SPI_RSER |= 1uL << RSER_RFDF_RE_BIT;     // Enable the interrupt.
      r = FS_X_OS_Wait(WAIT_TIMEOUT_MS);
      if (r != 0) {
        break;
      }
    }
#endif // FS_NOR_HW_SPI_USE_OS
  }
  //
  // Discard the received data.
  //
  SPI_MCR |= 1uL << MCR_CLR_RXF_BIT;
}

#endif // FS_NOR_HW_SPI_UNIT_NO == 0

/*********************************************************************
*
*       Public code (via callback)
*
**********************************************************************
*/
```



```
/*********************************************************************
*
*       _HW_Init
*
*  Function description
*    Initialize the SPI for use with the NOR flash.
*
*  Parameters
*    Unit   Device index
*
*  Return value
*    SPI frequency that is set - given in kHz, 0 in case of an error.
*/
static int _HW_Init(U8 Unit) {
  int      Freq_Hz;
  unsigned NumLoops;
  unsigned Scaler;
  unsigned Div;

  FS_USE_PARA(Unit);
#if (FS_NOR_HW_SPI_UNIT_NO == 2)
  //
  // Enable the clock of GPIOs and SPI0.
  //
  SIM_SCGC3 |= 1uL << SCGC3_SPI2_BIT;
  SIM_SCGC5 |= 1uL << SCGC5_PORTB_BIT;
  //
  // Clock line (controlled by the SPI module)
  //
  PORTB_PCR21 = 0
              | (PCR_MUX_SPI2 << PCR_MUX_BIT)
              | (1uL          << PCR_PE_BIT)
              | (1uL          << PCR_PS_BIT)
              | (1uL          << PCR_DSE_BIT)
              ;
  //
  // MOSI line (controlled by the SPI module)
  //
  PORTB_PCR22 = 0
              | (PCR_MUX_SPI2 << PCR_MUX_BIT)
              | (1uL          << PCR_PE_BIT)
              | (1uL          << PCR_PS_BIT)
              | (1uL          << PCR_DSE_BIT)
              ;
  //
  // MISO line (controlled by the SPI module)
  //
  PORTB_PCR23 = 0
              | (PCR_MUX_SPI2 << PCR_MUX_BIT)
              | (1uL          << PCR_PE_BIT)
              | (1uL          << PCR_PS_BIT)
              | (1uL          << PCR_DSE_BIT)
              ;
  //
  // Chip select (controlled by the HW layer)
  //
  PORTB_PCR20 = 0
              | (PCR_MUX_GPIO << PCR_MUX_BIT)
              | (1uL          << PCR_PE_BIT)
              | (1uL          << PCR_PS_BIT)
              ;
  GPIOB_PSOR  = 1uL << NOR_CS_PIN;         // Active low
  GPIOB_PDDR |= 1uL << NOR_CS_PIN;
  //
  // Disable the digital filtering on all port pins controlled by the SPI module.
  //
  PORTB_DFER &= ~((1uL << NOR_CLK_PIN)  |
                  (1uL << NOR_MISO_PIN) |
                  (1uL << NOR_MOSI_PIN));
#else
  //
  // Enable the clock of GPIOs and SPI0.
  //
  SIM_SCGC5 |= 1uL << SCGC5_PORTC_BIT;
  SIM_SCGC6 |= 1uL << SCGC6_SPI0_BIT;
```



```
  //
  // Clock line (controlled by the SPI module)
  //
  PORTC_PCR5 = 0
             | (PCR_MUX_SPI0 << PCR_MUX_BIT)
             | (1uL          << PCR_PE_BIT)
             | (1uL          << PCR_PS_BIT)
             | (1uL          << PCR_DSE_BIT)
             ;
  //
  // MOSI line (controlled by the SPI module)
  //
  PORTC_PCR6 = 0
             | (PCR_MUX_SPI0 << PCR_MUX_BIT)
             | (1uL          << PCR_PE_BIT)
             | (1uL          << PCR_PS_BIT)
             | (1uL          << PCR_DSE_BIT)
             ;
  //
  // MISO line (controlled by the SPI module)
  //
  PORTC_PCR7 = 0
             | (PCR_MUX_SPI0 << PCR_MUX_BIT)
             | (1uL          << PCR_PE_BIT)
             | (1uL          << PCR_PS_BIT)
             | (1uL          << PCR_DSE_BIT)
             ;
  //
  // Chip select (controlled by the HW layer)
  //
  PORTC_PCR4 = 0
             | (PCR_MUX_GPIO << PCR_MUX_BIT)
             | (1uL          << PCR_PE_BIT)
             | (1uL          << PCR_PS_BIT)
             ;
  GPIOC_PSOR  = 1uL << NOR_CS_PIN;         // Active low
  GPIOC_PDDR |= 1uL << NOR_CS_PIN;
  //
  // Disable the digital filtering on all port pins controlled by the SPI module.
  //
  PORTC_DFER &= ~((1uL << NOR_CLK_PIN)  |
                  (1uL << NOR_MISO_PIN) |
                  (1uL << NOR_MOSI_PIN));
#endif // FS_NOR_HW_SPI_UNIT_NO == 2
  //
  // Disable the interrupt and configure the priority.
  //
#if FS_NOR_HW_SPI_USE_OS
  NVIC_DisableIRQ(SPI0_IRQn);
  NVIC_SetPriority(SPI0_IRQn, SPI_IRQ_PRIO);
#endif // FS_NOR_HW_SPI_USE_OS
  //
  // Calculate the clock scaler based on the configured clock frequency.
  //
  Scaler  = 0;
  Div     = 2;
  Freq_Hz = FS_NOR_HW_SPI_BUS_CLK_HZ;
  while (1) {
    if (((unsigned)Freq_Hz / Div) <= FS_NOR_HW_SPI_NOR_CLK_HZ) {
      break;
    }
    //
    // We do this here since the 3rd divisor is not power of 2.
    //
    if (Div == 2) {
      Div = 4;
    } else if (Div == 4) {
      Div = 6;
    } else if (Div == 6) {
      Div = 8;
    } else {
      Div <<= 1;
    }
    ++Scaler;
    if (Scaler == CTAR_BR_MAX) {
      break;
```



```
    }
  }
  //
  // Stop the SPI module.
  //
  SPI_MCR |= 1uL << MCR_HALT_BIT;
  NumLoops = FS_NOR_HW_SPI_INIT_WAIT_LOOPS;
  do {
    if ((SPI_SR & (1uL << SR_TXRXS_BIT)) == 0) {
      break;
    }
  } while (--NumLoops);
  //
  // Configure the SPI module.
  //
  SPI_MCR   = 0
            | (1uL << MCR_MSTR_BIT)
            | (1uL << MCR_CLR_TXF_BIT)
            | (1uL << MCR_CLR_RXF_BIT)
            | (1uL << MCR_HALT_BIT)
            ;
  SPI_TCR   = 0;
  SPI_CTAR0 = 0
            | (1uL    << CTAR_DBR_BIT)
            | (7uL    << CTAR_FMSZ_BIT)     // 8 bits
            | (1uL    << CTAR_CPOL_BIT)
            | (1uL    << CTAR_CPHA_BIT)
            | (Scaler << CTAR_BR_BIT)
            ;
  SPI_CTAR1 = 0
            | (1uL    << CTAR_DBR_BIT)
            | (15uL   << CTAR_FMSZ_BIT)     // 16 bits
            | (1uL    << CTAR_CPOL_BIT)
            | (1uL    << CTAR_CPHA_BIT)
            | (Scaler << CTAR_BR_BIT)
            ;
  SPI_SR    = 0
            | (1uL << SR_TCF_BIT)
            | (1uL << SR_EOQF_BIT)
            | (1uL << SR_TFUF_BIT)
            | (1uL << SR_TFFF_BIT)
            | (1uL << SR_RFOF_BIT)
            | (1uL << SR_RFDF_BIT)
            ;
  //
  // Start the SPI module.
  //
  SPI_MCR &= ~(1uL << MCR_HALT_BIT);
  NumLoops = FS_NOR_HW_SPI_INIT_WAIT_LOOPS;
  do {
    if (SPI_SR & (1uL << SR_TXRXS_BIT)) {
      break;
    }
  } while (--NumLoops);
  //
  // Make a dummy transfer so that the polarity of the clock signal is set correctly.
  // Without this, the first byte transfer will send/receive the wrong value.
  //
  _Transfer8Bit(0xFF);
#if FS_NOR_HW_SPI_USE_OS
  NVIC_EnableIRQ(SPI0_IRQn);
#else
  SPI_RSER  = 0;
#endif
  return Freq_Hz;
}

/*********************************************************************
*
*       _HW_EnableCS
*
*  Function description
*    Activates chip select signal (CS) of the NOR flash chip.
*
*  Parameters
*    Unit   Device index
```



```
*/
static void _HW_EnableCS(U8 Unit) {
  FS_USE_PARA(Unit);
#if (FS_NOR_HW_SPI_UNIT_NO == 2)
  GPIOB_PCOR = 1uL << NOR_CS_PIN;         // Active low
#else
  GPIOC_PCOR = 1uL << NOR_CS_PIN;         // Active low
#endif
}

/*********************************************************************
*
*       _HW_DisableCS
*
*  Function description
*    Deactivates chip select signal (CS) of the NOR flash chip.
*
*  Parameters
*    Unit   Device index
*/
static void _HW_DisableCS(U8 Unit) {
  FS_USE_PARA(Unit);
#if (FS_NOR_HW_SPI_UNIT_NO == 2)
  GPIOB_PSOR = 1uL << NOR_CS_PIN;         // Active low
#else
  GPIOC_PSOR = 1uL << NOR_CS_PIN;         // Active low
#endif
}

/*********************************************************************
*
*       _HW_Read
*
*  Function description
*    Reads a specified number of bytes from NOR flash to buffer.
*
*  Parameters
*    Unit       Device index
*    pData      Pointer to a data buffer
*    NumBytes   Number of bytes to be read
*/
static void _HW_Read(U8 Unit, U8 * pData, int NumBytes) {
  FS_USE_PARA(Unit);
  if (NumBytes) {
#if (FS_NOR_HW_SPI_UNIT_NO == 2)
    do {
      *pData++ = _Transfer8Bit(0xFF);
    } while (--NumBytes);
#else
    //
    // Read 64 bits at a time if possible.
    //
    if (((unsigned)NumBytes & 7) == 0) {      // Multiple of 8 bytes?
      unsigned NumLoops;

      NumLoops = (unsigned)NumBytes >> 3;
      do {
        _Read64Bits(pData);
        pData += 8;
      } while (--NumLoops);
    } else {
      do {
        *pData++ = _Transfer8Bit(0xFF);
      } while (--NumBytes);
    }
#endif
  }
}

/*********************************************************************
*
*       _HW_Write
*
*  Function description
*    Writes a specified number of bytes from data buffer to NOR flash.
*
```



```
*  Parameters
*    Unit       Device index
*    pData      Pointer to a data buffer
*    NumBytes   Number of bytes to be written
*/
static void _HW_Write(U8 Unit, const U8 * pData, int NumBytes) {
  FS_USE_PARA(Unit);
  if (NumBytes) {
#if (FS_NOR_HW_SPI_UNIT_NO == 2)
    do {
      (void)_Transfer8Bit(*pData++);
    } while (--NumBytes);
#else
    //
    // Write 64 bits at a time if possible.
    //
    if (((unsigned)NumBytes & 7) == 0) {      // Multiple of 8 bytes?
      unsigned NumLoops;

      NumLoops = (unsigned)NumBytes >> 3;
      do {
        _Write64Bits(pData);
        pData += 8;
      } while (--NumLoops);
    } else {
      do {
        (void)_Transfer8Bit(*pData++);
      } while (--NumBytes);
    }
#endif
  }
}

/*********************************************************************
*
*      Public code
*
**********************************************************************
*/

#if FS_NOR_HW_SPI_USE_OS

/*********************************************************************
*
*       SPI0_IRQHandler
*/
void SPI0_IRQHandler(void);
void SPI0_IRQHandler(void) {
  OS_EnterInterrupt();
  SPI_RSER &= ~(1uL << RSER_RFDF_RE_BIT);       // Disable the interrupt.
  FS_X_OS_Signal();
  OS_LeaveInterrupt();
}

#endif // FS_NOR_HW_SPI_USE_OS

/*********************************************************************
*
*       Public data
*
**********************************************************************
*/
const FS_NOR_HW_TYPE_SPI FS_NOR_HW_SPI_K66_SEGGER_emPower = {
  _HW_Init,
  _HW_EnableCS,
  _HW_DisableCS,
  _HW_Read,
  _HW_Write,
  NULL,
  NULL,
  NULL,
  NULL,
  NULL,
  NULL,
  NULL
};
```



```
/************************** End of file ***************************/
```


6.4.5.4 Hardware layer API - FS_NOR_HW_TYPE_SPIFI

This hardware layer supports serial NOR flash devices that are interfaced via SPI with their data content being mapped to the system memory of the MCU. This type of interface translates read accesses to the system memory region where the serial NOR flash device is mapped to read operations via SPI bus. Therefore, the hardware layer can read the data from the serial NOR flash device by simply reading data from the system memory. Any other operations with the serial NOR flash device are performed by sending the commands directly via the SPI bus.

That is this hardware layer is able to transfer the data via one, two or four data lines. The functions of this hardware layer are grouped in a structure of type `FS_NOR_HW_TYPE_SPIFI`. The following sections describe these functions in detail.

6.4.5.4.1 FS_NOR_HW_TYPE_SPIFI

Description

Hardware layer for serial NOR flash devices that are memory-mapped.

Type definition

```
typedef struct {
  FS_NOR_HW_TYPE_SPIFI_INIT          * pfInit;
  FS_NOR_HW_TYPE_SPIFI_SET_CMD_MODE * pfSetCmdMode;
  FS_NOR_HW_TYPE_SPIFI_SET_MEM_MODE * pfSetMemMode;
  FS_NOR_HW_TYPE_SPIFI_EXEC_CMD      * pfExecCmd;
  FS_NOR_HW_TYPE_SPIFI_READ_DATA     * pfReadData;
  FS_NOR_HW_TYPE_SPIFI_WRITE_DATA    * pfWriteData;
  FS_NOR_HW_TYPE_SPIFI_POLL          * pfPoll;
  FS_NOR_HW_TYPE_SPIFI_DELAY         * pfDelay;
  FS_NOR_HW_TYPE_SPIFI_LOCK          * pfLock;
  FS_NOR_HW_TYPE_SPIFI_UNLOCK        * pfUnlock;
} FS_NOR_HW_TYPE_SPIFI;
```

Structure members

Member	Description
pfInit	Initializes the hardware.
pfSetCmdMode	Configures the hardware for direct access to serial NOR flash.
pfSetMemMode	Configures the hardware for access to serial NOR flash via system memory.
pfExecCmd	Sends a command to serial NOR flash that does not transfer any data.
pfReadData	Transfers data from serial NOR flash to MCU.
pfWriteData	Transfers data from MCU to serial NOR flash.
pfPoll	Checks periodically the value of a status flag.
pfDelay	Blocks the execution for the specified time.
pfLock	Requests exclusive access to SPI bus.
pfUnlock	Releases the exclusive access to SPI bus.

6.4.5.4.2 FS_NOR_HW_TYPE_SPIFI_INIT

Description

Initializes the SPI hardware.

Type definition

```
typedef int FS_NOR_HW_TYPE_SPIFI_INIT(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).

Return value

> 0 OK, frequency of the SPI clock in Hz.
= 0 An error occurred.

Additional information

This function is a member of the SPIFI NOR hardware layer API and it has to be implemented by any hardware layer.

This function is called by the NOR physical layer once and before any other function of the hardware layer each time the file system mounts the serial NOR flash. FS_NOR_HW_TYPE_SPIFI_INIT has to perform the initialization of clock signals, GPIO ports, SPI controller, etc.

6.4.5.4.3 FS_NOR_HW_TYPE_SPIFI_SET_CMD_MODE

Description

Configures the hardware for direct access to serial NOR flash.

Type definition

```
typedef void FS_NOR_HW_TYPE_SPIFI_SET_CMD_MODE(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).

Additional information

This function is a member of the SPIFI NOR hardware layer API and it has to be implemented by any hardware layer.

The physical layer calls this function before it starts sending commands directly to the serial NOR flash. Typically, the SPI hardware can operate in two modes: command and memory. In command mode the physical layer can access the serial NOR flash directly by sending commands to it. In the memory mode the SPI hardware translates read accesses to an assigned memory region to serial NOR flash read commands. In this way, the contents of the contents of the serial NOR flash is mapped into this memory region.

If the command and memory mode are mutually exclusive, `FS_NOR_HW_TYPE_SPIFI_SET_CMD_MODE` has to disable the memory mode.

6.4.5.4.4 FS_NOR_HW_TYPE_SPIFI_SET_MEM_MODE

Description

Configures the hardware for access to serial NOR flash via system memory.

Type definition

```
typedef void FS_NOR_HW_TYPE_SPIFI_SET_MEM_MODE(U8       Unit,
                                                U8       ReadCmd,
                                                unsigned NumBytesAddr,
                                                unsigned NumBytesDummy,
                                                U16      BusWidth);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).
ReadCmd	Code of the command to be used for reading the data.
NumBytesAddr	Number of address bytes to be used for the read command.
NumBytesDummy	Number of dummy bytes to be sent after the read command.
BusWidth	Number of data lines to be used for the data transfer.

Additional information

This function is a member of the SPIFI NOR hardware layer API. The implementation of this function is optional. If the hardware layer does not implement this function then the data is read from the serial NOR flash device using FS_NOR_HW_TYPE_SPIFI_READ_DATA.

This function is called by the physical layer when it no longer wants to send commands directly to the serial NOR flash device. After the call to this function the physical layer expects that the contents of the serial NOR flash device is available in the system memory region assigned to SPI hardware and that the data can be accessed by simple memory read operations.

If the command and memory mode are mutually exclusive, then FS_NOR_HW_TYPE_SPIFI_SET_MEM_MODE has to disable the command mode.

BusWidth encodes the number of data lines to be used when transferring the command, address and data. The value can be decoded using FS_BUSWIDTH_GET_CMD(), FS_BUSWIDTH_GET_ADDR() and FS_BUSWIDTH_GET_DATA(). The dummy bytes are sent using the same number of data lines as the address. For additional information refer to *SPI bus width decoding* on page 529

6.4.5.4.5 FS_NOR_HW_TYPE_SPIFI_EXEC_CMD

Description

Sends a command to serial NOR flash that does not transfer any data.

Type definition

```
typedef void FS_NOR_HW_TYPE_SPIFI_EXEC_CMD(U8 Unit,
                                            U8 Cmd,
                                            U8 BusWidth);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).
Cmd	Code of the command to be sent.
BusWidth	Number of data lines to be used for sending the command.

Additional information

This function is a member of the SPIFI NOR hardware layer API and it has to be implemented by any hardware layer.

Typically, the physical layer calls FS_NOR_HW_TYPE_SPIFI_EXEC_CMD to enable or disable the write mode in the serial NOR flash, to set the number of address bytes, etc. FS_NOR_HW_TYPE_SPIFI_EXEC_CMD is called by the physical layer only with the SPI hardware configured in command mode. FS_NOR_HW_TYPE_SPIFI_EXEC_CMD has to wait for the command to complete before it returns.

BusWidth is not encoded as is the case with the other functions of this physical layer but instead is stores the number of data lines. That is FS_BUSWIDTH_GET_CMD() is not required for decoding the value. Permitted values are 1, 2 and 4.

6.4.5.4.6 FS_NOR_HW_TYPE_SPIFI_READ_DATA

Description

Transfers data from serial NOR flash to MCU.

Type definition

```
typedef void FS_NOR_HW_TYPE_SPIFI_READ_DATA(      U8       Unit,
                                                  U8       Cmd,
                                            const U8     * pPara,
                                                  unsigned NumBytesPara,
                                                  unsigned NumBytesAddr,
                                                  U8     * pData,
                                                  unsigned NumBytesData,
                                                  U16      BusWidth);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).
Cmd	Code of the command to be sent.
pPara	in Command parameters (address and dummy bytes). Can be NULL.
NumBytesPara	Total number of address and dummy bytes to be sent. Can be 0.
NumBytesAddr	Number of address bytes to be send. Can be 0.
pData	out Data read from the serial NOR flash device. Can be NULL.
NumBytesData	Number of bytes to read from the serial NOR flash device. Can be 0.
BusWidth	Number of data lines to be used for the data transfer.

Additional information

This function is a member of the SPIFI NOR hardware layer API and it has to be implemented by any hardware layer.

This function is called with the SPI hardware in command mode. Typically, the physical layer calls this function when it wants to read parameters or status information from the serial NOR flash device. FS_NOR_HW_TYPE_SPIFI_READ_DATA has to wait for the data transfer to complete before it returns.

If FS_NOR_HW_TYPE_SPIFI_SET_MEM_MODE is provided then the physical layer reads the contents of the serial NOR flash device via reads accesses to the system memory. FS_NOR_HW_TYPE_SPIFI_READ_DATA is used by the physical layer to read the contents of the serial NOR flash device if FS_NOR_HW_TYPE_SPIFI_SET_MEM_MODE is not provided.

The first address byte is stored at *pPara. NumBytesAddr is always smaller than or equal to NumBytesPara. If NumBytesAddr is equal to NumBytesPara then no dummy bytes have to be sent. The number of dummy bytes to be sent can be calculated as NumBytesPara - NumBytesAddr. If the hardware is sending dummy cycles instead of bytes then the number of dummy bytes have to converted to clock cycles by taking into account the number of data lines used for the data transfer. The dummy bytes are sent via the same number of data lines as the address.

BusWidth encodes the number of data lines to be used when transferring the command, address and data. The value can be decoded using FS_BUSWIDTH_GET_CMD(), FS_BUSWIDTH_GET_ADDR() and FS_BUSWIDTH_GET_DATA(). The dummy bytes are sent using the same number of data lines as the address. For additional information refer to *SPI bus width decoding* on page 529

6.4.5.4.7 FS_NOR_HW_TYPE_SPIFI_WRITE_DATA

Description

Transfers data from MCU to serial NOR flash.

Type definition

```
typedef void FS_NOR_HW_TYPE_SPIFI_WRITE_DATA(      U8       Unit,
                                                   U8       Cmd,
                                             const U8     * pPara,
                                                   unsigned NumBytesPara,
                                                   unsigned NumBytesAddr,
                                             const U8     * pData,
                                                   unsigned NumBytesData,
                                                   U16      BusWidth);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).
Cmd	Code of the command to be sent.
pPara	in Command parameters (address and dummy bytes). Can be NULL.
NumBytesPara	Total number of address and dummy bytes to be sent. Can be 0.
NumBytesAddr	Number of address bytes to be send. Can be 0.
pData	in Data to be sent to the serial NOR flash device. Can be NULL.
NumBytesData	Number of bytes to be written the serial NOR flash device. Can be 0.
BusWidth	Number of data lines to be used for the data transfer.

Additional information

This function is a member of the SPIFI NOR hardware layer API and it has to be implemented by any hardware layer.

This function is called with the SPI hardware in command mode. Typically, the physical layer calls this function when it wants to modify the data in a page of the serial NOR flash device or when to erase a NOR physical sector. FS_NOR_HW_TYPE_SPIFI_WRITE_DATA has to wait for the data transfer to complete before it returns.

The first address byte is stored at *pPara. NumBytesAddr is always smaller than or equal to NumBytesPara. If NumBytesAddr is equal to NumBytesPara then no dummy bytes have to be sent. The number of dummy bytes to be sent can be calculated as NumBytesPara - NumBytesAddr. If the hardware is sending dummy cycles instead of bytes then the number of dummy bytes have to converted to clock cycles by taking into account the number of data lines used for the data transfer. The dummy bytes are sent via the same number of data lines as the address.

BusWidth encodes the number of data lines to be used when transferring the command, address and data. The value can be decoded using FS_BUSWIDTH_GET_CMD(), FS_BUSWIDTH_GET_ADDR() and FS_BUSWIDTH_GET_DATA(). The dummy bytes are sent using the same number of data lines as the address. For additional information refer to *SPI bus width decoding* on page 529

6.4.5.4.8 FS_NOR_HW_TYPE_SPIFI_POLL

Description

Checks periodically the value of a status flag.

Type definition

```
typedef int FS_NOR_HW_TYPE_SPIFI_POLL(U8  Unit,
                                      U8  Cmd,
                                      U8  BitPos,
                                      U8  BitValue,
                                      U32 Delay,
                                      U32 TimeOut_ms,
                                      U16 BusWidth);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).
Cmd	Code of the command to be sent.
BitPos	Position of the bit to be checked.
BitValue	Value of the bit to wait for.
Delay	Number of clock cycles to wait between two queries.
TimeOut_ms	Maximum number of milliseconds to wait for the bit to be set.
BusWidth	Number of data lines to be used for the data transfer.

Additional information

This function is a member of the SPIFI NOR hardware layer API. The implementation of this function is optional.

This function is called with the SPI hardware in command mode and has to send periodically a command and to read one byte from the serial NOR flash device. FS_NOR_HW_TYPE_SPIFI_POLL has to wait until the bit at BitPos in the response returned by the serial NOR flash device is set to the value specified by BitValue. A BitPos of 0 specifies the position of the least significant bit in the response.

BusWidth encodes the number of data lines to be used when transferring the command and data. The value can be decoded using FS_BUSWIDTH_GET_CMD(), and FS_BUSWIDTH_GET_DATA(). For additional information refer to *SPI bus width decoding* on page 529

6.4.5.4.9 FS_NOR_HW_TYPE_SPIFI_DELAY

Description

Blocks the execution for the specified time.

Type definition

```
typedef int FS_NOR_HW_TYPE_SPIFI_DELAY(U8  Unit,
                                       U32 ms);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).
ms	Number of milliseconds to block the execution.

Return value

= 0 OK, delay executed.
< 0 Functionality not supported.

Additional information

This function is a member of the SPIFI NOR hardware layer API. The implementation of this function is optional. FS_NOR_HW_TYPE_SPIFI_DELAY can block the execution for longer than the number of milliseconds specified but not less than that. Typically, the function is implemented using a delay function of the used RTOS if any. If the hardware layer chooses not to implement FS_NOR_HW_TYPE_SPIFI_DELAY then the pfDelay member of FS_NOR_HW_TYPE_SPIFI can be set to NULL or to the address of a function that returns a negative value. If the function is not implemented by the hardware layer then the NOR physical layer blocks the execution by using a software loop. The number of software loops is calculated based in the SPI clock frequency returned by FS_NOR_HW_TYPE_SPIFI_INIT.

6.4.5.4.10 FS_NOR_HW_TYPE_SPIFI_LOCK

Description

Requests exclusive access to SPI bus.

Type definition

```
typedef void FS_NOR_HW_TYPE_SPIFI_LOCK(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).

Additional information

This function is a member of the SPIFI NOR hardware layer API. The implementation of this function is optional.

The NOR physical layer calls this function to indicate that it needs exclusive to access the NOR flash device via the SPI bus. It is guaranteed that the NOR physical layer does not attempt to exchange any data with the serial NOR flash device via the SPI bus before calling this function first. It is also guaranteed that FS_NOR_HW_TYPE_SPIFI_LOCK and FS_NOR_HW_TYPE_SPIFI_UNLOCK are called in pairs. Typically, this function is used for synchronizing the access to SPI bus when the SPI bus is shared between the serial NOR flash and other SPI devices.

A possible implementation would make use of an OS semaphore that is acquired in FS_NOR_HW_TYPE_SPIFI_LOCK and released in FS_NOR_HW_TYPE_SPIFI_UNLOCK.

6.4.5.4.11 FS_NOR_HW_TYPE_SPIFI_UNLOCK

Description

Requests exclusive access to SPI bus.

Type definition

```
typedef void FS_NOR_HW_TYPE_SPIFI_UNLOCK(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer (0-based).

Additional information

This function is a member of the SPIFI NOR hardware layer API. The implementation of this function is optional.

The NOR physical layer calls this function when it no longer needs to access the serial NOR flash device via the SPI bus. It is guaranteed that the NOR physical layer does not attempt to exchange any data with the serial NOR flash device via the SPI bus before calling FS_NOR_HW_TYPE_SPIFI_LOCK. It is also guaranteed that FS_NOR_HW_TYPE_SPIFI_UNLOCK and FS_NOR_HW_TYPE_SPIFI_LOCK are called in pairs.

FS_NOR_HW_TYPE_SPIFI_UNLOCK and FS_NOR_HW_TYPE_SPIFI_LOCK can be used to synchronize the access to the SPI bus when other devices than the serial NAND flash are connected to it. A possible implementation would make use of an OS semaphore that is acquired FS_NOR_HW_TYPE_SPIFI_LOCK and released in FS_NOR_HW_TYPE_SPIFI_UNLOCK.

6.4.5.4.12 Sample implementation

The following sample implementation uses the SPIFI controller of an NXP LPC4322 MCU to interface with a serial NOR flash device. This NOR hardware layer has been tested on a SGGER internal test board.

```
/*********************************************************************
*                SEGGER MICROCONTROLLER GmbH & Co. KG                *
*        Solutions for real time microcontroller applications        *
**********************************************************************
*                                                                    *
*        (c) 2003-2007     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        : NOR_PHY_SPIFI_NXP_LPC4322_SEGGER_QSPIFI_Test_Board.c
Purpose     : Low-level flash driver for NXP SPIFI interface.
Literature  : [1] \\fileserver\Techinfo\Company\NXP\MCU\LPC43xx
\UserManual_LPC43xx_UM10503_Rev1.90_150218.pdf
------------  END-OF-HEADER  ---------------------------------------
*/

/*********************************************************************
*
*       #include section
*
**********************************************************************
*/
#include "FS_Int.h"

/*********************************************************************
*
*      Defines, configurable
*
**********************************************************************
*/
#define SPIFI_CLK_HZ          60000000uL
  // Frequency of the clock supplied to SPIFI unit.
#define SPIFI_DATA_ADDR       0x80000000
  // This is the start address of the memory region used by the file system

  // to read the data from the serial NOR flash device. The hardware layer

  // performs a dummy read from this address when switching to memory mode

  // in order to clean the caches. It should be set to the value passed

  // as second parameter to FS_NOR_Configure()/FS_NOR_BM_Configure() in FS_X_AddDevices().
#define RESET_DELAY_LOOPS     100000
  // Number of software loops to wait for NOR device to reset
#define USE_OS                0
  // Enables / disables the interrupt driver operation

/*********************************************************************
*
*       #include section, conditional
*
**********************************************************************
*/
#if USE_OS
  #include "RTOS.h"
  #include "lpc43xx.h"
#endif

/*********************************************************************
*
*       Defines, fixed
```



```
*
**********************************************************************
*/

/**********************************************************************
*
*       SPIFI unit
*/
#define SPIFI_BASE_ADDR       0x40003000
#define SPIFI_CTRL            (*(volatile U32*)(SPIFI_BASE_ADDR + 0x00))
  // SPIFI control register
#define SPIFI_CMD             (*(volatile U32*)(SPIFI_BASE_ADDR + 0x04))
  // SPIFI command register
#define SPIFI_ADDR            (*(volatile U32*)(SPIFI_BASE_ADDR + 0x08))
  // SPIFI address register
#define SPIFI_IDATA           (*(volatile U32*)(SPIFI_BASE_ADDR + 0x0C))
  // SPIFI intermediate data register
#define SPIFI_CLIMIT          (*(volatile U32*)(SPIFI_BASE_ADDR + 0x10))
  // SPIFI cache limit register
#define SPIFI_DATA32          (*(volatile U32*)(SPIFI_BASE_ADDR + 0x14))
  // SPIFI data register (32-bit access, 4 bytes at once)
#define SPIFI_DATA16          (*(volatile U16*)(SPIFI_BASE_ADDR + 0x14))
  // SPIFI data register (16-bit access, 2 bytes at once)
#define SPIFI_DATA8           (*(volatile U8*) (SPIFI_BASE_ADDR + 0x14))
  // SPIFI data register (8-bit access, 1 byte at once)
#define SPIFI_MCMD            (*(volatile U32*)(SPIFI_BASE_ADDR + 0x18))
  // SPIFI memory command register
#define SPIFI_STAT            (*(volatile U32*)(SPIFI_BASE_ADDR + 0x1C))
  // SPIFI status register

/**********************************************************************
*
*       Clock generation unit
*/
#define CGU_BASE_ADDR         0x40050000
#define IDIVA_CTRL            (*(volatile U32*)(CGU_BASE_ADDR + 0x0048))
#define IDIVB_CTRL            (*(volatile U32*)(CGU_BASE_ADDR + 0x004C))
#define BASE_SPIFI_CLK        (*(volatile U32*)(CGU_BASE_ADDR + 0x0070))

/**********************************************************************
*
*       Clock control unit 1
*/
#define CCU1_BASE_ADDR        0x40051000
#define CLK_SPIFI_CFG         (*(volatile U32*)(CCU1_BASE_ADDR + 0x0300))
#define CLK_SPIFI_STAT        (*(volatile U32*)(CCU1_BASE_ADDR + 0x0304))

/**********************************************************************
*
*       System control unit
*/
#define SCU_BASE_ADDR         0x40086000
#define SFSP3_2               (*(volatile U32*)(SCU_BASE_ADDR + 0x188))
  // Pin configuration register for pin P3_2
#define SFSP3_3               (*(volatile U32*)(SCU_BASE_ADDR + 0x18C))
  // Pin configuration register for pin P3_3
#define SFSP3_4               (*(volatile U32*)(SCU_BASE_ADDR + 0x190))
  // Pin configuration register for pin P3_4
#define SFSP3_5               (*(volatile U32*)(SCU_BASE_ADDR + 0x194))
  // Pin configuration register for pin P3_5
#define SFSP3_6               (*(volatile U32*)(SCU_BASE_ADDR + 0x198))
  // Pin configuration register for pin P3_6
#define SFSP3_7               (*(volatile U32*)(SCU_BASE_ADDR + 0x19C))
  // Pin configuration register for pin P3_7
#define SFSP3_8               (*(volatile U32*)(SCU_BASE_ADDR + 0x1A0))
  // Pin configuration register for pin P3_8

/**********************************************************************
*
*       GPIO unit
*/
#define GPIO_BASE_ADDR        0x400F4000
#define GPIO_DIR5             (*(volatile U32*)(GPIO_BASE_ADDR + 0x2014))
  // Direction registers port 5
#define GPIO_SET5             (*(volatile U32*)(GPIO_BASE_ADDR + 0x2214))
  // Set register for port 5
```



```
#define GPIO_CLR5             (*(volatile U32*)(GPIO_BASE_ADDR + 0x2294))
  // Clear register for port 5

/*********************************************************************
*
*       SPIFI command register
*/
#define CMD_BITPOS_BIT                0
#define CMD_BITPOS_MASK               0x7uL
#define CMD_BITVAL_BIT                3
#define CMD_BITVAL_MASK               0x1uL
#define CMD_DATALEN_BIT               0
#define CMD_DATALEN_MASK              0x3FFFuL
#define CMD_POLL_BIT                  14
#define CMD_DOUT_BIT                  15
#define CMD_INTLEN_BIT                16
#define CMD_INTLEN_MASK               0x7uL
#define CMD_FIELDFORM_BIT             19
#define CMD_FIELDFORM_ALL_SERIAL      0x0uL
#define CMD_FIELDFORM_QUAD_DUAL_DATA  0x1uL
#define CMD_FIELDFORM_SERIAL_OPCODE   0x2uL
#define CMD_FIELDFORM_ALL_QUAD_DUAL   0x3uL
#define CMD_FRAMEFORM_BIT             21
#define CMD_FRAMEFORM_OPCODE          0x1uL
#define CMD_FRAMEFORM_OPCODE_1BYTE    0x2uL
#define CMD_FRAMEFORM_OPCODE_2BYTES   0x3uL
#define CMD_FRAMEFORM_OPCODE_3BYTES   0x4uL
#define CMD_FRAMEFORM_OPCODE_4BYTES   0x5uL
#define CMD_OPCODE_BIT                24
#define CMD_OPCODE_MASK               0xFFuL

/*********************************************************************
*
*       SPIFI control register
*/
#define CTRL_TIMEOUT_BIT              0
#define CTRL_CSHIGH_BIT               16
#define CTRL_D_PRFTCH_DIS_BIT         21
#define CTRL_MODE3_BIT                23
#define CTRL_INTEN_BIT                22
#define CTRL_PRFTCH_DIS_BIT           27
#define CTRL_DUAL_BIT                 28
#define CTRL_FBCLK_BIT                30

/*********************************************************************
*
*       SPIFI status register
*/
#define STAT_MCINIT_BIT               0
#define STAT_CMD_BIT                  1
#define STAT_RESET_BIT                4
#define STAT_INTRQ_BIT                5

/*********************************************************************
*
*       Misc. defines
*/
#define CGU_IDIV_BIT                  2
#define CGU_AUTOBLOCK_BIT             11
#define CGU_CLK_SEL_BIT               24
#define SFS_MODE_BIT                  0
#define SFS_EPUN_BIT                  4
#define SFS_EHS_BIT                   5
#define SFS_EZI_BIT                   6
#define SFS_ZIF_BIT                   7
#define CLK_STAT_RUN_BIT              0
#define CLK_CFG_RUN_BIT               0
#define NOR_RESET_BIT                 9
#define SPIFI_PRIO                    15

/*********************************************************************
*
*       ASSERT_IS_LOCKED
*/
#if FS_SUPPORT_TEST
  #define ASSERT_IS_LOCKED()                  \
```



```
    if (_LockCnt != 1) {                      \
      FS_X_PANIC(FS_ERRCODE_INVALID_USAGE);   \
    }
#else
  #define ASSERT_IS_LOCKED()
#endif

/*********************************************************************
*
*       IF_TEST
*/
#if FS_SUPPORT_TEST
  #define IF_TEST(Expr)       Expr
#else
  #define IF_TEST(Expr)
#endif

/*********************************************************************
*
*      Static data
*
**********************************************************************
*/
#if FS_SUPPORT_TEST
  static U8 _LockCnt = 0;
  // Just to test if the Lock()/Unlock function are called correctly.
#endif

/*********************************************************************
*
*      Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       _GetFieldForm
*
*  Function description
*    Determines the value to be stored in the FIELDFORM field
*    of the SPIFI_CMD and SPIFI_MCMD registers. It also returns
*    the value of the SPIFI_CTRL.DUAL field.
*/
static U32 _GetFieldForm(U16 BusWidth, int * pIsDual) {
  U32 FieldForm;
  int IsDual;

  IsDual = 0;
  if (BusWidth == 0x112u) {
    FieldForm = CMD_FIELDFORM_QUAD_DUAL_DATA;
  // Data field is dual other fields are serial.
    IsDual    = 1;
  } else if (BusWidth == 0x114u) {
    FieldForm = CMD_FIELDFORM_QUAD_DUAL_DATA;
  // Data field is quad other fields are serial.
  } else if (BusWidth == 0x122u) {
    FieldForm = CMD_FIELDFORM_SERIAL_OPCODE;
  // Opcode field is serial. Other fields are dual.
    IsDual    = 1;
  } else if (BusWidth == 0x144u) {
    FieldForm = CMD_FIELDFORM_SERIAL_OPCODE;
  // Opcode field is serial. Other fields are quad.
  } else if (BusWidth == 0x444u) {
    FieldForm = CMD_FIELDFORM_ALL_QUAD_DUAL;
  // Opcode field is serial. Other fields are quad.
  } else {
    FieldForm = CMD_FIELDFORM_ALL_SERIAL;       // All fields of the command are serial.
  }
  if (pIsDual) {
    *pIsDual = IsDual;
  }
  return FieldForm;
}

/*********************************************************************
```



```
*
*       _GetFieldFormEx
*
*  Function description
*    Same functionality as _GetFieldForm() with BusWidth not encoded.
*/
static U32 _GetFieldFormEx(U8 BusWidth, int * pIsDual) {
  U32 FieldForm;
  int IsDual;

  IsDual = 0;
  if (BusWidth == 1) {
    FieldForm = CMD_FIELDFORM_ALL_SERIAL;       // All fields of the command are serial.
  } else if (BusWidth == 2) {
    FieldForm = CMD_FIELDFORM_ALL_QUAD_DUAL;    // All fields are dual.
    IsDual = 1;
  } else if (BusWidth == 4) {
    FieldForm = CMD_FIELDFORM_ALL_QUAD_DUAL;    // All fields are quad.
  } else {
    FieldForm = CMD_FIELDFORM_ALL_SERIAL;       // All fields of the command are serial.
  }
  if (pIsDual) {
    *pIsDual = IsDual;
  }
  return FieldForm;
}

/*********************************************************************
*
*       _GetFrameForm
*
*  Function description
*    Determines the value to be stored in the FAMEFORM field
*    of the SPIFI_CMD and SPIFI_MCMD registers.
*/
static U32 _GetFrameForm(unsigned NumBytesAddr) {
  U32 FrameForm;

  if (NumBytesAddr == 1) {
    FrameForm = CMD_FRAMEFORM_OPCODE_1BYTE;     // Opcode + 1 address byte.
  } else if (NumBytesAddr == 2) {
    FrameForm = CMD_FRAMEFORM_OPCODE_2BYTES;    // Opcode + 2 address bytes.
  } else if (NumBytesAddr == 3) {
    FrameForm = CMD_FRAMEFORM_OPCODE_3BYTES;    // Opcode + 3 address bytes.
  } else if (NumBytesAddr == 4) {
    FrameForm = CMD_FRAMEFORM_OPCODE_4BYTES;    // Opcode + 4 address bytes.
  } else {
    FrameForm = CMD_FRAMEFORM_OPCODE;           // Opcode, no address bytes.
  }
  return FrameForm;
}

/*********************************************************************
*
*       _Reset
*/
static void _Reset(void) {
  SPIFI_STAT = (1uL << STAT_RESET_BIT);
  // Reset the SPIFI controller. The controller is put into SPI mode.
  while (1) {
    if ((SPIFI_STAT & ((1uL << STAT_CMD_BIT)   |
                       (1uL << STAT_RESET_BIT) |
                       (1uL << STAT_MCINIT_BIT))) == 0) {
      break;
  // Wait until the SPIFI controller is ready again.
    }
  }
}

/*********************************************************************
*
*      Public code
*
**********************************************************************
*/
```



```
#if USE_OS

/*********************************************************************
*
*       SPIFI_IRQHandler
*/
void SPIFI_IRQHandler(void);
void SPIFI_IRQHandler(void) {
  OS_EnterInterrupt();
  SPIFI_STAT |= 1uL << STAT_INTRQ_BIT;          // Prevent other interrupts from occurring.
  FS_X_OS_Signal();
  OS_LeaveInterrupt();
}

#endif

/*********************************************************************
*
*      Public code (via callback)
*
**********************************************************************
*/

/*********************************************************************
*
*       _HW_Init
*
*  Function description
*    HW layer function. It is called before any other function of the physical layer.
*    It should configure the HW so that the other functions can access the NOR flash.
*
*  Return value
*    Frequency of the SPI clock in Hz.
*/
static int _HW_Init(U8 Unit) {
  unsigned NumLoops;

  FS_USE_PARA(Unit);    // This device has only one HW unit.
  //
  // Use IDIVA as clock for SPIFI. The input of IDIVA is PLL1.
  //
  IDIVA_CTRL     = 0
                 | (9uL << CGU_CLK_SEL_BIT)     // Use PLL1 as input
                 | (2uL << CGU_IDIV_BIT)        // Divide PLL1 by 3
                 | (1uL << CGU_AUTOBLOCK_BIT)
                 ;
  BASE_SPIFI_CLK = 0
                 | 12uL << CGU_CLK_SEL_BIT    // Use IDIVA as clock generator
                 | 1uL  << CGU_AUTOBLOCK_BIT
                 ;
  //
  // Enable the SPIFI clock if required.
  //
  if ((CLK_SPIFI_STAT & (1uL << CLK_STAT_RUN_BIT)) == 0) {
    CLK_SPIFI_CFG |= (1uL << CLK_CFG_RUN_BIT);
    while (1) {
      if (CLK_SPIFI_STAT & (1uL << CLK_CFG_RUN_BIT)) {
        ;
      }
    }
  }
  //
  // Clock pin.
  //
  SFSP3_3 = 0
          | (3uL << SFS_MODE_BIT)
          | (1uL << SFS_EPUN_BIT)
          | (1uL << SFS_EHS_BIT)
          | (1uL << SFS_EZI_BIT)
          ;
  //
  // Data pins.
  //
  SFSP3_4 = 0
          | (3uL << SFS_MODE_BIT)
          | (1uL << SFS_EPUN_BIT)
```



```
          | (1uL << SFS_EZI_BIT)
          ;
  SFSP3_5 = 0
          | (3uL << SFS_MODE_BIT)
          | (1uL << SFS_EPUN_BIT)
          | (1uL << SFS_EZI_BIT)
          ;
  SFSP3_6 = 0
          | (3uL << SFS_MODE_BIT)
          | (1uL << SFS_EPUN_BIT)
          | (1uL << SFS_EZI_BIT)
          ;
  SFSP3_7 = 0
          | (3uL << SFS_MODE_BIT)
          | (1uL << SFS_EPUN_BIT)
          | (1uL << SFS_EZI_BIT)
          ;
  //
  // Chip select pin.
  //
  SFSP3_8 = 0
          | (3uL << SFS_MODE_BIT)                   // Controlled by this HW layer.
          | (1uL << SFS_EPUN_BIT)
          ;
  //
  // Reset pin. Present only on 16-pin devices.
  //
  SFSP3_2 = 0
          | (4uL << SFS_MODE_BIT)                   // Controlled by this HW layer.
          | (1uL << SFS_EPUN_BIT)
          ;
  GPIO_DIR5 |= 1uL << NOR_RESET_BIT;
  //
  // Reset the NOR device.
  //
  GPIO_CLR5 = 1uL << NOR_RESET_BIT;                 // Assert reset signal (active low).
  NumLoops = RESET_DELAY_LOOPS;
  do {
    ;
  } while (--NumLoops);
  GPIO_SET5 = 1uL << NOR_RESET_BIT;                 // De-assert reset signal (active low).
#if USE_OS
  //
  // Disable the interrupt and configure the priority.
  //
  NVIC_DisableIRQ(SPIFI_IRQn);
  NVIC_SetPriority(SPIFI_IRQn, SPIFI_PRIO);
#endif
  //
  // Configure the SPIFI unit.
  //
  SPIFI_CTRL = 0
             | (0xFFFFuL << CTRL_TIMEOUT_BIT)       // Use the maximum timeout.
             | (0xFuL    << CTRL_CSHIGH_BIT)
  // Set minimum CS high time to maximum value.
             | (0x1uL    << CTRL_D_PRFTCH_DIS_BIT)  // Disable memory prefetches.
             | (0x1uL    << CTRL_MODE3_BIT)         // CLK idle mode is high.
             | (0x1uL    << CTRL_PRFTCH_DIS_BIT)    // Disables prefetching of cache lines.
             | (0x1uL    << CTRL_FBCLK_BIT)
  // Read data is sampled using a feedback clock from the SCK pin.
             ;
#if USE_OS
  //
  // Clear interrupt flag.
  //
  SPIFI_STAT = 1uL << STAT_INTRQ_BIT;
  NVIC_EnableIRQ(SPIFI_IRQn);
#endif
  return SPIFI_CLK_HZ;
}

/*********************************************************************
*
*       _HW_SetCmdMode
*
*  Function description
```



```
*    HW layer function. It enables the direct access to NOR flash via SPI.
*    This function disables the memory-mapped mode.
*/
static void _HW_SetCmdMode(U8 Unit) {
  ASSERT_IS_LOCKED();
  FS_USE_PARA(Unit);            // This device has only one HW unit.
  _Reset();
}

/*********************************************************************
*
*       _HW_SetMemMode
*
*  Function description
*    HW layer function. It enables the memory-mapped mode. In this mode
*    the data can be accessed by doing read operations from memory.
*    The HW is responsible to transfer the data via SPI.
*    This function disables the direct access to NOR flash via SPI.
*/
static void _HW_SetMemMode(U8 Unit, U8 ReadCmd, unsigned NumBytesAddr, unsigned NumBytesDummy, U16 BusWidth) {
  U32 FieldForm;
  U32 FrameForm;
  U32 OpCode;
  U32 IntLen;
  int IsDual;
  U32 MCmdReg;
  U32 v;

  ASSERT_IS_LOCKED();
  FS_USE_PARA(Unit);    // This device has only one HW unit.
  //
  // Determine how the data transferred is sent (serial, dual or quad)
  // and how many address bytes to send.
  //
  IsDual    = 0;
  FieldForm = _GetFieldForm(BusWidth, &IsDual);
  FrameForm = _GetFrameForm(NumBytesAddr);
  IntLen    = NumBytesDummy & CMD_INTLEN_MASK;
  OpCode    = ReadCmd & CMD_OPCODE_MASK;
  MCmdReg   = 0
            | (IntLen    << CMD_INTLEN_BIT)
            | (FieldForm << CMD_FIELDFORM_BIT)
            | (FrameForm << CMD_FRAMEFORM_BIT)
            | (OpCode    << CMD_OPCODE_BIT)
            ;
  //
  // Configure the SPIFI controller to work in memory-mapped mode.
  //
  v = SPIFI_CTRL;
  if (IsDual) {
    v |=   1uL << CTRL_DUAL_BIT;
  } else {
    v &= ~(1uL << CTRL_DUAL_BIT);
  }
  SPIFI_CTRL = v;
  SPIFI_MCMD = MCmdReg;
  //
  // Wait until MCMD is written
  //
  while (1) {
    if (SPIFI_STAT & (1uL << STAT_MCINIT_BIT)) {
      break;
    }
  }
  //
  // Perform dummy read to bring controller into sync and invalidate all caches etc.
  // It seems to be necessary but not documented in the datasheet.
  //
  *(volatile U32*)SPIFI_DATA_ADDR;
}

/*********************************************************************
*
*       _HW_ExecCmd
*
*  Function description
```



```
*    HW layer function. It requests the NOR flash to execute a simple command.
*    The HW has to be in SPI mode.
*/
static void _HW_ExecCmd(U8 Unit, U8 Cmd, U8 BusWidth) {
  U32 FieldForm;
  int IsDual;
  U32 FrameForm;
  U32 CmdReg;
  U32 v;

  ASSERT_IS_LOCKED();
  FS_USE_PARA(Unit);    // This device has only one HW unit.
  IsDual    = 0;
  FieldForm = _GetFieldFormEx(BusWidth, &IsDual);
  FrameForm = _GetFrameForm(0);                 // 0 - No address bytes are sent.
  CmdReg    = 0
            | (FieldForm << CMD_FIELDFORM_BIT)  // Configure the transfer mode.
            | (FrameForm << CMD_FRAMEFORM_BIT)  // Send only the command byte.
            | ((U32)Cmd  << CMD_OPCODE_BIT)     // Command code.
            ;
  v = SPIFI_CTRL;
  if (IsDual) {
    v |=   1uL << CTRL_DUAL_BIT;
  } else {
    v &= ~(1uL << CTRL_DUAL_BIT);
  }
  SPIFI_CTRL = v;
  SPIFI_CMD  = CmdReg;
  //
  // Wait until nCS is set high, indicating that the command has been completed.
  //
  while (1) {
    if ((SPIFI_STAT & (1uL << STAT_CMD_BIT)) == 0) {
      break;
    }
  }
}

/*********************************************************************
*
*       _HW_ReadData
*
*  Function description
*    HW layer function. It transfers data from NOR flash to MCU.
*    The HW has to be in SPI mode.
*/
static void _HW_ReadData(U8 Unit, U8 Cmd, const U8 * pPara, unsigned NumBytesPara, unsigned NumBytesAddr, U8
  U32      DataLen;
  U32      AddrReg;
  U32      IDataReg;
  U32      CmdReg;
  U32      FieldForm;
  int      IsDual;
  U32      FrameForm;
  U32      IntLen;
  unsigned NumBytes;
  U32      v;

  ASSERT_IS_LOCKED();
  FS_USE_PARA(Unit);    // This device has only one HW unit.
  //
  // Fill local variables.
  //
  IsDual    = 0;
  FieldForm = _GetFieldForm(BusWidth, &IsDual);
  FrameForm = _GetFrameForm(NumBytesAddr);
  IntLen    = 0;
  DataLen   = NumBytesData & CMD_DATALEN_MASK;
  AddrReg   = 0;
  IDataReg  = 0;
  //
  // Encode the address.
  //
  if (NumBytesAddr) {
    NumBytes = NumBytesAddr;
    do {
```



```
      AddrReg <<= 8;
      AddrReg  |= (U32)(*pPara++);
    } while (--NumBytes);
  }
  //
  // Encode the dummy and mode bytes.
  //
  if (NumBytesPara > NumBytesAddr) {
    NumBytes = NumBytesPara - NumBytesAddr;
    IntLen   = NumBytes;
    do {
      IDataReg <<= 8;
      IDataReg  |= (U32)(*pPara++);
    } while (--NumBytes);
  }
  CmdReg = 0
         | (DataLen   << CMD_DATALEN_BIT)
         | (IntLen    << CMD_INTLEN_BIT)
         | (FieldForm << CMD_FIELDFORM_BIT)
         | (FrameForm << CMD_FRAMEFORM_BIT)
         | ((U32)Cmd  << CMD_OPCODE_BIT)
         ;
  //
  // Set dual/quad mode.
  //
  v = SPIFI_CTRL;
  if (IsDual) {
    v |=   1uL << CTRL_DUAL_BIT;
  } else {
    v &= ~(1uL << CTRL_DUAL_BIT);
  }
  SPIFI_CTRL = v;
  //
  // Execute the command.
  //
  SPIFI_ADDR  = AddrReg;
  SPIFI_IDATA = IDataReg;
  SPIFI_CMD   = CmdReg;
  //
  // Read data from NOR flash.
  //
  if (NumBytesData) {
    do {
      *pData++ = SPIFI_DATA8;
    } while (--NumBytesData);
  }
  //
  // Wait until nCS is set high, indicating that the command has been completed.
  //
  while (1) {
    if ((SPIFI_STAT & (1uL << STAT_CMD_BIT)) == 0) {
      break;
    }
  }
}

/*********************************************************************
*
*       _HW_WriteData
*
*  Function description
*    HW layer function. It transfers data from MCU to NOR flash.
*    The HW has to be in SPI mode.
*/
static void _HW_WriteData(U8 Unit, U8 Cmd, const U8 * pPara, unsigned NumBytesPara, unsigned NumBytesAddr, c
  U32      DataLen;
  U32      AddrReg;
  U32      IDataReg;
  U32      FieldForm;
  int      IsDual;
  U32      FrameForm;
  U32      IntLen;
  unsigned NumBytesAtOnce;
  unsigned NumBytes;
  U32      v;
  U32      CmdReg;
```



```
  ASSERT_IS_LOCKED();
  FS_USE_PARA(Unit);    // This device has only one HW unit.
  //
  // Fill local variables.
  //
  IsDual    = 0;
  FieldForm = _GetFieldForm(BusWidth, &IsDual);
  FrameForm = _GetFrameForm(NumBytesAddr);
  IntLen    = 0;
  DataLen   = NumBytesData & CMD_DATALEN_MASK;
  AddrReg   = 0;
  IDataReg  = 0;
  //
  // Encode the address.
  //
  if (NumBytesAddr) {
    NumBytes = NumBytesAddr;
    do {
      AddrReg <<= 8;
      AddrReg  |= (U32)(*pPara++);
    } while (--NumBytes);
  }
  //
  // Encode the dummy and mode bytes.
  //
  if (NumBytesPara > NumBytesAddr) {
    NumBytes = NumBytesPara - NumBytesAddr;
    IntLen   = NumBytes;
    do {
      IDataReg <<= 8;
      IDataReg  |= (U32)(*pPara++);
    } while (--NumBytes);
  }
  CmdReg = 0
         | (DataLen   << CMD_DATALEN_BIT)
         | (1uL       << CMD_DOUT_BIT)     // Transfer data to NOR flash
         | (IntLen    << CMD_INTLEN_BIT)
         | (FieldForm << CMD_FIELDFORM_BIT)
         | (FrameForm << CMD_FRAMEFORM_BIT)
         | ((U32)Cmd  << CMD_OPCODE_BIT)
         ;
  //
  // Set dual/quad mode.
  //
  v = SPIFI_CTRL;
  if (IsDual) {
    v |=   1uL << CTRL_DUAL_BIT;
  } else {
    v &= ~(1uL << CTRL_DUAL_BIT);
  }
  SPIFI_CTRL = v;
  //
  // Execute the command.
  //
  SPIFI_ADDR  = AddrReg;
  SPIFI_IDATA = IDataReg;
  SPIFI_CMD   = CmdReg;
  //
  // Wait for the command to start execution.
  //
  if (NumBytesData) {
    while (1) {
      if (SPIFI_STAT & (1uL << STAT_CMD_BIT)) {
        break;
      }
    }
  }
  //
  // Write data to NOR flash.
  //
  if (NumBytesData) {
    do {
      NumBytesAtOnce = NumBytesData >= 4 ? 4 : 1;
      if (NumBytesAtOnce == 4) {
        v = 0
```



```
          | (U32)*pData
          | ((U32)(*(pData + 1)) << 8)
          | ((U32)(*(pData + 2)) << 16)
          | ((U32)(*(pData + 3)) << 24)
          ;
        SPIFI_DATA32 = v;
      } else {
        SPIFI_DATA8 = *pData;
      }
      NumBytesData -= NumBytesAtOnce;
      pData        += NumBytesAtOnce;
    } while (NumBytesData);
  }
  //
  // Wait until nCS is set high, indicating that the command has been completed.
  //
  while (1) {
    if ((SPIFI_STAT & (1uL << STAT_CMD_BIT)) == 0) {
      break;
    }
  }
}

/*********************************************************************
*
*       _HW_Poll
*
*  Function description
*    HW layer function. Sends a command repeatedly and checks the
*    response for a specified condition.
*
*  Return value
*    > 0    Timeout occurred.
*    ==0    OK, bit set to specified value.
*    < 0    Feature not supported.
*
*  Additional information
*    The function executes periodically a command and waits
*    until specified bit in the response returned by the NOR flash
*    is set to a value specified by BitValue. The position of the bit
*    that has to be checked is specified by BitPos where 0 is the
*    position of the least significant bit in the byte.
*/
static int _HW_Poll(U8 Unit, U8 Cmd, U8 BitPos, U8 BitValue, U32 Delay, U32 TimeOut_ms, U16 BusWidth) {
  int r;

  ASSERT_IS_LOCKED();
  FS_USE_PARA(Unit);
  FS_USE_PARA(Delay);
  r = -1;                 // Set to indicate that the feature is not supported.
#if USE_OS
  {
    U32 AddrReg;
    U32 IDataReg;
    U32 CmdReg;
    U32 FieldForm;
    int IsDual;
    U32 FrameForm;
    U32 v;

    //
    // Fill local variables.
    //
    IsDual    = 0;
    FieldForm = _GetFieldForm(BusWidth, &IsDual);
    FrameForm = _GetFrameForm(0);
  // The command does not require any address bytes.
    AddrReg   = 0;
    IDataReg  = 0;
    CmdReg = 0
           | ((BitValue & CMD_BITVAL_MASK) << CMD_BITVAL_BIT)
           | ((BitPos & CMD_BITPOS_MASK)   << CMD_BITPOS_BIT)
           | (1uL                          << CMD_POLL_BIT)
           | (FieldForm                    << CMD_FIELDFORM_BIT)
           | (FrameForm                    << CMD_FRAMEFORM_BIT)
           | (Cmd                          << CMD_OPCODE_BIT)
```



```
          ;
    //
    // Set dual/quad mode.
    //
    v = SPIFI_CTRL;
    if (IsDual) {
      v |=   1uL << CTRL_DUAL_BIT;
    } else {
      v &= ~(1uL << CTRL_DUAL_BIT);
    }
    SPIFI_CTRL = v;
    //
    // Execute the command.
    //
    SPIFI_STAT  = 1uL << STAT_INTRQ_BIT;    // Clear the interrupt flag.
    SPIFI_CTRL |= 1uL << CTRL_INTEN_BIT;    // Enable interrupt.
    SPIFI_ADDR  = AddrReg;
    SPIFI_IDATA = IDataReg;
    SPIFI_CMD   = CmdReg;
    r = FS_X_OS_Wait(TimeOut_ms);
    SPIFI_CTRL &= ~(1uL << CTRL_INTEN_BIT); // Disable interrupt.
    if (r) {
      r = 1;                                // A timeout occurred.
    } else {
      r = 0;                                // Bit set to the requested value.
    }
    _Reset();                               // Cancel the polling operation.
    //
    // Wait until nCS is set high, indicating that the command has been completed.
    //
    while (1) {
      if ((SPIFI_STAT & (1uL << STAT_CMD_BIT)) == 0) {
        break;
      }
    }
  }
#else
  FS_USE_PARA(Cmd);
  FS_USE_PARA(BitPos);
  FS_USE_PARA(BitValue);
  FS_USE_PARA(Delay);
  FS_USE_PARA(TimeOut_ms);
  FS_USE_PARA(BusWidth);
#endif
  return r;
}

/*********************************************************************
*
*       _HW_Delay
*
*  Function description
*    HW layer function. Blocks the execution for the specified number
*    of milliseconds.
*/
static int _HW_Delay(U8 Unit, U32 ms) {
  int r;

  FS_USE_PARA(Unit);
  r = -1;                 // Set to indicate that the feature is not supported.
#if USE_OS
  if (ms) {
    FS_X_OS_Delay(ms);
    r = 0;
  }
#else
  FS_USE_PARA(ms);
#endif
  return r;
}

/*********************************************************************
*
*       _HW_Lock
*
*  Function description
```



```
*    HW layer function. Requests exclusive access to SPI bus.
*/
static void _HW_Lock(U8 Unit) {
  FS_USE_PARA(Unit);
  IF_TEST(_LockCnt++);
  ASSERT_IS_LOCKED();
}

/*********************************************************************
*
*       _HW_Unlock
*
*  Function description
*    HW layer function. Releases the exclusive access of SPI bus.
*/
static void _HW_Unlock(U8 Unit) {
  FS_USE_PARA(Unit);
  ASSERT_IS_LOCKED();
  IF_TEST(_LockCnt--);
}

/*********************************************************************
*
*       Global data
*
**********************************************************************
*/
const FS_NOR_HW_TYPE_SPIFI FS_NOR_HW_SPIFI_LPC4322_SEGGER_QSPIFI_Test_Board = {
  _HW_Init,
  _HW_SetCmdMode,
  _HW_SetMemMode,
  _HW_ExecCmd,
  _HW_ReadData,
  _HW_WriteData,
  _HW_Poll,
  _HW_Delay,
  _HW_Lock,
  _HW_Unlock
};

/************************** End of file ***************************/
```


6.5 MMC/SD card driver

6.5.1 General information

emFile supports the use of MultiMediaCard (MMC) and Secure Digital (SD) cards as well as of eMMC (Embedded MMC) as storage device.

MMC/SD cards are mechanically small, removable mass storage devices. The main design goal of these devices is 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. eMMC devices are operate in the same way as an MMC card with the difference that they are not removable devices. Instead, they can be soldered on a circuit board.

In order to meet the requirements of any system, the MMC/SD cards are designed to operate in two different I/O modes:

- SPI mode
- Card mode

Two device drivers are provided that support these different operating modes:

- *SPI MMC/SD driver* - Supports legacy MMC and SD cards and it transfers the data via standard SPI.
- *Card Mode MMC/SD driver* - Supports legacy eMMC and SD cards and it can transfer the data via one, four or eight data lines. The drivers require very little RAM and provide and extremely efficient data transfer. The selection of the driver depends on the access mode supported by the target hardware.

For more technical details about MMC and SD cards, check the documents and specifications available on https://www.jedec.org and https://www.sdcard.org

6.5.1.1 Fail-safe operation

The data will be preserved in case of unexpected reset but a power failure can be critical. If the card does not have sufficient time to complete a write operation, data may be lost. As a countermeasure the hardware has to make sure that the power supply to the card drops slowly. The SD specification states that a write operation should finish within 250 ms. Typically, the industrial-grade SD cards are able to handle power failures internally. Please note that an SD card can buffer write operations which can increase the write time. If required, the write buffering can be disabled in the Card Mode MMC/SD driver using `FS_MMC_CM_AllowBufferedWrite()`.

6.5.1.2 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.

6.5.1.3 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. `FS_MMC_ActivateCRC()` and `FS_MMC_DeactivateCRC()` can be used to control the CRC calculation in SPI mode. In card mode the CRC is computed in the hardware by the SD host controller and can not be activated or deactivated.

6.5.1.4 Power control

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.

6.5.1.5 Supported hardware

The following memory card types are supported:

- MMC - eMMC, MMC card, RS-MMC, RS-MMC DV, MMCplus, MMCmobile, MMCmicro
- SD - SDSC (SD Standard Capacity), SDHC (SD High Capacity), SDXC (SD eXtended Capacity) in any form factor.

> **Note**
>
> The MMC cards conforming with the version 4.x of the JEDEC specification (MMCplus, MMCmobile, MMCmicro, eMMC) work only with the Card Mode MMC/SD driver since these devices do not support the SPI mode.

The difference between MMC and SD cards is that SD cards can operate with a higher clock frequency. In normal mode the range range of the clock frequency supplied to an SD card can be between 0 and 25 MHz, whereas MMC cards can only operate up to 20 MHz. The newer MMC cards that comply to the version 4.x or higher of the MMC specification can also operate at higher frequencies up to 26 MHz.

An SD card can also operate in high speed mode with a clock frequency up to 50 MHz. The MMC cards conforming to the version 3.x or lower of the MMC specification do not support the high speed mode. The version 4.x of the MMC specification improves this and allows the MMC cards to operate with a clock frequency of up to 52 MHz 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 interface signals with MMC cards typically requiring more signals than the SD cards. The number of signals used depends on the operating mode.

Card mode

In this mode the MMC cards compliant to version 3.x or smaller of the MMC specification require 6 signals: 1 command, 1 clock, 1 data and 3 power lines. The MMC cards compliant with the version 4.x or higher of the MMC specification require up to 13 signals: 1 command, 1 clock, up to 8 data and 3 power lines. In contrast to the MMC cards, SD cards require up to 9 signals: 1 command, 1 clock, up to 4 data and 3 power lines.

SPI mode

Both card systems use the same interface signals: 1 chip select, 1 data input, 1 data output, 1 clock and 3 power lines.

6.5.1.6 Pin description - MMC/SD card in card mode

The table below describes the pin assignment of the electrical interface of SD and MMC cards working 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 dis-

Pin No.	Name	Type	Description
			abled 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 MMC bus master (card host controller) to the card and responses are sent from the cards to the host.
3	VSS	Power supply	Supply voltage ground.
4	VDD	Power supply	Supply voltage.
5	CLK	Input	Clock signal. With each cycle of this signal a one bit transfer on the command and data lines is done. The frequency may vary between zero and the maximum clock frequency.
6	VSS2	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]. 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]. 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 MMC storage devices. On SD cards this line does not exist. Not used for data transfers via four data lines. For data transfer via eight data lines, this line is carries the bit 4.
11	DAT5	Input/Output using push pull drivers	Data line [Bit 5]. Relevant only for MMC storage devices. On SD cards this line does not exist. Not used for data transfers via four data lines. For data transfer via eight data lines, this line is carries the bit 5.
12	DAT6	Input/Output using push pull drivers	Data line [Bit 6]. Relevant only for MMC storage devices. On SD cards this line does not exist. Not used for data transfers via four data lines. For data transfer via eight data lines, this line is carries the bit 6.
13	DAT7	Input/Output using push pull drivers	Data line [Bit 7]. Relevant only for MMC storage devices. On SD cards this line does not exist. Not used for data transfers via four data

Pin No.	Name	Type	Description
			lines. For data transfer via eight data lines, this line is carries the bit 7.

The table below describes the pin assignment of the electrical interface of a micro SD card working in 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 MMC bus master (card host controller) to the card and responses are sent from the cards to the host.
4	VDD	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	VSS	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.

Additional information

External pull-up resistors must be connected to all data lines even it they are not used. Otherwise, unexpected high current consumption may occur due to the floating of these signals.

6.5.1.7 Pin description - MMC/SD card in SPI mode

The table below describes the pin assignment of the electrical interface of SD and MMC cards working 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	VSS	Supply ground	Power supply ground. Supply voltage ground.
4	VDD	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	VSS2	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.

The table below describes the pin assignment of the electrical interface of a micro SD card working in 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	VDD	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	VSS	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.

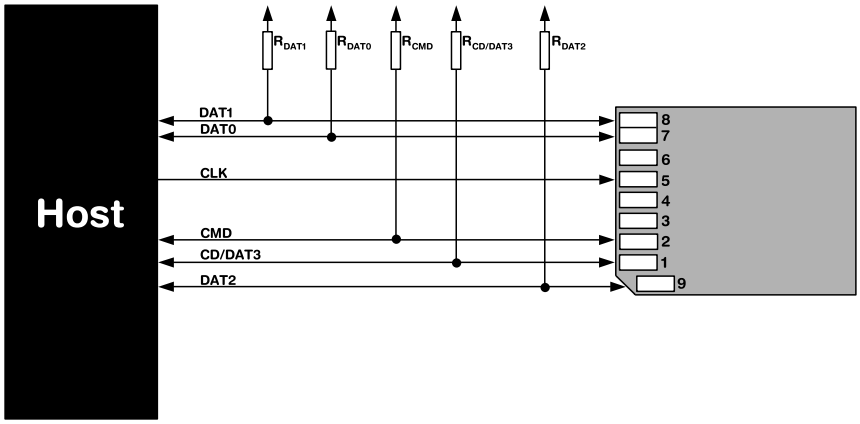
Additional information

- 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.

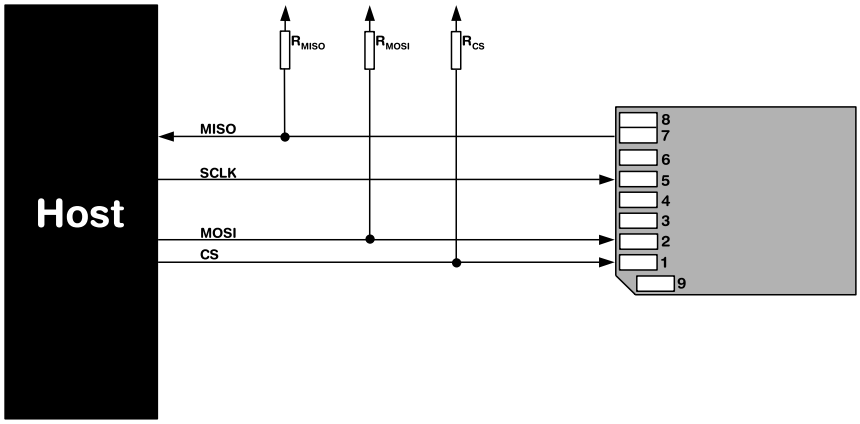
- Data polarity is normal, which means a logical “1” is represented with a high level on the data line and a logical “0” is represented with low-level.
- MMC/SD cards support different voltage ranges. Initial voltage should be 3.3V.

6.5.1.8 Interfacing with an MMC/SD card

The following picture shows how an MMC/SD card working in card mode can be interfaced to a system.



The following picture shows how an MMC/SD card working in SPI mode can be interfaced to a system.



6.5.2 SPI MMC/SD driver

6.5.2.1 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).

6.5.2.2 Configuring the driver

This section describes how to configure the file system to make use of the SPI MMC/SD driver.

6.5.2.2.1 Runtime configuration

The driver must be added to the file system by calling `FS_AddDevice()` with the driver identifier set to the address of `FS_MMC_SPI_Driver` structure. This function call together with other function calls that configure the driver operation have to be added to `FS_X_AddDevices()` as demonstrated in the following example. This example shows how to configure the file system to access an MMC/SD card connected via SPI.

```
#include "FS.h"
#include "FS_MMC_HW_SPI_Template.h"

#define ALLOC_SIZE 0x1300               // Size defined 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().
*/
void FS_X_AddDevices(void) {
  //
  // Give the file system memory to work with.
  //
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add and configure the SPI MMC/SD driver.
  //
  FS_AddDevice(&FS_MMC_SPI_Driver);
  FS_MMC_SetHWType(0, &FS_MMC_HW_SPI_Template);
  // FS_MMC_ActivateCRC();   // Uncommenting this line activates
                             // the CRC calculation of the SPI MMC/SD driver.
  //
  // Configure the file system for fast write operations.
  //
#if FS_SUPPORT_FILE_BUFFER
  FS_ConfigFileBufferDefault(512, FS_FILE_BUFFER_WRITE);
#endif
  FS_SetFileWriteMode(FS_WRITEMODE_FAST);
}
```

The API functions listed in the next table can be used by the application to configure the behavior of the SPI MMC/SD driver. The application can call them only at the file system initialization in `FS_X_AddDevices()`.

Function	Description
`FS_MMC_ActivateCRC()`	Enables the data verification.
`FS_MMC_DeactivateCRC()`	Disables the data verification.
`FS_MMC_SetHWType()`	Configures the hardware access routines.

6.5.2.2.1.1 FS_MMC_ActivateCRC()

Description

Enables the data verification.

Prototype

```
void FS_MMC_ActivateCRC(void);
```

Additional information

This function is optional. The data verification uses a 16-bit CRC to detect any corruption of the data being exchanged with the storage device. By default, the data verification is disabled to improve performance. `FS_MMC_ActivateCRC()` can be used at runtime to enable the data verification for all driver instances. The data verification can be disabled via `FS_MMC_DeactivateCRC()`.

6.5.2.2.1.2 FS_MMC_DeactivateCRC()

Description

Disables the data verification.

Prototype

```
void FS_MMC_DeactivateCRC(void);
```

Additional information

This function is optional. It can be used by an application to disable the data verification previously enabled via `FS_MMC_ActivateCRC()`.

6.5.2.2.1.3 FS_MMC_SetHWType()

Description

Configures the hardware access routines.

Prototype

```
void FS_MMC_SetHWType(      U8                    Unit,
                      const FS_MMC_HW_TYPE_SPI * pHWType);
```

Additional information

This function is mandatory and it has to be called once for each instance of the driver.

6.5.2.3 Additional driver functions

These functions are optional. They can be used to get information about the operation of the Card Mode MMC/SD driver and to perform additional operations on the used SD card or MMC device.

Function	Description
FS_MMC_GetCardId()	This function retrieves the card Id of SD/ MMC card.
FS_MMC_GetStatCounters()	Returns the value of statistical counters.
FS_MMC_ResetStatCounters()	Sets the values of all statistical counters to 0.

6.5.2.3.1 FS_MMC_GetCardId()

Description

This function retrieves the card Id of SD/MMC card.

Prototype

```
int FS_MMC_GetCardId(U8               Unit,
                     FS_MMC_CARD_ID * pCardId);
```

Parameters

Parameter	Description
Unit	Device index number.
pCardId	out Card identification data.

Return value

= 0 CardId has been read.
≠ 0 An error has occurred.

Additional information

This function is optional. The application can call this function to get the information stored in the CID register of an MMC or SD card. The CID register stores information which can be used to uniquely identify the card such as serial number, product name, manufacturer id, etc. For more information about the information stored in this register refer to SD or MMC specification.

Example

The following example shows how to read and decode the contents of the CID register.

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

void SampleMMCGetCardId(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_MEMSET(&CardId, 0, sizeof(CardId));
  FS_MMC_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);
}
```


6.5.2.3.2 FS_MMC_GetStatCounters()

Description

Returns the value of statistical counters.

Prototype

```
void FS_MMC_GetStatCounters(U8                      Unit,
                            FS_MMC_STAT_COUNTERS * pStat);
```

Parameters

Parameter	Description
Unit	Index of the driver (0-based)
pStat	out The values of statistical counters.

Additional information

This function is optional. The SPI SD/MMC driver collects statistics about the number of internal operations such as the number of logical sectors read or written by the file system layer. The application can use FS_MMC_GetStatCounters() to get the current value of these counters. The statistical counters are automatically set to 0 when the storage device is mounted or when the application calls FS_MMC_ResetStatCounters().

The statistical counters are available only when the file system is compiled with FS_DEBUG_LEVEL greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL or with FS_MMC_ENABLE_STATS set to 1.

6.5.2.3.3 FS_MMC_ResetStatCounters()

Description

Sets the values of all statistical counters to 0.

Prototype

```
void FS_MMC_ResetStatCounters(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).

Additional information

This function is optional. The statistical counters are automatically set to 0 when the storage device is mounted. The application can use FS_MMC_ResetStatCounters() at any time during the file system operation. The statistical counters can be queried via FS_MMC_GetStatCounters().

The statistical counters are available only when the file system is compiled with FS_DEBUG_LEVEL greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL or with FS_MMC_ENABLE_STATS set to 1.

6.5.2.4 Performance and resource usage

6.5.2.4.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the Card Mode MMC/SD driver was measured using the SEGGER Embedded Studio IDE V4.20 configured to generate code for a Cortex-M4 CPU in Thumb mode and with the size optimization enabled.

Usage: 2.8 Kbytes

6.5.2.4.2 Static RAM usage

Static RAM usage refers to the amount of RAM required by the Card Mode MMC/SD driver internally for all the driver instances. The number of bytes can be seen in the compiler list file of the `FS_MMC_CM_Drv.c` file.

Usage: 12 bytes

6.5.2.4.3 Dynamic RAM usage

Dynamic RAM usage is the amount of RAM allocated by the driver at runtime. The amount of RAM required depends on the compile time and runtime configuration.

Usage: 22 bytes

6.5.2.4.4 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. All values are given in Mbytes/second.

CPU type	MMC/SD device	Write speed	Read speed
Atmel AT91SAM7S (48 MHz)	Ultron 256MB SD card	**2.3**	**2.3**
Atmel AT91SAM9261 (178 MHz)	Ultron 256MB SD card	**2.3**	**2.5**
TI TM4C129	SanDisk 16GB Ultra PLUS SD card	**2.0**	**2.2**

6.5.3 Card Mode MMC/SD driver

6.5.3.1 Configuring the driver

This section describes how to configure the file system to make use of the Card Mode MMC/SD driver.

6.5.3.1.1 Runtime configuration

The driver must be added to the file system by calling `FS_AddDevice()` with the driver identifier set to the address of `FS_MMC_CM_Driver` structure. This function call together with other function calls that configure the driver operation have to be added to `FS_X_AddDevices()` as demonstrated in the following example. This example shows how to configure the file system to access an MMC/SD card in card mode.

```
#include "FS.h"
#include "FS_MMC_HW_CM_Template.h"

#define ALLOC_SIZE 0x1300               // Size defined 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().
*/
void FS_X_AddDevices(void) {
  //
  // Give the file system memory to work with.
  //
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add and configure the Card mode MMC/SD driver.
  //
  FS_AddDevice(&FS_MMC_CardMode_Driver);
  FS_MMC_CM_Allow4bitMode(0, 1);
  FS_MMC_CM_SetHWType(0, &FS_MMC_HW_CM_Template);
  //
  // Configure the file system for fast write operations.
  //
#if FS_SUPPORT_FILE_BUFFER
  FS_ConfigFileBufferDefault(512, FS_FILE_BUFFER_WRITE);
#endif
  FS_SetFileWriteMode(FS_WRITEMODE_FAST);
}
```

The API functions listed in the next table can be used by the application to configure the behavior of the Card Mode MMC/SD driver. The application can call them only at the file system initialization in `FS_X_AddDevices()`.

Function	Description
`FS_MMC_CM_Allow4bitMode()`	Allows the driver to exchange the data via 4 lines.
`FS_MMC_CM_Allow8bitMode()`	Allows the driver to exchange the data via 8 data lines.
`FS_MMC_CM_AllowBufferedWrite()`	Enables / disables the write buffering.
`FS_MMC_CM_AllowHighSpeedMode()`	Allows the driver to exchange the data in high speed mode.
`FS_MMC_CM_AllowPowerSaveMode()`	Configures if the driver has to request the eMMC to save power.

Function	Description
FS_MMC_CM_AllowReliableWrite()	Allows the driver to use reliable write operations for MMC devices.
FS_MMC_CM_SetSectorRange()	Configures an area for data storage.
FS_MMC_CM_SetHWType()	Configures the HW access routines.

6.5.3.1.1.1 FS_MMC_CM_Allow4bitMode()

Description

Allows the driver to exchange the data via 4 lines.

Prototype

```
void FS_MMC_CM_Allow4bitMode(U8 Unit,
                             U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
OnOff	Indicates if the 4-bit mode should be enabled or disabled. • 0 Data is exchanged via 1 line. • 1 Data is exchanged via 4 lines.

Additional information

This function is optional. By default, the 4-bit mode is disabled which means that the Card mode MMC/SD driver exchanges data via only one data line. Using 4-bit mode can help increase the performance of the data transfer. The 4-bit mode is used for the data transfer only if the connected MMC/SD card supports it which is typically the case with all modern cards. If not then the Card mode MMC/SD driver falls back to 1-bit mode.

The application can query the actual number of data lines used by the Card mode MMC/SD driver for the data transfer by evaluating the value of BusWith member of the FS_MMC_CARD_INFO structure returned by FS_MMC_CM_GetCardInfo().

The application is permitted to call this function only at the file system initialization in FS_X_AddDevices().

6.5.3.1.1.2 FS_MMC_CM_Allow8bitMode()

Description

Allows the driver to exchange the data via 8 data lines.

Prototype

```
void FS_MMC_CM_Allow8bitMode(U8 Unit,
                             U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
OnOff	Indicates if the 8-bit mode should be enabled or disabled. • 0 Data is exchanged via 1 or 4 lines. • 1 Data is exchanged via 8 lines.

Additional information

This function is optional. By default, the 8-bit mode is disabled which means that the Card mode MMC/SD driver exchanges data via only one data line. Using 8-bit mode can help increase the performance of the data transfer. The 8-bit mode is used for the data transfer only if the connected MMC/SD card supports it. If not then the Card mode MMC/SD driver falls back to either 4- or 1-bit mode. Only MMC devices support the 8-bit mode. The SD cards are not able to transfer the data via 8-bit lines.

The application can query the actual number of data lines used by the Card mode MMC/SD driver for the data transfer by evaluating the value of BusWith member of the FS_MMC_CARD_INFO structure returned by FS_MMC_CM_GetCardInfo().

The application is permitted to call this function only at the file system initialization in FS_X_AddDevices().

6.5.3.1.1.3 FS_MMC_CM_AllowBufferedWrite()

Description

Enables / disables the write buffering.

Prototype

```
void FS_MMC_CM_AllowBufferedWrite(U8 Unit,
                                  U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
OnOff	Specifies if the buffered write operation should be enabled or not. • 0 Buffered write operation is disabled. • 1 BUffered write operation is enabled.

Additional information

SD and MMC storage devices can perform write operations in parallel to receiving data from the host by queuing write requests. This feature is used by the driver in order to achieve the highest write performance possible. In case of a power fail the hardware has to prevent that the write operation is interrupted by powering the storage device until the write queue is emptied. The time it takes the storage device to empty the queue is not predictable and it can take from a few hundreds o milliseconds to a few seconds to complete. This function allows the application to disable the buffered write and thus reduce the time required to supply the storage device at power fail. With the write buffering disabled the driver writes only one sector at time and it waits for the previous write sector operation to complete.

Disabling the write buffering can considerably reduce the write performance. Most of the industrial grade SD and MMC storage devices are fail safe so that disabling the write buffering is not required. For more information consult the data sheet of your storage device.

The application is permitted to call this function only at the file system initialization in `FS_X_AddDevices()`.

6.5.3.1.1.4 FS_MMC_CM_AllowHighSpeedMode()

Description

Allows the driver to exchange the data in high speed mode.

Prototype

```
void FS_MMC_CM_AllowHighSpeedMode(U8 Unit,
                                  U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
OnOff	Indicates if the high-speed mode should be enabled or disabled. • 0 Use standard speed mode. • 1 Use high speed mode.

Additional information

This function is optional. The application can use this function to request the Card mode MMC/SD driver to use the highest clock frequency supported by the used MMC/SD card. The standard clock frequency supported by an SD card is 25 MHz and 26 MHz by an MMC device. This is the clock frequency used by the Card mode MMC/SD driver after the initialization of the MMC/SD card. However, most of the modern SD cards and MMC devices are able to exchange the data at higher clock frequencies up to 50 MHz for SD cards and 52 MHz for MMC devices. This high speed mode has to be explicitly enabled in the SD card or MMC device after initialization. The Card mode SD/MMC driver automatically enables the high speed mode in the SD card or MMC device if FS_MMC_CM_AllowHighSpeedMode() is called with OnOff set to 1 and the used SD card or MMC device actually supported.

The application can check if the Card mode MMC/SD driver is actually using the high speed mode for the data transfer by evaluating the IsHighSpeedMode member of the FS_MMC_CARD_INFO structure returned by FS_MMC_CM_GetCardInfo().

The high speed mode can be used only if the SD/MMC host controller supports it. The availability of this functionality is not checked by the Card mode MMC/SD driver.

The application is permitted to call this function only at the file system initialization in FS_X_AddDevices().

6.5.3.1.1.5 FS_MMC_CM_AllowPowerSaveMode()

Description

Configures if the driver has to request the eMMC to save power.

Prototype

```
int FS_MMC_CM_AllowPowerSaveMode(U8 Unit,
                                 U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
OnOff	Specifies if the MMC device has to be put to sleep between data transfers • 0 The MMC device remains active. • 1 The MMC device is put to sleep.

Return value

= 0 OK, feature configured.
≠ 0 An error occurred.

Additional information

This function is optional and active only if the sources are compiled with the FS_MMC_SUPPORT_POWER_SAVE set to 1.

6.5.3.1.1.6 FS_MMC_CM_AllowReliableWrite()

Description

Allows the driver to use reliable write operations for MMC devices.

Prototype

```
void FS_MMC_CM_AllowReliableWrite(U8 Unit,
                                  U8 OnOff);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
OnOff	Specifies if the reliable write operation should be enabled or disabled. • 0 Reliable write operation is disabled (default). • 1 Reliable write operation is enabled.

Additional information

This function is optional. A reliable write operation makes sure that the sector data is not corrupted in case of a unexpected reset. MMC devices compliant with the version 4.3 or newer of the MMC specification support a fail-safe write feature which makes sure that the old data remains unchanged until the new data is successfully programed. Using this type of write operation the data remains valid in case of an unexpected reset which improves the data reliability. The support for this feature is optional and the Card mode MMC/SD driver activates it only if the used MMC device actually supports it and it `FS_MMC_CM_AllowReliableWrite()` has been called with `OnOff` set to 1.

Please note that enabling the reliable write feature can possibly reduce the write performance.

The application is permitted to call this function only at the file system initialization in `FS_X_AddDevices()`.

6.5.3.1.1.7 FS_MMC_CM_SetSectorRange()

Description

Configures an area for data storage.

Prototype

```
void FS_MMC_CM_SetSectorRange(U8  Unit,
                              U32 StartSector,
                              U32 MaxNumSectors);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
StartSector	Index of the first sector to be used as storage.
MaxNumSectors	Maximum number of sectors to be used as storage.

Additional information

This function is optional. It allows an application to use only a specific area of an SD/MMC storage device as storage. By default the Card mode MMC/SD driver uses the entire available space as storage.

StartSector is relative to the beginning of the SD/MMC storage device. For example, if StartSector is set to 3 than the sectors with the indexes 0, 1, and 2 are not used for storage. The initialization of SD/MMC storage device fails if StartSector is out of range.

If MaxNumSectors is set to 0 the Card mode SD/MMC driver uses for storage the remaining sectors starting from StartSector. If MaxNumSectors is larger than the available number or sectors the actual number of sectors used for storage is limited to the number of sectors available.

6.5.3.1.1.8 FS_MMC_CM_SetHWType()

Description

Configures the HW access routines.

Prototype

```
void FS_MMC_CM_SetHWType(      U8                    Unit,
                         const FS_MMC_HW_TYPE_CM * pHWType);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
pHWType	in Specifies the type of the hardware layer to be used to communicate with the storage device.

Additional information

This function is mandatory. It has to be called in FS_X_AddDevices() once for each instance of the card mode SD/MMC driver. The driver instance is identified by the Unit parameter.

6.5.3.2 Additional driver functions

These functions are optional. They can be used to get information about the operation of the Card Mode MMC/SD driver and to perform additional operations on the used SD card or MMC device.

Function	Description
FS_MMC_CM_EnterPowerSaveMode()	Puts the MMC to sleep in order to save power.
FS_MMC_CM_Erase()	Erases the contents of one or more logical sectors.
FS_MMC_CM_GetCardId()	Returns the identification data of the SD/MMC device.
FS_MMC_CM_GetCardInfo()	This function returns information about the SD/MMC device.
FS_MMC_CM_GetStatCounters()	Returns the value of statistical counters.
FS_MMC_CM_ReadExtCSD()	Reads the contents of the EXT_CSD register of an MMC device.
FS_MMC_CM_ResetStatCounters()	Sets to 0 all statistical counters.
FS_MMC_CM_UnlockCardForced()	Unlocks an SD card.
FS_MMC_CM_WriteExtCSD()	Writes to the EXT_CSD register of the MMC device.

6.5.3.2.1 FS_MMC_CM_EnterPowerSaveMode()

Description

Puts the MMC to sleep in order to save power.

Prototype

```
int FS_MMC_CM_EnterPowerSaveMode(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).

Return value

= 0 The eMMC device is in Sleep state.
≠ 0 An error occurred.

Additional information

This function is optional. It can be used to explicitly put the eMMC device into Sleep state in order to reduce power consumption.

This function is active only if the sources are compiled with the define FS_MMC_SUPPORT_POWER_SAVE set to 1.

6.5.3.2.2 FS_MMC_CM_Erase()

Description

Erases the contents of one or more logical sectors.

Prototype

```
int FS_MMC_CM_Erase(U8  Unit,
                    U32 StartSector,
                    U32 NumSectors);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
StartSector	Index of the first sector to be erased.
NumSectors	Number of sector to be erased.

Return value

= 0 Sectors erased.
≠ 0 An error has occurred.

Additional information

This function is optional. The application can use it to set the contents of the specified logical sectors to a predefined value. The erase operation sets all the bits in the specified logical sectors either to 1 or to 0. The actual value is implementation defined in the EXT_CSD register.

The erase operation is supported only for MMC devices.

6.5.3.2.3 FS_MMC_CM_GetCardId()

Description

Returns the identification data of the SD/MMC device.

Prototype

```
int FS_MMC_CM_GetCardId(U8            Unit,
                        MMC_CARD_ID * pCardId);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
pCardId	out Card identification data.

Return value

= 0 CardId has been read.
≠ 0 An error has occurred.

Additional information

This function is optional. The application can call this function to get the information stored in the CID register of an MMC or SD card. The CID register stores information which can be used to uniquely identify the card such as serial number, product name, manufacturer id, etc. For more information about the information stored in this register refer to SD or MMC specification.

Example

The following example shows how to read and decode the contents of the CID register.

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

void SampleMMC_CMGetCardId(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_MEMSET(&CardId, 0, sizeof(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);
}
```


6.5.3.2.4 FS_MMC_CM_GetCardInfo()

Description

This function returns information about the SD/MMC device.

Prototype

```
int FS_MMC_CM_GetCardInfo(U8                 Unit,
                          FS_MMC_CARD_INFO * pCardInfo);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
pCardInfo	out Information about the device.

Return value

= 0 Card information returned.
≠ 0 An error has occurred.

Additional information

This function is optional. It can be used to get information about the type of the storage card used, about how many data lines are used for the data transfer, etc.

Example

The following example shows how to get information about the storage device.

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

void SampleMMC_CMGetCardInfo(void) {
  FS_MMC_CARD_INFO   CardInfo;
  const char       * pCardType;

  FS_MEMSET(&CardInfo, 0, sizeof(CardInfo));
  FS_MMC_CM_GetCardInfo(0, &CardInfo);
  switch (CardInfo.CardType) {
  case FS_MMC_CARD_TYPE_MMC:
    pCardType = "MMC";
    break;
  case FS_MMC_CARD_TYPE_SD:
    pCardType = "SD";
    break;
  case FS_MMC_CARD_TYPE_UNKNOWN:
    // through
  default:
    pCardType = "UNKNOWN";
    break;
  }
  printf("  Card type:        %s\n"
         "  Bus width:        %d line(s)\n"
         "  Write protected:  %s\n"
         "  High speed mode:  %s\n"
         "  Bytes per sector: %d\n"
         "  Num sectors:      %lu\n", pCardType,
                                     CardInfo.BusWidth,
                                     CardInfo.IsWriteProtected ? "yes" : "no",
                                     CardInfo.IsHighSpeedMode ? "yes" : "no",
                                     (int)CardInfo.BytesPerSector,
                                     CardInfo.NumSectors);
}
```


6.5.3.2.5 FS_MMC_CM_GetStatCounters()

Description

Returns the value of statistical counters.

Prototype

```
void FS_MMC_CM_GetStatCounters(U8                    Unit,
                               FS_MMC_STAT_COUNTERS * pStat);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
pStat	out Current value of statistical counters.

Additional information

This function is optional. The Card mode SD/MMC driver collects statistics about the number of internal operations such as the number of logical sectors read or written by the file system layer. The application can use FS_MMC_CM_GetStatCounters() to get the current value of these counters. The statistical counters are automatically set to 0 when the storage device is mounted or when the application calls FS_MMC_CM_ResetStatCounters().

The statistical counters are available only when the file system is compiled with FS_DEBUG_LEVEL greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL or with FS_MMC_ENABLE_STATS set to 1.

6.5.3.2.6 FS_MMC_CM_ReadExtCSD()

Description

Reads the contents of the EXT_CSD register of an MMC device.

Prototype

```
int FS_MMC_CM_ReadExtCSD(U8    Unit,
                         U32 * pBuffer);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
pBuffer	out Contents of the EXT_CSD register.

Return value

= 0 Register contents returned.
≠ 0 An error has occurred.

Additional information

This function is optional. For more information about the contents of the EXT_CSD register refer to the MMC specification. The contents of the EXT_CSD register can be modified via FS_MMC_CM_WriteExtCSD().

pBuffer has to be 512 at least bytes large.

Example

The following example shows how to read data from the EXT_CSD register of an MMC device.

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

void SampleMMC_CMReadExtCSD(void) {
  U32   aExtCSD[512 / 4];
  U8  * pData8;
  int   r;

  FS_MEMSET(aExtCSD, 0, sizeof(aExtCSD));
  r = FS_MMC_CM_ReadExtCSD(0, aExtCSD);
  if (r == 0) {
    pData8 = (U8 *)aExtCSD;
    printf("POWER_CLASS: %d\n"
           "BUS_WIDTH:   %d\n"
           "HS_TIMING:   %d\n", (int)pData8[187],
                                (int)pData8[183],
                                (int)pData8[185]);
  }
}
```


6.5.3.2.7 FS_MMC_CM_ResetStatCounters()

Description

Sets to 0 all statistical counters.

Prototype

```
void FS_MMC_CM_ResetStatCounters(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).

Additional information

This function is optional. The statistical counters are automatically set to 0 when the storage device is mounted. The application can use FS_MMC_CM_ResetStatCounters() at any time during the file system operation. The statistical counters can be queried via FS_MMC_CM_GetStatCounters().

The statistical counters are available only when the file system is compiled with FS_DEBUG_LEVEL greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL or with FS_MMC_ENABLE_STATS set to 1.

6.5.3.2.8 FS_MMC_CM_UnlockCardForced()

Description

Unlocks an SD card.

Prototype

```
int FS_MMC_CM_UnlockCardForced(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).

Return value

= 0 OK, the SD card has been erased.
≠ 0 An error has occurred.

Additional information

This function is optional. SD cards can be locked with a password in order to prevent inadvertent access to sensitive data. It is not possible to access the data on a locked SD card without knowing the locking password. The application can use `FS_MMC_CM_UnlockCardForced()` to make locked SD card accessible again. The unlocking operation erases all the data stored on the SD card including the lock password.

6.5.3.2.9 FS_MMC_CM_WriteExtCSD()

Description

Writes to the EXT_CSD register of the MMC device.

Prototype

```
int FS_MMC_CM_WriteExtCSD(      U8         Unit,
                                unsigned   Off,
                          const U8       * pData,
                                unsigned   NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
Off	Byte offset in the EXT_CSD register.
pData	out Register contents.
NumBytes	Number of bytes in pData buffer.

Return value

= 0 Register contents returned
≠ 0 An error has occurred

Additional information

This function is optional. Only the byte range 0-191 of the EXT_CSD is modifiable. For more information about the contents of the EXT_CSD register refer to the MMC specification. The contents of the EXT_CSD register can be read via FS_MMC_CM_ReadExtCSD().

6.5.3.2.10 FS_MMC_CARD_ID

Description

Information about the storage card.

Type definition

```
typedef struct {
  U8  aData[];
} FS_MMC_CARD_ID;
```

Structure members

Member	Description
aData	Indentification information as returned by the SD card or MMC device as response to CMD10 command.

Additional information

The card identification can be read via `FS_MMC_GetCardId()` or `FS_MMC_CM_GetCardId()`

6.5.3.2.11 FS_MMC_CARD_INFO

Description

Information about the storage device.

Type definition

```
typedef struct {
  U8   CardType;
  U8   BusWidth;
  U8   IsWriteProtected;
  U8   IsHighSpeedMode;
  U16  BytesPerSector;
  U32  NumSectors;
} FS_MMC_CARD_INFO;
```

Structure members

Member	Description
CardType	Type of the storage card.
BusWidth	Number of data lines used for the data transfer.
IsWriteProtected	Set to 1 if the card is write protected.
IsHighSpeedMode	Set to 1 if the card operates in the high-speed mode.
BytesPerSector	Number of bytes in a sector.
NumSectors	Total number of sectors on the card.

Additional information

For a list of permitted values for CardType refer to *Storage card types* on page 777. BusWidth can take one of these values: 1, 4 or 8. BytesPerSector is typically 512 bytes.

6.5.3.2.12 FS_MMC_STAT_COUNTERS

Description

Statistical counters maintained by the MMC/SD driver

Type definition

```
typedef struct {
  U32  WriteSectorCnt;
  U32  WriteErrorCnt;
  U32  ReadSectorCnt;
  U32  ReadErrorCnt;
  U32  CmdExecCnt;
} FS_MMC_STAT_COUNTERS;
```

Structure members

Member	Description
WriteSectorCnt	Number of logical sectors read.
WriteErrorCnt	Number of write errors.
ReadSectorCnt	Number of logical sectors written.
ReadErrorCnt	Number of read errors.
CmdExecCnt	Number of commands executed.

6.5.3.2.13 Storage card types

Description

Type of storage devices supported by the MMC/SD driver.

Definition

```
#define FS_MMC_CARD_TYPE_UNKNOWN    0
#define FS_MMC_CARD_TYPE_MMC        1
#define FS_MMC_CARD_TYPE_SD         2
```

Symbols

Definition	Description
FS_MMC_CARD_TYPE_UNKNOWN	The driver was not able to identify the device.
FS_MMC_CARD_TYPE_MMC	The storage device conforms to MMC specification.
FS_MMC_CARD_TYPE_SD	The storage device conforms to SD specification.

6.5.3.3 Performance and resource usage

6.5.3.3.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the Card Mode MMC/SD driver was measured using the SEGGER Embedded Studio IDE V4.20 configured to generate code for a Cortex-M4 CPU in Thumb mode and with the size optimization enabled.

Usage: 3.9 Kbytes

6.5.3.3.2 Static RAM usage

Static RAM usage refers to the amount of RAM required by the Card Mode MMC/SD driver internally for all the driver instances. The number of bytes can be seen in the compiler list file of the `FS_MMC_CM_Drv.c` file.

Usage: 12 bytes

6.5.3.3.3 Dynamic RAM usage

Dynamic RAM usage is the amount of RAM allocated by the driver at runtime. The amount of RAM required depends on the compile time and runtime configuration.

Usage: 42 bytes

6.5.3.3.4 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. All values are given in Mbytes/second.

CPU type	MMC/SD device	Write speed	Read speed
Atmel AT91SAM9263 (200 MHz)	Ultron 256MB SD card	**10.0**	**9.3**
NXP LPC2478 (57 MHz)	Ultron 256MB SD card	**2.4**	**3.1**
NXP LPC3250 (208 MHz)	Ultron 256MB SD card	**3.9**	**8.4**
NXP K66 (168 MHz)	SanDisk 16GB Ultra PLUS SD card	**3.6**	**11.1**
Microchip SAME70 (300 MHz)	SanDisk 16GB Ultra PLUS SD card	**11.1**	**20.4**
NXP iMX6U5	SanDisk 16GB Ultra PLUS SD card	**15.1**	**18.4**
NXP iMXRT1052	SanDisk 16GB Ultra PLUS SD card	**16.9**	**19.5**
NPX LPC54608	SanDisk 16GB Ultra PLUS SD card	**11.6**	**18.4**
Renesas RZGE1	Silicon Motion SM661G8 eMMC	**5.5**	**16.5**
ST STM32H743	SanDisk 16GB Ultra PLUS SD card	**10.0**	**14.4**
ST STM32F746	Transcend 16GB Class 10 SD card	**12.2**	**18.9**
ST STM32F469	SanDisk 16GB Ultra PLUS SD card	**16.1**	**17.8**

6.5.4 MMC/SD hardware layer

The hardware layer provides the functions required to the MMC/SD driver to access the target hardware via SPI controller, SD host controller, GPIO, etc. The functions of the MMC/SD hardware layer are called by the MMC/SD driver to exchange commands and data with an MMC or SD card. Since these functions are hardware dependent, they have to be implemented by the user. emFile comes with template hardware layers and sample implementations for popular evaluation boards that can be used as starting point for the implementation of new hardware layers. The relevant files are located in the `/Sample/FS/Driver/MMC_SPI` and `/Sample/FS/Driver/MMC_CM` folders of the emFile shipment.

6.5.4.1 Hardware layer types

The functions of the MMC/SD hardware layer are organized in a function table implemented a C structure. Different hardware layer types are provided to support different ways of interfacing an MMC or SD card. The type of hardware layer an application has to use depends on the MMC/SD driver used. The following table lists what hardware layer is required by each MMC/SD driver.

MMC/SD hardware layer	MMC/SD driver
`FS_MMC_HW_TYPE_SPI`	*SPI MMC/SD driver*
`FS_MMC_HW_TYPE_CM`	*Card Mode MMC/SD driver*

6.5.4.2 Hardware layer API - FS_MMC_HW_TYPE_SPI

This hardware layer supports MMC cards that comply with the version 3.x or older of the MMC specification and SD cards of all types that are interfaced to the target MCU via SPI. The functions of this hardware layer are grouped in a structure of type `FS_MMC_HW_TYPE_SPI`. The following sections describe these functions in detail.

6.5.4.2.1 FS_MMC_HW_TYPE_SPI

Description

Hardware layer for the SPI MMC/SD driver.

Type definition

```
typedef struct {
  FS_MMC_HW_TYPE_SPI_ENABLE_CS          * pfEnableCS;
  FS_MMC_HW_TYPE_SPI_DISABLE_CS         * pfDisableCS;
  FS_MMC_HW_TYPE_SPI_IS_PRESENT         * pfIsPresent;
  FS_MMC_HW_TYPE_SPI_IS_WRITE_PROTECTED * pfIsWriteProtected;
  FS_MMC_HW_TYPE_SPI_SET_MAX_SPEED      * pfSetMaxSpeed;
  FS_MMC_HW_TYPE_SPI_SET_VOLTAGE        * pfSetVoltage;
  FS_MMC_HW_TYPE_SPI_READ               * pfRead;
  FS_MMC_HW_TYPE_SPI_WRITE              * pfWrite;
  FS_MMC_HW_TYPE_SPI_READ_EX            * pfReadEx;
  FS_MMC_HW_TYPE_SPI_WRITE_EX           * pfWriteEx;
  FS_MMC_HW_TYPE_SPI_LOCK               * pfLock;
  FS_MMC_HW_TYPE_SPI_UNLOCK             * pfUnlock;
} FS_MMC_HW_TYPE_SPI;
```

Structure members

Member	Description
pfEnableCS	Enables the card.
pfDisableCS	Disables the card.
pfIsPresent	Checks if the card is inserted.
pfIsWriteProtected	Checks if the card is write protected.
pfSetMaxSpeed	Configures the clock frequency supplied to the card.
pfSetVoltage	Configures the voltage level of the card power supply.
pfRead	Transfers data from card to MCU.
pfWrite	Transfers data from MCU to card.
pfReadEx	Transfers data from card to MCU.
pfWriteEx	Transfers data from MCU to card.
pfLock	Requires exclusive access to SPI bus.
pfUnlock	Releases exclusive access to SPI bus.

6.5.4.2.2 FS_MMC_HW_TYPE_SPI_ENABLE_CS

Description

Enables the card.

Type definition

```
typedef void FS_MMC_HW_TYPE_SPI_ENABLE_CS(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)

Additional information

This function is a member of the SPI MMC/SD hardware layer and it has to be implemented by any hardware layer.

The Chip Select (CS) signal is used to address a specific MMC or SD card connected via SPI. Enabling the card is equal to setting the CS signal to a logic low level.

6.5.4.2.3 FS_MMC_HW_TYPE_SPI_DISABLE_CS

Description

Disables the card.

Type definition

```
typedef void FS_MMC_HW_TYPE_SPI_DISABLE_CS(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)

Additional information

This function is a member of the SPI MMC/SD hardware layer and it has to be implemented by any hardware layer.

The Chip Select (CS) signal is used to address a specific MMC or SD card connected via SPI. Disabling the card is equal to setting the CS signal to a logic high level.

6.5.4.2.4 FS_MMC_HW_TYPE_SPI_IS_PRESENT

Description

Checks if the card is present.

Type definition

```
typedef int FS_MMC_HW_TYPE_SPI_IS_PRESENT(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)

Return value

FS_MEDIA_STATE_UNKNOWN	The presence state of the card is unknown.
FS_MEDIA_NOT_PRESENT	The card is present.
FS_MEDIA_IS_PRESENT	The card is not present.

Additional information

This function is a member of the SPI MMC/SD hardware layer and it has to be implemented by any hardware layer.

Typically, the card presence detection is implemented via a dedicated Card Detect (CD) signal. This signal is connected to a switch inside the card slot that changes its state each time the card is removed or inserted. If the hardware does not provide such signal the function has to return FS_MEDIA_STATE_UNKNOWN.

6.5.4.2.5 FS_MMC_HW_TYPE_SPI_IS_WRITE_PROTECTED

Description

Checks if the card is write protected.

Type definition

```
typedef int FS_MMC_HW_TYPE_SPI_IS_WRITE_PROTECTED(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)

Return value

= 0 The card data can be modified.
≠ 0 The card data cannot be modified.

Additional information

This function is a member of the SPI MMC/SD hardware layer and it has to be implemented by any hardware layer.

Typically, the card protection status is implemented via a dedicated Write Protected (WP) signal. This signal is connected to a switch inside the card slot that reflects the status of the write protect switch found on SD cards. If the hardware does not provide such signal the function has to return 0.

MMC and micro SD cards do not have a write protection switch. Please note that the write protect switch does not really protect the data on an SD card form being modified. It is merely an indication that the data has to be protected. It is the responsibility of the host MCU to respect the status of this switch.

6.5.4.2.6 FS_MMC_HW_TYPE_SPI_SET_MAX_SPEED

Description

Configures the clock frequency supplied to the card.

Type definition

```
typedef U16 FS_MMC_HW_TYPE_SPI_SET_MAX_SPEED(U8  Unit,
                                             U16 MaxFreq);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
MaxFreq	Maximum allowed frequency of the SPI clock in kHz.

Return value

≠ 0 The actual clock frequency in kHz.
= 0 An error occurred.

Additional information

This function is a member of the SPI MMC/SD hardware layer and it has to be implemented by any hardware layer.

The hardware layer is allowed to set a clock frequency smaller than the value specified via MaxFreq but never greater than that. The SPI MMC/SD driver calls this function at least two times during the initialization of the MMC or SD card:

- Before the initialization starts to set the clock frequency to 400 kHz.
- After the card identification to set the standard clock frequency which is 25 MHz for SD cards or 20 MHz for MMC cards.

The function has to return the actual configured clock frequency. If the precise frequency is unknown such as for implementation using I/O port “bit-banging”, the return value has to be less than the specified frequency. This may lead to longer timeout values but is in general does not cause any problems. The SPI MMC/SD driver uses the returned value to calculate timeout values.

6.5.4.2.7 FS_MMC_HW_TYPE_SPI_SET_VOLTAGE

Description

Configures the voltage level of the card power supply.

Type definition

```
typedef int FS_MMC_HW_TYPE_SPI_SET_VOLTAGE(U8  Unit,
                                           U16 Vmin,
                                           U16 Vmax);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
Vmin	Minimum supply voltage level in mV.
Vmax	Maximum supply voltage level in mV.

Return value

1 Card slot supports the voltage range.
0 Card slot does not support the voltage range.

Additional information

This function is a member of the SPI MMC/SD hardware layer and it has to be implemented by any hardware layer.

By default, all cards work with the initial voltage of 3.3V. If hardware layer has to save power, then the supply voltage can be adjusted within the specified range.

6.5.4.2.8 FS_MMC_HW_TYPE_SPI_READ

Description

Transfers data from card to MCU.

Type definition

```
typedef void FS_MMC_HW_TYPE_SPI_READ(U8   Unit,
                                     U8 * pData,
                                     int  NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
pData	out Data transfered from card.
NumBytes	Number of bytes to be transfered.

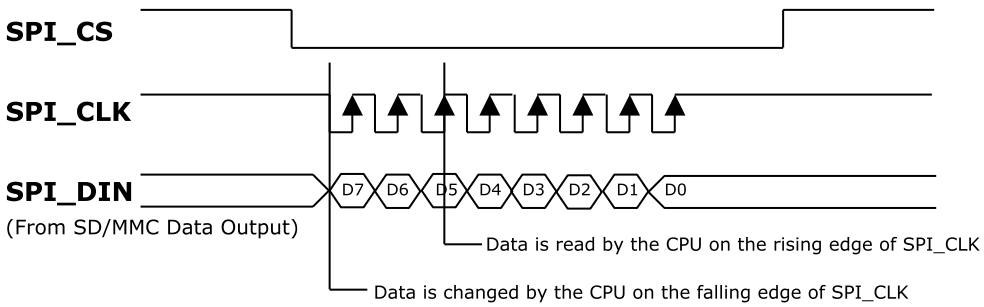
Additional information

This function is a member of the SPI MMC/SD hardware layer. FS_MMC_HW_TYPE_SPI_READ does not have to be implemented if the hardware layer provides an implementation for FS_MMC_HW_TYPE_SPI_READ_EX.

The data is received via the DIN (MISO) signal of the SPI interface with the data being sampled on the rising edge of the clock signal. According to the SD specification the DOUT (MOSI) signal must be driven to logic high during the data transfer, otherwise the SD card will not work properly.

The SPI MMC/SD driver calls this function only when the card is enabled that is with the CS signal at logic low level.

The following picture shows the waveforms of the SPI signals during the transfer of 8 bits from card to MCU.



6.5.4.2.9 FS_MMC_HW_TYPE_SPI_WRITE

Description

Transfers data from MCU to card.

Type definition

```
typedef void FS_MMC_HW_TYPE_SPI_WRITE(      U8   Unit,
                                      const U8 * pData,
                                            int  NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
pData	in Data to be transfered to card.
NumBytes	Number of bytes to be transfered to card.

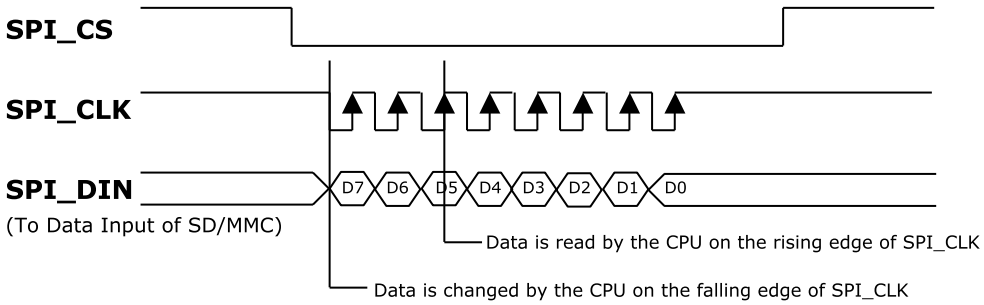
Additional information

This function is a member of the SPI MMC/SD hardware layer. FS_MMC_HW_TYPE_SPI_WRITE does not have to be implemented if the hardware layer provides an implementation for FS_MMC_HW_TYPE_SPI_WRITE_EX.

The data is sent via the DOUT (MOSI) signal of the SPI interface with the data being changed on the falling edge of the clock signal.

The SPI MMC/SD driver also calls this function when the card is not enabled that is with the CS signal at logic low in order to generate empty cycles.

The following picture shows the waveforms of the SPI signals during the transfer of 8 bits from MCU to card.



6.5.4.2.10 FS_MMC_HW_TYPE_SPI_READ_EX

Description

Transfers data from card to MCU.

Type definition

```
typedef int FS_MMC_HW_TYPE_SPI_READ_EX(U8   Unit,
                                       U8 * pData,
                                       int  NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
pData	out Data transfered from card.
NumBytes	Number of bytes to be transfered.

Return value

= 0 OK, data transferred.
≠ 0 An error occurred.

Additional information

This function is a member of the SPI MMC/SD hardware layer. FS_MMC_HW_TYPE_SPI_READ_EX does not have to be implemented if the hardware layer provides an implementation for FS_MMC_HW_TYPE_SPI_READ. It provides the same functionality as FS_MMC_HW_TYPE_SPI_READ with the difference that it can report the result of the operation to the SPI MMC/SD driver.

The SPI MMC/SD driver calls this function only when the card is enabled that is with the CS signal at logic low level.

6.5.4.2.11 FS_MMC_HW_TYPE_SPI_WRITE_EX

Description

Transfers data from MCU to card.

Type definition

```
typedef int FS_MMC_HW_TYPE_SPI_WRITE_EX(      U8   Unit,
                                        const U8 * pData,
                                              int  NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
pData	in Data to be transfered to card.
NumBytes	Number of bytes to be transfered to card.

Return value

= 0 OK, data transferred.
≠ 0 An error occurred.

Additional information

This function is a member of the SPI MMC/SD hardware layer. FS_MMC_HW_TYPE_SPI_WRITE_EX does not have to be implemented if the hardware layer provides an implementation for FS_MMC_HW_TYPE_SPI_WRITE. It provides the same functionality as FS_MMC_HW_TYPE_SPI_WRITE with the difference that it can report the result of the operation to the SPI MMC/SD driver.

The SPI MMC/SD driver also calls this function when the card is not enabled that is with the CS signal at logic low in order to generate empty cycles.

6.5.4.2.12 FS_MMC_HW_TYPE_SPI_LOCK

Description

Requires exclusive access to SPI bus.

Type definition

```
typedef void FS_MMC_HW_TYPE_SPI_LOCK(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)

Additional information

This function is a member of the SPI MMC/SD hardware layer. The implementation of this function is optional.

The SPI MMC/SD driver calls this function to indicate that it needs exclusive to access the card via the SPI bus. It is guaranteed that the SPI MMC/SD driver does not attempt to exchange any data with the card via the SPI bus before calling this function first. It is also guaranteed that FS_MMC_HW_TYPE_SPI_LOCK and FS_MMC_HW_TYPE_SPI_UNLOCK are called in pairs. Typically, this function is used for synchronizing the access to SPI bus when the SPI bus is shared between the card and other SPI devices.

A possible implementation would make use of an OS semaphore that is acquired in FS_MMC_HW_TYPE_SPI_LOCK and released in FS_MMC_HW_TYPE_SPI_UNLOCK.

FS_MMC_HW_TYPE_SPI_LOCK is called only when the SPI MMC/SD driver is compiled with FS_MMC_SUPPORT_LOCKINGset to 1.

6.5.4.2.13 FS_MMC_HW_TYPE_SPI_UNLOCK

Description

Releases exclusive access to SPI bus.

Type definition

```
typedef void FS_MMC_HW_TYPE_SPI_UNLOCK(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)

Additional information

This function is a member of the SPI MMC/SD hardware layer. The implementation of this function is optional.

The SPI MMC/SD driver calls this function when it no longer needs to access the card via the SPI bus. It is guaranteed that the SPI MMC/SD driver does not attempt to exchange any data with the card via the SPI bus before calling FS_MMC_HW_TYPE_SPI_LOCK. It is also guaranteed that FS_MMC_HW_TYPE_SPI_UNLOCK and FS_MMC_HW_TYPE_SPI_LOCK are called in pairs.

FS_MMC_HW_TYPE_SPI_UNLOCK and FS_MMC_HW_TYPE_SPI_LOCK can be used to synchronize the access to the SPI bus when other devices than the serial NAND flash are connected to it. A possible implementation would make use of an OS semaphore that is acquired FS_MMC_HW_TYPE_SPI_LOCK and released in FS_MMC_HW_TYPE_SPI_UNLOCK.

FS_MMC_HW_TYPE_SPI_LOCK is called only when the SPI MMC/SD driver is compiled with FS_MMC_SUPPORT_LOCKINGset to 1.

6.5.4.2.14 Sample implementation

The following sample implementation uses the SPI controller of an TI TM4C129 MCU to interface with an SD card. This hardware layer was tested on the TI TIVAC129 evaluation board.

```
/*********************************************************************
*                  (c) SEGGER Microcontroller GmbH                   *
*                       The Embedded Experts                         *
*                          www.segger.com                            *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File        : FS_MMC_HW_SPI_TM4C129_TI_TIVAC129.c
Purpose     : Generic HW layer for the MMC / SD driver working in SPI mode.
Literature  :
  [1] Tiva TM4C129XNCZAD Microcontroller DATA SHEET
    (\\fileserver\Techinfo\Company\TI\MCU\TM4C\TM4C129\tm4c129xnczad.pdf)
  [2] Tiva TM4C129X Development Board User's Guide
    (\\fileserver\Techinfo\Company\TI\MCU\TM4C\Evalboard\Tiva_C_series\spmu360.pdf)

Additional information
  The SD card and NOR flash device share the same CS signal. In order to
  be able to work with the SD card, the jumper J7 has to be configured
  according to "2.1.10 EEPROM and SD Card" in [2].
*/

/*********************************************************************
*
*       #include Section
*
**********************************************************************
*/
#include "FS.h"
#include "FS_Int.h"

/*********************************************************************
*
*       Defines, configurable
*
**********************************************************************
*/
#ifndef   FS_MMC_HW_SPI_PERIPH_CLOCK_KHZ
  #define FS_MMC_HW_SPI_PERIPH_CLOCK_KHZ            120000uL
  // Clock of SSI peripheral in kHz
#endif

#ifndef   FS_MMC_HW_SPI_MAX_SPI_CLOCK_KHZ
  #define FS_MMC_HW_SPI_MAX_SPI_CLOCK_KHZ           24000uL
  // Maximum clock frequency supplied to SD card
#endif

#ifndef   FS_MMC_HW_SPI_WAIT_TIMEOUT_CYCLES
  #define FS_MMC_HW_SPI_WAIT_TIMEOUT_CYCLES         100000uL
  // Time to wait for an operation to finish (software loops)
#endif

#ifndef   FS_MMC_HW_SPI_USE_DMA
  #define FS_MMC_HW_SPI_USE_DMA                     0
  // Enables/disables the data transfer via DMA
#endif

#ifndef   FS_MMC_HW_SPI_DEFAULT_SUPPLY_VOLTAGE
  #define FS_MMC_HW_SPI_DEFAULT_SUPPLY_VOLTAGE      3300        // in mV, example means 3.3V
#endif

/*********************************************************************
*
*       Defines, fixed
*
**********************************************************************
*/

/*********************************************************************
```



```
*
*       SSI registers
*/
#define SSI_BASE_ADDR             0x4000B000uL
#define SSICR0                    (*(volatile U32 *)(SSI_BASE_ADDR + 0x000))
  // SSI Control 0
#define SSICR1                    (*(volatile U32 *)(SSI_BASE_ADDR + 0x004))
  // SSI Control 1
#define SSIDR                     (*(volatile U32 *)(SSI_BASE_ADDR + 0x008))
  // SSI Data
#define SSISR                     (*(volatile U32 *)(SSI_BASE_ADDR + 0x00C))
  // SSI Status
#define SSICPSR                   (*(volatile U32 *)(SSI_BASE_ADDR + 0x010))
  // SSI Clock Prescale
#define SSIIM                     (*(volatile U32 *)(SSI_BASE_ADDR + 0x014))
  // SSI Interrupt Mask
#define SSIRIS                    (*(volatile U32 *)(SSI_BASE_ADDR + 0x018))
  // SSI Raw Interrupt Status
#define SSIMIS                    (*(volatile U32 *)(SSI_BASE_ADDR + 0x01C))
  // SSI Masked Interrupt Status
#define SSIICR                    (*(volatile U32 *)(SSI_BASE_ADDR + 0x020))
  // SSI Interrupt Clear
#define SSIDMACTL                 (*(volatile U32 *)(SSI_BASE_ADDR + 0x024))
  // SSI DMA Control
#define SSICC                     (*(volatile U32 *)(SSI_BASE_ADDR + 0xFC8))
  // SSI Clock Configuration

/*********************************************************************
*
*       System control registers
*/
#define SYS_BASE_ADDR             0x400FE000uL
#define RCGCGPIO                  (*(volatile U32 *)(SYS_BASE_ADDR + 0x608))
  // GPIO Run Mode Clock Gating Control
#define RCGCDMA                   (*(volatile U32 *)(SYS_BASE_ADDR + 0x60C))
  // DMA Run Mode Clock Gating Control
#define RCGCSSI                   (*(volatile U32 *)(SYS_BASE_ADDR + 0x61C))
  // SSI Run Mode Clock Gating Control
#define PRGPIO                    (*(volatile U32 *)(SYS_BASE_ADDR + 0xA08))
  // GPIO Run Mode Peripheral Ready
#define PRDMA                     (*(volatile U32 *)(SYS_BASE_ADDR + 0xA0C))
  // DMA Run Mode Peripheral Ready
#define PRSSI                     (*(volatile U32 *)(SYS_BASE_ADDR + 0xA1C))
  // SSI Run Mode Peripheral Ready

/*********************************************************************
*
*       GPIO Port F
*/
#define PF_BASE_ADDR              0x4005D000uL
#define PF_GPIOAFSEL              (*(volatile U32 *)(PF_BASE_ADDR + 0x420))
  // GPIO Alternate Function Select
#define PF_GPIODR2R               (*(volatile U32 *)(PF_BASE_ADDR + 0x500))       // GPIO
 2-mA Drive Select
#define PF_GPIODR4R               (*(volatile U32 *)(PF_BASE_ADDR + 0x504))       // GPIO
 4-mA Drive Select
#define PF_GPIODR8R               (*(volatile U32 *)(PF_BASE_ADDR + 0x508))       // GPIO
 8-mA Drive Select
#define PF_GPIOODR                (*(volatile U32 *)(PF_BASE_ADDR + 0x50C))
  // GPIO Open Drain Select
#define PF_GPIOPUR                (*(volatile U32 *)(PF_BASE_ADDR + 0x510))
  // GPIO Pull-Up Select
#define PF_GPIOPDR                (*(volatile U32 *)(PF_BASE_ADDR + 0x514))
  // GPIO Pull-Down Select
#define PF_GPIODEN                (*(volatile U32 *)(PF_BASE_ADDR + 0x51C))
  // GPIO Digital Enable
#define PF_GPIOPCTL               (*(volatile U32 *)(PF_BASE_ADDR + 0x52C))
  // GPIO Port Control
#define PF_GPIOPC                 (*(volatile U32 *)(PF_BASE_ADDR + 0xFC4))
  // GPIO Peripheral Configuration

/*********************************************************************
*
*       GPIO Port H
*/
#define PH_BASE_ADDR              0x4005F000uL
```



```
#define PH_GPIODATA               (*(volatile U32 *)(PH_BASE_ADDR + 0x000))
  // GPIO Data
#define PH_GPIODIR                (*(volatile U32 *)(PH_BASE_ADDR + 0x400))
  // GPIO Direction
#define PH_GPIOAFSEL              (*(volatile U32 *)(PH_BASE_ADDR + 0x420))
  // GPIO Alternate Function Select
#define PH_GPIODR2R               (*(volatile U32 *)(PH_BASE_ADDR + 0x500))       // GPIO
 2-mA Drive Select
#define PH_GPIODR4R               (*(volatile U32 *)(PH_BASE_ADDR + 0x504))       // GPIO
 4-mA Drive Select
#define PH_GPIODR8R               (*(volatile U32 *)(PH_BASE_ADDR + 0x508))       // GPIO
 8-mA Drive Select
#define PH_GPIOODR                (*(volatile U32 *)(PH_BASE_ADDR + 0x50C))
  // GPIO Open Drain Select
#define PH_GPIOPUR                (*(volatile U32 *)(PH_BASE_ADDR + 0x510))
  // GPIO Pull-Up Select
#define PH_GPIOPDR                (*(volatile U32 *)(PH_BASE_ADDR + 0x514))
  // GPIO Pull-Down Select
#define PH_GPIODEN                (*(volatile U32 *)(PH_BASE_ADDR + 0x51C))
  // GPIO Digital Enable
#define PH_GPIOPCTL               (*(volatile U32 *)(PH_BASE_ADDR + 0x52C))
  // GPIO Port Control
#define PH_GPIOPC                 (*(volatile U32 *)(PH_BASE_ADDR + 0xFC4))
  // GPIO Peripheral Configuration

/*********************************************************************
*
*       GPIO Port Q
*/
#define PQ_BASE_ADDR              0x40066000uL
#define PQ_GPIODATA               (*(volatile U32 *)(PQ_BASE_ADDR + 0x000))
  // GPIO Data
#define PQ_GPIODIR                (*(volatile U32 *)(PQ_BASE_ADDR + 0x400))
  // GPIO Direction
#define PQ_GPIOAFSEL              (*(volatile U32 *)(PQ_BASE_ADDR + 0x420))
  // GPIO Alternate Function Select
#define PQ_GPIODR2R               (*(volatile U32 *)(PQ_BASE_ADDR + 0x500))       // GPIO
 2-mA Drive Select
#define PQ_GPIODR4R               (*(volatile U32 *)(PQ_BASE_ADDR + 0x504))       // GPIO
 4-mA Drive Select
#define PQ_GPIODR8R               (*(volatile U32 *)(PQ_BASE_ADDR + 0x508))       // GPIO
 8-mA Drive Select
#define PQ_GPIOODR                (*(volatile U32 *)(PQ_BASE_ADDR + 0x50C))
  // GPIO Open Drain Select
#define PQ_GPIOPUR                (*(volatile U32 *)(PQ_BASE_ADDR + 0x510))
  // GPIO Pull-Up Select
#define PQ_GPIOPDR                (*(volatile U32 *)(PQ_BASE_ADDR + 0x514))
  // GPIO Pull-Down Select
#define PQ_GPIODEN                (*(volatile U32 *)(PQ_BASE_ADDR + 0x51C))
  // GPIO Digital Enable
#define PQ_GPIOPCTL               (*(volatile U32 *)(PQ_BASE_ADDR + 0x52C))
  // GPIO Port Control
#define PQ_GPIOPC                 (*(volatile U32 *)(PQ_BASE_ADDR + 0xFC4))
  // GPIO Peripheral Configuration

/*********************************************************************
*
*       DMA controller
*/
#define DMA_BASE_ADDR             0x400FF000uL
#define DMACFG                    (*(volatile U32 *)(DMA_BASE_ADDR + 0x004))
  // DMA configuration
#define DMACTLBASE                (*(volatile U32 *)(DMA_BASE_ADDR + 0x008))
  // DMA Channel Control Base Pointer
#define DMAUSEBURSTCLR            (*(volatile U32 *)(DMA_BASE_ADDR + 0x01C))
  // DMA Channel Useburst Clear
#define DMAREQMASKCLR             (*(volatile U32 *)(DMA_BASE_ADDR + 0x024))
  // DMA Channel Request Mask Clear
#define DMAENASET                 (*(volatile U32 *)(DMA_BASE_ADDR + 0x028))
  // DMA Channel Enable Set
#define DMAENACLR                 (*(volatile U32 *)(DMA_BASE_ADDR + 0x02C))
  // DMA Channel Enable Clear
#define DMAALTCLR                 (*(volatile U32 *)(DMA_BASE_ADDR + 0x034))
  // DMA Channel Primary Alternate Clear
#define DMAPRIOCLR                (*(volatile U32 *)(DMA_BASE_ADDR + 0x03C))
  // DMA Channel Priority Clear
```



```
#define DMACHMAP0                 (*(volatile U32 *)(DMA_BASE_ADDR + 0x510))
  // DMA Channel Map Select 0
#define DMACHMAP1                 (*(volatile U32 *)(DMA_BASE_ADDR + 0x514))
  // DMA Channel Map Select 1
#define DMACHMAP2                 (*(volatile U32 *)(DMA_BASE_ADDR + 0x518))
  // DMA Channel Map Select 2
#define DMACHMAP3                 (*(volatile U32 *)(DMA_BASE_ADDR + 0x51C))
  // DMA Channel Map Select 3

/*********************************************************************
*
*       Flash controller
*/
#define FLASH_BASE_ADDR           0x400FD000uL
#define FLASHPP                   (*(volatile U32 *)(FLASH_BASE_ADDR + 0xFC0))
  // Flash Peripheral Properties
#define FLASHDMASZ                (*(volatile U32 *)(FLASH_BASE_ADDR + 0xFD0))
  // Flash DMA Address Size
#define FLASHDMAST                (*(volatile U32 *)(FLASH_BASE_ADDR + 0xFD4))
  // Flash DMA Starting Address

/*********************************************************************
*
*       Pin assignment
*/
#define CS_PIN                    4     // Port H
#define CLK_PIN                   0     // Port Q
#define DAT0_PIN                  2     // Port Q
#define DAT1_PIN                  0     // Port F

/*********************************************************************
*
*       Flags in system registers
*/
#define RCGCSSI_SSI3              3
#define RCGCGPIO_GPIOF            5
#define RCGCGPIO_GPIOH            7
#define RCGCGPIO_GPIOQ            14

/*********************************************************************
*
*       SSI related defines
*/
#define CR0_DSS                   0
#define CR0_DSS_8BIT              7uL
#define CR0_SPO                   6
#define CR0_SCR                   8
#define CR0_SPH                   7
#define CR0_SCR_MASK              0xFFuL
#define CR1_SSE                   1
#define CR1_MODE                  6
#define CR1_MODE_BI               1uL
#define CR1_MODE_QUAD             2uL
#define CR1_MODE_ADV              3uL
#define CR1_MODE_MASK             3uL
#define CR1_DIR                   8
#define SR_TNF                    1
#define SR_RNE                    2
#define SR_BSY                    4

/*********************************************************************
*
*       DMA related defines
*/
#define DMA_MAX_TRANSFER_SIZE     1024    // in items
#define DMA_CHANNEL_READ          14
#define DMA_CHANNEL_WRITE         15
#define CHMAP1_SSI3_RX            2uL
#define CHMAP1_SSI3_TX            2uL
#define CHMAP_CHSEL_MASK          0xFuL
#define CHCTL_XFERMODE            0
#define CHCTL_XFERMODE_BASIC      1uL
#define CHCTL_XFERSIZE            4
#define CHCTL_XFERSIZE_MASK       0x3FFuL
#define CHCTL_ARBSIZE             14
#define CHCTL_ARBSIZE_4           2uL
```



```
#define CHCTL_SRCINC              26
#define CHCTL_SRCINC_NOINC        3uL
#define CHCTL_DSTINC              30
#define CHCTL_DSTINC_NOINC        3uL
#define DMACTL_RXDMAE             0
#define DMACTL_TXDMAE             1
#define CFG_MASTEREN              0
#define PP_DFA                    28

/*********************************************************************
*
*       Misc. defines
*/
#define DEFAULT_SPI_CLOCK_KHZ     400uL
#if (DMA_CHANNEL_READ < DMA_CHANNEL_WRITE)
  #define NUM_DMA_DESC            (DMA_CHANNEL_WRITE + 1)
#else
  #define NUM_DMA_DESC            (DMA_CHANNEL_READ + 1)
#endif

/*********************************************************************
*
*       Local data types
*
*********************************************************************
*/

/*********************************************************************
*
*       DMA_DESC
*/
typedef struct DMA_DESC {
  volatile U32 SRCENDP;
  volatile U32 DESTENDP;
  volatile U32 CHCTL;
  volatile U32 Reserved;
} DMA_DESC;

/*********************************************************************
*
*       Static data
*
*********************************************************************
*/
static U8            _IsInited;
#if FS_MMC_HW_SPI_USE_DMA
  //
  // The DMA descriptors have to aligned on a 1KB boundary.
  //
#ifdef __ICCARM__
  #pragma data_alignment=1024
  static DMA_DESC    _aDMADesc[NUM_DMA_DESC];
#endif
#ifdef __SES_ARM
  static DMA_DESC    _aDMADesc[NUM_DMA_DESC] __attribute__ ((aligned (1024)));
#endif
  static volatile U8 _Dummy;
#endif

/*********************************************************************
*
*       Static code
*
*********************************************************************
*/

/*********************************************************************
*
*       _EnableClocks
*/
static void _EnableClocks(void) {
  U32 TimeOut;

  //
  // Provide a clock to SSI module and wait for it to become ready.
  //
```



```
  RCGCSSI |= 1uL << RCGCSSI_SSI3;
  TimeOut = FS_MMC_HW_SPI_WAIT_TIMEOUT_CYCLES;
  while (1) {
    if (PRSSI & (1uL << RCGCSSI_SSI3)) {
      break;
    }
    if (--TimeOut == 0) {
      break;
    }
  }
  //
  // Provide a clock to GPIO ports and wait for them to become ready.
  //
  RCGCGPIO |= 0
           | (1uL << RCGCGPIO_GPIOF)
           | (1uL << RCGCGPIO_GPIOH)
           | (1uL << RCGCGPIO_GPIOQ)
           ;
  TimeOut = FS_MMC_HW_SPI_WAIT_TIMEOUT_CYCLES;
  while (1) {

  if (PRGPIO & ((1uL << RCGCGPIO_GPIOF) | (1uL << RCGCGPIO_GPIOQ) | (1uL << RCGCGPIO_GPIOH))) {
      break;
    }
    if (--TimeOut == 0) {
      break;
    }
  }
#if FS_MMC_HW_SPI_USE_DMA
  //
  // Provide a clock to DMA controller and wait for it to become ready.
  //
  RCGCDMA |= 1uL << 0;
  TimeOut = FS_MMC_HW_SPI_WAIT_TIMEOUT_CYCLES;
  while (1) {
    if (PRDMA & ((1uL << 0))) {
      break;
    }
    if (--TimeOut == 0) {
      break;
    }
  }
#endif
}

/*********************************************************************
*
*       _InitPins
*/
static void _InitPins(void) {
  //
  // Give the control of CLK, DAT0-3 signals to SSI module.
  //
  PQ_GPIOAFSEL |= 0
               |  (1uL << CLK_PIN)
               |  (1uL << DAT0_PIN)
               ;
  PF_GPIOAFSEL |= 0
               |  (1uL << DAT1_PIN)
               ;
  PQ_GPIOPCTL  &= ~((0xFuL << (CLK_PIN  << 2)) |
                    (0xFuL << (DAT0_PIN << 2)));
  PQ_GPIOPCTL  |= 0
               |  (0xEuL << (CLK_PIN  << 2))
               |  (0xEuL << (DAT0_PIN << 2))
               ;
  PF_GPIOPCTL  &= ~((0xFuL << (DAT1_PIN << 2)));
  PF_GPIOPCTL  |= 0
               |  (0xEuL << (DAT1_PIN << 2))
               ;
  //
  // Set the output drive strength to 2mA as done in the TI sample code.
  //
  PQ_GPIOPC    &= ~((0x3uL << (CLK_PIN  << 1)) |
                    (0x3uL << (DAT0_PIN << 1)));
  PF_GPIOPC    &= ~((0x3uL << (DAT1_PIN << 1)));
```



```
  PQ_GPIODR2R  |= 0
               |  (1uL << CLK_PIN)
               |  (1uL << DAT0_PIN)
               ;
  PQ_GPIODR4R  &= ~((0x1uL << CLK_PIN) |
                    (0x1uL << DAT0_PIN));
  PQ_GPIODR8R  &= ~((0x1uL << CLK_PIN) |
                    (0x1uL << DAT0_PIN));
  PF_GPIODR2R  |= 0
               |  (1uL << DAT1_PIN)
               ;
  PF_GPIODR4R  &= ~((0x1uL << DAT1_PIN));
  PF_GPIODR8R  &= ~((0x1uL << DAT1_PIN));
  //
  // Enable pull-ups on all signal lines.
  //
  PQ_GPIOPUR   |= 0
               |  (1uL << CLK_PIN)
               |  (1uL << DAT0_PIN)
               ;
  PF_GPIOPUR   |= 0
               |  (1uL << DAT1_PIN)
               ;
  PQ_GPIOODR   &= ~((0x1uL << CLK_PIN) |
                    (0x1uL << DAT0_PIN));
  PF_GPIOODR   &= ~((0x1uL << DAT1_PIN));
  PQ_GPIOPDR   &= ~((0x1uL << CLK_PIN) |
                    (0x1uL << DAT0_PIN));
  PF_GPIOPDR   &= ~((0x1uL << DAT1_PIN));
  //
  // Enable the pins.
  //
  PQ_GPIODEN   |= 0
               |  (1uL << CLK_PIN)
               |  (1uL << DAT0_PIN)
               ;
  PF_GPIODEN   |= 0
               |  (1uL << DAT1_PIN)
               ;
  //
  // The CS signal is controlled by the driver.
  //
  PH_GPIODIR   |=   1uL << CS_PIN;
  PH_GPIOAFSEL &= ~(1uL << CS_PIN);
  PH_GPIOPUR   &= ~(1uL << CS_PIN);
  PH_GPIOPDR   &= ~(1uL << CS_PIN);
  PH_GPIOODR   &= ~(1uL << CS_PIN);
  PH_GPIODEN   |=   1uL << CS_PIN;
}

#if FS_MMC_HW_SPI_USE_DMA

/*********************************************************************
*
*       _InitDMA
*/
static void _InitDMA(void) {
  DMACFG         |= 1uL << CFG_MASTEREN;        // Enable the DMA controller.
  DMACTLBASE      = (U32)_aDMADesc;             // The descriptor table must be
 1024 aligned.
  DMAPRIOCLR      = 0                           // Set the priority to default.
                  | (1uL << DMA_CHANNEL_READ)
                  | (1uL << DMA_CHANNEL_WRITE)
                  ;
  DMAALTCLR       = 0
  // Use the primary control structure in the DMA descriptor table.
                  | (1uL << DMA_CHANNEL_READ)
                  | (1uL << DMA_CHANNEL_WRITE)
                  ;
  DMAUSEBURSTCLR  = 0                           // Respond to single and burst requests.
                  | (1uL << DMA_CHANNEL_READ)
                  | (1uL << DMA_CHANNEL_WRITE)
                  ;
  DMAREQMASKCLR   = 0                           // Allow the recongition of requests.
                  | (1uL << DMA_CHANNEL_READ)
                  | (1uL << DMA_CHANNEL_WRITE)
```



```
                  ;
  //
  // Receive requests from SSI.
  //
  DMACHMAP1      &= ~((CHMAP_CHSEL_MASK << ((DMA_CHANNEL_READ - 8) << 2)) |
                      (CHMAP_CHSEL_MASK << ((DMA_CHANNEL_WRITE - 8) << 2)));
  DMACHMAP1      |= (CHMAP1_SSI3_RX << ((DMA_CHANNEL_READ  - 8) << 2))
                  |  (CHMAP1_SSI3_TX << ((DMA_CHANNEL_WRITE - 8) << 2))
                  ;
  //
  // Give DMA access to internal flash memory.
  //
  FLASHPP    |= 1uL << PP_DFA;
  FLASHDMASZ  = 0x3FFFFuL;
  FLASHDMAST  = 0;               // Internal flash memory starts at address 0.
}

/*********************************************************************
*
*       _StartDMAWrite
*/
static void _StartDMAWrite(const U8 * pData, U32 NumBytes) {
  U32        TimeOut;
  U32        CHCTLReg;
  DMA_DESC * pDMADesc;

  //
  // Stop the DMA channels if required.
  //
  if (DMAENASET & ((1uL << DMA_CHANNEL_WRITE) | (1uL << DMA_CHANNEL_READ))) {
    DMAENACLR = 0
              | (1uL << DMA_CHANNEL_WRITE)
              | (1uL << DMA_CHANNEL_READ)
              ;
    TimeOut = FS_MMC_HW_SPI_WAIT_TIMEOUT_CYCLES;
    while (1) {
      if ((DMAENASET & ((1uL << DMA_CHANNEL_WRITE) | (1uL << DMA_CHANNEL_READ))) == 0) {
        break;
      }
      if (--TimeOut == 0) {
        break;
      }
    }
  }
  //
  // Configure the write data transfer.
  //
  NumBytes = (NumBytes - 1) & CHCTL_XFERSIZE_MASK;
  CHCTLReg = 0
           | (CHCTL_DSTINC_NOINC   << CHCTL_DSTINC)
           | (CHCTL_ARBSIZE_4      << CHCTL_ARBSIZE)
           | (NumBytes             << CHCTL_XFERSIZE)
           | (CHCTL_XFERMODE_BASIC << CHCTL_XFERMODE)
           ;
  //
  // Fill in the DMA descriptor.
  //
  pDMADesc = &_aDMADesc[DMA_CHANNEL_WRITE];
  pDMADesc->SRCENDP  = (U32)(pData + NumBytes);
  pDMADesc->DESTENDP = (U32)&SSIDR;
  pDMADesc->CHCTL    = CHCTLReg;
  //
  // Configure the read data transfer.
  //
  CHCTLReg = 0
           | (CHCTL_DSTINC_NOINC   << CHCTL_DSTINC)
           | (CHCTL_SRCINC_NOINC   << CHCTL_SRCINC)
           | (CHCTL_ARBSIZE_4      << CHCTL_ARBSIZE)
           | (NumBytes             << CHCTL_XFERSIZE)
           | (CHCTL_XFERMODE_BASIC << CHCTL_XFERMODE)
           ;
  //
  // Fill in the DMA descriptor.
  //
  pDMADesc = &_aDMADesc[DMA_CHANNEL_READ];
  pDMADesc->SRCENDP  = (U32)&SSIDR;
```



```
  pDMADesc->DESTENDP = (U32)&_Dummy;
  pDMADesc->CHCTL    = CHCTLReg;
  //
  // Start the DMA transfer.
  //
  SSIDMACTL  = 0
             | (1uL << DMACTL_RXDMAE)
             | (1uL << DMACTL_TXDMAE)
             ;
  DMAENASET |= 0
            | (1uL << DMA_CHANNEL_READ)
            | (1uL << DMA_CHANNEL_WRITE)
            ;
}

/*********************************************************************
*
*       _StartDMARead
*/
static void _StartDMARead(U8 * pData, U32 NumBytes) {
  U32 TimeOut;
  U32 CHCTLReg;
  DMA_DESC * pDMADesc;

  _Dummy = 0xFF;
  FS_USE_PARA(_Dummy);
  //
  // Stop the DMA channels if required.
  //
  if (DMAENASET & ((1uL << DMA_CHANNEL_WRITE) | (1uL << DMA_CHANNEL_READ))) {
    DMAENACLR = 0
              | (1uL << DMA_CHANNEL_WRITE)
              | (1uL << DMA_CHANNEL_READ)
              ;
    TimeOut = FS_MMC_HW_SPI_WAIT_TIMEOUT_CYCLES;
    while (1) {
      if ((DMAENASET & ((1uL << DMA_CHANNEL_WRITE) | (1uL << DMA_CHANNEL_READ))) == 0) {
        break;
      }
      if (--TimeOut == 0) {
        break;
      }
    }
  }
  //
  // Configure the write data transfer.
  //
  NumBytes = (NumBytes - 1) & CHCTL_XFERSIZE_MASK;
  CHCTLReg = 0
           | (CHCTL_DSTINC_NOINC   << CHCTL_DSTINC)
           | (CHCTL_SRCINC_NOINC   << CHCTL_SRCINC)
           | (CHCTL_ARBSIZE_4      << CHCTL_ARBSIZE)
           | (NumBytes             << CHCTL_XFERSIZE)
           | (CHCTL_XFERMODE_BASIC << CHCTL_XFERMODE)
           ;
  //
  // Fill in the DMA descriptor.
  //
  pDMADesc = &_aDMADesc[DMA_CHANNEL_WRITE];
  pDMADesc->SRCENDP  = (U32)&_Dummy;
  pDMADesc->DESTENDP = (U32)&SSIDR;
  pDMADesc->CHCTL    = CHCTLReg;
  //
  // Configure the read data transfer.
  //
  CHCTLReg = 0
           | (CHCTL_SRCINC_NOINC   << CHCTL_SRCINC)
           | (CHCTL_ARBSIZE_4      << CHCTL_ARBSIZE)
           | (NumBytes             << CHCTL_XFERSIZE)
           | (CHCTL_XFERMODE_BASIC << CHCTL_XFERMODE)
           ;
  //
  // Fill in the DMA descriptor.
  //
  pDMADesc = &_aDMADesc[DMA_CHANNEL_READ];
  pDMADesc->SRCENDP  = (U32)&SSIDR;
```



```
  pDMADesc->DESTENDP = (U32)(pData + NumBytes);
  pDMADesc->CHCTL    = CHCTLReg;
  //
  // Start the DMA transfer.
  //
  SSIDMACTL  = 0
             | (1uL << DMACTL_RXDMAE)
             | (1uL << DMACTL_TXDMAE)
             ;
  DMAENASET |= 0
            | (1uL << DMA_CHANNEL_READ)
            | (1uL << DMA_CHANNEL_WRITE)
            ;
}

/*********************************************************************
*
*       _WaitForEndOfDMAOperation
*/
static void _WaitForEndOfDMAOperation(void) {
  U32 TimeOut;

  TimeOut = FS_MMC_HW_SPI_WAIT_TIMEOUT_CYCLES;
  while (1) {
    if ((DMAENASET & (1uL << DMA_CHANNEL_READ)) == 0) {
      break;
    }
    if (--TimeOut == 0) {
      break;
    }
  }
}

#endif

/*********************************************************************
*
*       _SetMaxSpeed
*/
static U16 _SetMaxSpeed(U16 MaxFreq_kHz) {
  U32 SCRValue;
  U32 CPSValue;
  U32 Factor;

  if (MaxFreq_kHz > FS_MMC_HW_SPI_MAX_SPI_CLOCK_KHZ) {
    MaxFreq_kHz = FS_MMC_HW_SPI_MAX_SPI_CLOCK_KHZ;
  }
  CPSValue = 0;
  Factor = (FS_MMC_HW_SPI_PERIPH_CLOCK_KHZ + (MaxFreq_kHz - 1)) / MaxFreq_kHz;
  do {
    CPSValue += 2;
    SCRValue = ((Factor + (CPSValue - 1)) / CPSValue) - 1;
  } while (SCRValue > 255);
  SSICPSR  = CPSValue;
  SSICR0  &= ~(CR0_SCR_MASK << CR0_SCR);
  SSICR0  |=   SCRValue     << CR0_SCR;
  MaxFreq_kHz = (U16)(FS_MMC_HW_SPI_PERIPH_CLOCK_KHZ / (CPSValue * (1 + SCRValue)));
  return MaxFreq_kHz;
}

/*********************************************************************
*
*       _Write
*/
static void _Write(const U8 * pData, int NumBytes) {
#if FS_MMC_HW_SPI_USE_DMA
  int NumBytesAtOnce;

  do {
    NumBytesAtOnce = SEGGER_MIN(NumBytes, DMA_MAX_TRANSFER_SIZE);
    _StartDMAWrite(pData, (U32)NumBytesAtOnce);
    _WaitForEndOfDMAOperation();
    NumBytes -= (int)NumBytesAtOnce;
    pData    += NumBytesAtOnce;
  } while (NumBytes);
#else
```



```
  do {
    //
    // Wait for room in FIFO.
    //
    while ((SSISR & (1uL << SR_TNF)) == 0) {
      ;
    }
    SSIDR = *pData++;
    //
    // Discard received data.
    //
    while ((SSISR & (1ul << SR_RNE)) == 0) {
      ;
    }
    (U8)SSIDR;
  } while (--NumBytes);
#endif
  //
  // Wait for the data transfer to finish.
  //
  while (SSISR & (1uL << SR_BSY)) {
    ;
  }
}

/*********************************************************************
*
*       _Read
*/
static void _Read(U8 * pData, int NumBytes) {
#if FS_MMC_HW_SPI_USE_DMA
  int NumBytesAtOnce;

  do {
    NumBytesAtOnce = SEGGER_MIN(NumBytes, DMA_MAX_TRANSFER_SIZE);
    _StartDMARead(pData, (U32)NumBytesAtOnce);
    _WaitForEndOfDMAOperation();
    NumBytes -= NumBytesAtOnce;
    pData    += NumBytesAtOnce;
  } while (NumBytes);
#else
  do {
    //
    // Wait for room in FIFO.
    //
    while ((SSISR & (1uL << SR_TNF)) == 0) {
      ;
    }
    SSIDR = 0xFF;
    //
    // Wait for some data to be received.
    //
    while ((SSISR & (1ul << SR_RNE)) == 0) {
      ;
    }
    *pData++ = (U8)SSIDR;
  } while (--NumBytes);
#endif
}

/*********************************************************************
*
*       _Init
*/
static void _Init(void) {
  _EnableClocks();
  _InitPins();
  SSICR1 = 0;
  SSICR0 = 0
         | (CR0_DSS_8BIT << CR0_DSS)
         | (1uL          << CR0_SPO)
         | (1uL          << CR0_SPH)
         ;
  SSICC  = 0;
  SSIICR = ~0uL;
  SSIIM  = 0;
```



```
  SSIDMACTL = 0;
  (void)_SetMaxSpeed(DEFAULT_SPI_CLOCK_KHZ);
  SSICR1 |= 1uL << CR1_SSE;
  //
  // Empty the receive queue.
  //
  while (1) {
    if ((SSISR & (1uL << SR_RNE)) == 0) {
      break;
    }
    (void)SSIDR;
  }
#if FS_MMC_HW_SPI_USE_DMA
  _InitDMA();
#endif
  _IsInited = 1;
}

/*********************************************************************
*
*       Public code (through callback)
*
**********************************************************************
*/

/*********************************************************************
*
*       _HW_EnableCS
*
*  Function description
*    FS low level function. Sets the card slot active using the
*    chip select (CS) line.
*
*  Parameters
*    Unit     Device index
*/
static void _HW_EnableCS(U8 Unit) {
  volatile U32 * pReg;

  FS_USE_PARA(Unit);
  pReg  = (volatile U32 *)((U8 *)&PH_GPIODATA + ((1uL << CS_PIN) << 2));
  *pReg = ~(1uL << CS_PIN);         // Active low
}

/*********************************************************************
*
*       _HW_DisableCS
*
*  Function description
*    FS low level function. Clears the card slot inactive using the
*    chip select (CS) line.
*
*  Parameters
*    Unit     Device index
*/
static void _HW_DisableCS(U8 Unit) {
  volatile U32 * pReg;

  FS_USE_PARA(Unit);
  pReg  = (volatile U32 *)((U8 *)&PH_GPIODATA + ((1uL << CS_PIN) << 2));
  *pReg = 1uL << CS_PIN;            // Active low
}

/*********************************************************************
*
*       _HW_IsWriteProtected
*
*  Function description
*    FS low level function. Returns the state of the physical write
*    protection of the SD cards.
*
*  Parameters
*    Unit     Device index
*
*  Return value
*    ==1      The card is write protected
```



```
*    ==0      The card is not write protected
*/
static int _HW_IsWriteProtected(U8 Unit) {
  FS_USE_PARA(Unit);
  return 0;
}

/*********************************************************************
*
*       _HW_SetMaxSpeed
*
*  Function description
*    FS low level function. Sets the SPI interface to a maximum frequency.
*    Make sure that you set the frequency lower or equal but never higher
*    than the given value. Recommended startup frequency is 100kHz - 400kHz.
*
*  Parameters
*    Unit       Device index
*    MaxFreq    SPI clock frequency in kHz
*
*  Return value
*    max. frequency     The maximum frequency set in kHz
*    ==0                The frequency could not be set
*/
static U16 _HW_SetMaxSpeed(U8 Unit, U16 MaxFreq) {
  MaxFreq = _SetMaxSpeed(MaxFreq);
  return MaxFreq;         // We are not faster than this.
}

/*********************************************************************
*
*       _HW_SetVoltage
*
*  Function description
*    FS low level function. Be sure that your card slot si within the given
*    voltage range. Return 1 if your slot can support the required voltage,
*    and if not, return 0;
*
*  Parameters
*    Unit       Device index
*    Vmin       Minimum supply voltage supported by host
*    Vmax       Maximum supply voltage supported by host
*
*  Return value
*    ==1        The card slot supports the voltage range
*    ==0        The card slot does not support the voltage range
*/
static int _HW_SetVoltage(U8 Unit, U16 Vmin, U16 Vmax) {
  FS_USE_PARA(Unit);
  if ((FS_MMC_HW_SPI_DEFAULT_SUPPLY_VOLTAGE >= Vmin) && (FS_MMC_HW_SPI_DEFAULT_SUPPLY_VOLTAGE <= Vmax)) {
    return 1;         // Supply voltage supported.
  }
  return 0;           // Supply voltage not supported.
}

/*********************************************************************
*
*       _HW_IsPresent
*
*  Function description
*    Returns the state of the media. If you do not know the state, return
*    FS_MEDIA_STATE_UNKNOWN and the higher layer will try to figure out if
*    a media is present.
*
*  Parameters
*    Unit     Device index
*
*  Return value
*    FS_MEDIA_STATE_UNKNOWN     Media state is unknown
*    FS_MEDIA_NOT_PRESENT       Media is not present
*    FS_MEDIA_IS_PRESENT        Media is present
*/
static int _HW_IsPresent(U8 Unit) {
  FS_USE_PARA(Unit);
  if (_IsInited == 0) {
    _Init();
```



```
  }
  return FS_MEDIA_IS_PRESENT;
}

/*********************************************************************
*
*       _HW_Read
*
*  Function description
*    FS low level function. Reads a specified number of bytes from MMC
*    card to buffer.
*
*  Parameters
*    Unit       Device index
*    pData      Pointer to a data buffer
*    NumBytes   Number of bytes
*/
static void _HW_Read(U8 Unit, U8 * pData, int NumBytes) {
  FS_USE_PARA(Unit);
  _Read(pData, NumBytes);
}

/*********************************************************************
*
*       _HW_Write
*
*  Function description
*    FS low level function. Writes a specified number of bytes from
*    data buffer to the MMC/SD card.
*
*  Parameters
*    Unit       Device index
*    pData      Pointer to a data buffer
*    NumBytes   Number of bytes
*/
static void _HW_Write(U8 Unit, const U8 * pData, int NumBytes) {
  FS_USE_PARA(Unit);
  _Write(pData, NumBytes);
}

/*********************************************************************
*
*       Public data
*
**********************************************************************
*/
const FS_MMC_HW_TYPE_SPI FS_MMC_HW_SPI_TM4C129_TI_TIVAC129 = {
  _HW_EnableCS,
  _HW_DisableCS,
  _HW_IsPresent,
  _HW_IsWriteProtected,
  _HW_SetMaxSpeed,
  _HW_SetVoltage,
  _HW_Read,
  _HW_Write,
  NULL,
  NULL,
  NULL,
  NULL
};

/************************** End of file ****************************/
```


6.5.4.3 Hardware layer API - FS_MMC_HW_TYPE_CM

This hardware layer supports MMC cards and eMMC devices that comply with the version 4.x or newer of the MMC specification and SD cards of all types that are interfaced to the target MCU via a host controller that works in card mode. The functions of this hardware layer are grouped in a structure of type `FS_MMC_HW_TYPE_CM`. The following sections describe these functions in detail.

6.5.4.3.1 FS_MMC_HW_TYPE_CM

Description

Hardware layer API for Card mode MMC/SD driver

Type definition

```
typedef struct {
  FS_MMC_HW_TYPE_CM_INIT                      * pfInitHW;
  FS_MMC_HW_TYPE_CM_DELAY                     * pfDelay;
  FS_MMC_HW_TYPE_CM_IS_PRESENT                * pfIsPresent;
  FS_MMC_HW_TYPE_CM_IS_WRITE_PROTECTED        * pfIsWriteProtected;
  FS_MMC_HW_TYPE_CM_SET_MAX_SPEED             * pfSetMaxSpeed;
  FS_MMC_HW_TYPE_CM_SET_RESPONSE_TIMEOUT      * pfSetResponseTimeOut;
  FS_MMC_HW_TYPE_CM_SET_READ_DATA_TIMEOUT     * pfSetReadDataTimeOut;
  FS_MMC_HW_TYPE_CM_SEND_CMD                  * pfSendCmd;
  FS_MMC_HW_TYPE_CM_GET_RESPONSE              * pfGetResponse;
  FS_MMC_HW_TYPE_CM_READ_DATA                 * pfReadData;
  FS_MMC_HW_TYPE_CM_WRITE_DATA                * pfWriteData;
  FS_MMC_HW_TYPE_CM_SET_DATA_POINTER          * pfSetDataPointer;
  FS_MMC_HW_TYPE_CM_SET_BLOCK_LEN             * pfSetHWBlockLen;
  FS_MMC_HW_TYPE_CM_SET_NUM_BLOCKS            * pfSetHWNumBlocks;
  FS_MMC_HW_TYPE_CM_GET_MAX_READ_BURST        * pfGetMaxReadBurst;
  FS_MMC_HW_TYPE_CM_GET_MAX_WRITE_BURST       * pfGetMaxWriteBurst;
  FS_MMC_HW_TYPE_CM_GET_MAX_REPEAT_WRITE_BURST * pfGetMaxWriteBurstRepeat;
  FS_MMC_HW_TYPE_CM_GET_MAX_FILL_WRITE_BURST  * pfGetMaxWriteBurstFill;
} FS_MMC_HW_TYPE_CM;
```

Structure members

Member	Description
pfInitHW	Initializes the hardware layer.
pfDelay	Blocks the execution for the specified time.
pfIsPresent	Checks if the card is inserted.
pfIsWriteProtected	Checks if the card is write protected.
pfSetMaxSpeed	Configures the clock frequency supplied to the card.
pfSetResponseTimeOut	Configures the maximum time to wait for a response from card.
pfSetReadDataTimeOut	Configures the maximum time to wait for data from card.
pfSendCmd	Sends a command to card.
pfGetResponse	Receives a response from card.
pfReadData	Transfers data from card to MCU.
pfWriteData	Transfers data from MCU to card.
pfSetDataPointer	Configures the pointer to the data to be exchanged.
pfSetHWBlockLen	Configures the size of the data block to be exchanged.
pfSetHWNumBlocks	Configures the number of data blocks to be exchanged.
pfGetMaxReadBurst	Returns the maximum number of data blocks that can be transfered at once from card to MCU.
pfGetMaxWriteBurst	Returns the maximum number of data blocks that can be transfered at once from MCU to card.
pfGetMaxWriteBurstRepeat	Returns the maximum number of blocks with identical data that can be transfered at once from MCU to card.
pfGetMaxWriteBurstFill	Returns the maximum number of blocks with identical data that can be transfered at once from MCU to card.

6.5.4.3.2 FS_MMC_HW_TYPE_CM_INIT

Description

Initializes the hardware layer.

Type definition

```
typedef void FS_MMC_HW_TYPE_CM_INIT(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)

Additional information

This function is a member of the Card mode MMC/SD hardware layer and it has to be implemented by any hardware layer.

The Card mode MMC/SD driver calls FS_MMC_HW_TYPE_CM_INIT before any other function of the hardware layer. FS_MMC_HW_TYPE_CM_INIT must perform any steps required to initialize the target hardware such as enabling the clocks, setting up special function registers, etc.

6.5.4.3.3 FS_MMC_HW_TYPE_CM_DELAY

Description

Blocks the execution for the specified time.

Type definition

```
typedef void FS_MMC_HW_TYPE_CM_DELAY(int ms);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
ms	Number of milliseconds to block the execution.

Additional information

This function is a member of the Card mode MMC/SD hardware layer and it has to be implemented by any hardware layer.

The specified time 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 should be added to the delay parameter before passing it to an RTOS delay function.

6.5.4.3.4 FS_MMC_HW_TYPE_CM_IS_PRESENT

Description

Checks if the card is present.

Type definition

```
typedef int FS_MMC_HW_TYPE_CM_IS_PRESENT(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)

Return value

FS_MEDIA_STATE_UNKNOWN	The presence state of the card is unknown.
FS_MEDIA_NOT_PRESENT	The card is present.
FS_MEDIA_IS_PRESENT	The card is not present.

Additional information

This function is a member of the Card mode MMC/SD hardware layer and it has to be implemented by any hardware layer.

Typically, the card presence detection is implemented via a dedicated Card Detect (CD) signal. This signal is connected to a switch inside the card slot that changes its state each time the card is removed or inserted. If the hardware does not provide such signal the function has to return FS_MEDIA_STATE_UNKNOWN.

6.5.4.3.5 FS_MMC_HW_TYPE_CM_IS_WRITE_PROTECTED

Description

Checks if the card is write protected.

Type definition

```
typedef int FS_MMC_HW_TYPE_CM_IS_WRITE_PROTECTED(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)

Return value

= 0 The card data can be modified.
≠ 0 The card data cannot be modified.

Additional information

This function is a member of the Card mode MMC/SD hardware layer and it has to be implemented by any hardware layer.

Typically, the card protection status is implemented via a dedicated Write Protected (WP) signal. This signal is connected to a switch inside the card slot that reflects the status of the write protect switch found on SD cards. If the hardware does not provide such signal the function has to return 0.

eMMC devices and micro SD cards do not have a write protection switch. Please note that the write protect switch does not really protect the data on an SD card form being modified. It is merely an indication that the data has to be protected. It is the responsibility of the host MCU to respect the status of this switch.

6.5.4.3.6 FS_MMC_HW_TYPE_CM_SET_MAX_SPEED

Description

Configures the clock frequency supplied to the card.

Type definition

```
typedef U16 FS_MMC_HW_TYPE_CM_SET_MAX_SPEED(U8  Unit,
                                            U16 Freq);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
MaxFreq	Maximum allowed frequency of the bus clock in kHz.

Return value

≠ 0 The actual clock frequency in kHz.
= 0 An error occurred.

Additional information

This function is a member of the Card mode MMC/SD hardware layer and it has to be implemented by any hardware layer.

The hardware layer is allowed to set a clock frequency smaller than the value specified via MaxFreq but never greater than that. The Card mode MMC/SD driver calls this function at least two times during the initialization of the MMC or SD card:

- Before the initialization starts to set the clock frequency to 400 kHz.
- After the card identification to set the standard clock frequency which is 25 MHz for SD cards or 26 MHz for MMC cards and eMMC devices.

FS_MMC_HW_TYPE_CM_SPI_SET_MAX_SPEED is called a third time by the Card mode MMC/SD driver during the initialization if the high speed mode is enabled in the driver via FS_MMC_CM_AllowHighSpeedMode() and the card supports this mode. In this case the clock frequency is set to 50 MHz for SD cards and to 52 MHz for MMC cards and eMMC devices.

The function has to return the actual configured clock frequency. If the precise frequency is unknown such as for implementation using I/O port “bit-banging”, the return value has to be less than the specified frequency. This may lead to longer timeout values but is in general does not cause any problems. The Card mode MMC/SD driver uses the returned value to calculate timeout values.

6.5.4.3.7 FS_MMC_HW_TYPE_CM_SET_RESPONSE_TIMEOUT

Description

Configures the maximum time to wait for a response from card.

Type definition

```
typedef void FS_MMC_HW_TYPE_CM_SET_RESPONSE_TIMEOUT(U8  Unit,
                                                    U32 Value);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
Value	Number of clock cycles to wait.

Additional information

This function is a member of the Card mode MMC/SD hardware layer and it has to be implemented by any hardware layer.

The specified value is the maximum number of clock cycles the hardware layer has to wait for a response from the card. If the card does not respond within the specified timeout then FS_MMC_HW_TYPE_CM_GET_RESPONSE has to return an error to the Card mode MMC/SD driver.

6.5.4.3.8 FS_MMC_HW_TYPE_CM_SET_READ_DATA_TIMEOUT

Description

Configures the maximum time to wait for data from card.

Type definition

```
typedef void FS_MMC_HW_TYPE_CM_SET_READ_DATA_TIMEOUT(U8  Unit,
                                                     U32 Value);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
Value	Number of clock cycles to wait.

Additional information

This function is a member of the Card mode MMC/SD hardware layer and it has to be implemented by any hardware layer.

The specified value is the maximum number of clock cycles the hardware layer has to wait for data to be received from the card. If the card does not starts sending the data within the specified timeout then FS_MMC_HW_TYPE_CM_READ_DATA has to return an error to the Card mode MMC/SD driver. The timeout value specified via FS_MMC_HW_TYPE_CM_SET_READ_DATA_TIMEOUT has to be used as timeout for a write data operation if the MMC/SD host controller supports this feature.

6.5.4.3.9 FS_MMC_HW_TYPE_CM_SEND_CMD

Description

Sends a command to card.

Type definition

```
typedef void FS_MMC_HW_TYPE_CM_SEND_CMD(U8       Unit,
                                        unsigned Cmd,
                                        unsigned CmdFlags,
                                        unsigned ResponseType,
                                        U32      Arg);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
Cmd	Command code.
CmdFlags	Command options.
ResponseType	Type of response expected.
Arg	Command arguments.

Additional information

This function is a member of the Card mode MMC/SD hardware layer and it has to be implemented by any hardware layer.

Cmd is a command code defined in the SD or MMC specification.

CmdFlags specifies additional information about the command execution such as the direction of the data transfer if any, the number of data lines to be used for the data exchange, etc. It is and bitwise OR-combination of *Card mode command flags* on page 831

ResponseType specifies the format of the response expected for the sent command. It can be one of *Card mode response formats* on page 830. If the command requests a response from the card then the Card mode MMC/SD driver calls FS_MMC_HW_TYPE_CM_GET_RESPONSE to get it.

Arg is the value that is stored at the bit positions 8:39 in a command codeword.

A command codeword is always sent via the CMD signal. The same signal is used by a card to send a response back to MCU.

6.5.4.3.10 FS_MMC_HW_TYPE_CM_GET_RESPONSE

Description

Receives a response from card.

Type definition

```
typedef int FS_MMC_HW_TYPE_CM_GET_RESPONSE(U8     Unit,
                                           void * pData,
                                           U32    NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
pData	out Received response.
NumBytes	Number of bytes to be received.

Return value

= 0 OK, response received.
≠ 0 Error code indicating the failure reason.

Additional information

This function is a member of the Card mode MMC/SD hardware layer and it has to be implemented by any hardware layer.

The response is sent by the card via the CMD signal. A response codeword can be either 48- or 136-bit large.

Refer to *Card mode error codes* on page 829 for possible return values.

The MMC and SD specifications describe the structure of a response in terms of bit units with bit 0 being the first transmitted via the CMD signal. The following table shows at which byte offsets the response has to be stored to pData.

48-bit response:

Byte offset pData	Bit range response
0	47-40
1	39-32
2	31-24
3	23-16
4	15-8
5	7-0

136-bit response:

Byte offset pData	Bit range response
0	135-128
1	127-120
2	119-112
3	111-104
4	103-96
5	95-88
6	87-80

Byte offset `pData`	Bit range response
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

The first and last byte of a response codeword store control and check information that is not used by the Card mode MMC/SD driver. If this information is not provided by the SD and MMC host controller then the hardware layer does not have to store any data to these locations of `pData`. That is the response data must be stored at byte offset 1 in `pData`.

6.5.4.3.11 FS_MMC_HW_TYPE_CM_READ_DATA

Description

Transfers data from card to MCU.

Type definition

```
typedef int FS_MMC_HW_TYPE_CM_READ_DATA(U8       Unit,
                                        void   * pData,
                                        unsigned BlockSize,
                                        unsigned NumBlocks);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
pData	out Data read from the card.
BlockSize	Number of bytes in a block to be transferred.
NumBlocks	Number of blocks be transferred.

Return value

= 0 OK, data read.
≠ 0 Error code indicating the failure reason.

Additional information

This function is a member of the Card mode MMC/SD hardware layer and it has to be implemented by any hardware layer.

The Card mode MMC/SD driver initiates a data transfer by sending a read command via FS_MMC_HW_TYPE_CM_SEND_CMD. After it checks the response via FS_MMC_HW_TYPE_CM_GET_RESPONSE the Card mode MMC/SD driver calls FS_MMC_HW_TYPE_CM_READ_DATA to transfer the data from card.

Typically, an MMC or SD host controller transfers the data one block at a time. BlockSize specifies the size of one block in bytes with the number of blocks being specified via NumBlocks. BlockSize and NumBlocks can never be 0. Typically the size of block is 512 bytes but during the initialization of the card the Card mode MMC/SD driver can request blocks of different (smaller) sizes. The total number of bytes to be transferred by FS_MMC_HW_TYPE_CM_READ_DATA is BlockSize * NumBlocks.

The number of data signals to be used for the data transfer is specified by the Card mode MMC/SD driver via the command flags when the command that initiates the read operation is sent. FS_MMC_CMD_FLAG_USE_SD4MODE is set in the command flags when the data has to be exchanged via 4 data signals. This flag can be set for SD cards as well as MMC devices. FS_MMC_CMD_FLAG_USE_MMC8MODE is set only for MMC devices to transfer the data via 8 data signals.

Refer to *Card mode error codes* on page 829 for possible return values.

6.5.4.3.12 FS_MMC_HW_TYPE_CM_WRITE_DATA

Description

Transfers data from MCU to card.

Type definition

```
typedef int FS_MMC_HW_TYPE_CM_WRITE_DATA(      U8       Unit,
                                         const void   * pData,
                                               unsigned BlockSize,
                                               unsigned NumBlocks);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
pData	in Data to be written to card.
BlockSize	Number of bytes in a block to be transferred.
NumBlocks	Number of blocks be transferred.

Return value

= 0 OK, data written.
≠ 0 Error code indicating the failure reason.

Additional information

This function is a member of the Card mode MMC/SD hardware layer and it has to be implemented by any hardware layer.

The Card mode MMC/SD driver initiates a data transfer by sending a write command via FS_MMC_HW_TYPE_CM_SEND_CMD. After it checks the response via FS_MMC_HW_TYPE_CM_GET_RESPONSE the Card mode MMC/SD driver calls FS_MMC_HW_TYPE_CM_WRITE_DATA to transfer the data to card.

Typically, an MMC or SD host controller transfers the data one block at a time. BlockSize specifies the size of one block in bytes with the number of blocks being specified via NumBlocks. BlockSize and NumBlocks can never be 0. Typically the size of block is 512 bytes but during the initialization of the card the Card mode MMC/SD driver can request blocks of different (smaller) sizes. The total number of bytes to be transferred by FS_MMC_HW_TYPE_CM_WRITE_DATA is BlockSize * NumBlocks.

The number of data signals to be used for the data transfer is specified by the Card mode MMC/SD driver via the command flags when the command that initiates the write operation is sent. FS_MMC_CMD_FLAG_USE_SD4MODE is set in the command flags when the data has to be exchanged via 4 data signals. This flag can be set for SD cards as well as MMC devices. FS_MMC_CMD_FLAG_USE_MMC8MODE is set only for MMC devices to transfer the data via 8 data signals.

Refer to *Card mode error codes* on page 829 for possible return values.

6.5.4.3.13 FS_MMC_HW_TYPE_CM_SET_DATA_POINTER

Description

Configures the pointer to the data to be exchanged.

Type definition

```
typedef void FS_MMC_HW_TYPE_CM_SET_DATA_POINTER(      U8     Unit,
                                                const void * pData);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
pData	Data exchanged with the card.

Additional information

This function is a member of the Card mode MMC/SD hardware layer It has to be implemented only by a hardware layer that has to prepare the data transfer such as via DMA before the read or write command that initiates that data transfer is sent to card.

FS_MMC_HW_TYPE_CM_SET_DATA_POINTER is called by the Card mode MMC/SD driver before each command that exchanges the data with the card. pData points to the same memory region as pData passed to FS_MMC_HW_TYPE_CM_READ_DATA and FS_MMC_HW_TYPE_CM_WRITE_DATA.

6.5.4.3.14 FS_MMC_HW_TYPE_CM_SET_BLOCK_LEN

Description

Configures the size of the data block to be exchanged.

Type definition

```
typedef void FS_MMC_HW_TYPE_CM_SET_BLOCK_LEN(U8  Unit,
                                             U16 BlockSize);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
BlockSize	Number of bytes in a block.

Additional information

This function is a member of the Card mode MMC/SD hardware layer It has to be implemented only by a hardware layer that has to prepare the data transfer such as via DMA before the read or write command that initiates that data transfer is sent to card.

FS_MMC_HW_TYPE_CM_SET_BLOCK_LEN is called by the Card mode MMC/SD driver before each command that exchanges the data with the card. The same value is passed via BlockSize as the value passed via BlockSize to FS_MMC_HW_TYPE_CM_READ_DATA and FS_MMC_HW_TYPE_CM_WRITE_DATA.

6.5.4.3.15 FS_MMC_HW_TYPE_CM_SET_NUM_BLOCKS

Description

Configures the number of data blocks to be exchanged.

Type definition

```
typedef void FS_MMC_HW_TYPE_CM_SET_NUM_BLOCKS(U8  Unit,
                                               U16 NumBlocks);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
NumBlocks	Number of bytes to be exchanged.

Additional information

This function is a member of the Card mode MMC/SD hardware layer It has to be implemented only by a hardware layer that has to prepare the data transfer such as via DMA before the read or write command that initiates that data transfer is sent to card.

FS_MMC_HW_TYPE_CM_SET_NUM_BLOCKS is called by the Card mode MMC/SD driver before each command that exchanges the data with the card. The same value is passed via NumBlocks as the value passed via NumBlocks to FS_MMC_HW_TYPE_CM_READ_DATA and FS_MMC_HW_TYPE_CM_WRITE_DATA.

6.5.4.3.16 FS_MMC_HW_TYPE_CM_GET_MAX_READ_BURST

Description

Returns the maximum number of data blocks that can be transfered at once from card to MCU.

Type definition

```
typedef U16 FS_MMC_HW_TYPE_CM_GET_MAX_READ_BURST(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)

Return value

Number of blocks that can be transfered at once. It cannot be 0.

Additional information

This function is a member of the Card mode MMC/SD hardware layer and it has to be implemented by any hardware layer.

FS_MMC_HW_TYPE_CM_GET_MAX_READ_BURST is called by the Card mode MMC/SD driver during initialization. It has to return the maximum number of 512 byte blocks the hardware layer is able to transfer from the card via a single read command. The larger the number of blocks the hardware layer can exchange at once the better is the read performance.

6.5.4.3.17 FS_MMC_HW_TYPE_CM_GET_MAX_WRITE_BURST

Description

Returns the maximum number of data blocks that can be transfered at once from MCU to card.

Type definition

```
typedef U16 FS_MMC_HW_TYPE_CM_GET_MAX_WRITE_BURST(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)

Return value

Number of blocks that can be transfered at once. It cannot be 0.

Additional information

This function is a member of the Card mode MMC/SD hardware layer and it has to be implemented by any hardware layer.

FS_MMC_HW_TYPE_CM_GET_MAX_READ_BURST is called by the Card mode MMC/SD driver during initialization. It has to return the maximum number of 512 byte blocks the hardware layer is able to transfer to the card via a single write command. The larger the number of blocks the hardware layer can exchange at once the better is the write performance.

6.5.4.3.18 FS_MMC_HW_TYPE_CM_GET_MAX_REPEAT_WRITE_BURST

Description

Returns the maximum number of blocks of identical data that can be transfered from MCU to card.

Type definition

```
typedef U16 FS_MMC_HW_TYPE_CM_GET_MAX_REPEAT_WRITE_BURST(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)

Return value

Number of blocks that can be transfered at once.

Additional information

This function is a member of the Card mode MMC/SD hardware layer. A hardware layer is not required to implement this function.

FS_MMC_HW_TYPE_CM_GET_MAX_REPEAT_WRITE_BURST is called by the Card mode MMC/SD driver during initialization. It has to return the maximum number of 512 byte blocks the hardware layer is able to repeatedly transfer to the card via a single write command. The command that starts a repeated write operation has FS_MMC_CMD_FLAG_WRITE_BURST_REPEAT set in the command flags passed to FS_MMC_HW_TYPE_CM_SEND_CMD. The Card mode MMC/SD driver transfers the data by calling FS_MMC_HW_TYPE_CM_WRITE_DATA with pData always pointing to a single block of BlockSize regardless of the NumBlock value. That FS_MMC_HW_TYPE_CM_WRITE_DATA does not have to increment pData by BlockSize after each block transfer. The larger the number of blocks the hardware layer can exchange at once the better is the write performance.

A return value of 0 indicates that the feature is not supported.

6.5.4.3.19 FS_MMC_HW_TYPE_CM_GET_MAX_FILL_WRITE_BURST

Description

Returns the maximum number of blocks of identical data that can be transfered from MCU to card.

Type definition

```
typedef U16 FS_MMC_HW_TYPE_CM_GET_MAX_FILL_WRITE_BURST(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)

Return value

Number of blocks that can be transfered at once.

Additional information

This function is a member of the Card mode MMC/SD hardware layer. A hardware layer is not required to implement this function.

FS_MMC_HW_TYPE_CM_GET_MAX_FILL_WRITE_BURST is called by the Card mode MMC/SD driver during initialization. It has to return the maximum number of 512 byte blocks the hardware layer is able to repeatedly transfer to the card via a single write command. The command that starts a repeated write operation has FS_MMC_CMD_FLAG_WRITE_BURST_FILL set in the command flags passed to FS_MMC_HW_TYPE_CM_SEND_CMD. The Card mode MMC/SD driver transfers the data by calling FS_MMC_HW_TYPE_CM_WRITE_DATA with pData always pointing to a single block of BlockSize regardless of the NumBlocks value. FS_MMC_HW_TYPE_CM_GET_MAX_FILL_WRITE_BURST has to take the first four bytes from pData and send them repeatedly to card until BlockSize * NumBlocks are sent. Typically this operation is realized by using DMA configured to transfer four bytes at once and without incrementing the destination address. The larger the number of blocks the hardware layer can exchange at once the better is the write performance.

A return value of 0 indicates that the feature is not supported.

6.5.4.3.20 Card mode error codes

Description

Values that indicate the result of an operation.

Definition

```
#define FS_MMC_CARD_NO_ERROR                  0
#define FS_MMC_CARD_RESPONSE_TIMEOUT          1
#define FS_MMC_CARD_RESPONSE_CRC_ERROR        2
#define FS_MMC_CARD_READ_TIMEOUT              3
#define FS_MMC_CARD_READ_CRC_ERROR            4
#define FS_MMC_CARD_WRITE_CRC_ERROR           5
#define FS_MMC_CARD_RESPONSE_GENERIC_ERROR    6
#define FS_MMC_CARD_READ_GENERIC_ERROR        7
#define FS_MMC_CARD_WRITE_GENERIC_ERROR       8
```

Symbols

Definition	Description
FS_MMC_CARD_NO_ERROR	Success.
FS_MMC_CARD_RESPONSE_TIMEOUT	No response received.
FS_MMC_CARD_RESPONSE_CRC_ERROR	CRC error in response detected.
FS_MMC_CARD_READ_TIMEOUT	No data received.
FS_MMC_CARD_READ_CRC_ERROR	CRC error in received data detected.
FS_MMC_CARD_WRITE_CRC_ERROR	Card detected an CRC error in the received data.
FS_MMC_CARD_RESPONSE_GENERIC_ERROR	Start bit, end bit or command index error.
FS_MMC_CARD_READ_GENERIC_ERROR	An error occurred during while receiving data from card.
FS_MMC_CARD_WRITE_GENERIC_ERROR	An error occurred during while sending data to card.

6.5.4.3.21 Card mode response formats

Description

Response types returned by different commands.

Definition

```
#define FS_MMC_RESPONSE_FORMAT_NONE    0
#define FS_MMC_RESPONSE_FORMAT_R1      1
#define FS_MMC_RESPONSE_FORMAT_R2      2
#define FS_MMC_RESPONSE_FORMAT_R3      3
#define FS_MMC_RESPONSE_FORMAT_R6      FS_MMC_RESPONSE_FORMAT_R1
#define FS_MMC_RESPONSE_FORMAT_R7      FS_MMC_RESPONSE_FORMAT_R1
```

Symbols

Definition	Description
FS_MMC_RESPONSE_FORMAT_NONE	No response expected.
FS_MMC_RESPONSE_FORMAT_R1	Card status (48-bit large)
FS_MMC_RESPONSE_FORMAT_R2	CID or CSD register (128-bit large)
FS_MMC_RESPONSE_FORMAT_R3	OCR register (48-bit large)
FS_MMC_RESPONSE_FORMAT_R6	Published RCA response (48-bit large)
FS_MMC_RESPONSE_FORMAT_R7	Card interface condition (48-bit large)

6.5.4.3.22 Card mode command flags

Description

Additional options for the executed command.

Definition

```
#define FS_MMC_CMD_FLAG_DATATRANSFER          (1uL << 0)
#define FS_MMC_CMD_FLAG_WRITETRANSFER         (1uL << 1)
#define FS_MMC_CMD_FLAG_SETBUSY               (1uL << 2)
#define FS_MMC_CMD_FLAG_INITIALIZE            (1uL << 3)
#define FS_MMC_CMD_FLAG_USE_SD4MODE           (1uL << 4)
#define FS_MMC_CMD_FLAG_STOP_TRANS            (1uL << 5)
#define FS_MMC_CMD_FLAG_WRITE_BURST_REPEAT    (1uL << 6)
#define FS_MMC_CMD_FLAG_USE_MMC8MODE          (1uL << 7)
#define FS_MMC_CMD_FLAG_NO_CRC_CHECK          (1uL << 8)
#define FS_MMC_CMD_FLAG_WRITE_BURST_FILL      (1uL << 9)
```

Symbols

Definition	Description
FS_MMC_CMD_FLAG_DATATRANSFER	Command that exchanges data with the card.
FS_MMC_CMD_FLAG_WRITETRANSFER	Command that sends data to card. Implies FS_MMC_CMD_FLAG_DATATRANSFER.
FS_MMC_CMD_FLAG_SETBUSY	Command that expects an R1b command.
FS_MMC_CMD_FLAG_INITIALIZE	Indicates that the initialization delay has to be performed. According to SD specification this is the maximum of 1 millisecond, 74 clock cycles and supply ramp up time.
FS_MMC_CMD_FLAG_USE_SD4MODE	Command that transfers the data via 4 data lines.
FS_MMC_CMD_FLAG_STOP_TRANS	Command that stops a data transfer (typically CMD12)
FS_MMC_CMD_FLAG_WRITE_BURST_REPEAT	Indicates that the same sector data is written to consecutive sector indexes.
FS_MMC_CMD_FLAG_USE_MMC8MODE	Command that transfers the data via 8 data lines (MMC only).
FS_MMC_CMD_FLAG_NO_CRC_CHECK	CRC verification has to be disabled for the command. Typically used with the MMC bus test commands.
FS_MMC_CMD_FLAG_WRITE_BURST_FILL	Indicates that the 32-bit value is used to fill the contents of consecutive sector indexes.

6.5.4.3.23 Sample implementation

The following sample implementation uses the SD host controller of an NXP K66 MCU to interface with an SD card. This NOR hardware layer was tested on the SGGER emPower board (https://www.segger.com/evaluate-our-software/segger/empower/)

```
/*********************************************************************
*                  (c) SEGGER Microcontroller GmbH                   *
*                       The Embedded Experts                         *
*                          www.segger.com                            *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File        : FS_MMC_CM_HW_K66_SEGGER_emPower.c
Purpose     : SD/MMC (card mode) hardware layer for Freescale Kinetis K66
Literature  :
  [1] UM06001 emPower Evaluation and prototyping platform for SEGGER software User Guide &
 Reference Manual
    (\\FILESERVER\Product\Doc4Review\UM06001_emPower.pdf)
  [2] K66P144M180SF5RMV2 K66 Sub-Family Reference Manual
    (\\fileserver\Techinfo\Company\NXP\MCU\Kinetis_K-series
\K66P144M180SF5RMV2_Rev2_1505.pdf)
  [3] SD Specifications Part 1 Physical Layer Simplified Specification Version 2.00
    (\\fileserver\Techinfo\Company\SDCard_org\SDCardPhysicalLayerSpec_V200.pdf)
*/

/*********************************************************************
*
*       #include section
*
**********************************************************************
*/
#include "FS.h"

/*********************************************************************
*
*       Defines, configurable
*
**********************************************************************
*/
#ifndef   FS_MMC_CM_HW_USE_DMA
  #define FS_MMC_CM_HW_USE_DMA              1           // Turns the DMA support on and off
#endif

#ifndef   FS_MMC_CM_HW_PERIPH_CLOCK
  #define FS_MMC_CM_HW_PERIPH_CLOCK         168000000uL // Peripheral clock speed in Hz
#endif

#ifndef   FS_MMC_CM_HW_MAX_SPEED
  #define FS_MMC_CM_HW_MAX_SPEED            50000
  // Limits the maximum communication speed to this value (kHz)
#endif

#ifndef   FS_MMC_CM_HW_NUM_DMA_DESC
  #define FS_MMC_CM_HW_NUM_DMA_DESC         32
  // Number of DMA descriptors to allocate
#endif

#ifndef   FS_MMC_CM_HW_POWER_GOOD_DELAY
  #define FS_MMC_CM_HW_POWER_GOOD_DELAY     1000
  // Number of milliseconds to wait for the power to stabilize.
#endif

/*********************************************************************
*
*       Defines, fixed
*
**********************************************************************
*/

/*********************************************************************
*
*       System integration module
*/
```



```
#define SIM_BASE                  0x40047000
#define SIM_SCGC3                 (*(volatile U32*)(SIM_BASE + 0x1030))
  // System Clock Gating Control Register 3
#define SIM_SCGC5                 (*(volatile U32*)(SIM_BASE + 0x1038))
  // System Clock Gating Control Register 5

/*********************************************************************
*
*       PORT E
*/
#define PORTE_BASE                0x4004D000
#define PORTE_PCR0                (*(volatile U32*)(PORTE_BASE + 0x00))
  // Pin control register 0
#define PORTE_PCR1                (*(volatile U32*)(PORTE_BASE + 0x04))
  // Pin control register 1
#define PORTE_PCR2                (*(volatile U32*)(PORTE_BASE + 0x08))
  // Pin control register 2
#define PORTE_PCR3                (*(volatile U32*)(PORTE_BASE + 0x0C))
  // Pin control register 3
#define PORTE_PCR4                (*(volatile U32*)(PORTE_BASE + 0x10))
  // Pin control register 4
#define PORTE_PCR5                (*(volatile U32*)(PORTE_BASE + 0x14))
  // Pin control register 5
#define PORTE_PCR6                (*(volatile U32*)(PORTE_BASE + 0x18))
  // Pin control register 5
#define PORTE_PCR27               (*(volatile U32*)(PORTE_BASE + 0x6C))
  // Pin control register 27
#define PORTE_DFER                (*(volatile U32*)(PORTE_BASE + 0xC0))
  // Digital filter enable register

/*********************************************************************
*
*       GPIO E
*/
#define GPIOE_BASE                0x400FF000
#define GPIOE_PDOR                (*(volatile U32*)(GPIOE_BASE + 0x100))
  // Port Data Output Register
#define GPIOE_PDIR                (*(volatile U32*)(GPIOE_BASE + 0x110))
  // Port Data Input Register
#define GPIOE_PDDR                (*(volatile U32*)(GPIOE_BASE + 0x114))
  // Port Data Direction Register

/*********************************************************************
*
*       SDHC
*/
#define SDHC_BASE                 0x400B1000
#define SDHC_BLKATTR              (*(volatile U32*)(SDHC_BASE + 0x04))
  // Block Attributes Register
#define SDHC_CMDARG               (*(volatile U32*)(SDHC_BASE + 0x08))
  // Command Argument Register
#define SDHC_XFERTYP              (*(volatile U32*)(SDHC_BASE + 0x0C))
  // Transfer Type Register
#define SDHC_CMDRSP0              (*(volatile U32*)(SDHC_BASE + 0x10))   // Command Response
 0
#define SDHC_DATPORT              (*(volatile U32*)(SDHC_BASE + 0x20))
  // Buffer Data Port Register
#define SDHC_PRSSTAT              (*(volatile U32*)(SDHC_BASE + 0x24))
  // Present State Register
#define SDHC_PROCTL               (*(volatile U32*)(SDHC_BASE + 0x28))
  // Protocol Control Register
#define SDHC_SYSCTL               (*(volatile U32*)(SDHC_BASE + 0x2C))
  // System Control Register
#define SDHC_IRQSTAT              (*(volatile U32*)(SDHC_BASE + 0x30))
  // Interrupt Status Register
#define SDHC_IRQSTATEN            (*(volatile U32*)(SDHC_BASE + 0x34))
  // Interrupt Status Enable Register
#define SDHC_WML                  (*(volatile U32*)(SDHC_BASE + 0x44))
  // Watermark Level Register
#define SDHC_ADSADDR              (*(volatile U32*)(SDHC_BASE + 0x58))
  // ADMA System Address Register
#define SDHC_VENDOR               (*(volatile U32*)(SDHC_BASE + 0xC0))
  // Vendor Specific Register

/*********************************************************************
*
```



```
*       MPU
*/
#define MPU_BASE                  0x4000D000
#define MPU_RGDAAC0               (*(volatile U32*)(MPU_BASE + 0x800))
  // Region Descriptor Alternate Access Control 0

/*********************************************************************
*
*       Pin assignments of signals
*/
#define SD_CMD_PIN                3   // Command         (GPIO E)
#define SD_CLK_PIN                2   // Clock           (GPIO E)
#define SD_D0_PIN                 1   // Data 0          (GPIO E)
#define SD_D1_PIN                 0   // Data 1          (GPIO E)
#define SD_D2_PIN                 5   // Data 2          (GPIO E)
#define SD_D3_PIN                 4   // Data 3          (GPIO E)
#define SD_CD_PIN                 6   // Card detect     (GPIO E)

/*********************************************************************
*
*       Port control register bits
*/
#define PCR_MUX                   8   // Alternate port setting
#define PCR_MUX_GPIO              1   // Controlled directly by the HW layer
#define PCR_MUX_SDHC              4   // Controlled by SDHC
#define PCR_DSE                   6   // Driver strength enable
#define PCR_PE                    1   // Pull enable
#define PCR_PS                    0   // Pull-up enable

/*********************************************************************
*
*       Clock enable bits
*/
#define SCGC3_SDHC                17    // SDHC
#define SCGC5_PORTE               13    // GPIO E

/*********************************************************************
*
*       SDHC interrupt request bits
*/
#define IRQSTAT_CC                0   // Command complete
#define IRQSTAT_TC                1   // Transfer complete
#define IRQSTAT_DINT              3   // DMA interrupt
#define IRQSTAT_BWR               4   // Buffer write ready
#define IRQSTAT_BRR               5   // Buffer read ready
#define IRQSTAT_CTOE              16  // Command timeout error
#define IRQSTAT_CCE               17  // Command CRC error
#define IRQSTAT_CEBE              18  // Command end bit error
#define IRQSTAT_CIE               19  // Command index error
#define IRQSTAT_DTOE              20  // Data timeout error
#define IRQSTAT_DCE               21  // Data CRC error
#define IRQSTAT_DEBE              22  // Data end bit error
#define IRQSTAT_DMAE              28  // DMA error

/*********************************************************************
*
*       SDHC present status bits
*/
#define PRSSTAT_CIHB              0   // Command inhibit (CMD)
#define PRSSTAT_CDIHB             1   // Command inhibit (DAT)
#define PRSSTAT_SDSTB             3   // Set to 1 if the clock is stable
#define PRSSTAT_SDOFF             7
#define PRSSTAT_BWEN              10
#define PRSSTAT_DLSL_DAT0         24

/*********************************************************************
*
*       SDHC system control bits
*/
#define SYSCTL_IPGEN              0
#define SYSCTL_HCKEN              1
#define SYSCTL_PEREN              2
#define SYSCTL_SDCLKEN            3
#define SYSCTL_DVS                4
#define SYSCTL_DVS_MASK           0xFuL
#define SYSCTL_DVS_MAX            (SYSCTL_DVS_MASK + 1)
```



```
#define SYSCTL_SDCLKFS            8
#define SYSCTL_SDCLKFS_MASK       0xFFuL
#define SYSCTL_DTOCV              16
#define SYSCTL_DTOCV_MASK         0xFuL
#define SYSCTL_DTOCV_MAX          (SYSCTL_DTOCV_MASK - 1)
#define SYSCTL_RSTA               24
#define SYSCTL_RSTC               25
#define SYSCTL_RSTD               26
#define SYSCTL_INITA              27

/*****************************************************************
*
*       SDHC transfer type bits
*/
#define XFERTYP_DMAEN             0     // DMA enable
#define XFERTYP_BCEN              1
#define XFERTYP_DTDSEL            4
#define XFERTYP_MSBSEL            5
#define XFERTYP_RSPTYP            16
#define XFERTYP_RSPTYP_NONE       0uL
#define XFERTYP_RSPTYP_136BIT     1uL
#define XFERTYP_RSPTYP_48BIT      2uL
#define XFERTYP_RSPTYP_48BIT_BUSY 3uL
#define XFERTYP_CCCEN             19
#define XFERTYP_CICEN             20
#define XFERTYP_DPSEL             21
#define XFERTYP_CMDIDX            24
#define XFERTYP_CMDIDX_MASK       0x3FuL

/*****************************************************************
*
*       SDHC protocol control bits
*/
#define PROCTL_DTW                1
#define PROCTL_DTW_MASK           0x3uL
#define PROCTL_DTW_4BIT           0x1uL
#define PROCTL_EMODE              4
#define PROCTL_EMODE_LE           2uL
#define PROCTL_DMAS               8
#define PROCTL_DMAS_MASK          0x3uL
#define PROCTL_DMAS_DMA2          2uL

/*****************************************************************
*
*       SDHC block attributes bits
*/
#define BLKATTR_BLKCNT            16

/*****************************************************************
*
*       SDHC Watermark level
*/
#define WML_RDWML                 0
#define WML_RDWML_MASK            0xFFuL
#define WML_WRWML                 16
#define WML_WRWML_MASK            0xFFuL

/*****************************************************************
*
*       DMA attributes
*/
#define DMA_ATTR_VALID            0
#define DMA_ATTR_END              1
#define DMA_ATTR_INT              2
#define DMA_ATTR_ACT              4
#define DMA_ATTR_ACT_TRAN         2uL

/*****************************************************************
*
*       MPU region descriptor bits
*/
#define RGD_M5RE                  27
#define RGD_M5WE                  26

/*****************************************************************
*
```



```
*       Number of sectors to transfer at once
*/
#define MAX_BLOCK_SIZE            512
#define MAX_DMA_LEN               65532

#if FS_MMC_CM_HW_USE_DMA
  #define NUM_BLOCKS_AT_ONCE
  ((FS_MMC_CM_HW_NUM_DMA_DESC * MAX_DMA_LEN) / MAX_BLOCK_SIZE)
#else
  #define NUM_BLOCKS_AT_ONCE      65535
#endif // FS_MMC_CM_HW_USE_DMA

#if FS_MMC_CM_HW_USE_DMA
  #define NUM_BLOCKS_AT_ONCE_REPEAT_SAME      FS_MMC_CM_HW_NUM_DMA_DESC
#else
  #define NUM_BLOCKS_AT_ONCE_REPEAT_SAME      65535
#endif

/*********************************************************************
*
*       Timeouts and delays
*/
#define CYCLES_PER_MS             2100
#define WAIT_TIMEOUT_MS           1000
#define WAIT_TIMEOUT_CYCLES       (WAIT_TIMEOUT_MS * CYCLES_PER_MS)

/*********************************************************************
*
*       Local data types
*
**********************************************************************
*/

#if FS_MMC_CM_HW_USE_DMA

/*********************************************************************
*
*       DMA_DESC
*
*  Description
*    Memory layout of a ADMA2 descriptor
*/
typedef struct {
  U16 Attr;       // Descriptor attributes
  U16 NumBytes;   // Number of bytes to transfer
  U32 Addr;       // Destination/source memory address (32-bit aligned)
} DMA_DESC;

#endif

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/
static U16          _BlockSize;   // Size of the block to transfer as set by the upper layer
static U16          _NumBlocks;   // Number of blocks to transfer as set by the upper layer
static U8           _RepeatSame;  // Set to
 1 if the same data has to be written to consecutive block indexes.
#if FS_MMC_CM_HW_USE_DMA
  static DMA_DESC   _aDMADesc[FS_MMC_CM_HW_NUM_DMA_DESC];    // List of DMA descriptors
  static void     * _pData;
  // Destination/source buffer for the DMA transfers
#endif

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       _WaitForResponse
```



```
*
*  Function description
*    Waits for a command response to arrive.
*
*  Return values
*    FS_MMC_CARD_NO_ERROR                Success
*    FS_MMC_CARD_RESPONSE_CRC_ERROR      CRC error in response
*    FS_MMC_CARD_RESPONSE_TIMEOUT        No response received
*    FS_MMC_CARD_RESPONSE_GENERIC_ERROR  Any other error
*/
static int _WaitForResponse(void) {
  U32 Status;
  U32 TimeOut;

  //
  // Wait for command to finish
  //
  TimeOut = WAIT_TIMEOUT_CYCLES;
  for (;;) {
    Status = SDHC_IRQSTAT;
    if ((Status & (1uL << IRQSTAT_CTOE)) != 0u) {
      return FS_MMC_CARD_RESPONSE_TIMEOUT;        // Error, no response received.
    }
    if ((Status & (1uL << IRQSTAT_CCE)) != 0u) {
      return FS_MMC_CARD_RESPONSE_CRC_ERROR;      // Error, CRC error detected in response.
    }
    if ((Status & (1uL << IRQSTAT_CEBE)) != 0u) {
      return FS_MMC_CARD_RESPONSE_GENERIC_ERROR;  // Error, end bit of response is 0.
    }
    if ((Status & (1uL << IRQSTAT_CIE)) != 0u) {
      return FS_MMC_CARD_RESPONSE_GENERIC_ERROR;  // Error, command index do not match.
    }
#if FS_MMC_CM_HW_USE_DMA
    if ((Status & (1uL << IRQSTAT_DMAE)) != 0u) {
      return FS_MMC_CARD_RESPONSE_GENERIC_ERROR;  // Error, DMA failure.
    }
#endif
    if ((Status & (1uL << IRQSTAT_CC)) != 0u) {
      return FS_MMC_CARD_NO_ERROR;                // OK, valid response received.
    }
    if (TimeOut-- == 0u) {
      return FS_MMC_CARD_RESPONSE_TIMEOUT;        // Error, no response received.
    }
  }
}

#if (FS_MMC_CM_HW_USE_DMA == 0)

/*********************************************************************
*
*       _WaitForRxReady
*
*   Function description
*     Waits until new data is received. The function returns in case of a receiving error.
*
*   Return values
*     FS_MMC_CARD_NO_ERROR              Success
*     FS_MMC_CARD_READ_CRC_ERROR        CRC error in received data
*     FS_MMC_CARD_READ_TIMEOUT          No data received
*     FS_MMC_CARD_READ_GENERIC_ERROR    Any other error
*/
static int _WaitForRxReady(void) {
  U32 Status;
  U32 TimeOut;

  TimeOut = WAIT_TIMEOUT_CYCLES;
  for (;;) {
    Status = SDHC_IRQSTAT;
    if ((Status & (1uL << IRQSTAT_DEBE)) != 0u) {
      return FS_MMC_CARD_READ_GENERIC_ERROR;      // Error, 0 detected on the end bit.
    }
    if ((Status & (1uL << IRQSTAT_DCE)) != 0u) {
      return FS_MMC_CARD_READ_CRC_ERROR;          // Error, CRC check failed.
    }
    if ((Status & (1uL << IRQSTAT_DTOE)) != 0u) {
      return FS_MMC_CARD_READ_TIMEOUT;            // Error, data timeout.
```



```
    }
    if ((Status & (1uL << IRQSTAT_BRR)) != 0u) {
      return FS_MMC_CARD_NO_ERROR;               // OK, data can be read from queue.
    }
    if (TimeOut-- == 0u) {
      return FS_MMC_CARD_READ_TIMEOUT;           // Error, data timeout.
    }
  }
}

/*********************************************************************
*
*       _WaitForTxReady
*
*   Function description
*     Waits until the transmitter is ready to send new data.
*     The function returns in case of an underrun condition.
*     This can happen when we are not able to deliver the data fast enough.
*
*   Return values
*     FS_MMC_CARD_NO_ERROR              Success
*     FS_MMC_CARD_WRITE_GENERIC_ERROR   Any other error
*/
static int _WaitForTxReady(void) {
  U32 Status;
  U32 TimeOut;

  TimeOut = WAIT_TIMEOUT_CYCLES;
  for (;;) {
    Status = SDHC_IRQSTAT;
    if ((Status & (1uL << IRQSTAT_DTOE)) != 0u) {
      return FS_MMC_CARD_WRITE_GENERIC_ERROR;     // Error, data timeout.
    }
    if ((Status & (1uL << IRQSTAT_DCE)) != 0u) {
      return FS_MMC_CARD_WRITE_CRC_ERROR;         // Error, CRC check failed.
    }
    if ((Status & (1uL << IRQSTAT_BWR)) != 0u) {
      return FS_MMC_CARD_NO_ERROR;                // OK, data can be written to queue.
    }
    if (TimeOut-- == 0u) {
      return FS_MMC_CARD_WRITE_GENERIC_ERROR;     // Error, data timeout.
    }
  }
}
#endif // FS_MMC_CM_HW_USE_DMA == 0

/*********************************************************************
*
*       _WaitForCmdReady
*
*   Function description
*     Waits until for the last command to finish.
*/
static void _WaitForCmdReady(void) {
  U32 Status;
  U32 TimeOut;

  TimeOut = WAIT_TIMEOUT_CYCLES;
  for (;;) {
    Status = SDHC_PRSSTAT;
    if ((Status & (1uL << PRSSTAT_CIHB)) == 0u) {
      if ((Status & (1uL << PRSSTAT_CDIHB)) == 0u) {
        if ((Status & (1uL << PRSSTAT_DLSL_DAT0)) != 0u) {
          break;
        }
      }
    }
    if (TimeOut-- == 0u) {
      break;
    }
  }
}

/*********************************************************************
*
*       _Delay1ms
```



```
*
*   Function description
*     Busy loops for about 1 millisecond.
*/
static void _Delay1ms(void) {
  volatile int i;

  //
  // This loop takes about 1ms at 25MHz.
  //
  for (i = 0; i < CYCLES_PER_MS; ++i) {
    ;
  }
}

/*********************************************************************
*
*       _Reset
*
*   Function description
*     Resets the command and data state machines. Typ. called in case of an error.
*/
static void _Reset(void) {
  //
  // Reset command path.
  //
  SDHC_SYSCTL |=   1uL << SYSCTL_RSTC;
  SDHC_SYSCTL &= ~(1uL << SYSCTL_RSTC);
  //
  // Reset data path.
  //
  SDHC_SYSCTL |=   1uL << SYSCTL_RSTD;
  SDHC_SYSCTL &= ~(1uL << SYSCTL_RSTD);
}

/*********************************************************************
*
*       _SetWMLRead
*
*   Function description
*     Sets the watermark level for read operations.
*/
static void _SetWMLRead(U32 NumBytes) {
  U32 NumWords;

  NumWords  = NumBytes >> 2;
  NumWords /= 3;
  if (NumWords == 0) {
    NumWords = 1;
  }
  if (NumWords > WML_RDWML_MASK) {
    NumWords = WML_RDWML_MASK;
  }
  SDHC_WML &= ~(WML_RDWML_MASK << WML_RDWML);
  SDHC_WML |= NumWords << WML_RDWML;
}

/*********************************************************************
*
*       _SetWMLlWrite
*
*   Function description
*     Sets the watermark level for write operations.
*/
static void _SetWMLWrite(U32 NumBytes) {
  U32 NumWords;

  NumWords  = NumBytes >> 2;
  NumWords  = (NumWords << 1) / 3;
  if (NumWords == 0) {
    NumWords = 1;
  }
  if (NumWords > WML_WRWML_MASK) {
    NumWords = WML_WRWML_MASK;
  }
  SDHC_WML &= ~(WML_WRWML_MASK << WML_WRWML);
```



```
  SDHC_WML |= NumWords << WML_WRWML;
}

#if (FS_MMC_CM_HW_USE_DMA == 0)

/*********************************************************************
*
*       _GetWMLRead
*/
static U32 _GetWMLRead(void) {
  U32 NumWords;

  NumWords = (SDHC_WML >> WML_RDWML) & WML_RDWML_MASK;
  return NumWords;
}

/*********************************************************************
*
*       _GetWMLWrite
*/
static U32 _GetWMLWrite(void) {
  U32 NumWords;

  NumWords = (SDHC_WML >> WML_WRWML) & WML_WRWML_MASK;
  return NumWords;
}

#endif // FS_MMC_CM_HW_USE_DMA == 0

#if FS_MMC_CM_HW_USE_DMA

/*********************************************************************
*
*       _StartDMATransfer
*
*  Function description
*    Starts a DMA data transfer.
*/
static void _StartDMATransfer(void) {
  U32        NumBytes;
  DMA_DESC * pDesc;
  int        i;
  U8       * pAddr;
  U32        NumBytesAtOnce;

  pDesc    = _aDMADesc;
  pAddr    = (U8 *)_pData;
  NumBytes = _NumBlocks * _BlockSize;
  for (i = 0; i < FS_MMC_CM_HW_NUM_DMA_DESC; ++i) {
    if (_RepeatSame == 0) {
      NumBytesAtOnce = SEGGER_MIN(NumBytes, MAX_DMA_LEN);
    } else {
      NumBytesAtOnce = SEGGER_MIN(NumBytes, MAX_BLOCK_SIZE);
    }
    pDesc->Addr      = (U32)pAddr;
    pDesc->NumBytes  = NumBytesAtOnce;
    pDesc->Attr      = (DMA_ATTR_ACT_TRAN << DMA_ATTR_ACT)
                     | (1uL               << DMA_ATTR_VALID)
                     ;
    if (_RepeatSame == 0) {
      pAddr         += NumBytesAtOnce;
    }
    NumBytes        -= NumBytesAtOnce;
    if (NumBytes == 0) {
      pDesc->Attr   |= (1uL << DMA_ATTR_INT)
  // Set the DMA interrupt flag at the end of transfer.
                     |  (1uL << DMA_ATTR_END)
                     ;
      break;
    }
    ++pDesc;
  }
  SDHC_ADSADDR  = (U32)_aDMADesc;
  SDHC_PROCTL  &= ~(PROCTL_DMAS_MASK << PROCTL_DMAS);
  SDHC_PROCTL  |=  (PROCTL_DMAS_DMA2 << PROCTL_DMAS);
}
```



```
/*********************************************************************
*
*       _WaitForEndOfDMARead
*/
static int _WaitForEndOfDMARead(void) {
  U32 Status;
  U32 TimeOut;

  TimeOut = WAIT_TIMEOUT_CYCLES;
  for (;;) {
    Status = SDHC_IRQSTAT;
    if ((Status & (1uL << IRQSTAT_DTOE)) != 0u) {
      return FS_MMC_CARD_READ_TIMEOUT;            // Error, data timeout.
    }
    if ((Status & (1uL << IRQSTAT_DCE)) != 0u) {
      return FS_MMC_CARD_READ_CRC_ERROR;          // Error, CRC check failed.
    }
    if ((Status & (1uL << IRQSTAT_DMAE)) != 0u) {
      return FS_MMC_CARD_READ_GENERIC_ERROR;      // Error, DMA transfer failed.
    }
    if ((Status & (1uL << IRQSTAT_DINT)) != 0u) {
      return FS_MMC_CARD_NO_ERROR;                // OK, DMA transfer complete.
    }
    if (TimeOut-- == 0u) {
      return FS_MMC_CARD_READ_TIMEOUT;            // Error, data timeout.
    }
  }
}

/*********************************************************************
*
*       _WaitForEndOfDMAWrite
*/
static int _WaitForEndOfDMAWrite(void) {
  U32 Status;
  U32 TimeOut;

  TimeOut = WAIT_TIMEOUT_CYCLES;
  for (;;) {
    Status = SDHC_IRQSTAT;
    if ((Status & (1uL << IRQSTAT_DTOE)) != 0u) {
      return FS_MMC_CARD_WRITE_GENERIC_ERROR;     // Error, data timeout.
    }
    if ((Status & (1uL << IRQSTAT_DCE)) != 0u) {
      return FS_MMC_CARD_WRITE_CRC_ERROR;         // Error, CRC check failed.
    }
    if ((Status & (1uL << IRQSTAT_DMAE)) != 0u) {
      return FS_MMC_CARD_WRITE_GENERIC_ERROR;     // Error, DMA transfer failed.
    }
    if ((Status & (1uL << IRQSTAT_DINT)) != 0u) {
      return FS_MMC_CARD_NO_ERROR;                // OK, DMA transfer complete.
    }
    if (TimeOut-- == 0u) {
      return FS_MMC_CARD_WRITE_GENERIC_ERROR;     // Error, data timeout.
    }
  }
}

#endif // FS_MMC_CM_HW_USE_DMA

/*********************************************************************
*
*       Public code (via callback)
*
**********************************************************************
*/

/*********************************************************************
*
*       _HW_Init
*
*  Function description
*    Initialize the SD / MMC host controller.
*
*  Parameters
```



```
*    Unit     Index of the SD / MMC host controller (0-based).
*/
static void _HW_Init(U8 Unit) {
  int i;

  FS_USE_PARA(Unit);
  //
  // Enable the clock of GPIOs and of SDHC
  //
  SIM_SCGC3 |= 1uL << SCGC3_SDHC;
  SIM_SCGC5 |= 1uL << SCGC5_PORTE;
  //
  // Clock line
  //
  PORTE_PCR2  = (PCR_MUX_SDHC << PCR_MUX)
              | (1uL          << PCR_PE)
              | (1uL          << PCR_PS)
              | (1uL          << PCR_DSE)
              ;
  //
  // Command line
  //
  PORTE_PCR3  = (PCR_MUX_SDHC << PCR_MUX)
              | (1uL          << PCR_PE)
              | (1uL          << PCR_PS)
              | (1uL          << PCR_DSE)
              ;
  //
  // Data 0 line
  //
  PORTE_PCR1  = (PCR_MUX_SDHC << PCR_MUX)
              | (1uL          << PCR_PE)
              | (1uL          << PCR_PS)
              | (1uL          << PCR_DSE)
              ;
  //
  // Data 1 line
  //
  PORTE_PCR0  = (PCR_MUX_SDHC << PCR_MUX)
              | (1uL          << PCR_PE)
              | (1uL          << PCR_PS)
              | (1uL          << PCR_DSE)
              ;
  //
  // Data 2 line
  //
  PORTE_PCR5  = (PCR_MUX_SDHC << PCR_MUX)
              | (1uL          << PCR_PE)
              | (1uL          << PCR_PS)
              | (1uL          << PCR_DSE)
              ;
  //
  // Data 3 line
  //
  PORTE_PCR4  = (PCR_MUX_SDHC << PCR_MUX)
              | (1uL          << PCR_PE)
              | (1uL          << PCR_PS)
              | (1uL          << PCR_DSE)
              ;
  //
  // Write protect is controlled by the HW layer.
  //
  PORTE_PCR27 = (PCR_MUX_GPIO << PCR_MUX)
              | (1uL          << PCR_PE)
              | (1uL          << PCR_PS)
              ;
  //
  // Card detect is controlled by the HW layer
  //
  PORTE_PCR6  = (PCR_MUX_GPIO << PCR_MUX)
              | (1uL          << PCR_PE)
              | (1uL          << PCR_PS)
              ;
  //
  // Disable the digital filtering on all port pins assigned to SDHC.
  //
```



```
  PORTE_DFER &= ~((1uL << SD_CD_PIN)  |
                  (1uL << SD_CLK_PIN) |
                  (1uL << SD_CMD_PIN) |
                  (1uL << SD_D0_PIN)  |
                  (1uL << SD_D1_PIN)  |
                  (1uL << SD_D2_PIN)  |
                  (1uL << SD_D3_PIN));
#if FS_MMC_CM_HW_USE_DMA
  //
  // Give DMA access to system memory.
  //
  MPU_RGDAAC0 |= (1uL << RGD_M5RE)
              |  (1uL << RGD_M5WE)
              ;
#endif
  //
  // Configure SDHC.
  //
  SDHC_SYSCTL = SYSCTL_RSTA;    // Reset peripheral.
  SDHC_SYSCTL = 0;              // Start peripheral.
  SDHC_PROCTL = PROCTL_EMODE_LE  << PROCTL_EMODE;
  SDHC_VENDOR = 0;              // No external DMA requests.
  //
  // Wait for the power to stabilize before the first access to SD card
  //
  for (i = 0; i < FS_MMC_CM_HW_POWER_GOOD_DELAY; ++i) {
    _Delay1ms();
  }
}

/*********************************************************************
*
*       _HW_Delay
*
*  Function description
*    Blocks the execution for the specified time.
*
*  Parameters
*    ms   Number of milliseconds to delay.
*/
static void _HW_Delay(int ms) {
  int i;

  for (i = 0; i < ms; ++i) {
    _Delay1ms();
  }
}

/*********************************************************************
*
*       _HW_IsPresent
*
*  Function description
*    Returns the state of the media. If you do not know the state, return
*    FS_MEDIA_STATE_UNKNOWN and the higher layer will try to figure out if
*    a media is present.
*
*  Parameters
*    Unit     Index of the SD / MMC host controller (0-based).
*
*  Return value
*    FS_MEDIA_STATE_UNKNOWN     The state of the media is unknown
*    FS_MEDIA_NOT_PRESENT       No card is present
*    FS_MEDIA_IS_PRESENT        A card is present
*/
static int _HW_IsPresent(U8 Unit) {
  int r;

  FS_USE_PARA(Unit);
  r = (GPIOE_PDIR & (1uL << SD_CD_PIN)) ? FS_MEDIA_IS_PRESENT : FS_MEDIA_NOT_PRESENT;
  return r;
}

/*********************************************************************
*
*       _HW_IsWriteProtected
```



```
*
*  Function description
*    Returns whether card is write protected or not.
*
*  Parameters
*    Unit     Index of the SD / MMC host controller (0-based).
*
*  Notes
*    (1) The emPower V2 board does not have an WP detect pin.
*/
static int _HW_IsWriteProtected(U8 Unit) {
  FS_USE_PARA(Unit);
  return 0;           // Note 1
}

/*********************************************************************
*
*       _HW_SetMaxSpeed
*
*  Function description
*    Sets the frequency of the MMC/SD card controller.
*    The frequency is given in kHz.
*
*  Parameters
*    Unit     Index of the SD / MMC host controller (0-based).
*
*  Additional information
*    This function is called two times:
*    1. During card initialization
*       Initialize the frequency to not more than 400kHz.
*
*    2. After card initialization
*       The CSD register of card is read and the max frequency
*       the card can operate is determined.
*       [In most cases: MMC cards 20MHz, SD cards 25MHz]
*/
static U16 _HW_SetMaxSpeed(U8 Unit, U16 Freq) {
  U32 Prescaler;
  U32 Divisor;
  U32 Factor;
  U32 SDClock;

  FS_USE_PARA(Unit);
  //
  // Limit the communication speed.
  //
  if (Freq > FS_MMC_CM_HW_MAX_SPEED) {
    Freq = FS_MMC_CM_HW_MAX_SPEED;
  }
  SDClock = Freq * 1000;              // Convert to Hz
  Factor  = (FS_MMC_CM_HW_PERIPH_CLOCK + SDClock - 1) / SDClock;
  //
  // Determine the prescaler and the divisor values.
  //
  if (Factor == 0) {
    Factor = 1;
  }
  if (Factor <= SYSCTL_DVS_MAX) {
    Prescaler  = 0;
    Divisor    = Factor - 1;
  } else {
    Prescaler = 1;
    while (1) {
      //
      // Compensate for integer division errors. We must generate a clock frequency <= Freq.
      //
      if (Factor & 1) {
        ++Factor;
      }
      Factor >>= 1;
      if (Factor <= SYSCTL_DVS_MAX) {
        Divisor = Factor - 1;
        break;
      }
      Prescaler <<= 1;
    }
```



```
  }
  //
  // Stop the clock to be able to change its frequency.
  //
  SDHC_SYSCTL &= ~(1uL << SYSCTL_SDCLKEN);
  //
  // Set the prescaler and the divisor.
  //
  SDHC_SYSCTL &= ~((SYSCTL_SDCLKFS_MASK << SYSCTL_SDCLKFS) |
                   (SYSCTL_DVS_MASK     << SYSCTL_DVS));
  SDHC_SYSCTL |= ((Prescaler & SYSCTL_SDCLKFS_MASK) << SYSCTL_SDCLKFS)
              |  ((Divisor & SYSCTL_DVS_MASK)       << SYSCTL_DVS)
              ;
  //
  // Enable the clock to SD card.
  //
  SDHC_SYSCTL |= (1uL << SYSCTL_SDCLKEN)
              |  (1uL << SYSCTL_PEREN)
              |  (1uL << SYSCTL_HCKEN)
              |  (1uL << SYSCTL_IPGEN)
              ;
  //
  // Wait for the clock to stabilize.
  //
  while ((SDHC_PRSSTAT & (1uL << PRSSTAT_SDSTB)) == 0) {
    ;
  }
  //
  // Return the actual clock frequency.
  //
  Factor = 1;
  if (Prescaler) {
    Factor *= Prescaler << 1;
  }
  Factor *= Divisor + 1;
  Freq = FS_MMC_CM_HW_PERIPH_CLOCK / 1000 / Factor;
  return Freq;
}

/*********************************************************************
*
*       _HW_SetResponseTimeOut
*
*  Function description
*    Sets the response time out value given in MMC/SD card cycles.
*
*  Parameters
*    Unit     Index of the SD / MMC host controller (0-based).
*
*  Notes
*    (1) The SD host controller has only a short and a long timeout.
*        We always set the long timeout before we send a command.
*/
static void _HW_SetResponseTimeOut(U8 Unit, U32 Value) {
  FS_USE_PARA(Unit);
  FS_USE_PARA(Value);
  //
  // Note 1
  //
}

/*********************************************************************
*
*       _HW_SetReadDataTimeOut
*
*  Function description
*    Sets the read data time out value given in MMC/SD card cycles.
*
*  Parameters
*    Unit     Index of the SD / MMC host controller (0-based).
*/
static void _HW_SetReadDataTimeOut(U8 Unit, U32 Value) {
  FS_USE_PARA(Unit);

  if (Value > SYSCTL_DTOCV_MAX) {
    Value = SYSCTL_DTOCV_MAX;
```



```
  }
  SDHC_SYSCTL &= ~(SYSCTL_DTOCV_MASK << SYSCTL_DTOCV);
  SDHC_SYSCTL |= Value << SYSCTL_DTOCV;
}

/*********************************************************************
*
*       _HW_SendCmd
*
*  Function description
*    Sends a command to the card.
*
*  Parameters
*    Unit           Index of the SD / MMC host controller (0-based).
*    Cmd            Command number according to [4]
*    CmdFlags       Additional information about the command to execute
*    ResponseType   Type of response as defined in [4]
*    Arg            Command parameter
*/
static void _HW_SendCmd(U8 Unit, unsigned Cmd, unsigned CmdFlags, unsigned ResponseType, U32 Arg) {
  U32 RegValue;

  FS_USE_PARA(Unit);
  _WaitForCmdReady();
  _Reset();
  _RepeatSame = 0;
  if (CmdFlags & FS_MMC_CMD_FLAG_WRITE_BURST_REPEAT) {
    _RepeatSame = 1;
  }
  RegValue = (Cmd & XFERTYP_CMDIDX_MASK) << XFERTYP_CMDIDX;   // Set the command index.
  switch (ResponseType) {
  //
  // No response is expected
  //
  case FS_MMC_RESPONSE_FORMAT_NONE:
  default:
    RegValue |= XFERTYP_RSPTYP_NONE << XFERTYP_RSPTYP;
    break;
  //
  // Short response is expected (48bit)
  //
  case FS_MMC_RESPONSE_FORMAT_R3:
    RegValue |= XFERTYP_RSPTYP_48BIT << XFERTYP_RSPTYP;
    break;

  case FS_MMC_RESPONSE_FORMAT_R1:
    if (CmdFlags & FS_MMC_CMD_FLAG_SETBUSY) {
      RegValue |= XFERTYP_RSPTYP_48BIT_BUSY << XFERTYP_RSPTYP;
    } else {
      RegValue |= XFERTYP_RSPTYP_48BIT      << XFERTYP_RSPTYP;
    }
    RegValue |= 0
             |  (1uL << XFERTYP_CICEN)      // Check the received command index.
             |  (1uL << XFERTYP_CCCEN)      // Check the CRC of response.
             ;
    break;
  //
  // Long response is expected (136bit)
  //
  case FS_MMC_RESPONSE_FORMAT_R2:
    RegValue |= (XFERTYP_RSPTYP_136BIT << XFERTYP_RSPTYP)
             |  (1uL << XFERTYP_CCCEN)      // Check the CRC of response.
             ;
    break;
  }
  //
  // If required, Setup the transfer over data lines.
  //
  if (CmdFlags & FS_MMC_CMD_FLAG_DATATRANSFER) {
    RegValue |= (1uL << XFERTYP_DPSEL)
             ;
    if (_NumBlocks > 1) {
      RegValue |= (1uL << XFERTYP_MSBSEL)
               |  (1uL << XFERTYP_BCEN)
               ;
    }
```



```
    if (CmdFlags & FS_MMC_CMD_FLAG_WRITETRANSFER) {
      _SetWMLWrite(_BlockSize);
    } else {
      _SetWMLRead(_BlockSize);
      RegValue |= (1uL << XFERTYP_DTDSEL);  // Read data from SD card.
    }
    //
    // Configure the size and the number of blocks to transfer
    //
    SDHC_BLKATTR = (_NumBlocks << BLKATTR_BLKCNT) | _BlockSize;
    //
    // Configure the number of data lines to use for the transfer.
    //
    SDHC_PROCTL &= ~(PROCTL_DTW_MASK << PROCTL_DTW);
    if (CmdFlags & FS_MMC_CMD_FLAG_USE_SD4MODE) {
      SDHC_PROCTL |= (PROCTL_DTW_4BIT << PROCTL_DTW);
    }
#if FS_MMC_CM_HW_USE_DMA
    RegValue |= (1uL << XFERTYP_DMAEN);
    _StartDMATransfer();
#endif // FS_MMC_CM_HW_USE_DMA
  }
#if FS_MMC_CM_HW_USE_DMA
  //
  // Enable the status flags we handle in DMA mode.
  //
  SDHC_IRQSTATEN = (1uL << IRQSTAT_CC)
                 | (1uL << IRQSTAT_TC)
                 | (1uL << IRQSTAT_CTOE)
                 | (1uL << IRQSTAT_CCE)
                 | (1uL << IRQSTAT_CEBE)
                 | (1uL << IRQSTAT_CIE)
                 | (1uL << IRQSTAT_DTOE)
                 | (1uL << IRQSTAT_DCE)
                 | (1uL << IRQSTAT_DEBE)
                 | (1uL << IRQSTAT_DINT)
                 | (1uL << IRQSTAT_DMAE)
                 ;
#else
  //
  // Enable the status flags we handle in polling mode.
  //
  SDHC_IRQSTATEN = (1uL << IRQSTAT_CC)
                 | (1uL << IRQSTAT_TC)
                 | (1uL << IRQSTAT_BWR)
                 | (1uL << IRQSTAT_BRR)
                 | (1uL << IRQSTAT_CTOE)
                 | (1uL << IRQSTAT_CCE)
                 | (1uL << IRQSTAT_CEBE)
                 | (1uL << IRQSTAT_CIE)
                 | (1uL << IRQSTAT_DTOE)
                 | (1uL << IRQSTAT_DCE)
                 | (1uL << IRQSTAT_DEBE)
                 ;
#endif
  //
  // This is the initialization delay defined by the SD card specification as:
  // maximum of 1 msec, 74 clock cycles and supply ramp up time.
  // The sequence below sends 80 clock cycles which is enough for
  // the power supply of SD card to raise to Vdd min.
  //
  if (CmdFlags & FS_MMC_CMD_FLAG_INITIALIZE) {
    SDHC_SYSCTL |= 1uL << SYSCTL_INITA;
    while (SDHC_SYSCTL & (1uL << SYSCTL_INITA)) {
      ;
    }
  }
  SDHC_IRQSTAT = ~0uL;                      // Reset all status flags.
  SDHC_CMDARG  = Arg;
  SDHC_XFERTYP = RegValue;                  // Send the command.
}

/*********************************************************************
*
*       _HW_GetResponse
*
```



```
*  Function description
*    Receives the responses that was sent by the card after
*    a command was sent to the card.
*
*  Parameters
*    Unit     Index of the SD / MMC host controller (0-based).
*    pBuffer  User allocated buffer where the response is stored.
*    Size     Size of the buffer in bytes
*
*  Return values
*     FS_MMC_CARD_NO_ERROR               Success
*     FS_MMC_CARD_RESPONSE_CRC_ERROR     CRC error in response
*     FS_MMC_CARD_RESPONSE_TIMEOUT       No response received
*     FS_MMC_CARD_RESPONSE_GENERIC_ERROR  Any other error
*
*  Notes
*     (1) The response data has to  be stored at byte offset 1 since
*         the controller does not provide the first byte of response.
*/
static int _HW_GetResponse(U8 Unit, void * pBuffer, U32 Size) {
  U8           * pData;
  volatile U32 * pReg;
  U32 NumWords;
  U32 Value;
  U32 i;
  int r;

  FS_USE_PARA(Unit);
  r = _WaitForResponse();
  if (r != FS_MMC_CARD_NO_ERROR) {
    _Reset();
    return r;                     // Error
  }
  //
  // The size we get from the upper driver is total response size in bytes.
  // We compute here the number of read accesses to the 32-bit register which
  // holds the response. Take into account that the first byte of the response
  // is not delivered by the hardware
  //
  NumWords = (Size - 1) / 4;
  pData    = (U8 *)pBuffer;
  pReg     = &SDHC_CMDRSP0 + (NumWords - 1);
  //
  // In case of a 136 bit response SDHC does not deliver the checksum byte.
  // By not incrementing the data pointer the bytes are saved at expected positions.
  //
  if (NumWords != 4) {
    ++pData;                      // See note (1)
  }
  for (i = 0; i < NumWords; ++i) {
    Value = *pReg;
    *pData++ = (U8)(Value >> 24);
    *pData++ = (U8)(Value >> 16);
    *pData++ = (U8)(Value >> 8);
    *pData++ = (U8)Value;
    --pReg;
  }
  return FS_MMC_CARD_NO_ERROR;  // OK, valid command received.
}

/*********************************************************************
*
*       _HW_ReadData
*
*  Function description
*    Reads data from the card using the SD / MMC host controller.
*
*  Return values
*    FS_MMC_CARD_NO_ERROR             Success
*    FS_MMC_CARD_READ_CRC_ERROR       CRC error in received data
*    FS_MMC_CARD_READ_TIMEOUT         No data received
*    FS_MMC_CARD_READ_GENERIC_ERROR   Any other error
*/
static int _HW_ReadData(U8 Unit, void * pBuffer, unsigned NumBytes, unsigned NumBlocks) {
#if FS_MMC_CM_HW_USE_DMA
  int r;
```



```
  FS_USE_PARA(Unit);
  FS_USE_PARA(pBuffer);
  FS_USE_PARA(NumBytes);
  FS_USE_PARA(NumBlocks);
  r = _WaitForEndOfDMARead();
  return r;
#else
  U32   NumWords;
  U32 * pData32;
  U32   NumWordsAtOnce;
  U32   WordsPerBlock;
  U32   WMLRead;
  int   r;

  FS_USE_PARA(Unit);
  pData32       = (U32 *)pBuffer;
  WordsPerBlock = NumBytes / 4;
  WMLRead       = _GetWMLRead();
  do {
    NumWords = WordsPerBlock;
    do {
      r = _WaitForRxReady();
      if (r != FS_MMC_CARD_NO_ERROR) {
        _Reset();
        return r;                               // Error
      }
      NumWordsAtOnce  = SEGGER_MIN(NumWords, WMLRead);
      NumWords       -= NumWordsAtOnce;
      do {
        *pData32++ = SDHC_DATPORT;
      } while (--NumWordsAtOnce);
      SDHC_IRQSTAT = 1uL << IRQSTAT_BRR;
  // This flag is only set by SDHC. We have to clear it here to get the next notification.
    } while (NumWords);
  } while (--NumBlocks);
  return FS_MMC_CARD_NO_ERROR;                  // OK, data received.
#endif
}

/*********************************************************************
*
*       _HW_WriteData
*
*  Function description
*    Writes the data to SD / MMC card using the SD / MMC host controller.
*
*  Return values
*    FS_MMC_CARD_NO_ERROR      Success
*    FS_MMC_CARD_READ_TIMEOUT  No data received
*/
static int _HW_WriteData(U8 Unit, const void * pBuffer, unsigned NumBytes, unsigned NumBlocks) {
#if FS_MMC_CM_HW_USE_DMA
  int r;

  FS_USE_PARA(Unit);
  FS_USE_PARA(pBuffer);
  FS_USE_PARA(NumBytes);
  FS_USE_PARA(NumBlocks);
  r = _WaitForEndOfDMAWrite();
  return r;
#else
  U32         NumWords;
  const U32 * pData32;
  int         r;
  U32         NumWordsAtOnce;
  U32         WordsPerBlock;
  U32         WMLWrite;

  FS_USE_PARA(Unit);
  pData32       = (U32 *)pBuffer;
  WordsPerBlock = NumBytes / 4;
  WMLWrite      = _GetWMLWrite();
  do {
    NumWords = WordsPerBlock;
    if (_RepeatSame) {
```



```
      pData32 = (U32 *)pBuffer;
    }
    do {
      r = _WaitForTxReady();
      if (r != FS_MMC_CARD_NO_ERROR) {
        _Reset();
        return r;                               // Error
      }
      NumWordsAtOnce  = SEGGER_MIN(NumWords, WMLWrite);
      NumWords       -= NumWordsAtOnce;
      do {
        SDHC_DATPORT = *pData32++;
      } while (--NumWordsAtOnce);
      SDHC_IRQSTAT = 1uL << IRQSTAT_BWR;
  // This flag is only set by SDHC. We have to clear it here to get the next notification.
    } while (NumWords);
  } while (--NumBlocks);
  return FS_MMC_CARD_NO_ERROR;                  // OK, data sent.
#endif
}

/*********************************************************************
*
*       _HW_SetDataPointer
*
*  Function description
*    Tells the hardware layer where to read data from  or write data to.
*    Some SD host controllers require the address of the data buffer
*    before sending the command to the card, eg. programming the DMA.
*    In most cases this function can be left empty.
*
*  Parameters
*    Unit     Index of the SD / MMC host controller (0-based).
*    p        Data buffer.
*/
static void _HW_SetDataPointer(U8 Unit, const void * p) {
  FS_USE_PARA(Unit);
#if FS_MMC_CM_HW_USE_DMA
  _pData = (void *)p;
  // Cast const away. The same buffer is used also for read operations.
#else
  FS_USE_PARA(p);
#endif
}

/*********************************************************************
*
*       _HW_SetBlockLen
*
*  Function description
*    Sets the block size (sector size) that has to be transferred.
*
*  Parameters
*    Unit     Index of the SD / MMC host controller (0-based).
*/
static void _HW_SetBlockLen(U8 Unit, U16 BlockSize) {
  FS_USE_PARA(Unit);
  _BlockSize = BlockSize;
}

/*********************************************************************
*
*       _HW_SetNumBlocks
*
*  Function description
*    Sets the number of blocks (sectors) to be transferred.
*
*  Parameters
*    Unit     Index of the SD / MMC host controller (0-based).
*/
static void _HW_SetNumBlocks(U8 Unit, U16 NumBlocks) {
  FS_USE_PARA(Unit);
  _NumBlocks = NumBlocks;
}

/*********************************************************************
```



```
*
*       _HW_GetMaxReadBurst
*
*  Function description
*    Returns the number of block (sectors) that can be read at once
*    with a single READ_MULTIPLE_SECTOR command.
*
*  Parameters
*    Unit     Index of the SD / MMC host controller (0-based).
*
*  Return value
*    Number of sectors that can be read at once.
*/
static U16 _HW_GetMaxReadBurst(U8 Unit) {
  FS_USE_PARA(Unit);
  return NUM_BLOCKS_AT_ONCE;
}

/*********************************************************************
*
*       _HW_GetMaxWriteBurst
*
*  Function description
*    Returns the number of block (sectors) that can be written at once
*    with a single WRITE_MULTIPLE_SECTOR command.
*
*  Parameters
*    Unit     Index of the SD / MMC host controller (0-based).
*
*  Return value
*    Number of sectors that can be written at once.
*/
static U16 _HW_GetMaxWriteBurst(U8 Unit) {
  FS_USE_PARA(Unit);
  return NUM_BLOCKS_AT_ONCE;
}

/*********************************************************************
*
*       _HW_GetMaxWriteBurstRepeat
*
*  Function description
*    Returns the number of block (sectors) that can be written at once
*    with a single WRITE_MULTIPLE_SECTOR command and that contain the
*    same data.
*
*  Parameters
*    Unit     Index of the SD / MMC host controller (0-based).
*
*  Return value
*    Number of sectors that can be written at once. The function has
*    to return 0 if the feature is not supported.
*/
static U16 _HW_GetMaxWriteBurstRepeat(U8 Unit) {
  FS_USE_PARA(Unit);
  return NUM_BLOCKS_AT_ONCE_REPEAT_SAME;
}

/*********************************************************************
*
*       FS_MMC_CM_HW_K66_SEGGER_emPower
*/
const FS_MMC_HW_TYPE_CM FS_MMC_CM_HW_K66_SEGGER_emPower = {
  _HW_Init,
  _HW_Delay,
  _HW_IsPresent,
  _HW_IsWriteProtected,
  _HW_SetMaxSpeed,
  _HW_SetResponseTimeOut,
  _HW_SetReadDataTimeOut,
  _HW_SendCmd,
  _HW_GetResponse,
  _HW_ReadData,
  _HW_WriteData,
  _HW_SetDataPointer,
  _HW_SetBlockLen,
```



```
  _HW_SetNumBlocks,
  _HW_GetMaxReadBurst,
  _HW_GetMaxWriteBurst,
  _HW_GetMaxWriteBurstRepeat,
  NULL
};

/************************** End of file ****************************/
```


6.5.5 Troubleshooting

If the driver test fails or if the card cannot be accessed at all, please follow the trouble shooting guidelines below.

6.5.5.1 SPI mode troubleshooting guide

First, verify the SPI configuration. The SPI data transfer has to be configured as follows:

- 8 bits per transfer.
- Most significant bit first.
- Data changes on falling edge.
- Data is sampled on rising edge.

Then verify the waveform of the SPI signals using an oscilloscope. The following reference images were taken with an oscilloscope that was set up to trigger a single time on falling edge of Chip Select (CS) signal. The color coding of the signals in these pictures is as follows:

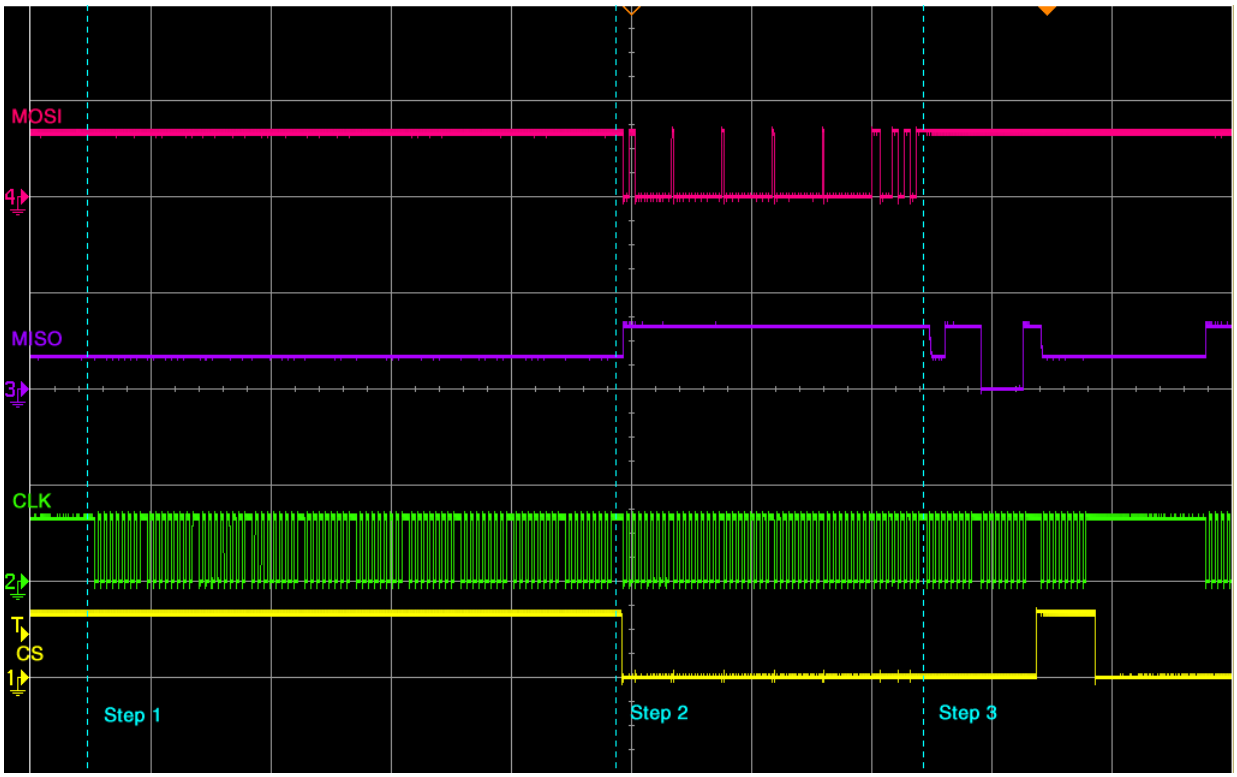
Color	Description
RED	MOSI - Master Out Slave In (Pin 2)
PURPLE	MISO - Master In Slave Out (Pin 7)
GREEN	CLK - Clock (Pin 5)
YELLOW	CS - Chip Select (Pin 1)

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 pictures. 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).

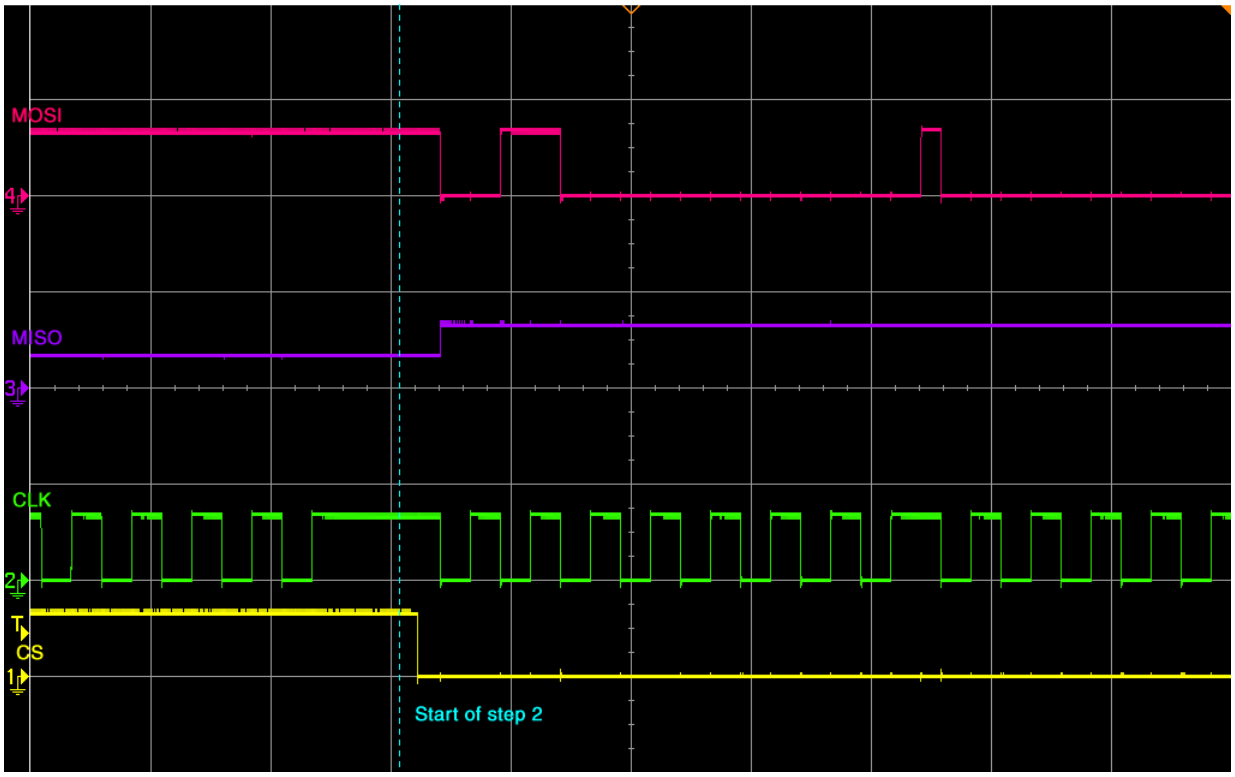
The initial communication sequence consists of the following three parts:

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.

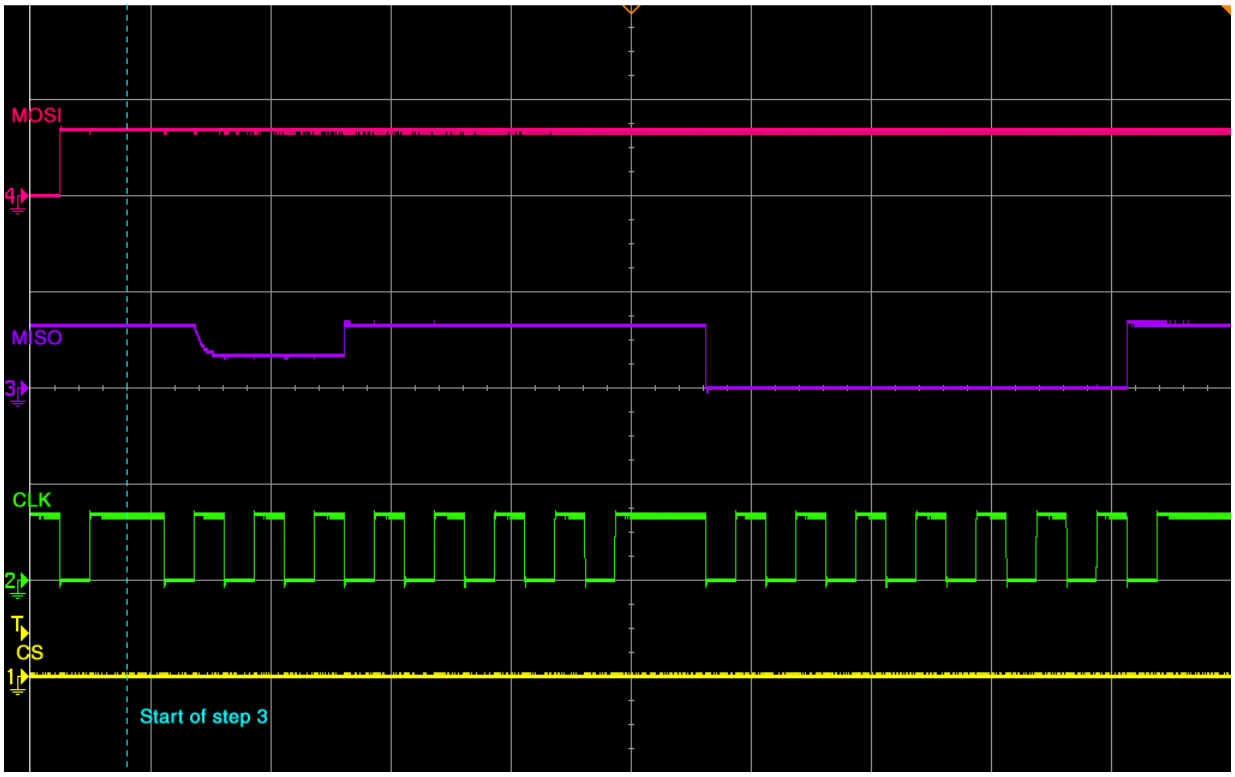
The following picture shows the data flow of a correct initialization of an SD card.



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.



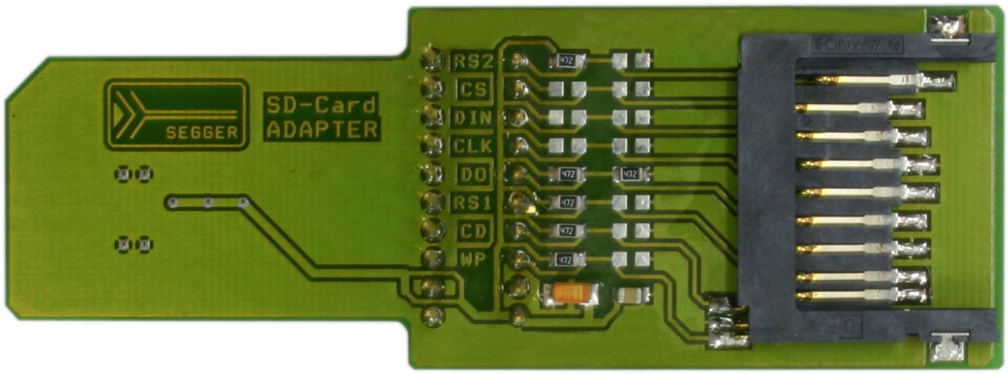
The card responses to a command with two bytes. The SD Card Association defines that the first byte of the response should always be ignored. The second byte is the answer from the card. The answer to GO_IDLE_STATE command should be 0x01. This means that the card is in idle state.



If the card does not return `0x01` then check initialization sequence of your hardware. At the end of the command sequence the CS signal has to be deselected that is the signal has to be set to logic high.

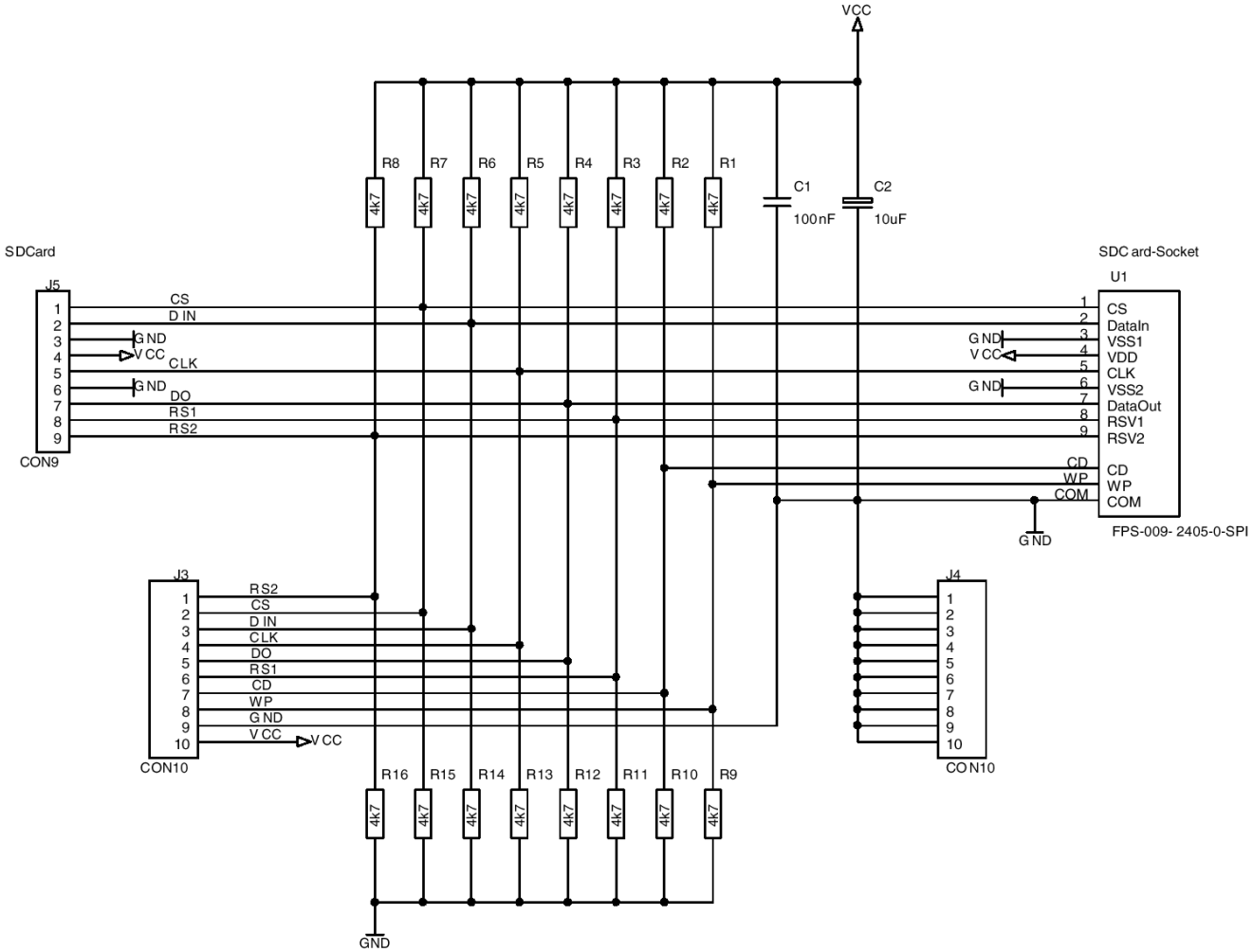
6.5.6 Test hardware

The SEGGER SD Card Adapter is an easy to use measurement board that can be used to troubleshoot problems related to interfacing emFile with MMC and SD cards in SPI as well as card mode.



The adapter board is shaped in the form of an SD card on one side so that it can be directly inserted into any SD card slot. The pads on the same side of the board provide mechanical contact with the signals of SD card slot of the target hardware. The signals are routed on the adapter board to an SD card socket mounted on it where the SD card can be inserted. All the signals are connected to a header where they can be easily captured using an oscilloscope probe. Another header is provided for GND signals.

The picture below shows the schematic of the adapter board.



6.6 CompactFlash card and IDE driver

6.6.1 General information

emFile supports the use of CompactFlash (CF) cards and IDE devices as storage device. Both types of storage devices share a common interface and are supported by the CF/IDE driver that is able to operate in different modes specified by this interface. The access to the hardware is realized via a set of I/O routines called CF/IDE hardware layer.

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 that is responsible for the ATA standard: http://www.t13.org/

6.6.1.1 Fail-safe operation

The data will be preserved in case of an unexpected reset but a power failure can be critical. If the card does not have sufficient time to complete a write operation, data may be lost. As a countermeasure the hardware has to make sure that the power supply to the device drops slowly.

6.6.1.2 Wear-leveling

CF cards and IDE devices have an internal controller, that is responsible for performing the wear-leveling. Therefore, the CF/IDE driver does not need to handle wear-leveling.

6.6.1.3 Supported hardware

The CF/IDE driver can be used to access most ATA HD drives and CF cards using the True IDE or Memory card mode.

6.6.1.3.1 Pin description - True IDE mode

The following table describes the signals used in True IDE mode.

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.
CD1, 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.
CS0, 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	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.

Signal name	Dir	Pin	Description
		47 - 49 2 - 6 21 - 23	
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).
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.
VS1, 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.

6.6.1.3.2 Pin description - Memory card mode

The following table describes the signals used in Memory card mode.

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.
CD1, 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.

Signal name	Dir	Pin	Description
CE1, 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.
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.
VS1, 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.

Signal name	Dir	Pin	Description
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.

6.6.1.4 Theory of operation

6.6.1.4.1 CompactFlash card

A CF card is a mechanically small, removable mass storage device. It contains a single chip controller and one or more flash memory modules 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 modules.

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.

CF cards are designed with flash technology, a nonvolatile storage solution that does not require a battery to retain data indefinitely. The CF 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).

CF cards can operate in three modes:

- Memory card mode
- I/O Card mode
- True IDE mode

Currently, only the True IDE and Memory card mode are supported by the CD/IDE driver.

6.6.1.5 IDE (ATA) Drives

Just like Compact Flash cards, ATA drives have a built-in controller to drive and control the mechanical hardware. 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.

IDE ATA drives can operate in three different modes:

- PIO (Programmed I/O)
- Multiword DMA
- Ultra DMA

Currently, the CF/IDE driver supports only PIO mode via True IDE.

6.6.1.6 Configuring the driver

This section describes how to configure the file system to make use of the CF/IDE driver.

6.6.1.7 Runtime configuration

The driver must be added to the file system by calling `FS_AddDevice()` with the driver identifier set to the address of `FS_IDE_Driver` structure. This function call together with

other function calls that configure the driver operation have to be added to FS_X_AddDevices() as demonstrated in the following example.

```
#include "FS.h"
#include "FS_IDE_HW_TemplateTrueIDE.h"

#define ALLOC_SIZE 0x800                // Size defined 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().
*/
void FS_X_AddDevices(void) {
  //
  // Give the file system memory to work with.
  //
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add and configure the CF/IDE driver in master mode.
  //
  FS_AddDevice(&FS_IDE_Driver);
  FS_IDE_Configure(0, 0);
  FS_IDE_SetHWType(0, &FS_IDE_HW_TemplateTrueIDE);
}
```

The API functions listed in the next table can be used by the application to configure the behavior of the CF/IDE driver. The application can call them only at the file system initialization in FS_X_AddDevices().

Function	Description
FS_IDE_Configure()	Configures a driver instance.
FS_IDE_SetHWType()	Configures the hardware access routines.

6.6.1.7.1 FS_IDE_Configure()

Description

Configures a driver instance.

Prototype

```
void FS_IDE_Configure(U8 Unit,
                      U8 IsSlave);
```

Parameters

Parameter	Description
Unit	Driver index (0-based)
IsSlave	Working mode. • 1 Slave mode. • 0 Master mode.

Additional information

This function is optional. The application has to call this function only when the device does not use the default IDE master/slave configuration. By default, all even-numbered units (0, 2, 4…) work in master mode, while all odd-numbered units work in slave mode.

FS_IDE_Configure function has to be called from FS_X_AddDevices() and it can be called before or after adding the device driver to the file system.

Example

The following example demonstrates how to configure two different CF cards or IDE devices:

```
#include "FS.h"
#include "FS_IDE_HW_TemplateTrueIDE.h"

#define ALLOC_SIZE 0x800                // Size defined 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().
*/
void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add and configure the first driver.
  // The device works as master on the IDE bus. Volume name: "ide:0:"
  //
  FS_AddDevice(&FS_IDE_Driver);
  FS_IDE_Configure(0, 0);
  FS_IDE_SetHWType(0, &FS_IDE_HW_TemplateTrueIDE);
  //
  // Add and configure the second driver.
  // The device works also as master on the IDE bus. Volume name: "ide:1:"
  //
  FS_AddDevice(&FS_IDE_Driver);
  FS_IDE_Configure(1, 0);
  FS_IDE_SetHWType(1, &FS_IDE_HW_TemplateTrueIDE);
}
```


6.6.1.7.2 FS_IDE_SetHWType()

Description

Configures the hardware access routines.

Prototype

```
void FS_IDE_SetHWType(      U8                Unit,
                      const FS_IDE_HW_TYPE * pHWType);
```

Parameters

Parameter	Description
Unit	Driver index (0-based)
pHWType	in Hardware access routines (hardware layer).

Additional information

This function is mandatory. The FS_IDE_HW_Default hardware layer is provided to help the porting to the new hardware layer API. This hardware layer contains pointers to the public functions used by the device driver to access the hardware in the version 3.x of emFile. Configure FS_IDE_HW_Default as hardware layer if you do not want to port your existing hardware layer to the new hardware layer API.

Example

For an example usage refer to FS_IDE_Configure().

6.6.2 CF/IDE hardware layer

The CF/IDE hardware layer provides the functions to access the target hardware (IDE controller, GPIO, etc.). They have to be implemented by the user since they are hardware dependent. emFile comes with template hardware layers and some sample implementations for popular evaluation boards. These files can be found in the Sample/FS/Driver/IDE folder of the emFile shipment. The functions are organized in a function table which is a structure of type FS_IDE_HW_TYPE.

6.6.2.1 Hardware layer API - FS_IDE_HW_TYPE

This hardware layer supports all CF cards and IDE devices. The following sections describe the functions of this hardware layer in detail.

6.6.2.1.1 FS_IDE_HW_TYPE

Description

Hardware layer API

Type definition

```
typedef struct {
  FS_IDE_HW_TYPE_RESET       * pfReset;
  FS_IDE_HW_TYPE_IS_PRESENT  * pfIsPresent;
  FS_IDE_HW_TYPE_DELAY_400NS * pfDelay400ns;
  FS_IDE_HW_TYPE_READ_REG    * pfReadReg;
  FS_IDE_HW_TYPE_WRITE_REG   * pfWriteReg;
  FS_IDE_HW_TYPE_READ_DATA   * pfReadData;
  FS_IDE_HW_TYPE_WRITE_DATA  * pfWriteData;
} FS_IDE_HW_TYPE;
```

Structure members

Member	Description
pfReset	Resets the bus interface.
pfIsPresent	Checks if the device is connected.
pfDelay400ns	Blocks the execution for 400ns.
pfReadReg	Reads the value of a register.
pfWriteReg	Writes the value of a register.
pfReadData	Transfers data from device to MCU.
pfWriteData	Transfers data from MCU do device.

6.6.2.1.2 FS_IDE_HW_TYPE_RESET

Description

Resets the bus interface.

Type definition

```
typedef void FS_IDE_HW_TYPE_RESET(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)

Additional information

This function is a member of the CF/IDE hardware layer. All hardware layers are required to implement this function.

This function is called when the driver detects a new media. For ATA HD drives, there is no action required and this function can be left empty.

6.6.2.1.3 FS_IDE_HW_TYPE_IS_PRESENT

Description

Checks if the device is connected.

Type definition

```
typedef int FS_IDE_HW_TYPE_IS_PRESENT(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)

Return value

≠ 0 Device is connected.
= 0 Device is not connected.

Additional information

This function is a member of the CF/IDE hardware layer. All hardware layers are required to implement this function.

6.6.2.1.4 FS_IDE_HW_TYPE_DELAY_400NS

Description

Blocks the execution for 400ns.

Type definition

```
typedef void FS_IDE_HW_TYPE_DELAY_400NS(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)

Additional information

This function is a member of the CF/IDE hardware layer. All hardware layers are required to implement this function.

The function is always called when a command is sent or parameters are set in the CF card or IDE device to give the integrated logic time to finish a command. When using slow CF cards or IDE devices with fast processors this function should guarantee that a delay of 400ns is respected. However, this function may be left empty if you fast devices are used (modern CF-Cards and IDE drives are faster than 400ns when executing commands.)

6.6.2.1.5 FS_IDE_HW_TYPE_READ_REG

Description

Reads the value of a register.

Type definition

```
typedef U16 FS_IDE_HW_TYPE_READ_REG(U8       Unit,
                                    unsigned AddrOff);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
AddrOff	Offset of the register that has to be read.

Return value

Value read from the register.

Additional information

This function is a member of the CF/IDE hardware layer. All hardware layers are required to implement this function.

A register is 16-bit large.

6.6.2.1.6 FS_IDE_HW_TYPE_WRITE_REG

Description

Writes the value of a register.

Type definition

```
typedef void FS_IDE_HW_TYPE_WRITE_REG(U8       Unit,
                                      unsigned AddrOff,
                                      U16      Data);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
AddrOff	Offset of the register that has to be written.
Data	Value to be written to the specified register.

Additional information

This function is a member of the CF/IDE hardware layer. All hardware layers are required to implement this function.

A register is 16-bit large.

6.6.2.1.7 FS_IDE_HW_TYPE_READ_DATA

Description

Transfers data from device to MCU.

Type definition

```
typedef void FS_IDE_HW_TYPE_READ_DATA(U8       Unit,
                                      U8     * pData,
                                      unsigned NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
pData	out Data transferred from device.
NumBytes	Number of bytes to be transferred.

Additional information

This function is a member of the CF/IDE hardware layer. All hardware layers are required to implement this function.

6.6.2.1.8 FS_IDE_HW_TYPE_WRITE_DATA

Description

Transfers data from MCU do device.

Type definition

```
typedef void FS_IDE_HW_TYPE_WRITE_DATA(      U8       Unit,
                                       const U8     * pData,
                                             unsigned NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the hardware layer instance (0-based)
pData	in Data to be transferred to device.
NumBytes	Number of bytes to be transferred.

Additional information

This function is a member of the CF/IDE hardware layer. All hardware layers are required to implement this function.

6.6.3 Performance and resource usage

6.6.3.1 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 was measured using the SEGGER Embedded Studio IDE V4.20 configured to generate code for a Cortex-M4 CPU in Thumb mode and with the size optimization enabled.

Usage: 1.6 Kbytes

6.6.3.2 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.

Usage: 24 bytes

6.6.3.2.1 Dynamic RAM usage

Dynamic RAM usage is the amount of RAM allocated by the driver at runtime. The amount of RAM required depends on the runtime configuration.

Usage: 18 bytes

6.6.3.3 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. All values are given in Mbytes/sec.

CPU type	Storage device	Write speed	Read speed
LogicPD H79520 (51 MHz)	IDE memory-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**

6.7 WinDrive driver

This driver can be used to access to access the data stored on any drive of a Windows operating system via the emFile API functions. Typical usage includes testing and evaluation emFile before deployment on a target hardware. The WinDrive driver is also able to use a regular file stored on the file system of the host PC as storage.

6.7.1 Supported hardware

The WinDrive driver is compatible with any Windows operating system version newer as Windows 98. Older Windows version not supported, because they do not provide access to a Windows drive via the API functions. Windows drives formatted as NTFS are not supported.

6.7.2 Theory of operation

The driver works by reading and writing entire logical sectors. The size of a logical sector can be specified at compile time via `FS_WINDRIVE_SECTOR_SIZE` or at runtime via `FS_WINDRIVE_SetGeometry()`. The size of the logical sector has to match the size of the logical sector used by the Windows drive. Any logical sector size can be used if the WinDrive driver is configured to use a regular file as storage. On some Windows systems, administrator privileges are required to access a Windows drive. The WinDrive driver reports an error if the user does not have the required privileges.

> **Note**
>
> Do not use this driver on partitions containing important data. WinDrive driver is primarily meant to be used for evaluation purposes. Problems may occur if the program using emFile is debugged or terminated unexpectedly using the Windows 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 perform any wear leveling. The wear leveling is handled by the storage device.

6.7.5 Configuring the WinDrive driver

6.7.5.1 Compile time configuration

The compile time configuration is realized via preprocessor defines that have to be added to the `FS_Conf.h` file which is the main configuration file of emFile. For detailed information about the configuration of emFile and of the configuration define types, refer to *Configuration of emFile* on page 927 The WinDrive driver does not require any hardware access functions. The following table lists the configuration defines supported by the journaling component.

Type	Define	Default value	Description
N	`FS_WINDRIVE_SECTOR_SIZE`	512	Default size of a logical sector in bytes.
N	`FS_WINDRIVE_NUM_UNITS`	4	Maximum number of driver instances.

6.7.5.1.1 FS_WINDRIVE_SECTOR_SIZE

This define specifies the default size of the logical sector when the driver is configured to use a regular file as storage. This value can be modified at runtime via `FS_WINDRIVE_SetGeometry()`

6.7.5.1.2 FS_WINDRIVE_NUM_UNITS

This define specifies the maximum number of driver instances that can be created by the application. The memory for each driver instance is allocated dynamically from the memory pool assigned to file system.

6.7.5.2 Runtime configuration

The driver can be added to the file system by calling `FS_AddDevice()` with the driver label `FS_WINDRIVE_Driver`. This function call together with other function calls that configure the driver operation has to be added to `FS_X_AddDevices()` as demonstrated in the following example.

Example

```
#include "FS.h"

#define FS_ALLOC_SIZE       0x1000          // Memory pool for the file system in bytes

static U32 _aMemBlock[FS_ALLOC_SIZE / 4];   // Memory pool used for semi-dynamic allocation.

void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
#ifdef _WIN32
  //
  // Add and configure the driver to access the Windows drive with the letter 'E'.
  //
  FS_AddDevice(&FS_WINDRIVE_Driver);
  FS_WINDRIVE_Configure(0, "\\\\.\\E:");
#endif // _WIN32
#if FS_SUPPORT_FILE_BUFFER
  //
  // Enable the file buffer to increase the performance when reading/writing a small number of bytes.
  //
  FS_ConfigFileBufferDefault(512, FS_FILE_BUFFER_WRITE);
#endif // FS_SUPPORT_FILE_BUFFER
}
```

The API functions listed in the next table can be used by the application to configure the behavior of the WinDrive driver. The application can call them only at the file system initialization in `FS_X_AddDevices()`.

Function	Description
FS_WINDRIVE_Configure()	Configures a driver instance.
FS_WINDRIVE_ConfigureEx()	Configures a driver instance.
FS_WINDRIVE_SetGeometry()	Configures the storage capacity of an image file.

6.7.5.2.1 FS_WINDRIVE_Configure()

Description

Configures a driver instance.

Prototype

```
void FS_WINDRIVE_Configure(      U8     Unit,
                           const char * sName);
```

Parameters

Parameter	Description
Unit	Index of the instance to configure (0-based)
sName	in Name of the Windows drive or of the image file to be used as storage. Can be NULL.

Additional information

Either FS_WINDRIVE_Configure() or FS_WINDRIVE_ConfigureEx() has to be called once for each instance of the WINDRIVE driver. sName is a 0-terminated wide char string that stores the path to the Windows drive or to the image file to be used as storage.

If sName is set to NULL the driver shows a dialog box that allows the user to select a specific drive from a list a list of available Windows drives. If sName is a path to a regular file that file has to exists before FS_WINDRIVE_ConfigureEx() is called. Selecting a Windows drive as storage requires administrator privileges. The file system reports an error to the application if this is not the case and the application will not be able to access the to use the Windows drive as storage.

The size of the logical sector used by the WinDrive driver can be configured at compile time via FS_WINDRIVE_SECTOR_SIZE or at runtime via FS_WINDRIVE_SetGeometry().

Example

Refer to *Runtime configuration* on page 875 for a sample usage.

6.7.5.2.2 FS_WINDRIVE_ConfigureEx()

Description

Configures a driver instance.

Prototype

```
void FS_WINDRIVE_ConfigureEx(U8      Unit,
                             LPCWSTR sName);
```

Parameters

Parameter	Description
Unit	Index of the instance to configure (0-based)
sName	in Name of the Windows drive or of the image file to be used as storage. Can be NULL.

Additional information

This function performs the same operation as FS_WINDRIVE_Configure() with the difference that sName is a pointer to a 0-terminated string containing wide characters. FS_WINDRIVE_ConfigureEx() has to be used when the path to the Windows drive or image file can contain non-ASCII characters.

sName has to point to 0-terminated string containing wide characters. A string literal can be declared in C by prefixing it with the 'L' character. For example the path to the drive with the letter 'E' can be specified as L"\\\\.\\E:"

If sName is set to NULL the driver shows a dialog box that allows the user to select a specific drive from a list a list of available Windows drives. If sName is a path to a regular file that file has to exists before FS_WINDRIVE_ConfigureEx() is called. Selecting a Windows drive as storage requires administrator privileges. The file system reports an error to the application if this is not the case and the application will not be able to access the to use the Windows drive as storage.

The size of the logical sector used by the WINDRIVE driver can be configured at compile time via FS_WINDRIVE_SECTOR_SIZE or at runtime via FS_WINDRIVE_SetGeometry().

Example

```
#include "FS.h"

#define FS_ALLOC_SIZE       0x1000          // Memory pool for the file system in bytes

static U32 _aMemBlock[FS_ALLOC_SIZE / 4];   // Memory pool used for semi-dynamic allocation.

void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
#ifdef _WIN32
  //
  // Add and configure the driver to access the Windows drive with
  // the letter 'E' assigned to it.
  //
  FS_AddDevice(&FS_WINDRIVE_Driver);
  FS_WINDRIVE_ConfigureEx(0, L"\\\\.\\E:");
#endif // _WIN32
#if FS_SUPPORT_FILE_BUFFER
  //
  // Enable the file buffer to increase the performance when reading/writing a small number of bytes.
  //
  FS_ConfigFileBufferDefault(512, FS_FILE_BUFFER_WRITE);
#endif // FS_SUPPORT_FILE_BUFFER
}
```


6.7.5.2.3 FS_WINDRIVE_SetGeometry()

Description

Configures the storage capacity of an image file.

Prototype

```
int FS_WINDRIVE_SetGeometry(U8  Unit,
                            U32 BytesPerSector,
                            U32 NumSectors);
```

Parameters

Parameter	Description
Unit	Index of the instance to configure (0-based)
BytesPerSector	Number of bytes in a logical sector. Has to be a power of 2 value.
NumSectors	Number of logical sectors that can be stored to the image file.

Return value

= 0 OK, parameters set.
≠ 0 Error code indicating the failure reason.

Additional information

This function is optional. When not called the driver uses the sector size configured via FS_WINDRIVE_SECTOR_SIZE. The number of sectors is calculated by dividing the size of the image file to FS_WINDRIVE_SECTOR_SIZE. This implies that by default the driver fails to initialize if the image file is missing.

Calling FS_WINDRIVE_SetGeometry() changes the behavior of the driver during initialization in that the driver will try to create the image file if missing. In addition, if an image file is present the driver checks verifies if the size of the image file matches the size configured via FS_WINDRIVE_SetGeometry() and if not it recreates the image file.

The size of the image file in bytes is NumSectors * BytesPerSector. Image files larger than or equal to 4 Gbytes are not supported.

Example

```
#include "FS.h"

#define FS_ALLOC_SIZE       0x1000          // Memory pool for the file system in bytes
#define BYTES_PER_SECTOR    512             // Logical sector size in bytes
#define NUM_SECTORS         1024            // Number of logical sectors

static U32 _aMemBlock[FS_ALLOC_SIZE / 4];   // Memory pool used for semi-dynamic allocation.

void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
#ifdef _WIN32
  //
  // Add and configure the driver to use a regular file as storage.
  //
  FS_AddDevice(&FS_WINDRIVE_Driver);
  FS_WINDRIVE_Configure(0, "C:\\Temp\\WinDrive.bin");
  FS_WINDRIVE_SetGeometry(0, BYTES_PER_SECTOR, NUM_SECTORS);
#endif // _WIN32
#if FS_SUPPORT_FILE_BUFFER
  //
  // Enable the file buffer to increase the performance when reading/writing a small number of bytes.
  //
  FS_ConfigFileBufferDefault(512, FS_FILE_BUFFER_WRITE);
#endif // FS_SUPPORT_FILE_BUFFER
```



```
}
```


Chapter 7

Logical drivers

This chapter describes the file system components that provide block processing.

7.1 General information

A logical driver is a file system component that provides block processing of the data exchanged between the Storage layer and the storage device such as partitioning and encryption. It implements the same interface as a device driver but in contrast to a device driver it does not access any storage device. Instead, the requests received from the Storage layer are processed internally and then forwarded to either another logical driver or to a device driver to perform the data access.

The following table lists the available logical drivers.

Driver name	Acronym	Identifier	Volume name
Disk Partition driver	DISKPART	`FS_DISKPART_Driver`	`“diskpart:”`
Encryption driver	CRYPT	`FS_CRYPT_Driver`	`“crypt:”`
Sector Read-Ahead driver	READAHEAD	`FS_READAHEAD_Driver`	`“rah:”`
Sector Size Adapter driver	SECSIZE	`FS_SECSIZE_Driver`	`“secsize:”`
Sector Write Buffer driver	WRBUF	`FS_WRBUF_Driver`	`“wrbuf:”`
RAID1 driver	RAID1	`FS_RAID1_Driver`	`“raid:”`
RAID5 driver	RAID5	`FS_RAID5_Driver`	`“raid:”`
Logical Volume driver	LOGVOL	`FS_LOGVOL_Driver`	`“lvol:”`

A logical driver has to be added to the file system in the same way as a device driver. To add a logical driver to emFile, `FS_AddDevice()` has to be called with the address of the corresponding driver identifier as parameter. For more information about the usage of the names listed in the “Volume name” column refer to *General information* on page 321

7.2 Disk Partition driver

This logical driver can be used to access storage device partitions as defined in a Master Boot Record (MBR). MBR contains information about how the storage device is divided and an optional machine code for bootstrapping PC-compatible computers. It is always stored on the first physical sector of the storage device. 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 Any other value is 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.

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 and sector information as well as 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 has to access only the first storage device partition, as emFile uses this partition by default.

7.2.1 Configuring the driver

7.2.1.1 Runtime configuration

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()` for more information. The following table lists the API functions that can be called by the application to configure the driver.

Function	Description
`FS_DISKPART_Configure()`	Configures the parameters of a driver instance.

7.2.1.1.1 FS_DISKPART_Configure()

Description

Configures the parameters of a driver instance.

Prototype

```
void FS_DISKPART_Configure(      U8                Unit,
                           const FS_DEVICE_TYPE * pDeviceType,
                                 U8                DeviceUnit,
                                 U8                PartIndex);
```

Parameters

Parameter	Description
Unit	Index of the DISKPART instance to configure.
pDeviceType	Type of device driver that is used to access the storage device.
DeviceUnit	Index of the device driver instance that is used to access the storage device (0-based).
PartIndex	Index of the partition in the partition table stored in MBR.

Additional information

This function has to be called once for each instance of the driver. The application can use FS_DISKPART_Configure() to set the parameters that allows the driver to access the partition table stored in Master Boot Record (MBR). The size and the position of the partition are read from MBR on the first access to storage device.

Example

This example demonstrates how to configure the file system to access the first two MBR partitions of an SD card.

```
#include "FS.h"

#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().
*/
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 SEGGER Embedded Studio IDE for Cortex-M V4.20 in Thumb mode and with the highest size optimization enabled.

Usage: 1.5 Kbytes

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 the compiler list file.

Usage: 20 bytes

7.2.2.3 Dynamic RAM usage

Dynamic RAM usage is the amount of RAM allocated by the driver at runtime.

Usage: 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 device.

A separate volume is assigned to each driver instance. The volumes can be accessed using the following names: “crypt:0:”, “crypt:1:”, etc.

7.3.1 Theory of operation

The sector data is transformed to make it unreadable for anyone who 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 content 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.

7.3.2 Configuring the driver

7.3.2.1 Runtime configuration

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() for more information. The following table lists the API functions that can be called by the application to configure the driver.

Function	Description
FS_CRYPT_Configure()	Sets the working parameters of a driver instance.

7.3.2.1.1 FS_CRYPT_Configure()

Description

Sets the working parameters of a driver instance.

Prototype

```
void FS_CRYPT_Configure(      U8                    Unit,
                        const FS_DEVICE_TYPE      * pDeviceType,
                              U8                    DeviceUnit,
                        const FS_CRYPT_ALGO_TYPE  * pAlgoType,
                              void                * pContext,
                        const U8                  * pKey);
```

Parameters

Parameter	Description
Unit	Index of driver to be configured (0-based).
pDeviceType	in Device to access the storage medium.
DeviceUnit	Number of device to access the storage medium (0-based).
pAlgoType	in The type of encryption algorithm.
pContext	in Data specific to cryptographic algorithm.
pKey	in Password for data encryption/decryption.

Additional information

This function has to be called once for each driver instance. pAlgoType is a pointer to one of the following structures:

Algorithm type	Description
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

pContext is passed to as parameter to the encryption and decryption routines via pContext. The memory region pContext points to has to remain valid from the moment the driver is configured until the FS_DeInit() function is called.

The number of bytes in pKey array has to match the size of the key required by the used cryptographic algorithm as follows:

Algorithm type	Size in bytes
FS_CRYPT_ALGO_DES	8
FS_CRYPT_ALGO_AES128	16
FS_CRYPT_ALGO_AES256	32

Example

```
#include "FS.h"

#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().
*/
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);
}
```


7.3.3 Performance and resource usage

7.3.3.1 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 SEGGER Embedded Studio IDE for Cortex-M V4.20 in Thumb mode and with the highest size optimization enabled.

Usage: 1.3 Kbytes

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**

7.3.3.2 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.

Usage: 24 bytes

7.3.3.3 Dynamic RAM usage

Dynamic 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** of RAM. In addition, the context of the AES encryption algorithm requires **480 bytes** and that of the DES encryption algorithm **128 bytes** of RAM.

7.3.3.4 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 992. All values are given in Kbytes/sec.

CPU type	Storage device	Write speed	Read speed
ST STM32F207 (96 MHz)	SD card as storage device using AES with an 128-bit key.	639	661
NXP Kinetis K60 (120 MHz)	NAND flash device interfaced via 8-bit bus using AES with an 128-bit key.	508	550

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 devices 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. The logical driver works with the FAT as well as with the EFS file system.

7.4.1 Configuring the driver

7.4.1.1 Runtime configuration

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() for more information. The following table lists the API functions that can be called by the application to configure the driver.

Function	Description
FS_READAHEAD_Configure()	Sets the parameters which allows the driver instance to access the storage medium.

7.4.1.1.1 FS_READAHEAD_Configure()

Description

Sets the parameters which allows the driver instance to access the storage medium.

Prototype

```
void FS_READAHEAD_Configure(      U8                Unit,
                            const FS_DEVICE_TYPE * pDeviceType,
                                  U8                DeviceUnit,
                                  U32             * pData,
                                  U32               NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based)
pDeviceType	in Device driver used to access the storage device.
DeviceUnit	Index of the storage device (0-based)
pData	in Buffer to store the sector data read from storage device.
NumBytes	Number of bytes in the read buffer.

Additional information

This function is mandatory and it has to be called once for each instance of the driver. The read buffer has to be sufficiently large to store at least one logical sector.

Example

This example demonstrates how to configure the driver in order to access the data stored on an SD card.

```
#include "FS.h"

#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().
*/
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 Additional driver functions

The following table lists the API functions that can be called by the application at runtime to perform operations on the driver.

Function	Description
FS_READAHEAD_GetStatCounters()	Returns the values of the statistical counters.
FS_READAHEAD_ResetStatCounters()	Sets to 0 the values of all statistical counters.

7.4.2.1 FS_READAHEAD_GetStatCounters()

Description

Returns the values of the statistical counters.

Prototype

```
void FS_READAHEAD_GetStatCounters(U8                              Unit,
                                  FS_READAHEAD_STAT_COUNTERS * pStat);
```

Parameters

Parameter	Description
Unit	Driver index (0-based)
pStat	out Values of statistical counters.

Additional information

This function is optional. The statistical counters are updated only when the file system is compiled with FS_READAHEAD_ENABLE_STATS set to 1 or with FS_DEBUG_LEVEL set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL.

7.4.2.2 FS_READAHEAD_ResetStatCounters()

Description

Sets to 0 the values of all statistical counters.

Prototype

```
void FS_READAHEAD_ResetStatCounters(U8 Unit);
```

Parameters

Parameter	Description
Unit	Driver index (0-based)

Additional information

This function is optional. It is available only when the file system is compiled with FS_READAHEAD_ENABLE_STATS set to 1 or with FS_DEBUG_LEVEL set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_ALL.

7.4.2.3 FS_READAHEAD_STAT_COUNTERS

Type definition

```
typedef struct {
  U32  ReadSectorCnt;
  U32  ReadSectorCachedCnt;
} FS_READAHEAD_STAT_COUNTERS;
```

Structure members

Member	Description
ReadSectorCnt	Internal use.
ReadSectorCachedCnt	Internal use.

7.4.3 Performance and resource usage

7.4.3.1 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 SEGGER Embedded Studio IDE for Cortex-M V4.20 in Thumb mode and with the highest size optimization enabled.

Usage: 1.4 Kbytes

7.4.3.2 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 the compiler list file.

Usage: 20 bytes

7.4.3.3 Dynamic RAM usage

Dynamic RAM usage is the amount of RAM allocated by the driver at runtime.

Usage: 30 bytes

7.4.3.4 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 access to a storage device using a sector size different than that of the underlying layer (typically a storage driver). The sector size of the 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()` for more information. The following table lists the API functions that can be called by the application to configure the driver.

Function	Description
`FS_SECSIZE_Configure()`	Sets the parameters of a driver instance.

7.5.1.1 FS_SECSIZE_Configure()

Description

Sets the parameters of a driver instance.

Prototype

```
void FS_SECSIZE_Configure(      U8                Unit,
                          const FS_DEVICE_TYPE * pDeviceType,
                                U8                DeviceUnit,
                                U16               BytesPerSector);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
pDeviceType	in Storage device.
DeviceUnit	Unit number of storage device.
BytesPerSector	Sector size in bytes presented to upper layer.

Additional information

This function is mandatory and it has to be called once for each instance of the driver. BytesPerSector has to 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.

```
#include "FS.h"

#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().
*/
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 SEGGER Embedded Studio IDE for Cortex-M V4.20 in Thumb mode and with the highest size optimization enabled.

Usage: 1.1 Kbytes

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 the compiler list file.

Usage: 20 bytes

7.5.2.3 Dynamic RAM usage

Dynamic RAM usage is the amount of RAM allocated by the driver at runtime.

Each driver instance requires **30 bytes** of RAM. In addition, a driver instance requires a working buffer of the size of one storage driver logical sector if the sector size of the storage driver is larger than the sector size configured for the logical driver configured via `FS_SECSIZE_Configure()`.

7.6 Sector Write Buffer driver

This driver was designed to help improve the write performance of the file system. It operates by temporarily storing the sector data to RAM which takes significantly less time than writing 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.

The advantages of using this driver are:

- a file system write operation blocks for a very short period of time.
- the number of 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

7.6.1.1 Runtime configuration

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()` for more information. The following table lists the API functions that can be called by the application to configure the driver.

Function	Description
`FS_WRBUF_Configure()`	Sets the parameters of a driver instance.

7.6.1.1.1 FS_WRBUF_Configure()

Description

Sets the parameters of a driver instance.

Prototype

```
void FS_WRBUF_Configure(      U8                Unit,
                        const FS_DEVICE_TYPE * pDeviceType,
                              U8                DeviceUnit,
                              void            * pBuffer,
                              U32               NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
pDeviceType	in Storage device.
DeviceUnit	Unit number of storage device.
pBuffer	in Storage for sector data.
NumBytes	Number of bytes in pBuffer.

Additional information

This function is mandatory and it has to be called once for each instance of the driver. FS_SIZEOF_WRBUF() can be used to calculate the number of bytes required to be allocated in order to store a specified number of logical sectors.

Example

This example demonstrates how to configure the logical driver to access an SD card via the SD/MMC card mode driver.

```
#include "FS.h"

#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().
*/
void FS_X_AddDevices(void) {
  FS_AssignMemory(_aMemBlock, sizeof(_aMemBlock));
  //
  // Add and configure the NOR flash driver.
  //
  FS_AddPhysDevice(&FS_MMC_CardMode_Driver);
  FS_MMC_CM_Allow4bitMode(0, 1);
  FS_MMC_CM_SetHWType(0, &FS_MMC_HW_CM_Default);
  //
  // Add and configure the write buffer driver.
  //
  FS_AddDevice(&FS_WRBUF_Driver);
  FS_WRBUF_Configure(0, &FS_MMC_CardMode_Driver, 0,
                     _aWriteBuffer, sizeof(_aWriteBuffer));
}
```


7.6.1.1.2 Write buffer size

Description

Calculates the write buffer size.

Definition

```
#define FS_SIZEOF_WRBUF(NumSectors,    BytesPerSector)
  ((FS_SIZEOF_WRBUF_SECTOR_INFO + (BytesPerSector)) * NumSectors)
```

Symbols

Definition	Description
FS_SIZEOF_WR- BUF(NumSectors,	Calculates the write buffer size.

Additional information

This define can be used in an application to calculate the number of bytes that have to be allocated in order to store the specified number of logical sectors. `NumSectors` specifies the number of logical sectors while `BytesPerSector` the size of a logical sector in bytes.

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 SEGGER Embedded Studio IDE for Cortex-M V4.20 in Thumb mode and with the highest size optimization enabled.

Usage: 1.2 Kbytes

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.

Usage: 20 bytes

7.6.2.3 Dynamic RAM usage

Dynamic RAM usage is the amount of RAM allocated by the driver at runtime.

Usage: 32 bytes

7.7 RAID1 driver

RAID1 is a logical file system driver that can be used to increase the integrity and reliability of the data saved on a storage device. This is realized by keeping a copy of all sector data on a separate partition also known as mirroring. When a read error occurs the data is recovered by reading it from the mirror partition.

7.7.1 Theory of operation

The logical driver uses a master and a mirror partition. On a file system write request, the sector data is written to both master and mirror partitions. On a file system read request, the logical driver tries to read the sector data from the master partition first. If a read error occurs, then the sector data is read from the mirror partition. This way the data is recovered and no error is reported to the file system.

The logical driver can be configured to store the sector data either on the same or on two separate storage devices, according to user requirements. If a single storage device is used, the first half is used as master partition. When using two different storage devices the size of the volumes does not have to be equal. The number of sectors available to the file system will be that of the smallest storage device. However, it is required that the sector size of both storage devices are equal. If necessary, the sector size can be adapted using the SECSIZE logical driver. For more information refer to *Sector Size Adapter driver* on page 897.

7.7.2 NAND flash error recovery

The Universal NAND driver can make use of the RAID driver to avoid a data loss when an uncorrectable 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`.

7.7.3 Sector data synchronization

An unexpected reset that interrupts a write operation may lead to a data inconsistency. It is possible that the data of the last written sector is stored only to the master, but not to the mirror partition. After restart, the file system will continue to operate correctly but in case of a read error affecting this sector, old data is read from the mirror partition which may cause a data corruption. This situation can be prevented by synchronizing all the sectors on the RAID volume. The application can perform the synchronization by calling the `FS_STORAGE_SyncSectors()` API function. For example, a low priority task can call the function in parallel to other file system activities. For an example refer to `FS_RAID1_SetSyncBuffer()`.

7.7.4 Configuring the driver

7.7.4.1 Runtime configuration

The API functions listed in the next table can be used by the application to configure the behavior of the RAID1 driver. The application can call them only at the file system initialization in `FS_X_AddDevices()`.

Function	Description
`FS_RAID1_Configure()`	Configures the parameters of a driver instance.
`FS_RAID1_SetSectorRanges()`	Configures the position and the size of the RAID partitions.
`FS_RAID1_SetSyncBuffer()`	Sets a buffer for the synchronization operation.

Function	Description
FS_RAID1_SetSyncSource()	Specifies the source storage device for the synchronization operation.

7.7.4.1.1 FS_RAID1_Configure()

Description

Configures the parameters of a driver instance.

Prototype

```
void FS_RAID1_Configure(      U8                Unit,
                        const FS_DEVICE_TYPE * pDeviceType0,
                              U8                DeviceUnit0,
                        const FS_DEVICE_TYPE * pDeviceType1,
                              U8                DeviceUnit1);
```

Parameters

Parameter	Description
Unit	Index of the driver instance.
pDeviceType0	in Type of the primary (master) storage device.
DeviceUnit0	Index of the primary (master) storage device (0-based).
pDeviceType1	in Type of the secondary (mirror) storage device.
DeviceUnit1	Index of the secondary (mirror) storage device.

Additional information

This function has to be called once for each driver instance. The same storage device can used both as primary and as secondary device in which case the first half is used as primary device the other half is used as secondary device.

Example

The following example shows how to configure RAID1 on a single volume.

```
#include "FS.h"

#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().
*/
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 RAID1 on 2 separate volumes.

```
#include "FS.h"

#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().
*/
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);
}
```


7.7.4.1.2 FS_RAID1_SetSectorRanges()

Description

Configures the position and the size of the RAID partitions.

Prototype

```
void FS_RAID1_SetSectorRanges(U8  Unit,
                              U32 NumSectors,
                              U32 StartSector0,
                              U32 StartSector1);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
NumSectors	Number of sectors in the partition.
StartSector0	Start sector of first partition (0-based).
StartSector1	Start sector of second partition (0-based).

Additional information

This function is optional. Per default, the RAID1 driver uses the entire available space of a storage device. The application can use FS_RAID1_SetSectorRanges() to specify areas of the primary and secondary storage devices the RAID1 driver has to use as storage.

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.

```
#include "FS.h"

#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().
*/
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);
}
```


7.7.4.1.3 FS_RAID1_SetSyncBuffer()

Description

Sets a buffer for the synchronization operation.

Prototype

```
void FS_RAID1_SetSyncBuffer(U8     Unit,
                            void * pBuffer,
                            U32    NumBytes);
```

Parameters

Parameter	Description
Unit	Index of the driver instance.
pBuffer	Pointer to a memory location to be used as buffer.
NumBytes	Number of bytes in the buffer.

Additional information

This function is optional. The configured buffer is used for when the sector data is synchronized between the primary and the secondary storage device via FS_STORAGE_SyncSectors(). The size of the buffer has to be sufficiently large to store at least two logical sectors otherwise the synchronization operation will fail. A larger buffer allows the driver to read and write multiple logical sectors at once which help increase the performance of the synchronization operation.

Example

The following example shows how to configure a synchronization buffer of 16KB and how to synchronize the RAID volume after an unexpected reset.

```
#include "FS.h"

#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().
*/
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));
}

/*********************************************************************
*
*       _SampleSyncRAID1
*
*  Function description
*    Performs synchronization of the RAID1 partitions. This function may
*    be called from a low-priority task in order to minimize the effect
*    on the normal file system activity.
*/
static void _SampleSyncRAID1(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 application for too long time.
  //
  for (iSector = 0; iSector < DevInfo.NumSectors; ++iSector) {
    FS_STORAGE_SyncSectors("", iSector, 1);
  }
}
```


7.7.4.1.4 FS_RAID1_SetSyncSource()

Description

Specifies the source storage device for the synchronization operation.

Prototype

```
void FS_RAID1_SetSyncSource(U8       Unit,
                            unsigned StorageIndex);
```

Parameters

Parameter	Description
Unit	Index of the driver instance.
StorageIndex	Index of the storage device to be used as source. • 0 - primary storage device. • 1 - secondary storage device.

Additional information

This function is optional. It can be used by an application to specify the storage device that has to be used as source when synchronizing the RAID1 partitions via FS_STORAGE_SyncSectors(). Per default the primary storage device is used as source.

Example

The following example shows how to use the secondary device as source for the synchronization operation.

```
#include "FS.h"

#define ALLOC_SIZE        0x8000            // Size of the memory pool in bytes
#define BUFFER_SIZE       0x8000            // Size of the sync buffer in bytes
#define NUM_SECTORS       2048              // Number of sectors on the RAM disk
#define BYTES_PER_SECTOR  512               // Size of a RAM disk sector
#define BASE_ADDR         0x80000000        // Address in memory of the first byte
                                            // in the NOR flash

static U32 _aMemBlock[ALLOC_SIZE / 4];      // Memory pool used for
                                            // semi-dynamic allocation.
static U32 _aRAMDisk[(NUM_SECTORS * BYTES_PER_SECTOR) / 4];  // Storage for RAM disk.
static U32 _aSyncBuffer[BUFFER_SIZE / 4];   // Buffer for the 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));
  //
  // Add and configure the RAM disk first for maximum read performance.
  //
  FS_AddDevice(&FS_RAMDISK_Driver);
  FS_RAMDISK_Configure(0, _aRAMDisk, BYTES_PER_SECTOR, NUM_SECTORS);
  //
  // Add and configure the driver for NOR flash.
  //
```



```
  FS_AddDevice(&FS_NOR_BM_Driver);
  FS_NOR_BM_SetPhyType(0, &FS_NOR_PHY_CFI_1x16);
  FS_NOR_BM_Configure(0, BASE_ADDR, BASE_ADDR, BYTES_PER_SECTOR * NUM_SECTORS);
  //
  // Add and configure the RAID driver.
  //
  FS_AddDevice(&FS_RAID1_Driver);
  FS_RAID1_Configure(0, &FS_RAMDISK_Driver, 0, &FS_NOR_BM_Driver, 0);
  FS_RAID1_SetSyncBuffer(0, _aSyncBuffer, sizeof(_aSyncBuffer));
  //
  // Copy data from NOR flash to RAM disk when synchronizing at restart.
  //
  FS_RAID1_SetSyncSource(0, 1);
}
```


7.7.5 Performance and resource usage

7.7.5.1 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 SEGGER Embedded Studio IDE for Cortex-M V4.20 in Thumb mode and with the highest size optimization enabled.

Usage: 1.5 Kbytes

7.7.5.2 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.

Usage: 20 bytes

7.7.5.3 Runtime RAM usage

Runtime RAM usage is the amount of RAM allocated by the driver at runtime.

Usage: 48 bytes

7.8 RAID5 driver

RAID5 is a logical file system driver that can be used to increase the integrity and reliability of the data saved on a storage device. This is realized by keeping a parity check of the sector data of two or more consecutive sectors. In case of a read error the sector data is recovered the parity check information.

7.8.1 Theory of operation

RAID5 requires at least three storage partitions. Storage space equivalent to the size of one partition is used for parity check information, making it unavailable for data storage. Parity checking means adding an extra piece of information (a formula result) that can be used to replicate data in the case of a hardware defect or failure. The storage capacity required for the parity checking information can be reduced by increasing the number of partitions, thus increasing overall available capacity. The RAID5 add-on calculates the parity checking information using the "eXclusive Or" function (XOR). The result of an XOR function performed on any number of inputs is such that if one input is lost (i.e. one partition fails) the missing piece of information can be calculated (i.e. recovered or replicated) using the XOR result and the other known inputs.

7.8.2 NAND flash error recovery

The Universal NAND driver can make use of the RAID5 driver to avoid a data loss when an uncorrectable 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` configuration define to 1 in `FS_Conf.h`.

7.8.3 Configuring the driver

7.8.3.1 Runtime configuration

The API functions listed in the next table can be used by the application to configure the behavior of the RAID5 driver. The application can call them only at the file system initialization in `FS_X_AddDevices()`.

Function	Description
`FS_RAID5_AddDevice()`	Configures a storage device for data storage.
`FS_RAID5_SetNumSectors()`	Specifies the number of logical sectors in a single RAID partition.
`FS_RAID5_GetOperatingMode()`	Returns the operating mode of the specified RAID5 partition.

7.8.3.1.1 FS_RAID5_AddDevice()

Description

Configures a storage device for data storage.

Prototype

```
void FS_RAID5_AddDevice(      U8               Unit,
                        const FS_DEVICE_TYPE * pDeviceType,
                              U8               DeviceUnit,
                              U32              StartSector);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
pDeviceType	in Type of storage device.
DeviceUnit	Index of the first storage device (0-based).
StartSector	Index of the first sector to be used for data storage (0-based).

Additional information

This function has be called at least three times in FS_X_AddDevices(). because the RAID5 logical driver requires at least three partitions for correct operation. One partition stores the parity check data while the the other partitions the payload. The partitions do not have to be located on different physical storage devices.

The number of logical sectors covered by a single parity check sector is one less than the total number of partitions added to the RAID5 logical driver. A configuration with one or two partitions is invalid and is reported as an error during the initialization of the RAID5 logical driver instance.

It is mandatory that all the configured partitions have the same logical sector size. If the configured partitions have different logical sector sizes, than the SECSIZE logical driver can be used to adapt them.

The RAID5 logical driver uses the same number of sectors from each partition since the data is distributed equally between partitions. Typically, the RAID5 logical driver uses the entire partition for data storage. This is the case when all configured partitions contain the same number of sectors. When the configured partitions have different number of sectors, then the RAID5 logical driver uses the number of logical sectors of the partition with smallest capacity. The number of logical sectors used by the RAID logical driver from each partition can be also explicitly configured via FS_RAID5_SetNumSectors().

StartSector can be used to locate the RAID partition at other logical sector offset on the storage device than 0, when the first number of logical sectors on the partition are reserved for other purposes.

Example

The following code snippet shows how to configure a the RAID 5 add-on that uses three partitions located on the same NAND flash device.

```
#include "FS.h"
#include "FS_NAND_HW_Template.h"

#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().
*/
void FS_X_AddDevices(void) {
  //
  // Give the file system some memory to work with.
  //
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add and configure the RAID5 volume. Volume name: "raid5:0:"
  //
  FS_AddDevice(&FS_RAID5_Driver);
  FS_RAID5_AddDevice(0, &FS_NAND_UNI_Driver, 0, 0);
  FS_RAID5_AddDevice(0, &FS_NAND_UNI_Driver, 0, 1000);
  FS_RAID5_AddDevice(0, &FS_NAND_UNI_Driver, 0, 2000);
  FS_RAID5_SetNumSectors(0, 1000);
  //
  // Add and configure the driver NAND driver. Volume name: "nand:0:"
  //
  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);
  FS_NAND_ONFI_SetHWType(0, &FS_NAND_HW_Template);
}
```


7.8.3.1.2 FS_RAID5_SetNumSectors()

Description

Specifies the number of logical sectors in a single RAID partition.

Prototype

```
void FS_RAID5_SetNumSectors(U8  Unit,
                            U32 NumSectors);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
NumSectors	Number of sectors in the RAID partition.

Additional information

This function is optional and can be called only from FS_X_AddDevices(). By default, the RAID5 logical driver uses all the logical sectors of a partition to store the data. FS_RAID5_SetNumSectors() can be used to configure a different value for the number of logical sectors when for example the end of a partition is reserved and cannot be used for data storage.

7.8.3.1.3 FS_RAID5_GetOperatingMode()

Description

Returns the operating mode of the specified RAID5 partition.

Prototype

```
int FS_RAID5_GetOperatingMode(U8 Unit);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).

Return value

≥ 0 Operating mode.
< 0 An error occurred.

Additional information

This function is optional and can be called from an application to get the operating mode of the RAID5 logical driver. The RAID5 logical driver can operate in one of the flowing modes:

- FS_RAID_OPERATING_MODE_NORMAL All storage devices are present and are operating normally.
- FS_RAID_OPERATING_MODE_DEGRADED One storage device is not operating properly.
- FS_RAID_OPERATING_MODE_FAILURE Two or more storage devices are not operating properly.

7.8.3.1.4 RAID operating modes

Description

Operating modes of a RAID logical driver.

Definition

```
#define FS_RAID_OPERATING_MODE_NORMAL      0
#define FS_RAID_OPERATING_MODE_DEGRADED    1
#define FS_RAID_OPERATING_MODE_FAILURE     2
```

Symbols

Definition	Description
FS_RAID_OPERATING_MODE_NORMAL	All storage devices present and operating normally.
FS_RAID_OPERATING_MODE_DEGRADED	One storage device is not operating properly.
FS_RAID_OPERATING_MODE_FAILURE	Two or more storage devices are not operating properly.

Additional information

The operating mode of RAID5 logical driver can be queried via the `FS_RAID5_GetOperatingMode()` API function.

7.8.4 Performance and resource usage

7.8.4.1 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 SEGGER Embedded Studio IDE for Cortex-M V4.20 in Thumb mode and with the highest size optimization enabled.

Usage: 2.2 Kbytes

7.8.4.2 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 taken from the compiler list file.

Usage: 30 bytes

7.8.4.3 Dynamic RAM usage

Dynamic RAM usage is the amount of RAM allocated by the driver at runtime.

Each driver instance requires **23 bytes** of RAM. In addition, **11 bytes** of RAM are required for each different storage device used by the RAID5 driver instance and **12 bytes** for each configured partition.

7.9 Logical Volume driver

This driver can be used by an application to create a volume that extents over two or more existing volumes. In addition, the Logical Volume driver can also be configured to create a volume using as storage only a specified sector range of an already existing volume. The volumes created via the Logical Volume driver can be accessed by the application in the same way as any other volume attached to a device or logical driver. The type of the storage device attached to the volumes combined via the Logical Volume driver is not relevant as long as they have the same logical sector size. The combined volumes can be attached to device as well as logical drivers.

The Logical Volume driver is included in the LOGVOL component that also provides the implementation of the `FS_LOGVOL_Create()` and `FS_LOGVOL_AddDevice()` API functions. These functions were the only way for creating logical volumes in emFile versions older than 5.x The functionality provided by the Logical Volume driver is similar to the functionality of the API functions `FS_LOGVOL_Create()` and `FS_LOGVOL_AddDevice()`. The difference is that the volume created using the later method is not attached to a driver and as a consequence the volume cannot be used with any of the logical drivers for example to encrypt its contents via the *Encryption driver* on page 886. Therefore SEGGER recommends the Logical Volume driver as the preferred method of creating logical volumes.

The volume attached to a Logical Volume driver has a fixed name as described in *General information* on page 881. If the application wishes to access the volume under a different name then it can do so by configuring an alias for the volume name via `FS_SetVolumeAlias()`.

7.9.1 Configuring the driver

The Logical Volume driver has to be configured at compile time and at runtime. This section describes how this can be realized.

7.9.1.1 Compile time configuration

The next table lists the available configuration defines followed by a detailed description of them. The configuration defines must be added to the `FS_Conf.h` file which is the main configuration file of emFile. For detailed information about the configuration of emFile and of the configuration define types, refer to *Configuration of emFile* on page 927

Define	Default value	Type	Description
`FS_LOGVOL_NUM_UNITS`	4	N	Maximum number of driver instances.
`FS_LOGVOL_SUPPORT_DRIVER_MODE`	0	B	Configures the working mode of the driver.

7.9.1.1.1 FS_LOGVOL_NUM_UNITS

This define specifies the maximum number of driver instances an application is allowed create. Four bytes of static RAM are reserved for each driver instance. If the maximum number of driver instances is smaller than the default then `FS_LOGVOL_NUM_UNITS` can be set to the to that value in order to reduce the RAM usage.

7.9.1.1.2 FS_LOGVOL_SUPPORT_DRIVER_MODE

`FS_LOGVOL_SUPPORT_DRIVER_MODE` can be used to configure if the LOGVOL component works in legacy or driver mode. By default the component works in legacy mode which means that the application has to use the functions `FS_LOGVOL_Create()` and `FS_LOGVOL_AddDevice()` to configure a volume. If `FS_LOGVOL_SUPPORT_DRIVER_MODE` is set to 1 then the LOGVOL component works in driver mode. In this mode the LOGVOL component behaves like any other logical driver in that an instance of the driver can be added to the file system via `FS_AddDevice()`. Each instance of the Logical Volume driver must be configured at runtime by calling `FS_LOGVOL_AddDeviceEx()` in `FS_X_AddDevice()`.

7.9.1.2 Runtime configuration

The Logical Volume driver has to be added to the file system by calling `FS_AddDevice()` in `FS_X_AddDevices()` with the driver identifier set to `FS_LOGVOL_Driver`. Refer to `FS_X_AddDevices()` for more information. The following table lists the API functions that can be called by the application to configure the driver.

Function	Description
`FS_LOGVOL_AddDeviceEx()`	Adds a storage device to a logical volume.

7.9.1.2.1 FS_LOGVOL_AddDeviceEx()

Description

Adds a storage device to a logical volume.

Prototype

```
int FS_LOGVOL_AddDeviceEx(      U8               Unit,
                          const FS_DEVICE_TYPE * pDeviceType,
                                U8               DeviceUnit,
                                U32              StartSector,
                                U32              NumSectors);
```

Parameters

Parameter	Description
Unit	Index of the driver instance (0-based).
pDeviceType	Type of the storage device that has to be added.
DeviceUnit	Index of the storage device that has to be added (0-based).
StartSector	Index of the first sector that has to be used as storage (0-based).
NumSectors	Number of sectors that have to be used as storage.

Return value

= 0 OK, storage device added.
≠ 0 An error occurred.

Additional information

This function has to be called at least once for each instance of a LOGVOL driver. Each call to FS_LOGVOL_AddDeviceEx() defines a range of sectors to be used a storage from a volume attached to a device or logical driver. The volume to be used as storage is identified by pDeviceType and DeviceUnit. If the defined logical volume extends over two or more existing sectors then all of these volumes need to have the same logical sector size.

If NumSectors is set to 0 then all the sectors of the specified volume are used as storage.

FS_LOGVOL_AddDeviceEx does nothing if FS_LOGVOL_SUPPORT_DRIVER_MODE is set to 0.

Example

This following example demonstrates how to configure the logical driver to use as storage only a sector range of an SD card via the SD/MMC card mode driver. The application can access the volume under the name “sdcard”.

```
#include "FS.h"
#include "FS_MMC_HW_CM_Template.h"

#define ALLOC_SIZE          0x2000        // Size of emFile memory pool
#define START_SECTOR        0             // Index of the first sector to be used.
#define NUM_SECTORS         10000         // Number of sectors in the LOGVOL volume.

static U32 _aMemBlock[ALLOC_SIZE / 4];    // emFile memory pool.

/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*/
void FS_X_AddDevices(void) {
  //
  // Give file system memory to work with.
```



```
  //
  FS_AssignMemory(_aMemBlock, sizeof(_aMemBlock));
  //
  // Add SD/MMC card device driver but do not attach any volume to it.
  //
  FS_AddPhysDevice(&FS_MMC_CM_Driver);
  FS_MMC_CM_Allow4bitMode(0, 1);
  FS_MMC_CM_AllowHighSpeedMode(0, 1);
  FS_MMC_CM_SetHWType(0, &FS_MMC_HW_CM_Template);
  //
  // Add and configure the logical volume. Volume name: "sdcard".
  //
#if FS_LOGVOL_SUPPORT_DRIVER_MODE
  FS_AddDevice(&FS_LOGVOL_Driver);
  FS_LOGVOL_AddDeviceEx(0, &FS_MMC_CM_Driver, 0, START_SECTOR, NUM_SECTORS);
#endif // FS_LOGVOL_SUPPORT_DRIVER_MODE
#if (FS_MAX_LEN_VOLUME_ALIAS > 0)
  FS_SetVolumeAlias("lvol:0:", "sdcard");
#endif // FS_MAX_LEN_VOLUME_ALIAS > 0
}
```

This next example demonstrates how to configure the logical driver to use as storage volumes attached to a NAND and NOR flash device.

```
#include "FS.h"
#include "FS_NAND_HW_Template.h"
#include "FS_NOR_HW_SPIFI_Template.h"

#define ALLOC_SIZE          0x4000        // Size of emFile memory pool
#define BYTES_PER_SECTOR    2048          // Number of bytes in a logical sector.

static U32 _aMemBlock[ALLOC_SIZE / 4];    // emFile memory pool.

/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*/
void FS_X_AddDevices(void) {
  //
  // Give file system memory to work with.
  //
  FS_AssignMemory(_aMemBlock, sizeof(_aMemBlock));
  //
  // Configure the size of the logical sector and activate the file buffering.
  //
  FS_SetMaxSectorSize(BYTES_PER_SECTOR);
  //
  // Add and configure the Universal NAND 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);
  FS_NAND_ONFI_SetHWType(0, &FS_NAND_HW_Template);
  //
  // Add and configure the NOR driver.
  //
  FS_AddPhysDevice(&FS_NOR_BM_Driver);
  FS_NOR_BM_SetPhyType(0, &FS_NOR_PHY_SPIFI);
  FS_NOR_BM_Configure(0, 0x80000000, 0x80000000, 0x01000000);
  FS_NOR_BM_SetSectorSize(0, BYTES_PER_SECTOR);
  FS_NOR_SPIFI_Allow2bitMode(0, 1);
  FS_NOR_SPIFI_Allow4bitMode(0, 1);
  FS_NOR_SPIFI_SetHWType(0, &FS_NOR_HW_SPIFI_Template);
#if FS_LOGVOL_SUPPORT_DRIVER_MODE
  //
  // Add and configure the logical volume.
  // The entire storage space of the NAND and NOR device is used.
  // Volume name: "lvol:0:".
  //
  FS_AddDevice(&FS_LOGVOL_Driver);
  FS_LOGVOL_AddDeviceEx(0, &FS_NAND_UNI_Driver, 0, 0, 0);
  FS_LOGVOL_AddDeviceEx(0, &FS_NOR_BM_Driver, 0, 0, 0);
```



```
#endif // FS_LOGVOL_SUPPORT_DRIVER_MODE
}
```


7.9.2 Performance and resource usage

7.9.2.1 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 SEGGER Embedded Studio IDE for Cortex-M V4.20 in Thumb mode and with the highest size optimization enabled.

Usage: 1.0 Kbytes

7.9.2.2 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.

Usage: 24 bytes

7.9.2.3 Dynamic RAM usage

Dynamic RAM usage is the amount of RAM allocated by the driver at runtime.

Usage: 6 bytes for each driver instance + **21 bytes** for each sector range defined via `FS_LOGVOL_AddDeviceEx()`.

Chapter 8

Configuration of emFile

This chapter describes how to configure emFile at compile time as well as at runtime.

8.1 General information

emFile was designed to be highly configurable in order to meet the different requirements of target applications. Typically, the configuration is performed by the application runtime and optionally by the user at compile time.

emFile can be used without the need for changing any of the configuration defines at compile time. All configuration defines are preconfigured with valid values, that meet the requirements of most the applications. Device and logical drivers do not have to be configured at compile time. The application can added them at runtime as required.

The default configuration of emFile can be changed at compile time by specifying different values for the configuration defines in `FS_Conf.h` that is dedicated for this purpose. This is the main configuration file for the file system. It is not recommended to modify any file system files in order to change the default values of the configuration defines.

8.2 Compile time configuration

This section describes the configuration defines of the File system and Storage layer that can be used to customize the behavior of emFile. The configuration defines of other file system components are described in their respective sections of the emFile manual.

The following types of configuration defines 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 defines.
- 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 device.
- Character values (C) - This is a configuration define that takes a single character as value. A typical example is the character used to separate directory name in a path to file or directory.
- Character strings (S) - A configuration define of this type can take as value a 0-terminated array of characters. A typical usage is a configuration define that specify file name.
- Text replacement (R) - This type of configuration define contains text to be replaced by the preprocessor. It is typically used for functions and variable specifiers the enable compiler specific optimizations.
- Function replacement (F) - Configuration defines of this type are function replacement macros that can also take parameters. They are typical used to substitute at compile time the default implementation of functions used by the file system.

8.2.1 General file system configuration

The following table lists the configuration defines that apply to the entire file system.

Define	Type	Default value	Description
FS_DIRECTORY_DELIMITER	C	'\\'	Character used to delimit directories in a path.
FS_DRIVER_ALIGNMENT	N	4	Defines the minimum data alignment for 0-copy operation.
FS_MAX_LEN_FULL_FILE_NAME	N	256	Maximum number of characters in a fully qualified file name.
FS_MAX_LEN_VOLUME_ALIAS	N	0	Maximum number of characters in a volume alias.
FS_MAX_PATH	N	260	Maximum number of characters in a path.
FS_MULTI_HANDLE_SAFE	B	0	Enables/disables simultaneous read/write access to file.
FS_NUM_DIR_HANDLES	N	1	Maximum number of file system directory handles.
FS_NUM_VOLUMES	N	4	Maximum number of file system volumes.
FS_OPTIMIZE	R		Function specifier for optimized functions.
FS_SUPPORT_BUSY_LED	B	1	Enables/disables support for busy LED.
FS_SUPPORT_CACHE	B	1	Enables/disables support for sector cache.

Define	Type	Default value	Description
FS_SUPPORT_CHECK_MEMORY	B	0	Enables/disables the check for 0-copy operations.
FS_SUPPORT_DEINIT	B	0	Enables/disables support for deinitialization.
FS_SUPPORT_EFS	B	0	Enables/disables support for EFS file system.
FS_SUPPORT_EXT_ASCII	B	0	Enables/disables support for extended ASCII characters in file names.
FS_SUPPORT_EXT_MEM_MANAGER	B	0	Enables/disables support for external memory manager.
FS_SUPPORT_FAT	B	1	Enables/disables support for FAT file system.
FS_SUPPORT_FILE_BUFFER	B	1	Enables/disables support for file buffer.
FS_SUPPORT_FILE_NAME_ENCODING	B	0	Enables/disables support for encoded file names.
FS_SUPPORT_FREE_SECTOR	B	1	Enables/disables support for unused sector notification.
FS_SUPPORT_MBCS	B	0	Enables/disables support for multi-byte character sets in file names.
FS_SUPPORT_POSIX	B	0	Enables/disables support for POSIX compatibility.
FS_SUPPORT_SECTOR_BUFFER_CACHE	B	0	Enables/disables support for caching of internal sector buffers.
FS_SUPPRESS_EOF_ERROR	B	0	Enables/disables end of file error reporting.
FS_VERIFY_BUFFER_SIZE	N	128	Size of the work buffer used by FS_Verify().
FS_VERIFY_WRITE	B	0	Enables/disables write verification.

8.2.1.1 FS_DIRECTORY_DELIMITER

FS_DIRECTORY_DELIMITER defines the character that is used to separate the names of the directories in a path to a file or directory. Typically the value of this define is set either to '\' or to '/'.

8.2.1.2 FS_DRIVER_ALIGNMENT

The value of this define is used by the file system to check if a 0-copy operation can be executed. A 0-copy operation is performed only when the address of the data is aligned to FS_DRIVER_ALIGNMENT and the number of data bytes in the block is a multiple of the logical sector size. If the data to be read or written is not aligned to FS_DRIVER_ALIGNMENT then the file system uses the internal sector buffers for the data transfers which is typically slower in comparison with a 0-copy operation.

8.2.1.3 FS_MAX_LEN_FULL_FILE_NAME

This configuration define is used when FS_MULTI_HANDLE_SAFE is set to 1 to reserve space for the fully qualified file name in the file object assigned to an opened file. The value

specified via this configuration define is also used by the BigFile component to reserves space in the file handle for the name of the opened file.

8.2.1.4 FS_MAX_LEN_VOLUME_ALIAS

This configuration define specifies the maximum number of characters allowed in a volume alias including the 0-terminator. If set to 0 the volume alias feature is disabled. A volume alias can be set via `FS_SetVolumeAlias()`. The RAM usage of the file system increases if the volume alias feature is enabled.

8.2.1.5 FS_MAX_PATH

`FS_MAX_PATH` specifies the maximum number of characters in a path to a file or directory including the 0-terminator. It is used for example by `FS_CreateDir()` for allocating a work buffer for path processing.

8.2.1.6 FS_MULTI_HANDLE_SAFE

This configuration define has to be set to 1 if the application is expected to write and read from a file at the same time. Activating this feature increases the RAM usage of the file system. For more information refer to *Resource usage* on page 986.

8.2.1.7 FS_NUM_DIR_HANDLES

The value of this configuration define specifies the maximum of directory handles that can be opened at the same time via `FS_OpenDir()`. Please note that `FS_OpenDir()` is deprecated. It is recommended that the applications use `FS_FindFirstFile()` and `FS_FindNextFile()` instead for listing the contents of a directory.

8.2.1.8 FS_NUM_VOLUMES

`FS_NUM_VOLUMES` specifies the maximum number of volumes the file system can handle. This value is currently used only by the Journaling component and it represents the maximum number of volumes on which the journal can be enabled.

8.2.1.9 FS_OPTIMIZE

The declaration of some of the computing intensive file system functions is prefixed with this configuration define. The application can set it to a compiler specific specifier that makes the function runs faster such as executing the function from RAM.

8.2.1.10 FS_SUPPORT_BUSY_LED

`FS_SUPPORT_BUSY_LED` can be used to enable or disable the support for a callback function that is invoked by the file system at the beginning and at the end of a sector read or write operation. As the name of this configuration define suggests the registered callback function can be used to indicate that the file system is busy accessing the storage device by turning an LED on and off. The callback function must be registered at runtime via `FS_SetBusyLEDCallback()`

8.2.1.11 FS_SUPPORT_CACHE

This configuration define can be used to enable or disable the support for sector cache. The sector cache must be enabled at runtime by the application. Refer to *Sector cache* on page 300 for more information about this.

8.2.1.12 FS_SUPPORT_CHECK_MEMORY

`FS_SUPPORT_CHECK_MEMORY` can be used to enable or disable additional checks on the data subject to 0-copy operations. The check is performed by a callback function registered by the application via `FS_SetMemAccessCallback()`. If `FS_SUPPORT_CHECK_MEMORY` is set to 1

then the callback is invoked on each read and write operation to check if a 0-copy operation can be performed.

8.2.1.13 FS_SUPPORT_DEINIT

By default the file system does not free any resources it allocates at runtime such as dynamic memory. By setting FS_SUPPORT_DEINIT to 1 support is enabled in the file system for freeing the allocated resources. This can be useful when the application does not longer use the file system and the resources are required for other purposes. The application can free the resources at runtime by calling FS_DeInit(). If the application wishes to use the file system again it has to call FS_Init() before doing this in order to reallocate the resources.

8.2.1.14 FS_SUPPORT_EFS

This configuration define can be used to enable or disable the support for the EFS file system. The support for EFS file system is disabled by default.

8.2.1.15 FS_SUPPORT_EXT_ASCII

FS_SUPPORT_EXT_ASCII can be used to enable or disable the support for extended ASCII characters in the file names. Extended ASCII characters are characters with a value between 128 and 255. Typically, this is the range where the language-specific characters are stored. The file system requires this information for the case insensitive comparison of the file names. If the application uses file names that contain only ASCII characters that is with a value between 0 and 127 then FS_SUPPORT_EXT_ASCII can be set to 0 which is the default. Setting FS_SUPPORT_EXT_ASCII to 1 increases the ROM usage of the file system.

8.2.1.16 FS_SUPPORT_EXT_MEM_MANAGER

This configuration define can be used to enable or disable the support for external memory allocation. By default the file system uses a very simple internal memory allocator that is suitable for any embedded application. If FS_SUPPORT_EXT_MEM_MANAGER is set to 1 then the application can configure at runtime a pair of callback functions for the management of the memory allocation. This can be realized vis FS_SetMemHandler().

8.2.1.17 FS_SUPPORT_FAT

This configuration define can be used to enable or disable the support for the FAT file system. The support for FAT file system is enabled by default.

8.2.1.18 FS_SUPPORT_FILE_BUFFER

This configuration define controls the activation of the file buffer functionality. File buffers make the access to a file faster when the application is reading or writing small amounts of data. Enabling this functionality increases the ROM and RAM usage of the file system. The file buffer functionality has to be explicitly enabled at runtime. For more information refer to *File buffer* on page 319.

8.2.1.19 FS_SUPPORT_FILE_NAME_ENCODING

FS_SUPPORT_FILE_NAME_ENCODING can be used to enable the support for encoded file names such as UTF-8 and Shift JIS. The encoding type of the file and directory names has to be configured at runtime via FS_FAT_SetLFNConverter() or FS_EFS_SetFileNameConverter()

8.2.1.20 FS_SUPPORT_FREE_SECTOR

If FS_SUPPORT_FREE_SECTOR is set to 1 then the file system informs the device and logical drivers about the unused sectors. For example when a file is deleted the file system sends a notification to the driver indicating that the range or ranges of sectors that were used to store the file contents are no longer in use. The driver can use this information to optimize

the management of data. Not all the drivers make use of this information. This functionality is similar to the TRIM command of the ATA command set.

8.2.1.21 FS_SUPPORT_MBCS

This configuration define can be used to enable the support for file and directory names that use multi-byte character sets. Enabling this feature increases the ROM usage of the file system. The type of character set used by the file system can be configured at runtime via `FS_SetCharSetType()`.

8.2.1.22 FS_SUPPORT_POSIX

This configuration define can be used to change the behavior of the API functions so that they work in the same way as their equivalent POSIX function. The POSIX behavior is enabled by setting `FS_SUPPORT_POSIX` to 1. The behavior can also be changed at runtime using `FS_ConfigPOSIXSupport()`.

8.2.1.23 FS_SUPPORT_SECTOR_BUFFER_CACHE

The `FS_SUPPORT_SECTOR_BUFFER_CACHE` configuration define can be used to activate a cache that stores information about the data present in the internal sector buffers of the file system. Enabling this feature can help accelerate file system operations that access the same logical sector such as the creation or removal of files located in the same directory.

8.2.1.24 FS_SUPPRESS_EOF_ERROR

`FS_SUPPRESS_EOF_ERROR` can be used to configure the file system to not generate an error in case of an end-of-file condition. An end-of-file condition occurs when the application is trying to read past the end of file. If `FS_SUPPRESS_EOF_ERROR` is set to 1 then no `FS_ERRCODE_EOF` error is reported via `FS_FError()`.

8.2.1.25 FS_VERIFY_BUFFER_SIZE

`FS_Verify()` uses a work buffer located on the stack for reading the data from the file to be verified. `FS_VERIFY_BUFFER_SIZE` can be used to specify the size of this work buffer.

8.2.1.26 FS_VERIFY_WRITE

`FS_VERIFY_WRITE` can be used for testing purposes to verify if the sector data was correctly written to storage. If the configuration define is set to 1 the file system reads back and compares the contents of each written logical sector to the sector data actually written. Enabling this functionality can have a negative impact on the writer? performance of the file system. Therefore SEGGER recommends to enable it only when investigating a problem related to the file system. The functionality can be enabled and disabled at runtime via `FS_ConfigWriteVerification()`. The write verification requires a work buffer of the logical sector size that is allocated dynamically at runtime.

8.2.2 FAT configuration

This section describes the configuration defines specific to the FAT file system.

Define	Type	Default value	Description
`FS_FAT_LFN_LOWER_CASE_SHORT_NAMES`	B	1	Enables/disables optimization for short file names.
`FS_FAT_LFN_BIT_ARRAY_SIZE`	N	256	Maximum number of short file name indexes to handle at a time.
`FS_FAT_LFN_MAX_SHORT_NAME`	N	1000	Maximum index of a short file name.

Define	Type	Default value	Description
FS_FAT_LFN_UNICODE_CONV_DEFAULT	R	&FS_UNICODE_CONV_CP437	Configures the default Unicode file name converter.
FS_FAT_OPTIMIZE_DELETE	B	1	Enables/disables optimization for large file removal.
FS_FAT_PERMIT_RO_FILE_MOVE	B	0	Allows/denies removal of read-only file names.
FS_FAT_SUPPORT_FAT32	B	1	Enables/disables support for FAT32.
FS_FAT_SUPPORT_FREE_CLUSTER_CACHE	B	1	Enables/disables support for free cluster cache.
FS_FAT_UPDATE_DIRTY_FLAG	B	0	Enables/disables support for boot sector dirty flag.
FS_FAT_USE_FSINFO_SECTOR	B	1	Enables/disables support for FSInfo sector.
FS_MAINTAIN_FAT_COPY	B	0	Enables/disables update of the second allocation table.
FS_UNICODE_UPPERCASE_EXT	R	{0x0000, 0x0000}	Additional entries in the Unicode letter case conversion table.

8.2.2.1 FS_FAT_LFN_LOWER_CASE_SHORT_NAMES

This configuration define specifies how the file system handles the creation of files or directories with names consisting only of small letters and punctuation that fit in 8.3 format. If FS_FAT_LFN_LOWER_CASE_SHORT_NAMES is set to 1 then the file system creates only a short directory entry for the file or directory name if it consists only of small letters and punctuation characters. This is done in order to save storage space. The information about the case of the letters in the name is preserved. If FS_FAT_LFN_LOWER_CASE_SHORT_NAMES is set to 0 then in addition to the short entry a long file name entry is generated. The described handling is available only when the support for long file name is enabled via FS_FAT_SupportLFN().

8.2.2.2 FS_FAT_LFN_BIT_ARRAY_SIZE

FS_FAT_LFN_BIT_ARRAY_SIZE can be used to specify the size of the work buffer of the functionality that generates short file names from long file names. The value represents size in bits of the array used to remember the index of the short file names found. The work buffer is allocated on the stack.

8.2.2.3 FS_FAT_LFN_MAX_SHORT_NAME

The short names generated automatically by the file system for each long file name have a prefix that consists of the first characters of the long file name followed by an generated index. FS_FAT_LFN_MAX_SHORT_NAME can be used to specify the maximum value of that index. The maximum index value of a short file name is FS_FAT_LFN_MAX_SHORT_NAME + FS_FAT_LFN_BIT_ARRAY_SIZE - 1.

8.2.2.4 FS_FAT_LFN_UNICODE_CONV_DEFAULT

FS_FAT_LFN_UNICODE_CONV_DEFAULT specifies the set of routines to be used for the conversion of file name characters to Unicode. It can be set to NULL in order to save ROM space if the file names are encoded using another code page. In this case, the application must specify a Unicode converter at runtime via FS_FAT_SetLFNConverter().

8.2.2.5 FS_FAT_OPTIMIZE_DELETE

FS_FAT_OPTIMIZE_DELETE can be used to enable or disable an optimization related to the deletion of large files. If FS_FAT_OPTIMIZE_DELETE is set to 1 the delete operation is accelerated by updating the allocation table at once if possible. FS_FAT_OPTIMIZE_DELETE can be set to 0 for applications that do not deal with large files in order to reduce the ROM usage.

8.2.2.6 FS_FAT_PERMIT_RO_FILE_MOVE

By default the files and directories with the read-only file attribute set can not be moved or renamed via the API functions FS_Move() and FS_Rename() respectively. If FS_FAT_PERMIT_RO_FILE_MOVE is set to 1 then the read-only file attribute is ignored and the files and directories with this attribute set can be moved and renamed. This behavior can also be modified at runtime via FS_FAT_ConfigROFileMovePermission()

8.2.2.7 FS_FAT_SUPPORT_FAT32

FS_FAT_SUPPORT_FAT32 can be used to enable or disable the handling of FAT32 formatted volumes. If FS_FAT_SUPPORT_FAT32 is set to 0 then the file system will not be able to recognize volumes formatted as FAT32. In this case the file system will also not be able to format a volume as FAT32.

8.2.2.8 FS_FAT_SUPPORT_FREE_CLUSTER_CACHE

This configuration define can be used to enable or disable an optimization related to the handling of the allocation table. If FS_FAT_SUPPORT_FREE_CLUSTER_CACHE is set to 1 then the file system caches the information about which clusters are free in order to reduce the number of accesses to the allocation table. This functionality is used only in the FAST write mode. In the SAFE or MEDIUM write modes the functionality is not used even when enabled at compile time. FS_FAT_SUPPORT_FREE_CLUSTER_CACHE can be set to 0 in order to reduce the ROM usage.

8.2.2.9 FS_FAT_UPDATE_DIRTY_FLAG

If FS_FAT_UPDATE_DIRTY_FLAG is set 1 then the file system sets a dirty flag in the Boot Parameter Block (BPB) of the FAT volume on the first write operation to indicate that the volume has been modified and that the file system may not have updated all the information to the storage device. The dirty flag is cleared later when FS_Unmount(), FS_UnmountForced(), FS_Sync(), or FS_DeInit() is called. The application can query the volume dirty flag using FS_GetVolumeInfo(). A dirty flag set to 1 at file system mount indicates that the file system was not properly unmounted before reset. In this case, it is recommended to call FS_CheckDisk() to check the integrity of the file system structure. The feature can also be enabled or disabled at runtime via FS_FAT_ConfigDirtyFlagUpdate(). FS_FAT_UPDATE_DIRTY_FLAG can be set to 0 to reduce the ROM usage.

8.2.2.10 FS_FAT_USE_FSINFO_SECTOR

FS_FAT_USE_FSINFO_SECTOR can be used to enable or disable the handling of the information stored to FSInfo sector of a volume formatted as FAT32. The FSInfo sector stores information about the number of free clusters and the index of the first free cluster. This information can be used to accelerate the calculation of available free space. If the information stored in the FSInfo sector is not valid then the file system falls back to calculating the free space by evaluating the entire contents of the allocation table that may take a relatively long time to complete for volumes with a large capacity. The handling of the FSInfo sector can be enabled or disabled at runtime vie FS_FAT_ConfigFSInfoSectorUse(). FS_FAT_USE_FSINFO_SECTOR can be set to 0 to reduce ROM usage.

8.2.2.11 FS_MAINTAIN_FAT_COPY

This configuration define can be used to enable or disable the update of the backup allocation table of a FAT file system. It FS_MAINTAIN_FAT_COPY is set to 1 then the file system updates the main as well as the backup allocation table. The backup allocation table is

not evaluated by the emFile but some other file systems my fail to mount a volume it the backup allocation is not present and up to date. This feature can also be enabled or disabled at runtime via FS_FAT_ConfigFATCopyMaintenance(). Setting FS_MAINTAIN_FAT_COPY to 0 increases the write performance and reduces the ROM usage.

8.2.2.12 FS_UNICODE_UPPERCASE_EXT

The long file name component of the file system uses a table allocated in ROM for the case conversion of Unicode letters. In order to save ROM space this table does not contain entries for every possible letter supported by Unicode. FS_UNICODE_UPPERCASE_EXT can be used to extend this table with new entries. Each entry is an initializer for a C structure containing two 16-bit values. The first value is the code point of the small letter and the second value is the code point of corresponding capital letter. The entries are separated by comma characters. The last entry must always be {0x0000, 0x0000} which is the end of table indicator.

8.2.3 EFS configuration

This section describes the configuration defines specific to the EFS file system.

Define	Type	Default value	Description
FS_EFS_CASE_SENSITIVE	B	0	Enables/disables file name case sensitivity.
FS_EFS_MAX_DIR_ENTRY_SIZE	N	255	Maximum directory entry size.
FS_EFS_NUM_DIRENTRY_BUFFERS	N	2	Maximum number of directory entry differs.
FS_EFS_OPTIMIZE_DELETE	B	0	Enables/disables optimization for large file removal.
FS_EFS_SUPPORT_DIREN-TRY_BUFFERS	B	1	Enables/disables support for directory entry buffers.
FS_EFS_SUP-PORT_FREE_CLUSTER_CACHE	B	0	Enables/disables support for free cluster cache.
FS_EFS_SUPPORT_STATUS_SECTOR	B	1	Enables/disables support for status sector.

8.2.3.1 FS_EFS_CASE_SENSITIVE

FS_EFS_CASE_SENSITIVE specifies if the comparison of file or directory names should consider the case of the letters or not. It FS_EFS_CASE_SENSITIVE is set to 0 then the case of the letters in a file or directory name is not relevant. That is TEST.TXT and test.txt represent the same file. It FS_EFS_CASE_SENSITIVE is set to 1 the TEST.TXT, test.txt and TEST.txt represent different file names. This feature can also be enabled or disabled at runtime via FS_EFS_ConfigCaseSensitivity().

8.2.3.2 FS_EFS_MAX_DIR_ENTRY_SIZE

FS_EFS_MAX_DIR_ENTRY_SIZE can be used to specify the maximum directory entry size. The minimum size of a directory entry is 21 bytes while the maximum is 255 bytes. The first 20 bytes in the directory store the size of the file, attributes, time stamps, etc. The remaining bytes are used to store the file name.

8.2.3.3 FS_EFS_NUM_DIRENTRY_BUFFERS

FS_EFS_NUM_DIRENTRY_BUFFERS specifies the number of work buffers the file system has to allocate for the handling of directory entries. The value of this configuration define is evaluated only when FS_EFS_SUPPORT_DIRENTRY_BUFFERS is set to 1.

8.2.3.4 FS_EFS_OPTIMIZE_DELETE

FS_EFS_OPTIMIZE_DELETE can be used to enable or disable an optimization related to the deletion of large files. If FS_EFS_OPTIMIZE_DELETE is set to 1 the delete operation is accelerated by updating the allocation table at once if possible. FS_EFS_OPTIMIZE_DELETE can be set to 0 for applications that do not deal with large files in order to reduce the ROM usage.

8.2.3.5 FS_EFS_SUPPORT_STATUS_SECTOR

This configuration define can be used to enable or disable the handling of the Status sector of an EFS formatted volume. The status sector is the equivalent of FSInfo sector for FAT32 formatted volumes. It stores information about the the number of free clusters. This information is used by the file system to accelerate the operation that calculates the available free space. The handling of the Status sector can also be enabled or disabled at runtime via FS_EFS_ConfigStatusSectorSupport().

8.2.3.6 FS_EFS_SUPPORT_DIRENTRY_BUFFERS

FS_EFS_SUPPORT_DIRENTRY_BUFFERS can be used to specify where the work buffers for the handling of directory entries are allocated. If FS_EFS_SUPPORT_DIRENTRY_BUFFERS is set to 1 the the work buffers are allocated from the memory pool of file system. Each work buffer is FS_EFS_MAX_DIR_ENTRY_SIZE + 1 bytes large. If FS_EFS_SUPPORT_DIRENTRY_BUFFERS is set to 0 then the work buffers are allocated on the stack which is the default.

8.2.3.7 FS_EFS_SUPPORT_FREE_CLUSTER_CACHE

This configuration define can be used to enable or disable an optimization related to the handling of the allocation table. If FS_EFS_SUPPORT_FREE_CLUSTER_CACHE is set to 1 then the file system caches the information about which clusters are free in order to reduce the number of accesses to the allocation table. This functionality is used only in the FAST write mode. In the SAFE or MEDIUM write modes the functionality is not used even when enabled at compile time. FS_EFS_SUPPORT_FREE_CLUSTER_CACHE can be set to 0 in order to reduce the ROM usage.

8.2.4 Storage layer configuration

This section describes the configuration defines specific to the Storage layer.

Define	Type	Default value	Description
FS_STORAGE_ENABLE_STAT_COUNTERS	B	0	Enables/disables support for statistical counters.
FS_STORAGE_SUPPORT_DEVICE_ACTIVITY	B	0	Enables/disables the support for device activity callback.

8.2.4.1 FS_STORAGE_ENABLE_STAT_COUNTERS

This configuration define can be used to enable the statistical counters of the Storage layer. The statistical counters are enabled by default when the file system is configured at compile time with FS_DEBUG_LEVEL set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_PARA. The statistical counters can be queried using FS_STORAGE_GetCounters() and cleared using FS_STORAGE_ResetCounters()

8.2.4.2 FS_STORAGE_SUPPORT_DEVICE_ACTIVITY

FS_STORAGE_SUPPORT_DEVICE_ACTIVITY enables or disables the support for a callback that is invoked by the file system at each sector read or write operation. The callback function can be registered by the application at runtime via FS_STORAGE_SetOnDeviceActivityCallback(). The support for the device activity callback is enabled by default at compile time when the file system is configured with FS_DEBUG_LEVEL set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_PARA.

8.2.5 Standard C function replacement

This section describes the configuration defines that can be used to replace the standard C functions called by emFile by other functions.

Define	Type	Default value	Description
FS_MEMCMP(s1, s2, n)	F	memcmp(s1, s2, n)	Replaces memcmp() standard C function.
FS_MEMCPY(s1, s2, n)	F	memcpy(s1, s2, n)	Replaces memcpy() standard C function.
FS_MEMSET(s, c, n)	F	memset(s, c, n)	Replaces memset() standard C function.
FS_MEMXOR(pd, ps, n)	F	SEGGER_memxor(pd, ps, n)	Performs XOR operation on a memory range.
FS_NO_CLIB	B	0	Enables/disables standard C function replacements.
FS_STRCHR(s, c)	F	strchr(s, c)	Replaces strchr() standard C function.
FS_STRCMP(s1, s2)	F	strcmp(s1, s2)	Replaces strcmp() standard C function.
FS_STRCPY(s1, s2)	F	strcpy(s1, s2)	Replaces strcpy() standard C function.
FS_STRLEN(s)	F	strlen(s)	Replaces strlen() standard C function.
FS_STRNCAT(s1, s2, n)	F	strncat(s1, s2, n)	Replaces strncat() standard C function.
FS_STRNCMP(s1, s2, n)	F	strncmp(s1, s2, n)	Replaces strncmp() standard C function.
FS_STRNCPY(s1, s2, n)	F	strncpy(s1, s2, n)	Replaces strncpy() standard C function.
FS_TOLOWER(c)	F	tolower(c)	Replaces tolower() standard C function.
FS_TOUPPER(c)	F	toupper(c)	Replaces toupper() standard C function.
FS_ISLOWER(c)	F	islower(c)	Replaces islower() standard C function.
FS_ISUPPER(c)	F	isupper(c)	Replaces isupper() standard C function.
FS_VSNPRINTF(s, n, f, a)	F	SEGGER_vsnprintf(s, n, f, a)	Replaces vsnprintf() standard C function.

8.2.5.1 FS_NO_CLIB

emFile uses functions of the standard C library to preform specific operations. If FS_NO_CLIB is set to 1 then the file system calls internal functions instead to perform these operations instead of the equivalent standard C library functions.

8.2.6 Sample configuration

A sample configuration file FS_Conf.h can be found in the Config folder of the emFile shipment. If you want to change the default configuration, insert the corresponding configuration defines in this file.

```
/*********************************************************************
*                  (c) SEGGER Microcontroller GmbH                   *
```



```
*                       The Embedded Experts                       *
*                          www.segger.com                          *
**********************************************************************

------------------------- END-OF-HEADER ------------------------------

File    : FS_Conf.h
Purpose : emFile compile time configuration.
*/
#ifndef FS_CONF_H
#define FS_CONF_H               // Avoid multiple inclusion

//
// Enable the EFS file system.
//
#define FS_SUPPORT_FAT      0
#define FS_SUPPORT_EFS      1

#endif                          // Avoid multiple inclusion

/************************** End of file ***************************/
```


8.3 Runtime configuration

The runtime configuration of the file system is realized via a set of functions that have to be implemented by the application according to its requirements. This section describes these functions in detail.

emFile comes with sample implementations that can be used as a starting point for creating a custom file system configuration. Every folder of the emFile shipment that contain sample hardware layer implementations also contain a suitable configuration file (e.g. `Sample/FS/RAM/FS_ConfigRamDisk.c`) with implementations of runtime configuration functions explained in this chapter. The configuration files can be used without modification, to run emFile “out of the box”.

The following table lists the names of the functions that have to be provided by the application followed by a detailed description of these functions. Additional functions are required to be implemented by the application if the emFile is configured to generate debug messages. For additional information refer to *Debug messages* on page 960

Function	Description
`FS_X_AddDevices()`	Configures the file system.
`FS_X_GetTimeDate()`	Returns the current time and date.

8.3.1 FS_X_AddDevices()

Description

Configures the file system.

Prototype

```
void FS_X_AddDevices(void);
```

Additional information

This function is responsible for configuring the file system at runtime and it has to be implemented by any application that uses emFile.

`FS_X_AddDevices()` is called during the file system initialization from either `FS_Init()` or `FS_STORAGE_Init()`. At the minimum, this function has to provide memory for the file system to work with either via `FS_AssignMemory()` or `FS_SetMemHandler()`. In addition, `FS_X_AddDevices()` has to add at least one device driver to the file system via `FS_AddDevice()` and to configure the device driver.

API functions that access a storage device may not be called in `FS_X_AddDevices()` because the device drivers may not be ready at this point to perform any data transfers.

8.3.2 FS_X_GetTimeDate()

Description

Returns the current time and date.

Prototype

```
U32 FS_X_GetTimeDate(void);
```

Return value

Current time and date in a format suitable for the file system.

Additional information

This function is called by the file system in order to generate the time stamps for the accessed files and directories. `FS_X_GetTimeDate()` is not required when the application accesses the storage device only via the Storage layer and not via the File system layer.

Alternatively, the application can register a runtime a callback functionvia `FS_SetTimeDateCallback()` that returns the date and time.

The return value is formatted as follows:

Bit range	Description
0-4	2-second count (0-29)
5-10	Minutes (0-59)
11-15	Hours (0-23)
16-20	Day of month (1-31)
21-24	Month of year (1-12)
25-31	Count of years from 1980 (0-127)

Chapter 9

OS integration

This chapter provides information about how to configure emFile to work in a multitasking application.

9.1 General information

emFile does not require an operating system to work. However, when an application uses two or more tasks to concurrently access the file system then these accesses have to be synchronized in order to prevent a corruption of the file system structure. The access to the file system is synchronized via a set of API functions called OS layer. The implementation of the OS layer is specific to the OS used by the application. emFile comes with ready to use OS layers for SEGGER embOS and Micrium uC/OS. The implementation of these OS layers can be found in the `Sample/FS/OS` folder of the emFile shipment. Support for other operating systems is available on request.

9.2 Compile time configuration

By default, the support for OS is disabled in emFile because the majority of the applications either use only one task or if they use multiple tasks only one task accesses the file system at a time. Therefore, the OS support has to be explicitly enabled at compile time. The following table lists the configuration defines that can be used for this.

Macro	Default	Type	Description
FS_OS_LOCKING	0	N	Configures the OS support

9.2.1 FS_OS_LOCKING

This configuration define can take three values:

Value	Description
0	OS support disabled.
1	Coarse lock granularity.
2	Fine lock granularity.

Setting FS_OS_LOCKING to 0 disables the OS support. This is the default value. The OS layer is not required in this case.

If FS_OS_LOCKING is set to 1 or 2 then the support for OS is enabled and an OS layer is required. With FS_OS_LOCKING set to 1 (coarse lock granularity) the file system requires only one OS synchronization object. The task synchronization is realized at the API layer that is only one task can execute a file system operation at a time. With FS_OS_LOCKING set to 2 (fine lock granularity) an OS synchronization object is required for each device or logical driver type configured by the application. In this mode the task synchronization is realized at driver level therefore if is practical to use this type of locking only when the file system is configured to use different storage device types such as NOR and NAND flash, SD card and NAND flash etc.

9.3 Runtime configuration

emFile requires an OS layer specific to the OS used by the application if FS_OS_LOCKING is set to a value different than 0 at compile time. The OS layer is responsible for providing the OS specific synchronization routines the file system can use to make sure that only one task executes a critical section of the file system.

The following table lists the functions of the OS layer followed by a detailed description of each function.

Function	Description
FS_X_OS_Init()	Allocates the OS layer resources.
FS_X_OS_DeInit()	Releases the OS layer resources.
FS_X_OS_Delay()	Block the execution for the specified time.
FS_X_OS_Lock()	Acquires the specified OS synchronization object.
FS_X_OS_Unlock()	Releases the specified OS synchronization object.
FS_X_OS_Wait()	Waits for an OS synchronization object to be signaled.
FS_X_OS_Signal()	Signals an OS synchronization object.
FS_X_OS_GetTime()	Allocates the OS layer resources.

9.3.1 FS_X_OS_Init()

Description

Allocates the OS layer resources.

Prototype

```
void FS_X_OS_Init(unsigned NumLocks);
```

Parameters

Parameter	Description
NumLocks	Number of OS synchronization objects required.

Additional information

This function has to be implemented by any OS layer. FS_X_OS_Init() is called during the file system initialization. It has to create the number of specified OS synchronization objects. The type of the OS synchronization object is not relevant as long as it can be used to protect a critical section. The file system calls FS_X_OS_Lock() before it enters a critical section and FS_X_OS_Unlock() when the critical sector is leaved.

In addition, FS_X_OS_Init() has to create the OS synchronization object used by the optional functions FS_X_OS_Signal() and FS_X_OS_Wait().

9.3.2 FS_X_OS_DeInit()

Description

Releases the OS layer resources.

Prototype

```
void FS_X_OS_DeInit(void);
```

Additional information

This function has to be implemented only for file system configurations that set `FS_SUPPORT_DEINIT` to 1. `FS_X_OS_DeInit()` has to release all the OS synchronization objects that were allocated in `FS_X_OS_Init()`.

9.3.3 FS_X_OS_Delay()

Description

Block the execution for the specified time.

Prototype

```
void FS_X_OS_Delay(int ms);
```

Parameters

Parameter	Description
ms	Number of milliseconds to block the execution.

Additional information

The implementation of this function is optional. FS_X_OS_Delay() is called by implementations of the hardware layers to block efficiently the execution of a task.

9.3.4 FS_X_OS_Lock()

Description

Acquires the specified OS synchronization object.

Prototype

```
void FS_X_OS_Lock(unsigned LockIndex);
```

Parameters

Parameter	Description
LockIndex	Index of the OS synchronization object (0-based).

Additional information

This function has to be implemented by any OS layer. The file system calls FS_X_OS_Lock() when it tries to enter a critical section that is protected by the OS synchronization object specified via LockIndex. FS_X_OS_Lock() has to block the execution of the calling task until the OS synchronization object can be acquired. The OS synchronization object is later released via a call to FS_X_OS_Unlock(). All OS synchronization objects are created in FS_X_OS_Init().

It is guaranteed that the file system does not perform a recursive locking of the OS synchronization object. That is FS_X_OS_Lock() is not called two times in a row from the same task on the same OS synchronization object without a call to FS_X_OS_Unlock() in between.

9.3.5 FS_X_OS_Unlock()

Description

Releases the specified OS synchronization object.

Prototype

```
void FS_X_OS_Unlock(unsigned LockIndex);
```

Parameters

Parameter	Description
LockIndex	Index of the OS synchronization object (0-based).

Additional information

This function has to be implemented by any OS layer. The OS synchronization object to be released was acquired via a call to FS_X_OS_Lock(). All OS synchronization objects are created in FS_X_OS_Init().

9.3.6 FS_X_OS_Wait()

Description

Waits for an OS synchronization object to be signaled.

Prototype

```
int FS_X_OS_Wait(int TimeOut);
```

Parameters

Parameter	Description
TimeOut	Maximum time in milliseconds to wait for the OS synchronization object to be signaled.

Return value

= 0 OK, the OS synchronization object was signaled within the timeout.
≠ 0 An error or a timeout occurred.

Additional information

The implementation of this function is optional. FS_X_OS_Wait() is called by implementations of the hardware layers that work in event-driven mode. That is a condition is not check periodically by the CPU until is met but the hardware layer calls FS_X_OS_Wait() to block the execution while waiting for the condition to be met. The blocking is realized via an OS synchronization object that is signaled via FS_X_OS_Signal() in an interrupt that is triggered when the condition is met.

9.3.7 FS_X_OS_Signal()

Description

Signals an OS synchronization object.

Prototype

```
void FS_X_OS_Signal(void);
```

Additional information

The implementation of this function is optional. `FS_X_OS_Signal()` is called by implementations of the hardware layers that work in event-driven mode. Refer to `FS_X_OS_Wait()` for more details about how this works.

9.3.8 FS_X_OS_GetTime()

Description

Allocates the OS layer resources.

Prototype

```
U32 FS_X_OS_GetTime(void);
```

Return value

Number of milliseconds since the start of the application.

Additional information

The implementation of this function is optional. `FS_X_OS_GetTime()` is not called by the file system. It is typically used by some test applications for performance measurements.

9.3.9 Sample implementation

The following example shows the implementation of an OS layer for SEGGER embOS. The implementation file is located in the `Sample/FS/OS` folder of the emFile shipment.

```
/*********************************************************************
*                  (c) SEGGER Microcontroller GmbH                   *
*                       The Embedded Experts                         *
*                          www.segger.com                            *
**********************************************************************

------------------------- END-OF-HEADER -----------------------------

File    : FS_OS_embOS.c
Purpose : embOS OS Layer for the file system.
*/

/*********************************************************************
*
*       #include Section
*
**********************************************************************
*/
#include "FS.h"
#include "FS_OS.h"
#include "RTOS.h"

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/
static OS_RSEMA   * _paSema;
static OS_EVENT     _Event;
#if FS_SUPPORT_DEINIT
  static unsigned   _NumLocks;
#endif

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       FS_X_OS_Lock
*
*  Function description
*    Acquires the specified OS resource.
*
*  Parameters
*    LockIndex    Identifies the OS resource (0-based).
*
*  Additional information
*    This function has to block until it can acquire the OS resource.
*    The OS resource is later released via a call to FS_X_OS_Unlock().
*/
void FS_X_OS_Lock(unsigned LockIndex) {
  OS_RSEMA * pSema;

  pSema = _paSema + LockIndex;
  (void)OS_Use(pSema);
  FS_DEBUG_LOG((FS_MTYPE_OS, "OS: LOCK Index: %d\n", LockIndex));
}

/*********************************************************************
*
*       FS_X_OS_Unlock
*
*  Function description
*    Releases the specified OS resource.
*
```



```
*  Parameters
*    LockIndex    Identifies the OS resource (0-based).
*
*  Additional information
*    The OS resource to be released was acquired via a call to FS_X_OS_Lock()
*/
void FS_X_OS_Unlock(unsigned LockIndex) {
  OS_RSEMA * pSema;

  pSema = _paSema + LockIndex;
  FS_DEBUG_LOG((FS_MTYPE_OS, "OS: UNLOCK Index: %d\n", LockIndex));
  OS_Unuse(pSema);
}

/*********************************************************************
*
*       FS_X_OS_Init
*
*  Function description
*    Initializes the OS resources.
*
*  Parameters
*    NumLocks   Number of locks that should be created.
*
*  Additional information
*    This function is called by FS_Init(). It has to create all resources
*    required by the OS to support multi tasking of the file system.
*/
void FS_X_OS_Init(unsigned NumLocks) {
  unsigned   i;
  OS_RSEMA * pSema;
  unsigned   NumBytes;

  NumBytes = NumLocks * sizeof(OS_RSEMA);
  _paSema = SEGGER_PTR2PTR(OS_RSEMA, FS_AllocZeroed((I32)NumBytes));
  pSema =_paSema;
  for (i = 0; i < NumLocks; i++) {
    OS_CREATERSEMA(pSema++);
  }
  OS_EVENT_Create(&_Event);
#if FS_SUPPORT_DEINIT
  _NumLocks = NumLocks;
#endif
}

#if FS_SUPPORT_DEINIT

/*********************************************************************
*
*       FS_X_OS_DeInit
*
*  Function description
*    This function has to release all the resources that have been
*    allocated by FS_X_OS_Init().
*/
void FS_X_OS_DeInit(void) {
  unsigned i;
  OS_RSEMA  * pSema;
  unsigned    NumLocks;

  NumLocks = _NumLocks;
  pSema   = &_paSema[0];
  for (i = 0; i < NumLocks; i++) {
    OS_DeleteRSema(pSema);
    pSema++;
  }
  OS_EVENT_Delete(&_Event);
  FS_Free(_paSema);
  _paSema  = NULL;
  _NumLocks = 0;
}

#endif // FS_SUPPORT_DEINIT

/*********************************************************************
*
```



```
*       FS_X_OS_GetTime
*/
U32 FS_X_OS_GetTime(void) {
  return (U32)OS_GetTime32();
}

/*********************************************************************
*
*       FS_X_OS_Wait
*
*  Function description
*    Wait for an event to be signaled.
*
*  Parameters
*    Timeout  Time to be wait for the event object.
*
*  Return value:
*    ==0      Event object was signaled within the timeout value
*    !=0      An error or a timeout occurred.
*/
int FS_X_OS_Wait(int TimeOut) {
  int r;

  r = -1;
  if ((U8)OS_EVENT_WaitTimed(&_Event, TimeOut) == 0u) {
    r = 0;
  }
  return r;
}

/*********************************************************************
*
*       FS_X_OS_Signal
*
*  Function description
*    Signals a event.
*/
void FS_X_OS_Signal(void) {
  OS_EVENT_Set(&_Event);
}

/*********************************************************************
*
*       FS_X_OS_Delay
*
*  Function description
*    Blocks the execution for the specified number of milliseconds.
*/
void FS_X_OS_Delay(int ms) {
  OS_Delay(ms + 1);
}

/************************** End of file ****************************/
```


Chapter 10

Debugging

This chapter provides information about how to troubleshoot emFile in case that is not working as expected.

10.1 General information

emFile comes with an optional debug functionality that helps investigating potential issues related to the file system operation. The debug functionality can be used by an application to enable additional parameter and consistency checks and to enable the reporting of debug messages. The type of checks to be performed as well as the type of debug messages to be generated can be configured at compile time by assigning a different value (debug level) to *FS_DEBUG_LEVEL* on page 965 configuration define.

> **Note**
>
> Enabling the debug functionality can have a negative effect on the read and write performance and it can also increase the ROM and RAM usage of the file system.

10.2 Debug messages

Debug messages are human-readable text messages generated by the file system on error or warning conditions or when specific internal operations are performed. The file system provides the debug messages as 0-terminated strings to the application via calls to dedicated API functions. These functions have to be implemented in the application and the application is free to choose how to handle the debug messages. For example the debug messages can be forwarded to a host system via the SEGGER Real Time Transfer (https://www.segger.com/products/debug-probes/j-link/technology/about-real-time-transfer/)

The debug messages are organized in three classes: error, warning and trace. Each debug message class is passed to the application via a separate API function. In addition, the reporting of a specific class of debug messages can be enabled or disabled via different debug levels. The table below lists the supported debug message classes together with the API function that is used to pass them to the application.

Class	Handling function	Debug level	Description
error	`FS_X_ErrorOut()`	≥ `FS_DEBUG_LEVEL_LOG_ERRORS`	Fault condition the file system cannot recover from.
warning	`FS_X_Warn()`	≥ `FS_DEBUG_LEVEL_LOG_WARNINGS`	Fault condition the file system can recover from.
trace	`FS_X_Log()`	≥ `FS_DEBUG_LEVEL_LOG_ALL`	Information about internal processing.

The debug messages also have a type assigned to them that is used to identify the component of the file system that reported that message. Refer to section *Debug message types* on page 977 for a description of the supported message types. It is possible to configure at compile time as well as at runtime what type of debug messages are reported by the file system. The sections *FS_LOG_MASK_DEFAULT* on page 966 and *Runtime configuration* on page 946 provide more information about how this is realized.

10.2.1 Debug output and error functions

This section describes the API functions an application has to provide in order to be able to handle the debug messages and assertion errors generated by the file system. emFile comes with a sample implementation for these functions that demonstrate how to send the debug messages to a host system The implementation of the sample functions can be found in the `Config/FS_ConfigIO.c` file of the emFile shipment. These functions can be adapted according to the requirements of the target application. The typical release build does not use debugging and therefore these functions may be removed from the source code to save some ROM space if the linker is not able to eliminate unreferenced functions automatically.

The table below and the next sections provide more detailed information about these API functions.

Function	Description
`FS_X_ErrorOut()`	API function called to report an error debug message.
`FS_X_Log()`	API function called when a trace debug message is generated.
`FS_X_Panic()`	Handles an assertion error.
`FS_X_Warn()`	API function called to report a warning debug message.

10.2.2 FS_X_ErrorOut()

Description

API function called to report an error debug message.

Prototype

```
void FS_X_ErrorOut(const char * s);
```

Parameters

Parameter	Description
s	Reported text debug message (0-terminated). It can never be NULL.

Additional information

This function is optional. It is required to be present only when the file system is built with FS_DEBUG_LEVEL set to a value greater than or equal to FS_DEBUG_LEVEL_LOG_ERRORS. The generated text message is not terminated by a newline character.

Example

```
#include <stdio.h>

void FS_X_ErrorOut(const char * s) {
  printf("FS error: %s\n", s);
}
```


10.2.3 FS_X_Log()

Description

API function called when a trace debug message is generated.

Prototype

```
void FS_X_Log(const char * s);
```

Parameters

Parameter	Description
s	Generated text debug message (0-terminated). It can never be NULL.

Additional information

This function is optional. It is required to be present only when the file system is built with FS_DEBUG_LEVEL set to a value greater than or equal to FS_DEBUG_LEVEL_LOG_ALL. The generated text message terminated by a newline character as required.

Example

```
#include <stdio.h>

void FS_X_Log(const char * s) {
  puts(s);
}
```


10.2.4 FS_X_Panic()

Description

Handles an assertion error.

Prototype

```
void FS_X_Panic(int ErrorCode);
```

Parameters

Parameter	Description
ErrorCode	Identifies the type of error.

Additional information

This function is called by the file system only when the sources are built with FS_DEBUG_LEVEL set to a value greater than or equal to FS_DEBUG_LEVEL_CHECK_PARA. FS_X_Panic() is called only on a critical error condition such as insufficient memory or invalid parameter.

Refer to *Error codes* on page 269 for permitted values for ErrorCode.

Example

```
#include <stdio.h>

void FS_X_Panic(int ErrorCode) {
  printf("FS panic: %d\n", ErrorCode);
  while (1) {
    ;
  }
}
```


10.2.5 FS_X_Warn()

Description

API function called to report a warning debug message.

Prototype

```
void FS_X_Warn(const char * s);
```

Parameters

Parameter	Description
s	Reported text debug message (0-terminated). It can never be NULL.

Additional information

This function is optional. It is required to be present only when the file system is built with FS_DEBUG_LEVEL set to a value greater than or equal to FS_DEBUG_LEVEL_LOG_WARNINGS. The generated text message is not terminated by a newline character.

Example

```
#include <stdio.h>

void FS_X_Warn(const char * s) {
  printf("FS warning: %s\n", s);
}
```


10.3 Configuration

The debug functionality has to be configured at compiled time and can be optionally configured at runtime.

10.3.1 Compile time configuration

The compile time configuration is realized via preprocessor defines that have to be added to the FS_Conf.h file which is the main configuration file of emFile. For detailed information about the configuration of emFile and of the configuration define types, refer to *Configuration of emFile* on page 927 The following table lists the configuration defines supported by the debug functionality.

Define	Type	Default value	Description
FS_DEBUG_LEVEL	N	FS_DEBUG_LEVEL_CHECK_PARA	Default behavior of the debug functionality.
FS_DEBUG_MAX_LEN_MESSAGE	N	100	Size of the format buffer in bytes.
FS_LOG_MASK_DEFAULT	N	FS_MTYPE_INIT	Default debug message types to be traced.
FS_X_PANIC()	F	FS_X_Panic()	Handles an assertion error.

10.3.1.1 FS_DEBUG_LEVEL

This define can be used to configure the behavior of the debug functionality. The application can choose between six debug levels that enable different features of the debug functionality. The debug levels are structured hierarchically so that higher debug levels also enable the features assigned to lower debug levels.

10.3.1.1.1 Debug levels

Description

Permitted values for FS_DEBUG_LEVEL.

Definition

```
#define FS_DEBUG_LEVEL_NOCHECK          0
#define FS_DEBUG_LEVEL_CHECK_PARA       1
#define FS_DEBUG_LEVEL_CHECK_ALL        2
#define FS_DEBUG_LEVEL_LOG_ERRORS       3
#define FS_DEBUG_LEVEL_LOG_WARNINGS     4
#define FS_DEBUG_LEVEL_LOG_ALL          5
```

Symbols

Definition	Description
FS_DEBUG_LEVEL_NOCHECK	No run time checks are performed.
FS_DEBUG_LEVEL_CHECK_PARA	Parameter checks are performed.
FS_DEBUG_LEVEL_CHECK_ALL	Parameter checks and consistency checks are performed.
FS_DEBUG_LEVEL_LOG_ERRORS	Error conditions are reported.
FS_DEBUG_LEVEL_LOG_WARNINGS	Error and warning conditions are reported.
FS_DEBUG_LEVEL_LOG_ALL	Error and warning conditions as well as trace messages are reported.

Additional information

The debug levels are hierarchical so that so that higher debug levels also enable the features assigned to lower debug levels.

10.3.1.2 FS_DEBUG_MAX_LEN_MESSAGE

This define specifies the size of the buffer to be used for the formatting of debug messages. This buffer is allocated on the stack. It can be set to a smaller value than the default to save stack space. The debug messages will be truncated if they do not fit in the formatting buffer.

10.3.1.3 FS_LOG_MASK_DEFAULT

This configuration define specifies the type of debug messages to be generated from the class of log messages. The value is a bitwise-OR combination of the message types described in the section *Debug message types* on page 977. The type of log debug messages to be generated can be modified at runtime via FS_SetLogFilter() and FS_AddLogFilter().

10.3.1.4 FS_X_PANIC()

This configuration define can be used to replace the default handler for assertion errors. It takes the error code as parameter. Refer to FS_X_Panic() for more information.

10.3.2 Runtime configuration

The type of debug messages that are generated by the file system during the operation can be managed using the API functions listed in the following table.

Function	Description
FS_AddErrorFilter()	Enables error debug messages.
FS_AddLogFilter()	Enables trace debug messages.
FS_AddWarnFilter()	Enables warning debug messages.
FS_GetErrorFilter()	Queries activation status of error debug messages.
FS_GetLogFilter()	Queries activation status of trace debug messages.
FS_GetWarnFilter()	Queries activation status of warning debug messages.
FS_SetErrorFilter()	Enables and disables error debug messages.
FS_SetLogFilter()	Enables and disables trace debug messages.
FS_SetWarnFilter()	Enables and disables warning debug messages.

10.3.2.1 FS_AddErrorFilter()

Description

Enables error debug messages.

Prototype

```
void FS_AddErrorFilter(U32 FilterMask);
```

Parameters

Parameter	Description
FilterMask	Specifies the message types to be enabled.

Additional information

FS_AddErrorFilter() can be used to enable a specified set of debug message types of the error class.

This function is optional and is available only when the file system is built with FS_DEBUG_LEVEL set to a value equal or greater than FS_DEBUG_LEVEL_LOG_ERRORS.

FileMask is specified by or-ing one or more message types described at *Debug message types* on page 977.

10.3.2.2 FS_AddLogFilter()

Description

Enables trace debug messages.

Prototype

```
void FS_AddLogFilter(U32 FilterMask);
```

Parameters

Parameter	Description
FilterMask	Specifies the message types to be enabled.

Additional information

FS_AddLogFilter() can be used to enable a specified set of debug message types of the trace class.

This function is optional and is available only when the file system is built with FS_DEBUG_LEVEL set to a value equal or greater than FS_DEBUG_LEVEL_LOG_ALL.

FileMask is specified by or-ing one or more message types described at *Debug message types* on page 977.

10.3.2.3 FS_AddWarnFilter()

Description

Enables warning debug messages.

Prototype

```
void FS_AddWarnFilter(U32 FilterMask);
```

Parameters

Parameter	Description
FilterMask	Specifies the message types to be enabled.

Additional information

FS_AddWarnFilter() can be used to enable a specified set of debug message types of the warning class.

This function is optional and is available only when the file system is built with FS_DEBUG_LEVEL set to a value equal or greater than FS_DEBUG_LEVEL_LOG_WARNINGS.

FileMask is specified by or-ing one or more message types described at *Debug message types* on page 977.

10.3.2.4 FS_GetErrorFilter()

Description

Queries activation status of error debug messages.

Prototype

```
U32 FS_GetErrorFilter(void);
```

Return value

Value indicating the activation status for all debug message types of the error class.

Additional information

This function is optional and is available only when the file system is built with FS_DEBUG_LEVEL set to a value equal or greater than FS_DEBUG_LEVEL_LOG_ERRORS.

The return value is specified by or-ing one or more message types described at *Debug message types* on page 977.

10.3.2.5 FS_GetLogFilter()

Description

Queries activation status of trace debug messages.

Prototype

```
U32 FS_GetLogFilter(void);
```

Return value

Value indicating the activation status for all debug message types of the trace class.

Additional information

This function is optional and is available only when the file system is built with `FS_DEBUG_LEVEL` set to a value equal or greater than `FS_DEBUG_LEVEL_LOG_ALL`.

The return value is specified by or-ing one or more message types described at *Debug message types* on page 977.

10.3.2.6 FS_GetWarnFilter()

Description

Queries activation status of warning debug messages.

Prototype

```
U32 FS_GetWarnFilter(void);
```

Return value

Value indicating the activation status for all debug message types of the warning class.

Additional information

This function is optional and is available only when the file system is built with `FS_DEBUG_LEVEL` set to a value equal or greater than `FS_DEBUG_LEVEL_LOG_WARNINGS`.

The return value is specified by or-ing one or more message types described at *Debug message types* on page 977.

10.3.2.7 FS_SetErrorFilter()

Description

Enables and disables error debug messages.

Prototype

```
void FS_SetErrorFilter(U32 FilterMask);
```

Parameters

Parameter	Description
FilterMask	Specifies the message types.

Additional information

FS_AddErrorFilter() can be used to enable and disable a specified set of debug message types of the error class. The debug message types that have the bit set to 1 in FilterMask are enabled while the other debug message types are disabled.

This function is optional and is available only when the file system is built with FS_DEBUG_LEVEL set to a value equal or greater than FS_DEBUG_LEVEL_LOG_ERRORS.

FileMask is specified by or-ing one or more message types described at *Debug message types* on page 977.

10.3.2.8 FS_SetLogFilter()

Description

Enables and disables trace debug messages.

Prototype

```
void FS_SetLogFilter(U32 FilterMask);
```

Parameters

Parameter	Description
FilterMask	Specifies the message types.

Additional information

FS_SetLogFilter() can be used to enable and disable a specified set of debug message types of the trace class. The debug message types that have the bit set to 1 in FilterMask are enabled while the other debug message types are disabled.

This function is optional and is available only when the file system is built with FS_DEBUG_LEVEL set to a value equal or greater than FS_DEBUG_LEVEL_LOG_ALL.

FileMask is specified by or-ing one or more message types described at *Debug message types* on page 977.

10.3.2.9 FS_SetWarnFilter()

Description

Enables and disables warning debug messages.

Prototype

```
void FS_SetWarnFilter(U32 FilterMask);
```

Parameters

Parameter	Description
FilterMask	Specifies the message types.

Additional information

FS_SetWarnFilter() can be used to enable and disable a specified set of debug message types of the warning class. The debug message types that have the bit set to 1 in FilterMask are enabled while the other debug message types are disabled.

This function is optional and is available only when the file system is built with FS_DEBUG_LEVEL set to a value equal or greater than FS_DEBUG_LEVEL_LOG_WARNINGS.

FileMask is specified by or-ing one or more message types described at *Debug message types* on page 977.

10.3.2.10 Debug message types

Description

Flags that control the output of debug messages.

Definition

```
#define FS_MTYPE_INIT       (1uL << 0)
#define FS_MTYPE_API        (1uL << 1)
#define FS_MTYPE_FS         (1uL << 2)
#define FS_MTYPE_STORAGE    (1uL << 3)
#define FS_MTYPE_JOURNAL    (1uL << 4)
#define FS_MTYPE_CACHE      (1uL << 5)
#define FS_MTYPE_DRIVER     (1uL << 6)
#define FS_MTYPE_OS         (1uL << 7)
#define FS_MTYPE_MEM        (1uL << 8)
```

Symbols

Definition	Description
FS_MTYPE_INIT	Initialization log messages.
FS_MTYPE_API	Log messages from API functions.
FS_MTYPE_FS	Log messages from file system.
FS_MTYPE_STORAGE	Storage layer log messages.
FS_MTYPE_JOURNAL	Journal log messages
FS_MTYPE_CACHE	Cache log messages
FS_MTYPE_DRIVER	Log messages from device and logical drivers.
FS_MTYPE_OS	Log messages from OS integration layer.
FS_MTYPE_MEM	Log messages from internal memory allocator.

10.4 Troubleshooting

This section provides some ideas about how to analyze why the file system is not working as expected.

emFile comes with an API function called FS_FOpen() that is the equivalent of the fopen() function that is part of any standard C library. This function takes as parameters a file name and an access mode. If the specified file access type is allowed and no error occurs, the function returns a pointer to a file handle that can be used to perform operations on that file.

```
#include "FS.h"

int SampleFileOpen(const char * s) {
  FS_FILE * pFile;

  pFile = FS_FOpen("test.txt", "r");
  if (pFile == NULL) {
    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 three main 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).

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.

```
#include "FS.h"

#define BYTES_PER_SECTOR    512

int SampleWriteSector(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 pointer to a file handle although the sectors of the device are 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

Profiling with SystemView

This chapter describes the profiling instrumentation of emFile.

11.1 General information

emFile is instrumented to generate profiling information of API functions and driver-level functions. This profiling information expose the run-time behavior of emFile in an application, recording which API functions have been called, how long the execution took, and revealing which driver-level functions have been called by API functions or evens like interrupts.

The profiling information is recorded using SystemView. SystemView is a real-time recording and visualization tool for profiling data. It exposes the true run-time behavior of a system, going far deeper than the insight provided by debuggers. This is particularly effective when developing and working with complex systems comprising an OS with multiple threads and interrupts, and one or more middle ware components.

SystemView can ensure a system performs as designed, can track down inefficiencies, and show unintended interactions and resource conflicts. The recording of profiling information with SystemView is minimally intrusive to the system and can be done on virtually any system. With SEGGERs Real Time Technology (RTT) and a J-Link SystemView can record data in real-time and analyze the data live, while the system is running.

The emFile profiling instrumentation can be easily configured and set up.

11.2 Configuring profiling

Profiling can be included or excluded at compile-time and enabled at run-time. When profiling is excluded, no additional overhead in performance or memory usage is generated. Even when profiling is enabled the overhead is minimal, due to the efficient implementation of SystemView.

The SystemView module needs to be added to the application to enable profiling. If not already part of the project, download the sources from https://www.segger.com/systemview.html and add them to the project.

Also make sure that the `FS_SYSVIEW.c` file from the `FS` directory of the emFile shipment is included in the project.

11.2.1 Compile time 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 of the file system.

To include profiling, the configuration define `FS_SUPPORT_PROFILE` must be set to 1 in the emFile configuration (`FS_Conf.h`) or via the project preprocessor defines.

The following table lists the configuration defines related to profiling.

Define	Type	Default value	Description
`FS_SUPPORT_PROFILE`	B	0	Enables/disables profiling support.
`FS_SUPPORT_PROFILE_END_CALL`	B	0	Enables/disables the profiling of return values.

Refer to *Configuration of emFile* on page 927 for a description of configuration define types.

11.2.1.1 FS_SUPPORT_PROFILE

This configuration define can be used to enable the support for profiling instrumentation in emFile. By default the support is disabled to save ROM space.

11.2.1.2 FS_SUPPORT_PROFILE_END_CALL

This configuration define specifies if the return values of the profiled functions have to be recoded. This feature is disabled by default to save ROM space. `FS_SUPPORT_PROFILE_END_CALL` has effect only when `FS_SUPPORT_PROFILE` is set to 1.

For detailed information about the configuration of emFile and the types of the configuration defines, refer to *Configuration of emFile* on page 927.

11.2.2 Runtime configuration

To enable profiling at run-time, `FS_SYSVIEW_Init()` needs to be called. Profiling can be enabled at any time but it is recommended to do this in `FS_X_AddDevices()` as demonstrated in the sample below:

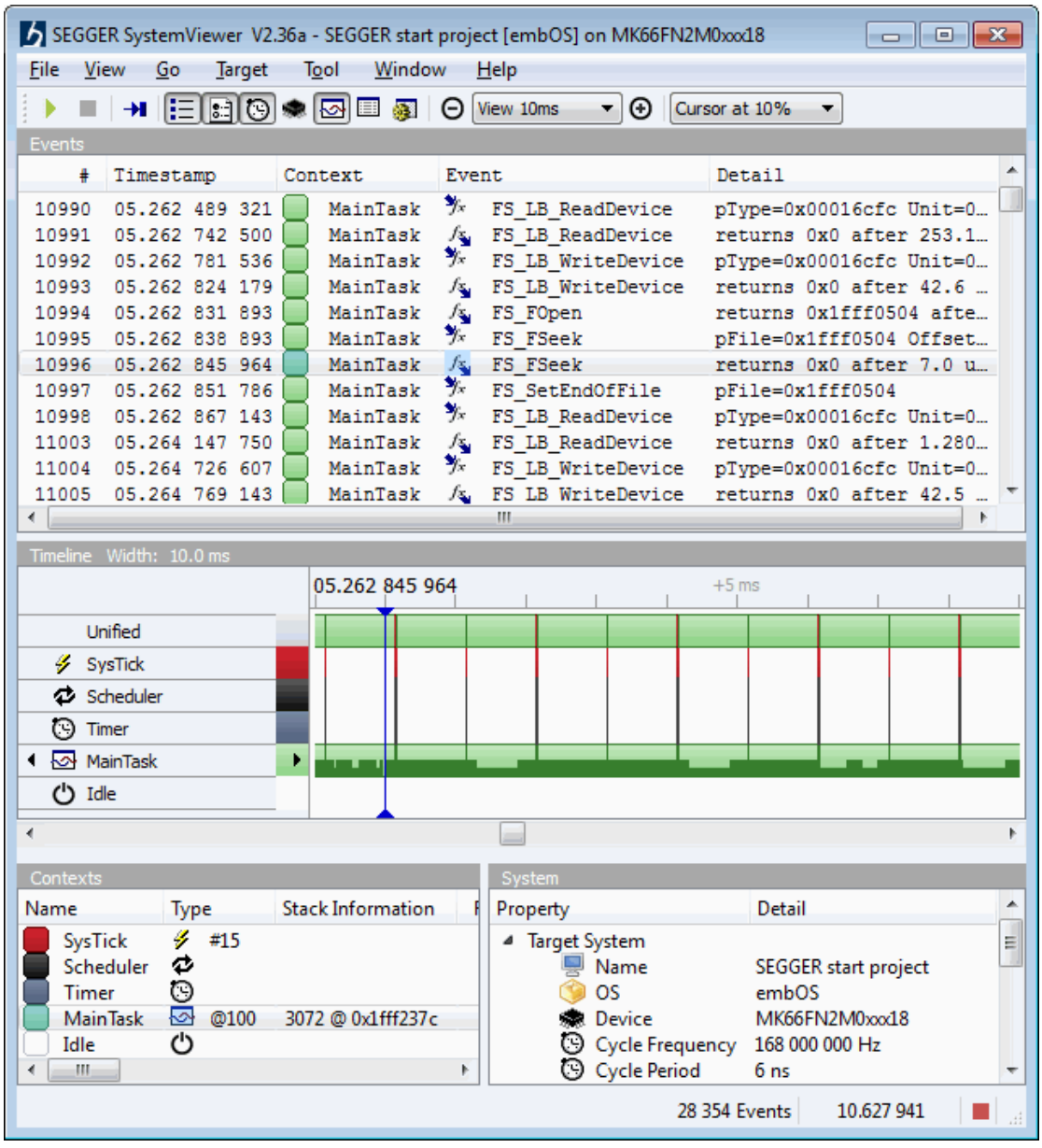
```
#include "FS.h"

/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*/
void FS_X_AddDevices(void) {
#if FS_SUPPORT_PROFILE
  FS_SYSVIEW_Init();
#endif
```



```
  //
  // Add device driver initialization here.
  //
}
```


11.3 Recording and analyzing profiling information

When profiling is included and enabled emFile generates profiling events. On a system which supports RTT (i.e. ARM Cortex-M and Renesas RX) the data can be read and analyzed with SystemView and a J-Link. Connect the J-Link to the target system using the default debug interface and start the SystemView host application. If the system does not support RTT, SystemView can be configured for single-shot or postmortem mode. Please refer to the SystemView User Manual for more information.



Chapter 12

Performance and resource usage

This chapter contains information about the RAM and ROM requirements and how to measure and improve performance of emFile.

12.1 Resource usage

emFile has been designed to meet the requirements of any embedded system. This means the the file system can be configured to include only the features required by a specific system in order to keep the RAM and ROM usage at a minimum while providing efficient access to the storage device.

The actual RAM and ROM memory requirements of emFile depend on the used features. The following section provides information about how to calculate the memory requirements and some values for 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 final executable.

12.1.1 ROM usage

12.1.1.1 Test procedure

The ROM usage has been measured by compiling the sources with SEGGER Embedded Studio IDE V4.20 for Cortex-M in Thumb mode and with the highest size optimization. The file system has been configured at the lowest debug level and without OS support:

```
#define FS_OS_LOCKING                0 // Disable OS support
#define FS_DEBUG_LEVEL               0 // Set debug level
```

No driver has been added to the file system. For information about the memory usage of a specific emFile driver refer to the "Performance and resource usage" section of the respective driver in the *Device drivers* on page 320 section.

The sample application listed below has been used to calculate the memory resources of commonly used functions. You can easily reproduce the measurement results by compile this sample application. 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.

FS_TEST_ROMUsage.c

```
/*********************************************************************
*                  (c) SEGGER Microcontroller GmbH                   *
*                       The Embedded Experts                         *
*                          www.segger.com                            *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File    : FS_TEST_ROMUsage.c
Purpose : Application to calculate the ROM usage of the file system.
*/

/*********************************************************************
*
*       #include section
*
**********************************************************************
*/
#include "FS.h"
#include "FS_Int.h"

#if FS_DEBUG_LEVEL > 0
  #error FS_DEBUG_LEVEL has to be to 0.
#endif
```



```
/*********************************************************************
*
*       Defines, configurable
*
**********************************************************************
*/
#ifndef   STEP
  #define STEP      0    // Change this line to adjust which portions of code are linked
#endif

#ifndef   SUBSTEP
  #define SUBSTEP   0
#endif

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/
#if STEP >= 9
  static U32 _aCache[FS_SIZEOF_CACHE_ANY(2, 512) / 4];
#endif // STEP >= 9
#if (STEP >= 10)
  static U32 _aBuffer[16];
  static U8  _NumErrors;
#endif // STEP >= 10

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

#if (STEP >= 10)

/*********************************************************************
*
*       _OnError
*/
static int _cbOnError(int ErrCode, ...) {
  (void)ErrCode;
  _NumErrors++;
  return FS_CHECKDISK_ACTION_DO_NOT_REPAIR;
}

#endif // STEP >= 10

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       MainTask
*/
void MainTask(void);
void MainTask(void) {
#if STEP >= 1            // Step 1: Minimum file system
  FS_FILE * pFile;

  FS_Init();
#if FS_SUPPORT_JOURNAL
  FS_JOURNAL_Create("", 100uL * 1024uL);
#endif
  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;
    char         acFileName[16];

    FS_FindFirstFile(&fd, "", acFileName, sizeof(acFileName));
    FS_FindClose(&fd);
   }
#endif
#if STEP >= 5          // Step 5: Create a directory
  FS_MkDir("TestDir");
#endif
#if STEP >= 6          // Step 6: Add long file name support
#if (FS_SUPPORT_FAT != 0)
  FS_FAT_SupportLFN();
#endif
#endif
#if STEP >= 7          // Step 7: Low-level format a storage device
  FS_FormatLow("");
#endif
#if STEP >= 8          // Step 8: High-level format a storage device
  FS_Format("", NULL);
#endif
#if STEP >= 9          // Step 9: Assign cache
#if (SUBSTEP == 0)
  FS_AssignCache("", _aCache, sizeof(_aCache), FS_CACHE_ALL);
#endif
#if (SUBSTEP == 1)
  FS_AssignCache("", _aCache, sizeof(_aCache), FS_CACHE_MAN);
#endif
#if (SUBSTEP == 2)
  FS_AssignCache("", _aCache, sizeof(_aCache), FS_CACHE_RW);
#endif
#if (SUBSTEP == 3)
  FS_AssignCache("", _aCache, sizeof(_aCache), FS_CACHE_RW_QUOTA);
#endif
#if (SUBSTEP == 4)
  FS_AssignCache("", _aCache, sizeof(_aCache), FS_CACHE_MULTI_WAY);
#endif

#endif
#if STEP >= 10         // Step 10: Disk checking
  _NumErrors = 0;
  FS_CheckDisk("", _aBuffer, sizeof(_aBuffer), 1, _cbOnError);
#endif
#if STEP >= 11         // Step 11: Get information about storage device
  {
    FS_DISK_INFO DiskInfo;

    FS_GetVolumeInfo("", &DiskInfo);
  }
#endif
#if STEP >= 12         // Step 12: Get the size of a file
  FS_GetFileSize(pFile);
#endif
#if STEP >= 1          // Step 1: Minimum file system
  FS_FClose(pFile);
#endif
}

/*********************************************************************
*
*       FS_X_AddDevices
*/
void FS_X_AddDevices(void) {
  ;
}

/*********************************************************************
*
*       FS_X_GetTimeDate
*
*  Description:
*    Current time and date in a format suitable for the file system.
*
```



```
*    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: Count of years from 1980 (0-127)
*
*/
U32 FS_X_GetTimeDate(void) {
  U32 r;
  U16 Sec, Min, Hour;
  U16 Day, Month, Year;

  Sec   = 0;        // 0 based.  Valid range: 0..59
  Min   = 0;        // 0 based.  Valid range: 0..59
  Hour  = 0;        // 0 based.  Valid range: 0..23
  Day   = 1;        // 1 based.    Means that 1 is 1. Valid range is 1..31
 (depending on month)
  Month = 1;        // 1 based.    Means that January is 1. Valid range is 1..12.
  Year  = 0;        // 1980 based. Means that 2007 would be 27.
  r   = Sec / 2 + (Min << 5) + (Hour  << 11);
  r  |= (U32)(Day + (Month << 5) + (Year  << 9)) << 16;
  return r;
}

/************************* End of file ****************************/
```

12.1.1.2 Test results

The following table shows the ROM usage of different API functions.

Step	Description	ROM [Kbytes]
1	File system core (without driver)	7.0
2	Read file.	1.1
3	Write file.	1.1
4	Remove file.	0.1
5	Open directory.	0.5
6	Create directory.	0.5
7	Long file name support.	2.0
8	Low-level format a medium.	0.2
9	High-level format a medium.	1.8
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
11	Check the storage.	3.3
12	Get device info.	0.1
13	Get the size of a file.	0.1

Summary

Typically, a simple file system uses about 10 Kbytes of ROM. To calculate the overall ROM usage, the ROM usage of the used driver(s) needs to be added.

12.1.2 Static RAM usage

The static RAM usage of the file system without any driver is about **150 bytes**.

12.1.3 Dynamic RAM requirements

During the initialization emFile dynamically allocates memory depending on the number of added devices, the number of simultaneously opened files and the OS locking type as follows:

Data structure	Size (Bytes)	Count
File handle	20	Maximum number of simultaneously open files. Depends on application, minimum is 1.
File object	44 FS_MULTI_HANDLE_SAFE = 1 + FS_MAX_LEN_FULL_FILE_NAME FS_SUPPORT_ENCRYPTION = 1 + 8	Maximum number of simultaneously open files. Depends on application, minimum is 1.
Volume instance	108	Number of FS_AddDevice() calls.
FS_OS_LOCKING = 0 (no OS is used)		
Sector buffer	8 + SectorSize By default, SectorSize is 512 bytes.	2 FS_SUPPORT_ENCRYPTION = 1 +1 FS_SUPPORT_JOURNAL = 1 +1
FS_OS_LOCKING = 1 (OS locking at API level)		
Sector buffer	8 + SectorSize By default, SectorSize is 512 bytes.	2 FS_SUPPORT_ENCRYPTION = 1 +1 FS_SUPPORT_JOURNAL = 1 +1
OS synchronization object	Depends on the OS used.	1
FS_OS_LOCKING = 2 (OS locking at driver level)		
Sector buffer	8 + SectorSize By default, SectorSize is 512 bytes.	2 FS_SUPPORT_ENCRYPTION = 1 +1 FS_SUPPORT_JOURNAL = 1 +1 * Number of used drivers
OS locking instance	16	Number of used drivers.
OS synchronization object	Depends on the OS used.	1 + Number of used drivers.

Note

File handles and objects can also be allocated after the file system initialization depending on how many files the application opens at the same time.

RAM usage 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**.

12.2 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.

12.2.1 Test procedure

The performance tests are executed using the `FS_PerformanceSimple.c` application that can be found in the `{Sample/FS/Application}` folder of the emFile shipment.

The test application performs the following steps:

1. Format the drive.
2. Create and open a file for writing.
3. Start measuring of write performance.
4. Write a multiple of 8 Kbytes.
5. Stop measuring of write performance.
6. Close the file.
7. Reopen the file for reading.
8. Start measuring of read performance.
9. Read a multiple of 8 Kbytes.
10. Stop measuring of read performance.
11. Close the file.
12. Show the performance results.

FS_PerformanceSimple.c

```
/*********************************************************************
*                   (c) SEGGER Microcontroller GmbH                  *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File    : FS_PerformanceSimple.c
Purpose : Sample program that can be used to measure the performance
          of the file system.

Additional information:
  Preparations:
    Works out-of-the-box with any storage device.

  Expected behavior:
    Measures the speed at witch the file system can write and read blocks
    of data to and from a file. The size of the file as well as the number
    of bytes that have to be written at once are configurable.

    The application always formats the storage device to make sure that
    the measurements are not influenced by the data already stored on the
    file system.

  Sample output:
    Start
    High-level format
    Writing 16 chunks of 524288 bytes...................OK
    Reading 16 chunks of 524288 bytes...................OK

    W Speed: 2115 Kbyte/s
    R Speed: 11130 Kbyte/s
    Finished
*/

/*********************************************************************
*
*       #include Section
*
**********************************************************************
```



```
*/
#include <string.h>
#include "FS.h"
#include "FS_OS.h"
#include "SEGGER.h"

/*********************************************************************
*
*       Defines, configurable
*
**********************************************************************
*/
#ifndef   FILE_SIZE
  #define FILE_SIZE             (8192L * 1024L)
  // Size of the file in bytes to be used for testing.
#endif

#ifndef   BLOCK_SIZE
  #define BLOCK_SIZE            (8 * 1024L)
  // Block size for individual read / write operation in bytes.
#endif

#ifndef   NUM_BLOCKS_MEASURE
  #define NUM_BLOCKS_MEASURE    (64)
  // Number of blocks for individual measurement
#endif

#ifndef   VOLUME_NAME
  #define VOLUME_NAME           ""
  // Name of the volume to be used for testing.
#endif

#ifndef   FILE_NAME
  #define FILE_NAME             "SEGGER.txt"
  // Name of the file to be used for testing.
#endif

/*********************************************************************
*
*       Types
*
**********************************************************************
*/

typedef struct {
  const char * sName;
  I32          Min;
  I32          Max;
  I32          Av;
  I32          Sum;
  I32          NumSamples;
  U32          NumBytes;
} RESULT;

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/
static U32    _aBuffer[BLOCK_SIZE / 4];
static RESULT _aResult[2];
static int    _TestNo;
static char   _ac[512];

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       _WriteFile
*
```



```
*  Function description
*    Measures the write time.
*/
static I32 _WriteFile(FS_FILE * pFile, const void * pData, U32 NumBytes, U32 NumBlocksMeasure) {
  I32 t;
  U32 i;

  t = (I32)FS_X_OS_GetTime();
  for (i = 0; i < NumBlocksMeasure; i++) {
    (void)FS_Write(pFile, pData, NumBytes);
  }
  return (I32)FS_X_OS_GetTime() - t;
}


/*********************************************************************
*
*       _ReadFile
*
*  Function description
*    Measures the read performance.
*/
static I32 _ReadFile(FS_FILE * pFile, void * pData, U32 NumBytes, U32 NumBlocksMeasure) {
  I32 t;
  U32 i;

  t = (I32)FS_X_OS_GetTime();
  for (i = 0; i < NumBlocksMeasure; i++) {
    (void)FS_Read(pFile, pData, NumBytes);
  }
  return (I32)FS_X_OS_GetTime() - t;
}
/*********************************************************************
*
*       _StartTest
*/
static void _StartTest(const char * sName, U32 NumBytes) {
  RESULT * pResult;

  if ((_TestNo + 1) < (int)SEGGER_COUNTOF(_aResult)) {
    pResult = &_aResult[++_TestNo];
    pResult->sName      = sName;
    pResult->Min        =  0x7fffffff;
    pResult->Max        = -0x7fffffff;
    pResult->NumSamples = 0;
    pResult->Sum        = 0;
    pResult->NumBytes   = NumBytes;
  }
}

/*********************************************************************
*
*       _StoreResult
*/
static void _StoreResult(I32 t) {
  RESULT * pResult;

  pResult = &_aResult[_TestNo];
  if (t > pResult->Max) {
    pResult->Max = t;
  }
  if (t < pResult->Min) {
    pResult->Min = t;
  }
  pResult->NumSamples++;
  pResult->Sum += (I32)t;
  pResult->Av   = pResult->Sum / pResult->NumSamples;
}

/*********************************************************************
*
*       _GetAverage
*/
static double _GetAverage(int Index) {
  RESULT   * pResult;
  double     v;
```



```
  unsigned   NumKBytes;

  pResult = &_aResult[Index];
  v = (double)pResult->Av;
  if (v == 0.0) {
    return (float)0.0;
  }
  v = (double)1000.0 / v;
  NumKBytes = pResult->NumBytes >> 10;
  v = v * (float)NumKBytes;
  return v;
}

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       MainTask
*/
#ifdef __cplusplus
extern "C" {     /* Make sure we have C-declarations in C++ programs */
#endif
void MainTask(void);
#ifdef __cplusplus
}
#endif
void MainTask(void) {
  int       i;
  U32       Space;
  unsigned  NumLoops;
  U32       NumBytes;
  U32       NumBytesAtOnce;
  FS_FILE * pFile;
  I32       t;
  int       r;
  U32       NumBlocksMeasure;
  char      acFileName[128];

  FS_X_Log("Start\n");
  FS_Init();
  _TestNo = -1;
  //
  // Check if we need to low-level format the volume
  //
  if (FS_IsLLFormatted(VOLUME_NAME) == 0) {
    FS_X_Log("Low-level format\n");
    (void)FS_FormatLow(VOLUME_NAME);
  }
  //
  // Volume is always high level formatted
  // before doing any performance tests.
  //
  FS_X_Log("High-level format\n");
#if FS_SUPPORT_FAT
  r = FS_FormatSD(VOLUME_NAME);
#else
  r = FS_Format(VOLUME_NAME, NULL);
#endif
  if (r == 0) {
    //
    // Configure the file system so that the
    // directory entry and the allocation table
    // are updated when the file is closed.
    //
    FS_SetFileWriteMode(FS_WRITEMODE_FAST);
    //
    // Fill the buffer with data.
    //
    memset(_aBuffer, (int)'a', sizeof(_aBuffer));
    //
    // Get some general info.
```



```
    //
    Space            = FS_GetVolumeFreeSpace(VOLUME_NAME);
    Space            = SEGGER_MIN(Space, (U32)FILE_SIZE);
    NumBytesAtOnce   = BLOCK_SIZE;
    NumBlocksMeasure = NUM_BLOCKS_MEASURE;
    for (;;) {
      NumBytes = NumBytesAtOnce * NumBlocksMeasure;
      if (NumBytes <= Space) {
        break;
      }
      NumBytesAtOnce   >>= 1;
      NumBlocksMeasure >>= 1;
    }
    NumLoops = Space / NumBytes;
    if (NumLoops != 0u) {
      //
      // Create file of full size.
      //
      _StartTest("W", NumBytes);
      SEGGER_snprintf(acFileName, (int)sizeof(acFileName), "%s%c
%s", VOLUME_NAME, FS_DIRECTORY_DELIMITER, FILE_NAME);
      pFile = FS_FOpen(acFileName, "w");
      //
      // Preallocate the file, setting the file pointer to the highest position
      // and declare it as the end of the file.
      //
      (void)FS_FSeek(pFile, (I32)Space, FS_SEEK_SET);
      (void)FS_SetEndOfFile(pFile);
      //
      // Set file position to the beginning.
      //
      (void)FS_FSeek(pFile, 0, FS_SEEK_SET);
      //
      // Check write performance with clusters/file size preallocated.
      //
      SEGGER_snprintf(_ac, (int)sizeof(_ac), "Writing %d chunks of %lu
 Kbytes...", NumLoops, NumBytes >> 10);
      FS_X_Log(_ac);
      for (i = 0; i < (int)NumLoops ; i++) {
        t = _WriteFile(pFile, _aBuffer, NumBytesAtOnce, NumBlocksMeasure);
        _StoreResult(t);
        FS_X_Log(".");
      }
      FS_X_Log("OK\n");
      (void)FS_FClose(pFile);
      //
      // Check read performance.
      //
      _StartTest("R", NumBytes);
      SEGGER_snprintf(_ac, (int)sizeof(_ac), "Reading %d chunks of %lu
 Kbytes..." , NumLoops, NumBytes >> 10);
      FS_X_Log(_ac);
      pFile = FS_FOpen(acFileName, "r");
      for (i = 0; i < (int)NumLoops; i++) {
        t = _ReadFile(pFile, _aBuffer, NumBytesAtOnce, NumBlocksMeasure);
        _StoreResult(t);
        FS_X_Log(".");
      }
      FS_X_Log("OK\n\n");
      (void)FS_FClose(pFile);
      (void)FS_Remove(acFileName);
      //
      // Show results for performance list.
      //
      for (i = 0; i <= _TestNo; i++) {
        SEGGER_snprintf(_ac, (int)sizeof(_ac), "%s Speed: %d Kbyte/s
\n", _aResult[i].sName, (int)_GetAverage(i));
        FS_X_Log(_ac);
      }
    } else {
      FS_X_Log("ERROR: Not enough free space available on the storage.\n");
    }
    FS_Unmount(VOLUME_NAME);
  } else {
    FS_X_Log("ERROR: Volume could not be formatted!\n");
  }
```



```
  FS_X_Log("Finished\n");
  for (;;) {
    ;
  }
}

/************************** End of file ***************************/
```


12.2.2 How to improve 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 a different 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()` to learn how the write mode can be configured. Please note that these write modes can not be used when journaling is enabled.

Pre-allocate the files

The pre-allocation of a file helps reduce the number of accesses to the storage device and therefore improves the read and write performance. When a file is pre-allocated the file system reserves storage space for that file in the allocation table before the file is actually filled with data. As such, a write access to the allocation table is no longer necessary when the data is actually stored to the file. A file can be pre-allocated by calling either `FS_SetEndOfFile()` or `FS_SetFileSize()` in the application after the file is opened.

Write multiples of a logical sector size

The file system implements a 0-copy mechanism in which 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 content of a file always begins at a sector boundary.

Use a 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()` 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.

Use a sector cache

The sector cache can be enabled to increase the overall performance of the file system. For more information refer to *Caching and buffering* on page 299.

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 runtime when the allocation table is accessed. This reduces, for example, the time it takes to determine the amount of free space. For more information refer to *Sector Read-Ahead driver* on page 890.

Optimize the hardware layer

Ensure that the routines of the hardware layer are fast. It depends on your compiler how to do that. 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`. By default it expands to nothing. You can use this macro to enable the compiler optimization only for these functions. It depends on your compiler how to define this macro.

Chapter 13

Journaling

This chapter documents and explains emFile’s Journaling component. The Journaling component is an extension to file system that makes the file operations fail-safe.

13.1 General information

emFile Journaling is an additional component that is located between the file system and the storage layers that makes the operations of the file system layer fail-safe. File systems without journaling support (for example FAT and EFS) are by design not fail-safe. The Journaling component works by storing temporarily all the data changes performed by the file system layer to a journal file before copying them at once to the actual destination on the storage device.

13.1.1 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 component makes only sure that the file system layer fail-safe. For fail-safety of the driver layer, refer to *Device drivers* on page 320.

13.1.2 Features

- Non fail-safe file systems will be fail-safe.
- Fully compatible to standard file system implementations (e.g. FAT).
- Every type of storage device is supported. No reformat required.
- Multiple write accesses to the storage medium can be combined in user application.

13.2 Theory of operation

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 therefore the only place where a typical data loss can occur is the file system layer. The file system can be corrupted when a write operation is interrupted for example in the event of power failure or system crash. This derives from the design of FAT file system and is true for all implementations from any vendor. The emFile Journaling component adds fail-safety to the file system layer.

The goal of this additional component 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 to `FS_FWrite()` that writes 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 component 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 file. The data stored in the journal file is copied to the actual destination on the storage device 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 does not lead to data loss. The interrupted copy process is restarted after a restart of the target hardware.

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 the journal	Consistent	Lost
4.	After finalization	Consistent	Preserved
5.	While copying from journal into file system	Consistent	Preserved
6.	After copy process, before invalidating of the journal	Consistent	Preserved
7.	While invalidating of the journal	Consistent	Preserved

13.3 Write optimization

The Journaling component has been optimized for write performance. When data is written at the end of a file, which is the case in the most applications, only the management information (allocation table and directory entry) is actually stored to journal file. The file contents is written directly to the actual destination on the storage device that helps improving the write performance. The fail-safety of the file system layer is not affected as the file contents overwrite storage blocks that are not allocated to any file or directory. The optimization is disabled as soon as a file or directory is deleted during a journal transaction. This is done in order to make sure that the data of the deleted file or directory is preserved in case of an unexpected reset.

13.4 How to use the Journaling component

Using Journaling component is very simple from the application's perspective. Only two changes have to be made to an existing application in order to enable the Journaling component:

1. The Journaling component has to be enabled in the file system configuration via the FS_SUPPORT_JOURNAL configuration define. Optionally, the name of the journal file can be configured via the FS_JOURNAL_FILE_NAME configuration define. For more information about additional compile time options refer to the section *Configuration* on page 965.
2. The application has to call either FS_JOURNAL_Create() or FS_JOURNAL_CreateEx() after formatting the volume to create the file required for the journal operation. Refer to FS_JOURNAL_Create() and FS_JOURNAL_CreateEx() for detailed information.

Everything else is done by the Journaling component.

13.4.1 Combining multiple write operations

The Journaling component can be used in the application to create a journal transaction that consists of multiple write operations. This can be useful when for example the changes made to two different files have to be performed in an atomic way. That is after an unexpected reset that interrupts the write operation the files have to contain either the old or the new data but never a combination of both. A journal transaction can be opened via FS_JOURNAL_Begin() and has to be closed using FS_JOURNAL_End() as shown in the sample below.

```
#include "FS.h"

void SampleJournalMultipleWrite(void) {
  FS_FILE * pFile;

  //
  // Begin an operations which have to be be fail-safe.
  // All following steps will be stored into journal.
  //
#if FS_SUPPORT_JOURNAL
  FS_JOURNAL_Begin("");
#endif // FS_SUPPORT_JOURNAL
  pFile = FS_FOpen("File1.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Test 1", 6);
    FS_FClose(pFile);
  }
  pFile = FS_FOpen("File2.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Test 2", 6);
    FS_FClose(pFile);
  }
  //
  // End an operation which has to be be fail-safe.
  // Data is copied from journal into file system.
  //
#if FS_SUPPORT_JOURNAL
  FS_JOURNAL_End("");
#endif // FS_SUPPORT_JOURNAL
}
```

13.4.2 Preserving the consistency of a file

The Journaling component can be used in the application to protect the contents of the file against unexpected resets. This means that after an unexpected reset that interrupted the file write operation, the file will contain either the old data or the new data but not a combination of both. The same applies to the particular case were the application writes to an empty file. The first method uses FS_JOURNAL_Begin() and FS_JOURNAL_End() to create a journal transaction as shown in the next example code. The advantage of this method is that a relatively small writing buffer can be used.


```
#include "FS.h"

void SampleJournalFileConsistency1(void) {
  U8        aBuffer[128];
  FS_FILE * pFile;

  pFile = FS_FOpen("File.txt", "w");
  if (pFile) {
#if FS_SUPPORT_JOURNAL
    FS_JOURNAL_Begin("");
#endif // FS_SUPPORT_JOURNAL
    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 file is sufficiently large to store
    // all the changes performed by the file system.
    //
#if FS_SUPPORT_JOURNAL
    FS_JOURNAL_End("");
#endif // FS_SUPPORT_JOURNAL
    FS_FClose(pFile);
  }
}
```

The second method uses a buffer sufficiently large to store the contents of the entire file that is then passed to `FS_FWrite()` or `FS_Write()` in single function call. This method is demonstrated in the following example code.

```
#include "FS.h"

void SampleJournalFileConsistency2(void) {
  U8        aBuffer[256];
  FS_FILE * pFile;

  pFile = FS_FOpen("File.txt", "w");
  if (pFile) {
    memset(aBuffer, 'a', 128);
    memset(&aBuffer[128], 'b', 128);
    //
    // The changes are committed in a single journal transaction
    // assuming that the journal file is sufficiently large to
    // store all the changes.
    //
    FS_Write(pFile, aBuffer, sizeof(aBuffer));
    FS_FClose(pFile);
  }
}
```

Both methods require that the journal file is sufficiently large to store all the changes made to the storage device by the file system during the write operation. When writing to a file that is not empty the management data and the file contents are stored the journal file. If the file the application is writing to is initially empty then only the management data has to be stored to journal file. In this case the contents of the file is written directly to the actual destination on the storage device which means that a smaller journal file can be used.

13.4.3 Journaling and write caching

In order to guarantee the correct operation of the Journaling component most of the write caching performed by the file system has to be disabled. That is the file system has to be configured to write the data immediately to storage device instead of keeping the data in its internally RAM buffers. If not configured in this way, the journaling is no longer able to ensure the fail safety of the file system on an unexpected reset because the data stored in the internal RAM buffers will be lost.

The correct operation of the Journaling component is guaranteed when:

- **The write sector cache is disabled**. emFile comes with support for caching at sector level. The sector cache is disabled by default and it has to be explicitly enabled in the application via `FS_AssignCache()`. Five types of sector caches are provided with three of them also supporting a write mode. The `FS_CACHE_ALL` and `FS_CACHE_MAN` cache types are pure read caches and can always be used together with the Journaling component. The other three types of caches `FS_CACHE_RW`, `FS_CACHE_RW_QUOTA`, and `FS_CACHE_MULTI_WAY` can work in read as well as in write mode. The application has to make sure that the write mode is disabled for this type of sector caches. For detailed information about the configuration of sector caches, refer to *Caching and buffering* on page 299.
- **The delayed update of directory entries is disabled**. The file system can be configured to update the directory entries only when the file is closed. This is done in order to increase the write performance. This feature is enabled via `FS_ConfigOnWriteDirUpdate()` with the OnOff parameter set to 0. The application has to make sure that this API function is not called at all or that it is called with the the OnOff parameter set to 1.
- **The file system write mode is set to `FS_WRITEMODE_SAFE`**. emFile can be configured to use different methods when it writes data to a file. The application has to make sure that the write mode is set to `FS_WRITEMODE_SAFE`. This write mode guarantees that the data is written immediately to storage device. By default the file system uses the `FS_WRITEMODE_SAFE` write mode. The write mode can be changed at runtime via `FS_SetFileWriteMode()`. If this function is called in the application than the WriteMode parameter has to be set to `FS_WRITEMODE_SAFE`. Alternatively, the call to `FS_SetFileWriteMode()` can be removed.

> **Note**
>
> Beginning with the version 4.04g of emFile and application is allowed to use a file buffer in write mode together with the Journaling component.

13.5 Configuration

The Journaling component has to be configured at compile as well as runtime to enable its operation.

13.5.1 Compile time configuration

The compile time configuration is realized via preprocessor defines that have to be added to the FS_Conf.h file which is the main configuration file of emFile. For detailed information about the configuration of emFile and of the configuration define types, refer to *Configuration of emFile* on page 927 The following table lists the configuration defines supported by the Journaling component.

Define	Type	Default value	Description
FS_SUPPORT_JOURNAL	B	0	Enables or disables the journaling support.
FS_JOURNAL_FILE_NAME	S	"Journal.dat"	Configures the default name of the journal file.
FS_MAX_LEN_JOURNAL_FILE_NAME	N	0	Configures the space reserved for the journal file name.
FS_JOURNAL_ENABLE_STATS	B	0	Enables or disables the support for statistical counters.
FS_JOURNAL_SUPPORT_FREE_SECTOR	B	1	Enables or disables the support for informing the device driver about unused logical sectors.
FS_JOURNAL_SUPPORT_FAST_SECTOR_SEARCH	B	1	Specifies which method to use when searching for a logical sector in the journal file.

13.5.1.1 FS_SUPPORT_JOURNAL

This define has to be set to 1 in order to enable the support for Journaling component in the file system. The journal must be initialized in the application at runtime by calling either FS_JOURNAL_Create() or FS_JOURNAL_CreateEx().

13.5.1.2 FS_JOURNAL_FILE_NAME

This define specifies the default name of the journal file. The name of the journal file can be changed at runtime by calling FS_JOURNAL_SetFileName().

13.5.1.3 FS_MAX_LEN_JOURNAL_FILE_NAME

FS_MAX_LEN_JOURNAL_FILE_NAME is the maximum number of characters (including the 0-terminator) in the name of the journal file. A value of 0 disables the feature which is the default. The RAM usage of a volume instance increases the number of bytes specified via this configuration define. The actual file name can be set at runtime using FS_JOURNAL_SetFileName()

13.5.1.4 FS_JOURNAL_ENABLE_STATS

The support for statistical counters can be enabled or disabled by using this define. The statistical counters provide information about the operation of the Journaling component. The values of the statistical counters can be queried via FS_JOURNAL_GetStatCounters() and cleared via FS_JOURNAL_ResetStatCounters()

13.5.1.5 FS_JOURNAL_SUPPORT_FREE_SECTOR

With this feature enabled, the Journaling component forwards information about unused logical sectors to the device driver. A device driver such as NAND or NOR driver can use this information to optimize the handling of the data. The operation has to be enabled at runtime via the SupportFreeSector of `FS_JOURNAL_CreateEx()`

13.5.1.6 FS_JOURNAL_SUPPORT_FAST_SECTOR_SEARCH

If this define set to 1 then each entry in the logical to physical mapping table of the Journaling component is 32-bit large which improves the read and write performance. At the same time setting `FS_JOURNAL_SUPPORT_FAST_SECTOR_SEARCH` to 1 it increases the RAM usage of the Journaling component.

13.5.2 Runtime configuration

The Journaling component requires a file to temporarily store the changes performed by the file system. This file has to be created at runtime after the storage device is formatted by calling either `FS_JOURNAL_Create()` or `FS_JOURNAL_CreateEx()`.

13.6 API functions

The table below lists the available API functions.

Function	Description
FS_JOURNAL_Begin()	Opens a journal transaction.
FS_JOURNAL_Create()	Creates the journal file.
FS_JOURNAL_CreateEx()	Creates the journal file.
FS_JOURNAL_Disable()	Deactivates the journal.
FS_JOURNAL_Enable()	Activates the journal.
FS_JOURNAL_End()	Closes a journal transaction.
FS_JOURNAL_GetInfo()	Returns information about the journal.
FS_JOURNAL_GetOpenCnt()	Returns the number times the current journal transaction has been opened.
FS_JOURNAL_GetStatCounters()	Returns statistical information about the operation.
FS_JOURNAL_Invalidate()	Cancels the pending journal transaction.
FS_JOURNAL_IsEnabled()	Checks the journal operational status.
FS_JOURNAL_IsPresent()	Checks the presence of journal file.
FS_JOURNAL_ResetStatCounters()	Sets to 0 all statistical counters.
FS_JOURNAL_SetFileName()	Configures the name of the journal file.
FS_JOURNAL_SetOnOverflowExCallback()	Registers a callback function for the journal full event.
FS_JOURNAL_SetOnOverflowCallback()	Registers a callback function for the journal full event.

13.6.1 FS_JOURNAL_Begin()

Description

Opens a journal transaction.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Journal instance identified by volume name (0-terminated string).

Return value

= 0 OK, journal transaction opened.
≠ 0 Error code indicating the failure reason.

Additional information

This function is optional. The file system opens and closes journal transactions automatically as required. The application can use this function together with FS_JOURNAL_End() to create journal transactions that extend over multiple file system operations. A journal transaction can be opened more than once using FS_JOURNAL_Begin() and it has to be closed by calling FS_JOURNAL_End() by the same number of times.

Following the call to FS_JOURNAL_Begin() all the data written by the application is stored to the journal file until either the application calls FS_JOURNAL_End() or the journal becomes full. An application can get informed about a journal full event by registering a callback function via FS_JOURNAL_SetOnOverflowCallback() or FS_JOURNAL_SetOnOverflowExCallback()

It is mandatory that FS_JOURNAL_Begin() and FS_JOURNAL_End() are called in pairs. The calls to these functions can be nested. The current nesting level can be queried via FS_JOURNAL_GetOpenCnt().

Example

```
#include "FS.h"

void SampleJournalBegin(void) {
  FS_FILE * pFile;

#if FS_SUPPORT_JOURNAL
  FS_JOURNAL_Begin("");               // Open the journal transaction.
#endif // FS_SUPPORT_JOURNAL
  pFile = FS_FOpen("File1.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Test 1", 6);
    FS_FClose(pFile);
  }
  pFile = FS_FOpen("File2.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Test 2...", 6);
    FS_FClose(pFile);
  }
  pFile = FS_FOpen("File3.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Test 3...", 6);
    FS_FClose(pFile);
  }
#if FS_SUPPORT_JOURNAL
  FS_JOURNAL_End("");                 // Close the journal transaction.
#endif // FS_SUPPORT_JOURNAL
```



```
}
```


13.6.2 FS_JOURNAL_Create()

Description

Creates the journal file.

Prototype

```
int FS_JOURNAL_Create(const char * sVolumeName,
                      U32    NumBytes);
```

Parameters

Parameter	Description
sVolumeName	Journal instance identified by volume name (0-terminated string).
NumBytes	Size of journal file in bytes.

Return value

= 1 OK, journal already exists.
= 0 OK, journal successfully created.
< 0 Error code indicating the failure reason.

Additional information

This function is mandatory. It has to be called after the file system initialization to create the journal file. FS_JOURNAL_Create() does nothing if the journal file already exists. The name of the journal file can be configured at compile time via FS_JOURNAL_FILE_NAME or at runtime via FS_JOURNAL_SetFileName().

The size of the journal file can be calculate by using this formula:

JournalSize = 3 * BytesPerSector + (16 + BytesPerSector) * NumSectors

Parameter	Description
JournalSize	Size of the journal file in bytes. This value has to be passed as second parameter to FS_JOURNAL_Create() or FS_JOURNAL_CreateEx().
BytesPerSector	Size of the file system logical sector in bytes.
NumSectors	Number of logical sectors the journal has to be able to store.

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 calculate the approximate number of sectors that are stored during a specific file system operation.

API function	Number of logical sectors
FS_CreateDir()	The number of sectors modified by FS_MkDir() times the number of directories that have to be created.
FS_DeleteDir()	The number of sectors modified by FS_RmDir() times the number of directories that have to be deleted plus the number of sectors modified by FS_Remove() times the number of files that have to be deleted.
FS_FClose()	One sector if the file has been modified else no sectors.
FS_FOpen()	One sector when creating the file else no sectors. If the file exists and is truncated to 0 then total number of sectors in the allocation table that have to be modified.
FS_FWrite()	The same number of sectors as FS_Write()
FS_MkDir()	Two sectors plus the number of sectors in cluster.

API function	Number of logical sectors
FS_ModifyFileAttributes()	One sector.
FS_Move()	Two sectors if the destination and source files or directories are located on the same volume else the number of sectors modified by FS_CopyFile().
FS_Remove()	One sector plus total number of sectors in the allocation table that have to be modified.
FS_Rename()	One sector.
FS_RmDir()	Two sectors.
FS_SetEndOfFile()	One sector plus the total number of sectors in the allocation table that have to be modified.
FS_SetFileAttributes()	One sector.
FS_SetFileSize()	The same number of sectors as FS_SetEndOfFile()
FS_SetFileTime()	One sector.
FS_SetFileTimeEx()	One sector.
FS_SetVolumeLabel()	One sector.
FS_SyncFile()	One sector if the file has been modified else no sectors.
FS_SetVolumeLabel()	One sector.
FS_Write()	Uses the remaining free space in the journal file at the start of the transaction. Two sectors and about 9 percent of the free space available in the journal file (rounded up to a multiple of sector size) are reserved for allocation table and directory entry updates. The remaining sectors are used to store the actual data. If more data is written than free space is available in the journal file, the operation is split into multiple journal transactions.

The values in the table above are for orientation only. The recommended procedure for determining the size of the journal file is as follows:

Step	Action
1	Set the journal file to an arbitrary value (for example 200 Kbytes)
2	Let the application perform typical file system operations.
3	Verify if any journal overflow events occurred. If yes, then increase the journal file by a multiple of the logical sector size of the volume on which the journal file is stored and go to step 2.
4	Done

An overflow event is reported by the Journaling component by invoking the callback function registered via either FS_JOURNAL_SetOnOverflowExCallback() or FS_JOURNAL_SetOnOverflowCallback(). In addition, the size of the journal file can be fine tuned by evaluating the value of the MaxWriteSectorCnt member of the FS_JOURNAL_STAT_COUNTERS returned via FS_JOURNAL_GetStatCounters().

If a journal is created using FS_JOURNAL_Create() the information about unused logical sectors is not forwarded to the device driver. FS_JOURNAL_CreateEx() can be used instead to specify how this information has to be handled.

Example

```
#include "FS.h"
```



```
void SampleJournalCreate(void) {
#if FS_SUPPORT_JOURNAL
  //
  // Create journal of 200 Kbytes on the first volume of the file system.
  //
  FS_JOURNAL_Create("", 200 * 1024);
#endif // FS_SUPPORT_JOURNAL
}
```


13.6.3 FS_JOURNAL_CreateEx()

Description

Creates the journal file.

Prototype

```
int FS_JOURNAL_CreateEx(const char * sVolumeName,
                        U32    NumBytes,
                        U8     SupportFreeSector);
```

Parameters

Parameter	Description
sVolumeName	Journal instance identified by volume name (0-terminated string).
NumBytes	Size of journal file in bytes.
SupportFreeSector	Handling of the information about unused sectors. • 1 Forwarded to the device driver • 0 Not forwarded to device driver.

Return value

= 1 OK, journal already exists.
= 0 OK, journal successfully created.
< 0 Error code indicating the failure reason.

Additional information

This function is mandatory. It performs the same operations as FS_JOURNAL_Create(). In addition, SupportFreeSector can be used to specify if the information about the logical sectors that are no longer in use has to be passed to the device driver. The NOR and NAND drivers as well as the SD/MMC driver with eMMC as storage device can use this information to improve the write performance.

Example

```
#include "FS.h"

void SampleJournalCreateEx(void) {
#if FS_SUPPORT_JOURNAL
  //
  // Create a journal file of 100 Kbytes on the first volume of the file system.
  // The journal component is configured to inform the device driver about
  // logical sectors that are no longer used by the file system.
  //
  FS_JOURNAL_CreateEx("", 100 * 1024, 1);
#endif // FS_SUPPORT_JOURNAL
}
```


13.6.4 FS_JOURNAL_Disable()

Description

Deactivates the journal.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Journal instance identified by volume name (0-terminated string).

Return value

= 0 OK, the journal operation is disabled.
≠ 0 Error code indicating the failure reason.

Additional information

This function is optional. FS_JOURNAL_Disable() can be used to disable the journal if the next file system operations do not have to be protected against unexpected resets. After the call to this function the integrity of the file system is no more guaranteed. The journal operation can be re-enabled by calling FS_JOURNAL_Enable().

The operational status of the journal can be queried using FS_JOURNAL_IsEnabled().

Example

```
#include "FS.h"

void SampleJournalDisableEnable(void) {
  FS_FILE * pFile;

#if FS_SUPPORT_JOURNAL
  FS_JOURNAL_Disable("");
#endif // FS_SUPPORT_JOURNAL
  //
  // The following file system operations are not fail safe.
  //
  pFile = FS_FOpen("File1.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Test 1", 6);
    FS_FClose(pFile);
  }
#if FS_SUPPORT_JOURNAL
  FS_JOURNAL_Enable("");
#endif // FS_SUPPORT_JOURNAL
  //
  // The following file system operations are fail safe.
  //
  pFile = FS_FOpen("File2.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Test 2...", 6);
    FS_FClose(pFile);
  }
}
```


13.6.5 FS_JOURNAL_Enable()

Description

Activates the journal.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Journal instance identified by volume name (0-terminated string).

Return value

= 0 OK, the journal operation is enabled.
≠ 0 Error code indicating the failure reason.

Additional information

This function is optional. The journal is enabled automatically when the file system is mounted if a valid journal file is found. FS_JOURNAL_Enable() can be used to re-enable the journal after the application disabled it via FS_JOURNAL_Disable().

After the call to FS_JOURNAL_Enable() all file system operations are protected against unexpected resets.

The operational status of the journal can be queried using FS_JOURNAL_IsEnabled().

Example

```
#include "FS.h"

void SampleJournalDisableEnable(void) {
  FS_FILE * pFile;

#if FS_SUPPORT_JOURNAL
  FS_JOURNAL_Disable("");
#endif // FS_SUPPORT_JOURNAL
  //
  // The following file system operations are not fail safe.
  //
  pFile = FS_FOpen("File1.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Test 1", 6);
    FS_FClose(pFile);
  }
#if FS_SUPPORT_JOURNAL
  FS_JOURNAL_Enable("");
#endif // FS_SUPPORT_JOURNAL
  //
  // The following file system operations are fail safe.
  //
  pFile = FS_FOpen("File2.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Test 2...", 6);
    FS_FClose(pFile);
  }
}
```


13.6.6 FS_JOURNAL_End()

Description

Closes a journal transaction.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Journal instance identified by volume name (0-terminated string).

Return value

= 0 OK, journal transaction closed.
≠ 0 Error code indicating the failure reason.

Additional information

This function is optional. The file system opens and closes journal transactions automatically as required. The application can use this function together with FS_JOURNAL_Begin() to create journal transactions that extend over multiple file system operations.

Following the outermost call to FS_JOURNAL_End() the sector data stored to journal file is copied to actual destination on the storage device. The other nested calls to FS_JOURNAL_End() simply close the transaction at that nesting level but do not copy any data.

It is mandatory that FS_JOURNAL_Begin() and FS_JOURNAL_End() are called in pair. The calls to these functions can be nested. The current nesting level can be queried via FS_JOURNAL_GetOpenCnt().

Example

```
#include "FS.h"

void SampleJournalBegin(void) {
  FS_FILE * pFile;

#if FS_SUPPORT_JOURNAL
  FS_JOURNAL_Begin("");               // Open the journal transaction.
#endif // FS_SUPPORT_JOURNAL
  pFile = FS_FOpen("File1.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Test 1", 6);
    FS_FClose(pFile);
  }
  pFile = FS_FOpen("File2.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Test 2...", 6);
    FS_FClose(pFile);
  }
  pFile = FS_FOpen("File3.txt", "w");
  if (pFile) {
    FS_Write(pFile, "Test 3...", 6);
    FS_FClose(pFile);
  }
#if FS_SUPPORT_JOURNAL
  FS_JOURNAL_End("");                 // Close the journal transaction.
#endif // FS_SUPPORT_JOURNAL
}
```


13.6.7 FS_JOURNAL_GetInfo()

Description

Returns information about the journal.

Prototype

```
int FS_JOURNAL_GetInfo(const char            * sVolumeName,
                       FS_JOURNAL_INFO * pInfo);
```

Parameters

Parameter	Description
sVolumeName	Journal instance identified by volume name (0-terminated string).
pInfo	out Journal information.

Return value

= 0 OK, information returned.
< 0 Error code indicating the failure reason.

Additional information

This function is optional. The application can call it to get information about the journal such as if the journal is enabled, the number of free sectors in the journal and so on.

FS_JOURNAL_GetInfo() mounts the specified volume if the auto mount feature is enabled for that volume and the volume is not mounted a the time of the call.

13.6.8 FS_JOURNAL_GetOpenCnt()

Description

Returns the number times the current journal transaction has been opened.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Journal instance identified by volume name (0-terminated string).

Return value

≥ 0 OK, number of nested calls.
≠ 0 Error code indicating the failure reason.

Additional information

This function is optional. The application can use FS_JOURNAL_GetOpenCnt() to check how many times FS_JOURNAL_Begin() has been called in a row without a call to FS_JOURNAL_End() in between.

13.6.9 FS_JOURNAL_GetStatCounters()

Description

Returns statistical information about the operation.

Prototype

```
int FS_JOURNAL_GetStatCounters(const char                    * sVolumeName,
                               FS_JOURNAL_STAT_COUNTERS * pStat);
```

Parameters

Parameter	Description
sVolumeName	Journal instance identified by volume name (0-terminated string).
pStat	out Statistical information.

Return value

= 0 OK, information returned.
≠ 0 Error code indicating the failure reason.

Additional information

This function is optional. It can be used to get information about the number of operations performed by the journal since the last file system mount operation or since the last call to FS_JOURNAL_ResetStatCounters().

FS_JOURNAL_GetStatCounters() is available only when the file system is compiled with either FS_JOURNAL_ENABLE_STATS set to 1 or with FS_DEBUG_LEVEL set to a value equal to or larger than FS_DEBUG_LEVEL_CHECK_ALL.

13.6.10 FS_JOURNAL_Invalidate()

Description

Cancels the pending journal transaction.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Journal instance identified by volume name (0-terminated string).

Return value

≥ 0 OK, journal data has been discarded.
≠ 0 Error code indicating the failure reason.

Additional information

This function is optional. It can be used to discard all the modifications stored in the journal during a journal transaction opened via FS_JOURNAL_Begin(). After the call to FS_JOURNAL_Invalidate() the current journal transaction is closed. In case of a journal transaction opened multiple times it is not necessary to call FS_JOURNAL_Invalidate() for the number of times the journal transaction has been opened.

A read sector cache has to be invalidated after canceling a journal transaction via FS_JOURNAL_Invalidate(). The application can configure a read sector cache via FS_AssignCache().

Example

```
#include "FS.h"

static int _GetData(U8 * pData) {
  int NumBytes;

  NumBytes = -1;
  //
  // ...
  //
  return NumBytes;
}

void SampleJournalInvalidate(void) {
  int       IsError;
  FS_FILE * pFile;
  U8        abBuffer[16];
  int       NumBytes;
  IsError = 0;
  //
  // Begin the journal transaction.
  //
#if FS_SUPPORT_JOURNAL
  FS_JOURNAL_Begin("");
#endif // FS_SUPPORT_JOURNAL
  //
  // Create the file and write to it.
  //
  pFile = FS_FOpen("Test.txt", "w");
  if (pFile) {
    while (1) {
      //
      // Get the data from an external source.
      //
      NumBytes = _GetData(abBuffer);
```



```
      if (NumBytes == 0) {
        FS_FClose(pFile);
        break;                      // No more data available.
      }
      if (NumBytes < 0) {
        IsError = 1;
        break;                      // Error, could not get data.
      }
      //
      // Write the data to file.
      //
      FS_Write(pFile, abBuffer, (U32)NumBytes);
    }
  }
  //
  // Close the transaction.
  //
#if FS_SUPPORT_JOURNAL
  if (IsError) {
    FS_JOURNAL_Invalidate("");
  } else {
    FS_JOURNAL_End("");
  }
#endif // FS_SUPPORT_JOURNAL
}
```


13.6.11 FS_JOURNAL_IsEnabled()

Description

Checks the journal operational status.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Journal instance identified by volume name (0-terminated string).

Return value

= 1 The journal is active. All the file system operations are fail safe.
= 0 The journal is not active. The file system operations are not fail safe.
< 0 Error code indicating the failure reason.

Additional information

This function is optional. The journal is automatically activated at file system mount if a valid journal file is present. The journal file can be created using `FS_JOURNAL_Create()` or `FS_JOURNAL_CreateEx()`. The journal can be enabled and disabled at runtime using `FS_JOURNAL_Enable()` and `FS_JOURNAL_Disable()` respectively.

13.6.12 FS_JOURNAL_IsPresent()

Description

Checks the presence of journal file.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Journal instance identified by volume name (0-terminated string).

Return value

= 1 OK, journal file is present and valid.
= 0 OK, journal file is not present.
< 0 Error code indicating the failure reason.

Additional information

This function is optional. The application can call it to check if a journal file is present on the specified volume and that the file is also valid.

FS_JOURNAL_IsPresent() mounts the specified volume if the auto mount feature is enabled for that volume and the volume is not mounted a the time of the call.

13.6.13 FS_JOURNAL_ResetStatCounters()

Description

Sets to 0 all statistical counters.

Prototype

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

Parameters

Parameter	Description
sVolumeName	Journal instance identified by volume name (0-terminated string).

Return value

= 0 OK, statistical counters cleared.
≠ 0 Error code indicating the failure reason.

Additional information

This function is optional. The statistical counters are cleared each time the volume is mounted. An application can use FS_JOURNAL_ResetStatCounters() to explicitly clear the statistical counters at runtime for example for testing purposes. The statistical counters can be queried via FS_JOURNAL_GetStatCounters()

FS_JOURNAL_ResetStatCounters() is available only when the file system is compiled with either FS_JOURNAL_ENABLE_STATS set to 1 or with FS_DEBUG_LEVEL set to a value equal to or larger than FS_DEBUG_LEVEL_CHECK_ALL.

13.6.14 FS_JOURNAL_SetFileName()

Description

Configures the name of the journal file.

Prototype

```
int FS_JOURNAL_SetFileName(const char * sVolumeName,
                           const char * sFileName);
```

Parameters

Parameter	Description
sVolumeName	Journal instance identified by volume name (0-terminated string).
sFileName	Name of the journal file (0-terminated string)

Return value

= 0 OK, file name set.
≠ 0 Error code indicating the failure reason.

Additional information

This function is optional. It can be used by an application to specify at runtime a name for the journal file. FS_JOURNAL_SetFileName() has to be called before the creation of the journal file via FS_JOURNAL_Create() or FS_JOURNAL_CreateEx().

FS_JOURNAL_SetFileName() is available only when the file system is compiled with the FS_MAX_LEN_JOURNAL_FILE_NAME configuration define set to a value greater than 0.

13.6.15 FS_JOURNAL_SetOnOverflowExCallback()

Description

Registers a callback function for the journal full event.

Prototype

```
void FS_JOURNAL_SetOnOverflowExCallback
                                (FS_JOURNAL_ON_OVERFLOW_EX_CALLBACK * pfOnOverflow);
```

Parameters

Parameter	Description
pfOnOverflow	Function to be invoked when the journal full event occurs.

Additional information

This function is optional. A journal full event occurs when there is no more free space in the journal file to store the modifications requested by the file system layer. When this event occurs, the data currently stored in the journal is copied to the actual destination on the storage device to make free space for the new data. This behavior can be changed via the return value of the callback function. Refer to FS_JOURNAL_ON_OVERFLOW_EX_CALLBACK for more information.

The file system is no longer fail safe in the time interval from the occurrence of the journal full event to the end of current journal transaction.

13.6.16 FS_JOURNAL_SetOnOverflowCallback()

Description

Registers a callback function for the journal full event.

Prototype

```
void FS_JOURNAL_SetOnOverflowCallback
                                  (FS_JOURNAL_ON_OVERFLOW_CALLBACK * pfOnOverflow);
```

Parameters

Parameter	Description
pfOnOverflow	Function to be invoked when the journal full event occurs.

Additional information

This function is optional. A journal full event occurs when there is no more free space in the journal file to store the modifications requested by the file system layer. When this event occurs, the data currently stored in the journal is copied to the actual destination on the storage device to make free space for the new data.

The file system is no longer fail safe in the time interval from the occurrence of the journal full event to the end of current journal transaction.

13.6.17 FS_JOURNAL_INFO

Description

Information about the journal.

Type definition

```
typedef struct {
  U8   IsEnabled;
  U8   IsFreeSectorSupported;
  U16  OpenCnt;
  U32  NumSectors;
  U32  NumSectorsFree;
} FS_JOURNAL_INFO;
```

Structure members

Member	Description
IsEnabled	Set to 1 if the journal is used to protect the integrity of the file system structure.
IsFreeSectorSupported	Set to 1 if the journal has been configured to forward “free sector” requests to device driver.
OpenCnt	Number of times the current transaction has been opened.
NumSectors	Number of logical sectors that can be stored in the journal.
NumSectorsFree	Number of logical sectors that do not store any file system data.

Additional information

The information about the journal can be queried via FS_JOURNAL_GetInfo()

NumSectorsFree is always smaller than or equal to NumSectors. The difference between NumSectors and NumSectorsFree represents the number of logical sectors that can still be stored to journal without causing a journal overflow.

IsEnabled and OpenCnt values are identical to the values returned by FS_JOURNAL_IsEnabled() and FS_JOURNAL_GetOpenCnt() respectively.

13.6.18 FS_JOURNAL_ON_OVERFLOW_EX_CALLBACK

Description

Prototype of the function that is called on a journal overflow event.

Type definition

```
typedef int (FS_JOURNAL_ON_OVERFLOW_EX_CALLBACK)
                                  (const FS_JOURNAL_OVERFLOW_INFO * pOverflowInfo);
```

Parameters

Parameter	Description
pOvrflowInfo	Information related to the overflow event.

Return value

0	The journal transaction has to continue to completion.
1	The journal transaction has to be aborted with an error.

Additional information

This type of callback function can be registered using `FS_JOURNAL_SetOnOverflowExCallback()`.

The return value indicates the journal module how to proceed when an overflow event occurs.

13.6.19 FS_JOURNAL_ON_OVERFLOW_CALLBACK

Description

Prototype of the function that is called on a journal overflow event.

Type definition

```
typedef void (FS_JOURNAL_ON_OVERFLOW_CALLBACK)(const char * sVolumeName);
```

Parameters

Parameter	Description
sVolumeName	Name of the volume on which the overflow event occurred.

Additional information

This type of callback function can be registered using FS_JOURNAL_SetOnOverflowCallback().

13.6.20 FS_JOURNAL_OVERFLOW_INFO

Description

Information about a journal overflow event.

Type definition

```
typedef struct {
  U8  VolumeIndex;
} FS_JOURNAL_OVERFLOW_INFO;
```

Structure members

Member	Description
VolumeIndex	Index of the volume on which the journal overflow occurred.

Additional information

This type of information is returned via a callback of type FS_JOURNAL_ON_OVERFLOW_EX_CALLBACK when a journal overflow event occurs.

13.6.21 FS_JOURNAL_STAT_COUNTERS

Description

Journal statistical counters.

Type definition

```
typedef struct {
  U32  WriteSectorCnt;
  U32  NumTransactions;
  U32  FreeSectorCnt;
  U32  OverflowCnt;
  U32  WriteSectorCntStorage;
  U32  ReadSectorCntStorage;
  U32  MaxWriteSectorCnt;
} FS_JOURNAL_STAT_COUNTERS;
```

Structure members

Member	Description
WriteSectorCnt	Number of sectors written by the file system to journal.
NumTransactions	Number of journal transactions performed.
FreeSectorCnt	Number of sectors freed.
OverflowCnt	Number of times the journal has been cleaned before the end of a transaction.
WriteSectorCntStorage	Number of sectors written by the journal to the storage device.
ReadSectorCntStorage	Number of sectors read by the journal from the storage device.
MaxWriteSectorCnt	Maximum number of sectors written by the file system to journal file in any transaction.

Additional information

The statistical counters can be queried via FS_JOURNAL_GetStatCounters(). The application can use FS_JOURNAL_ResetStatCounters() to set all the statistical counters to 0.

MaxWriteSectorCnt can be used to fine tune the size of the journal file.

13.7 Performance and resource usage

This section describes the memory requirements (RAM and ROM) of the Journaling component as well as the read and write performance.

13.7.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The following values have been measured using the SEGGER Embedded Studio IDE V4.20 configured to generate code for a Cortex-M4 CPU in Thumb mode and with the size optimization enabled.

Usage: 3.3 Kbytes

13.7.2 Static RAM usage

The static RAM usage represents the amount of RAM required by the Journaling component to store static variables. The number of bytes can be seen in the list file of the compiled Journaling component.

Usage: 16 bytes

13.7.3 Dynamic RAM usage

The dynamic RAM usage is the amount of RAM allocated by the Journaling component at runtime. The number of bytes allocated depends on the journal size and on the number of volumes on which the Journaling component is enabled. The approximate runtime RAM usage of the Journaling component 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 parameter specified in the call to `FS_JOURNAL_Create()` `FS_JOURNAL_CreateEx()`
`NumVolumes`	Number of volumes on which the journaling is active

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 the *Performance and resource usage* on page 1035 section.

All speed values are in Kbytes/second.

CPU type	Storage device	Write speed	Reed speed
NXP LPC2478 (57.6 MHz)	SST SST39VF201 (1x16 bit, no “write burst”)	**5.6**	**2534**
ST STM32F103 (72 MHz)	ST M29W128 (1x16, with “write burst”, 64 bytes)	**18.5**	**7877**

13.8 FAQs

Q: Can a journal be created when other files are already present on the disk?
A: Yes. The Journaling component saves the data in a regular file called per default "Journal.dat". If there is sufficient space left on the storage device there is no problem to create the journal even if other files are present.
Q: Can the journal file be re-created?
A: Yes. Follow this procedure to recreate the journal:

- disable the journaling via FS_JOURNAL_Disable()
- remove the journal file via FS_Remove()
- unmount the file system via FS_Unmount() or FS_UnmountForced()
- mount the file system via FS_Mount() or FS_MountEx()
- re-create the journal file via FS_JOURNAL_Create() or FS_JOURNAL_CreateEx()

Q: Can a journal be deleted?
A: Yes, by deleting the journal file.
Q: What if the journal isn't sufficiently large?
A: If the journal is not sufficiently large the data already present in the journal file is copied to the actual destination on the storage device in order to make room for the new data.
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. The Journaling component maintains an open counter that is incremented each time FS_JOURNAL_Begin() is called and is decremented with each call to FS_JOURNAL_End(). When the open counter reaches zero the data is copied from journal file to the actual destination on the storage device and the journal transaction is terminated.

Chapter 14

Encryption

This chapter documents the emFile Encryption component. The component can be used to stored the data in a secure way.

14.1 General information

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 data is stored in a readable form. Using the Encryption component the data can be made unreadable before using a secret key being stored to the storage device. Without the knowledge of the secret key it is not possible to make the data readable again.

The main features of the Encryption component are:

- 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 storage device or of individual files.
- An utility is available to decrypt/encrypt files on a PC.
- Two packages are available: Encryption (DES) and Extra Strong Encryption (DES and AES [1])

[1] AES encryption algorithm is subject to export regulations

14.2 How to use encryption

Using file encryption is very simple from the application's perspective. The following changes have to be made to an existing application in order to enable the encryption of individual files:

1. Enable file encryption in the emFile configuration. Refer to *Compile time configuration* on page 1007 for detailed information.
2. Call `FS_CRYPT_Prepare()` to initialize an encryption object. This operation has to be performed only once. Refer to `FS_CRYPT_Prepare()` for detailed information.
3. Open a file and call `FS_SetEncryptionObject()` to assign the encryption object to file handle. Refer to `FS_SetEncryptionObject()` for detailed information.

Everything else is done by the encryption component.

Example usage

This sample function opens a file and writes a text message to it. The file contents are encrypted using the DES encryption algorithm. The changes required to an application to support encryption are marked with the comment: `// Required for encryption`.

```
#include "FS.h"

void SampleEncryptionFile(void) {
  FS_FILE               * pFile;
#if FS_SUPPORT_ENCRYPTION
  const U8                aKey[8] = {1, 2, 3, 4};     // Required for encryption
  FS_CRYPT_OBJ            CryptObj;                   // Required for encryption
  static FS_DES_CONTEXT   _Context;                   // Required for encryption
  static int              _IsInited;                  // Required for encryption
#endif // FS_SUPPORT_ENCRYPTION

#if FS_SUPPORT_ENCRYPTION
  //
  // 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) {                               // Required for encryption
    FS_CRYPT_Prepare(&CryptObj, &FS_CRYPT_ALGO_DES,   // Required for encryption
                     &_Context, 512, aKey);           // Required for encryption
    _IsInited = 1;                                    // Required for encryption
  }
#endif // FS_SUPPORT_ENCRYPTION
  pFile = FS_FOpen("cipher.bin", "w");
  if (pFile) {
#if FS_SUPPORT_ENCRYPTION
    //
    // Assign the created encryption object to file handle.
    //
    FS_SetEncryptionObject(pFile, &CryptObj);         // Required for encryption
#endif // FS_SUPPORT_ENCRYPTION
    //
    // Write data to file using encryption.
    //
    FS_Write(pFile, "This message has been encrypted using SEGGER emFile.\n", 53);
    FS_FClose(pFile);
  }
}
```

The encryption component also support the encryption of entire volumes. For more information refer to *Encryption driver* on page 886.

14.3 Compile time configuration

The configuration of emFile can be changed via compile time flags that can be added to `FS_Conf.h` which is the main configuration file of the file system.

Macro	Type	Default value	Description
`FS_SUPPORT_ENCRYPTION`	B	0	Specifies whether support for the file encryption should be compiled in or not.

For detailed information about the configuration of emFile and the switch types, refer to *Configuration of emFile* on page 927.

14.4 API functions

The following table lists the available API functions.

Function	Description
FS_CRYPT_Prepare()	Initializes the object required for the encryption/decryption of file contents.
FS_CRYPT_Decrypt()	Decrypts the specified number of bytes.
FS_CRYPT_Encrypt()	Encrypts the specified number of bytes.
FS_SetEncryptionObject()	Assigns an encryption object to a file handle.

14.4.1 FS_CRYPT_Prepare()

Description

Initializes the object required for the encryption/decryption of file contents.

Prototype

```
void FS_CRYPT_Prepare(      FS_CRYPT_OBJ       * pCryptObj,
                      const FS_CRYPT_ALGO_TYPE * pAlgoType,
                            void               * pContext,
                            U32                  BytesPerBlock,
                      const U8                 * pKey);
```

Parameters

Parameter	Description
pCryptObj	Instance of the object to be initialized.
pAlgoType	in Encryption algorithm to be used.
pContext	in Data for the encryption algorithm.
BytesPerBlock	Size of the block to encrypt/decrypt at once.
pKey	in Password for data encryption.

Additional information

This function has to be called once for each encryption object. pAlgoType is a pointer to one of the following structures:

Algorithm type	Description
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 specified 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.

BytesPerBlock is a power of two value that has to be smaller than or equal to the logical sector size of the volume that stores the file to be encrypted or decrypted.

The number of bytes in pKey depends on the algorithm type as follows:

Algorithm type	Size in 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 *How to use encryption* on page 1039.

14.4.2 FS_CRYPT_Decrypt()

Description

Decrypts the specified number of bytes.

Prototype

```
void FS_CRYPT_Decrypt(const FS_CRYPT_OBJ * pCryptObj,
                            U8           * pDest,
                      const U8           * pSrc,
                            U32            NumBytes,
                            U32          * pBlockIndex);
```

Parameters

Parameter	Description
pCryptObj	in The instance of the encryption object.
pDest	out Decrypted data (plain text).
pSrc	in Data to be decrypted (cipher text).
NumBytes	Number of bytes to decrypt.
pBlockIndex	in Index of the decrypted data block. out Index of the next block to be decrypted.

Additional information

This function can be used on a host computer to decrypt a file encrypted on a target system using the encryption component. On a target system the data is decrypted automatically by the file system. pBlockIndex 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 the call to FS_CRYPT_Prepare() that was used to initialize the encryption object.

Example

For an example usage refer to the source code of the FSFileEncrypter.exe utility located in the Windows\FS\FS_FileEncrypter\Src folder of the emFile shipment.

14.4.3 FS_CRYPT_Encrypt()

Description

Encrypts the specified number of bytes.

Prototype

```
void FS_CRYPT_Encrypt(const FS_CRYPT_OBJ * pCryptObj,
                            U8           * pDest,
                      const U8           * pSrc,
                            U32            NumBytes,
                            U32          * pBlockIndex);
```

Parameters

Parameter	Description
pCryptObj	in The instance of the encryption object.
pDest	out Encrypted data (cipher text).
pSrc	in Data to be encrypted (plain text).
NumBytes	Number of bytes to encrypt.
pBlockIndex	in Index of the encrypted data block. out Index of the next block to be encrypted.

Additional information

This function can be used on a host computer to encrypt a file that can be later decrypted on a target system using the encryption component. On a target system the data is decrypted automatically by the file system when the application reads the data from file. pBlockIndex 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() that was used to initialize the encryption object.

Example

For an example usage refer to the source code of the FSFileEncrypter.exe utility located in the Windows\FS\FS_FileEncrypter\Src folder of the emFile shipment.

14.4.4 FS_SetEncryptionObject()

Description

Assigns an encryption object to a file handle.

Prototype

```
int FS_SetEncryptionObject(FS_FILE      * pFile,
                           FS_CRYPT_OBJ * pCryptObj);
```

Parameters

Parameter	Description
pFile	Handle to opened file.
pCryptObj	Instance of the object to be initialized.

Return value

= 0 OK, configured the object to be used for file encryption.
≠ 0 Error code indicating the failure reason.

Additional information

This function has to be called once immediately after the file has been opened and before any other operation on that file. The pointer to encryption object is saved internally by the file system to the file handle. This means that the memory it points to has to remain valid until the file is closed or until the FS_SetEncryptionObject() is called for the same file handle with pCryptObj set to NULL.

The encryption object can be initialized using FS_CRYPT_Prepare().

Example

Refer to *How to use encryption* on page 1039.

14.5 Encryption utility

emFile comes with command line utilities that allow the encryption and decryption of files on a host PC. Due to export regulations of encryption software two separate executables are provided that support cryptographic algorithms with different strengths.

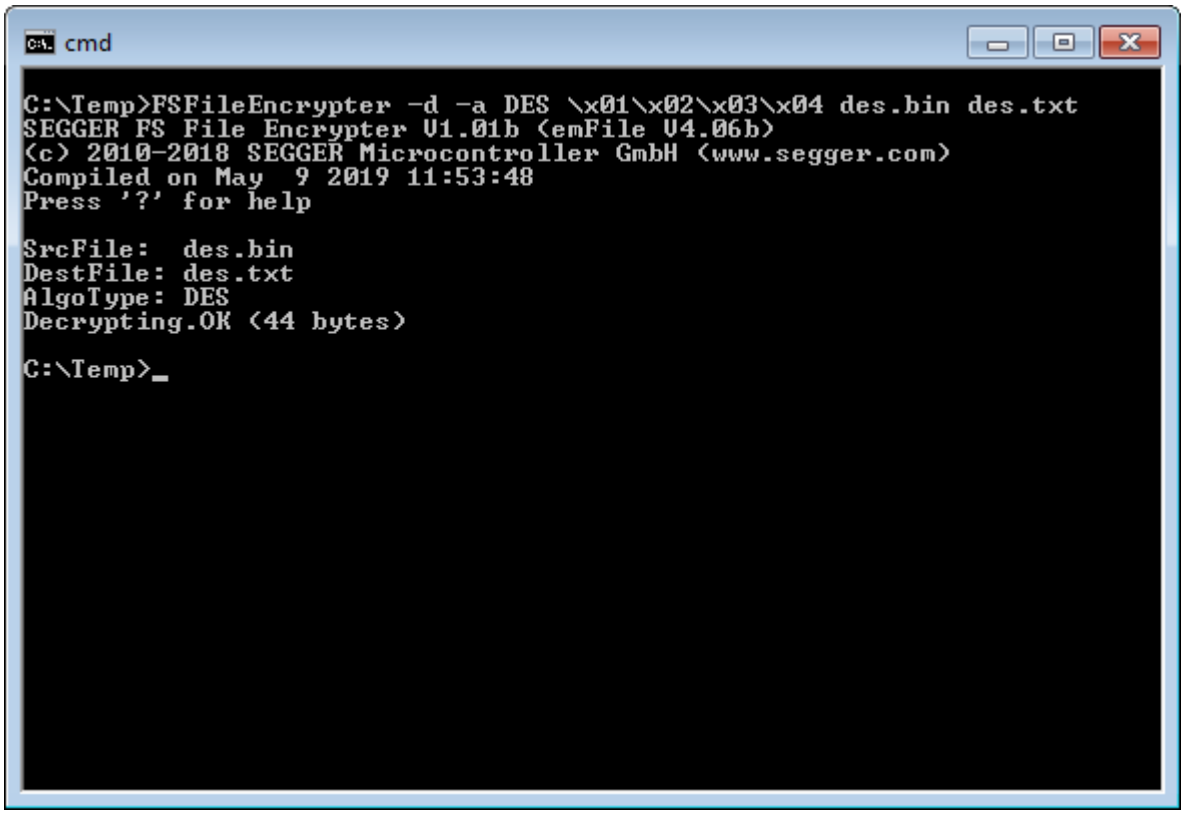
`FSFileEncrypter.exe` supports only the encryption algorithms with a key length smaller than or equal to 56-bit that are not subject to any export regulations. This utility supports only the DES cryptographic algorithm.

Encryption and decryption of files with any key length can be done using the `FSFileEncrypterES.exe` utility that supports the DES as well as the AES cryptographic algorithms.

14.5.1 Using the file encryption utility

The utilities can be invoked directly from the command line or via a batch file. In order to use the utility directly, a terminal window must be opened first. First, the name of the executable either `FSFileEncrypter.exe` or `FSFileEncrypterES.exe`, has to be input on the command line followed by optional and required arguments. By pressing the Enter key the utility starts the encryption or decryption operation as specified.

Below is a screen shot of the `FSFileEncrypter.exe` utility that is invoked to decrypt the contents of the file `des.bin`. The decrypted plain text is placed into the file `des.txt`. The used cryptographic algorithm is DES as specified via the `-a` option. Information about the decrypting process is displayed on the terminal window. In case of an error, a message is displayed indicating the failure reason and the utility returns with a status of 1. No destination file is created in this case.



14.5.2 Command line options

The following table lists the parameters which can be omitted when invoking the utilities.

Option	Description
-a	Selects the encryption algorithm.
-b	Sets the size of the encryption block.

Option	Description
-d	Performs decryption.
-h	Shows the usage message and exits.
-q	Suppresses log information.
-v	Shows version information and exits.

14.5.2.1 -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 cryptographic algorithm. The encryption key is the string `secret`.

```
C:>FSFileEncrypter -a DES secret plain.txt cipher.bin
```

14.5.2.2 -b

Description

Sets the size of the encryption block. Default block size is 512 bytes.

Syntax

-b <BlockSize>

Additional information

The parameter has to be be a power of twp value and represents the number of bytes in the block. The block size has to be equal to the value passed to BytesPerBlock in the call to `FS_CRYPT_Prepare()` function that was used to initializes the encryption object on the target application.

Example

Shows how to encrypt the contents of the file `plain.txt` file to `cipher.bin` file using the DES cryptographic 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
```


14.5.2.3 -d

Description

Performs decryption. Default is encryption.

Syntax

-d

Example

Shows how to decrypt the contents of the file `cipher.bin` to the `plain.txt` file using the DES cryptographic algorithm. The encryption key is the string `secret`.

```
FSFileEncrypter -d secret cipher.bin plain.txt
```

14.5.2.4 -h

Description

Shows the usage message and exits.

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.
```

14.5.2.5 -q

Description

Suppresses log information. Default is to log messages to console.

Syntax

`-q`

14.5.2.6 -v

Description

Shows version information and exits.

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
```

14.5.3 Command line arguments

The following table lists the mandatory parameters. They must be specified on the command line in the order they are described in this table.

Parameter	Description
<Key>	A string which specifies the encryption key.
<SrcFile>	Path to the file to read from.
<DestFile>	Path to the file to write to.

14.5.3.1 <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 cryptographic algorithm and the result is stored to the `cipher.bin` file. 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
```

14.5.3.2 <SrcFile>

Description

Path to the file to read from.

Additional information

It specifies the plain text file in case encryption is performed. When decrypting this parameter specifies the encrypted file. The parameters `SrcFile` and `DestFile` must specify two different files.

Example

Shows how to encrypt the contents of the file `plain.txt` to the file `cipher.bin` using the AES cryptographic algorithm. `plain.txt` is the source file.


```
C:>FSFileEncrypterES -a AES128 pass plain.txt cipher.bin
```

14.5.3.3 <DestFile>

Description

Path to the file to write to.

Additional information

It specifies the cipher text file in case encryption is performed. When decrypting this parameter specifies the plain text file. The parameters `SrcFile` and `DestFile` must specify two different files.

Example

Shows how to decrypt the contents of the file `cipher.bin` to the file `plain.txt` using the AES cryptographic algorithm. `plain.txt` is the destination file.

```
C:>FSFileEncrypterES -d -a AES256 pass cipher.bin plain.txt
```


14.6 Performance and resource usage

This section describes the memory requirements (RAM and ROM) of the encryption component as well as the read and write performance. This section lists only the values for the encryption at the file level. Refer to *Performance and resource usage* on page 889 for the performance and resource usage of volume encryption.

14.6.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The following values have been measured using the SEGGER Embedded Studio IDE V4.20 configured to generate code for a Cortex-M4 CPU in Thumb mode and with the size optimization enabled.

Usage: 0.4 Kbytes

In addition, one of the following cryptographic algorithms is required:

Algorithm type	ROM usage
FS_CRYPT_ALGO_DES	**3.2 Kbytes**
FS_CRYPT_ALGO_AES128	**12 Kbytes**
FS_CRYPT_ALGO_AES256	**12 Kbytes**

14.6.2 Static RAM usage

Static RAM usage is the amount of RAM required by the encryption component for static variables. The encryption component does not allocate any static variables.

14.6.3 Dynamic RAM usage

Dynamic RAM usage is the amount of RAM allocated by the encryption component at runtime. The encryption component requires one sector buffer for the encryption and decryption of data. The size of a sector buffer is by default 512 bytes. This value can be changed in the application via `FS_SetMaxSectorSize()`.

14.6.4 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 1051.

All values are in Kbytes/second.

CPU type	Storage device	Write speed	Reed speed
NXP 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**

Chapter 15

BigFile

This chapter describes the component for files larger than 4 Gbytes.

15.1 General information

The FAT and EFS file systems are not able to handle files larger than 4 Gbytes because the size of the file is stored a as 32-bit value. The BigFile component can be used in a target application to remove this limitation. Using the BigFile component a target application its able to create, read, write and otherwise perform typical operations on files larger than 4 Gbytes. This is realized by transparently distributing the data across fragment files smaller than 4 Gbytes. The created fragment files are compatible with the file system on which they are stored.

The main features of the BigFile component are:

- Can be used with both FAT and EFS file systems.
- All storage types such as NAND, NOR, SD/MMC/CompactFlash cards are supported.
- An utility available to merge or split files on a PC.

15.2 Theory of operation

BigFile is built on top of the FAT and EFS file systems. It transparently splits the data of a file that grows larger than 4 GB into smaller files. Consequently, they can be stored as normal files to the file system. The file system operations are redirected to the appropriate file, depending on the file position that is accessed by the embedded application.

Files with a size smaller than 2^{32} - 128 Kbytes are treated like normal files. In this case, the file access works as in a standard file system implementation. As soon as the file grows above the limit, the file becomes extended. An extended file consists of one base file and one or more continuation files. The continuation files are stored in the same directory as the base file, with the name derived from the name of the base file plus an additional file extension. The name of the base file remains unchanged. The file extension of a continuation file has the format `.<Index>.BigFile`, where the Index is a 3 digit decimal number. The index of the first continuation file is set to 1 and it increases by 1 with each additional continuation file. BigFile requires support for long file names on the underlying file system layer in order to be able to apply the additional file extension of continuation files.

When the base file grows larger than 2^{32} - 128 Kbytes, the first continuation file is created. This procedure is required in order to guarantee that the first byte in the continuation file is aligned to a logical sector boundary. This helps increase the performance of the file system accessing the file data. The same procedure is applied when the first continuation file exceeds the 2^{32} - 128 Kbytes limit, with the second continuation file being created and so on.

15.3 Configuration

The BigFile component can optionally be configured at compile time and it does not have to be configured at runtime to enable its operation.

15.3.1 Compile time configuration

The file handle of an opened BigFile has to store the fully qualified name of the file. The maximum number of characters in the file name including the 0-terminator is configured via the `FS_MAX_LEN_FULL_FILE_NAME` configuration define.

15.3.2 Runtime configuration

The component does not require any specific runtime configuration but the application has to make sure that the file system is initialized and the support for long file names is enabled before any of the BigFile API functions are called.

15.4 API functions

The table below lists the available API functions.

Function	Description
FS_BIGFILE_Close()	Closes a big file.
FS_BIGFILE_Copy()	Duplicates a big file.
FS_BIGFILE_FindClose()	Ends a directory scanning operation.
FS_BIGFILE_FindFirst()	Initiates a directory scanning operation and returns information about the first file or directory.
FS_BIGFILE_FindNext()	Returns information about the next file or directory in a directory scanning operation.
FS_BIGFILE_GetInfo()	Returns information about a big file.
FS_BIGFILE_GetPos()	Returns the read and write position in the big file.
FS_BIGFILE_GetSize()	Returns the size of the big file.
FS_BIGFILE_ModifyAttr()	Modifies file attributes.
FS_BIGFILE_Move()	Changes the location and the name of a big file.
FS_BIGFILE_Open()	Opens a big file.
FS_BIGFILE_Read()	Reads data from the big file.
FS_BIGFILE_Remove()	Deletes a big file.
FS_BIGFILE_SetPos()	Changes the access position in the big file.
FS_BIGFILE_SetSize()	Increases or reduces the size of a big file.
FS_BIGFILE_Sync()	Saves cached information to storage device.
FS_BIGFILE_Write()	Writes data to the big file.

15.4.1 FS_BIGFILE_Close()

Description

Closes a big file.

Prototype

```
int FS_BIGFILE_Close(FS_BIGFILE_OBJ * pFileObj);
```

Parameters

Parameter	Description
pFileObj	File object that identifies the opened file.

Return value

= 0 OK, file closed.
≠ 0 Error code indicating the failure reason.

Additional information

The application has to call FS_BIGFILE_Close() after it no longer needs to access the big file. FS_BIGFILE_Close() frees all the file system resources allocated to the opened file.

pFileObj has to point to a structure that has been successfully initialized via a call to FS_BIFILE_Open(). After the call to FS_BIGFILE_Close() the file object pFileObj is no longer valid and the memory allocated for it can be released by the application. The application can also use pFileObj as parameter to FS_BIFILE_Open() to open a file again.

Example

```
#include "FS.h"

void SampleBigFileClose(void) {
  FS_BIGFILE_OBJ FileObj;
  int            r;

  //
  // Opens a file for reading and then closes it.
  //
  r = FS_BIGFILE_Open(&FileObj, "Test.txt", FS_BIGFILE_OPEN_FLAG_READ);
  if (r == 0) {

    //
    // Perform read accesses to file
    //

    FS_BIGFILE_Close(&FileObj);
  }
}
```


15.4.2 FS_BIGFILE_Copy()

Description

Duplicates a big file.

Prototype

```
int FS_BIGFILE_Copy(const char * sFileNameSrc,
                    const char * sFileNameDest,
                          void * pBuffer,
                          U32    NumBytes);
```

Parameters

Parameter	Description
sFileNameSrc	Name of the file to be copied (0-terminated).
sFileNameDest	Name of the created file (0-terminated).
pBuffer	in Buffer for the copy operation (optional). Can be set to NULL.
NumBytes	Size of the copy buffer.

Return value

= 0 OK, file duplicated.
≠ 0 Error code indicating the failure reason.

Additional information

The source and destination files can be located on different volumes. If no working buffer is specified the function uses a buffer of 512 bytes that is allocated on the stack.

Example

```
#include "FS.h"

void SampleBigFileCopy(void) {
  //
  // Copies the contents of "Test.txt" to file "TestCopy.txt" in the folder "SubDir".
  //
  FS_BIGFILE_Copy("Test.txt", "SubDir\\TestCopy.txt", NULL, 0);
}
```


15.4.3 FS_BIGFILE_FindClose()

Description

Ends a directory scanning operation.

Prototype

```
int FS_BIGFILE_FindClose(FS_BIGFILE_FIND_DATA * pFD);
```

Parameters

Parameter	Description
pFD	Context of the directory scanning operation.

Return value

= 0 OK, scan context closed.
< 0 Error code indicating the failure reason.

Example

Refer to the sample usage of `FS_BIGFILE_FindFirst()`.

15.4.4 FS_BIGFILE_FindFirst()

Description

Initiates a directory scanning operation and returns information about the first file or directory.

Prototype

```
int FS_BIGFILE_FindFirst(      FS_BIGFILE_FIND_DATA * pFD,
                         const char                 * sDirName,
                               char                 * sFileName,
                               int                    SizeOfFileName);
```

Parameters

Parameter	Description
pFD	Context of the directory scanning operation.
sDirName	Name of the directory to search in.
sFileName	Buffer that receives the name of the file found.
SizeOfFileName	Size in bytes of the sFileName buffer.

Return value

= 1 OK, no files or directories available in directory.
= 0 OK, information about the file or directory returned.
< 0 Error code indicating the failure reason.

Example

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

void SampleBigFileFind(void) {
  FS_BIGFILE_FIND_DATA FindData;
  int                  r;
  char                 acFileName[32];

  //
  // Lists the names of all files and directories located in the root directory
  // of the default volume.
  //
  memset(acFileName, 0, sizeof(acFileName));
  r = FS_BIGFILE_FindFirst(&FindData, "", acFileName, sizeof(acFileName));
  if (r < 0) {
    printf("Cannot list directory (%s)\n", FS_ErrorNo2Text(r));
  } else {
    if (r == 1) {
      printf("Empty directory\n");
    } else {
      for (;;) {
        printf("%s\n", acFileName);
        r = FS_BIGFILE_FindNext(&FindData);
        if (r < 0) {
          printf("Cannot list directory (%s)\n", FS_ErrorNo2Text(r));
          break;
        } else {
          if (r == 1) {
            break;              // No more files or directories available.
          }
        }
      }
    }
  }
  FS_BIGFILE_FindClose(&FindData);
}
```


15.4.5 FS_BIGFILE_FindNext()

Description

Returns information about the next file or directory in a directory scanning operation.

Prototype

```
int FS_BIGFILE_FindNext(FS_BIGFILE_FIND_DATA * pFD);
```

Parameters

Parameter	Description
pFD	Context of the directory scanning operation.

Return value

= 1 OK, no files or directories available in directory.
= 0 OK, information about the file or directory returned.
< 0 Error code indicating the failure reason.

Example

Refer to the sample usage of `FS_BIGFILE_FindFirst()`.

15.4.6 FS_BIGFILE_GetInfo()

Description

Returns information about a big file.

Prototype

```
int FS_BIGFILE_GetInfo(const char            * sFileName,
                       FS_BIGFILE_INFO * pInfo);
```

Parameters

Parameter	Description
sFileName	Name of the file (partially qualified)
pInfo	out Information about the big file.

Return value

= 0 OK, information returned.
≠ 0 Error code indicating the failure reason.

Example

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

void SampleBigFileGetInfo(void) {
  FS_BIGFILE_INFO FileInfo;
  int             r;
  FS_FILETIME     FileTime;

  //
  // Gets information about a file and shows them.
  //
  memset(&FileInfo, 0, sizeof(FileInfo));
  r = FS_BIGFILE_GetInfo("Test.txt", &FileInfo);
  if (r == 0) {
    printf("Size:          %ull bytes\n", FileInfo.FileSize);
    printf("Attributes:    0x%02X\n", FileInfo.Attributes);
    FS_TimeStampToFileTime(FileInfo.CreationTime, &FileTime);
    printf("Creation time: %d-%.2d-%.2d %.2d:%.2d:%.2d",
           FileTime.Year, FileTime.Month,  FileTime.Day,
           FileTime.Hour, FileTime.Minute, FileTime.Second);
    FS_TimeStampToFileTime(FileInfo.LastAccessTime, &FileTime);
    printf("Access time:   %d-%.2d-%.2d %.2d:%.2d:%.2d",
           FileTime.Year, FileTime.Month,  FileTime.Day,
           FileTime.Hour, FileTime.Minute, FileTime.Second);
    FS_TimeStampToFileTime(FileInfo.LastWriteTime, &FileTime);
    printf("Write time:    %d-%.2d-%.2d %.2d:%.2d:%.2d",
           FileTime.Year, FileTime.Month,  FileTime.Day,
           FileTime.Hour, FileTime.Minute, FileTime.Second);
  }
}
```


15.4.7 FS_BIGFILE_GetPos()

Description

Returns the read and write position in the big file.

Prototype

```
int FS_BIGFILE_GetPos(FS_BIGFILE_OBJ * pFileObj,
                      U64            * pPos);
```

Parameters

Parameter	Description
pFileObj	File object that identifies the opened file.
pPos	out Position in the file as byte offset (0-based).

Return value

= 0 OK, file position returned.
≠ 0 Error code indicating the failure reason.

Additional information

The current file position can be modified by using FS_BIGFILE_SetPos() and is increased after each read or write operation by the number of bytes accessed.

Example

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

void SampleBigFileGetPos(void) {
  FS_BIGFILE_OBJ FileObj;
  U64            FilePos;
  int            r;

  //
  // Opens a file for writing, writes some data to it and queries the file position.
  //
  r = FS_BIGFILE_Open(&FileObj, "Test.txt",
                      FS_BIGFILE_OPEN_FLAG_WRITE | FS_BIGFILE_OPEN_FLAG_CREATE);
  if (r == 0) {
    FS_BIGFILE_GetPos(&FileObj, &FilePos);
    printf("File position before write: %ull\n", FilePos);
    FS_BIGFILE_Write(&FileObj, "Test", 4, NULL);
    FS_BIGFILE_GetPos(&FileObj, &FilePos);
    printf("File position after write: %ull\n", FilePos);
    FS_BIGFILE_Close(&FileObj);
  }
}
```


15.4.8 FS_BIGFILE_GetSize()

Description

Returns the size of the big file.

Prototype

```
int FS_BIGFILE_GetSize(FS_BIGFILE_OBJ * pFileObj,
                       U64            * pSize);
```

Parameters

Parameter	Description
pFileObj	File object that identifies the opened file.
pSize	out Size of the file in bytes.

Return value

= 0 OK, file size returned.
≠ 0 Error code indicating the failure reason.

Example

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

void SampleBigFileGetSize(void) {
  FS_BIGFILE_OBJ FileObj;
  U64            FileSize;
  int            r;

  //
  // Opens a file for writing, writes some data to it and queries the file size.
  //
  r = FS_BIGFILE_Open(&FileObj, "Test.txt",
                      FS_BIGFILE_OPEN_FLAG_WRITE | FS_BIGFILE_OPEN_FLAG_CREATE);
  if (r == 0) {
    FS_BIGFILE_Write(&FileObj, "Test", 4, NULL);
    FS_BIGFILE_GetSize(&FileObj, &FileSize);
    printf("File size: %ull\n", FileSize);
    FS_BIGFILE_Close(&FileObj);
  }
}
```


15.4.9 FS_BIGFILE_ModifyAttr()

Description

Modifies file attributes

Prototype

```
int FS_BIGFILE_ModifyAttr(const char     * sFileName,
                          unsigned   SetMask,
                          unsigned   ClrMask);
```

Parameters

Parameter	Description
sFileName	Name of the file (partially qualified)
SetMask	Attributes that have to be set.
ClrMask	Attributes that have to be cleared.

Return value

= 0 OK, attributes modified.
≠ 0 Error code indicating the failure reason.

Example

```
#include "FS.h"

void SampleBigFileModifyAttr(void) {
  //
  // Clears the ARCHIVE attribute and sets the SYSTEM and HIDDEN attributes
  // of the file "Test.txt".
  //
  FS_BIGFILE_ModifyAttr("Test.txt", FS_ATTR_SYSTEM | FS_ATTR_HIDDEN, FS_ATTR_ARCHIVE);
}
```


15.4.10 FS_BIGFILE_Move()

Description

Changes the location and the name of a big file.

Prototype

```
int FS_BIGFILE_Move(const char * sFileNameSrc,
                    const char * sFileNameDest);
```

Parameters

Parameter	Description
sFileNameSrc	Actual file name (partially qualified)
sFileNameDest	New file name (partially qualified)

Return value

= 0 OK, file moved.
≠ 0 Error code indicating the failure reason.

Example

```
#include "FS.h"

void SampleBigFileMove(void) {
  //
  // Moves the contents of "Test.txt" to file "TestCopy.txt" in the folder "SubDir".
  //
  FS_BIGFILE_Move("Test.txt", "SubDir\\TestCopy.txt");
}
```


15.4.11 FS_BIGFILE_Open()

Description

Opens a big file.

Prototype

```
int FS_BIGFILE_Open(      FS_BIGFILE_OBJ * pFileObj,
                    const char           * sFileName,
                          unsigned         Flags);
```

Parameters

Parameter	Description
pFileObj	File object that identifies the opened file.
sFileName	in Partially qualified name of the file to be opened (0-terminated character string).
Flags	Specifies how the file has to be accessed.

Return value

= 0 OK, file opened.
≠ 0 Error code indicating the failure reason.

Additional information

The application has to call this function once for each file it wants to access. FS_BIGFILE_Open() creates or opens the base file according to Flags and initializes pFileObj. The application has to allocate pFileObj and to make sure that it stays valid until FS_BIGFILE_Close() is called. pFileObj identifies the opened file within the application and is passed as parameter to all the other API functions that access the file.

FS_BIGFILE_Open() makes a copy of sFileName that is then stored in pFileObj. Therefore, the application can release the memory allocated to sFileName after FS_BIGFILE_Open() returns.

Flags is a bitwise OR combination of *File open flags* on page 1078. Permitted combinations are:

Flag combination	Description
FS_BIGFILE_OPEN_FLAG_READ	Open an existing file for reading
FS_BIGFILE_OPEN_FLAG_WRITE FS_BIGFILE_OPEN_FLAG_CREATE	Create a new file or truncate an existing one for writing
FS_BIGFILE_OPEN_FLAG_WRITE FS_BIGFILE_OPEN_FLAG_CREATE FS_BIGFILE_OPEN_FLAG_APPEND	Create a new file or append to an existing one for writing
FS_BIGFILE_OPEN_FLAG_READ FS_BIGFILE_OPEN_FLAG_WRITE	Open an existing file for reading and writing
FS_BIGFILE_OPEN_FLAG_READ FS_BIGFILE_OPEN_FLAG_WRITE FS_BIGFILE_OPEN_FLAG_CREATE	Create a new file or truncate an existing one for reading and writing
FS_BIGFILE_OPEN_FLAG_READ FS_BIGFILE_OPEN_FLAG_WRITE FS_BIGFILE_OPEN_FLAG_CREATE	Create a new file or append to an existing one for reading and writing

Flag combination	Description
FS_BIGFILE_OPEN_FLAG_APPEND	

Example

The next example demonstrates how to open a file for reading.

```
#include "FS.h"

void SampleBigFileOpenRead(void) {
  FS_BIGFILE_OBJ FileObj;
  int            r;

  //
  // Opens a file for reading on the root directory of the default volume.
  // The file has to exist before opening it.
  //
  r = FS_BIGFILE_Open(&FileObj, "Test.txt", FS_BIGFILE_OPEN_FLAG_READ);
  if (r == 0) {

    //
    // Perform read accesses to file
    //

    FS_BIGFILE_Close(&FileObj);
  }
}
```

This example demonstrates how to open a file for writing.

```
#include "FS.h"

void SampleBigFileOpenWrite(void) {
  FS_BIGFILE_OBJ FileObj;
  int            r;

  //
  // Opens a file for writing on the root directory of the default volume.
  // The file has to exist before opening it.
  //
  r = FS_BIGFILE_Open(&FileObj, "Test.txt", FS_BIGFILE_OPEN_FLAG_WRITE);
  if (r == 0) {

    //
    // Perform write accesses to file.
    //

    FS_BIGFILE_Close(&FileObj);
  }
}
```

This example demonstrates how to open a file for reading and writing.

```
#include "FS.h"

void SampleBigFileOpenReadWrite(void) {
  FS_BIGFILE_OBJ FileObj;
  int            r;

  //
  // Opens a file for reading and writing on the root directory of the "nand:0:" volume.
  // The file has to exist before opening it.
  //
  r = FS_BIGFILE_Open(&FileObj, "nand:0:\\Test.txt",
                      FS_BIGFILE_OPEN_FLAG_READ | FS_BIGFILE_OPEN_FLAG_WRITE);
  if (r == 0) {

    //
    // Perform read and write accesses to file
    //
```



```
    FS_BIGFILE_Close(&FileObj);
  }
}
```

This example demonstrates how to create a file.

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

void SampleBigFileOpenWriteCreate(void) {
  FS_BIGFILE_OBJ FileObj;
  int            r;

  //
  // Opens a file for writing on the folder "SubDir" of the default volume.
  // The directory delimiter is the '\' character. If the file does
  // not exist it is created. If the file exists it is truncated to 0.
  //
  r = FS_BIGFILE_Open(&FileObj, "\\SubDir\\Test.txt",
                      FS_BIGFILE_OPEN_FLAG_WRITE | FS_BIGFILE_OPEN_FLAG_CREATE);
  if (r == 0) {

    //
    // Perform write accesses to file.
    //

    FS_BIGFILE_Close(&FileObj);
  }
}
```

The next example demonstrates how open a file for writing at the end of it.

```
#include "FS.h"

void SampleBigFileOpenAppend(void) {
  FS_BIGFILE_OBJ FileObj;
  int            r;

  //
  // Opens a file for writing at the end on the root directory of the default volume.
  // The file has to exist before opening it.
  //
  r = FS_BIGFILE_Open(&FileObj, "Test.txt",
                      FS_BIGFILE_OPEN_FLAG_WRITE | FS_BIGFILE_OPEN_FLAG_APPEND);
  if (r == 0) {

    //
    // Perform write accesses to file.
    //

    FS_BIGFILE_Close(&FileObj);
  }
}
```


15.4.12 FS_BIGFILE_Read()

Description

Reads data from the big file.

Prototype

```
int FS_BIGFILE_Read(FS_BIGFILE_OBJ * pFileObj,
                    void           * pData,
                    U32              NumBytesToRead,
                    U32            * pNumBytesRead);
```

Parameters

Parameter	Description
pFileObj	File object that identifies the opened big file. It cannot be set to NULL.
pData	out Data read from big file.
NumBytesToRead	Number of bytes requested to be read from the big file.
pNumBytesRead	out Number of bytes actually read from the big file. It can be set to NULL.

Return value

= 0 OK, data read from file.
≠ 0 Error code indicating the failure reason.

Additional information

The data is read from the current position in the file that is stored in the file object. The current file position is set to first byte in the file (that is 0) when the file is opened. The file position is advanced by the number of bytes actually read from file.

If either pData is set to NULL or NumBytesToRead is set to 0 success is returned but no data is read. *pNumBytesRead is set to 0 in this case.

Example

```
#include "FS.h"

void SampleBigFileRead(void) {
  FS_BIGFILE_OBJ FileObj;
  int            r;
  U8             abData[4];

  //
  // Opens a file for reading on the root directory of the default volume.
  //
  r = FS_BIGFILE_Open(&FileObj, "Test.txt", FS_BIGFILE_OPEN_FLAG_READ);
  if (r == 0) {
    //
    // Read some data from the beginning of the opened file
    //
    FS_BIGFILE_Read(&FileObj, abData, sizeof(abData), NULL);
    //
    // Close the file.
    //
    FS_BIGFILE_Close(&FileObj);
  }
}
```


15.4.13 FS_BIGFILE_Remove()

Description

Deletes a big file.

Prototype

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

Parameters

Parameter	Description
sFileName	Name of the file to be removed (partially qualified)

Return value

= 0 OK, file removed.
≠ 0 Error code indicating the failure reason.

Example

```
#include "FS.h"

void SampleBigFileRemove(void) {
  //
  // Deletes the file "Test.txt" located in the folder "SubDir".
  //
  FS_BIGFILE_Remove("SubDir\\Test.txt");
}
```


15.4.14 FS_BIGFILE_SetPos()

Description

Changes the access position in the big file.

Prototype

```
int FS_BIGFILE_SetPos(FS_BIGFILE_OBJ * pFileObj,
                      U64              Pos);
```

Parameters

Parameter	Description
pFileObj	File object that identifies the opened file.
Pos	Position in the file as byte offset (0-based).

Return value

= 0 OK, file position changed.
≠ 0 Error code indicating the failure reason.

Additional information

The current file position can be queried using FS_BIGFILE_SetPos().

Example

```
#include "FS.h"

void SampleBigFileSetPos(void) {
  FS_BIGFILE_OBJ FileObj;
  U64            FilePos;
  int            r;

  //
  // Opens a file for writing, writes some data to it and then overwrites it.
  //
  r = FS_BIGFILE_Open(&FileObj, "Test.txt",
                      FS_BIGFILE_OPEN_FLAG_WRITE | FS_BIGFILE_OPEN_FLAG_CREATE);
  if (r == 0) {
    FS_BIGFILE_Write(&FileObj, "Test", 4, NULL);
    FS_BIGFILE_SetPos(&FileObj, 0);
    FS_BIGFILE_Write(&FileObj, "Done", 4, NULL);
    FS_BIGFILE_Close(&FileObj);
  }
}
```


15.4.15 FS_BIGFILE_SetSize()

Description

Increases or reduces the size of a big file.

Prototype

```
int FS_BIGFILE_SetSize(FS_BIGFILE_OBJ * pFileObj,
                       U64              Size);
```

Parameters

Parameter	Description
pFileObj	File object that identifies the opened file.
Size	New file size.

Return value

= 0 OK, file size changed.
≠ 0 Error code indicating the failure reason.

Additional information

If the file is extended then the additional bytes contain uninitialized data. In addition, the position in the file is not modified.

Example

```
#include "FS.h"

void SampleBigFileSetSize(void) {
  FS_BIGFILE_OBJ FileObj;
  int            r;

  //
  // Opens a file for writing and sets it size to 100 Kbytes.
  //
  r = FS_BIGFILE_Open(&FileObj, "Test.txt",
                      FS_BIGFILE_OPEN_FLAG_WRITE | FS_BIGFILE_OPEN_FLAG_CREATE);
  if (r == 0) {
    FS_BIGFILE_SetSize(&FileObj, 100 * 1024);
    FS_BIGFILE_Close(&FileObj);
  }
}
```


15.4.16 FS_BIGFILE_Sync()

Description

Saves cached information to storage device.

Prototype

```
int FS_BIGFILE_Sync(FS_BIGFILE_OBJ * pFileObj);
```

Parameters

Parameter	Description
pFileObj	File object that identifies the opened file.

Return value

= 0 OK, file synchronized.
≠ 0 Error code indicating the failure reason.

Example

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

void SampleBigFileSync(void) {
  FS_BIGFILE_OBJ FileObj;
  int            r;

  //
  // Opens a file for writing, writes some data to it while making sure that
  // the data is written to storage.
  //
  r = FS_BIGFILE_Open(&FileObj, "Test.txt",
                      FS_BIGFILE_OPEN_FLAG_WRITE | FS_BIGFILE_OPEN_FLAG_CREATE);
  if (r == 0) {
    FS_BIGFILE_Write(&FileObj, "Test1", 5, NULL);
    FS_BIGFILE_Sync(&FileObj);
    FS_BIGFILE_Write(&FileObj, "Test2", 5, NULL);
    FS_BIGFILE_Close(&FileObj);
  }
}
```


15.4.17 FS_BIGFILE_Write()

Description

Writes data to the big file.

Prototype

```
int FS_BIGFILE_Write(      FS_BIGFILE_OBJ * pFileObj,
                     const void           * pData,
                           U32              NumBytesToWrite,
                           U32            * pNumBytesWritten);
```

Parameters

Parameter	Description
pFileObj	File object that identifies the opened big file.
pData	in Data be written to big file.
NumBytesToWrite	Number of bytes requested to be written to the big file.
pNumBytesWritten	out Number of bytes actually written to the big file. Can be NULL.

Return value

= 0 OK, data written to file.
≠ 0 Error code indicating the failure reason.

Additional information

The data is written at the current position in the file that is stored in the file object. The current file position is set to first byte in the file (that is 0) when the file is opened. The file position is advanced by the number of bytes actually written to file.

Example

```
#include "FS.h"

void SampleBigFileWrite(void) {
  FS_BIGFILE_OBJ FileObj;
  int            r;

  //
  // Opens a file for writing on the root directory of the default volume.
  // The file has to exist before opening it.
  //
  r = FS_BIGFILE_Open(&FileObj, "Test.txt", FS_BIGFILE_OPEN_FLAG_WRITE);
  if (r == 0) {
    //
    // Write some data to file.
    //
    FS_BIGFILE_Write(&FileObj, "Test", 4, NULL);
    FS_BIGFILE_Close(&FileObj);
  }
}
```


15.4.18 FS_BIGFILE_INFO

Description

Information about a big file.

Type definition

```
typedef struct {
  U8   Attributes;
  U32  CreationTime;
  U32  LastAccessTime;
  U32  LastWriteTime;
  U64  FileSize;
} FS_BIGFILE_INFO;
```

Structure members

Member	Description
Attributes	File or directory attributes.
CreationTime	Date and time when the file was created.
LastAccessTime	Date and time when the file was accessed last.
LastWriteTime	Date and time when the file was written to last.
FileSize	Size of the file in bytes.

Additional information

The Attributes member is an or-combination of the following values: FS_ATTR_READ_ONLY, FS_ATTR_HIDDEN, FS_ATTR_SYSTEM, FS_ATTR_ARCHIVE, or FS_ATTR_DIRECTORY.

15.4.19 FS_BIGFILE_FIND_DATA

Description

Information about a file or directory.

Type definition

```
typedef struct {
  U8              Attributes;
  U32             CreationTime;
  U32             LastAccessTime;
  U32             LastWriteTime;
  U64             FileSize;
  char          * sFileName;
  U16             SizeOfFileName;
  FS_FIND_DATA    FindData;
  FS_BIGFILE_OBJ  FileObj;
} FS_BIGFILE_FIND_DATA;
```

Structure members

Member	Description
Attributes	Attributes of the file or directory.
CreationTime	Date and time when the file or directory was created.
LastAccessTime	Date and time when the file or directory was accessed last.
LastWriteTime	Date and time when the file or directory was modified last.
FileSize	Size of the file in bytes.
sFileName	Name of the file or directory as 0-terminated string. It points to the buffer passed as argument to FS_BIGFILE_Find-First().
SizeOfFileName	Internal. Not to be used by the application.
FindData	Internal. Not to be used by the application.
FileObj	Internal. Not to be used by the application.

Additional information

This structure contains also the context for the file listing operation. These members are considered internal and should not be used by the application. FS_BIGFILE_FIND_DATA is used as context by the FS_BIGFILE_FindFirst() and FS_BIGFILE_FindNext() pair of functions.

15.4.20 File open flags

Description

Specify how to open a file.

Definition

```
#define FS_BIGFILE_OPEN_FLAG_READ      0x01u
#define FS_BIGFILE_OPEN_FLAG_WRITE     0x02u
#define FS_BIGFILE_OPEN_FLAG_CREATE    0x04u
#define FS_BIGFILE_OPEN_FLAG_APPEND    0x08u
```

Symbols

Definition	Description
FS_BIGFILE_OPEN_FLAG_READ	The application can read from the opened file.
FS_BIGFILE_OPEN_FLAG_WRITE	The application can write to the opened file.
FS_BIGFILE_OPEN_FLAG_CREATE	The file is created if it does not exists or it is truncated to 0 if it exists.
FS_BIGFILE_OPEN_FLAG_APPEND	The application can write only at the end of the file.

Additional information

These flags can be used as values for the Flags parameter of FS_BIGFILE_Open(). More than one flag can be specified and in this case they have to be separated by a bitwise OR operator ('|').

15.5 BigFile utility

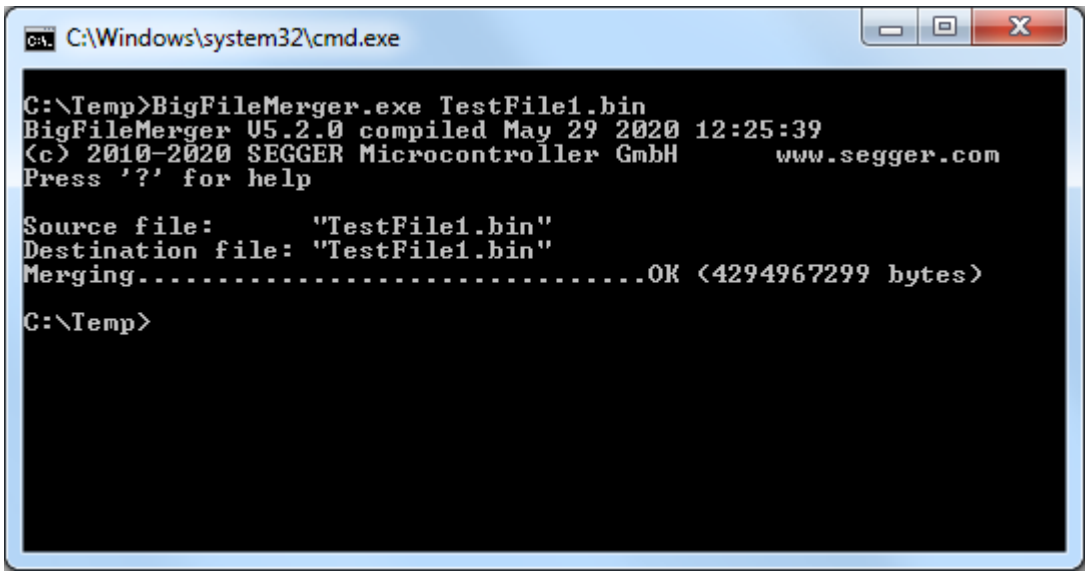
emFile comes with a convenient command line utility for PC that helps convert between a regular host file and one or more fragment files that can be managed by a target application using the BigFile component. The command line utility is able to merge the fragment files generated on a target device into a single file larger than 4 Gbytes. In addition, the command line utility is also able to split a file larger than 4 Gbytes into fragment files that can later be copied to a target device in order to be accessed by a target application.

15.5.1 Using the utility

The utility can be invoked directly from the command line or via a batch file. A terminal window must be opened first in order to use the utility interactively. First, the name of the executable (`BigFileMerger.exe`) has to be input on the command line followed by optional and required arguments. By pressing the Enter key the utility starts the merge or split operation as specified.

Below is a screen shot of the `BigFileMerger.exe` utility that is invoked to merge the contents of the file `TestFile1.bin` created on the target device.

The merged data replaces the contents of the original file. In case of an error a message is displayed indicating the failure reason and the utility returns with a status of 1. The original target file is left unmodified in this case.



If the original `TestFile1.bin` file is 6 Gbytes large then at the end of execution it is truncated to 4 Gbytes - 128 Kbytes. The remaining number of bytes is stored to a file named `TestFile1.bin.001.BigFile`

15.5.2 Command line options

The following table lists the parameters which can be omitted when invoking the utilities.

Option	Description
-d	Specifies a destination file.
-h	Shows the usage message and exits.
-q	Suppresses log information.
-s	Performs a split operation.
-v	Shows version information and exits.

15.5.2.1 -d

Description

Specifies the name of a destination file. By default the source file is overwritten.

Syntax

-d <DestFile>

Example

The following example shows how to merge the contents of the file `TestFileTarget.bin` to the file `TestFileHost.bin`

```
BigFileMerger -d TestFileHost.bin TestFileTarget.bin
```

The following example shows how to split the contents of the file `TestFileHost.bin` to the file `TestFileTarget.bin`

```
BigFileMerger -s -d TestFileTarget.bin TestFileHost.bin
```

15.5.2.2 -h

Description

Shows the usage message and exits.

Syntax

-h

Example

```
C:>BigFileMerger -h
BigFileMerger V5.2.0 compiled May 29 2020 16:56:35
(c) 2010-2020 SEGGER Microcontroller GmbH      www.segger.com
Press '?' for help

DESCRIPTION
  BigFile merge/split utility for SEGGER emFile.
USAGE
  BigFileMerger [-d <DestFile>] [-h] [-q] [-s] [-v] <SrcFile>
OPTIONS
  -d <DestFile>   Path to merged/split file to be created.
                  Default is the name of the source file.
  -h              Show this help information.
  -q              Do not show log messages.
  -s              Split a big file. Default is merging.
  -v              Show version information.
ARGUMENTS
  <SrcFile>       Path to file to be merged/split.
```

15.5.2.3 -q

Description

Suppresses log information. By default all the log messages are shown in the terminal window.

Syntax

-q

15.5.2.4 -s

Description

Performs a split operation. By default a merge operation is performed.

Syntax

`-s`

Example

The following example demonstrates how to split the contents of the file `TestFile1.bin`. The original file is overwritten. The created fragment files can be copied to the target device.

```
BigFileMerger -s TestFile1.bin
```

15.5.2.5 -v

Description

Shows version information and exits.

Syntax

`-v`

Example

```
C:>BigFileMerger -v
BigFileMerger V5.1.0 compiled May 27 2020 18:30:45
(c) 2010-2020 SEGGER Microcontroller GmbH      www.segger.com
Press '?' for help
```

15.5.3 Command line arguments

The following table lists the mandatory parameters.

Parameter	Description
SrcFile	Path to the file to be converted.

15.5.3.1 SrcFile

Description

Path to the file to be converted.

Additional information

In case of a merge operation `SrcFile` specifies the name of the first fragment file of a file created via the BigFile component on a target device. If a split operation is executed then `SrcFile` specifies the name of the host file to be converted.

Example

The following example demonstrates how to merge the contents of the file `TestFile1.bin` generated on a target device via the BigFile component.

```
BigFileMerger TestFile3.bin
```


Chapter 16

NAND Image Creator

This chapter provides information about an utility for creating NAND images.

16.1 General information

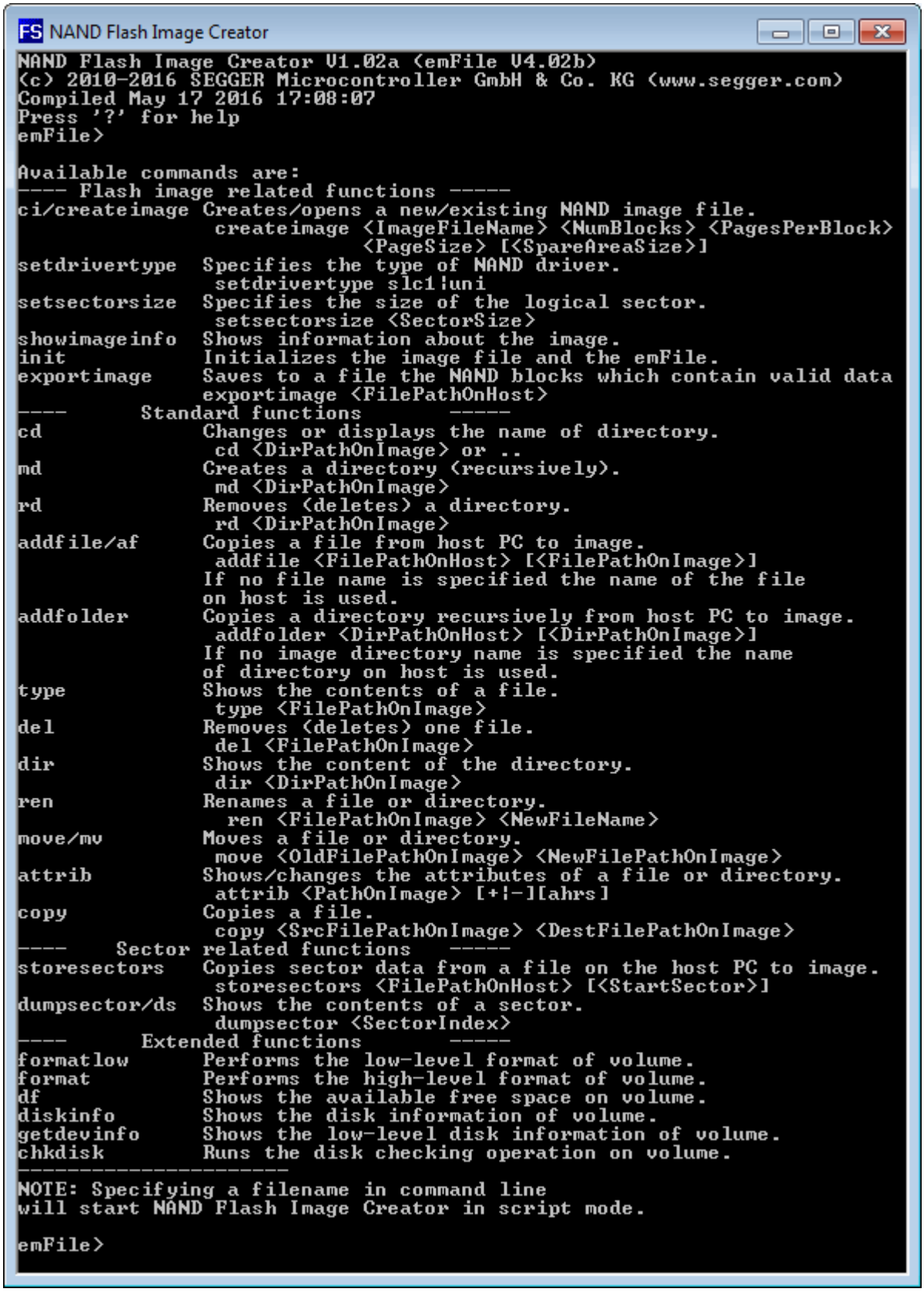
The NAND Image Creator is a command line utility that can be used to create a file containing 1-to-1 representation of the contents of a NAND flash device (image file) with preselected files and directories found on the host PC.

The image file is created by copying files and directories from the host PC to the image file using emFile API functions creating the file structure which is required on the target device.

The image file created using the NAND Image Creator is a standard binary file that can be programmed directly into the NAND flash device of the target via a SEGGER Flasher programmer (https://www.segger.com/products/flash-in-circuit-programmers/) or any other third party utility.

16.2 Using the NAND Image Creator

The NAND Image Creator utility can be used like any other command line based program. For the ease of usage many commands are kept similar to their DOS counterparts. By entering "?" or just pressing the Enter key an overview of the commands is printed on the screen and it should look like the picture below.



The NAND Image Creator utility can work in two modes: interactive and script. The interactive mode is entered when the utility is started with no command line arguments. In this mode the user is prompted to type in the commands to be executed. The script mode is entered when the utility is started with a command line argument that specifies the name of a file containing the commands to be executed, one command per line.

16.2.1 Sample script file

The following sample shows a simple script file which does the following:

- Creates a 128 Mbyte NAND flash image file `C:\flash.bin`
- Sets the type of NAND driver which will be used on the target to access the NAND flash device to Universal NAND driver. This command is optional since the default driver type.
- Creates the image file.
- Low-level formats the volume on the image file.
- High-level formats the volume on the image file.
- Copies the `Test.txt` file from drive `C:` of the host Windows PC to root directory of the volume stored on the image file.
- Copies recursively the contents of `Data` directory from host PC to `Data` directory of the volume stored on the image file.

```
createimage C:\flash.bin 1024 64 2048
setdrivertype uni
init
formatlow
format
addfile C:\Test.txt
addfolder C:\Data\
q
```


16.3 Supported commands

The image file can be processed by using build it commands of the NAND Image Creator utility. The table below lists all the available commands followed by a detailed description. The optional command parameters are enclosed in square brackets in the syntax description of a command.

Command	Description
Image related commands	
createimage	Creates/opens a new/existing image file.
setdrivertype	Configures the NAND driver type.
setsectorsize	Configures the logical sector size.
addpartition	Configures a logical partition.
setecclevel	Configures the error correction type.
setnumworkblocks	Configures the number of work blocks.
showimageinfo	Shows information about the image file.
init	Initializes the image file.
exportimage	Saves to a file the NAND blocks that contain valid data.
File and directory related commands	
cd	Changes the current working directory.
md	Creates a directory.
rd	Removes a directory.
addfile	Copies a file from the host PC to the image file.
addfolder	Copies a folder including its subfolders and files from the host PC to the image file.
type	Shows the contents of a file.
del	Deletes a file.
dir	Lists the contents of the directory.
ren	Renames a file or directory.
move	Moves a file or directory to another location.
attr	Shows/changes the file/directory attributes.
copy	Copies a file to another location.
exportfile	Copies a file from image file to host PC.
Logical sector related commands	
storesectors	Copies logical sector data from a file on the PC host to image file.
dumpsector	Shows the contents of a logical sector.
copysectors	Copies logical sector data to another location.
Volume specific commands	
formatlow	Low-level formats a volume.
format	High-level formats a volume.
df	Shows the available free space on a volume.
diskinfo	Shows information about a volume.
getdevinfo	Shows information about a storage device.
checkdisk	Checks a volume for errors.
creatembr	Writes partition table to master boot record (MBR).
listvolumes	Shows the names of all available volumes.

16.3.1 Image related commands

The following sections describe the commands required to create an image file.

16.3.1.1 createimage

Description

Creates/opens a new/existing image file.

Syntax

```
createimage <ImageFileName> <NumBlocks> <PagesPerBlock> <PageSize>
            [<SpareAreaSize>]
```

Parameters

Parameter	Description
ImageFileName	Name of the image file to be created/opened.
NumBlocks	Number of NAND blocks.
PagesPerBlock	Number of pages in a NAND block.
PageSize	Number of bytes in a NAND page excluding the spare area.
SpareAreaSize	Number of bytes in the sparea are of a NAND page.

Additional information

This command is mandatory and it must be the first command to be executed.

The size of the image file is NumBlocks * PagesPerBlock * (PageSize + SpareAreaSize) bytes large. The image file is silently overwritten if the image file already exists but the calculated size of the file does not match. If SpareAreaSize is not specified then the utility calculates it as 1/32 of PageSize.

PageSize has to be a power of 2 value. The values for PagesPerBlock, PageSize and SpareAreaSize have to match the values of the NAND flash device the file image will be stored to otherwise the the application will not be able to access the data via emFile. NumBlocks has to match either the maximum number of blocks in the NAND flash device or the number of blocks specified as the second parameter in the call to FS_NAND_SetBlockRange() or FS_NAND_UNI_SetBlockRange().

Example

The example shows how to create 128 Mbytes image file.

```
createimage C:\nand.bin 1024 64 2048
```


16.3.1.2 setdrivertype

Description

Configures the NAND driver type.

Syntax

```
setdrivertype <DriverType>
```

Parameters

Parameter	Description
DriverType	The type of NAND driver used to access the files on the target. • slc1 - SLC1 NAND driver • uni - Universal NAND driver

Additional information

This command is optional. If not executed an image file is created that can be accessed on the target using the Universal NAND driver.

Example

The examples shows how to specify that the SLC1 NAND driver is used on the target to access the NAND flash.

```
setdrivertype slc1
```


16.3.1.3 setsectorsize

Description

Configures the logical sector size.

Syntax

```
setsectorsize <SectorSize>
```

Parameters

Parameter	Description
SectorSize	The size of the logical sector in bytes.

Additional information

This command is optional. It can be used to set the size of the logical sector when the SLC1 NAND driver type is selected. Any power of 2 value between 512 bytes and the number of bytes in a NAND page can be used. When this command is not executed the size of the logical sector is set automatically to the configured page size of NAND flash device. The size of the logical sector as to be smaller than or equal to the size of the logical sector configured on the target application via FS_SetMaxSectorSize().

Example

This example shows how to set the logical sector size to 512 bytes.

```
setsectorsize 512
```


16.3.1.4 addpartition

Description

Configures a logical partition.

Syntax

```
addpartition <StartSector> <NumSectors> [<PartType>]
```

Parameters

Parameter	Description
StartSector	Index of the first sector in the partition.
NumSectors	Number of sectors in the partition.
PartType	Type of the partition to create. Permitted values are: • `normal` - The entire partition space is used by the file system as storage (default). • `raid1` - The file system can use only half of the partition space as storage. The other half is used as mirror partition.

Additional information

This command is optional. It allows the creation of logical partitions on the image file. The command fails if the defined sector range overlaps the range of an already defined logical partition. A maximum of four logical partitions can be created. The partition list has to be stored to image after initialization of the image using the `creatembr` command. The data on the created logical partitions can be accessed using the Disk Partition driver. The logical partitions can be formatted using the `format` command. The value of StartSector is relative to the beginning of the storage device.

Example

This example shows how to configure 2 logical partitions. The sector 0 is reserved for Master Boot Record where the partition list is stored.

```
addpartition 1 10000
addpartition 10001 20000
```

A possible emFile configuration that allows the application to access the logical partitions looks like this:

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

#define ALLOC_SIZE      0x9000          // Memory pool for the file system in bytes

static U32 _aMemBlock[ALLOC_SIZE / 4];  // Memory pool used for semi-dynamic allocation.

void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  FS_SetMaxSectorSize(2048);
  //
  // Add and configure the NAND partition. Volume name: "nand:0:"
  //
  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);
  FS_NAND_ONFI_SetHWType(0, &FS_NAND_HW_Template);
  //
  // Add and configure the first logical volume. Volume name: "diskpart:0"
  //
  FS_AddDevice(&FS_DISKPART_Driver);
  FS_DISKPART_Configure(0, &FS_NAND_UNI_Driver, 0, 0);
  //
  // Add and configure the second non-RAID volume. Volume name: "diskpart:1"
```



```
  //
  FS_AddDevice(&FS_DISKPART_Driver);
  FS_DISKPART_Configure(1, &FS_NAND_UNI_Driver, 0, 1);
}
```


16.3.1.5 setecclevel

Description

Configures the error correction type.

Syntax

```
setecclevel <NumBitErrors>
```

Parameters

Parameter	Description
NumBitErrors	Number of bit errors the ECC stored to spare area must be able to correct. Permitted values are: • 0 - hardware ECC. • 1 - 1-bit software ECC.

Additional information

This command is optional. If not executed a 1-bit software calculated ECC is stored to spare area. If NumBitErrors is set to 0 then the bytes in the spare area reserved for the ECC are set to 0xFF.

Example

This example shows how to set the correction level to hardware ECC.

```
setecclevel 0
```


16.3.1.6 setnumworkblocks

Description

Configures the number of work blocks.

Syntax

```
setnumworkblocks <NumBlocks>
```

Parameters

Parameter	Description
NumBlocks	Number of NAND blocks reserved for work blocks.

Additional information

This command is optional. If this command is not executed the number of work blocks is set to 3. The number of work blocks specified via this command has to match the number of work blocks configured by the target application via `FS_NAND_SetNumWorkBlocks()` or `FS_NAND_UNI_SetNumWorkBlocks()`.

Example

This example shows how to set the number of work blocks to 8.

```
setnumworkblocks 8
```


16.3.1.7 showimageinfo

Description

Shows information about the image file.

Syntax

```
showimageinfo
```

Example

```
showimageinfo
  Name:             nand.bin
  Driver type:      Universal
  Num blocks:       1024
  Pages per block:  64
  Page size:        2048 bytes
  Spare area size:  64 bytes
  Bytes per sector: 2048 bytes
```


16.3.1.8 init

Description

Initializes the image file.

Syntax

```
init
```

Additional information

This command must be executed before any other command which operates on files and directories.

Example

```
init
```


16.3.1.9 exportimage

Description

Saves to a file the NAND blocks that contain valid data.

Syntax

```
exportimage <FilePathOnHost>
```

Parameters

Parameter	Description
FilePathOnHost	Path to destination file on host PC.

Additional information

This command is optional. This command can be used to create an image file that contains only NAND blocks with valid data. The NAND blocks that contain invalid data or are empty are skipped. The created file can be used to program the NAND flash device. The programming tool has to make sure that the the NAND blocks that are not programmed are empty that is all the bytes in these blocks are set to `0xFF`. Typically, the created file is smaller than the original image file which helps reduce the programming time.

Example

```
exportimage C:\NAND.bin
```


16.3.2 File and directory related commands

The following sections describe the commands that can be used to operate on the files and directories stored on the image file. The commands described in this section can be executed only after the execution of the `init` command.

16.3.2.1 cd

Description

Changes the current working directory.

Syntax

```
cd <DirPathOnImage>
```

Parameters

Parameter	Description
DirPathOnImage	Path to new directory to change to.

Additional information

If DirPathOnImage is set to .. the command changes to the parent directory. After initialization the current directory is set to the root directory.

Example

In the following sample the working directory is changed to Test and then back to the original directory.

```
cd Test
cd ..
```


16.3.2.2 md

Description

Creates a directory.

Syntax

```
md <DirPathOnImage>
```

Parameters

Parameter	Description
DirPathOnImage	Path to directory to be created.

Additional information

This command is able to create any missing directories in the path.

Example

```
md Test\Dir\
```


16.3.2.3 rd

Description

Removes a directory.

Syntax

```
rd <DirPathOnImage>
```

Parameters

Parameter	Description
DirPathOnImage	Path to directory to be deleted.

Additional information

The directory to be deleted must be empty.

Example

```
rd Test
```


16.3.2.4 addfile

Description

Copies a file from the host PC to the image file.

Syntax

```
addfile <FilePathOnHost> [<FilePathOnImage>]
```

Parameters

Parameter	Description
FilePathOnHost	Path to the source file on host PC.
FilePathOnImage	Path to destination file on the image.

Additional information

If FilePathOnImage is not specified then the name of the source file is used.

Example

The following command copies the contents of the Test.txt file from host PC to Dest.txt file on image.

```
addfile C:\Test.txt Dest.txt
```

The following command copies the contents of Test.txt file to the file with the same name of the image.

```
addfile C:\Test.txt
```


16.3.2.5 addfolder

Description

Copies a folder including its subfolders and files from the host PC to the image file.

Syntax

```
addfolder <DirPathOnHost> [<DirPathOnImage>]
```

Parameters

Parameter	Description
DirPathOnHost	Path to source directory on the host PC.
DirPathOnImage	Path to destination directory on the image.

Additional information

If DirPathOnImage is not specified then the name of the source directory is used.

Example

The following command copies the contents of the Data directory on PC to Dest directory on the image.

```
addfolder C:\Data Dest
```

The following command copies the contents of the Data directory on PC to a directory with the same name on the image.

```
addfolder C:\Data
```


16.3.2.6 type

Description

Shows the contents of a file.

Syntax

```
type <FilePathOnImage>
```

Parameters

Parameter	Description
FilePathOnImage	Path to the file on image to be shown.

Example

The following command line shows the contents of the file `Test.txt` located on the root directory of the image.

```
type Test.txt
```


16.3.2.7 del

Description

Deletes a file.

Syntax

```
del <FilePathOnImage>
```

Parameters

Parameter	Description
FilePathOnImage	Path to the file on image to be deleted.

Example

The following command line deletes the file `Test.txt` located on the root directory of the image.

```
del Test.txt
```


16.3.2.8 dir

Description

Lists the contents of the directory.

Syntax

```
dir [<DirPathOnImage>]
```

Parameters

Parameter	Description
DirPathOnImage	Path to directory on image to be listed.

Additional information

If DirPathOnImage is not specified then the command lists the contents of the current working directory.

Example

The following command line shows the contents of the directory TestDir.

```
dir nand:0:\TestDir
.                    (Dir) Attributes: ----
..                   (Dir) Attributes: ----
SubDir               (Dir) Attributes: ----
Test1.txt                  Attributes: A--- Size: 30
Test2.txt                  Attributes: A--- Size: 31
```


16.3.2.9 ren

Description

Renames a file or directory.

Syntax

```
ren <FilePathOnImage> <NewFileName>
```

Parameters

Parameter	Description
FilePathOnImage	Path to file or directory that has to be renamed.
NewFileName	New name of the file or directory.

Additional information

If DirPathOnImage is not specified then the command lists the contents of the current working directory.

Example

The following command line changes the name of Test.txt file located in the Test directory to Test2.txt.

```
ren Test\Test.txt Test2.txt
```


16.3.2.10 move

Description

Moves a file or directory to another location.

Syntax

```
move <SrcFilePathOnImage> <DestFilePathOnImage>
```

Parameters

Parameter	Description
SrcFilePathOnImage	Path to file or directory to be moved.
DestFilePathOnImage	Path to destination file or directory on the image.

Example

The following command line moves the `Test.txt` file located in the `Test` directory to the root directory. The file is renamed to `New.txt`.

```
move Test\Test.txt New.txt
```


16.3.2.11 attr

Description

Shows/changes the file/directory attributes.

Syntax

```
attr <PathOnImage> [<Operation> <Attributes>]
```

Parameters

Parameter	Description
PathOnImage	Path to file or directory.
Operation	Operation to be executed on attributes. Permitted values are: • + sets the attributes. • - clears the attributes.
Attributes	The list of attributes to be changed The following attributes are supported: • a - archive. • h - hidden. • r - read-only. • s - system.

Example

The following example the clears the archive and read-only attributes of the Test.txt file.

```
attr Test.txt -ar
```


16.3.2.12 copy

Description

Copies a file to another location.

Syntax

```
copy <SrcFilePathOnImage> <DestFilePathOnImage>
```

Parameters

Parameter	Description
SrcFilePathOnImage	Path to the file to be copied.
DestFilePathOnImage	Path to destination the file.

Example

The following example copies the contents of the `Test.txt` to the `Test2.txt` file.

```
copy Test.txt Test2.txt
```


16.3.2.13 exportfile

Description

Copies a file from image file to host PC.

Syntax

```
exportfile <FilePathOnImage> <FilePathOnHost>
```

Parameters

Parameter	Description
FilePathOnImage	Path to the source file on image.
FilePathOnHost	Path to the destination file on PC.

Example

The following command copies the contents of the `Test.txt` file from image to `Dest.txt` file on host PC.

```
exportfile Test.txt C:\Dest.txt
```


16.3.3 Logical sector related commands

The following sections describe commands that can be used to operate on logical sectors.

16.3.3.1 storesectors

Description

Copies logical sector data from a file on the PC host to image file.

Syntax

```
storesectors <FilePathOnImage> [<StartSector>]
```

Parameters

Parameter	Description
FilePathOnImage	Path to the file on host that contains sector data.
StartSector	Index of the first logical sector to be written.

Additional information

This command is optional. The sector data must be stored sequentially in the file. For example if the sector size is 2048 bytes and the data is stored starting from sector index 100 then the byte at offset 0 in the file stores the first byte of sector 100, the byte at offset 2048 in the file stores the first byte of sector 101 and so on. The size of the file has to be a multiple of sector size. The size of the sector is the size specified as the argument to `setsectorsize` command.

Example

This example reads sector data from the file `SectorData.bin` and writes it starting from sector index 100.

```
storesectors SectorData.bin 100
```


16.3.3.2 dumpsector

Description

Shows the contents of a logical sector.

Syntax

```
dumpsector <SectorIndex>
```

Parameters

Parameter	Description
SectorIndex	Index of the logical sector to be displayed.

Example

The following command shows the contents of the logical sector 0.

```
dumpsector 0
00000000 34 33 32 31 34 33 32 31 34 33 32 31 34 33 32 31 4321432143214321
00000010 34 33 32 31 34 33 32 31 34 33 32 31 34 33 32 31 4321432143214321
00000020 34 33 32 31 34 33 32 31 34 33 32 31 34 33 32 31 4321432143214321
00000030 0D 0A 34 33 32 31 34 33 32 31 34 33 32 31 34 33 ..43214321432143
00000040 32 31 34 33 32 31 34 33 32 31 34 33 32 31 34 33 2143214321432143
00000050 32 31 34 33 32 31 34 33 32 31 34 33 32 31 34 33 2143214321432143
00000060 32 31 0D 0A 34 33 32 31 34 33 32 31 34 33 32 31 21..432143214321
00000070 34 33 32 31 34 33 32 31 34 33 32 31 34 33 32 31 4321432143214321
00000080 34 33 32 31 34 33 32 31 34 33 32 31 34 33 32 31 4321432143214321
00000090 34 33 32 31 0D 0A 34 33 32 31 34 33 32 31 34 33 4321..4321432143
000000A0 32 31 34 33 32 31 34 33 32 31 34 33 32 31 34 33 2143214321432143
000000B0 32 31 34 33 32 31 34 33 32 31 34 33 32 31 34 33 2143214321432143
000000C0 32 31 34 33 32 31 0D 0A 34 33 32 31 34 33 32 31 214321..43214321
000000D0 34 33 32 31 34 33 32 31 34 33 32 31 34 33 32 31 4321432143214321
000000E0 34 33 32 31 34 33 32 31 34 33 32 31 34 33 32 31 4321432143214321
000000F0 34 33 32 31 34 33 32 31 0D 0A 34 33 32 31 34 33 43214321..432143
00000100 32 31 34 33 32 31 34 33 32 31 34 33 32 31 34 33 2143214321432143
00000110 32 31 34 33 32 31 34 33 32 31 34 33 32 31 34 33 2143214321432143
00000120 32 31 34 33 32 31 34 33 32 31 0D 0A 34 33 32 31 2143214321..4321
00000130 34 33 32 31 34 33 32 31 34 33 32 31 34 33 32 31 4321432143214321
00000140 34 33 32 31 34 33 32 31 34 33 32 31 34 33 32 31 4321432143214321
00000150 34 33 32 31 34 33 32 31 34 33 32 31 0D 0A 34 33 432143214321..43
00000160 32 31 34 33 32 31 34 33 32 31 34 33 32 31 34 33 2143214321432143
00000170 32 31 34 33 32 31 34 33 32 31 34 33 32 31 34 33 2143214321432143
00000180 32 31 34 33 32 31 34 33 32 31 34 33 32 31 0D 0A 21432143214321..
00000190 34 33 32 31 34 33 32 31 34 33 32 31 34 33 32 31 4321432143214321
000001A0 34 33 32 31 34 33 32 31 34 33 32 31 34 33 32 31 4321432143214321
000001B0 34 33 32 31 34 33 32 31 34 33 32 31 34 33 32 31 4321432143214321
000001C0 0D 0A 34 33 32 31 34 33 32 31 34 33 32 31 34 33 ..43214321432143
000001D0 32 31 34 33 32 31 34 33 32 31 34 33 32 31 34 33 2143214321432143
000001E0 32 31 34 33 32 31 34 33 32 31 34 33 32 31 34 33 2143214321432143
000001F0 32 31 0D 0A 34 33 32 31 34 33 32 31 34 33 32 31 21..432143214321
```


16.3.3.3 copysectors

Description

Copies logical sector data to another location.

Syntax

```
copysectors <SrcSector> <DestSector> <NumSectors>
```

Parameters

Parameter	Description
SrcSector	Index of the first source sector.
DestSector	Index of the first destination sector.
NumSectors	Number of sectors to copy.

Additional information

This command operates only on the `nand:0:` volume. The command reports and error if the source and destination sector ranges overlap.

Example

This example copies 10000 logical sectors from sector index 0 to sector index 20000.

```
copysectors 0 20000 10000
```


16.3.4 Volume specific commands

The following sections describe commands that operate on the file system volumes located on the image file.

16.3.4.1 formatlow

Description

Low-level formats a volume.

Syntax

```
formatlow [<VolumeName>]
```

Parameters

Parameter	Description
VolumeName	Name of the volume that has to be formatted.

Additional information

If VolumeName is not specified then the command formats the current volume. The current volume can be selected via the cd command.

Every NAND volume has to be low-level formatted once when the image file is newly created.

Example

This example demonstrates how to format the volume nand:1:

```
formatlow nand:1:
Low-level format...OK
```


16.3.4.2 format

Description

High-level formats a volume.

Syntax

```
format [<VolumeName>]
```

Parameters

Parameter	Description
VolumeName	Name of the volume that has to be formatted.

Additional information

If VolumeName is not specified then the command formats the current first volume. The current volume can be selected via the cd command.

Every NAND volume has to be high-level formatted once when the image file is newly created.

Example

This example demonstrates how to format the volume nand:0:

```
format
High-level format...OK
```


16.3.4.3 df

Description

Shows the available free space on a volume.

Syntax

```
df [<VolumeName>]
```

Parameters

Parameter	Description
VolumeName	Name of the volume to be queried.

Additional information

If VolumeName is not specified than the command calculates the free space of the current volume. The current volume can be selected via the cd command.

Example

This example demonstrates how to get the frees space on the first volume.

```
df
Available free space is 127647744 bytes
```


16.3.4.4 diskinfo

Description

Shows information about a storage device.

Syntax

```
diskinfo [<VolumeName>]
```

Parameters

Parameter	Description
VolumeName	Name of the volume to be queried.

Additional information

If VolumeName is not specified then the command returns information about the current volume. The current volume can be selected via the cd command.

Example

This example demonstrates how to get information about the volume nand:1:.

```
diskinfo nand:1:
    127647744 bytes total disk space
     17719296 bytes available free space
         8192 bytes per cluster
        15582 cluster available on volume
         2163 free cluster available on volume
```


16.3.4.5 getdevinfo

Description

Shows information about a volume.

Syntax

```
getdevinfo [<VolumeName>]
```

Parameters

Parameter	Description
VolumeName	Name of the volume to be queried.

Additional information

If VolumeName is not specified then the command returns information about the first volume. The current volume can be selected via the cd command.

Example

This example demonstrates how to get information about the storage device used by the first volume.

```
getdevinfo
        62370 available sectors
         2048 bytes per sector
```


16.3.4.6 checkdisk

Description

Checks a volume for errors.

Syntax

```
checkdisk [<VolumeName>]
```

Parameters

Parameter	Description
VolumeName	Name of the volume to be checked.

Additional information

If VolumeName is not specified then the command checks the current volume for errors. The current volume can be selected via the cd command.

Example

This example demonstrates how to check the file system structure of the first volume.

```
checkdisk
```


16.3.4.7 creatembr

Description

Writes partition table to master boot record (MBR).

Syntax

```
checkdisk [<VolumeName>]
```

Parameters

Parameter	Description
VolumeName	Name of the volume to be written.

Additional information

If VolumeName is not specified then the command writes the MBR to the current volume. The current volume can be selected via the cd command.

This command stores the partition table configured via addpartition command to the MBR of the specified volume. The command has to be executed after the low-level format and before any high-level format operation. Typically, the partition table is located on nand:0: volume.

Example

This example stores the MBR to the sector 0 of the nand:0: volume.

```
creatembr nand:0:
```


16.3.4.8 listvolumes

Description

Shows the names of all available volumes.

Syntax

```
listvolumes
```

Example

Shows the names of the volumes of in a configuration using five volumes.

```
listvolumes
Available volumes:
  nand:0:
  diskpart:0:
  diskpart:1:
  diskpart:2:
  diskpart:3:
```


Chapter 17

NOR Image Creator

This chapter provides information about an utility for creating NOR images.

17.1 General information

The NOR Image Creator is a command line utility that can be used to create a file containing 1-to-1 representation of the contents of a NOR flash device (image file) with preselected files and directories found on the host PC.

The image file is created by copying files and directories from the host PC to the image file using emFile API functions creating the file structure which is required on the target device.

The image file created using the NOR Image Creator is a standard binary file that can be programmed directly into the NOR flash device of the target via a SEGGER Flasher programmer (https://www.segger.com/products/flash-in-circuit-programmers/) or any other third party utility.

17.2 Using the NOR Image Creator

The NOR Image Creator utility can be used like any other command line based program. For the ease of usage many commands are kept similar to their DOS counterparts. By entering "?" or just pressing the Enter key an overview of the commands is printed on the screen and it should look like the picture below.


NOR Flash Image Creator
NOR Flash Image Creator V1.16a (emFile V4.04d)
(c) 2010-2017 SEGGER Microcontroller GmbH & Co. KG (www.segger.com)
Compiled Jun 14 2017 15:36:16
Press '?' for help
emFile->
Available commands are:
---- Flash image related functions -----
createimage Creates/Opens a new/existing NOR image file.
createimage <FileName> [<DriverType>]
DriverType can be 'sm' for the FS_NOR_Driver or
'bm' for the FS_NOR_BM_Driver. Default is 'sm'.
addsectorblock Adds a sector block to a specific NOR flash image.
addsectorblock <ImageNo>,<NumBlocks>,<BytesPerBlock>
BytesPerBlock must be specified as a hexadecimal value
The other parameters must be specified as a decimal values
setsectorsize Specifies the size of the logical sector.
setsectorsize <BytesPerSector>
The default is 512 bytes.
setlinesize Specifies the size of a NOR flash line.
setsectorsize <BytesPerLine>, default 4 bytes.
setrewritesupport Specifies if a flash line can be rewritten or not.
setrewritesupport <OnOff>
OnOff can be 0 (can not rewrite) or 1 (can rewrite).
The default is 1 (can rewrite).
showimageinfo Shows image related information about a specific image.
showimageinfo <ImageNo>
init Initialize the image file and the file system.
---- Standard functions -----
cd Changes or displays the name of directory.
cd [<VolumeName>][<PathToDir>] or '..'
md Creates a directory (recursively).
md [<VolumeName>]<PathToDir>
rd Removes (deletes) a directory. The directory must be empty.
rd [<VolumeName>]<PathToDir>
addfile/af Copies a file from the host to the image.
addfile [<PathToFileOnHost>]<FileNameOnHost>
[[<VolumeNameOnImage>][PathToFileOnImage]FileNameOnImage]
If no filename is specified the name of the file
on the host will be used.
addfolder Copies a directory recursively from the host to the image.
addfolder [<PathToDirOnHost>]<DirNameOnHost>
[[<VolumeNameOnImage>][PathToDirOnImage]DirNameOnImage]
[/S]
If no folder name is specified as destination the name of
the folder on the host will be used. The option /S
suppresses any information about the progress of
the operation.
type Shows the contents of a file.
type [<VolumeName>][<PathToFile>]<FileName>
del Deletes a file.
del [<VolumeName>][<PathToFile>]<FileName>
dir Lists the contents of a directory.
dir [<VolumeName>][<PathToDir>]
ren Renames a file or directory.
ren [<VolumeName>][<PathToFile>]<FileName> <NewFileName>
move/mv Moves a file or directory.
move [<SrcVolumeName>][<PathToSrcFile>]<SrcFileName>
[<DestVolumeName>][<PathToDestFile>][<DestFileName>]
attrib Shows/Changes the file/directory attributes.
attrib [<VolumeName>][<PathToFile>]<FileName> [+/-][ahrs]
copy Copies a file.
copy [<SrcVolumeName>][<PathToSrcFile>]<SrcFileName>
[<DestVolumeName>][<PathToDestFile>][<DestFileName>]
---- Extended functions -----
formatlow Low-level-formats a volume.
formatlow [<VolumeName>]
format Formats a volume.
format [<VolumeName>]
listvol Shows available volumes.
df Shows the available free space on a volume.
df [<VolumeName>]
diskinfo Shows the disk information of a volume.
diskinfo [<VolumeName>]
getdevinfo Shows the low level disk information of a volume.
getdevinfo [<VolumeName>]
checkdisk Runs checkdisk on a volume.
checkdisk [<VolumeName>]
NOTE: Specifying a filename in command line
will start NOR Flash Image Creator in script mode.
emFile->
emFile U

The NOR Image Creator utility can work in two modes: interactive and script. The interactive mode is entered when the utility is started with no command line arguments. In this mode the user is prompted to type in the commands to be executed. The script mode is entered when the utility is started with a command line argument that specifies the name of a file containing the commands to be executed, one command per line.

17.2.1 Sample script file

he following sample shows a simple script file which does the following thing:

- Creates a NOR flash image file `C:\flash.bin`
- Adds a sector block of 4 Mbytes (64 x 64 kByte) to the image file.
- Creates the image file.
- Low-level formats the volume on the image file.
- High-level formats the volume on the image file.
- Copies the `Test.txt` file from drive `C:` of the host Windows PC to root directory of the volume stored on the image file.
- Copies recursively the contents of `Data` directory from host PC to `Data` directory of the volume stored on the image file.
- Closes the NOR Image Creator utility.

```
createimage C:\flash.bin
addsectorblock 0, 64, 10000
init
formatlow
format
addfile C:\Test.txt
addfolder C:\Data\
q
```


17.3 Supported commands

The image file can be processed by using build it commands of the NAND Image Creator utility. The table below lists all the available commands followed by a detailed description. The optional command parameters are enclosed in square brackets in the syntax description of a command. If not otherwise stated all the numeric parameters have to be specified as decimal format.

Command	Description
Image related commands	
createimage	Creates/opens a new/existing image file.
addsectorblock	Configures a block of NOR physical sectors.
setsectorsize	Configures the logical sector size.
setlinesize	Specifies the minimum number of bytes that can be programmed at once
setrewritesupport	Configures if the NOR driver is allowed to perform rewrite operations.
setbyteorder	Specifies the byte order of the management data.
settimesource	Specifies how the timestamps of added files are generated.
setcrcsupport	Specifies if the data has to be protected by CRC.
showimageinfo	Shows information about the image file.
init	Initializes the image file.
File and directory related commands	
cd	Changes the current working directory.
md	Creates a directory.
rd	Removes a directory.
addfile	Copies a file from the host PC to the image file.
addfolder	Copies a folder including its subfolders and files from the host PC to the image file.
type	Shows the contents of a file.
del	Deletes a file.
dir	Lists the contents of the directory.
ren	Renames a file or directory.
move	Moves a file or directory to another location.
attrib	Shows/changes the file/directory attributes.
copy	Copies a file to another location.
exportfile	Copies a file from image file to host PC.
exportfolder	Copies a folder including its subfolders and files from image file to host PC.
Volume specific commands	
formatlow	Low-level formats a volume.
format	High-level formats a volume.
df	Shows the available free space on a volume.
diskinfo	Shows information about a volume.
getdevinfo	Shows information about a storage device.
checkdisk	Checks a volume for errors.
listvol	Shows the names of all available volumes.

Command	Description
clean	Runs a clean operation on a volume.
createjournal	Creates the journal file on a volume.

17.3.1 Image related commands

The following sections describe the commands required to create an image file.

17.3.1.1 createimage

Description

Creates/opens a new/existing image file.

Syntax

```
createimage <ImageFileName>[,<DriverType>]
```

Parameters

Parameter	Description
ImageFileName	Name of the image file to be created/opened.
DriverType	The type of NAND driver used to access the files on the target. • sm - Sector Map NOR driver (default). • bm - Block Map NOR driver.

Additional information

This command is mandatory and it must be the first command to be executed.

If DriverType is not specified then the Sector Map NOR driver is assumed.

Example

The example shows how to create an image file for the Block Map NOR driver.

```
createimage C:\flash.bin bm
```


17.3.1.2 addsectorblock

Description

Configures a block of NOR physical sectors.

Syntax

```
addsectorblock <ImageNo>,<NumSectors>,<SectorSize>
```

Parameters

Parameter	Description
ImageNo	Index of the image file.
NumSectors	Number of NOR physical sectors in the block.
SectorSize	Size of one NOR physical sector.

Additional information

This command is mandatory and has to be executed at least once. `addsectorblock` can be used to describe the geometry of the physical NOR flash device used on the target. It specifies the characteristics of a continuous block of NOR physical sectors. The command can be executed more that once with different parameters. `SectorSize` has to be specified in bytes as a hexadecimal number without any prefix or suffix. For example a NOR physical sector of 64 Kbytes can be specified as 10000.

`ImageNo` specifies the index of the image file. The first image file created via `createimage` has the index 0, the second image the index 1 and so on.

Example

The following example shows how to configure the image for a NOR flash device that has 32 physical sectors of 64 Kbytes and 8 physical sectors of 4 Kbytes.

```
addsectorblock 0,32,10000
addsectorblock 0,8,1000
```


17.3.1.3 setsectorsize

Description

Configures the logical sector size.

Syntax

```
setsectorsize <SectorSize>
```

Parameters

Parameter	Description
SectorSize	The size of the logical sector in bytes.

Additional information

This command is optional. It can be used to set the size of the logical sector used by the file system. By default the file system uses a logical sector size of 512 bytes. The size of the logical sector as to be smaller than or equal to the size of the logical sector configured on the target application via `FS_SetMaxSectorSize()`.

`SectorSize` has to be a power of 2 value.

Example

This example shows how to set the logical sector size to 512 bytes.

```
setsectorsize 2038
```


17.3.1.4 setlinesize

Description

Specifies the minimum number of bytes that can be programmed at once.

Syntax

```
setlinesize <LineSize>
```

Parameters

Parameter	Description
LineSize	Minimum number of bytes that can be programmed at once.

Additional information

This command is optional. By default, the minimum number of bytes that the NOR driver programs is 4. LineSize has to be a power of 2 value.

Example

This example shows how to set the line size to 16 bytes.

```
setlinesize 16
```


17.3.1.5 setrewritesupport

Description

Configures if the NOR driver is allowed to perform rewrite operations.

Syntax

```
setrewritesupport <OnOff>
```

Parameters

Parameter	Description
`OnOff`	Indicates if the feature is activated or not. • `1` - NOR driver is permitted to rewrite data (default). • `0` - NOR driver is not permitted to rewrite data.

Additional information

This command is optional. By default, the NOR driver is permitted to modify the same byte of the NOR flash device memory as long as bits are changed from 0 to 1 without an erase operation in between. The majority of NOR flash devices support this feature but some are not. For this type of NOR flash devices this command has to be called with `OnOff` set to 0.

Example

This example shows how to specify that the NOR flash can rewrite data.

```
setrewritesupport 1
```


17.3.1.6 setbyteorder

Description

Specifies the byte order of the management data.

Syntax

```
setbyteorder <FormatType>
```

Parameters

Parameter	Description
FormatType	Specifies the byte ordering. • le - little endian format. • be - big endian format.

Additional information

This command is optional. By default, the NOR driver stores multi-byte management data in the byte order of the host CPU which is typically little-endian. The byte order of the management data has to match the byte order of the target CPU. `setbyteorder` can be used to explicitly configure the byte order.

Example

This example shows how to configure the byte order to big-endian.

```
setbyteorder be
```


17.3.1.7 settimesource

Description

Specifies how the timestamps of added files are generated.

Syntax

```
settimesource <TimeSource>
```

Parameters

Parameter	Description
TimeSource	Specifies the time source to be used. • sys - time of host PC (default). • none - constant value.

Additional information

This command is optional. By default, the NOR Image Creator uses the current time and date of the host PC to generate the timestamps of the files and directories. In some cases it is desirable to have the same constant time stamp for all the files and directories stored on the image file so that for example the generated image file can be compared correctness against a reference image file.

Example

This example shows how to configure the NOR Image Creator to use a constant time stamp for all the files and directories it creates on the image file.

```
settimesource none
```


17.3.1.8 setcrcsupport

Description

Specifies if the data has to be protected by CRC.

Syntax

```
setcrcsupport <OnOff>
```

Parameters

Parameter	Description
OnOff	Indicates if the feature is activated or not. • 1 - Data is protected by CRC. • 0 - Data is not protected by CRC (default).

Additional information

This command is optional. By default, the management and user data stored by the NOR driver is not protected by CRC.

Example

This example shows how to configure the NOR Image Creator to protect the data via CRC.

```
setcrcsupport 1
```


17.3.1.9 showimageinfo

Description

Shows information about the image file.

Syntax

```
showimageinfo <ImageNo>
```

Parameters

Parameter	Description
ImageNo	Index of the image file.

Example

```
showimageinfo
Information about image 0:
  Name:             NORImageCreator_NOR.img
  ImageSize:        8388608 bytes
  NumSectorBlocks:  1
    Block 0: 128 sectors of 0x00010000 bytes
```


17.3.1.10 init

Description

Initializes the image file.

Syntax

```
init
```

Additional information

This command must be executed before any other command which operates on files and directories.

Example

```
init
```


17.3.2 File and directory related commands

The following sections describe the commands that can be used to operate on the files and directories stored on the image file. The commands described in this section can be executed only after the execution of the `init` command.

17.3.2.1 cd

Description

Changes the current working directory.

Syntax

```
cd <DirPathOnImage>
```

Parameters

Parameter	Description
DirPathOnImage	Path to new directory to change to.

Additional information

If DirPathOnImage is set to .. the command changes to the parent directory. After initialization the current directory is set to the root directory.

Example

In the following sample the working directory is changed to Test and then back to the original directory.

```
cd Test
cd ..
```


17.3.2.2 md

Description

Creates a directory.

Syntax

```
md <DirPathOnImage>
```

Parameters

Parameter	Description
DirPathOnImage	Path to directory to be created.

Additional information

This command is able to create any missing directories in the path.

Example

```
md Test\Dir\
```


17.3.2.3 rd

Description

Removes a directory.

Syntax

```
rd <DirPathOnImage>
```

Parameters

Parameter	Description
DirPathOnImage	Path to directory to be deleted.

Additional information

The directory to be deleted must be empty.

Example

```
rd Test
```


17.3.2.4 addfile

Description

Copies a file from the host PC to the image file.

Syntax

```
addfile <FilePathOnHost> [<FilePathOnImage>]
```

Parameters

Parameter	Description
FilePathOnHost	Path to the source file on host PC.
FilePathOnImage	Path to destination file on the image.

Additional information

If FilePathOnImage is not specified then the name of the source file is used.

Example

The following command copies the contents of the Test.txt file from host PC to Dest.txt file on image.

```
addfile C:\Test.txt Dest.txt
```

The following command copies the contents of Test.txt file to the file with the same name of the image.

```
addfile C:\Test.txt
```


17.3.2.5 addfolder

Description

Copies a folder including its subfolders and files from the host PC to the image file.

Syntax

```
addfolder <DirPathOnHost> [<DirPathOnImage>]
```

Parameters

Parameter	Description
DirPathOnHost	Path to source directory on the host PC.
DirPathOnImage	Path to destination directory on the image.

Additional information

If DirPathOnImage is not specified then the name of the source directory is used.

Example

The following command copies the contents of the Data directory on PC to Dest directory on the image.

```
addfolder C:\Data Dest
```

The following command copies the contents of the Data directory on PC to a directory with the same name on the image.

```
addfolder C:\Data
```


17.3.2.6 type

Description

Shows the contents of a file.

Syntax

```
type <FilePathOnImage>
```

Parameters

Parameter	Description
FilePathOnImage	Path to the file on image to be shown.

Example

The following command line shows the contents of the file `Test.txt` located on the root directory of the image.

```
type Test.txt
```


17.3.2.7 del

Description

Deletes a file.

Syntax

```
del <FilePathOnImage>
```

Parameters

Parameter	Description
FilePathOnImage	Path to the file on image to be deleted.

Example

The following command line deletes the file `Test.txt` located on the root directory of the image.

```
del Test.txt
```


17.3.2.8 dir

Description

Lists the contents of the directory.

Syntax

```
dir [<DirPathOnImage>]
```

Parameters

Parameter	Description
DirPathOnImage	Path to directory on image to be listed.

Additional information

If DirPathOnImage is not specified then the command lists the contents of the current working directory.

Example

The following command line shows the contents of the current directory.

```
dir
TestDir              (Dir) Attributes: ----
.                    (Dir) Attributes: ----
..                   (Dir) Attributes: ----
SubDir               (Dir) Attributes: ----
Test1.txt                  Attributes: A--- Size: 30
Test2.txt                  Attributes: A--- Size: 31
```


17.3.2.9 ren

Description

Renames a file or directory.

Syntax

```
ren <FilePathOnImage> <NewFileName>
```

Parameters

Parameter	Description
FilePathOnImage	Path to file or directory that has to be renamed.
NewFileName	New name of the file or directory.

Additional information

If DirPathOnImage is not specified then the command lists the contents of the current working directory.

Example

The following command line changes the name of Test.txt file located in the Test directory to Test2.txt.

```
ren Test\Test.txt Test2.txt
```


17.3.2.10 move

Description

Moves a file or directory to another location.

Syntax

```
move <SrcFilePathOnImage> <DestFilePathOnImage>
```

Parameters

Parameter	Description
SrcFilePathOnImage	Path to file or directory to be moved.
DestFilePathOnImage	Path to destination file or directory on the image.

Example

The following command line moves the `Test.txt` file located in the `Test` directory to the root directory. The file is renamed to `New.txt`.

```
move Test\Test.txt New.txt
```


17.3.2.11 attrib

Description

Shows/changes the file/directory attributes.

Syntax

```
attrib <PathOnImage> [<Operation> <Attributes>]
```

Parameters

Parameter	Description
PathOnImage	Path to file or directory.
Operation	Operation to be executed on attributes. Permitted values are: • + sets the attributes. • - clears the attributes.
Attributes	The list of attributes to be changed The following attributes are supported: • a archive. • h hidden. • r read-only. • s system.

Example

The following example the clears the archive and read-only attributes of the Test.txt file.

```
attr Test.txt -ar
```


17.3.2.12 copy

Description

Copies a file to another location.

Syntax

```
copy <SrcFilePathOnImage> <DestFilePathOnImage>
```

Parameters

Parameter	Description
SrcFilePathOnImage	Path to the file to be copied.
DestFilePathOnImage	Path to destination the file.

Example

The following example copies the contents of the `Test.txt` to the `Test2.txt` file.

```
copy Test.txt Test2.txt
```


17.3.2.13 exportfile

Description

Copies a file from image file to host PC.

Syntax

```
exportfile <FilePathOnImage> <FilePathOnHost>
```

Parameters

Parameter	Description
FilePathOnImage	Path to the source file on image.
FilePathOnHost	Path to the destination file on PC.

Example

The following command copies the contents of the `Test.txt` file from image to `Dest.txt` file on host PC.

```
exportfile Test.txt C:\Dest.txt
```


17.3.2.14 exportfolder

Description

Copies a folder including its subfolders and files from image file to host PC.

Syntax

```
exportfolder <DirPathOnImage> <DirPathOnHost>
```

Parameters

Parameter	Description
DirPathOnImage	Path to the source directory on image.
DirPathOnHost	Path to the destination directory on PC.

Additional information

The command copies the contents of a folder including all the folders and the files that are stored in it. An identical copy of the specified directory on the image file is created on the specified folder of the host PC.

Example

The following command copies the contents of the `Data` directory on the image file to `C:\Dest` directory on the host PC.

```
addfolder Data C:\Dest
```


17.3.3 Volume specific commands

The following sections describe commands that operate on the file system volumes located on the image file.

17.3.3.1 formatlow

Description

Low-level formats a volume.

Syntax

```
formatlow [<VolumeName>]
```

Parameters

Parameter	Description
VolumeName	Name of the volume that has to be formatted.

Additional information

If VolumeName is not specified then the command formats the current volume. The current volume can be selected via the cd command.

Every NOR volume has to be low-level formatted once when the image file is newly created.

Example

This example demonstrates how to format the volume nor:1:

```
formatlow nor:1:
Low-level format...OK
```


17.3.3.2 format

Description

High-level formats a volume.

Syntax

```
format [<VolumeName>]
```

Parameters

Parameter	Description
VolumeName	Name of the volume that has to be formatted.

Additional information

If VolumeName is not specified then the command formats the current first volume. The current volume can be selected via the cd command.

Every NOR volume has to be high-level formatted once when the image file is newly created.

Example

This example demonstrates how to format the volume nor:0:

```
format
High-level format...OK
```


17.3.3.3 df

Description

Shows the available free space on a volume.

Syntax

```
df [<VolumeName>]
```

Parameters

Parameter	Description
VolumeName	Name of the volume to be queried.

Additional information

If VolumeName is not specified than the command calculates the free space of the current volume. The current volume can be selected via the cd command.

Example

This example demonstrates how to get the frees space on the first volume.

```
df
Available free space is 127647744 bytes
```


17.3.3.4 diskinfo

Description

Shows information about a storage device.

Syntax

```
diskinfo [<VolumeName>]
```

Parameters

Parameter	Description
VolumeName	Name of the volume to be queried.

Additional information

If VolumeName is not specified then the command returns information about the current volume. The current volume can be selected via the cd command.

Example

This example demonstrates how to get information about the volume nor:1:.

```
diskinfo nor:1:
      7333888 bytes total disk space
      7333888 bytes available free space

         1024 bytes per cluster
         7162 cluster available on volume
         7162 free cluster available on volume
```


17.3.3.5 getdevinfo

Description

Shows information about a volume.

Syntax

```
getdevinfo [<VolumeName>]
```

Parameters

Parameter	Description
VolumeName	Name of the volume to be queried.

Additional information

If VolumeName is not specified then the command returns information about the first volume. The current volume can be selected via the cd command.

Example

This example demonstrates how to get information about the storage device used by the first volume.

```
getdevinfo
        14399 Available sectors
          512 bytes per sector

            0 sectors per track
            0 number of heads
```


17.3.3.6 checkdisk

Description

Checks a volume for errors.

Syntax

```
checkdisk [<VolumeName>]
```

Parameters

Parameter	Description
VolumeName	Name of the volume to be checked.

Additional information

If VolumeName is not specified then the command checks the current volume for errors. The current volume can be selected via the cd command.

Example

This example demonstrates how to check the file system structure of the first volume.

```
checkdisk
```


17.3.3.7 listvol

Description

Shows the names of all available volumes.

Syntax

```
listvol
```

Example

Shows the names of the volumes of in a configuration using five volumes.

```
listvol
Available volumes:
  nor:0:
```


17.3.3.8 clean

Description

Runs a clean operation on a volume.

Syntax

```
clean [<VolumeName>]
```

Parameters

Parameter	Description
VolumeName	Name of the volume to be cleaned.

Additional information

If VolumeName is not specified then the command cleans the current volume The current volume can be selected via the cd command.

Example

This example demonstrates how to clean the volume nor:1:.

```
clean nor:1:
```


17.3.3.9 createjournal

Description

Creates the journal file on a volume.

Syntax

```
createjournal [<VolumeName>][<PathToFile>]<FileName>
               <JournalSize> [<SupportFreeSector>]
```

Parameters

Parameter	Description
VolumeName	Name of the volume that stores the journal file.
PathToFile	Path to the journal file.
FileName	Name of the journal file.
JournalSize	Size of the journal file.
SupportFreeSector	Size of the journal file.

Additional information

If VolumeName is not specified then the command crates the journal on the current volume. The current volume can be selected via the cd command. If PathToFile is not specified then the journal file is created on the root directory of the specified volume. JournalSize has to be specified in bytes. The value of SupportFreeSector is passed as third argument to FS_JOURNAL_CreateEx(). If not specified SupportFreeSector is set to 0.

The parameters passed to the createjournal command have to match the parameters passed to FS_JOURNAL_CreateEx() on the target application.

Example

This example demonstrates how to create a journal file of 128 Kbytes on the current volume.

```
createjournal Journal.dat 131072 1
Create journal file 'Journal.dat'...OK
```


Chapter 18

Porting emFile 2.x to 3.x

This chapter describes the differences between the versions 2.x and 3.x of emFile and the changes required to port an existing application.

18.1 Differences between version 2.x and 3.x

Most of the differences between 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 were completely removed. Refer to section *API differences* on page 1170 for a complete overview of the removed and obsolete functions.

emFile version 3.x has a new driver handling. You can include device drivers and allocate the required memory for it without the need to recompile the entire file system. Refer to *Configuration of emFile* on page 927 for detailed information about the integration of a driver into emFile. For detailed information about the emFile device drivers, refer to the chapter *Device drivers* on page 320.

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 45 for detailed information about the best way to start to work with emFile version 3.x. The following sections give an overview about the changes introduced in the version 3.x of emFile.

18.2 API differences

Function	Description
Changed functions	
FS_GetFreeSpace()	Number of parameters reduced. Parameter DevIndex removed.
FS_GetTotalSpace()	
Removed functions	
FS_CloseDir()	The directory handling has been changed in emFile version 3.x. The functions should be replaced. Refer to FS_FindClose() 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.

In emFile version 3.x, the header file FS_Api.h is renamed to FS.h , therefore change the name of the file system header file in your application.

18.3 Configuration differences

The configuration of emFile version 3.x has been simplified compared to emFile version 2.x. emFile version 3.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 most applications. A lot of the compile time flags of emFile version 2.x were 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() for more information.

Chapter 19

Porting emFile 3.x to 4.x

This chapter describes the differences between the versions 3.x and 4.x of emFile and the changes required to port an application to the new version.

19.1 Differences between version 3.x and 4.x

Most of the differences between emFile version 3.x and version 4.x are internal. The API of emFile version 3.x is a subset of the version 4.x. There are two differences that require the porting of an application using emFile:

- The API of the hardware layer has been improved.
- The SD/MMC device driver for Atmel MCUs has been removed.

emFile version 4.x has a new hardware layer API. The name of the functions are no longer hard-coded in the source. Instead, the drivers call the functions indirectly via a function table. The hardware layer is specified as an instance of a structure that contains pointers to functions. This gives greater flexibility to the application allowing it to configure different hardware layers at runtime depending on the target hardware it runs on. The emFile configurations that do not require a hardware layer are not affected by this change.

The SD/MMC device driver for Atmel MCUs (`FS_MMC_CM_Driver4Atmel`) has been removed from the version 4.x of emFile. The reason for this change is to eliminate duplicated functionality. The SD/MMC card mode device driver (`FS_MMC_CardMode_Driver`) supports all the functionality of the SD/MMC device driver for Atmel MCUs and can be used as replacement.

The following sections give detailed information about the changes introduced in the version 4.x of emFile that require changes in the application.

19.2 Hardware layer API differences

The function prototypes of the new hardware layer API of version 4.x are identical to the hardware layer functions used in the version 3.x of emFile. No functions have been added to or removed from the new hardware layer API and no changes are required to the implementation of the existing hardware layer functions. Typically, the following changes are required for porting an application to version 4.x:

- A function table containing pointers to the hardware layer functions has to be declared as a global variable in the file that implements the old hardware layer. The type of the function table depends on the hardware layer used.
- A header file has to be created that declares the function table as external.
- The header file has to be included in the file that defines the `FS_X_AddDevices()` function.
- A call to one of the functions that configure the type of the hardware layer has to be added to `FS_X_AddDevices()`. These functions take as second argument a pointer to the function table.

Alternatively, the steps 1 to 3 can be skipped and a default hardware layer can be used instead. The function table of these hardware layers contain pointers to functions used in the version 3.x of emFile. More information can be found in the following sections that describe the differences for each hardware layer type.

19.2.1 NAND device driver differences

Sample hardware layers using the new API and the corresponding file system configuration files can be found in the `Sample/FS/Driver/NAND` folder of the emFile shipment and are located in the following files:

Hardware layer for parallel NAND flash via memory controller

- `FS_NAND_HW_Template.h`
- `FS_NAND_HW_Template.c`
- `FS_ConfigNAND_Template.c`

Hardware layer for parallel NAND flash via I/O port

- `FS_NAND_HW_TemplatePort.h`
- `FS_NAND_HW_TemplatePort.c`
- `FS_ConfigNAND_TemplatePort.c`

Hardware layer for DataFlash via SPI

- `FS_NAND_HW_DF_Template.h`
- `FS_NAND_HW_DF_Template.c`
- `FS_ConfigNAND_DF_Template.c`

Hardware layer for serial NAND flash via SPI

- `FS_NAND_HW_SPI_Template.h`
- `FS_NAND_HW_SPI_Template.c`
- `FS_ConfigNAND_UNI_SPI_Template.c`

19.2.1.1 Porting without hardware layer modification

The porting of the application to the new hardware layer API can be done without modifying the existing hardware layer. emFile comes with default hardware layers that contain pointers to the hardware layer functions used in the version 3.x of emFile. A call to the function that configures the hardware layer type has to be added to file system configuration and the file that defines the default hardware layer has to be added to the project. The following table shows in which file each default hardware layer is defined. All the files are located in the `Sample/FS/Driver/NAND` folder of the emFile shipment.

Default hardware layer	File name
`FS_NAND_HW_Default`	`FS_NAND_HW_Default.c`
`FS_NAND_HW_DF_Default`	`FS_NAND_HW_DF_Default.c`
`FS_NAND_HW_SPI_Default`	`FS_NAND_HW_SPI_Default.c`

The table below lists the default hardware layer required by each physical layer:

Physical layer	Default hardware layer
`FS_NAND_PHY_x`	`FS_NAND_HW_Default`
`FS_NAND_PHY_x8`	`FS_NAND_HW_Default`
`FS_NAND_PHY_512x8`	`FS_NAND_HW_Default`
`FS_NAND_PHY_2048x8`	`FS_NAND_HW_Default`
`FS_NAND_PHY_2048x16`	`FS_NAND_HW_Default`
`FS_NAND_PHY_4096x8`	`FS_NAND_HW_Default`
`FS_NAND_PHY_ONFI`	`FS_NAND_HW_Default`
`FS_NAND_PHY_DataFlash`	`FS_NAND_HW_DF_Default`
`FS_NAND_PHY_SPI`	`FS_NAND_HW_SPI_Default`

For more information about the functions of the physical layers refer to *NAND physical layer* on page 423.

Example

The following example shows how to port an existing configuration using the SLC1 NAND device driver. Two changes are required:

- The function call marked with the comment "Add this line" in the code snippet below has to be added to the existing emFile configuration.
- The file `Sample/FS/Driver/NAND/FS_NAND_HW_Default.c` that defines the `FS_NAND_HW_Default` hardware layer has to be added to the project.

```
void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // The next line is optional.
  //
  // FS_SetMaxSectorSize(2048);
  //
  // Add and configure the driver.
  //
  FS_AddDevice(&FS_NAND_Driver);
  FS_NAND_SetPhyType(0, &FS_NAND_PHY_2048x8);
  FS_NAND_2048x8_SetHWType(0, &FS_NAND_HW_Default);   // Add this line.
  //
  // Enable the file buffer to increase the performance
  // when reading/writing a small number of bytes.
  //
  FS_ConfigFileBufferDefault(512, FS_FILE_BUFFER_WRITE);
}
```

19.2.1.2 Replacement functions

The following table lists the replacement functions of the hardware layer for parallel NAND flash.

Old function	Member of FS_NAND_HW_TYPE structure
FS_NAND_HW_X_Init_x8()	(*pfInit_x8)()
FS_NAND_HW_X_Init_x16()	(*pfInit_x16)()
FS_NAND_HW_X_DisableCE()	(*pfDisableCE)()
FS_NAND_HW_X_EnableCE()	(*pfEnableCE)()
FS_NAND_HW_X_SetAddrMode()	(*pfSetAddrMode)()
FS_NAND_HW_X_SetCmdMode()	(*pfSetCmdMode)()
FS_NAND_HW_X_SetDataMode()	(*pfSetDataMode)()
FS_NAND_HW_X_WaitWhileBusy()	(*pfWaitWhileBusy)()
FS_NAND_HW_X_Read_x8()	(*pfRead_x8)()
FS_NAND_HW_X_Write_x8()	(*pfWrite_x8)()
FS_NAND_HW_X_Read_x16()	(*pfRead_x16)()
FS_NAND_HW_X_Write_x16()	(*pfWrite_x16)()

The following table lists the replacement functions of the hardware layer for DataFlash.

Old function	Member of FS_NAND_HW_TYPE_DF structure
FS_DF_HW_X_Init()	(*pfInit)()
FS_DF_HW_X_EnableCS()	(*pfEnableCS)()
FS_DF_HW_X_DisableCS()	(*pfDisableCS)()
FS_DF_HW_X_Read()	(*pfRead)()
FS_DF_HW_X_Write()	(*pfWrite)()

The following table lists the replacement functions of the hardware layer for serial NAND flash.

Old function	Member of FS_NAND_HW_TYPE_SPI structure
FS_NAND_HW_SPI_X_Init()	(*pfInit)()
FS_NAND_HW_SPI_X_DisableCS()	(*pfDisableCS)()
FS_NAND_HW_SPI_X_EnableCS()	(*pfEnableCS)()
FS_NAND_HW_SPI_X_Delay()	(*pfDelay)()
FS_NAND_HW_SPI_X_Read()	(*pfRead)()
FS_NAND_HW_SPI_X_Write()	(*pfWrite)()

19.2.2 NOR device driver differences

Sample hardware layers using the new API and the corresponding file system configurations can be found in the Sample/FS/Driver/NOR folder of the emFile shipment and are located in the following files:

Hardware layer for serial NOR flash

- FS_NOR_HW_SPI_Template.h
- FS_NOR_HW_SPI_Template.c
- FS_ConfigNOR_SPI_Template.c

Hardware layer for QSPI NOR flash

- FS_NOR_HW_SPIFI_Template.h
- FS_NOR_HW_SPIFI_Template.c
- FS_ConfigNOR_BM_SPIFI_Template.c

19.2.2.1 Porting without hardware layer modification

The porting of the application to the new hardware layer API can be done without modifying the existing hardware layer. emFile comes with default hardware layers that contain pointers to the hardware layer functions used in the version 3.x of emFile. A call to the function that sets the hardware layer type has to be added to file system configuration and the file that defines the default hardware layer has to be added to the project. The table below shows in which file each default hardware layer is defined. The files are located in the Sample/FS/Driver/NOR folder of the emFile shipment:

Default hardware layer	File name
FS_NOR_HW_ST_M25_Default	FS_NOR_HW_ST_M25_Default.c
FS_NOR_HW_SFDP_Default	FS_NOR_HW_SFDP_Default.c

The table below lists the default hardware layer required by each physical layer:

Physical layer	Default hardware layer
FS_NOR_PHY_ST_M25	FS_NOR_HW_ST_M25_Default
FS_NOR_PHY_SFDP	FS_NOR_HW_SFDP_Default

For more information about the functions of the physical layers refer to *NOR physical layer* on page 640.

Example

This example show how to port an existing configuration for a serial NOR flash. The function call marked with the comment “Add this line” to be added to existing configuration. No other changes of file system configuration are required. Additionally, the file Sample/FS/Driver/NOR/FS_NOR_HW_ST_M25_Default.c that defines the FS_NOR_HW_ST_M25_Default hardware layer has to be added to the project.

```
void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add and configure the driver.
  //
  FS_AddDevice(&FS_NOR_Driver);
  FS_NOR_SetPhyType(0, &FS_NOR_PHY_ST_M25);
  FS_NOR_SPI_SetHWType(0, &FS_NOR_HW_ST_M25_Default);  // Add this line.
  FS_NOR_Configure(0, FLASH0_BASE_ADDR, FLASH0_START_ADDR, FLASH0_SIZE);
  //
  // Enable the file buffer to increase the performance
  // when reading/writing a small number of bytes.
  //
  FS_ConfigFileBufferDefault(512, FS_FILE_BUFFER_WRITE);
```



```
}
```

19.2.2.2 Replacement functions

The following table lists the replacement functions of the SPI hardware layer.

Old function	Member of the FS_NOR_HW_TYPE_SPI structure
FS_NOR_SPI_HW_X_Init()	(*pfInit)()
FS_NOR_SPI_HW_X_EnableCS()	(*pfEnableCS)()
FS_NOR_SPI_HW_X_DisableCS()	(*pfDisableCS)()
FS_NOR_SPI_HW_X_Read()	(*pfRead)()
FS_NOR_SPI_HW_X_Write()	(*pfWrite)()
FS_NOR_SPI_HW_X_Read_x2()	(*pfRead_x2)()
FS_NOR_SPI_HW_X_Write_x2()	(*pfWrite_x2)()
FS_NOR_SPI_HW_X_Read_x4()	(*pfRead_x4)()
FS_NOR_SPI_HW_X_Write_x4()	(*pfWrite_x4)()

Note

The …_x2 and …_x4 functions have to be defined only when the FS_NOR_PHY_SFDP physical layer configured and the hardware supports dual and quad SPI data transfer modes.

19.2.3 MMC/SD SPI device driver differences

Sample hardware layers using the new API and the corresponding file system configurations can be found in the Sample/FS/Driver/MMC_SPI folder of the emFile shipment and are located in the following files:

Hardware layer for SD card via SPI

- FS_MMC_HW_SPI_Template.h
- FS_MMC_HW_SPI_Template.c
- FS_ConfigMMC_SPI_Template.c

Hardware layer for SD card via I/O port

- FS_MMC_HW_SPI_TemplatePort.h
- FS_MMC_HW_SPI_TemplatePort.c
- FS_ConfigMMC_SPI_TemplatePort.c

19.2.3.1 Porting without hardware layer modification

It is possible to port an existing application to the new hardware layer API without modifying the existing hardware layer. emFile comes with a default hardware layer that contains pointers to the hardware layer functions used in the version 3.x of emFile. A call to the function FS_MMC_SetHWType() has to be added to file system configuration and the file that defines the default hardware layer (FS_MMC_HW_SPI_Default.c) has to be added to the project. The file is located in the Sample/FS/Driver/MMC_SPI folder of the emFile shipment.

Example

This example shows how to modify an existing emFile configuration to work with the new hardware layer API. The function call marked marked with the comment "Add this line." has to be added to existing configuration. No other changes to file system configuration are required. In addition, the file Sample/FS/Driver/MMC_SPI/FS_MMC_HW_SPI_Default.c that defines the FS_MMC_HW_SPI_Default hardware layer has to be added to the project.

```
void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add and configure the driver.
  //
  FS_AddDevice(&FS_MMC_SPI_Driver);
  FS_MMC_SetHWType(0, &FS_MMC_HW_SPI_Default);    // Add this line.
//  FS_MMC_ActivateCRC();    // Uncommenting this line will activate
                             // the CRC calculation of the MMC/SD SPI driver.
  //
  // Enable the file buffer to increase the performance
  // when reading/writing a small number of bytes.
  //
  FS_ConfigFileBufferDefault(512, FS_FILE_BUFFER_WRITE);
}
```

19.2.3.2 Replacement functions

The following table lists the replacement functions of the SPI hardware layer.

Old function	Member of FS_MMC_HW_TYPE_SPI structure
FS_MMC_HW_X_EnableCS()	(*pfEnableCS)()
FS_MMC_HW_X_DisableCS()	(*pfDisableCS)()
FS_MMC_HW_X_IsPresent()	(*pfIsPresent)()
FS_MMC_HW_X_IsWriteProtect-ed()	(*pfIsWriteProtected)()
FS_MMC_HW_X_SetMaxSpeed()	(*pfSetMaxSpeed)()

Old function	Member of FS_MMC_HW_TYPE_SPI structure
FS_MMC_HW_X_SetVoltage()	(*pfSetVoltage)()
FS_MMC_HW_X_Read()	(*pfRead)()
FS_MMC_HW_X_Write()	(*pfWrite)()
FS_MMC_HW_X_Read()	(*pfReadEx)()
FS_MMC_HW_X_Write()	(*pfWriteEx)()
FS_MMC_HW_X_Lock()	(*pfLock)()
FS_MMC_HW_X_Unlock()	(*pfUnlock)()

Note

The locking functions have to be implemented only when the MMC / SD SPI device driver is compiled with the FS_MMC_SUPPORT_LOCKING define set to 1.

19.2.4 MMC/SD card mode device driver differences

Sample hardware layers using the new API and the corresponding file system configurations are available in the Sample/FS/Driver/MMC_CM folder of the emFile shipment and are located in the following files:

- FS_MMC_HW_CM_Template.h
- FS_MMC_HW_CM_Template.c
- FS_ConfigMMC_CardModeTemplate.c

19.2.4.1 Porting without hardware layer modification

It is possible to port the application to the new hardware layer API without modifying the existing hardware layer. emFile comes with a default hardware layer that contains pointers to the hardware layer functions used in the version 3.x of emFile. A call to the function FS_MMC_CM_SetHWType() has to be added to file system configuration and the file that defines the default hardware layer (FS_MMC_HW_CM_Default.c) has to be added to the project. The file is located in the Sample/FS/Driver/MMC_CM folder of the emFile shipment.

Example

This example shows the modifications that have to be done to an existing application. The function call marked with the comment "Add this line." has to be added to existing configuration. No other changes of file system configuration are required. Additionally, the file Sample/FS/Driver/MMC_CM/FS_MMC_HW_CM_Default.c that defines the FS_MMC_HW_CM_Default hardware layer has to be added to the project.

```
void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  //
  // Add and configure the driver.
  //
  FS_AddDevice(&FS_MMC_CardMode_Driver);
  FS_MMC_CM_Allow4bitMode(0, 1);
  FS_MMC_CM_SetHWType(0, &FS_MMC_HW_CM_Default);      // Add this line.
  //
  // Configure the file system for fast write operations.
  //
  FS_ConfigFileBufferDefault(512, FS_FILE_BUFFER_WRITE);
  FS_SetFileWriteMode(FS_WRITEMODE_FAST);
}
```

19.2.4.2 Replacement functions

The following table lists the replacement functions of the SD / MMC card mode hardware layer.

Old function	Member of FS_MMC_HW_TYPE_CM structure
FS_MMC_HW_X_InitHW()	(*pfInitHW)()
FS_MMC_HW_X_Delay()	(*pfDelay)()
FS_MMC_HW_X_IsPresent()	(*pfIsWriteProtected)()
FS_MMC_HW_X_IsWriteProtected()	(*pfIsPresent)()
FS_MMC_HW_X_SetMaxSpeed()	(*pfSetMaxSpeed)()
FS_MMC_HW_X_SetResponseTimeOut()	(*pfSetResponseTimeOut)()
FS_MMC_HW_X_SetReadDataTimeOut()	(*pfSetReadDataTimeOut)()
FS_MMC_HW_X_SendCmd()	(*pfSendCmd)()
FS_MMC_HW_X_GetResponse()	(*pfGetResponse)()

Old function	Member of FS_MMC_HW_TYPE_CM structure
FS_MMC_HW_X_ReadData()	(*pfReadData)()
FS_MMC_HW_X_WriteData()	(*pfWriteData)()
FS_MMC_HW_X_SetDataPointer()	(*pfSetDataPointer)()
FS_MMC_HW_X_SetHWBlockLen()	(*pfSetHWBlockLen)()
FS_MMC_HW_X_SetHWNumBlocks()	(*pfSetHWNumBlocks)()
FS_MMC_HW_X_GetMaxRead-Burst()	(*pfGetMaxReadBurst)()
FS_MMC_HW_X_GetMaxWrite-Burst()	(*pfGetMaxWriteBurst)()

19.2.5 CompactFlash / IDE device driver differences

Sample hardware layers using the new API and the corresponding file system configurations can be found in the Sample/FS/Driver/IDE folder of the emFile shipment in the following files:

Hardware layer for memory map mode

- FS_IDE_HW_TemplateMemMapped16bit.h
- FS_IDE_HW_TemplateMemMapped16Bit.c
- FS_ConfigIDE_TemplateMemMapped16Bit.c

Hardware layer for true IDE mode

- FS_IDE_HW_TemplateTrueIDE.h
- FS_IDE_HW_TemplateTrueIDE.c
- FS_ConfigIDE_TemplateTrueIDE.c

19.2.5.1 Porting without hardware layer modification

It is possible to port the application to the new hardware layer API without modifying the existing hardware layer. emFile comes with a default hardware layer that contains pointers to the hardware layer functions used in the version 3.x of emFile. A call to the function FS_IDE_SetHWType() has to be added to file system configuration and the file that defines the default hardware layer (FS_IDE_HW_Default.c) has to be added to the project. The file is located in the Sample/FS/Driver/IDE folder of the emFile shipment.

Example

This example shows the modifications that have to be done to an existing application. The function call marked mark with the comment "Add this line." has to be added to existing configuration. No other changes of file system configuration are required. In addition, the file Sample/FS/Driver/IDE/FS_IDE_HW_Default.c which defines the FS_IDE_HW_Default hardware layer has to be added to the project.

```
void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  FS_AddDevice(&FS_IDE_Driver);
  FS_IDE_SetHWType(0, &FS_IDE_HW_Default);    // Add this line.
  //
  // Enable the file buffer to increase the performance
  // when reading/writing a small number of bytes.
  //
  FS_ConfigFileBufferDefault(512, FS_FILE_BUFFER_WRITE);
}
```

19.2.5.2 Replacement functions

The following table lists the replacement functions of the hardware layer.

Old function	Member of FS_IDE_HW_TYPE structure
FS_IDE_HW_Reset()	(*pfReset)()
FS_IDE_HW_IsPresent()	(*pfIsPresent)()
FS_IDE_HW_Delay400ns()	(*pfDelay400ns)()
FS_IDE_HW_ReadReg()	(*pfReadReg)()
FS_IDE_HW_WriteReg()	(*pfWriteReg)()
FS_IDE_HW_ReadData()	(*pfReadData)()
FS_IDE_HW_WriteData()	(*pfWriteData)()

19.2.6 MMC / SD device driver for ATMEL MCUs

The MMC / SD device driver for ATMEL microcontrollers (FS_MMC_CM_Driver4Atmel) is no longer available in the version 4.x of emFile. The MMC/SD card mode device driver (FS_MMC_CardMode_Driver) provides the same functionality and can be used as replacement. Changes are required both to emFile configuration and to hardware layer in oder to port an existing application to the version 4.x of emFile.

19.2.6.1 Configuration changes

The MMC / SD device driver for ATMEL MCUs has to be replaced in the FS_X_AddDevices() function by the MMC/SD card mode device driver. In addition, the hardware layer has to be explicitly configured.

Example

This example shows what changes have to be done to emFile configuration. The function calls that marked with the comment "Remove this line." have to be removed from the file system configuration. The function calls that have to be added are marked with the commend "Add this line."

```
void FS_X_AddDevices(void) {
  FS_AssignMemory(&_aMemBlock[0], sizeof(_aMemBlock));
  FS_AddDevice(&FS_MMC_CM_Driver4Atmel);            // Remove this line.
  FS_AddDevice(&FS_MMC_CardMode_Driver);            // Add this line.
  FS_MMC_CM_Allow4bitMode(0, 1);                    // Add this line.
  FS_MMC_CM_SetHWType(0, &FS_MMC_CM_HW_Default);    // Add this line.
}
```

19.2.6.2 Hardware layer changes

The hardware layer APIs of the MMC / SD device driver for ATMEL MCUS and that of the MMC / SD card mode device driver are different. A new hardware layer has to be implemented for the MMC/SD card mode device driver to port an existing application. For more information about the functions of the hardware layer for MMC / SD card mode device driver refer to *Hardware layer* on page 321. Sample implementations for ATMEL AT91SAM9x MCU can be found in the Sample/FS/Driver/MMC_CM folder of the emFile shipment.

The table below lists the functions of the hardware layer for the MMC / SD device driver for ATMEL MCUS together with their possible replacement functions.

Old function	Possible replacement
FS_MCI_HW_EnableClock()	The clock to MCI host should be enabled in the (*pfInitHW)() function.
FS_MCI_HW_EnableISR()	The interrupt of MCI host should be enabled in the (*pfInitHW)() function.
FS_MCI_HW_GetMClk()	The information returned by this function is not required anymore.
FS_MCI_HW_GetMCIInfo()	The information returned by this function is not required anymore.
FS_MCI_HW_Init()	(*pfInitHW)()
FS_MCI_HW_IsCardPresent()	(*pfIsPresent)()
FS_MCI_HW_IsCardWriteProtected()	(*pfIsWriteProtected)()
FS_MCI_HW_GetTransferMem()	Non-cached memory should be allocated via a dedicated BSP function.

Chapter 20

Porting emFile 4.x to 5.x

This chapter describes the differences between the versions 4.x and 5.x of emFile and the changes required to port an existing application.

20.1 Differences between version 4.x and 5.x

Most of the differences between emFile version 4.x to version 5.x are internal. The API of emFile version 4.x is a subset of the version 5.x. There are two differences that require the porting of an application using emFile:

- The prototypes of the optional API functions are conditionally compiled.
- Some header and source files were removed or renamed.

20.2 Function prototype differences

emFile uses configuration defines to enable or disable different optional features of the file system at compile time. Typically, the optional features come with API functions that can be called by an application to make use of that optional feature. In older emFile versions these API functions performed no operation or unnecessarily allocated resources when the feature was disabled. The version 5.x changes this in that it defines the body and the prototype of these API functions only when the feature is enabled.

As a result of these changes it is possible that errors are reported when building an application that calls an API function of an optional feature that is not enabled at compile time. The solution to this problem is to enclose the calls to these API functions with #if/#endif processor commands that make sure the function is called only when the optional feature is enabled.

For example if the same application is expected to be compiled to make use of the memory allocator provided by emFile as well as of the memory allocation functions of the C library then the emFile configuration functions have to be called such in the following example:

```
#include <FS.h>
#include <stdlib.h>

#define ALLOC_SIZE        0x4000        // Size defined in bytes

#if (FS_SUPPORT_EXT_MEM_MANAGER == 0)
  static U32 _aMemBlock[ALLOC_SIZE / 4];  // Memory pool used for
                                          // semi-dynamic allocation.
#endif

/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*/
void FS_X_AddDevices(void) {
  //
  // Give the file system memory to work with.
  //
#if (FS_SUPPORT_EXT_MEM_MANAGER == 0)
  FS_AssignMemory(_aMemBlock, sizeof(_aMemBlock));
#else
  FS_SetMemHandler((FS_MEM_ALLOC_CALLBACK *)malloc, free);
#endif // FS_SUPPORT_EXT_MEM_MANAGER == 0

  //
  // Additional file system configuration...
  //
}
```

The following table lists all the prototypes of all optional API functions together with the name and the value of the configuration define that enables the optional feature. Note that the table contains all the function prototypes that are conditionally compiled including the function prototypes that were conditionally compiled in older emFile versions.

Function	Condition
FS_ConfigEOFErrorSuppression()	FS_SUPPRESS_EOF_ERROR ≠ 0
FS_SetVolumeAlias()	FS_MAX_LEN_VOLUME_ALIAS > 0
FS_DeInit()	FS_SUPPORT_DEINIT ≠ 0
FS_AddOnExitHandler()	FS_SUPPORT_DEINIT ≠ 0
FS_AssignMemory()	FS_SUPPORT_EXT_MEM_MANAGER = 0
FS_SetMemHandler()	FS_SUPPORT_EXT_MEM_MANAGER ≠ 0
FS_ConfigPOSIXSupport()	FS_SUPPORT_POSIX ≠ 0
FS_ConfigWriteVerification()	FS_VERIFY_WRITE ≠ 0

Function	Condition
FS_JOURNAL_Begin()	FS_SUPPORT_JOURNAL ≠ 0
FS_JOURNAL_Create()	FS_SUPPORT_JOURNAL ≠ 0
FS_JOURNAL_CreateEx()	FS_SUPPORT_JOURNAL ≠ 0
FS_JOURNAL_Disable()	FS_SUPPORT_JOURNAL ≠ 0
FS_JOURNAL_Enable()	FS_SUPPORT_JOURNAL ≠ 0
FS_JOURNAL_End()	FS_SUPPORT_JOURNAL ≠ 0
FS_JOURNAL_GetInfo()	FS_SUPPORT_JOURNAL ≠ 0
FS_JOURNAL_GetOpenCnt()	FS_SUPPORT_JOURNAL ≠ 0
FS_JOURNAL_GetStatCounters()	FS_SUPPORT_JOURNAL ≠ 0
FS_JOURNAL_Invalidate()	FS_SUPPORT_JOURNAL ≠ 0
FS_JOURNAL_IsEnabled()	FS_SUPPORT_JOURNAL ≠ 0
FS_JOURNAL_IsPresent()	FS_SUPPORT_JOURNAL ≠ 0
FS_JOURNAL_ResetStatCounters()	FS_SUPPORT_JOURNAL ≠ 0
FS_JOURNAL_SetFileName()	FS_SUPPORT_JOURNAL ≠ 0
FS_JOURNAL_SetOnOverflowExCallback()	FS_SUPPORT_JOURNAL ≠ 0
FS_JOURNAL_SetOnOverflowCallback()	FS_SUPPORT_JOURNAL ≠ 0
FS_FAT_FormatSD()	FS_SUPPORT_FAT ≠ 0
FS_FAT_GrowRootDir()	FS_SUPPORT_FAT ≠ 0
FS_FAT_SupportLFN()	FS_SUPPORT_FAT ≠ 0
FS_FAT_DisableLFN()	FS_SUPPORT_FAT ≠ 0
FS_FAT_SetLFNConverter()	FS_SUPPORT_FAT ≠ 0 && FS_SUPPORT_FILE_NAME_ENCODING ≠ 0
FS_FAT_ConfigFSInfoSectorUse()	FS_SUPPORT_FAT ≠ 0 && FS_FAT_USE_FSINFO_SECTOR ≠ 0
FS_FAT_ConfigFATCopyMaintenance()	FS_SUPPORT_FAT ≠ 0 && FS_MAINTAIN_FAT_COPY ≠ 0
FS_FAT_ConfigROFileMovePermission()	FS_SUPPORT_FAT ≠ 0 && FS_FAT_PERMIT_RO_FILE_MOVE ≠ 0
FS_FAT_ConfigDirtyFlagUpdate()	FS_SUPPORT_FAT ≠ 0 && FS_FAT_UPDATE_DIRTY_FLAG ≠ 0
FS_EFS_ConfigStatusSectorSupport()	FS_SUPPORT_EFS ≠ 0 && FS_EFS_SUPPORT_STATUS_SECTOR ≠ 0
FS_EFS_ConfigCaseSensitivity()	FS_SUPPORT_EFS ≠ 0 && FS_EFS_CASE_SENSITIVE ≠ 0
FS_EFS_SetFileNameConverter()	FS_SUPPORT_EFS ≠ 0 && FS_SUPPORT_FILE_NAME_ENCODING ≠ 0
FS_ConfigFileBufferDefault()	FS_SUPPORT_FILE_BUFFER ≠ 0
FS_SetFileBufferFlags()	FS_SUPPORT_FILE_BUFFER ≠ 0
FS_SetFileBufferFlagsEx()	FS_SUPPORT_FILE_BUFFER ≠ 0
FS_SetFileBuffer()	FS_SUPPORT_FILE_BUFFER ≠ 0
FS_SetBusyLEDCallback()	FS_SUPPORT_BUSY_LED ≠ 0
FS_SetMemCheckCallback()	FS_SUPPORT_CHECK_MEMORY ≠ 0
FS_STORAGE_DeInit()	FS_SUPPORT_DEINIT ≠ 0
FS_STORAGE_GetCounters()	FS_STORAGE_ENABLE_STAT_COUNTERS ≠ 0
FS_STORAGE_ResetCounters()	FS_STORAGE_ENABLE_STAT_COUNTERS ≠ 0

Function	Condition
FS_STORAGE_SetOnDeviceActivityCallback()	FS_STORAGE_SUPPORT_DEVICE_ACTIVITY ≠ 0
FS_AssignCache()	FS_SUPPORT_CACHE ≠ 0
FS_CACHE_Clean()	FS_SUPPORT_CACHE ≠ 0
FS_CACHE_Command()	FS_SUPPORT_CACHE ≠ 0
FS_CACHE_SetMode()	FS_SUPPORT_CACHE ≠ 0
FS_CACHE_SetQuota()	FS_SUPPORT_CACHE ≠ 0
FS_CACHE_SetAssocLevel()	FS_SUPPORT_CACHE ≠ 0
FS_CACHE_GetNumSectors()	FS_SUPPORT_CACHE ≠ 0
FS_CACHE_Invalidate()	FS_SUPPORT_CACHE ≠ 0
FS_SetFSType()	FS_SUPPORT_MULTIPLE_FS ≠ 0
FS_GetFSType()	FS_SUPPORT_MULTIPLE_FS ≠ 0
FS_Free()	FS_SUPPORT_DEINIT ≠ 0
FS_Lock()	FS_OS_LOCKING = FS_OS_LOCKING_API
FS_Unlock()	FS_OS_LOCKING = FS_OS_LOCKING_API
FS_LockVolume()	FS_OS_LOCKING = FS_OS_LOCKING_DRIVER
FS_UnlockVolume()	FS_OS_LOCKING = FS_OS_LOCKING_DRIVER
FS_SYSVIEW_Init()	FS_SUPPORT_PROFILE ≠ 0
FS_AddErrorFilter()	FS_DEBUG_LEVEL ≥ FS_DEBUG_LEVEL_LOG_ERRORS
FS_SetErrorFilter()	FS_DEBUG_LEVEL ≥ FS_DEBUG_LEVEL_LOG_ERRORS
FS_GetErrorFilter()	FS_DEBUG_LEVEL ≥ FS_DEBUG_LEVEL_LOG_ERRORS
FS_AddWarnFilter()	FS_DEBUG_LEVEL ≥ FS_DEBUG_LEVEL_LOG_WARNINGS
FS_SetWarnFilter()	FS_DEBUG_LEVEL ≥ FS_DEBUG_LEVEL_LOG_WARNINGS
FS_GetWarnFilter()	FS_DEBUG_LEVEL ≥ FS_DEBUG_LEVEL_LOG_WARNINGS
FS_AddLogFilter()	FS_DEBUG_LEVEL ≥ FS_DEBUG_LEVEL_LOG_ALL
FS_SetLogFilter()	FS_DEBUG_LEVEL ≥ FS_DEBUG_LEVEL_LOG_ALL
FS_GetLogFilter()	FS_DEBUG_LEVEL ≥ FS_DEBUG_LEVEL_LOG_ALL
FS_NOR_GetStatCounters()	FS_NOR_ENABLE_STATS ≠ 0
FS_NOR_ResetStatCounters()	FS_NOR_ENABLE_STATS ≠ 0
FS_NOR_SetBlankSectorSkip()	FS_NOR_SKIP_BLANK_SECTORS ≠ 0
FS_NOR_SetDeviceLineSize()	FS_NOR_SUPPORT_VARIABLE_LINE_SIZE ≠ 0
FS_NOR_SetDeviceRewriteSupport()	FS_NOR_SUPPORT_VARIABLE_LINE_SIZE ≠ 0
FS_NOR_SetDirtyCheckOptimization()	FS_NOR_OPTIMIZE_DIRTY_CHECK ≠ 0
FS_NOR_SetEraseVerification()	FS_NOR_VERIFY_ERASE ≠ 0
FS_NOR_SetWriteVerification()	FS_NOR_VERIFY_WRITE ≠ 0

Function	Condition
FS_NOR_BM_DisableCRC()	FS_NOR_SUPPORT_CRC ≠ 0
FS_NOR_BM_EnableCRC()	FS_NOR_SUPPORT_CRC ≠ 0
FS_NOR_BM_GetStatCounters()	FS_NOR_ENABLE_STATS ≠ 0
FS_NOR_BM_IsCRCEnabled()	FS_NOR_SUPPORT_CRC ≠ 0
FS_NOR_BM_ResetStatCounters()	FS_NOR_ENABLE_STATS ≠ 0
FS_NOR_BM_SetBlankSectorSkip()	FS_NOR_SKIP_BLANK_SECTORS ≠ 0
FS_NOR_BM_SetByteOrderBE()	FS_NOR_SUPPORT_VARIABLE_BYTE_ORDER ≠ 0
FS_NOR_BM_SetByteOrderLE()	FS_NOR_SUPPORT_VARIABLE_BYTE_ORDER ≠ 0
FS_NOR_BM_SetCRCHook()	FS_NOR_SUPPORT_CRC ≠ 0
FS_NOR_BM_SetDeviceLineSize()	FS_NOR_SUPPORT_VARIABLE_LINE_SIZE ≠ 0
FS_NOR_BM_SetDeviceRewriteSupport()	FS_NOR_SUPPORT_VARIABLE_LINE_SIZE ≠ 0
FS_NOR_BM_SetEraseVerification()	FS_NOR_VERIFY_ERASE ≠ 0
FS_NOR_BM_SetFailSafeErase()	FS_NOR_SUPPORT_FAIL_SAFE_ERASE ≠ 0
FS_NOR_BM_SetWriteVerification()	FS_NOR_VERIFY_WRITE ≠ 0
FS_NOR_SPI_AddDevice()	FS_NOR_MAX_NUM_DEVICES ≠ 0
FS_MMC_GetStatCounters()	FS_MMC_ENABLE_STATS ≠ 0
FS_MMC_ResetStatCounters()	FS_MMC_ENABLE_STATS ≠ 0
FS_MMC_CM_AllowPowerSaveMode()	FS_MMC_SUPPORT_POWER_SAVE ≠ 0
FS_MMC_CM_EnterPowerSaveMode()	FS_MMC_SUPPORT_POWER_SAVE ≠ 0
FS_MMC_CM_GetStatCounters()	FS_MMC_ENABLE_STATS ≠ 0
FS_MMC_CM_ResetStatCounters()	FS_MMC_ENABLE_STATS ≠ 0
FS_NAND_GetStatCounters()	FS_NAND_ENABLE_STATS ≠ 0
FS_NAND_ResetStatCounters()	FS_NAND_ENABLE_STATS ≠ 0
FS_NAND_SetEraseVerification()	FS_NAND_VERIFY_ERASE ≠ 0
FS_NAND_SetWriteVerification()	FS_NAND_VERIFY_WRITE ≠ 0
FS_NAND_UNI_GetStatCounters()	FS_NAND_ENABLE_STATS ≠ 0
FS_NAND_UNI_ResetStatCounters()	FS_NAND_ENABLE_STATS ≠ 0
FS_NAND_UNI_SetDriverBadBlockReclamation()	FS_NAND_RECLAIM_DRIVER_BAD_BLOCKS ≠ 0
FS_NAND_UNI_SetEraseVerification()	FS_NAND_VERIFY_ERASE ≠ 0
FS_NAND_UNI_SetNumBlocksPerGroup()	FS_NAND_SUPPORT_BLOCK_GROUPING ≠ 0
FS_NAND_UNI_SetWriteVerification()	FS_NAND_VERIFY_WRITE ≠ 0
FS_NAND_2048x8_EnableReadCache()	FS_NAND_SUPPORT_READ_CACHE ≠ 0
FS_NAND_2048x8_DisableReadCache()	FS_NAND_SUPPORT_READ_CACHE ≠ 0
FS_NAND_x8_Configure()	FS_NAND_SUPPORT_AUTO_DETECTION ≠ 0
FS_NAND_x_Configure()	FS_NAND_SUPPORT_AUTO_DETECTION = 0
FS_NAND_SPI_EnableReadCache()	FS_NAND_SUPPORT_READ_CACHE ≠ 0
FS_NAND_SPI_DisableReadCache()	FS_NAND_SUPPORT_READ_CACHE ≠ 0
FS_NAND_QSPI_EnableReadCache()	FS_NAND_SUPPORT_READ_CACHE ≠ 0
FS_NAND_QSPI_DisableReadCache()	FS_NAND_SUPPORT_READ_CACHE ≠ 0
FS_WINDRIVE_Configure()	_WIN32

Function	Condition
FS_WINDRIVE_ConfigureEx()	_WIN32
FS_WINDRIVE_SetGeometry()	_WIN32
FS_SetEncryptionObject()	FS_SUPPORT_ENCRYPTION ≠ 0
FS_READAHEAD_GetStatCounters()	FS_READAHEAD_ENABLE_STATS ≠ 0
FS_READAHEAD_ResetStatCounters()	FS_READAHEAD_ENABLE_STATS ≠ 0

20.3 File naming differences

The names of some of the emFile header and source files were renamed in order to improve consistency. This change makes sure that the names of all the header and source files that ship with emFile begin with the prefix "FS_". In addition, the contents of some of the header and source files was moved to other files in order to keep the number of files to a minimum. The removal of some of the header files can possibly cause compiler errors. The errors can be corrected by including the name of the header file specified in the following table instead of the missing file header.

The next table lists the names of the header ans source files that were renamed, moved or deleted in the version 5.x of emFile. The first column stores the original file name while the second column holds either the new file name or the file name that stores its contents. If the file has been deleted the name of the file is set to "-" in the second column. The third column stores a code indicating the operation that was performed on that file:

- M - contents of the file has been moved
- R - file has been renamed
- D - file has been deleted

Old file name	New file name	Operation
CRYPT_ALGO_AES.c	FS_CRYPT_ALGO_AES.c	R
CRYPT_ALGO_DES.c	FS_CRYPT_ALGO_DES.c	R
CRYPT_Drv.c	FS_CRYPT_Drv.c	R
CRYPT_File.c	FS_CRYPT_File.c	R
EFS.h	FS_EFS.h	R
EFS_API.c	FS_EFS_API.c	R
EFS_CheckDisk.c	FS_EFS_CheckDisk.c	R
EFS_Dir.c	FS_EFS_Dir.c	R
EFS_DirEntry.c	FS_EFS_DirEntry.c	R
EFS_Format.c	FS_EFS_Format.c	R
EFS_Intern.h	FS_EFS_Int.h	R
EFS_Misc.c	FS_EFS_Misc.c	R
EFS_Move.c	FS_EFS_Move.c	R
EFS_Open.c	FS_EFS_Open.c	R
EFS_Read.c	FS_EFS_Read.c	R
EFS_Rename.c	FS_EFS_Rename.c	R
EFS_SetEndOfFile.c	FS_EFS_SetEndOfFile.c	R
EFS_Write.c	FS_EFS_Write.c	R
FAT.h	FS_FAT.h	R
FAT_API.c	FS_FAT_API.c	R
FAT_CheckDisk.c	FS_FAT_CheckDisk.c	R
FAT_Dir.c	FS_FAT_Dir.c	R
FAT_DirEntry.c	FS_FAT_DirEntry.c	R
FAT_DiskInfo.c	FS_FAT_DiskInfo.c	R
FAT_Format.c	FS_FAT_Format.c	R
FAT_FormatSD.c	FS_FAT_FormatSD.c	R
FAT_Intern.h	FS_FAT_Int.h	R
FAT_Ioct.c	-	D
FAT_LFN.c	FS_FAT_LFN.c	R

Old file name	New file name	Operation
FAT_Misc.c	FS_FAT_Misc.c	R
FAT_Move.c	FS_FAT_Move.c	R
FAT_Open.c	FS_FAT_Open.c	R
FAT_Read.c	FS_FAT_Read.c	R
FAT_Rename.c	FS_FAT_Rename.c	R
FAT_SetEndOfFile.c	FS_FAT_SetEndOfFile.c	R
FAT_VolumeLabel.c	FS_FAT_VolumeLabel.c	R
FAT_Write.c	FS_FAT_Write.c	R
FS_AddSpaceHex.c	FS_Debug.c	M
FS_API.h	-	D
FS_AssignCache.c	FS_CACHE_Core.c	R
FS_CacheAll.c	FS_CACHE_All.c	R
FS_CacheMan.c	FS_CACHE_Man.c	R
FS_CacheMultiWay.c	FS_CACHE_MultiWay.c	R
FS_CacheRW.c	FS_CACHE_RW.c	R
FS_CacheRWQuota.c	FS_CACHE_RWQuota.c	R
FS_CLib.h	-	D
FS_Dev.h	-	D
FS_DF_X_HW.h	FS.h	M
FS_ErrorOut.c	FS_Debug.c	M
FS_FRead.c	FS_Read.c	M
FS_FWrite.c	FS_Write.c	M
FS_GetFilePos.c	-	D
FS_GetNumOpenFiles.c	FS_Misc.c	M
FS_Lbl.h	-	D
FS_Log.c	FS_Debug.c	M
FS_SetFilePos.c	-	D
FS_Warn.c	FS_Debug.c	M
FS__ECC256.c	FS_ECC256.c	R
IDE_Drv.c	FS_IDE_Drv.c	R
IDE_X_HW.h	FS.h	M
MMC_Drv.c	FS_MMC_Drv.c	R
MMC_SD_CardMode_Drv.c	FS_MMC_CM_Drv.c	R
MMC_SD_CardMode_X_HW.h	FS.h	M
MMC_X_HW.h	FS.h	M
NAND_Drv.c	FS_NAND_Drv.c	R
NAND_ECC_HW.c	FS_NAND_ECC_HW.c	R
NAND_ECC_SW_1Bit.c	FS_NAND_ECC_SW_1Bit.c	R
NAND_HW_SPI_X.h	FS.h	M
NAND_Misc.c	FS_NAND_Misc.c	R
NAND_PHY_2048x16.c	FS_NAND_PHY_2048x16.c	R
NAND_PHY_2048x8.c	FS_NAND_PHY_2048x8.c	R
NAND_PHY_4096x8.c	FS_NAND_PHY_4096x8.c	R

Old file name	New file name	Operation
NAND_PHY_512x8.c	FS_NAND_PHY_512x8.c	R
NAND_PHY_DataFlash.c	FS_NAND_PHY_DataFlash.c	R
NAND_PHY_ONFI.c	FS_NAND_PHY_ONFI.c	R
NAND_PHY_SPI.c	FS_NAND_PHY_SPI.c	R
NAND_PHY_x8.c	FS_NAND_PHY_x8.c	R
NAND_Private.h	FS_NAND_Int.h	R
NAND_UNI_Drv.c	FS_NAND_UNI_Drv.c	R
NAND_X_HW.h	FS.h	M
NOR_BM_Drv.c	FS_NOR_BM_Drv.c	R
NOR_Drv.c	FS_NOR_Drv.c	R
NOR_HW_CFI_1x16.c	FS_NOR_HW_CFI_1x16.c	R
NOR_HW_CFI_2x16.c	FS_NOR_HW_CFI_2x16.c	R
NOR_HW_SPI_X.h	FS.h	M
NOR_PHY_CFI.c	FS_NOR_PHY_CFI.c	R
NOR_PHY_DSPI.c	FS_NOR_PHY_DSPI.c	R
NOR_PHY_SFDP.c	FS_NOR_PHY_SFDP.c	R
NOR_PHY_SPIFI.c	FS_NOR_PHY_SPIFI.c	R
NOR_PHY_ST_M25.c	FS_NOR_PHY_ST_M25.c	R
NOR_Private.h	FS_NOR_Int.h	R
NOR_SPI_Dev.c	FS_NOR_SPI_Dev.c	R
NOR_SPI_Dev.h	FS.h	M
RAMDISK.c	FS_RAMDisk.c	R
WinDrive.c	FS_WinDrive.c	R
FS_X_embOS.c	FS_OS_embOS.c	R
FS_X_FreeRTOS.c	FS_OS_FreeRTOS.c	R
FS_X_None.c	FS_OS_None.c	R
FS_X_ucOS_II.c	FS_OS_ucOS_II.c	R
FS_X_Win32.c	FS_OS_Win32.c	R

Chapter 21

Support

This chapter should help if any problem occurs, e.g. with the use of the emFile functions, and describes how to contact the SEGGER support.

21.1 Contacting SEGGER support

If you are a registered emFile user and need to contact the SEGGER support, please send the following information via email to `support@segger.com`:

- The emFile version.
- Your emFile registration number.
- If you are unsure about the above information, you may also use the name of the shipped emFile ZIP-file (which contains the above information).
- A detailed description of the problem.
- (Optional) A project with which we can reproduce the problem.

Chapter 22

FAQs

This chapter contains a collection of frequently asked questions (FAQs) together with their respective answers.

22.1 Questions and answers

Q: Is my data safe when an unexpected reset occurs?

A: In general, the data which is already on the storage device is safe. If a read operation is interrupted, this is completely harmless. If a write operation is interrupted, then the data written during this operation may or may not be stored on the storage device. This depends on when when the unexpected reset occurred. In any case, the data which was on the storage device prior to the write operation is not affected. This is true if the storage device is not affected by the reset, meaning that the storage device is able to complete a pending write operation. All this is true for the device drivers that come with emFile. However, the structure of the FAT or EFS file systems can be corrupted by an unexpected reset. This can be prevented by using the Journaling component.

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. The size of the root directory can be specified at format, 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 size of the root directory 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 storage 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 equal to 512 bytes) one can use FAT32 on a storage medium 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 storage medium is smaller than or equal to 32 MB or the cluster size is greater than 512 bytes, FAT32 cannot be used. The second parameter of the `FS_Format()` API function can be used to set a custom root directory size for FAT12 / FAT16 volume. This parameter is a pointer to a `FS_FORMAT_INFO` structure that contains a member named NumRootDirEntries which can be used to specify the number of entries in the root directory.

Chapter 23

Glossary

List of some of the terms used in this manual together with a short explanation.

23.1 List of terms

Allocation table
A reserved storage area where the file system stores information about what clusters are allocated.

BPB
BIOS Parameter Block. Logical sector (typically the first in a partition) that stores information about how the storage device is formatted. This information is updated during the high-level format operation.

BSP
Board Support Package.

CHS address
Address on a mechanical storage device specified as Cylinder, Head and Sector.

Cluster
Smallest storage allocation unit of the file system. It consist of one or more logical sectors.

Cluster chain
File system data structure that stores information about what clusters are allocated to the same file or directory.

Directory entry
File system data structure that stores information about a file or directory such as name, size, time stamps, etc.

EFS
Embedded File System. SEGGER's proprietary file system specifically designed for embedded systems.

eMMC
embedded Multi-Media Card. A MMC compatible storage device that is packaged in an integrated circuit

FAT
File Allocation Table. Widely used file system designed by Microsoft.

LFN
Long File Name. Extension for FAT file system that provides support for file and directory names longer that 8 characters in the base name and 3 characters in the extension

Logical sector
Smallest storage allocation unit that a device or logical driver can handle.

MBR
Master Boot Record. File system data structure that stores information about how the storage device is partitioned.

MCU
MicroController Unit. A small computer that embeds a CPU, memory and peripherals on the same integrated circuit.

MMC
Multi-Media Card. Type of removable non-volatile storage device that conforms with MMC specification.

Flash memory
A type of non-volatile storage device that can be electrically erased and programmed.

OS layer
emFile interface to task synchronization services provided by an operating system.

SD card
Secure Digital card. Type of removable non-volatile storage device that conforms with SD specification.

Storage device
A device that can store and preserve data between power cycles such as NAND flash, NOR flash, SD card, eMMC, or USB drive.

