SafeFLASH for Adesto DataFlash Drives User Guide

Version 1.00

For use with SafeFLASH for Adesto® DataFlash Drives module versions 2.01 and above

Date: 19-Nov-2015 15:25

All rights reserved. This document and the associated software are the sole property of HCC Embedded. Reproduction or duplication by any means of any portion of this document without the prior written consent of HCC Embedded is expressly forbidden.

HCC Embedded reserves the right to make changes to this document and to the related software at any time and without notice. The information in this document has been carefully checked for its accuracy; however, HCC Embedded makes no warranty relating to the correctness of this document.

Table of Contents

System Overview 3
  Introduction 3
  Feature Check 4
  Packages and Documents 5
    Packages 5
    Documents 5
  Change History 6
Source File List 7
  API Header File 7
  Configuration File 7
  Source Files 7
  Version File 7
Configuration Options 8
System Features 9
  DataFlash Specifics 9
  NOR Physical Device Usage 9
    Reserved blocks 10
    File System Blocks 10
    Descriptor Blocks 11
Configuring DataFlash 12
  Device Types 12
  Flash Device-Specific Definitions 13
Application Programming Interface 14
  API Functions 14
    fs_getmem_dataflashdrive 14
    fs_mount_dataflashdrive 15
    Using f_mountdrive with DataFlash 16
    Mounting the DataFlash Drive 18
  Physical Interface Functions 19
    s_atmel_flash_fs_phy 20
    adf_read_flash 21
    adf_write_flash 22
    adf_erase_flash 23
    adf_verify_flash 24
    adf_block_copy 25
  Types and Definitions 26
    FS_FLASH Structure 26
Porting the Serial Peripheral Interface 27
  spi.h Header File 27
  spi.c Source Code 27

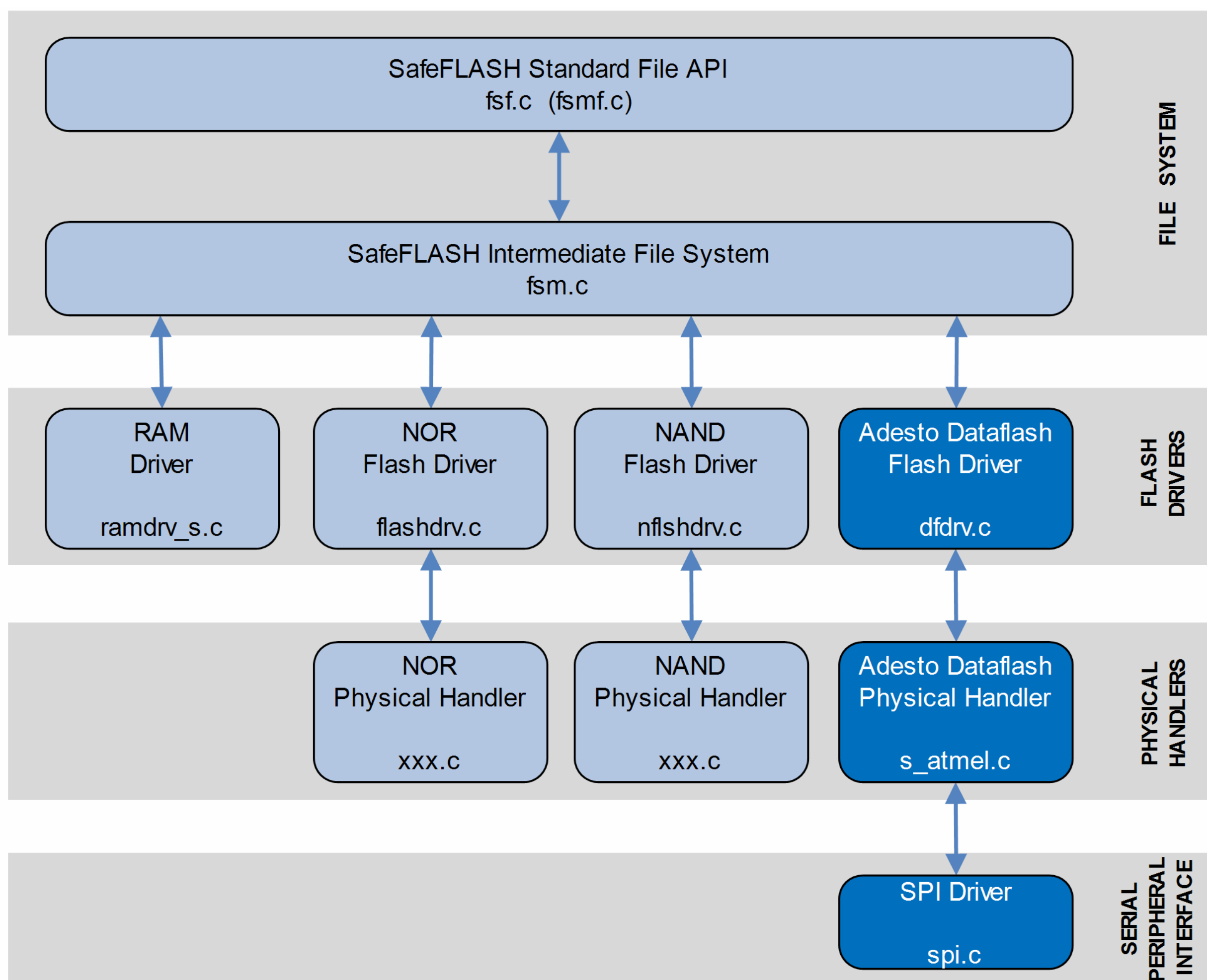
1 System Overview

1.1 Introduction

This guide is for those who wish to implement an Adesto® Technologies DataFlash drive for HCC Embedded's SafeFLASH file system.

> **Note:** This DataFlash technology was formerly owned by Atmel® but is now owned by Adesto® Technologies.

The following diagram illustrates the structure of the file system software:



In this diagram:

- The main SafeFLASH package provides the file API and intermediate file system. This is described in the *HCC SafeFLASH File System User Guide*.

- The DataFlash flash driver is the device driver. This guide shows how to add this to the build. Using the available sample drivers as a model, you can create a driver that meets your specific needs.
- The DataFlash physical handler performs the translation between the driver and the physical flash hardware. Generally only the physical handler needs to be modified when the hardware configuration changes (for example, a change to a different chip type, or use of 1, 2, or 4 devices in parallel).
- The physical interface to the DataFlash uses the Serial Peripheral Interface (SPI) standard.

Note:

- HCC Embedded has a range of physical handlers available to make the porting process as simple as possible. HCC also offers special porting services when required.
- HCC Embedded offers hardware and firmware development consultancy to assist developers with the implementation of flash file systems.

HCC's DataFlash Management Layer (DFML) is a clean interface layer between a file system and Adesto® DataFlash devices. DFML provides a robust Adesto® DataFlash management and file system for a PC-compatible or fail-safe application.

1.2 Feature Check

For a full list of SafeFlash features, see the *HCC SafeFlash File System User Guide*.

The system features which are especially relevant to Adesto® DataFlash are as follows:

- Support for all serial DataFlash and DataFlash cards, from 1Mbit to 64Mbit.
- A reliable and fail-safe page write operation, ensuring that critical data are always reliably written to the device, even after a power loss.
- A normal page write for non-critical data.
- Managed sectors set aside for non-file system usage.
- Unmanaged sector reservation for private functions, such as boot memory operations.
- 10K random write per sector refresh operations are managed transparently.
- Enhanced static and dynamic wear leveling operations.
- Support for multiple devices in a single array.
- Bad block management, including mapping of unusable areas to ensure that data is not corrupted.

The system supports the following Adesto® Technologies DataFlash devices:

- AT45DB021E
- AT45DB041E
- AT45DB081E
- AT45DB161E
- AT45DB321E
- AT45DB641E
- AT45DQ081
- AT45DB161

- AT45DB321

It also supports the following legacy devices:

- AT45DB021B/D
- AT45DB041B/D
- AT45DB081B/D
- AT45DB161B/D
- AT45DB321B/D
- AT45DB641B/D

1.3 Packages and Documents

Packages

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

Package	Description
hcc_base_doc	This contains the two guides that will help you get started.
fs_safe	The SafeFLASH base package.
fs_safe_df	The SafeFLASH package for Adesto® Dataflash, described in this document.

Documents

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

Readers should note the points in the HCC Documentation Guidelines on the HCC documentation website.

HCC Firmware Quick Start Guide

This document describes how to install packages provided by HCC in the target development environment. Also follow the *Quick Start Guide* when HCC provides package updates.

HCC Source Tree Guide

This document describes the HCC source tree. It gives an overview of the system to make clear the logic behind its organization.

HCC SafeFLASH File System User Guide

This document describes the base SafeFLASH system.

HCC SafeFLASH File System Adesto DataFlash Drive User Guide

This is this document.

Other HCC SafeFLASH Guides

These describe other SafeFLASH components:

- *HCC SafeFLASH System RAM Drive User Guide* – documents the SafeFLASH system for RAM.
- *HCC SafeFLASH System NAND Drive User Guide* – documents the SafeFLASH system for NAND flash.
- *HCC SafeFLASH System NOR Drive User Guide* – documents the SafeFLASH system for NOR flash.

HCC FileSystem Memory Calculator (FSmem.exe) User Guide

This document describes the **FSmem.exe** utility that should make it easier to derive the optimum solution for your requirements.

1.4 Change History

This section includes recent changes to this product. For a list of all the changes, refer to the file **src/history/safe-flash/safeflash_df.txt** in the distribution package.

Version	Changes
2.02	Added support for AT45DQ321.
2.01	Created files **api_safe_dflash.h** and **config_safe_dflash.h**. Moved FS_DFLASH_MAXFILE to **config_safe_dflash.h**.
1.02	Updated module to work with PSP_SPI major version 3.
1.01	Improved PHY drivers. Added support for AT45DB321E.
1.00	Initial release.

2 Source File List

This section lists and describes all the source code files included in the system. These files follow HCC Embedded's standard source tree system, described in the *HCC Source Tree Guide*. All references to file pathnames refer to locations within this standard source tree, not within the package you initially receive.

Note: Do not modify any files except the configuration file.

2.1 API Header File

The file **src/api/api_safe_dflash.h** is the only file that should be included by an application using this module. It defines the **fs_getmem_dataflashdrive()** and **fs_mount_dataflashdrive()** functions.

2.2 Configuration File

The file **src/config/config_safe_dflash.h** contains the single configurable parameter of the system. Configure this as required. This is the only file in the module that you should modify. For details of the option, see Configuration Options.

2.3 Source Files

These files should only be modified by HCC.

The file **src/safe-flash/dflash/dfdrv.c** is the DataFlash generic driver source.

The following files are in the directory **src/safe-flash/dflash/phy/atmel**:

File	Description
f_atmel.c	Source code.
f_atmel.h	Header file.
m_atmel.c	Source code.
m_atmel.h	Header file.
s_atmel.c	DataFlash physical handler.
s_atmel.h	DataFlash physical handler header file.

2.4 Version File

The file **src/version/ver_safe_dflash.h** contains the version number of this module. This version number is checked by all modules that use this module to ensure system consistency over upgrades.

3 Configuration Options

Set the single system configuration option in the file **src/config/config_safe_dflash.h**.

FS_DFLASH_MAXFILE

The maximum number of files that can be open at the same time. The default is 3.

4 System Features

4.1 DataFlash Specifics

These devices have their own particular characteristics; therefore, for reliable operation, drivers must be designed specifically for them.

The main features are:

- Data are written to a page buffer in the RAM of the device. Before it is written, the underlying buffer is erased. This means that if a power loss occurs all or part of a page can be lost.
- If there have been 10,000 write operations to a particular sector, then the system must ensure that every page in that sector has been written to during that time; otherwise data loss may occur.

This driver handles all DataFlash issues to provide an efficient and failsafe interface for using these devices.

The interface is straightforward. Include the **dfdrv.c**, **s_atmel.c** and **spi.c** files and their headers in your project and follow the sections below.

4.2 NOR Physical Device Usage

You must make some decisions about how to use the flash device, and must be aware that:

- All flash devices are divided into a set of erasable blocks. On some devices the size of these blocks may vary.
- It is only possible to write to an erased location.
- It is not possible to erase anything smaller than a block.

These facts mean that some complex management software has to be used.

> **Note**: The **FSmem.exe** utility in the **util** folder of the main **fs_safe** package should help you understand the use of the blocks and make it easier to get the optimum solution for your requirements.

SafeFLASH operates on a set of logical blocks that may be further divided into sectors. The physical driver must do two things in this respect:

1. Define for the file system which logical block numbers are to be used for what purpose. This is configured in the FS_FLASH structure and returned to the file system by the **s_atmel_flash_fs_phy()** function.
2. Provide a mapping between the logical block numbers used by the file system and the physical addresses of the blocks in the flash device (this is done by the **GetBlockAddr()** function).

You can assign three types of block to the device. The following sections describe these.

- **Reserved blocks** – for processes other than the file system, for example booting.
- **File system blocks** – for storing file information.
- **Descriptor blocks** – to hold information about the structure of the file system, wear, and so on. By using a minimum of two descriptor blocks (and management software), the system is made fail-safe.

Reserved blocks

Blocks can be reserved for private usage without restriction. To do this, simply omit those blocks from the **GetBlockAddr()** function.

Access reserved blocks by using the **GetBlockAddr()** function and also by selecting the physical block numbers to be used and ensuring that they are not specified in the descriptor and file system usage described below.

> **Note:** Take care in accessing reserved blocks and, to ensure interoperability, pay attention to the specification of the device used. Some devices allow an erase operation to be performed while another block is being read, others have different rules. In general it is sensible to use only the file system or the reserved sectors at any one time. In any case, careful understanding of the specific device is required.

File System Blocks

Allocate as many of these as required for file storage. Set the following parameters up by using **s_atmel_flash_fs_phy()** to create an FS_FLASH structure:

maxblock

The number of erasable blocks that are available for file storage.

blocksize

The size of the blocks to be used in the file storage area. This must be an erasable unit of the flash chip. All blocks in the file storage area must be the same size. This may be different from the **descsize** (see below) where the flash chip has erasable units of different sizes.

sectorsize

The sector size. Each block is divided into a number of sectors. sectorsize is the smallest usable unit in the system and thus represents the minimum file storage area. For best usage of the flash blocks, the sector size should always be a power of 2.

sectorperblock

The number of sectors in a block. It must always be true that:

sectorperblock = blocksize / sectorsize

blockstart

The logical number of the first of the block that may be used for file storage. This is the logical number used when the **GetBlockAddr()** function is called.

Descriptor Blocks

These blocks contain critical information about the file system, including block allocation, wear information, caching information, and file/directory information. They are allocated automatically from the file system blocks. At least two descriptor blocks that can be erased independently must be included in the system. An optional descriptor writing cache may be configured; this improves the performance of the file system.

On a flash device with different sized blocks, it is generally sensible to use some of the smaller blocks as descriptor blocks. This also improves the performance of the system. However, when using the cache this is not so important and it is preferable to allocate a larger cache.

Set the following parameters up by using **s_atmel_flash_fs_phy()** to create an FS_FLASH structure:

descsize

This is the size of a descriptor block.

> **Note:** Where RAM usage is a consideration, it is also possible to set the descriptor size to less than the physical block size, so long as it fits in a single physical block that is used only for this purpose.

descblockstart

The logical number of the first block to be used by the file system as a descriptor block.

descblockend

The logical number of the last block to be used by the file system as a descriptor block.

cacheddescsize

The descriptor write cache size. This number must be less than descsize, since the cache is allocated in the descriptor block. If it is set to zero the descriptor-write-cache method will not be used. The descriptor write cache is an efficient method of updating the changes in the descriptor, since the whole descriptor need not be re-written, while the 100% power-fail safe characteristics of the system will still be retained.

Use of the descriptor write cache reduces to an absolute minimum the wear leveling and the number of erases required when updating the system.

Using the descriptor write cache is highly recommended since performance and wear characteristics of the system are improved by a larger cache. However, a larger cache size also reduces the number of directory entries; use the **FSmem.exe** utility in the **util** folder of the main **fs_safe** package to check the effect of this.

5 Configuring DataFlash

5.1 Device Types

To configure the system, open the file **s_atmel.h** and define your device type from those listed below.

Note: Use the **FSmem.exe** utility to learn about the use of these settings, and to make it easier to derive the optimum solution for your requirements.

- AT45DB11B
- AT45DB21B
- AT45DB41B
- AT45DB81B
- AT45DB161B
- AT45DB321B
- AT45DB321D
- AT45DB321E
- AT45DB642B

Note: If your device is not one of those listed, contact support@hcc-embedded.com.

Defining the device type automatically sets the following parameters to their default values for the specified chip:

- ADF_PAGE_SIZE
- ADF_REAL_PAGE_COUNT
- ADF_NUM_OF_SECTORS
- ADF_PAGES_PER_SECTOR
- ADF_BYTE_ADDRESS_WIDTH
- F_ATMEL_DEFAULT_SECTORS_PER_BLOCK
- F_ATMEL_DEFAULT_CACHED_DESC
- F_ATMEL_DEFAULT_NO_OF_DESC_BLOCKS
- ADF_ID

Note: The default values comprise a tested configuration that uses the entire target device. **Do not change from the default settings without very good reason.**

5.2 Flash Device-Specific Definitions

The following defines give the standard definitions for the driver. These are in the **f_atmel.h**, **m_atmel.h**, and **s_atmel.h** files.

F_ATMEL_SECTORSIZE

The sector size to be used by the file system, the number of pages that form a sector. This should always be a multiple of eight pages.

F_ATMEL_BLOCKSIZE

The block size to be used by the file system, the number of sectors in a block. This should always be the sector size multiplied by a power of 2.

F_ATMEL_CACHED_DESC

The number of cached descriptors.

F_ATMEL_NO_OF_DESC_BLOCKS

The number of descriptor blocks.

6 Application Programming Interface

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

6.1 API Functions

There are just two functions in the API, **fs_getmem_dataflashdrive()** and **fs_mount_dataflashdrive()**.

fs_getmem_dataflashdrive

Use this function to get the required memory for a driver.

The function calculates and returns the amount of memory that must be allocated for the physical driver. You must then allocate the memory and pass its pointer and size to **f_mountdrive()**. See the example code in Mounting the DataFlash Drive for details.

Format

```
extern long fs_getmem_flashdrive ( FS_PHYGETID phyfunc )
```

Arguments

Argument	Description	Type
phyfunc	The **s_atmel_flash_fs_phy()** function of the physical chip driver to be mounted.	FS_PHYGETID

Return value

The required memory.

fs_mount_dataflashdrive

This function is called by **f_mountdrive()** to mount and map a new drive.

For more details, see Using f_mountdrive() with DataFlash.

Format

```
extern int fs_mount_flashdrive (
    FS_VOLUMEDESC *   vd,
    FS_PHYGETID       phyfunc )
```

Arguments

Argument	Description	Type
vd	The volume descriptor of the volume to mount.	FS_VOLUMEDESC *
phyfunc	The physical driver.	FS_PHYGETID

Return values

Return value	Description
0	Drive successfully mounted.
FS_VOL_NOTFORMATTED	Drive is mounted but is not formatted.
FS_VOL_NOMEMORY	Not enough memory; drive is not mounted.
FS_VOL_DRVERROR	Mount driver error; drive is not mounted.

Using f_mountdrive with DataFlash

The **f_mountdrive()** function is part of the main SafeFLASH API. It calls **fs_mount_dataflashdrive()**. This page shows how to use the function with Adesto® DataFlash. For a code example, see Mounting the DataFlash Drive.

> **Note:** The main *SafeFLASH File System User Guide* describes how to use this call for all drive types.

Format

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

Arguments

Argument	Description	Type
drivenum	The number of the drive to mount (0='A', 1='B', and so on).	int
buffer	The buffer pointer to be used by the file system.	void *
buffsize	The size of the allocated buffer.	long
mountfunc	The **fs_mount_flashdrive()** function.	FS_DRVMOUNT
phyfunc	A pointer to the **s_atmel_flash_fs_phy()** physical driver function that is called by the mount function to get information about how to use it.	FS_PHYGETID

Return values

Return value	Description
FS_VOL_OK	The drive is successfully mounted.
FS_VOL_NOTMOUNT	The drive is not mounted.
FS_VOL_NOTFORMATTED	The drive is mounted but is not formatted.
FS_VOL_NOMEMORY	Not enough memory; the drive is not mounted.
FS_VOL_NOMORE	No more drives are available (FS_MAXVOLUME).
FS_VOL_DRVERROR	Mount driver error; the drive is not mounted.

Mounting the DataFlash Drive

The following code shows how to mount your DataFlash drive:

```
long memsize;
char *p1buffer;

    memsize = fs_getmem_dataflashdrive( s_atmel_flash_fs_phy );
    if (!memsize)
    {
        /* Configuration error! */
    }

    p1buffer = (char*)malloc( memsize );

    if (!p1buffer)
    {
        /* Not enough memory to allocate! */
        return;
    }

    /* Mount drive number 1 (1 = B)  */
    f_mountdrive( 1, p1buffer, memsize,    fs_mount_dataflashdrive, s_atmel_flash_fs_phy );

/* The DataFlash drive is ready for use */
```

6.2 Physical Interface Functions

The functions in this section provide the interface to the upper layer and must be ported to meet the requirements of the particular flash devices that are used.

The **s_atmel_flash_fs_phy()** function is the key to understanding the interface between the specific physical driver and the file system. This is the only public function in this module and it must be passed to the file system's **f_mountdrive()** API function to initialize the physical driver.

The FS_FLASH structure returned by this call contains all the configuration information about block usage required by the upper layers, as well as a set of pointers to the following interface functions:

- **adf_read_flash()**
- **adf_write_flash()**
- **adf_erase_flash()**
- **adf_verify_flash()** – this is optional.
- **adf_block_copy()** – this is required only if static wear leveling is used.

s_atmel_flash_fs_phy

Use this function to initialize the flash device and also to detect the flash type.

This function gives information to the upper layer about the number of blocks, block sizes, sector size, cache size, and so on.

> **Note:** This is the first call made by the upper layer. It is used to discover the flash device configuration.

Format

```
int s_atmel_flash_fs_phy ( FS_FLASH * flash )
```

Arguments

Argument	Description	Type
flash	The flash structure that needs to be filled.	FS_FLASH *

Return values

Return value	Description
0	Successful execution.
Else	Operation failed.

adf_read_flash

Use this function to read data from flash.

Format

```
static int adf_read_flash (
    void *    data,
    long      block,
    long      blockrel,
    long      datalen )
```

Arguments

Argument	Description	Type
data	A pointer to the data storage area.	void *
block	The zero-based number of the block to be read.	long
blockrel	The relative position in the block to start reading at. This can range from 0 to the block size.	long
datalen	The length of the data to be read in bytes. This is always less than block size and never extends beyond a given block, even if *blockrel* points into the middle of the block.	long

Return values

Return value	Description
0	Successful execution.
Else	Operation failed.

adf_write_flash

Use this function to write data to the flash device.

Format

```
static int adf_write_flash (
    void *    data,
    long      block,
    long      relsector,
    long      size,
    long      relpos )
```

Arguments

Argument	Description	Type
data	A pointer to the source data to be written.	void *
block	The zero-based number of the block to store the data in.	long
sector	The zero-based relative sector number in the block.	long
size	The length of the data to be stored.	long
relpos	The relative position in the block to write the data to.	long

Return values

Return value	Description
0	Successful execution.
Else	Operation failed.

adf_erase_flash

Use this function to erase a block in flash.

Format

```
static int adf_erase_flash ( long block )
```

Arguments

Argument	Description	Type
block	The zero-based number of the block to erase.	long

Return values

Return value	Description
0	Successful execution.
Else	Operation failed.

adf_verify_flash

Use this function to compare written data with the original.

Call this after **adf_write_flash()** to verify written data against the original data. The incoming parameters are the same as for **adf_write_flash()**.

> **Note:** This function is not always necessary. This depends on the particular flash chip and what its datasheet specifies as being necessary to guarantee that a program operation has completed successfully. If this function is not needed, it should return 0.

Format

```
static int adf_verify_flash (
    void *   data,
    long     block,
    long     relsector,
    long     size,
    long     relpos )
```

Arguments

Argument	Description	Type
data	A pointer to the source data to be compared.	void *
block	The zero-based number of the block of data to be compared.	long
relsector	The zero-based relative sector number in the block.	long
size	The length of data to be compared.	long
relpos	The relative position in the block of the data to verify.	long

Return values

Return value	Description
0	Successful execution.
Else	Operation failed.

adf_block_copy

Use this function to copy one block to another block.

> **Note:** Use this only if static wear leveling is in use.

Implement this function to use any features of the target device that may be available to accelerate a block-to-block copy operation. Many devices have features to support block copy. These help to reduce CPU load and improve system performance.

Format

```
static int adf_block_copy (
    long     destblock,
    long     soublock )
```

Arguments

Argument	Description	Type
destblock	The number of the block to copy to.	long
soublock	The number of the block to copy from.	long

Return values

Return value	Description
0	Successful execution.
Else	Operation failed.

6.3 Types and Definitions

FS_FLASH Structure

This is the FS_FLASH structure that the module must set up by using **s_atmel_flash_fs_phy()**.

For more details of the block settings, see NOR Physical Device Usage.

This structure takes the following form:

Element	Type	Description
maxblock	long	The maximum number of blocks that can be used.
blocksize	long	The block size in bytes.
sectorsize	long	The sector size to use.
sectorperblock	long	Sector/block (block size/sector size).
blockstart	long	Where the physical block starts.
descsize	long	The maximum size of FAT+directory+block index.
descblockstart	long	Where to store the first descriptor block.
descblockend	long	Where to store the last descriptor block.
separatedir	long	Not used for DataFlash.
cacheddescsize	long	The size of the descriptor write cache.
cachedpagenum	long	Not used for DataFlash.
cachedescpagesize	long	Not used for DataFlash.

7 Porting the Serial Peripheral Interface

The physical interface to the DataFlash device uses the Serial Peripheral Interface (SPI) standard. A sample SPI driver is provided. Because all platforms implement their SPI interfaces in different ways, you must port the driver to work with your target platform.

To use the sample driver, include the **spi.c** and **spi.h** files in your project.

The higher level uses just five interface functions to the SPI. The ported sample driver must provide a logically identical interface. If your ported driver accurately provides these five interface functions, it will work correctly.

7.1 spi.h Header File

In **spi.h** you must replicate the following macros:

Function	Description
SPI_CS_LO	A macro that sets the SPI ChipSelect low.
SPI_CS_HI	A macro that sets the SPI ChipSelect high.

7.2 spi.c Source Code

The **spi.c** file contains these three interface functions:

Function	Description
spi_init()	Called by the higher level to initialize the interface.
spi_rx_8()	Called by the higher level to read 8 bits of data from the interface.
spi_tx_8()	Called by the higher level to transmit 8 bits of data through the interface.