



smxFFS™ User's Guide

NAND Flash File System

Version 1.90
January 9, 2009

© Copyright 2004-2009 Micro Digital Inc
© Copyright 1999-2004 OSIA Technology
All rights reserved.

Table of Contents

Theory of Operation	1
1. Overview.....	1
1.1 v1.80 and v1.90 Changes	2
1.2 Multi-Level Cell (MLC) Support	2
1.3 Large Flash Device Support	2
1.4 Flash Device and System Requirements	2
1.4 C Library Function Requirements	3
2. Files	3
3. Porting Notes.....	4
3.1 FFSPORT.H and FFSPORT.C	4
3.2 FLHDW.C or FLHDW.ASM and FLHDW_*.H.....	5
4. Virtual File System Introduction.....	7
4.1 Definitions	7
4.2 Memory Map	7
4.3 Description of Memory Map	7
4.4 How File Data is Stored.....	9
4.5 File Handle	11
4.6 File Cache.....	11
4.7 Power Fail Safety.....	11
5. NAND Flash Driver Introduction	13
5.1 Structure of the Flash Driver	13
5.2 LRU Block Cache System	15
5.3 Flash Block Replace Algorithm.....	16
5.4 Wear Leveling	17
5.5 Physical/Logic Address Translation Layer.....	18
5.6 Block Table Handler.....	18
5.7 Data Protection and Recovery	22
5.8 Garbage Collection	25
5.9 Bad Block Handler	25
5.10 Error Correction.....	26
6. Size and Performance.....	29

Virtual File System API	33
Data Types.....	33
Basic Calls	33
Extended Calls	41
Appendix A: Preprogramming Flash	48
Appendix B: Preprogramming Flash and Handling Bad Blocks	49
Appendix C: Flash Chip Array	50
C.1 Parallel.....	50
C.2 Serial	51

Theory of Operation

1. Overview

FFS is a simple flash file system for use with NAND flash memory. It has the standard C library file i/o API (fopen(), fread(), etc), but the functions are prefixed so their names are unique and can coexist with another file system. It also provides a high level of power fail safety. If power fails at any time, the file system can guarantee all the saved data (for closed files) are safe and the file system structure will not be corrupted.

The Flash File System has three layers:

1. **Virtual File System:** provides the standard C library API (fopen(), fread(), etc) to the application.
2. **Flash Driver:** provides the NAND flash algorithm support. It offers caching, physical/logical address translation, reclaim, wear leveling, data protection, garbage collection, ECC, etc.
3. **Hardware I/O Routines:** implement the basic I/O functions for the flash device. These can be ported to support other NAND flash devices.

Figure 1 shows the structure of the Flash File System.

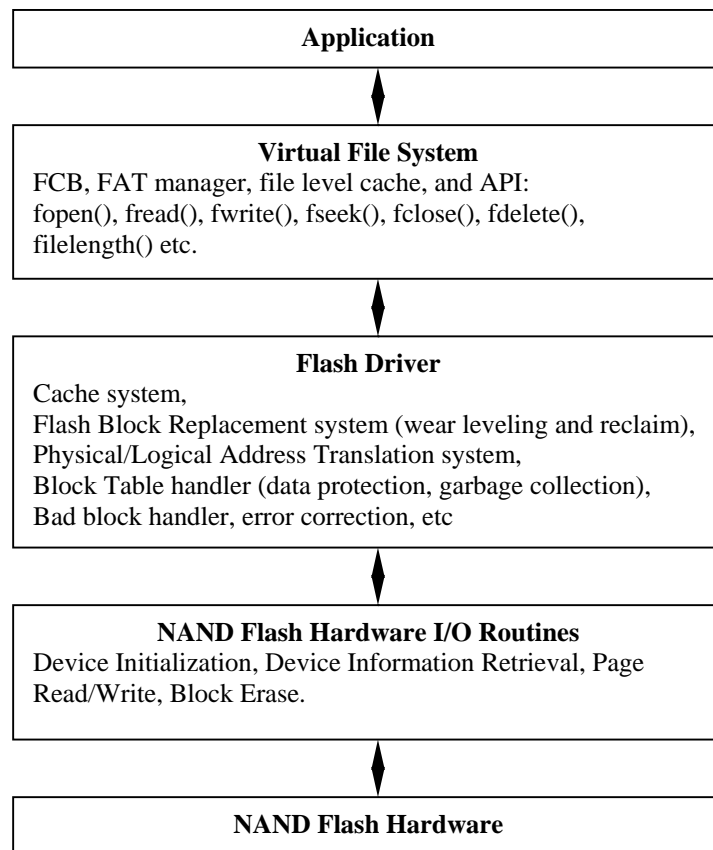


Figure 1: Flash File System Structure

1.1 v1.80 and v1.90 Changes

v1.80 and later are incompatible with older versions of FFS, changes were made to the low-level flash format. To upgrade to a newer version, you must reformat your flash disk. There are several improvements and changes made in these versions:

- The Virtual File System uses the block device interface to access the NAND flash chip so it can share the same chip (on a different partition) with another file system such as smxFS or smxUSB mass storage driver.
- Spare area data read/write operations are 16-bit aligned so it can support 16-bit bus NAND flash chips better.
- Multi-level cell (MLC) support changes were made in v1.80 but support was not complete until v1.90.

1.2 Multi-Level Cell (MLC) Support

Starting with v1.90, FFS supports Multi-Level Cell NAND flash. This flash achieves much higher flash density than Single-Level Cell (SLC) flash, so the capacity can be much higher in a package the same size. However, MLC flash has many limitations compared to SLC. Probably most significant is that it is highly unreliable and requires at least a 4-bit ECC vs. SLC which (in our experience) rarely has errors and 1-bit ECC sufficient. Calculating 1-bit ECC in software is reasonable, but 4-bit is impractical because it puts an enormous load on the processor (nearly a billion instructions to calculate one block (256KB) ECC!). It reduces performance to a crawl. For MLC, you really need to use a hardware 4-bit (or better) ECC controller, which may be combined with an MLC NAND controller. This may be built into your processor, available as an external chip, or available as IP for your custom ASIC or FPGA.

Another significant limitation is that page programming must be sequential and it is not possible to do any partial programming. Even the spare area must be written at the same time as the data area. To support MLC we had to modify the block table handling algorithm which makes it wear the flash a little more and waste some flash (using a whole page to store flags rather than just bytes in the spare area, for example).

Please see our whitepaper “MLC vs. SLC NAND Flash in Embedded Systems” (www.smxrtos.com/articles/mlcslc.htm) for more information before you decide to use MLC in your system.

1.3 Large Flash Device Support

Starting with v1.60, FFS supports very large NAND flash devices (or arrays of many flash devices), which are larger than 4GB and that have more than 16K blocks. These settings can be enabled individually:

- Define `SUPPORT_LARGE_FILESYS` in your make file or in `ffsdefs.h` if your flash device (or array of devices) is larger than 4GB. In this case, FFS Virtual File System uses an unsigned 64-bit integer as the physical address. (Your compiler must support 64-bit integers, such as unsigned long long or `__uint64`.)
- Define `SUPPORT_LARGE_BLOCKNUM` if your flash device’s block number is larger than 16K (2^{14}). In this case, the block table nodes and wear counters are each 32-bit.

1.4 Flash Device and System Requirements

This is a list of requirements that your flash device and target system must meet in order to be able to use FFS. The details are given in the text below; here we just summarize:

- NAND flash only
- For SLC (Single Level Cell) flash:
 - At least 16-byte spare area in each page, for flash with 512-byte page size (see note 1)
 - Ability to read/write the spare area of each page independently of the rest of the page
 - Must support partial programming to a page spare area at least 3 times (see section 5.7, Data Protection and Recovery)

- Maximum flash media size is the lesser of $2^{30} * \text{block_size}$ bytes or $\text{block_size}/8 * \text{block_size}$ bytes. Up to 512GB is supported for flash media with a 2MB block size (see note 2).
- RAM requirement: See section 6 below.

Notes:

1. The spare area is used to store the 6-byte meta information of the page and ECC codes. The ECC codes require 3 bytes (SLC) or 6 bytes (MLC) per 256 bytes. For SLC chips page size larger than 512 bytes, ECC uses $3 * \text{page_size}/256$ bytes. Example: 2048-byte page size requires $6 + 3 * 2048/256 == 30$ bytes. MLC flash uses 54 bytes for ECC. See section 5.10 Error Correction.
2. “Maximum flash media size” means the total size of all flash devices. If SUPPORT_LARGE_BLOCKNUM is defined, Block Table entries are 32-bit and the upper 2 bits are used as flags, so there is a maximum of 2^{30} entries. Also, the complete Block Table and Wear Counter Array must be stored in a single flash block, and since each block table entry is 4 bytes and each wear counter is also 4 bytes, the maximum number of blocks is further limited to $\text{blocksize}/8$. See sections 5.3 Flash Block Replace Algorithm and 5.4 Wear Leveling. If SUPPORT_LARGE_BLOCKNUM is not defined, Block Table entries and Wear Counters are 16-bit and the upper 2 bits are used as flags, so the maximum is $2^{14} * \text{block_size}$ bytes or $\text{block_size}/4 * \text{block_size}$ bytes.

1.4 C Library Function Requirements

This is a list of C library function that the FFS will call. If your compiler does not provide some of those functions, you should implement them in the `ffsport.c`

- `malloc()`
- `free()`
- `memcpy()`
- `memcmp()`
- `memset()`
- `strcpy()`
- `strlen()`
- `strstr()`
- `strcmp()`
- `stricmp()`
- `strnicmp()`

2. Files

FILE	DESCRIPTION
<code>flhdw.c</code> (<code>flhdw.asm</code>)	Low-level flash access functions for NAND flash. This file depends on how the user designs his hardware.
<code>flhdw.h</code>	Low-level flash access function declarations for NAND flash.
<code>flash.c</code>	Flash driver’s function implementations.
<code>flash.h</code>	Flash driver’s API prototype declarations.
<code>flashecc.c</code>	Software ECC code (1-bit for SLC and 4-bit for MLC).
<code>flashecc.h</code>	Software ECC API prototype declarations.
<code>flashcnf.h</code>	Flash driver’s configuration file. The user can change the constants without the need to change the source code.
<code>vfilefla.c</code>	Virtual File System API function implementations.
<code>vfilefla.h</code>	Virtual File System API prototype declarations.
<code>Vfilecnf.c</code>	Virtual File System’s configuration file. The user can change the constants without the need to change the source code.
<code>ffsport.c</code>	FFS OS-dependent function implementations; currently supports SMX [®] and Windows.
<code>ffsport.h</code>	FFS OS-dependent function prototypes and data type declarations.
<code>ffsdefs.h</code>	FFS common constant definitions.

mak.bat	Runs the make utility with ffs.mak to build the FFS library.
ffs.mak	Makefile for the FFS library.
EMU*.*	Emulator files to debug FFS on hard disk

3. Porting Notes

The porting layer consists of these modules:

3.1 FFSPORT.H and FFSPORT.C

These contain definitions, macros, and functions to port FFS to a particular OS, compiler, and CPU architecture. Currently, this file supports the SMX[®] RTOS and Windows/Linux. (For Windows/Linux, we supply emulator code in the EMU subdirectory, which uses the hard disk for debugging FFS.) The main purpose of this file is to implement semaphore protection of the API from reentrancy problems under multitasking environments.

A. The data types defined are:

```

u8;                8 bit, unsigned
u16;               16-bit, unsigned
u32;               32-bit, unsigned
byte;              8 bit, unsigned
uint16;            16-bit, unsigned
uint32;            32-bit, unsigned
uint64;            64-bit unsigned
FFS_LOCK_HANDLE;  handle for the semaphore
FFS_TASK_HANDLE;  handle for task (not used for FFS but for other product of OSIA)

```

B. Big-endian support macros are:

```

INVERTUINT16(w), INVERTUINT32(dw)

```

C. Packed structure and unaligned pointer support macros are:

```

PACKED, PACKED_GNU, UNALIGNED

```

D. The functions implemented are:

```

FFS_LOCK_HANDLE  CREATE_FFS_API_LOCK(void);
    Create the semaphore.

void  RELEASE_FFS_API_LOCK(FFS_LOCK_HANDLE handle);
    Release the semaphore.

void  FFS_API_ENTER(FFS_LOCK_HANDLE handle);
    Test the semaphore; wait if another API function has claimed it.

void  FFS_API_EXIT(FFS_LOCK_HANDLE handle);
    Signal the semaphore so another API functions can run.

```

E. Memory allocation macros are:

```

FFS_ALLOC
FFS_FREE
By default, they are mapped to malloc() and free().

```

3.2 FLHDW.C or FLHDW.ASM and FLHDW_*.H

These contain the functions that are specific to the NAND flash hardware design. The functions listed in FLHDW.H are the ones that need to be ported or implemented to run on a different flash device and/or a different hardware design. FLHDW.C contains the sample code for those low level functions, it supports 512 byte or 2K page size, and 8- or 16-bit bus.

We assume that you are familiar with NAND flash and how it works. We have tested our implementation for Samsung, Toshiba, SanDisk, and Fujitsu NAND flash of 4, 8, 16 and 32 MB size and Samsung, STMicro, Micron, and Numonyx 64MB, 128MB, 256MB, and 1GB size.

NOTES:

- 1. Your flash chip and hardware design MUST support reading and writing the spare area of each page independently of the rest of the page. Normally you should control the SE pin of the NAND flash chip and/or issue a different Flash Command. Our flash driver uses the spare area to store some additional information. Our NAND flash driver sample code also assumes that your flash device DOES NOT use the interrupt to check the R/B signal. We use polling to check it. You can use the interrupt by modifying the sample code.**
- 2. The code in flhdw.c is only sample code. It works for a lot of hardware but may not work for you. It is written for hardware that directly connects the NAND flash chip to the microprocessor's data and address buses. You can use it as a starting point and modify some macros that are related to the hardware design, such as the base address and GPIO setting. If you are using a different interface, such as a built-in NAND flash controller or FPGA, or if you want to use DMA to transfer data between your processor and flash chip, you need to implement a new driver. You only need to follow the function prototypes defined in flhdw.h.**
- 3. If you are using hardware ECC, for example, a built-in flash controller such as the one in i.MX31 or LPC3180/3250, you need to disable the software ECC and implement the hardware ECC generation and result check in the low level driver. If the error is correctable then just return 0 for `asm_Read_Page()`, otherwise, return a non-zero value so the flash driver will do the proper bad block handling.**

Basic functions include:

asm_Flash_Reset (uint iChipID);

Reset the flash hardware. Normally issues the 0xFF command to the chip. Please refer to your hardware spec for details.

asm_Flash_Init (void);

Initialize the interface hardware between your processor and the NAND flash chip, such as GPIO and MMU.

asm_Read_Device_ID (uint iChipID, DEVICE_INFO pDeviceInfo);

Read the device ID so the flash driver can retrieve the hardware information into the DeviceInfo structure. Please refer to the DeviceInfo definition to see which information is needed by the flash driver.

asm_Write_Page (uint iChipID, byte * write_data, uint32 page_index, uint offset, uint32 data_size);

Write some data to the NAND flash. The flash driver ensures that the whole block is already erased before writing to it. Please do not erase it before writing to it. Page_index and offset can be used to generate the physical address you want to write to.

Parameters:

iChipID	The chip index you want to use. Currently only pass 0.
write_data	Pointer to the source buffer
page_index	Page index number.
offset	Offset from the beginning of the main data area. Currently only pass 0.
data_size	Data size to be written. According to the spec for NAND flash, the data_size can be from 1 to page_size + spare_area_size. If the page size is 512 bytes and spare data size is 16

bytes, the data_size can up to 528 bytes. Currently only pass 512/2048 or 528/2112.

Return value:

If the write operation failed, it should return a non-zero value. Otherwise it should return 0.

asm_Read_Page (uint iChipID, byte * read_data, uint32 page_index, uint offset, uint32 data_size);

Read some data from the NAND flash.

Parameters:

iChipID The chip index you want to use. Currently only pass 0.
read_data Pointer to the target buffer
page_index Page index number.
offset Offset from the beginning of the main data area. Currently only pass 0.
data_size Data size to be read. According to the spec for NAND flash, the data_size can be from 1 to page_size + spare_area_size. If the page size is 512 bytes and spare data size is 16 bytes, the data_size can up to 528 bytes. Currently only pass 512/2048 or 528/2112.

asm_Write_Page_Spare (uint iChipID, byte * write_data, uint32 page_index, uint offset, uint32 data_size);

Write some data to the NAND flash spare area. The flash driver ensures that the whole block is already erased before writing to it. Please do not erase it before writing to it.

Parameters:

iChipID The chip index you want to use. Currently only pass 0.
write_data Pointer to the source buffer
page_index Page index number.
offset Offset from the beginning of the spare area. It must be 16-bit aligned, for example, 2, 4, or 6 for v1.80 or later. This is used to avoid the location where the hardware ECC is written (if you use a NAND controller that does ECC).
data_size Data size to be written. According to the spec for NAND flash, the data_size can be from 1 to spare_area_size. If the spare data size is 16 bytes, the data_size can up to 16 bytes. For v1.80 and later, data size is always 16 bits, that is, 2.

Return value:

If the write operation failed, it will return a non-zero value. Otherwise it will return 0.

asm_Read_Page_Spare (uint iChipID, byte * read_data, uint32 page_index, uint offset, uint32 data_size);

Read some data from the NAND flash spare area.

Parameters:

iChipID The chip index you want to use. Currently only pass 0.
read_data Pointer for the target buffer
page_index Page index number.
offset Offset from the beginning of the spare area. It must be 16-bit aligned, for example, 2, 4, or 6 for v1.80 or later. This is used to avoid the location where the hardware ECC is written (if you use a NAND controller that does ECC).
data_size Data size to be read. According to the spec for NAND flash, the data_size can be from 1 byte to spare_area_size. If the spare data size is 16 bytes, data_size can up to 16 bytes. For v1.80 and later, data size is always 16 bits, that is, 2.

asm_Erase_Block (uint32 block_index);

Erase one flash block.

Parameters:

block_index Block index. You may need to generate the block address by multiplying it by block size.

Return value:

If the erase operation failed, it will return a non-zero value. Otherwise it will return 0.

You can use ffstest.c to test if your implementation of these functions works.

4. Virtual File System Introduction

4.1 Definitions

A Sector is the minimum read/write unit of the actual storage media. For NAND Flash, it is a Page.

A Cluster is the minimum storage unit of a file. One Cluster may contain 1 or many sectors.

A Block is a group of Clusters.

The Sector and Cluster definitions are like MS-DOS, except that a Sector is not necessarily 512 bytes, since the page size is not 512 bytes for some flash chips.

4.2 Memory Map

The Virtual File System assumes the storage media is linear, so if the actual media is not linear, such as for NAND flash or a Hard/Floppy Disk, the media driver must provide an interface to convert the linear address to the media address. It also assumes that the media can only be accessed one Sector at a time (not byte by byte). All addresses of the media can be physical (for RAM) or logical (for NAND flash). The media driver should handle this.

Figure 2 shows the memory map for the Virtual File System.

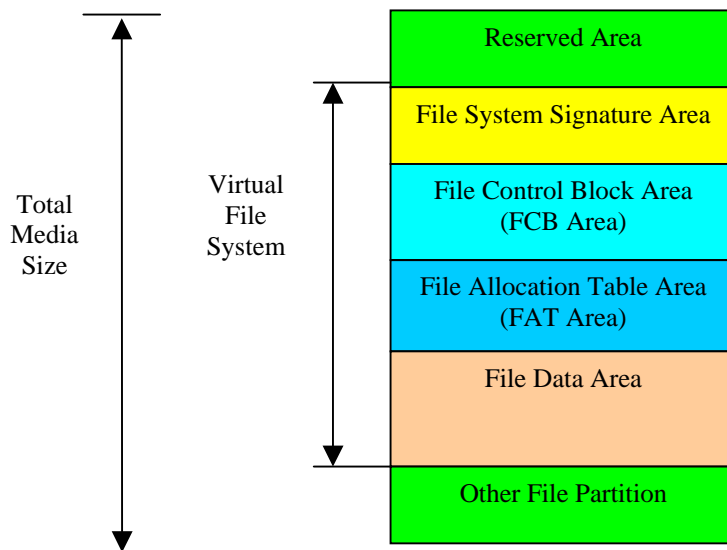


Figure 2: Memory Map of the Virtual File System.

4.3 Description of Memory Map

A. Total Media Size:

For flash memory, Total Media Size is calculated from the `wDataBlockNum` field of the `DeviceInfo` structure. When using RAM (for testing), size is specified by `TOTAL_MEDIA_SIZE` (`vfilecnf.h`).

B. Reserved Area:

There is no Boot Sector or Master Boot Record (MBR) in the file system. If your system requires these, you can use the Reserved Area for them. When using RAM, the size of the Reserved Area is specified by `RESERVED_MEDIA_SIZE` in `vfilecnf.h`. When using flash memory, it is specified by the macro

START_BLOCK_INDEX in vfilecnf.h. The Reserved Area should be block-aligned. It is divided into 2 parts (not shown): the first part stores the Block Table and Wear Counter Array when the Data Area is almost full (that is where they are normally stored); the second part is for the user's use, for any purpose. The size of these areas is controlled by the macro START_BLOCK_INDEX. Please see the section *Block Table Handler* below for a diagram and details.

C. Virtual File System Size:

The Virtual File System may not use the whole media size. Another file system may co-exist on the same flash chip in a different partition. Virtual File System size is defined by START_BLOCK_INDEX and PARTITION_BLOCK_NUM in vfilecnf.h

D. File System Signature Area:

This area stores a string to identify the flash file system. It is located at the beginning of a new block, and its size is defined by the macro FSID_LEN in vfilecnf.h. You can change the pre-defined string set by FS_FLAG_STRING in vfilecnf.h. For example: "My Company FFS v1.00". Please make sure your flag string does not exceed FSID_LEN (including the null terminator). If this string is less than FSID_LEN, padding is added. This string is the only thing in this area.

E. File Control Block Area (FCB Area):

This is the file directory. It contains many FCB's; each FCB represents one file. The total number of files (and FCB's) is defined by the macro MAX_FILES in vfilecnf.h. Figure 3 shows the structure of the FCB (defined in vfilefla.c). The file name length macros FILENAME_LEN and FILEEXT_LEN are defined in vfilecnf.h. The first FCB is reserved. It points to the list of free clusters and also records some system information such as the total number of open files and total free space.

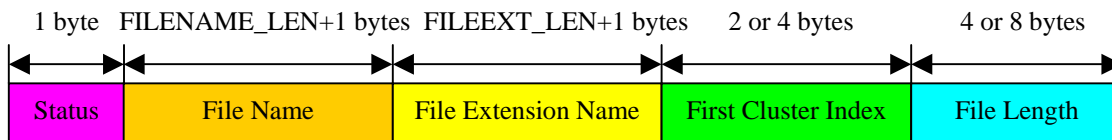


Figure 3: FCB Structure

- a. cStatus: Indicates the status of this FCB. FILE_DELETED means this FCB is empty and can be re-used. FILE_OPENED or FILE_CLOSED means this FCB represents a valid file. A value greater than FILE_OPENED means the file is opened by multiple tasks. When the file system is mounted, this field is reset to FILE_CLOSED for all valid entries.
- b. szFileName, szExtName: These store the file name and extension name, including the null character.
- c. nFirstCluster: Index of the first cluster. This is the head node of the linked list in the FAT (next section). If you defined SUPPORT_LARGE_FILESYS, this field's size is 4 bytes to support large media.
- d. nFileLength: Length of the file. There is no EOF char at the end of the file, so the file system uses this field to decide whether the end of the file is reached. If you defined SUPPORT_LARGE_FILESYS, this field's size is 8 bytes to support large media.

F. File Allocation Table Area (FAT Area):

This area contains many FAT nodes. Each node represents one cluster of a file. The total number of nodes in the FAT area is calculated by the program according to the total media size, reserved media size, and the size of the FCB area. The formula is:

- Assume:
- T = Total Media Size
 - R = Reserved Media Size
 - S = File System Signature Size = FSID_LEN
 - C = File Control Block Area Size = MAX_FILES * sizeof (FCB)
 - N = Number of File Allocation Table nodes
 - B = Bytes per Cluster
 - P = Padding Area (so Data Area starts on a block boundary)

Then:
$$T = R + S + C + N * \text{sizeof}(\text{FATNODE}) + N * B + P$$

$$N = (T - R - S - C - P) / (\text{sizeof}(\text{FATNODE}) + B)$$

The FATNODE type is defined in vfilecnf.h. It is 16-bit by default, so it can support up to 65534*512 bytes of data (we reserve two values, 0xffff (-1) and 0xfffe (-2), for end cluster and bad cluster, respectively), so by default, FFS supports media up to 32 MB in size. To support larger media, increase the size of the FATNODE type to 32-bit. However, note that increasing the size of this type makes every node in the FAT that much bigger, so a bigger FAT is required to map the same number of clusters. If you defined the macro SUPPORT_LARGE_FILESYS, the FATNODE size is set to 32 bits by default.

To avoid having a large FAT, we provide a macro to enable the user to change the size of a cluster. By default, a cluster contains only one sector (page). You can increase the setting PAGES_PER_CLUSTER in vfileconf.h to reduce the number of clusters needed to map your flash disk, so that you can continue to use a 16-bit FATNODE to support a flash disk larger than 32MB. For example, if you change it from the default value of 1 to 4 then your cluster will contain 4 pages (2KB data if your flash's page size is 512). However, remember that since a cluster is the minimum file allocation, a file of only 1 byte will use 2KB of the flash disk, in this example. Unless you need to store a lot of little files on your flash disk, it is a good idea to increase the cluster size for a flash disk larger than 32MB. For smaller media, we recommend using the default value, one sector (page) per cluster.

The File System Signature Area, FCB Area, and FAT Area are contiguous. The additional pad (P) is added after the FAT area to ensure the File Data Area is aligned on a block boundary. When the file system is mounted, the whole FCB and FAT are cached in RAM to improve performance.

G. File Data Area:

This area stores the file data and the Block Table and Wear Counter Array. (When this area is almost full, the Block Table and Wear Counter Array are stored at the beginning of the Reserved Area.) It is aligned on a block boundary. The total size of the File Data Area is calculated by the number of FAT nodes.

H. Other File Partition:

This area can be used by another file system, such as smxFS. The Virtual File System will never access that partition.

4.4 How File Data is Stored

We can access the data clusters of each file by a linked list in the FAT Area. The macro END_CLUSTER defines the flag used to mark the last cluster of the file. The first cluster is stored in the nFirstCluster field of the file's FCB. If the nFirstCluster field value is END_CLUSTER, then this file has no data at all.

Figure 4 shows how one file named "app.bin" is stored in the media.

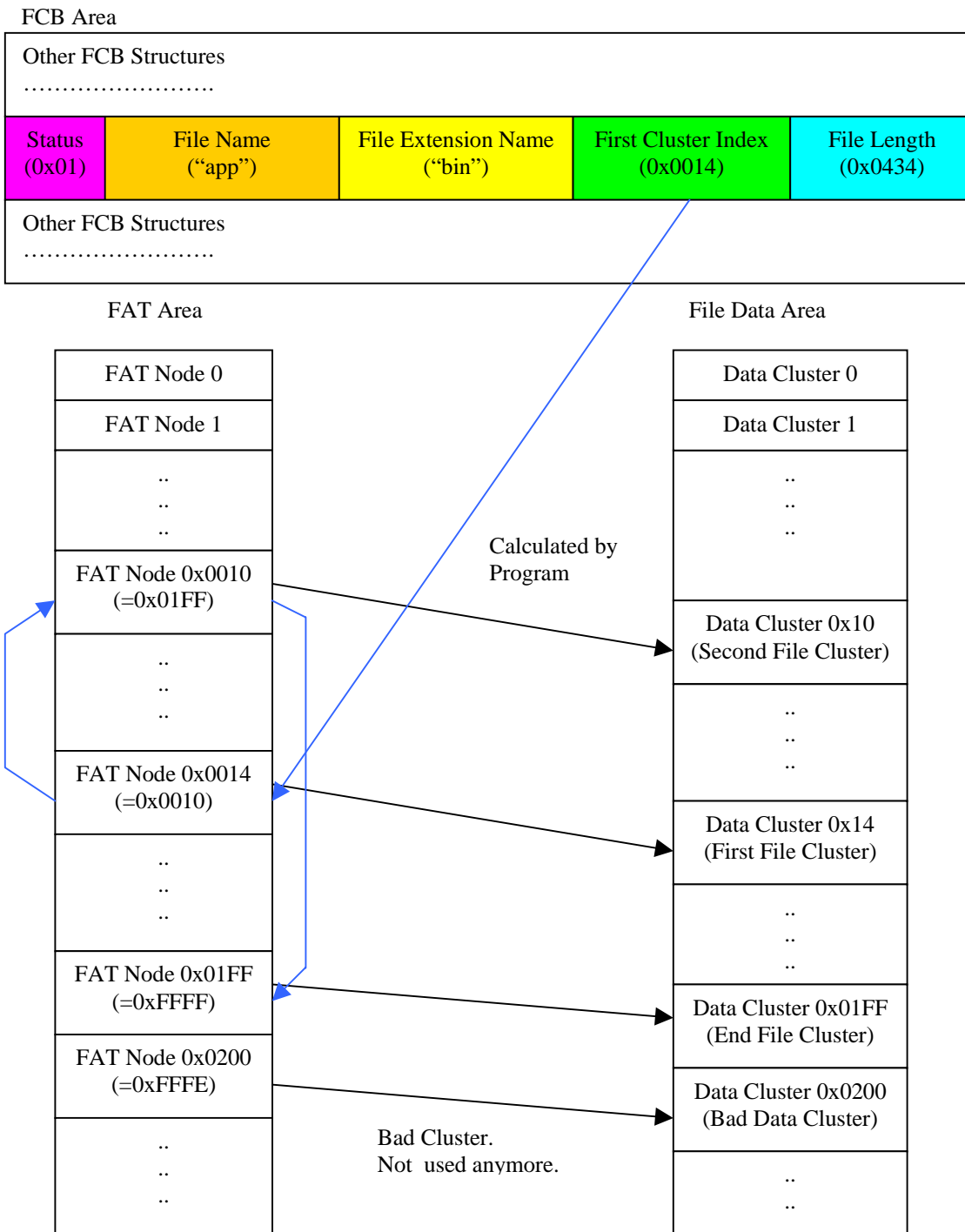


Figure 4: One file named "app.bin" is stored in the media.

FCB 0 points to the list of free (empty) clusters.

4.5 File Handle

A File Handle is used to control read/write operation for a file after it is successfully opened. The handle is a pointer to a structure containing status information about the file. See Figure 5, which shows this structure. The structure is allocated from the heap automatically and its handle is returned by the `fopen()` function. It is released by the `fclose()` function. Do not directly access the fields of the File Handle structure.

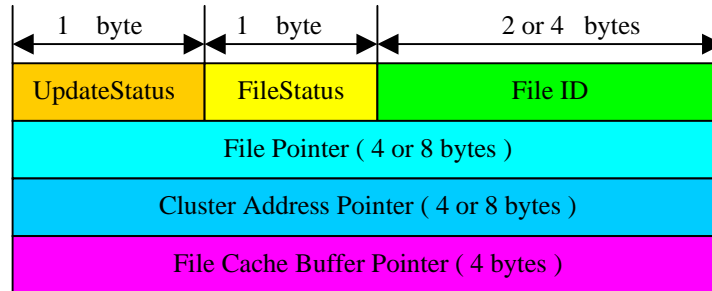


Figure 5: File Handle Structure

These fields have the larger sizes indicated when `SUPPORT_LARGE_FILESYS` is defined.

- a. `cUpdateStatus`:
This byte is used to record which parts of the current file have been updated. Those parts include: FCB, FAT, Data Cache.
- b. `cFileStatus`:
This byte is used to record the file attribute: Read Only or Read/Write.
- c. `nFileID`:
This is the index of the current file in the FCB Area. The file system can access the FCB structure via this index.
- d. `uFilePointer`:
This field stores the current file pointer. There is only one pointer; the file system does not maintain separate Read and Write pointers.
- e. `pPointer`:
This is the media address of current Data Cluster (the one the file pointer points to). This address is the device address and it should be cluster-aligned. The file system uses this address to handle the data cache.
- f. `pBuf`:
This is the pointer to the File Cache in RAM. Accessing the cache in RAM is much faster than accessing the flash.

4.6 File Cache

The file system uses a memory cache to store the read/written data. The cache is 1 data cluster in size. Each open file has its own file cache. The cache is not written to the media (flushed) until the user tries to access data located in another cluster, or the user calls `fclose()`. The contents of the File Cache are automatically updated according to the read/write operation. Because there is only one file pointer for both read and write operations, an `fread()` or `fseek()` operation to another cluster will cause the last `fwrite()` data to be written to the media, and the cache will be changed.

4.7 Power Fail Safety

Note: The FAT used in this file system is not like the FAT in the DOS/Windows FAT filesystem.

smxFFS is power fail safe. The only problem is the possible loss of new data that has not been flushed, but the filesystem will not be damaged by a power failure. The file system caches all the FCB and FAT data in RAM, and it

only modifies the FCB and FAT contents in RAM. It does not flush the changes to the flash chip until the file is closed. If power is lost before a file is closed, the flash driver restores the changed data block to the original one, so the FCB and FAT will also be restored to the original ones. There is no partial write case for the FCB and FAT data, so the file system can keep it consistent even power fails before a file is closed. Of course, the unclosed file's changes will be lost. The disadvantage is that the file system needs a lot of memory to buffer the FCB and FAT data.

The NAND driver is also power fail safe; it maintains a consistent state in its internal data structures.

5. NAND Flash Driver Introduction

The Flash Driver makes it appear to the File System as if the flash is an array of read/write units like a disk. This is difficult because flash bytes cannot be written and rewritten like a disk or RAM. Before a byte can be rewritten, the block it is in must be erased. This is because a write operation can only change a bit from '1' to '0'. Bits are changed back to '1' only by erasing the whole Block, which is the smallest erasable unit of flash. In order to change a byte in a block, it is necessary to read the whole block into a buffer, change the byte in the buffer, erase the flash block, and then write the updated buffer back to the block. Because this is time-consuming and wears out the flash, the Flash Driver minimizes how often this is done.

A Block is divided into some number of Pages (e.g. 16 or 32). Each page can be separately read/written. You can read/write any number bytes of a page, anywhere in the page. However, you can only write to a particular page a specified maximum number of times (to different areas) before erasing it. For example, if the page is blank, it is possible to write the first 10 bytes and then the last 10 bytes. This feature is called partial programming. The number of times partial programming can be done to a page depends on the flash chip being used. Please refer the data sheet of your chip to determine this. **This Flash Driver requires that it must be at least 3 for SLC flash.** Each page also contains a spare data area (e.g. 16 bytes), which is used by the flash driver to record additional information such the status and ECC.

Another difficulty of flash is that there is a finite limit to the number of times a byte of flash can be rewritten before it fails. To extend the life of the flash, the Flash Driver must try to wear the flash evenly.

The Flash Driver hides all of these complexities.

5.1 Structure of the Flash Driver

The Flash Driver contains the following layers:

- Least-Recently Used (LRU) Block Cache System,
- Flash Block Replace System (wear leveling and reclaim),
- Physical/Logical Address Translation Layer,
- ECC Generation and Checking,
- Block Table Handler (data protection/recovery, garbage collection),
- Bad Block Handler, etc

Figure 6 shows the structure of the Flash Driver.

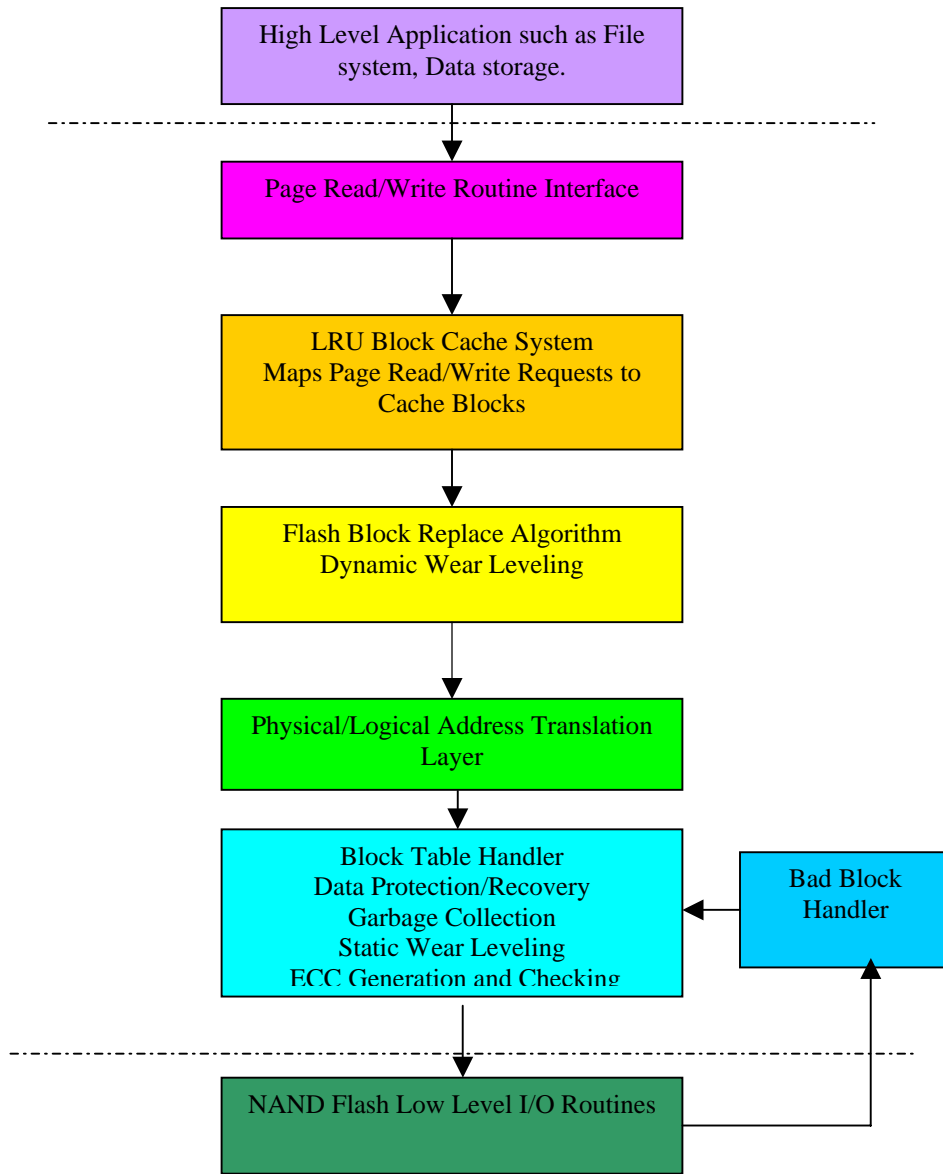


Figure 6: Flash Driver Structure

The NAND Flash driver provides only these functions to the high level application to access data: FFS_Page_Read()/FFS_Sector_Read() and FFS_Page_Write()/FFS_Sector_Write(). The application can only access the NAND Flash by page/sector. The Flash Driver uses a Least Recently Used (LRU) Cache system to hold full or partial flash blocks and reads/writes the pages in them. A cache block can be as small as 1 page or as big as a flash block (it must be a power of 2 * page size). Other parts of the Flash Driver always use the block as the physical data read/write unit, since the NAND flash can only be erased by block. **Although the Cache system may only cache parts of a block, when writing it back, the whole physical block's data will be updated at the same time. That is, the data, which are not in the Cache system, will be read from the old flash block and then copied to the new flash block.** This also simplifies the reclaim and garbage collection procedures. When writing new data to an empty flash block, only the pages with valid data are written. Later, new data can be written to the remaining empty pages, since pages can be written independently.

5.2 LRU Block Cache System

Because of the way NAND flash works, to write data to an area, the area must be pre-erased (all bytes set to 0xFF). To improve performance, the system caches the currently active pages in RAM and flushes them to the flash device only when necessary.

You can select to cache the whole block or only some pages. The macro `PAGES_PER_CACHE_BLOCK` is used to control this. For example, set it to 0 to cache a whole flash block; setting it to 2 means to cache only 2 pages of a flash block. It must be set to a power of 2. 0 is the default and recommended value for flash chips with a block size less than 32KB. If you are using a NAND flash that has a large block size, for example 2KB page size and 128KB block size, you can set it to 4, and then you only cache 4 pages (8KB). `CACHE_BLOCK_NUMBER` in `flashcnf.h` specifies how many cache blocks of size `PAGES_PER_CACHE_BLOCK` are stored in the cache. If RAM is tight in your system, consider the following guidelines when configuring these settings:

- If your flash device's block size is 8KB or less, we strongly recommend that you use 2 cache blocks that are each the size of a flash block (i.e. 2 x 8KB).
- If the block size is 16KB, we recommend at least 2 cache blocks each of size 8KB.
- In general, it is best if the total cache size is at least as big as a flash block or performance may suffer.

Cache blocks are replaced according to the Least Recently Used (LRU) algorithm. Figure 7 shows the LRU Block Cache Flow Chart:

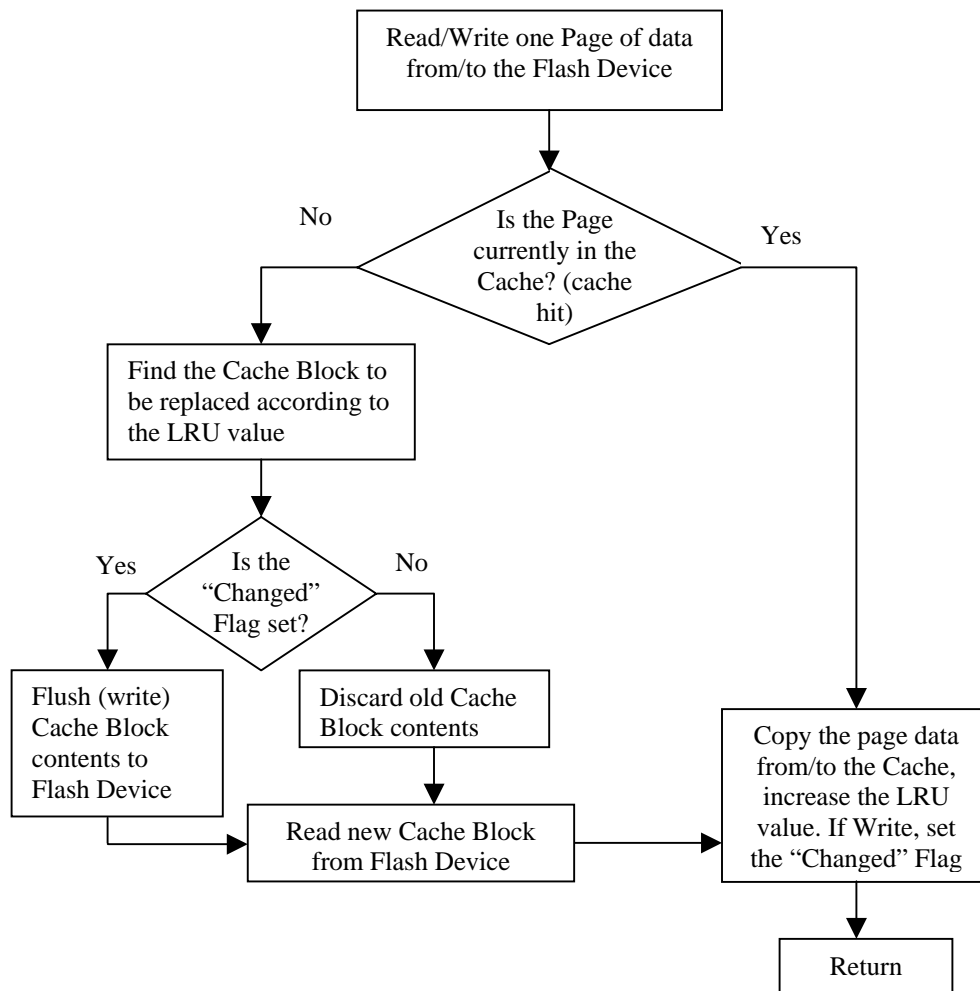


Figure 7: LRU Block Cache Flow Chart

5.3 Flash Block Replace Algorithm

When the contents of a block are changed, whether modifying the data there or adding to it, the flash driver will never write the data to the same place, since data in flash cannot be overwritten. Instead, a spare block is found (that has not been written to and is the least worn of all spare blocks), and the data (with corrections) is written to this new block. Then the old block is marked as discarded. The old data in the original block is not changed at all. This algorithm reduces memory demands and avoids excessive block usage. There are two situations in which the block will not be replaced:

- A. The target block is already a spare block or
- B. The new added data will be written to a spare page (i.e. one that is still erased). (The system will directly write the data to the spare page without the need to erase it.)

The algorithm uses the Block Table to find a spare block and to mark the old block as discarded. The system uses the spare area of each flash page to record the current page status so the system knows which pages are blank (can be immediately written to) and which pages need to be erased first. The algorithm tries to find a spare block that has been used least recently so it can provide wear leveling support. Figure 8 shows the structure of a Block Table entry.

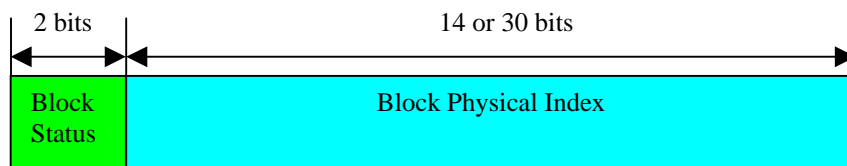


Figure 8: Block Table Entry Structure

The Block Table entries are each 16 bits in the normal configuration or 32 bits if `SUPPORT_LARGE_BLOCKNUM` is enabled in `flashcnf.h`. The highest 2 bits are reserved for the status of the Block. The low 14 or 30 bits are the physical index of the block. The system can calculate the physical address of the block. The highest 2 bits are defined as:

- 00b used block; this block contains valid data
- 10b discarded block; this block contains old data and needs to be erased
- 01b spare block; this block is empty (erased)
- 11b bad block; this block contains error bits and should not be used anymore

Note: If you enable `SUPPORT_LARGE_FILESYS` and `SUPPORT_LARGE_BLOCKNUM` in `ffsdefs.h` and `flashcnf.h`, then for a 2MB block size, up to 512GB flash is supported. (Each block node plus wear counter is 8 bytes, so a 2MB block can store 256K block table and wear counter entries, which gives a 512GB total flash size.)

Figure 9 shows the Flash Block Replace procedure. The new/modified data (green color) is written to a new block (M) and the old data block (N) (yellow color) is unchanged.

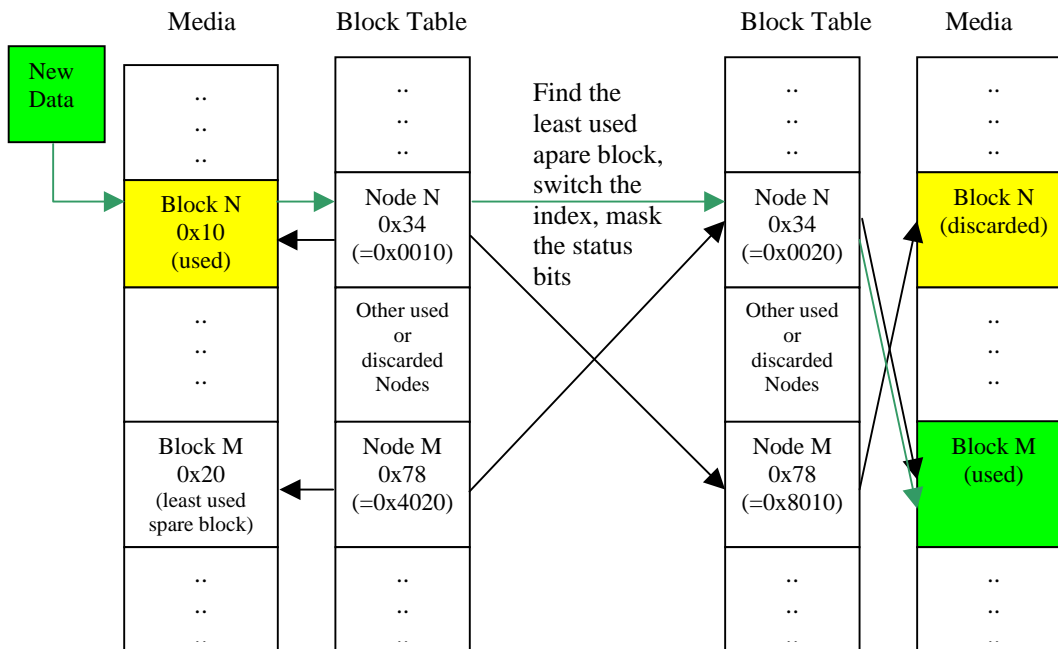


Figure 9: Flash Block Replace Procedure

Since a cache block can be smaller than a flash block (its size is specified in pages), there could be multiple cache blocks from the same flash block. These are all written to the flash block at once. This is the sequence of operations: The old block is processed page by page, starting with page 0. If the page is in the block cache, the data is copied from the block cache into the new flash block. Otherwise, it is copied from the old flash block. This process is repeated for each page of the old block.

5.4 Wear Leveling

NAND flash memory has limited life expectancy. For any given flash device, there is a limitation to the total number of erase operations that may be performed on a particular block before it becomes unreliable or damaged. Flash device lifetimes range from 10,000 write-erase cycles to 1,000,000 cycles, with most rated around 100,000. When a flash block approaches its rated limit, it may begin to fail or take longer to perform operations. To maximize the life cycle of a NAND flash device, it must be wear-leveled. Wear leveling is the process of ensuring all blocks are erased with the same frequency.

Wear-leveling, performed during the garbage collection process, evens usage across the blocks of a flash memory array and so compensates for the finite number of erase cycles available throughout its life. An effective system of wear-leveling must address three major issues:

- The writing of data cannot be confined to only a certain location in the flash.
- Static data in the flash must be moved periodically.
- Block deadlocks must be avoided.

FFS records a wear counter for each block that indicates how many times the block has been written. An array of these counters is stored in the flash, immediately following the Block Table, so this information is retained even with power off. The wear counter for a block is incremented each time the block is written. Since each wear counter is only 16 or 32 bits (just like the Block Table entries), a counter could overflow. FFS prevents this by periodically reducing all counters by the value of the least-worn counter. The Block Table and Wear Counter Array are treated as a unit; each time the Block Table is written to a new location, the Wear Counter Array is too. See the *Block Table Handler* section, below, for more information.

Static files that don't change, such as the application file, font files, icons, etc, and files that are rarely changed must be periodically moved, so all flash blocks in the flash device are evenly worn. FFS moves these blocks during garbage collection. Since it takes time to move blocks (and wears the flash), this must not be done too frequently or system performance will degrade. This is controlled by two configuration settings. The first setting, WEAR_LEVELING_GATE specifies when to do it: If the difference between the most worn-counter and least-worn counter exceeds this value, some of the static blocks are moved. The second setting, WEAR_LEVELING_BLOCK_NUM, specifies the maximum number of static blocks to move each time. Each time garbage collection is done, more of the remaining static blocks are moved.

5.5 Physical/Logic Address Translation Layer

In the Block Table, the system records the Flash Block index in the low 14 or 30 bits so the system can compute the physical device address for the memory space. Figure 10 shows how a Logical address is converted to a Physical address by the Block Table.

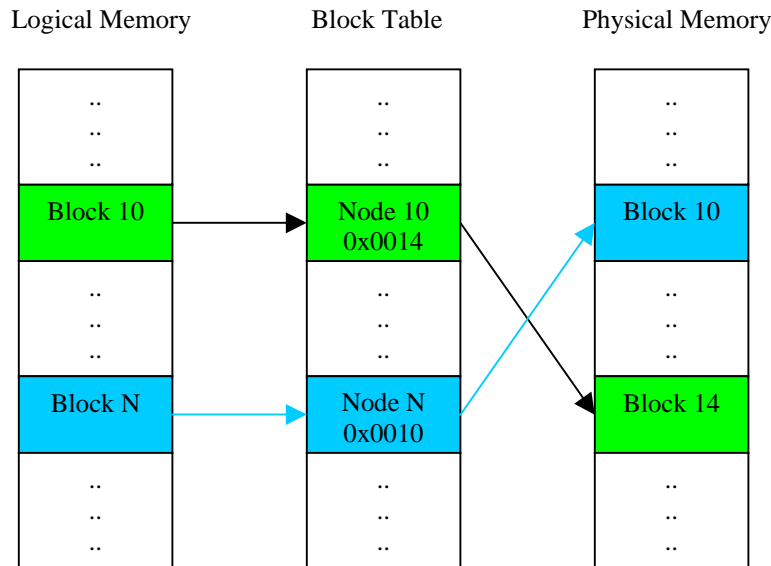


Figure 10: Logical addresses are converted to Physical addresses by the Block Table

5.6 Block Table Handler

The Block Table is the most important data structure in the Flash Driver, so the system must guarantee it is always correct even if power is lost or the flash memory is full. Normally the Block Table (and Wear Counter Array) is stored in a data block (in the Data Area) but in extreme cases when the flash is full, it is stored in the Reserved Area at the beginning of the flash memory. Almost all NAND flash chips guarantee that block 0 is good. So it is safe to store the Block Table to block 0.

The size of the Reserved Area is defined by RESERVED_BLOCK_NUM in flashcnf.h. If the user wants to reserve additional blocks for application use, define the macro START_BLOCK_NUM to be greater than RESERVED_BLOCK_NUM. The flash driver only uses blocks greater than START_BLOCK_NUM for the Data Area. This is shown in Figure 11.

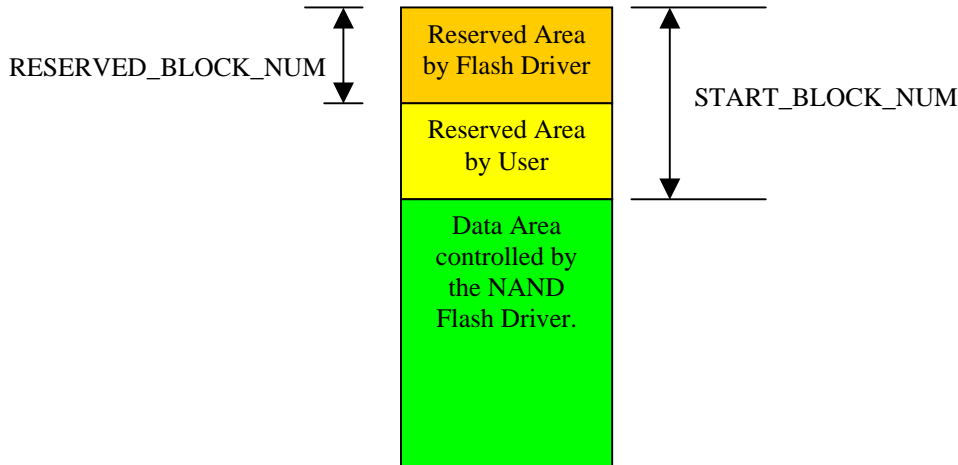


Figure 11: Physical Memory Map of the NAND Flash Driver

The top two areas in Figure 11 correspond to the Reserved Area in Figure 2. The Data Area holds everything else (Signature Area, FCB Area, FAT Area, and File Data Area). Remember that Figure 2 shows the high-level file system view, while Figure 11 shows the low-level NAND driver view.

FFS was designed to avoid having its data structures remain in the same location in the flash, since these are written frequently, which would wear out the flash faster in these areas. Hence, the Block Table and Wear Counter Array must be able to move in the flash, as well. A whole block is allocated to store the Block Table and Wear Counter Array even though they may be much smaller than a block. This allows both to be written several times to the same block as they are updated. When a new Block Table and Wear Counter Array needs to be written, the system selects the next spare (empty) pages in the block and then writes to this new place. This algorithm assumes:

- a. The combined Block Table and Wear Counter Array is always page aligned and
- b. Separate flash pages can be written separately.

Once the Block Table has been written to a new block, the old block is marked as discarded so it can be re-used later to store other data.

This algorithm provides:

- a. Wear leveling for the Block Table area, to avoid writing data to the same area, which would reduce the life of the flash device.
- b. Data protection — the old Block Table is never destroyed until the new one has been successfully written to the new place. If any error occurs during this phase, the data can be restored to previous status.

For example, a 16MB flash device has 1024 blocks, so the Block Table size is $1024 * 2 == 2048$ bytes, and the Wear Counter Array size is $1024 * 2 == 2048$ bytes. The total size is 8 pages. Each block has 32 pages. The Block Table update procedure is shown in Figures 12a (SLC) and 12b (MLC).

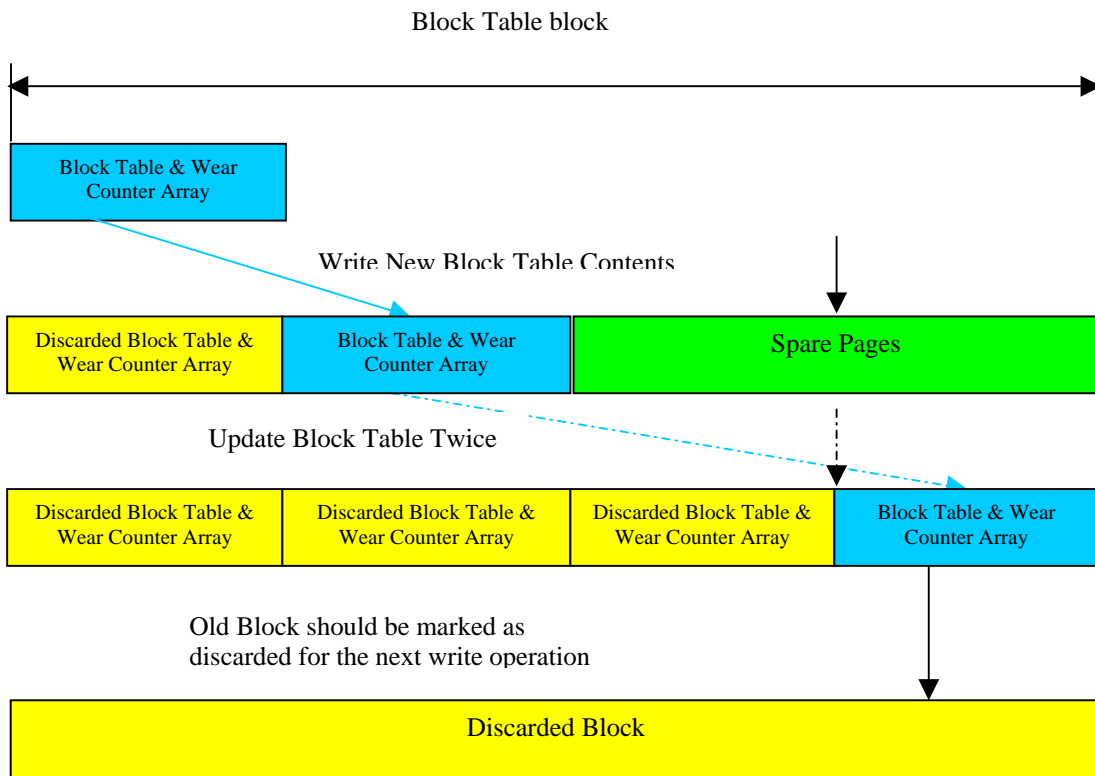


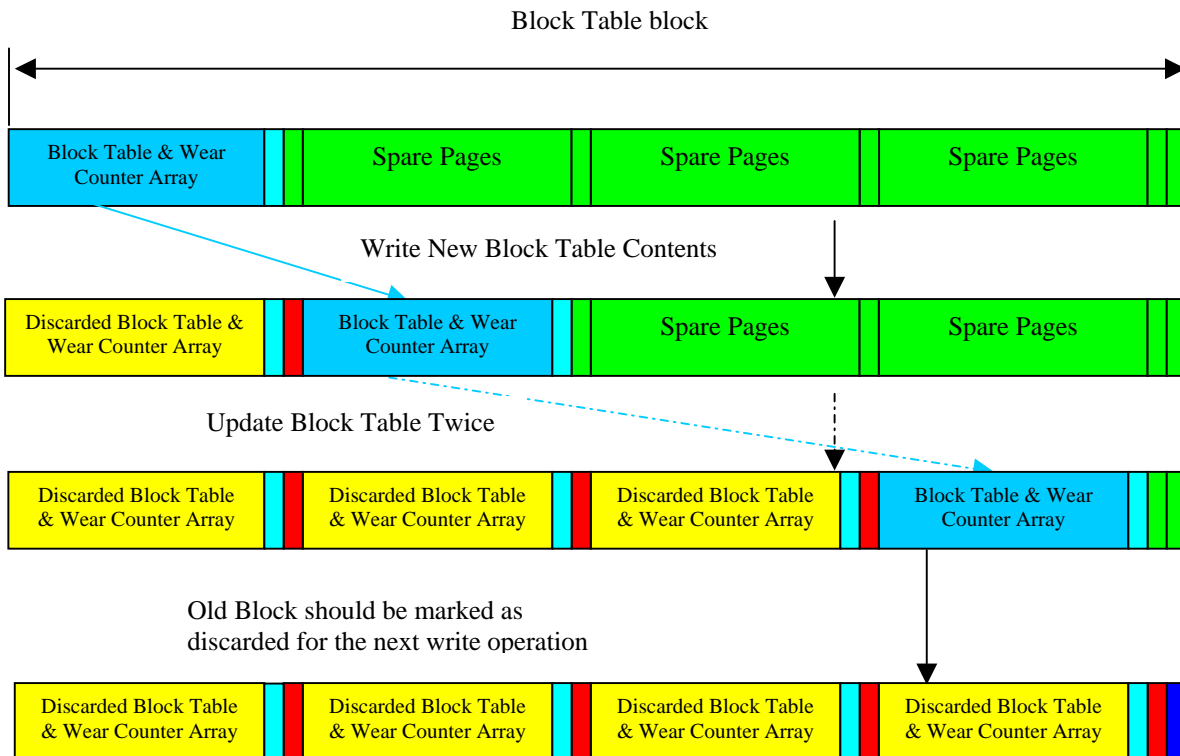
Figure 12a: How the Block Table is Updated (SLC)

Since the Block Table can be anywhere in the Data Area, the system uses a special flag to indicate where it is. This flag is stored in the second byte in the spare area of the last page of the Block Table. This flag is unique and will not be used by other data blocks. During initialization, FFS scans the entire flash memory array, checking all the possible positions that may contain this flag. If it is found, FFS will try to read the information and determine if it is valid. For details, refer to the following section. See Figure 12c for details about how the status byte is updated in the spare area for SLC flash.

MLC Flash

Because MLC flash does not support partial programming and all data write operations must be done sequentially from LSB to the MSB, a little modification is needed to update the block table for MLC flash. (Please see the next section for discussion of the Valid, In-Progress, and Discarded status flags.)

- A. We cannot use the same spare area to mark In-Progress and Discarded status of the block table because we cannot do partial programming. So for each block table, we need one additional page to write the In-Progress status. We put the In-progress status at the first page's spare area after the block table.
- B. To save the overhead, we don't write the discarded status unless the whole physical block should be discarded. When we initialize the flash driver. If we find an In-progress status. We will check the next possible block table Valid status. If we find a valid status. We know that the In-Progress Block Table is actually a discarded block table and we need to ignore it. If we found a discarded status at the end of that block then we know we should ignore the whole block.



Cyan box represents the Valid status in the last page's spare area. Red box represents the In-Progress status page. Blue box represents the Discarded status page. It is the last in the group of pages.

Figure 12b: How the Block Table is Updated (MLC)

See Figure 12d for details about how the status byte is updated in the spare area for MLC flash.

5.7 Data Protection and Recovery

The NAND flash driver must ensure data is never lost. The Flash Block Replace algorithm provides some data protection because it will not destroy the old data before the new data is successfully written into the flash . The Block Table is used to provide the data protection and recovery method. The algorithm is:

Before any data is to be written to the flash or before the Block Table is updated, the current Block Table (in flash) is first marked as “In-Progress”. When the Block Table is successfully updated, the old Block Table (in flash) is then marked as “Discarded”. The system uses the spare area of the last page of the Block Table to mark the status of the Block Table. The status value is defined as follows:

0x7F	Valid. This is the current Block Table.
0x7E	In-Progress. The Block Table is being moved.
0x00	Discarded Block Table.

We write each status value to a different offset in the last page’s spare area. Valid status is at offset 0, In-Progress is at offset 2 and Discarded is at offset 4. (The values are defined as they are because in early versions of smxFFS, we wrote the status 3 times to the same location, but we found this was unreliable.) **In order to do this, the flash chip must support partial programming to a page at least 3 times.** Writing the data and the Valid status are done together as the first partial programming. Writing In-progress status is the second partial programming, and Writing Discarded status is the third partial programming.

If any exception occurred during a write operation, there will always exist an In-Progress block table status but no Discarded status. When the NAND flash driver is initialized the first time, the system checks to see if an In-Progress Block Table exists. If so, the system knows the last operation did not complete and some checking must be done. Specifically, the flash driver does the following:

- A. Accept the “In-Progress” Block Table as the current valid Block Table. First, it is copied to another spare block. Then the old block is erased to ensure Block Table integrity is OK.
- B. Check every Spare Block to ensure it is really spare (empty) because the last write operation may have been writing some data to the block but it did not complete, so the status in the Block Table was not updated correctly.

After the above checking, the data has been restored to the last valid status. The only problem might be that data in the cache that had not been flushed will be lost, but the filesystem will be intact.

The following diagrams show how the status byte is updated in the spare area for SLC and MLC flash.

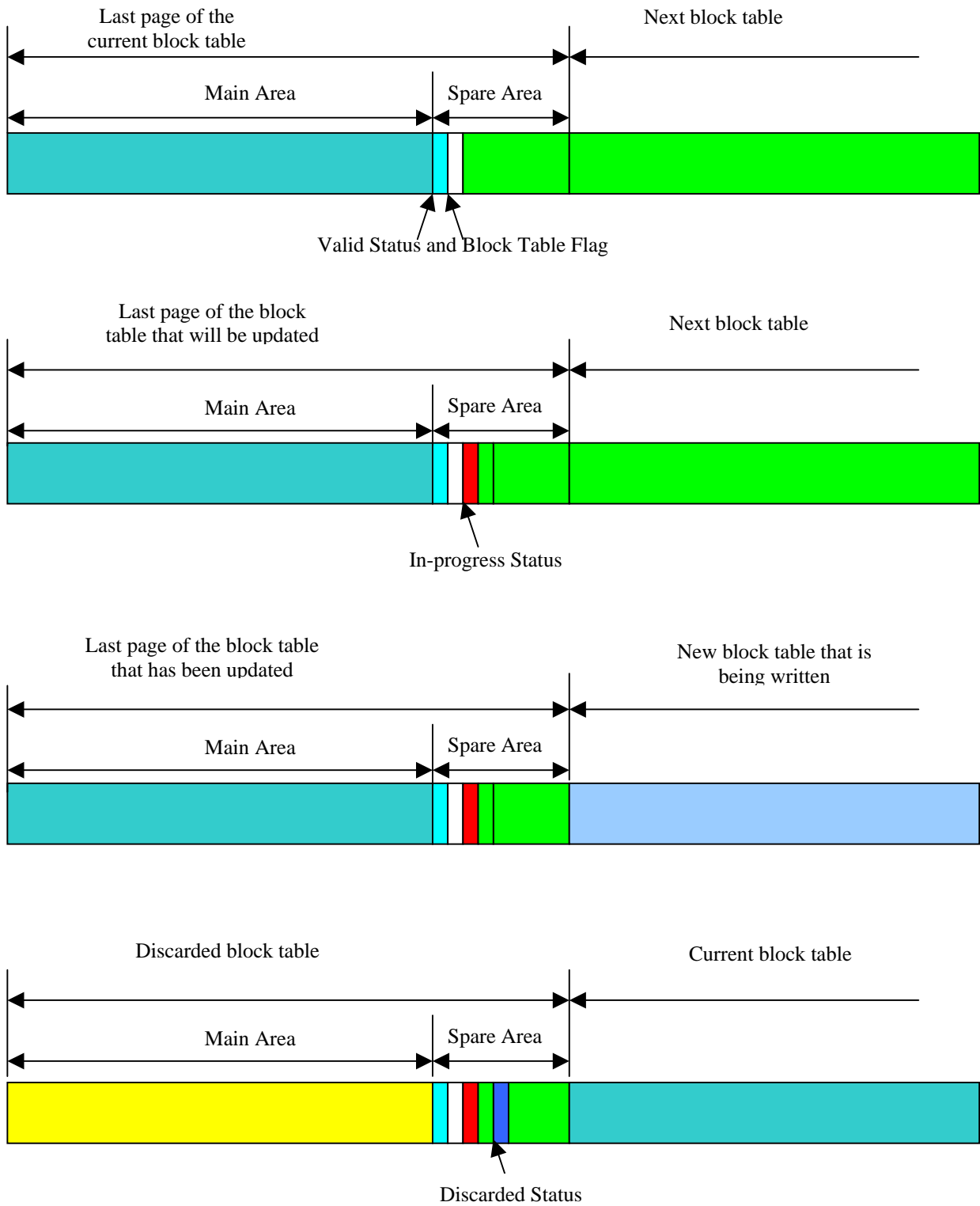


Figure 12c: How the Status Byte is Updated (SLC)

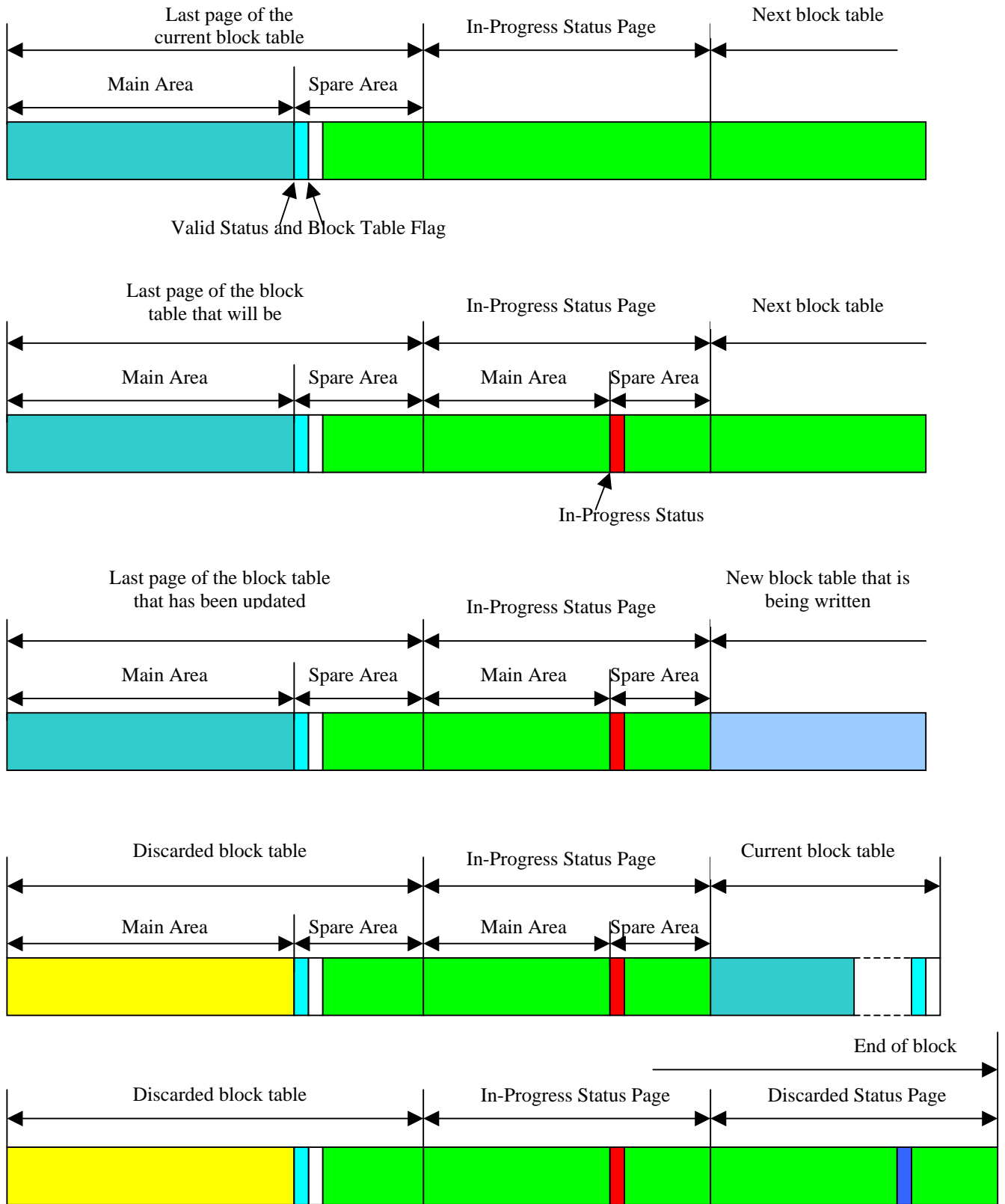


Figure 12d: How the Status Byte is Updated (MLC)

5.8 Garbage Collection

As mentioned before, the NAND Flash Driver only marks old data blocks as “discarded”. There should be a method to erase the discarded block during the idle time to convert it to a spare block so it can be used for a future block write operation. Garbage Collection performs this task. The Flash Driver, itself, does not provide automatic garbage collection because it does not know when the system is idle. It only provides a function named `FFS_Whole_Cache_Write_Back()`. This function flushes the cache (writes the contents to flash) and does the garbage collection procedure. It is the user’s responsibility to decide when to call this function. It should be called when the whole system is idle. It is recommended that you call it when you close a file or finish some data operation. This ensures the flash driver can work without any OS support.

If the file system is full, spare blocks are used from the Reserved Area, since a power fail during this time could otherwise cause the system to lose important data. If no spare block is found during the Flash Block Replace procedure, garbage collection is forced to run.

5.9 Bad Block Handler

If the low-level I/O routine returns an error for the write (programming) and/or erase operation, the system will retry a few times. This is specified by `RETRY_TIMES` in `flashcnf.h`. If all retries fail, the system considers the block to be a bad block. It will find a spare block from the end of the Block Table, write the whole data into the new block, and then mark the old block as bad. Figure 13 shows the bad block handler procedure.

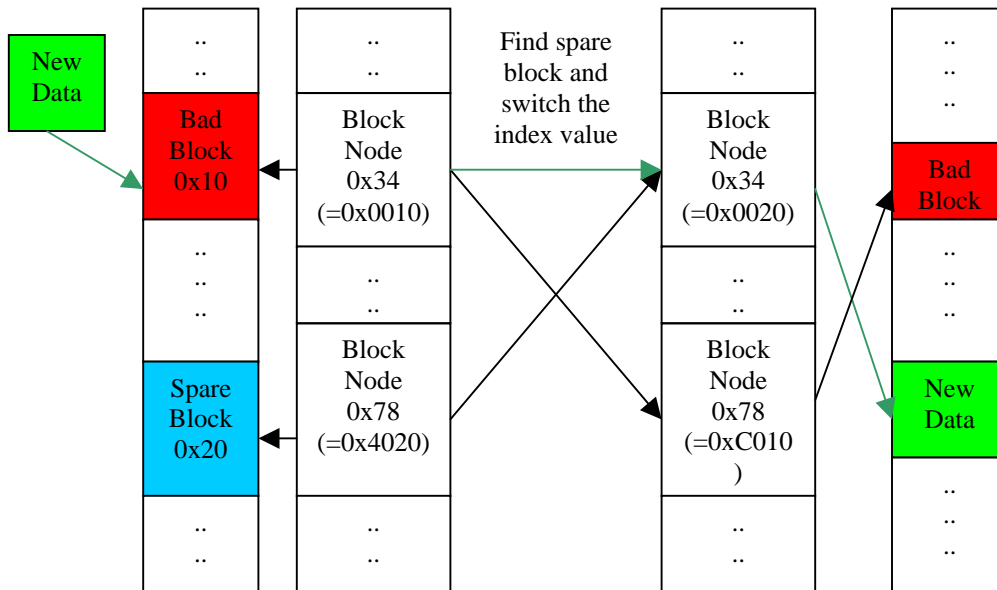


Figure 13: Bad Block Handler Procedure

(The high 2 bits of the block node (e.g. the 4 in 0x4020) indicate the status of the block. See Figure 8: Block Table Entry Structure.)

The bad block handler is transparent to the high level application. The application does not need to do any data replacement operation.

5.10 Error Correction

After the NAND flash chip has been used for a long time, it may develop some bad bits. Normally for SLC flash there is only one bad bit in a whole page but for MLC flash, there will be up to 4 bad bits. FFS implements two software ECC algorithms capable of fixing a 1-bit error for SLC or 4-bit for MLC. The ECC code is 3 or 6 bytes per 256 bytes, and the algorithm will detect if there are any error bits in the page. For SLC flash, if there is only a 1-bit error (from '1' to '0' or vice versa), ECC can also correct it. For MLC flash, if there is only a 4-bit error (or less), ECC will correct it.

The ECC is generated before the data is actually written to the NAND flash chip. When data is read back from the flash chip, if it has a correctable error, the corrected data is returned. If the data has an error that cannot be corrected, an error is reported.

Our ECC algorithm will only process 256 bytes, but a page contains 512 bytes or a multiple of 512 bytes, so each 512 bytes of data requires two ECC codes: one for the first 256 bytes and the other for the second 256 bytes. The ECC code is stored in the spare area of each page. Also a 6-byte metadata value is stored at the start of the spare area. Please see the structure definition for PAGE_HEADER in flash.c for details. If your flash chip's page size is a multiple of 512 bytes, several ECC codes may be created in the spare area, so your flash chip must have spare areas larger than 16 bytes. (The ECC codes require 3 or 6 bytes per 256 bytes, so ECC uses 3(or 6) * page_size/256 bytes, and 6 bytes are required for reserved status. For example a 2048-byte page size SLC flash requires 6 + 3 * 2048/256 == 30 bytes in each spare area and a 2048-byte page size MLC flash requires 6 + 6 * 2048/256 == 52 bytes in each spare area.) Normally, the manufacturer has handled this already. For example, if the page size is 2048, the spare area is 64 bytes instead of the normal 16 bytes, which is plenty to store the ECC information.

5.10.1 1-bit ECC Code

We use a Hamming code to implement 1bit ECC.

A. ECC code consists of 3 bytes per 256 bytes

- Actually 22 bit ECC code per 2048 bits
- 22 bit ECC code = 16 bit line parity + 6 bit column parity

B. Data bit assignment table with ECC code

1 st byte	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	LP00	LP02	LP04
2 nd byte	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	LP01		
3 rd byte	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	LP00	LP03	
4 th byte	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	LP01		

253 th byte	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	LP00	LP02	LP05
254 th byte	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	LP01		
255 th byte	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	LP00	LP03	
256 th byte	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	LP01		
	CP00	CP01	CP00	CP01	CP00	CP01	CP00	CP01			
	CP02		CP03		CP02		CP03				
	CP04				CP05						

Column Parity is calculated over the entire data block as each data byte is processed. Selected bits of each data byte are added to the previous value of each Column Parity bit. The equations for the Column Parity bits are:

$$\begin{aligned}
 \text{CP00} &= \text{bit7 XOR bit5 XOR bit3 XOR bit1 XOR CP00} \\
 \text{CP01} &= \text{bit6 XOR bit4 XOR bit2 XOR bit0 XOR CP01} \\
 \text{CP02} &= \text{bit7 XOR bit6 XOR bit3 XOR bit2 XOR CP02} \\
 \text{CP03} &= \text{bit5 XOR bit4 XOR bit1 XOR bit0 XOR CP03} \\
 \text{CP04} &= \text{bit7 XOR bit6 XOR bit5 XOR bit4 XOR CP04} \\
 \text{CP05} &= \text{bit3 XOR bit2 XOR bit1 XOR bit0 XOR CP05}
 \end{aligned}$$

Line parity is calculated over the entire data block as each data byte is processed. If the sum of the bits in one byte is 0, the line parity does not change when it is recalculated. The sum of the bits in 1 byte of data is:

$$Dall = \text{bit7 XOR bit6 XOR bit5 XOR bit4 XOR bit3 XOR bit2 XOR bit1 XOR bit0}$$

Sixteen line parity bits (LP15-LP00) are computed from 256 bytes of data. An 8 bit counter counts data bytes, bits of this counter are used as a mask for Line Parity bits. The counter increments by 1 for each new byte of data. Line Parity is computed by initializing all line parity bits to zero, reading in each byte, computing the byte sum (Dall), and adding Dall to the line parity bits when they are enabled by the appropriate counter bits.

The equations for the Line Parity bits are:

- LP00 = LP00 XOR (Dall AND Counter_bit0)
- LP01 = LP01 XOR (Dall AND Counter_bit0)
- LP02 = LP02 XOR (Dall AND Counter_bit1)
- LP03 = LP03 XOR (Dall AND Counter_bit1)
- LP04 = LP04 XOR (Dall AND Counter_bit2)
- LP05 = LP05 XOR (Dall AND Counter_bit2)
- LP06 = LP06 XOR (Dall AND Counter_bit3)
- LP07 = LP07 XOR (Dall AND Counter_bit3)
- LP08 = LP08 XOR (Dall AND Counter_bit4)
- LP09 = LP09 XOR (Dall AND Counter_bit4)
- LP10 = LP10 XOR (Dall AND Counter_bit5)
- LP11 = LP11 XOR (Dall AND Counter_bit5)
- LP12 = LP12 XOR (Dall AND Counter_bit6)
- LP13 = LP13 XOR (Dall AND Counter_bit6)
- LP14 = LP14 XOR (Dall AND Counter_bit7)
- LP15 = LP15 XOR (Dall AND Counter_bit7)

C. Error detect case

LP 15	LP 14	LP 13	LP 12	LP 11	LP 10	LP 09	LP 08	LP 07	LP 06	LP 05	LP- 04	LP 03	LP 02	LP 01	LP 00	CP 05	CP 04	CP 03	CP 02	CP 01	CP 00	code stored in Flash
																						XOR
LP 15	LP 14	LP 13	LP 12	LP 11	LP 10	LP 09	LP 08	LP 07	LP 06	LP 05	LP- 04	LP 03	LP 02	LP 01	LP 00	CP 05	CP 04	CP 03	CP 02	CP 01	CP 00	code read generated
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	No Error
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0	1	Correctable
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	2	1	Uncorrectable
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Code Error

No error

Since there is no difference between the code stored in the flash and the one generated after the read, it is assumed that there is no error in this case.

Correctable error

Since all parity bit pairs (CP00 and CP01),.....,(LP014 and LP15) have one error and one match in them as the result of the comparisons between the code stored in flash and the one generated after the read, this case is considered to be a correctable error.

Uncorrectable error

In this case, both CP00 and CP01 are in error as the results of the comparison between the code stored in flash and the one generated after the read. This represents a multiple bit error, and is therefore uncorrectable.

ECC code area error

When only one bit (LP13) is erroneous (the result of the comparison between the code stored in flash and the one generated after the read), it is assumed that the error occurred in the ECC area and not in the data area. This is because a single erroneous data bit should cause a difference in half of the Line Parity bits (by changing Dall, which affects half of the Line Parity bits based on the current counter value), and half of the Column Parity bits (based on the equations for the Column Parity bits, which each include half of the data bits).

D. Error Correction

The error location can be found by XORing the ECC parity bits stored in the flash with ECC bits calculated from the data read out of the flash. The error location is assembled from XORing the following stored and computed line parity bits:

(LP15,LP13,LP11,LP09,LP07,LP05,LP03,LP01) - this gives the byte address.

(CP05,CP03,CP01) - this gives the bit number.

5.10.2 4-bit ECC Code

There are several algorithms for a 4-bit or more ECC code. BCH (Bose, Ray-Chaudhuri, Hocquenghem) is more popular because of its improved efficiency over Reed-Solomon code. Although we provide software BCH code it is impractical to use software 4-bit ECC. Both codes need too many microprocessor cycles. For one 256KB flash block, it has $256 * 1024 * 8 = 2\text{Mbit}$. ECC need 48 bit and for each bit we need about 10 instructions to do the computation. So totally it needs $2\text{M} * 48 * 10$ (~1 billion) instructions to get one block's ECC! Even on a 2G Hz Windows PC, it needs about 400-500 milliseconds.

To use MLC flash, you need a flash controller which has built-in 4 or more ECC engine or some FPGA.

5.10.3 Add Hardware ECC

Hardware ECC should be done within the low level driver the function `asm_Read_Page()` and `asm_Write_Page()`. We will only use the first six (6) byte of the spare area of each page. Other bytes within the spare area can be used by the hardware ECC to store the generated ECC code.

If hardware ECC does not correct the error, you can use our software ECC code to correct it.

Here are some things we know about hardware ECC or NAND flash controllers:

1. Atmel AT91SAM9 processors have a 1-bit ECC controller.
2. Freescale i.MX31 processor has built-in NAND flash controller but only 1-bit ECC
3. NXP LPC3180/3250 processor has built-in MLC controller and Reed-Solomon ECC engine.
4. TI Davinci DM355 has built-in ECC engine for 1-bit and 4-bit ECC
5. TI OMAP 35xx processor has built-in ECC engine for 1-bit (Hamming) and 4-bit (BCH) ECC
6. Eureka EP501 is a NAND flash controller IP but only has 1-bit ECC
7. Micron has ECC Module but only 1-bit ECC

6. Size and Performance

Code Size

Code size will vary widely depending upon CPU, compiler, and optimization level. Below are two examples. The Mini Library excludes infrequently used VFile API calls. Driver Only is just the flash driver, without VFile.

<u>CPU and Compiler</u>	<u>Full Library</u>	<u>Mini Library</u>	<u>Driver Only</u>
ARM High C/C++ 4.2f	23.5 KB	19.5 KB	12 KB
X86 Borland C++ 32-bit	17.6 KB	15.3 KB	8.5 KB

Data Size (RAM Requirement)

Numbers shown are examples based upon 16MB flash and the indicated FFS configuration settings.

Flash Driver

Values Used Below

CACHE_BLOCK_NUMBER (2) and PAGES_PER_CACHE_BLOCK (32) are defined in the flashcnf.h

BlockNum = total number of flash data blocks = 1024

BlockDataSize = flash data block size = 16384

PageDataSize = flash data page size = 512

PageSize = flash page block + spare area = 512 + 16 = 528

FlashSize = BlockNum * BlockDataSize = 1024 * 16384 = 16777216 bytes (16MB)

CacheBlockSize = PAGES_PER_CACHE_BLOCK * PageSize = 32 * 528 = 16896

CacheSize = CACHE_BLOCK_NUMBER * CacheBlockSize = 2 * 16896 = 33792

BlockTableSize = sizeof(BLOCKNODE) * BlockNum = 2 * 1024 = 2048

WearCounterSize = sizeof(BLOCKNODE) * BlockNum = 2 * 1024 = 2048

TmpBufSize = CacheBlockSize = 16896

TmpWearLevelSize = 1024

DriverRAMSize =

BlockTableSize + WearCounterSize + CacheSize + TmpBufSize =

2048 + 2048 + 33792 + 16896 = 54784 (0.3% of flash)

This memory is allocated when the flash driver is initialized.

DriverTempRAMSize =

TmpWearLevelSize = 1024

This memory is allocated and freed while the system is running.

If you want to reduce RAM usage, you can reduce PAGES_PER_CACHE_BLOCK from 32 to 1, for example. This would reduce DriverRAMSize to 22048, but the system's performance may not be as good as with the default setting.

Virtual File System

Values Used Below

START_BLOCK_NUM (10) and RESERVED_DATA_BLOCK_NUM (10) are defined in flashcnf.h

BYTES_PER_CLUSTER (512) and MAX_FILES (1024) are defined in vfilecnf.h

OpenFilesNum = 1 (number of files simultaneously open; depends upon application)

ReservedBlockNum = START_BLOCK_NUM + RESERVED_DATA_BLOCK_NUM = 20

$TotalMediaSize = BlockDataSize * (BlockNum - ReservedBlockNum) = 16449536;$
 $FCBSize = (MAX_FILES * sizeof(FCB) + FSID_LEN = 1024 * 20 + 32 = 20512$
 $TotalClusters = (TotalMediaSize - FCBSize) / (BYTES_PER_CLUSTER +$
 $sizeof(FATNODE)) = (16449536 - 20512) / (512 + 2) = 31963$
 $FATSize = TotalClusters * sizeof(FATNODE) = 31963 * 2 = 63926$
 $TmpBuf = TotalClusters = 31963$
 $FileBuf = OpenFilesNum * (sizeof(filehandle) + BYTES_PER_CLUSTER) = 1 * (16 + 512) = 528$

VFileRAMSize = FCBSize + FATSize = 20512 + 63926 = 84438; to cluster alignment = 84480
 This memory is allocated when mounting the file system.

VFileTempRAMSize = TmpBuf + FileBuf = 31963 + 528 = 32491
 This memory is allocated and freed while the system is running.

VFileTotalRAMSize = 84480 + 32491 = 116971 (0.7% of flash)

If you want to reduce RAM usage, you can reduce MAX_FILES and increase BYTES_PER_CLUSTER. For example, set MAX_FILES to 100 and BYTES_PER_CLUSTER to 1024. This would reduce VFileRAMSize to 34816. The VFileTempRAMSize will be reduced to 17070. Total would be 51886 vs. 116971.

Performance

NAND Flash and Test Specifications

1 second = 1000 ms = 1000 * 1000 us = 1000 * 1000 * 1000 ns

CPU: Coldfire core @ 66 MHz

Flash Bus: 220 ns => 4.54 MHz (Reading one 8-bit byte from the NAND flash chip takes 220 ns for this flash memory, even though the flash chip requires only 50 ns.)

Flash Chip: 8-bit. 528 bytes/per page and 32 pages/per block. No bad blocks.

Delay between each page read operation (including command and address time): 50 us

Delay between each page write operation (including command and address time): 300 us

Test File: 4.0MB, non-fragmented in flash. For the write test, all blocks are empty (so there are no erase operations)

Read Tests

Theoretical Minimum Time

for each Page (512 valid data): 220 ns * 528 + 50 us = 167 us

for each Block (512*32 valid data): 167 us * 32 = 5.344 ms

for the whole Test File: 5.344 ms * (4096/16) = 1.37 s

1.37 seconds is the hardware limitation. It is the shortest possible time to read 4MB of data from this flash memory.

Read Test #1: Without ECC Checking

Actual Testing Result: 2.0 s

Overhead: 2.0 - 1.37 = 0.63 s

Write Tests

Theoretical Minimum Time

for each Page (512 valid data): 220 ns * 528 + 300 us = 416 us

for each Block (512*32 valid data): 416 us * 32 = 13.3 ms

for the whole Test File: 13.3 ms * (4096/16) = 3.4 s

3.4 seconds is the hardware limitation. It is the shortest possible time to write 4MB of data to this flash memory.

Write Test #1: Without ECC Generation (and without reading back data to verify)

Actual Test Result: 5 s

Overhead: 5-3.4 = 1.6 s

The MCF5282 and LPC2468 tests do not use DMA or software ECC.

<u>Samsung 16MB on MCF5282</u>	<u>Reading (KB/s)</u>	<u>Writing (KB/s)</u>
NAND Driver raw data	2730	1277
smxFFS with Flash Driver	2048	455

<u>ST 128MB on LPC2468</u>	<u>Reading (KB/s)</u>	<u>Writing (KB/s)</u>
NAND Driver raw data	1795	1638
smxFFS with Flash Driver	890	487

AT91SAM9263EK, CPU running at 240MHz. Bus speed is 120MHz. It is using external SDRAM. Instruction Cache is on and Data Cache is off. ECC is done in software. SAM9263 has hardware ECC engine but we have not implemented a driver for it.

<u>K9F2G08U on SAM9263 (non DMA)</u>	<u>Reading (KB/s)</u>	<u>Writing (KB/s)</u>
NAND low level raw data	5041	4161
Flash Driver	4266	1969
smxFFS with Flash Driver	3961	1785
smxFS with Flash Driver	3373	1689

<u>K9F2G08U on SAM9263 (DMA)</u>	<u>Reading (KB/s)</u>	<u>Writing (KB/s)</u>
NAND low level raw data	12483	7281
Flash Driver	8533	3657
smxFFS with Flash Driver	7787	3038
smxFS with Flash Driver	5953	2852

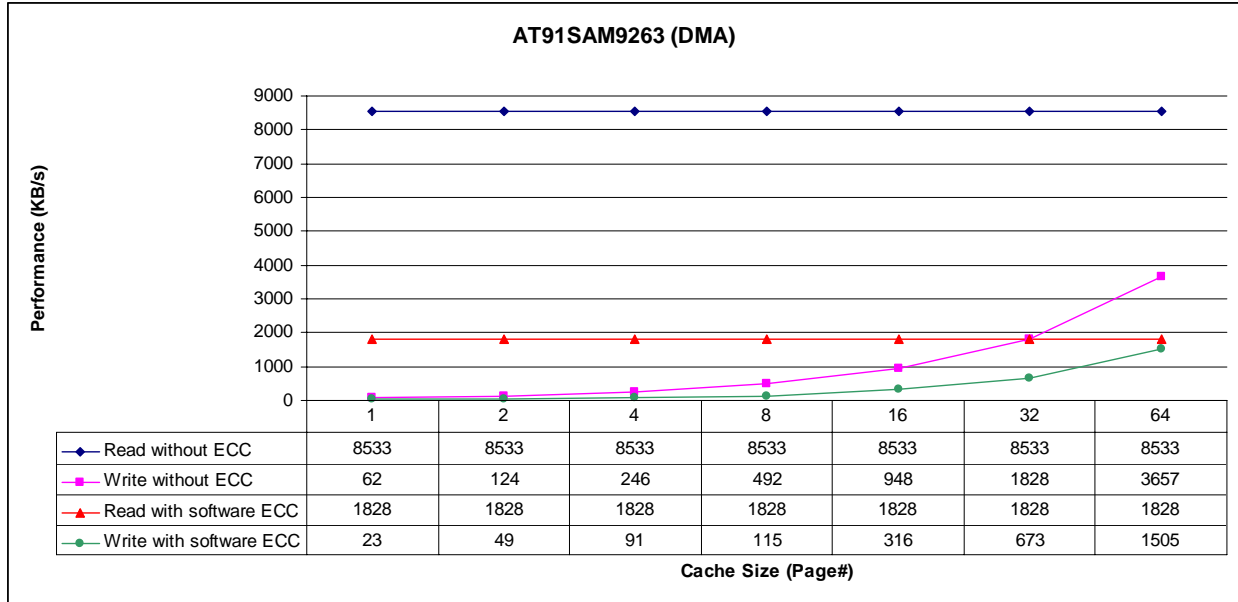
<u>K9F2G08U on SAM9263 (SW ECC+DMA)</u>	<u>Reading (KB/s)</u>	<u>Writing (KB/s)</u>
Flash Driver	1828	1505
smxFFS with Flash Driver	1801	883
smxFS with Flash Driver	1719	847

NAND Flash Performance vs. Cache Size

The following measurements are based upon a 256 MB NAND flash chip on an AT91SAM9263EK using the smxNAND flash driver. Entries are KB/sec. CPU is running at 240MHz. Bus speed is 120MHz. It is using external SDRAM. Instruction Cache is on and Data Cache is off. Driver uses DMA to transfer data.

CACHE BLOCKS	PAGES EACH	READ (1)	WRITE (1)	READ (2)	WRITE (2)
1	1	8533	62	1828	23
1	2	8533	124	1828	49
1	4	8533	246	1828	91
1	8	8533	492	1828	115
1	16	8533	948	1828	316
1	32	8533	1828	1828	673
1	64	8533	3657	1828	1505
2	64	8533	3657	1828	1505

(1) without ECC (2) with software ECC



The above write measurements are writing every page of the flash chip, sequentially. Increasing the number of pages per cache block increases write performance considerably up to caching one block. This is because the smxNAND driver must open a new flash block every time the cache is flushed to it. (I.E. it must copy what has been written so far to a new cache block, then write the contents of the cache after.) Hence, a considerable performance penalty is paid for caching less than one block. Also wear on the flash chip is increased. For reading, a one page cache is sufficient. Software ECC greatly decreases performance. Because of this, we recommend disabling software ECC and using hardware ECC.

For read/modify/write operations, caching multiple blocks may improve performance, depending upon file sizes and the nature of the operations performed. This is true of any file system.

Virtual File System API

Data Types

These are defined in *vfilefla.h* unless otherwise noted.

NFILEHANDLE Pointer to FILE_TAG structure. See section 4.5 File Handle for a summary of its fields.

FIND_DATA Structure containing 2 fields:
ADDRESSTYPE size;
byte name[FILENAMEBUF_LEN];

Basic Calls

int **v_ffs_fmount** (void)

Summary Mount the NAND Flash File System.

Descr This function must be called before calling any other FFS API functions. The return value must be checked to ensure PASS. Otherwise, making any FFS calls will cause unpredictable results.

Parameters none

Returns PASS success
FAIL FFS could not initialize the hardware or it cannot find a valid block table on the flash (the macro AUTO_FORMAT_FLASH defined in flashcnf.h must be set to '0'). You can call v_ffs_format() to force erase all the data on the flash device and rebuild the file system on it later. If the macro AUTO_FORMAT_FLASH is set to '1' then v_ffs_fmount() will erase the whole flash device automatically and rebuild the block table when FFS is mounted on the flash device for the first time. If you mount it again, this function call will return PASS.

See Also v_ffs_funmount()
v_ffs_format()

Example

```
void _cdecl appl_init()
{
    if(v_ffs_fmount() == FAIL)
        wr_string(0,0,WHITE,BLACK,!BLINK,"Error mounting file system.");
    else
        wr_string(0,0,WHITE,BLACK,!BLINK,"File system mounted.");
}
```

void **v_ffs_funmount** (void)

Summary Unmount the NAND Flash File System.

Descr This is the last FFS API call that should be made at exit. This function releases all resources allocated by v_ffs_fmout. Close all files with v_ffs_fclose() before calling this function. Do not call any other FFS function after this is called.

Parameters none

Returns none

See Also v_ffs_fmout()

Example

```
void _cdecl quit_appl()
{
    v_ffs_funmount();
}
```

int **v_ffs_format** (void)

Summary Format the NAND Flash File System.

Descr You can call this function to force FFS to format the flash device. **ALL YOUR DATA ON THE FLASH WILL BE LOST AFTER THIS CALL.** Normally, you should call this function if the v_ffs_fmout() function returns FAIL for the first time FFS is mounted on a new flash device.

Parameters none

Returns PASS success
FAIL FFS could not initialize the hardware or it could not format the flash device.

See Also v_ffs_fmout()

Example

```
void quit_appl()
{
    if(FAILL == v_ffs_fmout())
        v_ffs_format();
}
```

NFILEHANDLE **v_ffs_fopen** (char *filename, const char *mode)

Summary Open one file for read/write access.

Descr This function must be called before any file access operations. This function will open the file specified by filename with the specified access mode. It returns the file handle. Do not directly access the fields of the structure pointed to by the file handle.

The file is opened in binary mode. There is no text mode support. It is fine to pass “rb” instead of “r”, for example, but it is not necessary. If other characters are passed in addition to the characters below, they are ignored (e.g. “rt”).

Parameters filename: The file name format is 8.3 by default. Any printable char is accepted and the name is case sensitive. There is no path support.
mode: supported access modes are as follows (other characters are ignored):
"r" Opens for reading only. If the file does not exist or cannot be found, this call fails. The file pointer starts at the beginning of the file.
"w" Opens an empty file for reading and writing. If the given file exists, its contents are destroyed.
"a" Opens a file for reading and writing. The file pointer starts at the end of the file.
"r+" Opens for both reading and writing. (The file must exist.) The file pointer starts at the beginning of the file.
"w+" Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.
"a+" Same as “a”.

A file can be opened for reading with mode “r” by multiple tasks simultaneously, as long as there are enough memory resources (i.e. file handle structures and file cache memory). If one file is opened with the “r+” mode, the second open request of “r+” mode will be refused.

Returns file handle success
NULL file not found or other error; do not pass a NULL handle to other API calls

See Also v_ffs_fclose()

Example

```
/* single open request */
NFILEHANDLE fp;
fp = v_ffs_fopen("test.bin", "rb");
if(fp != NULL)
{
    v_ffs_fread(...);
    v_ffs_fclose(fp);
}

/* multiple opens of one file for read only*/
NFILEHANDLE fp1, fp2;
fp1 = v_ffs_fopen("test.bin", "rb");
fp2 = v_ffs_fopen("test.bin", "rb");
...

/* attempt to open one file multiple times for reading and writing */
NFILEHANDLE fp1, fp2;
fp1 = v_ffs_fopen("test.bin", "rb");
fp2 = v_ffs_fopen("test.bin", "r+b"); // this call will fail
...
```

int **v_ffs_fclose** (NFILEHANDLE filehandle)

Summary Close an opened file.

Descr Closing a file causes all the data to be flushed to the flash memory. All resources allocated by `v_ffs_fopen()` are released. Once the file is closed, the file handle is no longer valid, so do not use it in another API call.

Parameters filehandle: file handle returned by the `v_ffs_fopen()`.

Returns 0 in all cases

See Also `v_ffs_fopen()`

Example

```
NFILEHANDLE fp;
fp = v_ffs_fopen("test.bin", "wb");
if(fp != NULL)
{
    v_ffs_fwrite(...);
    v_ffs_fclose(fp);
}
```

int **v_ffs_fread** (void *buf, int size, int items, NFILEHANDLE filehandle)

Summary Read some data from an open file.

Descr This function reads up to (*items * size*) bytes from the current file position in the file and stores them in *buf*. The file pointer is increased by the number of bytes actually read. The file pointer position is indeterminate if an error occurs. The value of a partially read item cannot be determined.

Parameters buf: buffer to store the returned data in.
size: item size in bytes.
items: maximum number of items to be read.
filehandle: file handle returned by `v_ffs_fopen()`.

Returns value number of items read
0 error or reach the end of file

See Also `v_ffs_fopen()`, `v_ffs_fwrite()`

Example

```
NFILEHANDLE fp;
char buf[20];
fp = v_ffs_fopen("test.bin", "rb");
if(fp != NULL)
{
    v_ffs_fread(buf, 1, 20, fp); // if "test.bin" file size is 0, this call will return 0.
    v_ffs_fclose(fp);
}
```

int **v_ffs_fwrite** (void *buf, int size, int items, NFILEHANDLE filehandle)

Summary Write some data to an open file.

Descr This function writes up to (*items * size*) bytes from *buf* to the file starting at the current file position in the file. The file pointer is increased by the number of bytes actually written. The file pointer position is indeterminate if an error occurs. The value of a partially written item cannot be determined.

If the file was opened in read-only mode "r", v_ffs_fwrite() will return 0 and no data will be written to the file.

Parameters

buf:	Pointer to data to be written.
size:	Item size in bytes.
items:	Maximum number of items to be writtn.
filehandle:	file handle returned by the v_ffs_fopen().

Returns

value	number of items written
0	there is no free space to write the data
-1	there is no free space to write the data because some block(s) have become bad.

See Also v_ffs_fopen(), v_ffs_fread()

Example

```
/* normal write operation */
NFILEHANDLE fp;
char buf[20]="This is a test.";
fp = v_ffs_fopen("test.bin", "wb");
if(fp != NULL)
{
    v_ffs_fwrite(buf, 1, 20, fp);
    v_ffs_fclose(fp);
}

/* write to a read-only file will return error */
NFILEHANDLE fp;
char buf[20]="This is a test.";
fp = v_ffs_fopen("test.bin", "rb");
if(fp != NULL)
{
    v_ffs_fwrite(buf, 1, 20, fp); /* returns 0 and no data is written */
    v_ffs_fclose(fp);
}
```

int **v_ffs_fseek** (NFILEHANDLE filehandle, int offset, int whence)

Summary Moves the file pointer to the specified location.

Descr This function moves the file pointer associated with *filehandle* to a new location that is *offset* bytes from the origin, *whence*. The next read/write operation on the file takes place at this new location. You can NOT use this function to reposition the pointer anywhere in a file. Attempting to move the pointer before the beginning of file is an error; the pointer is moved to the beginning of file and the return value is 0. if the file is opened by Read/Write mode, move the pointer beyond the end of file will extend the file but the data in this new area is unpredictable until you write data there.

Parameters filehandle: file handle returned by the v_ffs_fopen()
offset: number of bytes from *whence*
whence: initial position; three predefined constants are:

FILE_SEEK_CUR	Current position of file pointer
FILE_SEEK_END	End of file
FILE_SEEK_SET	Beginning of file

Returns 0 success
1 fail

See Also v_ffs_fopen(), v_ffs_fread(), v_ffs_fwrite()

Example

```
/* normal seek operation */
NFILEHANDLE fp;
char buf[20];
fp = v_ffs_fopen("test.bin", "rb");
if(fp != NULL)
{
    v_ffs_fseek(fp, 10, FILE_SEEK_SET);
    v_ffs_fread(buf, 1, 20, fp);
    v_ffs_fclose(fp);
}

/* seek beyond the file area will cause error if it is Read-Only */
NFILEHANDLE fp;
char buf[20]="This is a test.";
fp = v_ffs_fopen("test.bin", "rb");
if(fp != NULL)
{
    v_ffs_fseek(fp, 10, FILE_SEEK_END); // this will move the pointer to the end of file.
    v_ffs_fclose(fp);
}

/* seek beyond the file area will increase the files size if it is Read/Write */
NFILEHANDLE fp;
char buf[20]="This is a test.";
fp = v_ffs_fopen("test.bin", "wb");
if(fp != NULL)
{
    v_ffs_fseek(fp, 10, FILE_SEEK_END); // file size is 10 bytes now but the contents are unpredictable.
    v_ffs_fclose(fp);
}
```

void **v_ffs_fdelete** (char * filename)

Summary Delete a file.

Descr This function deletes the file indicated by *filename*. If the file is currently open or does not exist, this function does nothing and returns.

Parameters filename: the name of the file to be deleted

Returns none

See Also v_ffs_findfile()

Example

```
NFILEHANDLE fp;
v_ffs_fdelete("test.bin");
v_ffs_fdelete("test.bin"); // attempting to delete a file that does not exist will not cause any damage.
fp = v_ffs_fopen("data.dat", "rb");
v_ffs_fdelete("data.dat"); // attempting to delete an open file does nothing; just returns.
v_ffs_fclose(fp);
```

int **v_ffs_findfile** (char *filename)

Summary Test if a file exists.

Descr The function searches for the file specified by *filename* but without the need to open it first. If the file exists, a positive value is returned; otherwise 0 is returned. If the file is opened by others, this function will also return the correct result.

Parameters filename: the name of the file to find

Returns >0 file found
0 file not found

See Also v_ffs_findfirst(), v_ffs_findnext()

Example

```
if(v_ffs_findfile("test.dat") > 0)
    printf("Found test.dat");
```

int **v_ffs_filelength** (char *filename)

Summary Return the length of a file, in bytes.

Descr This function returns the length of the file specified by *filename* if the file exists. If it does not exist, -1 is returned. If the file is currently open, the latest file length is returned.

Parameters filename: the name of the file whose length to determine

Returns >= 0 length of file
 -1 file not found

See Also v_ffs_findfirst(), v_ffs_findnext()

Example printf("%d", v_ffs_filelength("test.dat"));

char * **v_ffs_fgetversion** (void)

Summary Return the file system version number.

Descr This function returns FS_FLAG_STRING defined in vfilecnf.h.

Parameters none

Returns version string

See Also

Example printf("The file system version is %s", v_ffs_fgetversion());

Extended Calls

void **v_ffs_rewind** (NFILEHANDLE filehandle)

Summary Moves the file pointer to the beginning of the file.

Descr This is equivalent to fseek (filehandle, 0, FILE_SEEK_SET).

Parameters filehandle: file handle returned by v_ffs_fopen()

Returns none

See Also v_ffs_fopen(), v_ffs_fseek()

Example

```
NFILEHANDLE fp;
char buf[20];
fp = v_ffs_fopen("data.dat", "rb");
v_ffs_fread(buf, 1, 20, fp);
v_ffs_rewind(fp);
v_ffs_fclose(fp);
```

int **v_ffs_fflush** (NFILEHANDLE filehandle)

Summary Flush all data associated with the file handle to the storage media.

Descr The file system uses a memory cache to store file data to minimize writes to the storage media. This function forces all cached data for this file to be written to the real storage media (NAND flash).

Parameters filehandle: file handle returned by v_ffs_fopen()

Returns 1 in all cases

See Also v_ffs_fopen(), v_ffs_fwrite()

Example

```
NFILEHANDLE fp;
char buf[20]="Test data";
fp = v_ffs_fopen("data.dat", "r+b");
v_ffs_fwrite(buf, 1, 20, fp);
v_ffs_fflush(fp);
v_ffs_fclose(fp);
```

int **v_ffs_ftell** (NFILEHANDLE filehandle)

Summary Determines the current file pointer position.

Descr This function returns the current file pointer.

Parameters filehandle: file handle returned by v_ffs_fopen()

Returns value file pointer position

See Also v_ffs_fopen(), v_ffs_fseek()

Example

```
NFILEHANDLE fp;
char buf[20]="Test data";
fp = v_ffs_fopen("data.dat", "r+b");
v_ffs_fwrite(buf, 1, 20, fp);
v_ffs_fseek(fp, v_ffs_ftell(fp) -1 , FILE_SEEK_SET );
v_ffs_fclose(fp);
```

void **v_ffs_fsetend** (NFILEHANDLE filehandle)

Summary Truncates a file at the current file pointer.

Descr This function discards all data beyond the current file pointer. The file length is then updated to the current file pointer.

Parameters filehandle: file handle returned by v_ffs_fopen()

Returns none

See Also v_ffs_fopen(), v_ffs_fseek(), v_ffs_fwrite()

Example

```
NFILEHANDLE fp;
char buf[20]="Test data";
fp = v_ffs_fopen("data.dat", "r+b");
v_ffs_fwrite(buf, 1, 20, fp);
v_ffs_fseek(fp, v_ffs_ftell(fp) -10 , FILE_SEEK_SET );
v_ffs_fsetend(fp); //discard 10 bytes
v_ffs_fclose(fp);
```

int **v_ffs_feof** (NFILEHANDLE filehandle)

Summary Tests for end-of-file on a file.

Descr This function returns a nonzero value if the file pointer is at the end of file. It returns 0 if the current position is not end of file.

Parameters filehandle file handle returned by v_ffs_fopen()

Returns >0 EOF
0 not EOF

See Also v_ffs_fopen(), v_ffs_fseek(), v_ffs_fwrite(), v_ffs_fread()

Example

```
NFILEHANDLE fp;
char buf[20]="Test data";
fp = v_ffs_fopen("data.dat", "r+b");
while(!v_ffs_feof(fp))
    v_ffs_fread(buf, 1, 20, fp);
v_ffs_fclose(fp);
```

int **v_ffs_rename** (char * oldname, char * newname)

Summary Renames a file.

Descr This function renames the file specified by *oldname* to the name given by *newname*. The old name must be an existing file. The new name must not be the name of an existing file.

Parameters oldname: the old file name
newname: the new name for the file

Returns 0 success; file renamed
>0 fail; *oldname* does not exist or *newname* is used by other file

See Also v_ffs_findfile()

Example

```
NFILEHANDLE fp;
char buf[20]="Test data";
fp = v_ffs_fopen("data.dat", "w+b");
v_ffs_fwrite(buf, 1, 20, fp);
v_ffs_fclose(fp);
v_ffs_rename("data.dat", "data1.dat");
```

int **v_ffs_findfirst** (char * filespec, FIND_DATA * fileinfo)

Summary Provides information about the first instance of a file whose name matches the name specified by the *filespec* argument.

Descr If successful, this function returns a unique search id identifying the file matching the *filespec* specification, which can be used in a subsequent call to `v_ffs_findnext()`. Otherwise, `v_ffs_findfirst()` returns `-1`.

Parameters filespec: the search spec which may include wildcard "*" and "?". The following are valid *filespec*:
 "*.*"
 "*.*.dat"
 "test?.*"
 "test?2.dat"
fileinfo: the returned file info which includes the file's name and size.

Returns id file found matching *filespec*
-1 no file found

See Also `v_ffs_findfile()`, `v_ffs_findnext()`

Example

```
FIND_DATA fileinfo; // (see definition of FIND_DATA at beginning of this API section)
int id;
id = v_ffs_findfirst("data*.dat", &fileinfo);
while(id > 0)
{
    printf("File Name: %s, File Size: %d\n", fileinfo.name, fileinfo.size);
    id = v_ffs_findnext(id, &fileinfo);
}
```

int **v_ffs_findnext** (int id, FIND_DATA * fileinfo)

Summary Find the next file, if any, whose name matches the *filespec* argument in a previous call to `v_ffs_findfirst()`, and returns information about it in the *fileinfo* structure.

Descr If successful, this function returns a unique search id identifying the next file it finds that matches the *filespec* specification that was passed to `v_ffs_findfirst()`. Otherwise, returns `-1`.

Parameters id: the search id returned by the last `v_ffs_findnext()` or `v_ffs_findfirst()` call
fileinfo: the returned file info which include the file's name and size

Returns id file found matching *filespec*
-1 no file found

See Also `v_ffs_findfile()`, `v_ffs_findfirst()`

Example

```
FINDDATA fileinfo; // (see definition of FIND_DATA at beginning of this API section)
int id;
id = v_ffs_findfirst("data*.dat", &fileinfo);
while(id > 0)
{
    printf("File Name: %s, File Size: %d\n", fileinfo.name, fileinfo.size);
    id = v_ffs_findnext(id, &fileinfo);
}
```

int **v_ffs_fgetfilenum** (char * filespec)

Summary Determines the total number of files, if any, whose names match the *filespec*.

Descr If successful, this function returns the total number of files whose names match *filespec*.

Parameters filespec: the search spec which may include wildcard '*' and '?'

Returns num number of matching files

See Also v_ffs_findfile(), v_ffs_findfirst(), v_ffs_findnext()

Example

```
printf("The total file number which matches %s is %d", "data*.*", v_ffs_fgetfilenumber("data*.*");
```

int **v_ffs_flushall** (void)

Summary Flushes all data to the storage media.

Descr The file system uses caches to store data in memory to minimize writes to the storage media. This function force all cached data to be written to the real storage media (NAND flash).

Parameters none

Returns 1 in all cases

See Also v_ffs_fopen(), v_ffs_flush()

Example

```
NFILEHANDLE fp;
char buf[20]="Test data";
fp = v_ffs_fopen("data.dat", "r+");
v_ffs_fwrite(buf, 1, 20, fp);
v_ffs_flushall();
v_ffs_fclose(fp);
```

float **v_ffs_freespace** (void)

Summary Determines the percentage of free space on the storage media.

Descr This function returns the percentage, not the size of the free space.

Parameters none

Returns percentage of free space

See Also v_ffs_fmout(), v_ffs_freesize()

Example
printf("The free space percentage is %d %%", v_ffs_freespace());

int **v_ffs_freesize** (void)

Summary Determines the free space (bytes) on the storage media.

Descr This function returns the free bytes of the storage media.

Parameters none

Returns free bytes number of the space

See Also v_ffs_fmout(), v_ffs_freespace()

Example
printf("The free size is %d ", v_ffs_freesize());

int **v_ffs_devicemounted** (void)

Summary Determines if the device is mounted.

Descr Test if the device mounted.

Parameters none

Returns currently, always return PASS

See Also v_ffs_fmout()

Example
If(PASS == v_ffs_devicemounted())
 printf("The NAND flash has been mounted.");

int **v_ffs_formaterror** (void)

Summary Determines if the file system structure has error.

Descr If the file system's signature area is OK but the FAT has crossed cluster chains, this function will report an error.

Parameters none

Returns PASS if has error. FAIL if the structure is OK.

See Also v_ffs_fmout()

Example If(PASS == v_ffs_formaterror())
 printf("The NAND flash file system format has error.");

Appendix A: Preprogramming Flash

If you solder a new flash chip to your board and run your application that includes smxFFS (or smxFS + NAND flash driver), the filesystem structure will be created on the flash chip automatically. The software takes care of the details of doing the low-level flash format (including marking any bad blocks encountered), and formatting it with the high-level filesystem (e.g. smxFFS or FAT). If your device must have some files already saved in the filesystem, one approach is to run your device and copy the files to it. But this process may be too time consuming.

For manufacturing, it is convenient to be able to preprogram the flash chips before soldering them to the board, especially to program several at once (gang programming). However, this is complicated because each flash chip may have bad blocks in different locations, so the image that must be written to each may vary. With some flash programmers it is possible to define an algorithm for programming the flash, but this is complex and problematic because:

1. The algorithm must be changed if any changes are made to the internals of the flash driver (smxFFS).
2. The algorithm differs depending on the high-level filesystem (smxFFS, FAT12, FAT16, FAT32).
3. It is not possible to gang-program the devices at the same time because the bad blocks are in different places. If a gang programmer is used, all flash chips must be programmed individually.

Our solution is a hybrid of the two approaches. It makes the assumption that the first n flash blocks on a device are almost always good for the first few cycles of writing, where n is the number of flash blocks needed to store the initial image. Typically, the amount of space occupied by the initial files is a small fraction of total disk space. These are the steps we recommend:

1. Use our **FlashImage** utility to create an image of your flash (BIN\FlashImage). This utility creates the image in a file on your development PC. This image assumes there are no bad blocks in this area of the flash. A config file (cfg.h) is used to specify the flash type and list the files to store in the image. It is necessary to configure and build this utility. See the readme in its directory for directions.
2. Supply this image to your gang programmer to program all devices simultaneously.
3. Do a verify operation on each chip. The ones that pass are soldered to the boards. The others are collected; they can be programmed manually by running the application software on them. We expect that a very small percentage will require this.

Any bad blocks in the remainder of the media will be handled as encountered during normal use of your device.

The key point is that the utility is built using the same flash driver and filesystem code that is linked to your application. If any changes are made to the internals of the flash driver, it is only necessary to recompile the utility. It is not necessary to create and maintain complicated flash programmer files.

Note: The above solution cannot be used when you need to preprogram a large amount of data in the flash chip, because the bigger the image is, the more likely it is to span an area that has bad blocks. The smaller your image is, the higher your preprogramming yields will be. See the next appendix for preprogramming and handling bad blocks.

Appendix B: Preprogramming Flash and Handling Bad Blocks

Appendix A provides a way to preprogram flash without handling bad blocks. If you need to preprogram a large amount of data into the flash chip, the chances are high that your image will span an area with bad blocks. In this case, your flash burner must know the internal data structure of the flash driver so it can replace each bad block with a new block. Here are the steps to do this.

1. Find and read the Block Table in the image file into a memory buffer.
Calculate the block table size, $wBlockNum * sizeof(BLOCKNODE)*2$.
Allocate a buffer to hold the block table.
Calculate the page number of the block number.
For each block of the image file, check the block table flag. The block table flag is located on the last page of the block table, at spare area offset `BLOCK_TABLE_FLAG_OFFSET`, and the value should be `BLOCK_TABLE_FLAG`.
For each possible block table area within that block table, check if that block table area is valid, not the discarded one.
If we find the valid block table area, then read it out and exit that step.
2. Mark the Block Table as in-progress.
Write `IN_PROGRESS_BLOCK_TABLE` flag into offset `IN_PROGRESS_BLOCK_OFFSET`.
3. Find an empty block in the Block Table, and mark that empty block in the block table to be used. Mark the bad block in the block table to be bad.
Scan the whole block table to find a block with status bit `SPARE_BLOCK`.
Replace the bad block entry by that empty block table entry.
4. Write the data to the new empty block. If it is still bad, go to step 3 until we get a correct one.
5. Find a new block table position and write the new block table data to it.
Calculate the next block table position.
Check if the new position is a new block.
Update the block table if we need to (new block case).
Write data to the new place.
Write the current block table flag.
6. Mark the old block table as discarded.
Write `DISCARDED_BLOCK_TABLE` flag into offset `DISCARDED_BLOCK_OFFSET`.

We provide sample code to show this, in `badblock.c`. See the function `ReplaceBadBlock()` and `TODO` comments. You can translate that C code into your burner's language.

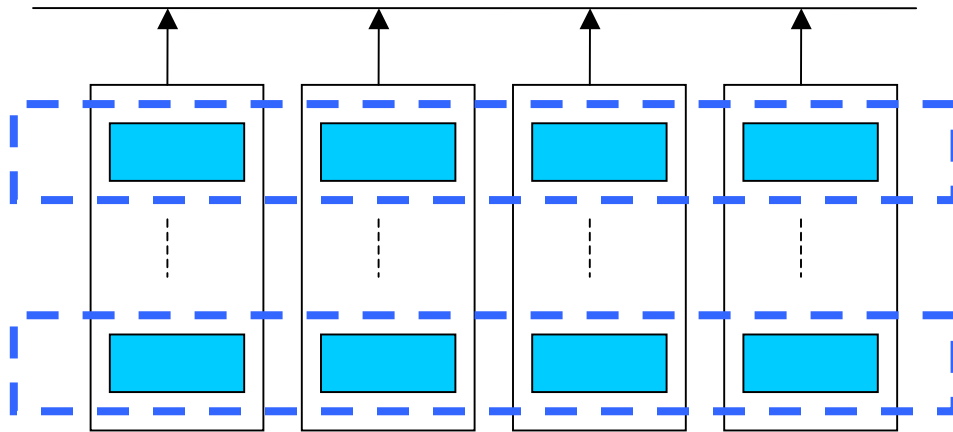
Appendix C: Flash Chip Array

If you need a NAND flash capacity that is larger than a single NAND flash chip or if you want to increase performance, you need to use an array of multiple flash chips. The smxFFS NAND flash driver can treat the whole flash chip array as a virtual single flash chip. You may need to write additional code in the low-level hardware routines. These are defined in flhdw.h

There are two ways to organize the flash chip array: parallel or serial.

C.1 Parallel

Parallel means you are expanding your bus width. Most NAND flash chips use 8-bit or 16-bit bus. You can use four 8-bit NAND flash chips or two 16-bit flash chips to generate a 32-bit bus, as in the following figure:



By using this approach, one physical block on each of the 4 chips will be combined to generate a virtual block that is 4 times bigger.

The advantage of this approach is better performance, since you can begin to program/erase the next physical block when the previous one is still busy; you do not need to wait until the previous operation is done.

The disadvantages are;

1. The flash driver needs to allocate more RAM to cache this bigger virtual block or virtual page.
2. You may need an FPGA to handle the details about how to parallel program/erase multiple chips.
3. The low-level driver routines are more complex.
4. If one chip has a bad block at certain position, then the corresponding block, located at the same position of all flash chips also cannot be used. This can waste a lot of flash.

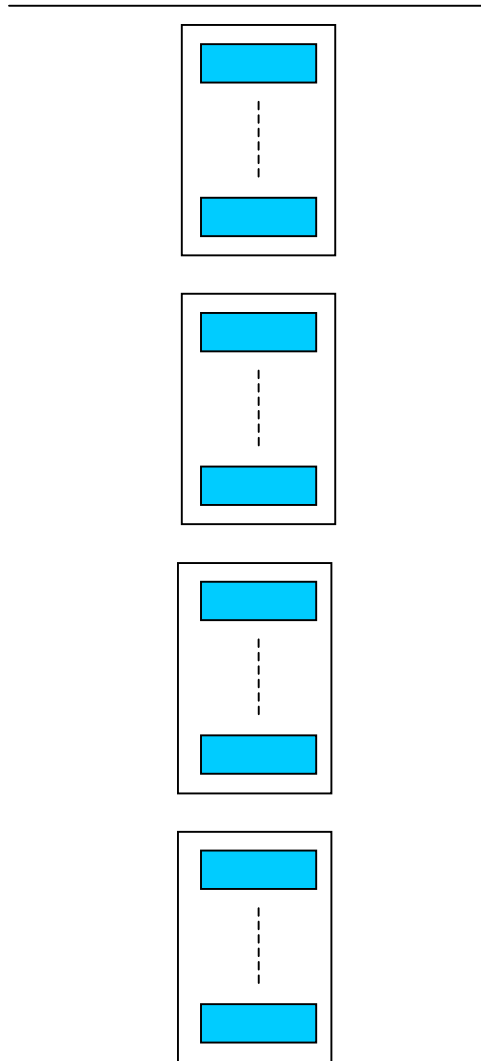
If you are using four 1GB NAND chips to generate a virtual 4GB chip, you will need to report the Device Info to the flash driver as follows. Notice that the block number is the same as for a single chip, but page size and block size are four times larger.

```
uint16 asm_Read_Device_ID(uint iChipID, DEVICE_INFO *pDeviceInfo)
{
    pDeviceInfo->wDeviceType    = 4*1024;
    pDeviceInfo->wBlockNum      = 8192;
```

```
pDeviceInfo->wPagesPerBlock = 64;  
pDeviceInfo->wPageSize      = 4*2112;  
pDeviceInfo->wPageDataSize  = 4*2048;  
pDeviceInfo->wPageSpareSize = 4*64;  
pDeviceInfo->wBlockSize     = 4*135168L;  
pDeviceInfo->wBlockDataSize = 4*131072L;  
return 0;  
}
```

C.2 Serial

Serial means you are expanding the number of blocks. For example, if one chip has 8192 blocks, a virtual flash chip with four chips will have 4*8192 blocks. At any time, the flash driver will only access one of those chips, as shown in the following figure:



By using this approach, block size is the same but the number of blocks is multiplied.

The advantages are:

1. The flash driver only needs to allocate more RAM for the block table.
2. You do not need a special controller to handle operation because the flash driver will not access the four chips at the same time.
3. The low-level routines are relatively simple.
4. Bad blocks in one chip will not affect the other chips.

The disadvantage is that the performance cannot be improved.

If you are using four 1GB NAND chips to generate a virtual 4GB chip, you will need to report the Device Info to the flash driver as follows. Notice that the block number is four times the number for a single chip, but page size and block size are the same.

```
uint16 asm_Read_Device_ID(uint iChipID, DEVICE_INFO *pDeviceInfo)
{
    pDeviceInfo->wDeviceType    = 4*1024;
    pDeviceInfo->wBlockNum      = 4*8192;
    pDeviceInfo->wPagesPerBlock = 64;
    pDeviceInfo->wPageSize      = 2112;
    pDeviceInfo->wPageDataSize  = 2048;
    pDeviceInfo->wPageSpareSize = 4*64;
    pDeviceInfo->wBlockSize     = 135168L;
    pDeviceInfo->wBlockDataSize = 131072L;
    return 0;
}
```

In `asm_Read_Page()`, `asm_Read_Page_Spare()`, `asm_Write_Page()`, or `asm_Write_Page_Spare()`, you need to determine which chip to access by testing the `page_addr` parameter passed to the function. The following is an example for a flash array of four 1GB chips.

```
uint16 asm_Write_Page(uint iChipID, byte* write_data, uint32 page_index, uint offset, uint32 page_size)
{
    if(page_index >=3*8192*64)
    {
        AccessChip3(write_data, page_index - 3*8192*64, offset, page_size);
    }
    else if(page_index >=2*8192*64)
    {
        AccessChip2(write_data, page_index - 2*8192*64, offset, page_size);
    }
    else if(page_index >=1*8192*64)
    {
        AccessChip1(write_data, page_index - 1*8192*64, offset, page_size());
    }
    else
    {
        AccessChip0(write_data, page_index, offset, page_size);
    }
}
```

Unless you need high performance, we recommend using serial organization of your flash array.