SafeFLASH File System NAND Drive User Guide

Version 1.50

For use with SafeFLASH File System NAND Drive Versions 2.03 and above

Date: 15-Feb-2018 11:31

Table of Contents

1 System Overview

This chapter contains the fundamental information for this module.

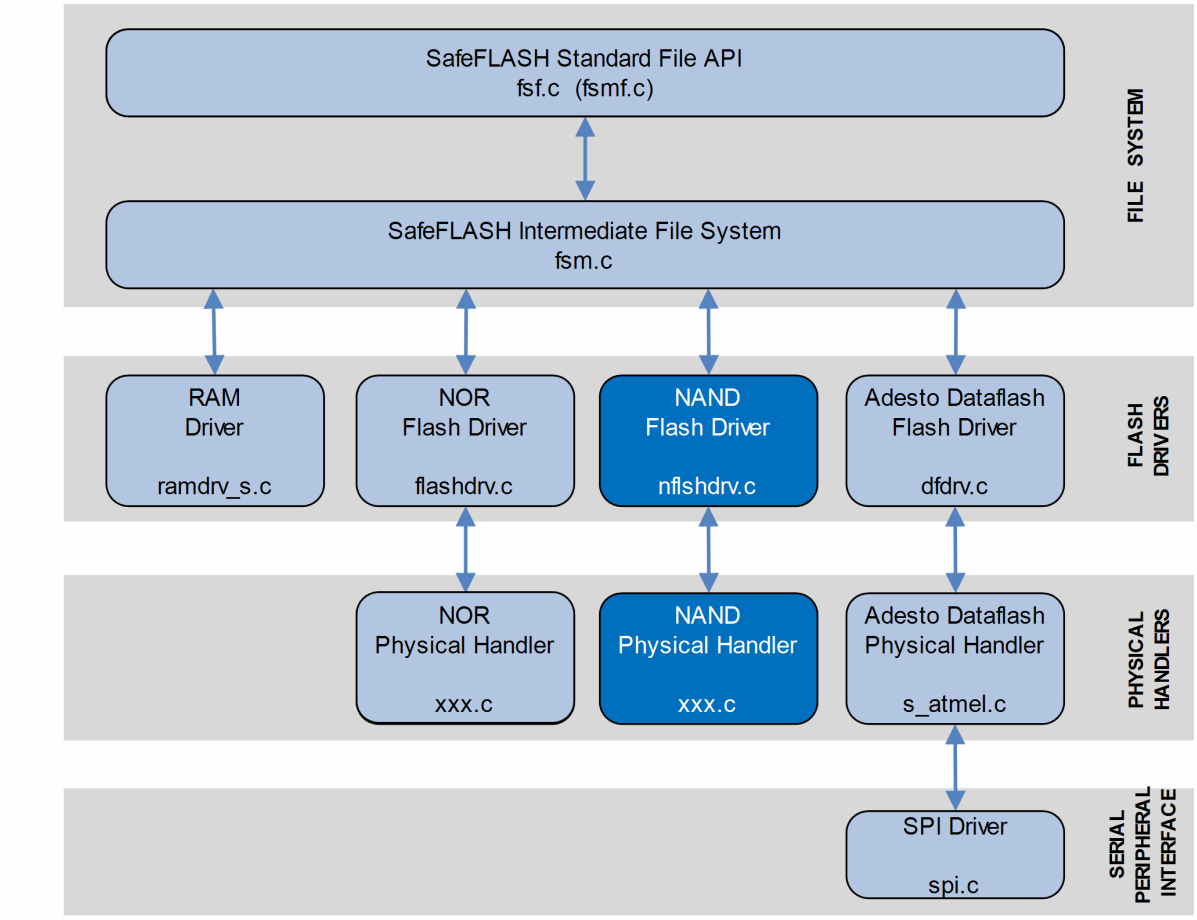
The component sections are as follows:

- Introduction – describes the main elements of the module.
- Feature Check – summarizes the main features of the module as bullet points.
- Packages and Documents – the *Packages* section lists the packages that you need in order to use this module. The *Documents* section lists the relevant user guides.
- Change History – lists the earlier versions of this manual, giving the software version that each manual describes.

1.1 Introduction

This guide is for those who wish to implement a NAND drive for HCC's SafeFLASH file system.

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 NAND 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 NAND 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 1/2/4 devices in parallel).

Note:

- HCC Embedded has a range of physical handlers available to make the porting process as simple as possible. HCC Embedded 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.
- The SafeFLASH file system was previously known as EFFS-STD. All references to STD in the code are historical and refer to the file system’s original name.

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 NAND are as follows:

- Supports all NAND flash types.
- Supports MCU/NAND controllers.
- Supports static and dynamic wear leveling.
- Supports bad block management.
- Supports the Error Correction Codes (ECC) algorithm.
- Porting is easy for all known device types.
- Provides a sample driver with a porting description.

1.3 Packages and Documents

Packages

The table below lists the packages that you need in order to use this module and other packages that HCC can provide:

Package	Description
hcc_base_doc	This contains the two guides that will help you get started.
fs_safe	The SafeFLASH base package required by the NAND package.
fs_safe_nand	The SafeFLASH NAND package described in this document.
fs_safe_nand_drv_sample	Sample drivers available to help with development.
fs_safe_nand_drv_xxx	Reference drivers available to help with development.

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 NAND Drive User Guide

This is this document.

Other HCC SafeFLASH Guides

These describe other SafeFLASH components:

- *HCC SafeFLASH File System RAM Drive User Guide* – documents the SafeFLASH system for RAM.
- *HCC SafeFLASH File System NOR Drive User Guide* – documents the SafeFLASH system for NOR flash.
- *HCC SafeFLASH for Adesto DataFlash Drives User Guide* – documents the SafeFLASH system for Adesto® DataFlash.

1.4 Change History

This section describes past changes to this manual.

- To view or download earlier manuals, see File System PDFs.
- For the history of changes made to the package code itself, see History: fs_safe_nand.

The current version of this manual is 1.50. The full list of versions is as follows:

Manual version	Date	Software version	Reason for change
1.50	2018-02-15	2.03	Improved/corrected description of FS_FLASH structure.
1.40	2017-10-10	2.03	Change to **fs_mount_nandflashdrive()**.
1.30	2017-08-30	2.02	Corrected *Packages* list.
1.20	2017-06-26	2.02	New *Change History* format.
1.10	2015-12-21	2.01	Added *Fail-safety* section.
1.00	2014-08-21	1.02	First online version.

2 Source File List

This section describes all the source code files included in the system. These files follow the HCC Embedded 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_nand.h** is the only file that should be included by an application using this module. For details of the API functions, see Application Programming Interface.

2.2 Configuration File

The file **src/config/config_safe_nand.h** contains the single configurable parameter. Configure this as required. For details of the option, see Configuration Option.

2.3 Source Files

The NAND flash interface to the file system requires the following files. The two files shown below are in **src/safe-flash/nand**:

File	Short description	Description
nflshdrv.c	Device-independent flash control layer.	This module provides a single clean interface between the physical chip and the intermediate file system. This module gets information about the configuration of the underlying flash chip from the physical chip handler module and builds a controller based on that information. This module also performs the wear leveling control for the device. **Note**: Normally this module does not require modification. If modification is required, we strongly recommend that you contact HCC Embedded about your requirements.
nflshdrv.h	Header file.	NAND flash driver header.

2.4 Physical Chip Handler

The physical chip handler module is located in the relevant sample driver folder in the **safe_nand_drv_sample** package. These folders are in **src/safe-flash/nand/phy/sample**.

The module depends on the specific flash device and its configuration. Relevant data are the manufacturer, chip size, number of interface bits (8, 16, or 32), and the number and arrangement of the chips (parallel or serial). All of these factors influence the code in this module.

2.5 Version File

The file **src/version/ver_safe_nand.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 Option

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

FS_NAND_MAXFILE

The maximum number of files that can be open simultaneously. The default value is 4.

4 Introduction to NAND Flash

SafeFLASH allows easy integration of all standard flash devices.

Two basic types of flash devices are generally available, NAND and NOR. These have quite distinct physical characteristics and thus require quite different handling, but they do have the following basic properties in common:

- They are designed for non-volatile storage of code and data.
- An area must be erased before it is written to. Erasing changes all erased bits to 1. Programming consists of changing 1s to 0s. To change a 0 to a 1, an erase operation must be performed.
- They are all divided into erase units (blocks). In order to erase any part, the whole block must be erased.
- Data areas wear out after a number of erase cycles. The guarantee for the number of successful erase cycles varies among chip types. Therefore, it is important for any file system that uses flash to manage the wearing of the flash. This is done by avoiding overuse of any one block.

This guide only covers NAND flash.

4.1 Features of NAND/AND Flash

NAND flash (also AND) is a newer flash chip technology than NOR. The primary difference is:

- NAND can store approximately four times as much data as NOR flash for about the same price.
- NAND has much faster erase and write times, so is a superior choice for applications that require regular data storage.

However, there is a price to be paid for the improved performance:

- Data cannot be accessed via a standard address/data bus. Instead commands must be sent to set the address and the data can then be read/written sequentially.
- Chips come from the factory with a number of bad blocks that can never be used.
- Bits may flip unexpectedly (but this is handled; see below).

Because of these complications, NAND chips are designed with some additional features:

- Each block is divided into a number of read/write pages (typically 512, 2048, or 4096 bytes in size).
- Each page has an associated "spare" area that is used to store error correction and block management information. If this area is used effectively, the general performance and reliability of the devices is very high.

The NAND flash driver contains the necessary spare area management and fast Error Correction Codes (ECC) algorithm.

4.2 SLC and MLC NAND Flash

There are two categories of NAND flash:

- Single Layer Cell (SLC) – generally characterized by having a typical reliable erase/write cycle count of 100K or more, as long as an ECC which supports 1 bit error correction per 512 bytes of data is used.
- Multi Layer Cell (MLC) – generally characterized by having a typical reliable erase/write cycle count of 5000, as long as an ECC which supports 3 bit error correction per 512 bytes of data is used.

In general:

- With SLC flash hardware, support in the form of an integrated NAND flash controller on the microprocessor can help to reduce the load on the CPU. External NAND flash controllers are also available. HCC offers the logic for an external NAND flash controller, realized in VHDL, which is suitable for use on many programmable logic type devices.
- With MLC flash hardware, support is required because the resources required to run the ECC operation are too high a burden on most embedded systems. Typically this hardware support involves an integrated NAND flash controller on a microprocessor which supports MLC flash.

Because the ECC algorithm for SLC flash is much simpler, it can also be realized in software. Several sample drivers that provide software ECC algorithms for SLC flash are supplied in the SafeFLASH package.

5 System Features

This chapter covers the following:

- Other Media Types – explains that any device that can emulate a logical block arrangement can be used as a storage medium for SafeFLASH.
- Maximum Number of Files – explains how the maximum number of file/directory entries that can be made on a file system is calculated.
- Timeouts – covers scheduling of other operations while waiting for flash operations to complete.
- Write Cache – shows how to define a write cache for the driver.
- Physical Device Usage – shows how to assign the three types of block to the device.

5.1 Other Media Types

The SafeFLASH design is based on the concept of a storage device with a logical block arrangement. Because of this, any device that can emulate a logical block arrangement can be used as a storage medium for SafeFLASH.

> **Note:** SafeFLASH does not support removable media and is not recommended for arrays of flash greater than 4GB. For removable media and very large arrays we recommend using the HCC FAT or SafeFAT system, with SafeFTL where NAND flash is required.

5.2 Maximum Number of Files

The maximum number of file/directory entries that can be made on a file system is restricted. This number may be calculated from the formula:

MaxNum Entries = (Descsize - (maxblock*((sectorperblock*2) + 6)))/32

So:

- If more files are required (without using the **separatedir** setting in the FS_FLASH structure), either increase the sector size (creating more space in the descriptor blocks), or choose a larger descriptor block.
- If fewer files are required, decrease the sector size or allocate smaller descriptor blocks.

If **separatedir** is used, the maximum number of file and directory entries is given by the formula:

MaxNum Entries = (Blocksize/32){}separatedir*

> **Note**: If files with long filenames are used, the number of files that can be stored is reduced.

5.3 Timeouts

Flash devices are normally controlled by hardware control signals. As a result there is no explicit need for any timeouts to control exception conditions. However, some operations on flash devices are relatively slow and it is often worth scheduling other operations while waiting for them to complete (for example, a NAND flash erase takes two milliseconds).

5.4 Write Cache

You can define a write cache for the driver. Using the write cache means that in most cases only changes to the descriptor block are stored in the flash device, thus improving the performance of the system (there are fewer erases and writes), and reducing wear on the system.

To use the write cache, **WriteVerifyPage()** must be present. If this function does not exist, write caching cannot proceed.

Additionally the following parameters in the FS_FLASH structure must be set up by using **fs_phy_nand_xxx()**:

Parameter	Description
cachedpagesize	This should be equal to the page size of the device.
cachedpagenum	Number of pages in the cache, which must equal the number of pages in an erasable block.

If either of these is set to zero, write caching is not used.

5.5 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.
- You can only write to an erased location.
- You cannot erase anything smaller than a block.

You can assign three types of block to the device:

- **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.

The following sections describe how to assign these.

Reserved Blocks

Blocks can be reserved for private usage without restriction. For example, if a particular physical device has 1024 erasable blocks and you want to reserve 256 blocks from the beginning for private use, you could use the **fs_phy_nand_xxx()** function to set the following values in the FS_FLASH structure:

- *maxblock* (number of blocks for use by the file system) = 768
- *blockstart* (first file storage block) = 256

> **Note:** Do not access reserved blocks while the file system is accessing the device. Operations must be performed atomically; that is, one command must be completed on the device before another is started.

FS_NAND_RESERVEDBLOCK Definition

The file system needs to have a reserve of several blocks to ensure its smooth operation for all eventualities. Set this in the file **nflshdrv.h**. The default value is 12. It must be at least 3 plus the number of separate directory blocks. Using a larger value than this is recommended to ensure that, if bad blocks develop, others are available for use.

> **Note:** At a certain point of usage all NAND blocks fail. Once the number of failed blocks becomes too large to maintain a stable file system, the system returns F_ERR_UNUSABLE and becomes a read-only file system.

File System Blocks

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

maxblock

The number of erasable blocks 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.

sectorsize

The sector size. Each block is divided (by 2^n) into a number of sectors. This number is the smallest usable unit in the system and thus represents the minimum file storage area.

sectorperblock

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

sectorperblock * sectorsize = blocksize

blockstart

The logical number of the first block that may be used by the file system.

Descriptor Blocks

These blocks contain critical information about the file system: block allocation information, wear information, and file/directory information. They are allocated automatically from the file system blocks. Set the following parameters up by using the **fs_phy_nand_xxx()** function:

descsize

This is the size of a descriptor block. Since all blocks are the same size on NAND flash devices, this is the same as the block size.

seperatedir

The maximum number of separate blocks that will be allocated for directory entries, a number ranging from 0 to 4. If this is set to a non-zero value, the directory entries are given blocks that are separate from the file system. This allows a much larger number of files to be stored in the file system.

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 Management Functions

The functions are the following:

Function	Description
fs_mount_nandflashdrive()	Mounts a NAND flash drive.
fs_getmem_nandflashdrive()	Returns the memory required for the driver in bytes.

fs_mount_nandflashdrive

Use this function to mount a NAND flash drive.

Format

```
extern int fs_mount_nandflashdrive (
    void *        vol_dsc,
    FS_PHYGETID   phyfunc )
```

Arguments

Arguments	Description	Type
vol_dsc	A pointer to the volume's volume descriptor.	void *
phyfunc	A pointer to a physical driver function for the desired device that is called by the generic mount function to get information about how to use the device. See the *HCC SafeFLASH File System User Guide* for full details.	FS_PHYGETID

Return Values

Return value	Description
0	Successful execution.
1	Operation failed.

fs_getmem_nandflashdrive

Use this function to return the memory required for the driver in bytes.

Format

```
extern int fs_getmem_nandflashdrive ( FS_PHYGETID phyfunc )
```

Arguments

Arguments	Description	Type
phyfunc	A pointer to a physical driver function for the desired device that is called by the generic mount function to get information about how to use the device. See the *HCC SafeFLASH File System User Guide* for full details.	FS_PHYGETID

Return Values

Return value	Description
0	Operation failed. This may be because the flash could not be identified.
Else	The memory required.

6.2 Physical Layer 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 **fs_phy_nand_xxx()** 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()** 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 interface function pointers to the following NAND interface functions:

The other functions are the following:

Function	Description
ReadFlash()	Reads data from flash.
WriteFlash()	Writes data to the flash device.
EraseFlash()	Erases a block in flash.
VerifyFlash()	Compares written data with the original. (Only required if verification is the method of checking that the device was correctly written.)
WriteVerifyPage()	Verifies that a page of data within the flash matches a buffer containing the written data. (Only required if verification is the method of checking that the device was correctly written.)
CheckBadBlock()	Used at file system initialization to determine which blocks are bad.
GetBlockSignature()	Gets the previously stored block signature data set by WriteFlash().
BlockCopy()	Copies one block to another block. (Only use this if static wear leveling is used.)

All these functions require subroutine calls to do their work, as described in Subroutine Descriptions and Notes for the Sample Driver.

fs_phy_nand_xxx

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 fs_phy_nand_xxx ( FS_FLASH * flash )
```

Arguments

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

Return values

Return value	Description
0	Successful execution.
Else	See Error Codes.

ReadFlash

Use this function to read data from flash.

Format

```
int ReadFlash (
    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 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 data to read. 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	See Error Codes.

WriteFlash

Use this function to write data to the flash device.

Format

```
int WriteFlash (
    void *   data,
    long     block,
    long     relsector,
    long     size,
    long     signdata )
```

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 data in.	long
relsector	The zero-based relative sector number in the block.	long
size	The length of data to be stored.	long
signdata	Block signature data. After this call, this can be obtained by using **GetBlockSignature()**.	long

Return values

Return value	Description
0	Successful execution.
Else	See Error Codes.

EraseFlash

Use this function to erase a block in flash.

Format

```
int EraseFlash ( 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	See Error Codes.

VerifyFlash

Use this function to compare written data with the original.

Call this after **WriteFlash()** to verify written data against the original data.

Note:

- This is required only when verification is the desired method of ensuring that the device has been correctly written.
- To decide whether or not to use a verify function, refer to the device datasheet. If, for example, ECC is being used and the guaranteed reliability is sufficient for your requirements, verification may be omitted. **This has a significant performance benefit**.
- To omit this function, set the VerifyFlash element of the FS_FLASH Structure structure to NULL when you set it up in your driver initialization function.

Format

```
int VerifyFlash (
    void *   data,
    long     block,
    long     relsector,
    long     size,
    long     signdata )
```

Arguments

Argument	Description	Type
data	A pointer to the source data to be compared.	void *
block	The zero-based number of the block where data are stored.	long
relsector	The zero-based number of the relative sector in the block.	long
size	The length of data to compare.	long
signdata	Block signature data.	long

Return values

Return value	Description
0	Successful execution and no difference between flash and buffer content.
Else	See Error Codes.

WriteVerifyPage

Use this function to verify that a page of data within the flash matches a buffer containing the written data.

Call this function after the write caching mechanism writes a page of data to the flash.

Note:

- This is required only when verification is the desired method of ensuring that the device has been correctly written.
- To decide whether or not to use a verify function, refer to the device datasheet. If, for example, ECC is being used and the guaranteed reliability is sufficient for your requirements, verification may be omitted. **This has a significant performance benefit**.
- To omit this function, set the WriteVerifyPage element of the FS_FLASH Structure structure to NULL when you set it up in your driver initialization function.

Format

```
int WriteVerifyPage (
    void *   data,
    long     block,
    long     startpage,
    long     pagenum,
    long     signdata )
```

Arguments

Argument	Description	Type
data	A pointer to the data to be written and verified.	void *
block	The number of the block to check.	long
startpage	The start page number in the block.	long
pagenum	The number of pages to be written.	long
signdata	The signature data for the block. After this call, this can be obtained by using **GetBlockSignature()**.	long

Return values

Return value	Description
0	Successful execution.
Else	See Error Codes.

CheckBadBlock

Use this function at file system initialization to determine which blocks are bad.

If the flash device contains invalid blocks, this function registers these so that the file system does not use them.

The higher level calls this function for all used blocks. The method used to check for bad blocks is device-dependent.

Format

```
int CheckBadBlock ( long block )
```

Arguments

Argument	Description	Type
block	The number of the block to check.	long

Return values

Return value	Description
0	The block is useable.
1	The block is bad or non-valid.

GetBlockSignature

Use this function to get the previously stored block signature data set by **WriteFlash()**.

Format

```
long GetBlockSignature ( long block )
```

Arguments

Argument	Description	Type
block	The number of the target block.	long

Return values

Return value	Description
Value	Signature data.

BlockCopy

Use this function to copy one block to another block.

> **Note:** Only use this function 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

```
int BlockCopy (
    long    destblock,
    long    soublock )
```

Arguments

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

Return values

Return value	Description
0	Successful execution.
Else	See Error Codes.

6.3 Types and Definitions

This section describes the *FS_FLASH* structure.

FS_FLASH Structure

This is the FS_FLASH structure that you must set up by using fs_phy_nand_xxx. For more details of the block settings, see Physical Device Usage.

Element	Type	Description
maxblock	long	Maximum number of blocks that can be used.
blocksize	long	Block size in bytes.
sectorsize	long	Sector size to use.
sectorperblock	long	Sector/block (block size/sector size).
blockstart	long	The logical number of the first block in the created partition. This number is returned by the driver at initialization. The driver must choose how to map the logical block numbers it provides to the file system to the physical blocks on the target flash.
descsize	long	Maximum size of descriptor: FAT+directory+block index.
descblockstart	long	Not used for NAND.
descblockend	long	Not used for NAND.
separatedir	long	Directories use separate block from FAT.
cacheddescsize	long	Not used for NAND.
cachedpagenum	long	Number of pages in cache.
cachedpagesize	long	Size of pages in cache. Note that *cachedpagenum * cachedpagesize = blocksize*
ReadFlash	FS_PHYREAD	Read content function.
EraseFlash	FS_PHYERASE	Erase a block function.
WriteFlash	FS_PHYWRITE	Write content function.
VerifyFlash	FS_PHYVERIFY	Verify content function.
CheckBadBlock	FS_PHYCHECK	Check whether block is bad function.
GetBlockSignature	FS_PHYSIGN	Get block signature data function.

Element	Type	Description
WriteVerifyPage	FS_PHYCACHE	Write and verify page function.
BlockCopy	FS_PHYBLKCPY	Accelerated block copy function.
chkeraseblk	unsigned char *	When the flash driver allows pre-erasing of blocks (that is, performing erase operations while the system is less busy), the driver must provide this buffer to the flash layer so that it can check whether a block is ready to be erased.
erasedblk	unsigned char *	When the flash driver allows pre-erasing of blocks (that is, performing erase operations while the system is less busy), the driver must provide this buffer to the flash layer so that it can check whether a block has already been erased.

6.4 Error Codes

The table below lists all the error codes that may be generated by API calls to HCC's file systems. Please note that only a few of these error codes relate specifically to NAND flash.

Error	Value	Meaning
F_NO_ERROR	0	Successful execution.
F_ERR_INVALIDDRIVE	1	The specified drive does not exist.
F_ERR_NOTFORMATTED	2	The specified volume has not been formatted.
F_ERR_INVALIDDIR	3	The specified directory is invalid.
F_ERR_INVALIDNAME	4	The specified file name is invalid.
F_ERR_NOTFOUND	5	The file or directory could not be found.
F_ERR_DUPLICATED	6	The file or directory already exists.
F_ERR_NOMOREENTRY	7	The volume is full.
F_ERR_NOTOPEN	8	The file access function requires the file to be open.
F_ERR_EOF	9	End of file.
F_ERR_RESERVED	10	Not used.
F_ERR_NOTUSEABLE	11	Invalid parameters for **f_seek()**.
F_ERR_LOCKED	12	The file has already been opened for writing /appending.
F_ERR_ACCESSDENIED	13	The necessary physical read and/or write functions are not present for this volume.
F_ERR_NOTEMPTY	14	The directory to be moved or deleted is not empty.
F_ERR_INITFUNC	15	No init function is available for a driver, or the function generates an error.
F_ERR_CARDREMOVED	16	The card has been removed.
F_ERR_ONDRIVE	17	Non-recoverable error on drive.
F_ERR_INVALIDSECTOR	18	A sector has developed an error.
F_ERR_READ	19	Error reading the volume.
F_ERR_WRITE	20	Error writing file to volume.

Error	Value	Meaning
F_ERR_INVALIDMEDIA	21	Media not recognized.
F_ERR_BUSY	22	The caller could not obtain the semaphore within the expiry time.
F_ERR_WRITEPROTECT	23	The physical medium is write protected.
F_ERR_INVFATTYPE	24	The type of FAT is not recognized.
F_ERR_MEDIATOOSMALL	25	Media is too small for the format type requested.
F_ERR_MEDIATOOLARGE	26	Media is too large for the format type requested.
F_ERR_NOTSUPPSECTORSIZE	27	The sector size is not supported. The only supported sector size is 512 bytes.
F_ERR_UNKNOWN	28	An unspecified error has occurred.
F_ERR_DRVALREADYMNT	29	The drive is already mounted.
F_ERR_TOOLONGNAME	30	The name is too long.
F_ERR_NOTFORREAD	31	Not for read.
F_ERR_DELFUNC	32	The delete drive driver function failed.
F_ERR_ALLOCATION	33	**psp_malloc()** failed to allocate the required memory.
F_ERR_INVALIDPOS	34	An invalid position is selected.
F_ERR_NOMORETASK	35	All task entries are exhausted.
F_ERR_NOTAVAILABLE	36	The called function is not supported by the target volume.
F_ERR_TASKNOTFOUND	37	The caller's task identifier was not registered. This is normally because **f_enterFS()** has not been called.
F_ERR_UNUSABLE	38	The file system has become unusable. This is normally a result of excessive error rates on the underlying media.
F_ERR_CRCERROR	39	A CRC error has been detected on the file.
F_ERR_CARDCHANGED	40	The card that was being accessed has been replaced with a different card.

6.5 Subroutine Descriptions and Notes for the Sample Driver

This section contains a complete list of subroutines, with descriptions of how to use them. Implementation for a particular device may vary from that documented here.

NANDcmd(cmd: long)

This subroutine sends a command to NAND flash.

NANDaddr(addr: long)

This subroutine sends an address to NAND flash.

NANDwaitrb()

This subroutine waits until RB (ready/busy) goes high on NAND flash.

ReadPage(pagenum: long)

This subroutine sends a command sequence to read a page.

When using this:

1. Read the whole page of data and calculate the ECC.
2. Get the saved ECC from the NAND flash spare area.
3. If ECC calculation is needed, perform ECC checking.

WritePage(data: ptr, pagenum: long, size: long)

This subroutine copies original data into a temporary 32 bit aligned buffer.

When using this:

1. Send a command sequence for programming a page to NAND flash.
2. Program a whole page and calculate the ECC.
3. Write the ECC into the NAND flash spare area.
4. Check whether programming was successful; if not, return with an error.

ReadFlash(data: ptr, block: long, blockrel: long, datalen: long)

When using this subroutine:

1. Calculate **pagenum**.
2. Find the starting page from **blockrel**.
3. ReadPage(**pagenum**).

4. Check whether data need to be copied (for example, for alignment reasons) and, if required, copy these.
5. ReadPage(**pagenum**) until **datalen**=0.

EraseFlash(block: long)

When using this subroutine:

1. Calculate **pagenum**.
2. Send a command sequence to erase a block to NAND flash.
3. Wait until erasing is finished.
4. Check whether the erase was successful; if not, return with an error.

WriteFlash(data: ptr, block: long, relsector: long, len: long, sdata: long)

When using this subroutine:

1. Calculate **pagenum**.
2. WritePage(**pagenum**++) until size=0 or any error.
3. Signal an error or return having written successfully.

VerifyFlash(data: ptr, block: long, relsector: long, len: long, sdata: long)

When using this subroutine:

1. Calculate **pagenum**.
2. ReadPage(**pagenum**++) until **len**=0.
3. Compare pages with the original data; if there are any differences, return with an error.

CheckBadBlock(block: long)

Use this subroutine to determine whether or not the given block is bad. When using this:

1. Calculate **pagenum**.
2. Send a read spare area command to NAND flash.
3. Check the sixth word; if it is not 0xFFFFFFFF return with an error, otherwise return 0 (OK).

GetBlockSignature(block: long)

This subroutine reads signature data from a block.

fs_phy_nand_xxx (flash: struct)

Use this subroutine to set function pointers for the driver. When using this:

1. Get the device ID and manufacturer ID from NAND flash.
2. Compare all supported devices/manufacturers and fill the flash structure with corresponding data (size, sectors, and block information).
3. If the device is not found, return with an error.

7 The Flash Driver Test Suite

Use the test suite to exercise the flash drivers and ensure that everything works correctly. This code tests your ported flash driver in isolation, to ensure that it is ported correctly and is stable.

The test program requires the functions defined and implemented (as samples) in the file **testdrv_s.c**. This is part of the **fs_safe** base package and is located, with its header file **testdrv_s.h**, in the folder **fs_safe_xxx_xx/hcc/src/safe-flash/test**.

Port these functions to your system. See the comments and simple code for reference.

To use the test program:

1. Include **testdrv_s.c** and **testdrv_s.h** in your test project.
2. Call the following to execute the test code:

```
void f_dotestdrv ( FS_PHYGETID phyfunc )
```

Errors in the execution of this test indicate that there is an error in the implementation of the driver. Contact support@hcc-embedded.com if you need further advice.

8 Integration

This section describes all aspects of the module that require integration with your target project. This includes porting and configuration of external resources.

8.1 PSP Porting

The Platform Support Package (PSP) is designed to hold all platform-specific functionality, either because it relies on specific features of a target system, or because this provides the most efficient or flexible solution for the developer. For full details of its functions and macros, see the *HCC Base Platform Support Package User Guide*.

The module makes use of the following standard PSP function:

Function	Package	Element	Description
psp_memset()	psp_base	psp_string	Sets the specified area of memory to the defined value.