

smxFFS

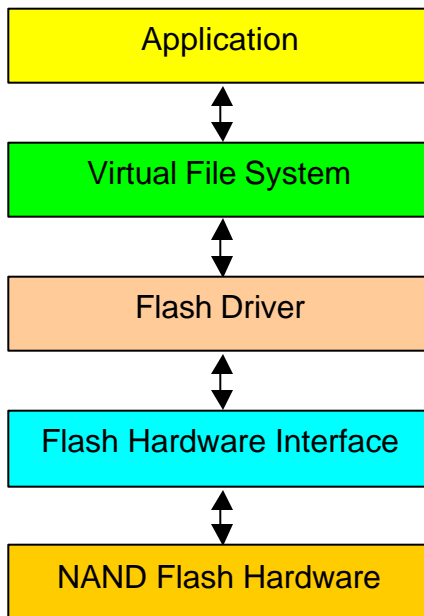
Flash File System

smxFFS is a flash file system for use with board-resident arrays of NAND flash memory.¹ 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.

Product Description

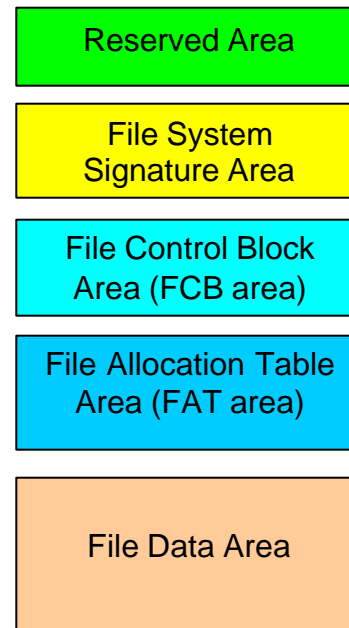
smxFFS has three layers:

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 Access Table, and the file sector cache.
2. **Flash Driver** supports 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, block erase.



Virtual File System (VFS)

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



- **Reserved Area** is divided into 2 parts: Block Table images (see the Flash Driver section) and space for a boot sector or for other purposes.
- **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.

¹ smxFFS is not for use with PCMCIA ATA flash cards or M-Systems DiskOnChip®. smxFile supports these devices; see the smxFile brochure.

- **File Control Block (FCB) Area** acts like a directory. It contains the FCB's. Each FCB represents one file and contains the following information: status, file name, file extension, first sector index, and file length.
- **File Allocation Table (FAT) Area** contains the FAT nodes. Each node represents one data sector.
- **File Data Area** stores the file data. It is aligned on a flash block boundary.

The first sector index in a file's FCB is the index of the file's first FAT node. If it contains END_SECTOR, then the file has no data. Each FAT node, by its location in the FAT, corresponds to a data sector 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_SECTOR flag marks the last node (and sector) of the file.

The FAT node size is 16 bits by default, so *smxFFS* can support up to 65534*512 bytes of data (two values are reserved for END_SECTOR and BAD_SECTOR). Hence, it supports media up to 32 MB in size. Larger media can be supported by increasing the FAT node size to 32-bit, in the configuration file.

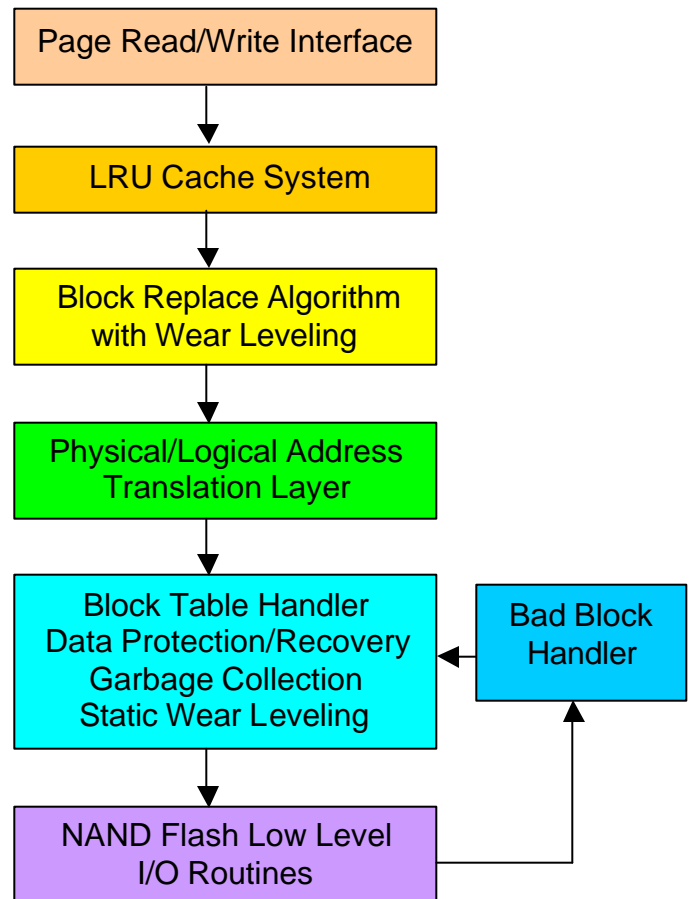
When the file system is mounted, the FCB and FAT areas are cached in RAM to improve performance. For each open file, the sector which is currently being read or written, is also cached in RAM. A cached sector is written to the Flash Driver, if it has been altered, and if another sector of the file needs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

Flash Driver (FD)

The Flash Driver converts linear addresses to NAND flash memory addresses. It provides only two interface functions to the VFS: PAGE READ and PAGE WRITE. A page is the same size as a sector. Page operations are performed on the Block Cache in RAM. A block is defined by the flash device type. It is the minimum unit of erasable

memory. (Typical block sizes are 8KB for 4 and 8 MB devices and 16KB for 16 to 128 MB devices.)

The structure of the flash driver is shown below:



- **LRU Cache System.** To improve performance, several data blocks are cached in RAM. (The number is user specified.) When it becomes necessary to access a block, that is not cached and the cache is full, the least-recently-used cached block is either flushed to flash, if it has been changed, or discarded, if not. Then, the new block is read into the cache from flash.
- **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 tries to find a spare block that has not been used recently in order to provide wear leveling for new blocks.
- **Wear Leveling of Static Blocks.** 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 relocated to other flash blocks. This is done during garbage collection. Since moving these frequently would hurt performance of the system, they are only moved when the difference between the most worn and least worn counter exceeds the configuration setting WEAR_LEVELING_GATE. Also, since there may be a lot of static data on your flash disk, another setting specifies the maximum number of blocks to move at a time.

- **Physical/Logical Address Translation Layer** is based upon the Block Table. Each location in the Block Table corresponds to a block in virtual address space (i.e. the space used by the VFS). Each entry contains a 14-bit flash memory index and a 2-bit status field. The index times 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 cached data block is written to flash, its Block Table entry is updated to point to the new flash block.
- **Block Table Handler.** The Flash File System reserves an area, at the beginning of flash memory, which is large enough for several Block Table images. (For example, an 8MB flash device has 1024 blocks, so the Block Table size is $1024 * 2 = 2048$ bytes = 4 pages.) Each time the Block Table is flushed to flash, it is written into an empty area.
- **Data Protection and Recovery.** Before modifying the Block Table, its image in flash is marked as being In-Progress. When a new Block Table image is successfully saved in flash, the old Block Table image is marked as Discarded. Hence, if power is lost there will always be one Valid or In-Progress Block Table to start from. All data blocks flushed since the last Block Table flush will be lost. The frequency of Block Table flushes is user-controlled. (This is similar to the operation of other file systems.)
- **Garbage Collection.** When writing new blocks to flash, old blocks are only marked as being discarded. *Garbage collection* is the process of

erasing discarded blocks so that they can be used again. It is the user's responsibility to decide when to call this function. It is best called when the system is idle, after closing a file, or when a data operation is finished.

- **Bad Block Handler.** If the flash hardware interface returns an error for a write or erase function, FD will retry a few times. If all retries fail, FD marks the block Bad and uses another free block.
- **Error Correction.** An ECC algorithm is used that can detect any bad bits in each page. A single-bit error in a half-page (256 bytes), is automatically corrected in the data returned. Otherwise an error is reported.

Flash Hardware Interface

smxFFS has been tested for Samsung, Toshiba, and Fujitsu NAND flash of 4 and 8 MB sizes. The flash hardware API functions (see page 4) may require some modification to support other NAND flash types.

Other flash types, such as NOR, are not supported. The hardware design must permit read/write to the spare area of each page. (Normally this requires controlling the SE pin of the NAND flash chip.) FD uses the spare area to store some additional page information. It is assumed that the flash device does not use an interrupt to check the R/B signal. FD uses polling to check it.

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 and Windows. (Under Windows, we have an emulator, which uses the hard disk for demoing *smxFFS*.) These files implement semaphore protection of the VFS API (see page 4) from reentrancy problems in multitasking environments.

Flash Hardware API

asm_Flash_Reset ()	Reset the flash hardware. Normally issues the 0xFF command to the chip
asm_Flash_Init ()	Initialize the interface hardware between the controller and NAND flash chip.
asm_Read_Device_ID ()	Read the device ID so the flash driver can load the hardware information into the DeviceInfo structure.
asm_Write_Page (write_data, page_add, data_size)	Write data to the NAND flash. The flash driver ensures that the whole block is already erased before writing to it.
asm_Read_Page (read_data, page_add, data_size)	Read data from the NAND flash.
asm_Write_Page_Spare (write_data, page_add, data_size)	Write data to the NAND flash spare area. The flash driver ensures that the whole block is already erased before writing to it.
asm_Read_Page_Spare (read_data, page_add, data_size)	Read data from the NAND flash spare area.
asm_Erase_Block (block_add)	Erase the whole block.

Virtual File System API

Basic Calls

v_ffs_fm ount ()	Mount the NAND Flash File System.
v_ffs_fun mount ()	Unmount the NAND Flash File System.
v_ffs_f open (filename, mode)	Open one file for read/write access.
v_ffs_f close (filehandle)	Close an open file.
v_ffs_f read (buf, size, items, filehandle)	Read data from an open file.
v_ffs_f write (buf, size, items, filehandle)	Write data to an open file.
v_ffs_f seek (filehandle, offset, whence)	Move the file pointer to the specified location.
v_ffs_f delete (filename)	Delete a file.
v_ffs_f indfile (filename)	Test if a file exists.
v_ffs_f ilelength (filename)	Return the length of a file, in bytes.
v_ffs_f getversion ()	Return the file system version number.

Extended Calls

v_ffs_r ewind (filehandle)	Move the file pointer to the beginning of the file.
v_ffs_f flush (filehandle)	Flush all data associated with the file handle to the storage media.
v_ffs_f tell (filehandle)	Determine the current file pointer position.
v_ffs_f setend (filehandle)	Truncate a file at the current file pointer.
v_ffs_f eof (filehandle)	Test for end-of-file on a file.
v_ffs_r ename (oldname, newname)	Rename a file.
v_ffs_f indfirst (filespec, fileinfo)	Provide information about the first instance of a file whose name matches the name specified by the <i>filespec</i> argument.
v_ffs_f indnext (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.
v_ffs_f getfilenum (filespec)	Determine the total number of files, if any, whose names match the <i>filespec</i> argument.
v_ffs_f lushall ()	Flush all data to the storage media.
v_ffs_f reespace ()	Determine the percentage of free space on the storage media.
v_ffs_f reesize ()	Determine the number of free bytes on the storage media.
v_ffs_d evicemounted ()	Determine the percentage of free space on the storage media.
v_ffs_f ormaterror ()	Determine if the file system structure has an error.