SafeFLASH File System NOR Drive User Guide

Version 1.60

For use with SafeFLASH File System NOR Drive Versions 2.03 and above

Date: 10-Oct-2017 12:59

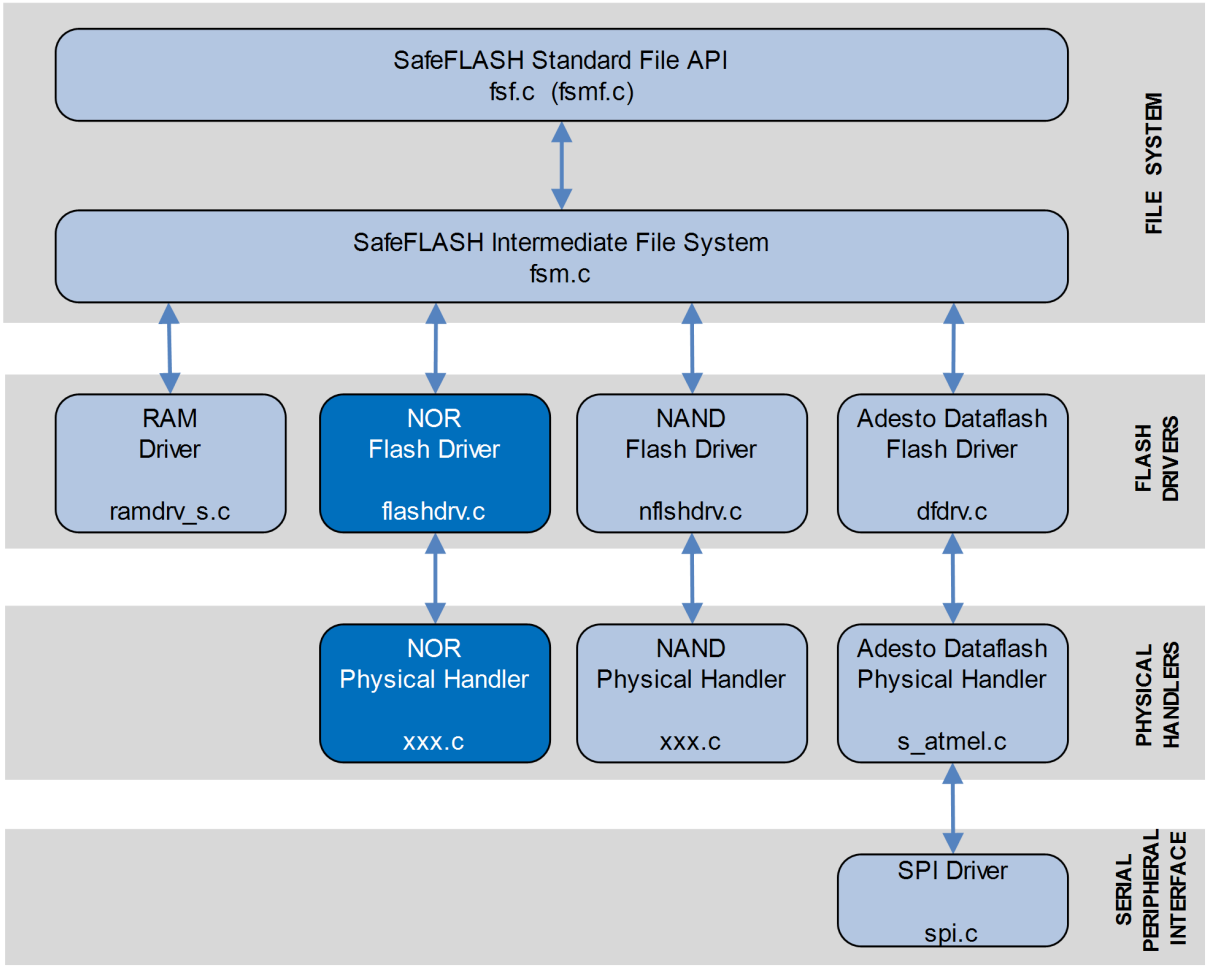
Table of Contents

1 System Overview

1.1 Introduction

This guide is for those who want to implement a NOR drive for HCC's SafeFLASH file system.

The following diagram shows 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 NOR 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 NOR 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 NOR flash are as follows:

- Support for all NOR flash types.
- Easy porting for all known device types.
- Static and dynamic wear leveling.
- Bad block management.
- Sample driver available with 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.
fs_safe_nor	The SafeFLASH NOR package described in this document.
fs_safe_nor_drv_sample	A range of sample drivers available to help with development.
fs_safe_nor_drv_xxx	A range of reference drivers available to help with development.

Documents

For an overview of HCC file systems and guidance on choosing a file system, see Product Information on 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 NOR 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 NAND Drive User Guide* – documents the SafeFLASH system for NAND 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_nor.

The current version of this manual is 1.60. The full list of versions is as follows:

Manual version	Date	Software version	Reason for change
1.60	2017-10-10	2.03	Change to **fs_mount_flashdrive()**.
1.50	2017-08-31	2.02	Corrected *Packages* list.
1.40	2017-06-26	2.02	New *Change History* format.
1.30	2017-03-27	2.02	Added function group descriptions to API.
1.20	2015-12-22	2.01	Added *Change History* and *API Functions* sections, other small changes.
1.10	2014-08-20	1.02	Reorganized *System Overview*.
1.00	2014-08-06	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_nor.h** is the only file that should be included by an application using this module. It defines the **fs_mount_flashdrive()** and **fs_getmem_flashdrive()** functions. For more details, see API Functions.

2.2 Configuration File

The file **src/config/config_safe_nor.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 Option.

2.3 Source Files

The NOR flash interface to the file system requires the following files which are in **src/safe-flash/nor**:

File	Short description	Description
flashdrv.c	Device-independent flash control layer.	The **flashdrv.c** module provides a single clean interface between the physical chip and the intermediate file system. It gets information about the configuration of the underlying flash chip and the interface routines to call from the physical chip handler module. It builds a controller based on that information. This module also performs the wear level control for the device. **Note**: Normally this module does not require modification. If changes are required, we strongly recommend that you contact HCC Embedded about your requirements.
flashdrv.h	Header file.	NOR flash driver header.

2.4 Physical Chip Handler

The physical chip handler module is located in the relevant sample driver folder in the **safe_nor_drv_sample** package. These folders are in **src/safe-flash/nor/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_nor.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_nor.h**.

FS_NOR_MAXFILE

The maximum number of files that can be open at the same time. The default is 4.

4 NOR Flash Explained

SafeFLASH allows the easy integration of all standard flash devices.

4.1 Flash Types

Two basic types of flash devices are generally available, NOR and NAND. 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 all wear out after a number of erase cycles. The guaranteed 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. HCC's wear leveling algorithms manage this efficiently.

This guide only covers NOR flash.

4.2 Features of NOR Flash

NOR flash has been the cornerstone of non-volatile memory in embedded systems for many years. NOR has two basic characteristics:

- It stores data in a non-volatile way.
- It can be accessed directly from an address bus (this is termed "random access") so can be used to run code.

The main drawback of NOR flash is that the erase/write time is very long. Even if small amounts of data are written, an erase may cause a delay of as much as two seconds. Careful design of the SafeFLASH file system has ensured that this kind of behavior is minimized, but in certain cases it is unavoidable.

5 System Features

5.1 Sectors and File Storage

The blocks of the file storage section of the file system are sub-divided into sectors of equal size. These sectors are the minimum writeable area on the device and the minimum area taken up by a file. For file systems with many small files, it is advantageous to keep the sector size small to maximize the number of files that may be stored. An additional benefit is that if the files are small, many more can be written before a block erase is required.

For example, if there is one sector per block, a block must be erased for every file. However, if there are 32 sectors per block then 32 small files can be written before it is necessary to erase another block.

There is, however, a balance to be struck between the maximum number of files and the number of sectors in the system.

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

A descriptor block must contain:

- Block descriptors (6 bytes each).
- Sector descriptors (2 bytes each).
- File descriptors (32 bytes each).

Thus the maximum number of files allowed in the system may be given by this formula:

Max Files < ((DescSize-DescCache) - 6*Maxblock – 2*Maxblock*sectorperblock)/32

You need to find a balance between having many sectors per block and allowing enough space in the descriptor for the required number of file descriptors. If you cannot find a balance, consider larger descriptor blocks, but this comes with a penalty: the erase time of the frequently-used descriptor blocks increases.

Again, use the **FSmem.exe** utility to calculate the capabilities of a particular file system on the basis of input configuration information.

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

5.2 Physical Device Usage

You must make some decisions about how to use the flash device. Note the following:

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

These factors 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 usage of the blocks and make it easier to derive 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 a particular purpose. This is configured in the FS_FLASH structure and returned to the file system by the **fs_phy_nor_xxx()** 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 performed by the **GetBlockAddr()** function).

You can assign three types of block to the device:

- **Reserved blocks** – for processes other than the file system. An example is booting.
- **File system blocks** – to store 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. They provide worked examples.

Reserved Blocks

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

Reserved blocks may be accessed by using the **GetBlockAddr()** function and also by selecting the physical block numbers to use and ensuring that these are not specified in the descriptor and file system usage described below.

Note: To ensure interoperability, take care in accessing reserved blocks and pay attention to the device specification. Some devices allow an erase operation to be performed while another block is being read, others have different rules. A general rule is to use either the file system or the reserved sectors at any one time, not both. In any case, clear understanding of specific devices is needed.

File System Blocks

Allocate as many of these as required for file storage. Set the following parameters up by using the **fs_phy_nor_xxx()** function 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 is 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* where the flash chip has erasable units of different sizes.

sectorsize

The sector size. Each block is divided into a number of sectors. The *sectorsize* is the smallest usable unit in the system so 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 block that may be used for file storage. This is the logical number used when the **GetBlockAddr()** function is called.

Descriptor Blocks

(Also see Sectors and File Storage.)

These blocks contain critical information about the file system, including block allocation, wear, caching, 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 file system performance.

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 the **fs_phy_nor_xxx()** function to create an FS_FLASH structure:

descsize

This is the size of a descriptor block. It is the maximum size of FAT+directory+block index.

> **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 this is set to zero the descriptor write cache method is not used. The descriptor write cache is an efficient method of updating the changes in the descriptor, since the whole descriptor need not be rewritten, while the 100% power-fail safe characteristics of the system are 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 to check the effect of this.

Example 1

Here the target flash device has 35 erasable blocks (1x16KB, 2x8KB, 1x32KB, 31x64KB) and the user wants to reserve blocks 0 and 3 for private usage. This table shows a possible configuration:

Block	Size	Description
BLOCKSIZE	64K	Size of file storage blocks.
BLOCKSTART	4	Logical first file storage block (4-18 used).
MAXBLOCKS	31	Number of blocks for use by file storage.
DESCSIZE	8K	Descriptor size.
DESCBLOCKSTART	1	Logical first descriptor block number.
DESCBLOCKS	2	Number of descriptor blocks.
DESCCACHE	2K	Set a write cache of 2KB.

The table below shows how the physical/logical blocks are arranged:

Physical Block Number	Physical Block Size	Logical Block Number	Usage
0	16KB	0	Reserved block.
1	8KB	1	Descriptor block.
2	8KB	2	Descriptor block.
3	32KB	3	Reserved block.
4…34	64KB	4-34	File storage blocks.

Thus the **GetBlockAddr** algorithm for this could be:

```
unsigned long GetBlockAddr (long block, long relsector)
{
    if(block==0)        /* free/unused block */
        return(0);
    if(block==1)        /* descriptor block */
        return(16K);
    if(block==2)        /* descriptor block */
        return(16K+8K);
    if(block==3)        /* free/unused block */
        return(16K+8K+8K);

    /* file system blocks */
    return(16K+8K+8K+32K+(block-BLOCKSTART)*BLOCKSIZE)+ (relsector*SECTORSIZE));
}
```

Example 2

This example uses a flash device with 512*128KB erasable blocks. A minimum of two erasable blocks must be used for descriptors. These blocks are quite large. Therefore it is a good idea to define a large part of this for a write cache and in this example we create a 32KB cache.

Using a cache of this size has two advantages:

- The number of required erases is reduced.
- The wear on the device is reduced.

We then decide to use the remaining 510 physical blocks for file system storage. So a configuration could look like this:

Block	Size	Description
BLOCKSIZE	128K	Size of file storage blocks.
BLOCKSTART	0	Logical first file storage block (0 – 509 are used).
MAXBLOCKS	510	Number of blocks for use by file storage.
DESCSIZE	128K	Descriptor size (4 per physical block).
DESCBLOCKSTART	510	Logical first descriptor block number.
DESCBLOCKS	2	Number of descriptor blocks.
DESCCACHE	32K	Size of write descriptor cache.

The table below shows how the physical/logical blocks are arranged:

Physical Block Number	Physical Block Size	Logical Block Number	Usage
0-509	64KB	0-509	File storage blocks
510-511	64KB	510-511	Descriptors

The **GetBlockAddr** algorithm in the driver for the above configuration could be modified to:

```
unsigned long GetBlockAddr (long block, long relsector)
{
    return((block*BLOCKSIZE)+(relsector*SECTORSIZE));
}
```

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

The API functions are the following:

Function	Description
fs_getmem_flashdrive()	Returns the memory required for the driver in bytes.
fs_mount_flashdrive()	Called by **f_mountdrive()** to mount and map a new drive.

fs_getmem_flashdrive

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 a NOR Drive for details.

Format

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

Arguments

Argument	Description	Type
phyfunc	The **fs_phy_nor_xxx()** function of the physical chip driver to be mounted (for example, **fs_phy_nor_29lvxxx()** for AMD flash).	FS_PHYGETID

Return value

The required memory.

fs_mount_flashdrive

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

For more details, see Using f_mountdrive with NOR Flash.

Format

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

Arguments

Argument	Description	Type
vol_dsc	The volume descriptor of the volume to mount.	void *
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, not mounted.

Using f_mountdrive with NOR Flash

The **f_mountdrive()** function is part of the main SafeFLASH API. It calls **fs_mount_flashdrive()**. This page shows how to use the fiunction with NOR flash. For a code example, see Mounting a NOR 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.). The maximum value of **drivenum** is set in FS_MAXVOLUME-1 in **fsm.h**.	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 **fs_phy_nor_xxx()** physical driver function for the desired device that is called by the mount function to get information about how to use it. Standard examples are: • **fs_phy_nor_sim()** – for PC emulation of physical NOR. • **fs_phy_nor_29lvxxx()** – for AMD flash.	FS_PHYGETID

Return values

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

Mounting a NOR Drive

The following code shows how to mount your NOR drive.

Note:

- Although the code sample shows dynamic memory allocation this can also be done statically.
- If the **f_getmem()** function is called during development the number returned can be used (and not calculated at run time), as long as the flash type and its configuration does not change.

```
long memsize;
char *p1buffer;

    memsize = fs_getmem_flashdrive( fs_phy_nor_29lvxxx );
    if (!memsize)
    {
        /* configuration error */
    }

    p1buffer = (char*)malloc( memsize );

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

    /* Drive A will be NOR flashdrive with AMD physical driver */
    /* The initial 0 is the drive number to use (0 = A) */
    f_mountdrive( 0, p1buffer, memsize, fs_mount_flashdrive, fs_phy_nor_29lvxxx );

    /* The 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 used.

The **fs_phy_nor_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()** 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 NOR 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. Call this after **WriteFlash()** to verify written data against the original data.
BlockCopy()	Copies one block to another block. (Only required 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_nor_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_nor_xxx ( 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	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 zero 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     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 data in.	long
sector	The zero-based relative sector number in the block.	long
size	The length of data to be stored.	long
relpos	The relative position in the block to write data to.	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 be erased.	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. The parameters are the same as for **WriteFlash()**.

> **Note:** This function is not always necessary; it depends on the particular flash chip and what is specified in the datasheet to guarantee that a program operation has completed successfully. If this function is not needed, then it should return with zero.

Format

```
int VerifyFlash (
    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 with 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 data to verify.	long

Return values

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

BlockCopy

Use this function to copy one block to another block.

> **Note:** Only use this 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 FS_FLASH Structure

This is the FS_FLASH structure that the module must set up by using fs_phy_nor_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	Where first physical block starts.
descsize	long	Maximum size of FAT+directory+block index.
descblock1	long	Not used for NAND.
descblock2	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.
cachedescpagesize	long	Size of pages in cache.
ReadFlash	FS_PHYREAD	Pointer to Read content function.
EraseFlash	FS_PHYERASE	Pointer to Erase a block function.
WriteFlash	FS_PHYWRITE	Pointer to Write content function.
VerifyFlash	FS_PHYVERIFY	Pointer to Verify content function.
CheckBadBlock	FS_PHYCHECK	Pointer to Check whether block is bad function.
GetBlockSignature	FS_PHYSIGN	Pointer to Get block signature data function.
WriteVerifyPage	FS_PHYCACHE	Pointer to Write and verify page function.
BlockCopy	FS_PHYBLKCPY	Pointer to accelerated block copy function.
chkeraseblk	unsigned char	Buffer for pre-erasing blocks optional request to erase.
erasedblk	unsigned char	Buffer for pre-erasing blocks optional set to 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 NOR 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 describes all the subroutines. It includes notes for porting the routines to a particular hardware design.

FS_FLASHBASE

This define specifies the base address for accessing the flash memory array. The value can be determined only from the hardware design. Sample code is based on an ARM implementation and reads the value from the flash chip selected.

RemoveWriteProtect

This routine removes hardware-supported write protect from flash's chip select. You may supply another function that is based on your hardware design. If write protection is not required, this function may be left empty.

SetWriteProtect

This routine sets hardware-supported write protection to flash's chip select (prevention for further writing). You may supply another function that is based on your hardware design. If write protection is not required, this function may be left empty.

GetBlockAddr(block: long, relsector: long)

This routine calculates the physical address of a relative sector in the specified block. When a descriptor block is specified, the sector field should be ignored and the base address of the block should be returned.

Modify this routine to return the correct block/sector addresses for the requested logical blocks that have been set up in the **fs_phy_nor_xxx()** function.

WriteCmd(cmd: ushort)

This routine writes a command sequence (0x555, 0xAA; 0x2AA, 0x55; 0x555, cmd) to a flash device.

Modify the commands so that they are appropriate for the type of flash device used.

DataPoll(addr: long, chk ushort)

This is an AMD-specific subroutine for checking that data have been written correctly.

The algorithm is:

```
for
    if timeout reached return 2     /* Timeout error */
    readdata from flash addr
    if (data == chk) return 0     /* Ok */
    if (no poll needed) check data and return ok or data error
end for
```

EraseFlash(block: long)

This routine is used by the higher level software to erase a logical block of flash memory.

Modify the commands so that they are appropriate for the specific type of flash device used.

The basic algorithm is:

```
addr = GetBlockAddr(block, 0)
RemoveWriteProtect()
Send Erase Command and addr of which block need to be erased
SetWriteProtect()
return DataPoll(addr)  /* wait until erase is finished and return with result */
```

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

This routine is called by the higher levels to write data to the flash device.

> **Note:** The **sdata** parameter is not used.

Modify the commands so that they are appropriate for the specific type of flash device used.

The basic algorithm is:

```
Destaddr = GetBlockAddr(block, relsector)
Do 16bit data length align
RemoveWriteProtect()
for
    Send Write Command to flash device and program 16bit
    If (DataPoll(addr,data)) return error
    /* wait program end, if error returns */
    If length is reached then end of programming
end for
exit program mode by sending exit command to flash device
SetWriteProtect()
Return OK
```

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

This routine is called by the higher levels after a write operation has been completed to ensure that the data has been written correctly.

> **Note:** The **sdata** parameter is not used.

Modify the commands so that they are appropriate for the specific type of flash device used.

The basic algorithm is:

```
Addr = GetBlockAddr(block, relsector) + Flash base
Do 16bit data length align
Verify programmed data with original data, if error then returns with error
If  all data is checked returns with no error
```

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

This routine reads the specified amount of data from the flash device.

Modify the commands so that they are appropriate for the specific type of flash device used.

The basic algorithm is:

```
Addr = GetBlockAddr(block, 0 ) + Flash base
Calculating start position from blockrel
Copy all data onto data address from flash device
```

fs_phy_nor_xxx (flash: struct)

This routine initializes internal functions of the flash structure.

The basic algorithm is:

```
RemoveWriteProtect()
Get device ID and manufacturer ID from the flash
SetWriteProtect()
Compare read values with all supported device/manufacture codes and fill the flash structure
with corresponding data (size, sectors, block information)
If a matching device is not found return with error
```

fnWriteWord (base: ptr, addr: long, data: ushort)

This routine adds the flash relative address to the base pointer, and writes 16 bits of data into the flash.

Modify this routine, and calls to it, depending on your hardware design.

7 Pre-erasing Blocks of Flash

This section explains how to implement efficient pre-erase functionality on your driver, based on the reference driver that is provided.

7.1 Requirements and Operation

Pre-erasing unused or dirty blocks of flash greatly improves system performance. The driver can be designed to do this.

Requirements

Note the following:

- Pre-erase can be performed only on devices that have commands for suspending and resuming the erase operation. Because erase operations can take a significant time on a NOR flash device, using these commands can greatly reduce system latencies. This section describes how to implement this feature.
- The host system must have some form of task switching and a priority mechanism to support suspension of and resumption of the erase.

The requirements for using pre-erase are:

- A low level task for erasing blocks. This task must be executed after calling NOR driver initialization, because the mutex used is created during driver initialization.
- A mutex for synchronization of processes.

Additional Variables Required

This section describes the items which are needed for pre-erasing.

Two additional fields must be provided in the FS_FLASH structure at driver initialization:

- ***fl_chkeraseblk*** – the array ***chkeraseblk*** contains information about the block to be erased.
- ***fl_erasedblk*** – the array ***erasedblk*** contains the erased state of the block.

These are simple character arrays containing as many entries as there are blocks available in the system. The file system and driver use these arrays to synchronize with each other. At startup both these arrays must be reset to zero (normally the C compiler does this, but not on all platforms).

Two additional variables must be provided:

- ***fl_blknum*** – used for communication between the tasks. This variable signals which block is currently erased or, when its value is 0xFFFF, indicates that no block is currently being erased. Initialize this variable to 0xFFFF.
- ***gl_initiated*** – ensures the mutex is created only once.

The four new variables which are needed for pre-erasing are defined in the reference driver as follows:

```
static unsigned char fl_chkeraseblk[NUMOFBLOCKS];
static unsigned char fl_erasedblk[NUMOFBLOCKS];
static volatile long fl_blknum=0xffffUL;
static FS_MUTEX_TYPE gl_premutex;
```

7.2 Suspend erase and Resume erase Functions

The pre-erase function can only be used with a NOR device that has the suspend/resume erase functions. These are used by the driver in this case to return control to the file system where the pre-erase operation is pre-empted. This operates as follows:

1. Before it executes any operation (read/write/erase), the driver checks whether a block is being erased.
2. If, say, a read operation is requested, the driver executes the **suspend erase** command to suspend the current erase operation.
3. Data can now be read from the NOR device.
4. When the read operation finishes, the driver resumes the interrupted erase by issuing a **resume erase** command.

The following static functions in the reference driver are used by the other driver calls:

Function	Description
Suspend	Checks for a pending erase and, if one is pending, suspends it.
Resume erase	Checks for a suspended erase and, if there is one, resumes it.
Wait for mutex	Waits until the mutex is put (released).
Put mutex	Puts (releases) the mutex for another task.

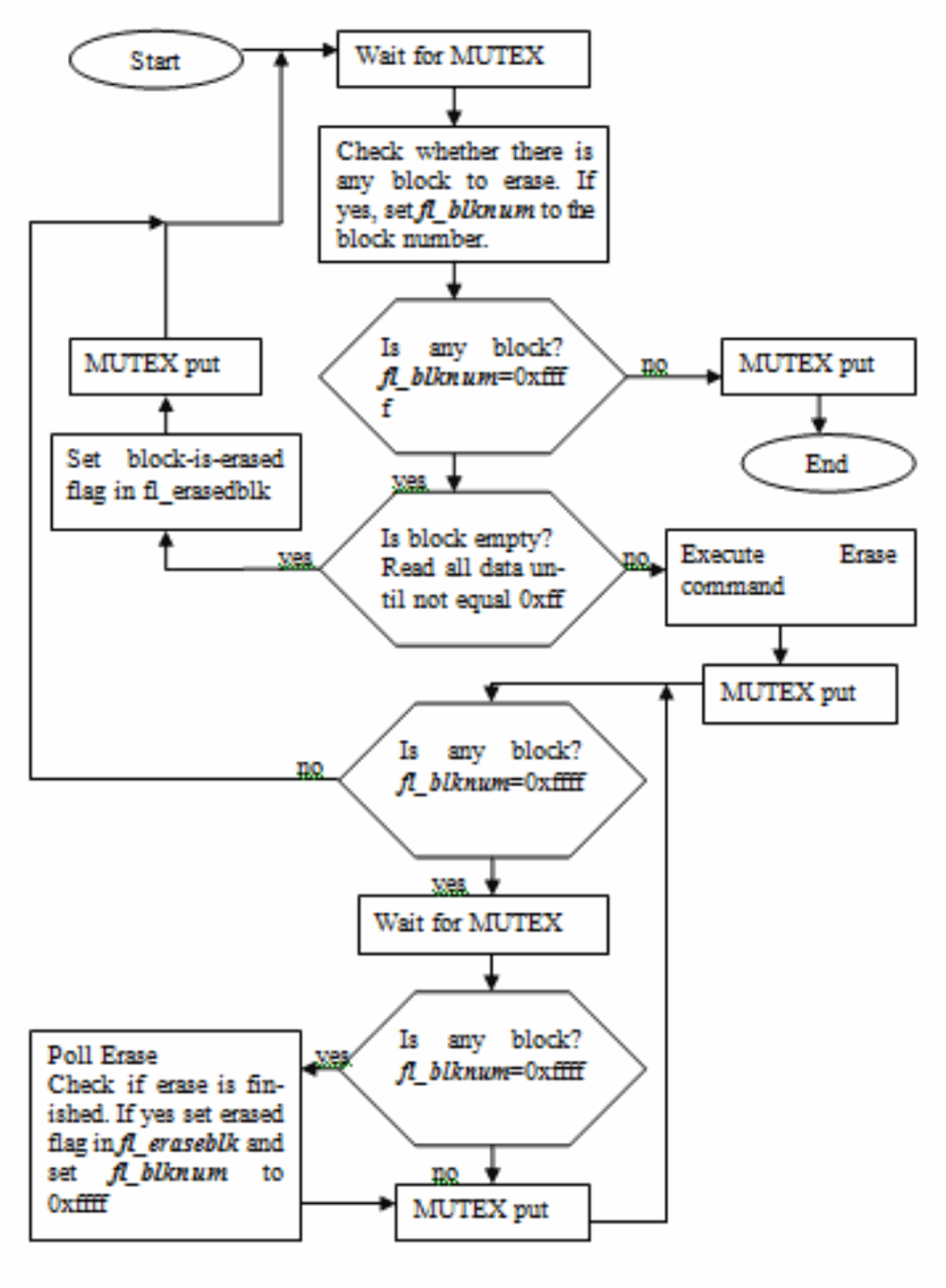
7.3 Flowchart Examples

This section explains how to implement efficient pre-erase functionality on your driver, based on the reference driver that is provided.

The following figures show how a pre-erased driver can be built and how the reference driver works. The highlighted boxes contain the generic NOR driver functions.

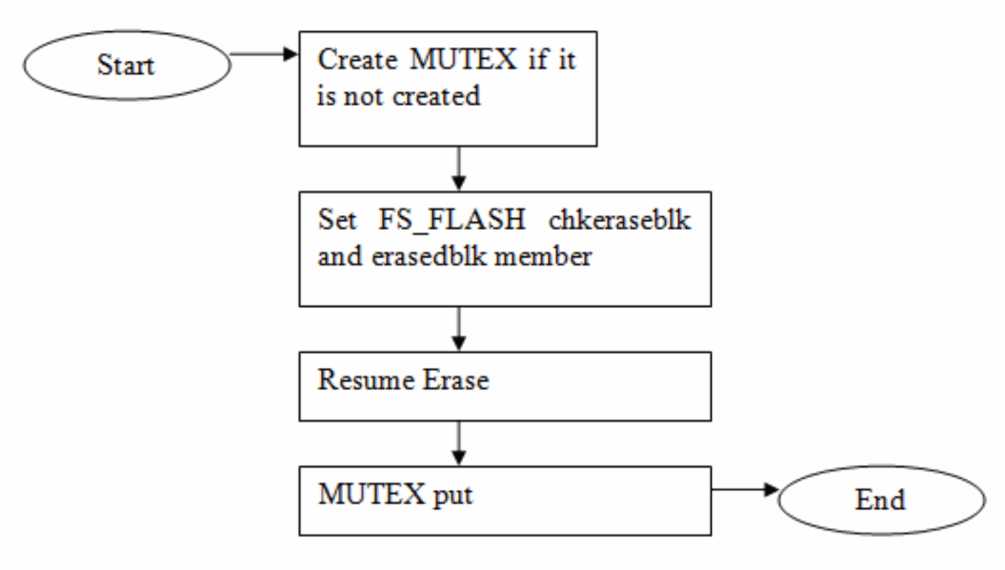
LowFlashErase

This figure shows the low level pre-erase task that must be called cyclically. It returns when there is no block to erase. The function is called **LowFlashErase**.



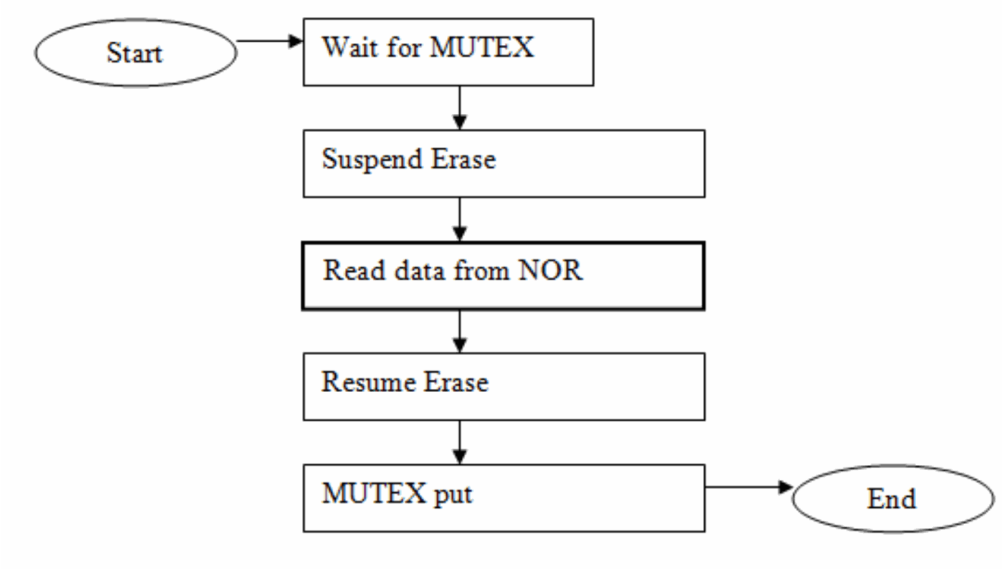
Initialization

This figure shows how to initialize the pre-erase system from driver initialization (the **fs_phy_nor_xxx()** function in the reference driver):



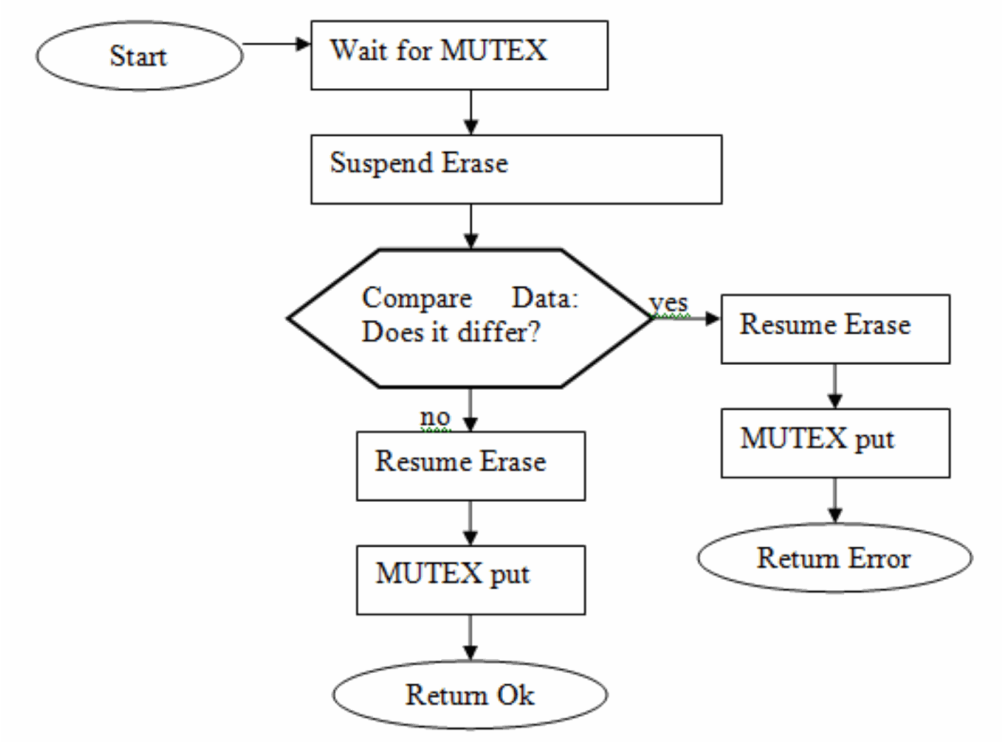
ReadFlash Function

This figure shows how to implement the **ReadFlash()** function:



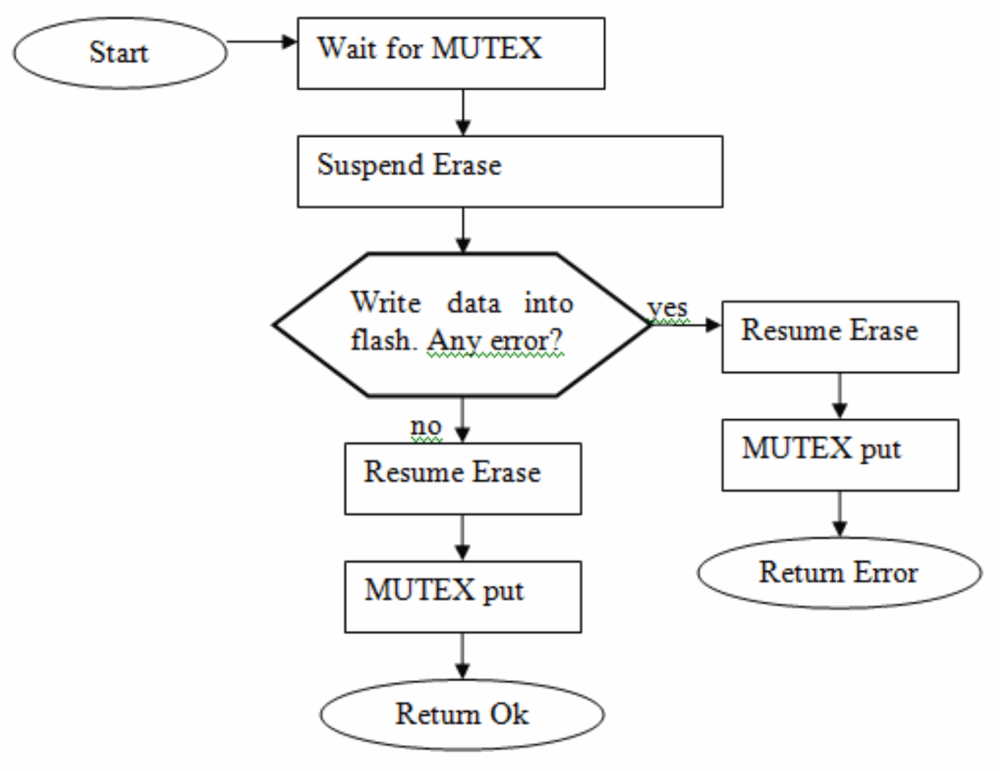
VerifyFlash Function

This figure shows how to implement the **VerifyFlash()** function:



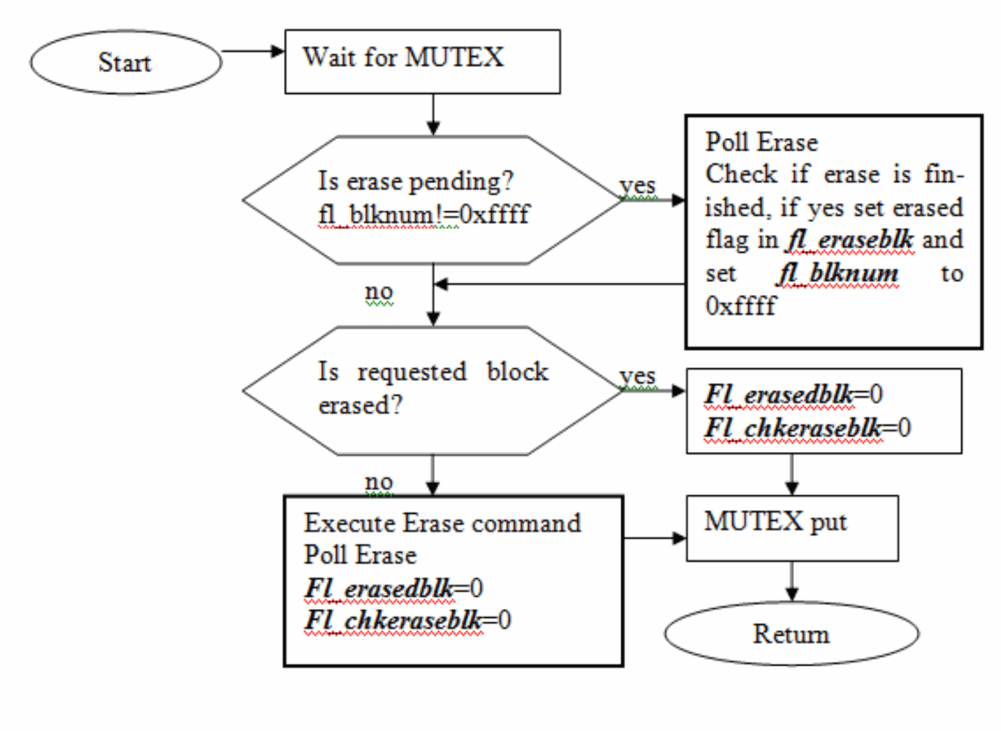
WriteFlash Function

This figure shows how to implement the **WriteFlash()** function:



EraseFlash Function

This figure shows how to implement the **EraseFlash()** function:



8 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.