

# smxFFS & smxFD

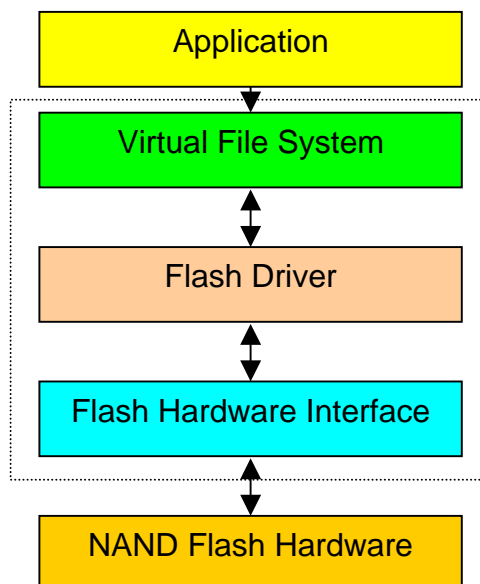
## Flash File System

*smxFFS is a flash file system for use with board-resident arrays of NAND flash memory.<sup>1</sup> It has the standard C library file API (fopen(), fread(), etc). smxFFS can coexist with other file systems such as smxFile and smxCD. It provides a high level of power-fail safety. Starting with v1.6 smxFFS supports very large NAND flash arrays. In some cases the flash size can be up to 512 GB.*

### Product Description

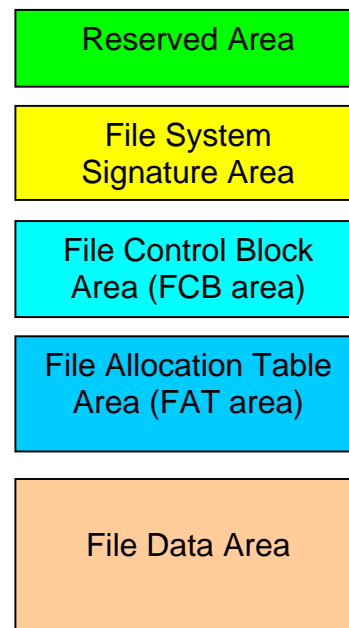
smxFFS has three layers:

1. **Virtual File System (VFS)** provides the standard C library API: fopen(), fread(), fwrite(), fseek(), fclose(), etc. to the application. It also manages File Control Blocks, the File Allocation Table (FAT), and the file cluster cache.
2. **Flash Driver** implements the block cache, logical to physical address translation, block reclaim, wear leveling, data protection, garbage collection, and error correction.
3. **Flash Hardware Interface** implements the hardware-dependent routines to provide support for a particular NAND flash device, including: device initialization, device query, page read/write, and block erase.



### Virtual File System (VFS)

The Virtual File System (VFS) assumes a linear address space for the flash storage media. It also assumes that the media can only be accessed by sector (e.g. 512 bytes). The structure of the VFS address space is shown below:



- **Reserved Area** is divided into 2 parts: An area used by the Flash Driver and a user area that could store a boot sector or other data.
- **File System Signature Area** stores a string to identify the flash file system. It is located at the beginning of the next flash block following the reserved area.

<sup>1</sup> smxFFS is not for use with SD/MMC, USB thumb drive, or CF flash devices. smxFFS supports these devices; see the smxFFS data sheet.

- **File Control Block (FCB) Area** acts like a directory. Each FCB represents one file and contains the following information: status, file name, file extension, first cluster index, and file length.
- **File Allocation Table (FAT) Area** contains the FAT nodes. Each node represents one data cluster. 16-bit and 32-bit FAT nodes are supported. FAT size can be minimized by increasing cluster size.
- **File Data Area** stores the file data. It is aligned on a flash block boundary.

The first cluster index in a file's FCB is the index of the file's first FAT node. If it contains END\_CLUSTER, then the file has no data. Each FAT node, by its location in the FAT, corresponds to a data cluster in the linear address space of the data area. Each FAT node contains a pointer to the next FAT node of the file. The END\_CLUSTER flag marks the last node (and cluster) of the file.

The FAT node size is 16 bits by default, so *smxFFS* can support up to (65534 \* SectorsPerCluster \* BytesPerSector) bytes of data (Note that two values are reserved for END\_CLUSTER and BAD\_CLUSTER).

For example, 16-bit FAT nodes, 8 sectors per cluster, and 512-byte sectors can support media up to 256 MB, in size. Media size includes the total size of all NAND chips in the array. Larger media can be supported by increasing the number of sectors per cluster or by increasing the FAT node size to 32-bits, in the configuration file.

When the file system is *mounted*, the FCB and FAT areas are cached in RAM to improve performance. For each open file, the cluster that is currently being read or written is cached in RAM. A cached sector is written to the Flash Driver if it has been altered and another sector of the file needs to be accessed, or if `fclose()` is called for the file. New sectors are obtained from the Flash Driver for reading or writing.

## Porting Layer

Two files, **ffsport.h** and **ffsport.c**, contain definitions, macros, and functions to port *smxFFS* to any OS, compiler, or processor. Currently, they support SMX<sup>®</sup> and Windows. (Under Windows, you can run an emulator that uses the hard disk for testing FFS.) These files implement semaphore protection of the VFS API (see below) from reentrancy problems in multitasking environments.

## Virtual File System API

### Basic Calls

<code>v_ffs_fm mount ()</code>	Mount the NAND Flash File System.
<code>v_ffs_funmount ()</code>	Unmount the NAND Flash File System.
<code>v_ffs_fopen (filename, mode)</code>	Open one file for read/write access.
<code>v_ffs_fclose (filehandle)</code>	Close an open file.
<code>v_ffs_fread (buf, size, items, filehandle)</code>	Read data from an open file.
<code>v_ffs_fwrite (buf, size, items, filehandle)</code>	Write data to an open file.
<code>v_ffs_fseek (filehandle, offset, whence)</code>	Move the file pointer to the specified location.
<code>v_ffs_fdelete (filename)</code>	Delete a file.
<code>v_ffs_findfile (filename)</code>	Test if a file exists.
<code>v_ffs_filelength (filename)</code>	Return the length of a file, in bytes.
<code>v_ffs_fgetversion ()</code>	Return the file system version number.

### Extended Calls

<code>v_ffs_rewind (filehandle)</code>	Move the file pointer to the beginning of the file.
<code>v_ffs_fflush (filehandle)</code>	Flush all data associated with the file handle to the storage media.
<code>v_ffs_ftell (filehandle)</code>	Determine the current file pointer position.
<code>v_ffs_fsetend (filehandle)</code>	Truncate a file at the current file pointer.
<code>v_ffs_feof (filehandle)</code>	Test for end-of-file on a file.
<code>v_ffs_rename (oldname, newname)</code>	Rename a file.

<b>v_ffs_findfirst</b> (filespec, fileinfo)	Provide information about the first instance of a file whose name matches the name specified by the <i>filespec</i> argument.
<b>v_ffs_findnext</b> (id, fileinfo)	Find the next file, if any, whose name matches the <i>filespec</i> argument in a previous call to <i>v_ffs_findfirst()</i> , and return information about it in the <i>fileinfo</i> structure.
<b>v_ffs_fgetfilenum</b> (filespec)	Determine the total number of files, if any, whose names match the <i>filespec</i> argument.
<b>v_ffs_flushall</b> ()	Flush all data to the storage media.
<b>v_ffs_freespace</b> ()	Determine the percentage of free space on the storage media.
<b>v_ffs_freesize</b> ()	Determine the number of free bytes on the storage media.
<b>v_ffs_devicemounted</b> ()	Determine the percentage of free space on the storage media.
<b>v_ffs_formaterror</b> ()	Determine if the file system structure has an error.

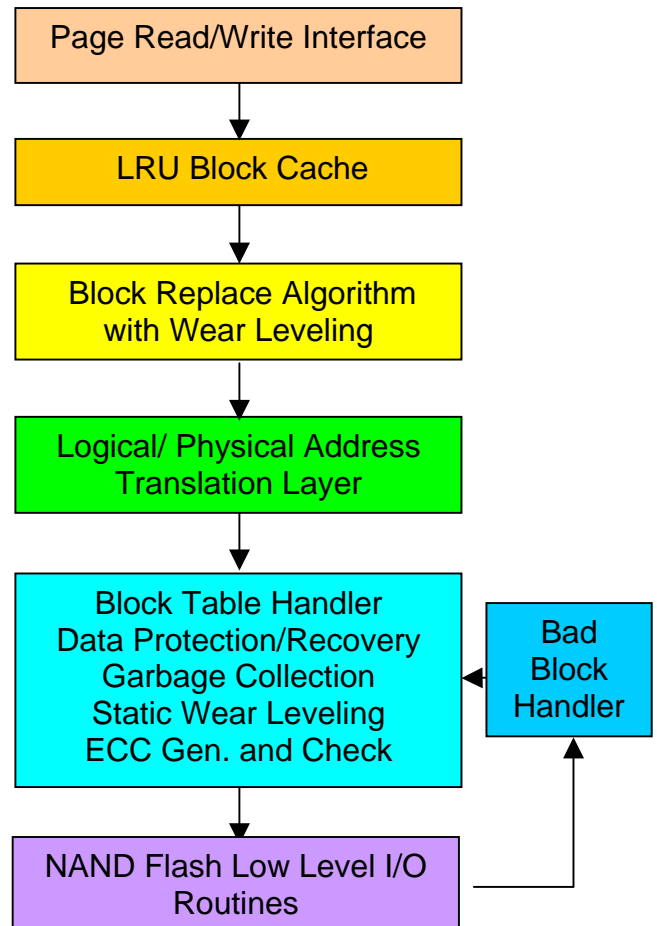
## Flash Driver

*The Flash Driver makes the flash memory appear to the file system as a linear array of read/write memory, like RAM. It supports both smxFFS and smxFS.*

smxFD provides two interface functions to the File System: *Page\_Read()* and *Page\_Write()*. A page of data is defined by the NAND flash chip and is 512 bytes or 2048 bytes. Page operations are performed on the Block Cache in RAM. A *flash block* is defined by the NAND flash chip as the minimum unit of erasable memory. It is usually a power of two times the page size. (Typical block sizes are 8KB for 4 and 8 MB devices, 16KB for 16 to 128 MB devices, and 128KB for larger devices.)

**LRU Block Cache.** The LRU (Least Recently Used) block cache contains cache blocks. The number of cache blocks is user-specified, as is their size, which may be as small as one page or as large as a flash block. When it becomes necessary to access a page, that is not cached and the cache is full, the least-recently-used cache block is either flushed to flash, if it has been changed, or discarded, if not. Then, the new cache block, containing the page, is read into the cache from flash. Smaller cache blocks reduce RAM requirements; larger or more cached blocks increase performance. Hence, cache size may be adjusted to achieve optimum RAM vs. performance for a specific application.

The structure of the flash driver is shown below:



- **Block Replace Algorithm.** When a cached block is flushed to flash, an empty flash block is found and the cached block is written to it. The algorithm finds a spare block that has not been used recently, in order to provide wear leveling. If multiple cache blocks correspond to the same flash block, all are written to the new flash block. Non-cached pages of the block are copied from the old block, in flash.
- **Static Wear Leveling.** Some files on the flash disk are likely to be permanent or rarely changed. To ensure wear leveling over the entire flash, these *static* files must be periodically relocated to other flash blocks. This is done during *garbage collection* (see below.) Since moving these files frequently could hurt system performance, they are only moved when the difference between the most-worn and their wear counters exceeds the `WEAR_LEVELING_GATE` configuration setting. Moving all static data, at once, could hurt system performance, so another setting specifies the maximum number of flash blocks to move at a time. Hence, several iterations may be required to move all of the static data.
- **Logical / Physical Address Translation** is done through the Block Table. Each location in the Block Table corresponds to a block in the logical address space (i.e. the address space used by the file system). Each entry contains a 14- or 30-bit physical flash memory index and a 2-bit status field. The index multiplied by the block size is the block's physical address in flash memory. The status field indicates if the block is: Valid, Discarded, Spare, or Bad. When a flash block is updated from the Block Cache, the Block Table entry is updated to point to the new flash block. Each Block Table entry is 16-bit by default but can be extended to 32-bit to support a large NAND flash array.
- **Block Table Handler.** The Block Table is normally stored in the Data Area of the flash and moves through that area just like any other data, so it does not wear out one area of the flash. When the Data Area fills up, the Block Table is written to the first part of the Reserved Area. Since it is usually smaller than a flash block, it can be re-written several times to successive pages in a flash block before it must be moved to a new flash block.
- **Data Protection and Recovery.** Before flushing the Block Table, its image in flash is marked as being In-Progress. When a new Block Table image is successfully saved in flash, it is marked Valid and the old Block Table image is marked as Discarded (in the first node of the Block Table). Hence, if power is lost there will always be at least a Valid or In-Progress Block Table to start from. (If both are present, the In-Progress Block Table is marked Invalid, and the Valid Block Table is used.) All cached data blocks that have not been flushed will be lost, if power is lost, but the file system will remain intact. The frequency of cache flushes is user-controlled either by closing a file or explicitly flushing the cache. (This is similar to the operation of other file systems.)
- **Garbage Collection.** When moving data to new blocks of flash, old blocks are marked as being discarded. *Garbage collection* is the process of erasing discarded blocks so that they can be used again. The user should call this function when the system is idle, after closing a file, or when a data operation is finished. Otherwise, it will be run automatically, during write, if an erased block is needed, but this hurts write performance.
- **Bad Block Handler.** If the flash hardware interface returns an error for a write or erase function, `smxFD` will retry a few times. If all retries fail, `smxFD` marks the block Bad in the Block Table and uses another free block.
- **Error Correction.** An ECC algorithm is used that can detect and correct any single-bit error in 256 bytes. If enabled, this is run automatically before the data is returned. Multi-bit errors are reported, if detected. (Not all are detectable.) The ECC's are stored in the spare area of each page. Software ECC greatly reduces read/write performance and should be disabled if hardware ECC is available.

## Flash Hardware Interface

*smxFFS* has been tested for Samsung, Toshiba, SanDisk, and Fujitsu NAND flash of 4, 8, 16, and 32 MB sizes and for STMicro NAND flash sizes of 64, 128, and 256 MB. The flash hardware API functions (see below) must be implemented for your NAND flash device, if it is not one of these. There should be no problem supporting any standard 8x or 16x NAND flash chip or array of chips.

### Flash Device Requirements

- NAND flash only
- At least 8-byte spare area in each page, for flash with 256-byte page size to store the 4-byte logical page address and ECC code. Larger page sizes need more space for ECC codes. The general formula is:  $(4 + 4 * \text{page\_size}/256)$  bytes of spare area.
- Ability to read/write the spare area of each page independently of the rest of the page
- Does not use interrupt to check R/B signal. Uses polling to check it, instead.
- Must support partial programming to a page at least 3 times.
- The maximum flash media size is the lesser of  $(2^{30} * \text{blocksize})$  or  $((\text{blocksize}/8) * \text{blocksize})$ . For example:

<u>Block Size</u>	<u>Max Size</u>
16 KB	32 MB
128 KB	2 GB
512 KB	32 GB
1 MB	128 GB

## Flash Hardware API

The file name and extension sizes can be user-specified. Default is 8.3.

<b>asm_Flash_Reset</b> ()	Reset the flash hardware. (Normally issues the 0xFF command to the chip.)
<b>asm_Flash_Init</b> ()	Initialize the interface hardware between the controller and NAND flash chip.
<b>asm_Read_Device_ID</b> ()	Read the device ID so the flash driver can load the hardware information into the DeviceInfo structure.
<b>asm_Write_Page</b> (wp, pa, ds)	Write data to the NAND flash. The flash driver ensures that the whole block has been erased before re-writing it.
<b>asm_Read_Page</b> (rp, pa, ds)	Read data from the NAND flash.
<b>asm_Write_Page_Spare</b> (wp, pa, ds)	Write data to the NAND flash spare area. The flash driver ensures that the whole block has been erased before re-writing it.
<b>asm_Read_Page_Spare</b> (rp, pa, ds)	Read data from the NAND flash spare area.
<b>asm_Erase_Block</b> (block_addr)	Erase the block.

ds = data size, pa = page address, rp = read pointer, wp = write pointer

## Size and Performance

### Code Size

Code size will vary widely depending upon the CPU, the compiler, and the optimization level. Below are two examples. The Mini Library excludes infrequently used VFile API calls. FHI = Flash Hardware Interface

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

### Data Size (RAM Requirement)

#### Flash Driver

RAM usage strongly depends upon the number of cache blocks (CACHE\_BLOCK\_NUMBER) and size of each (PAGES\_PER\_CACHE\_BLOCK), as shown by the examples, below:

#### 16 MB Flash

CACHE_BLOCK_NUMBER	PAGES_PER_CACHE_BLOCK	RAM
1	1	5152
1	16	20992
1	32	37888
2	32	54784

#### 64 MB Flash

CACHE_BLOCK_NUMBER	PAGES_PER_CACHE_BLOCK	RAM
1	1	17440
1	16	33280
1	32	50176
2	32	67072

### Virtual File System

#### Example 1:

MAX_FILES	BYTES_PER_CLUSTER	RAM
100	4096	36652
100	1024	51886
1024	512	116971

## Performance

### NAND Flash and Test Specifications

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 processor, 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 assumed.

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 initially empty (so there are no erase operations)

### Read Tests

Theoretical Minimum Time:

For each page (512 bytes valid data):  $220 \text{ ns} * 528 + 50 \text{ us} = 167 \text{ us}$

For each block (512 \* 32 bytes valid data):  $167 \text{ us} * 32 = 5.344 \text{ ms}$

For the whole test file  $5.344 \text{ ms} * (4096/16) = 1.37 \text{ s}$  (2.92 MB/s)

1.37 seconds (2.92 MB/s) is the hardware limitation. It is the shortest possible time to read 4MB of data from this flash memory over this processor bus.

Read Test #1: Without ECC Checking

Actual Test Result: 2.0 s

Overhead:  $2.0 - 1.37 = 0.63 \text{ s}$  (46%)

### Write Tests

Theoretical Minimum Time:

For each page (512 valid data):  $220 \text{ ns} * 528 + 300 \text{ us} = 416 \text{ us}$

For each block (512\*32 valid data):  $416 \text{ us} * 32 = 13.3 \text{ ms}$

For the whole test file  $13.3 \text{ ms} * (4096/16) = 3.4 \text{ s}$  (1.17 MB/s)

3.4 seconds (1.17 MB/s) is the hardware limitation. It is the shortest possible time to write 4MB of data to this flash memory over this processor bus.

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

Actual Test Result: 5 s

Overhead:  $5 - 3.4 = 1.6 \text{ s}$  (47%)

## PERFORMANCE vs. RAM

CACHE_BLOCK_NUMBER	PAGES_PER_CACHE_BLOCK	R/W KB/SEC (without ECC)	R/W KB/SEC (with ECC)
1	1	2035/7	N/A
1	16	2040/364	170/69
1	32	2037/532	171/93
2	32	2048/800	175/95

From the above table notice that increasing the number of cache blocks does not greatly improve read performance but reducing the cache block size does reduce read performance greatly. Increasing the number of cache blocks helps write performance. Using software to implement ECC greatly decreases performance. Because of this, we recommend disabling ECC and enabling the Read Back Check feature, instead.