smxNAND™

Flash Driver

The smxNAND flash driver makes NAND flash memory appear to a file system like a disk drive. It supports single-level cell (SLC) and multi-level cell (MLC) NAND flash.

General

smxNAND works with both the smxFFS Flash File System and the smxFS FAT file system. It is a two-level driver. The high-level driver presents a sector-oriented interface to a file system. The low-level driver presents a hardware-independent interface, or abstraction layer, to the high-level driver. The standard low-level driver interfaces directly to 8-bit and 16-bit NAND flash chips. It can be modified to interface to NAND flash chip arrays to increase capacity and/or performance (using parallel NAND chips). If higher speed operation is required, such as for video streaming, it can be replaced with a NAND flash controller and controller driver.

smxNAND is designed for use in embedded systems and has a small code footprint and small to moderate RAM footprint. Performance can be increased by increasing RAM. The algorithm favors performance over minimizing RAM usage. It is good for applications requiring many files with good performance and at least 50 KB of RAM is available. For applications where minimal RAM usage is more important than performance, the smxNOR driver may be more appropriate. For simple data logging operations, with very small flash and RAM space, smxFLog may be the best choice.

Definitions

- **Block**: Minimum erasable unit of a flash chip. (Note: some flash chip vendors use the term *sector*, instead.)

Features

- SLC and MLC support.
- High performance – 8.5 MB/s read, 3.6 MB/s write for ARM9 processor.
- Adjustable RAM usage to tradeoff performance vs. size.
- 12KB code footprint.
- Power fail safe.
- Static and dynamic wear leveling.
- Garbage collection.
- Bad block handler.
- Error detection and correction.
- Works with smxFFS and smxFS.
- Easily portable to other file systems.
- Two-level structure facilitates supporting different NAND devices.
- Hardware Interface Layer (HIL) interfaces directly to standard NAND flash chips.
- HIL may be modified for NAND flash arrays and NAND flash controllers.
- Can be restricted to a partition of flash memory
- Sector-oriented interface to file system.

- **ECC**: Error Correction Code. The ECC code used by smxNAND can detect and correct a single bit (SLC) or four bit (MLC) error per 256 bytes.
- **Page**: Minimum writeable unit of a NAND flash chip. A *small* page is 512 bytes and a *large* page is 2048 bytes. Large pages are used on flash chips of size greater than 128 MB.

- **Programming**: Means writing data to flash.
- **Sector**: Minimum unit of data for a file system. It is 512 bytes, by default.
- **Empty or Free Block**: The data is all 0xFF's. Can be used to store new data.
- **Current Sector**: The sector currently being read or written by the file system.
- **Spare Area**: Extra space per page in NAND chips that is used for control information and ECC. This space is not included in the page size.

NAND Flash Requirements

- Minimum of 12 bytes of *spare area* per 512 bytes of data area to store status information and ECC. The general formula is: $(6 + 3 * page\_size/256)$ bytes of spare area per page.
- Spare area and data area of each page can be read and written independently.
- Supports programming to the spare area of a page at least 3 times (for SLC).

Limitations

- Each block must be the same size. Small contiguous blocks may be combined to form full blocks that start on full block boundaries.
- The logical sector size (normally 512 bytes) must be defined at compile time.
- The maximum flash media size is the lesser of $(2_{30} * blocksize)$ or $((blocksize/8) * blocksize)$. For example:

Block Size (KB)	Max Flash Media Size
16	32 MB
128	2 GB
512	32 GB
1024	128 GB

- MLC requires at least 4-bit ECC. Hardware ECC is needed because it is too slow in software.

High-Level Driver

The high-level driver performs most of the functions of smxNAND and it makes a NAND flash memory look like a disk drive to a file system.

API
smxNAND provides the following API to a file system:

int FFS_Flash_Init (void)
Initializes the smxNAND driver and hardware. Gets NAND flash chip ID.

int FFS_Cache_Init (void)
Intializes the block cache (See Operation section, below). Retrieves NAND flash format information.

int FFS_Whole_Cache_Write_Back (void)
Flushes the data in the block cache to flash. Should be done periodically for power fail safety, but doing too often can result in excessive flash wear. Normally called when the file system needs to close a file or to do a flush operation

int FFS_Garbage_Collection (void)
Can be used to do garbage collection when the flash file system is idle. Doing so improves write performance.

int FFS_Is_BadBlock (bn)
Can be used to determine if a block is a bad block.

int FFS_Flash_Format (void)
Formats the flash memory. All information stored in flash memory will be lost. smxNAND calls this automatically if it can not find the correct format information.

void FFS_Cache_Release (void)
Can be used to release the RAM allocated for block cache when the flash is not being used, in order to free RAM for other usage.

int FFS_Page_Read (rp, pi)
Reads the page starting at pi into the page buffer starting at rp.

int FFS_Page_Write (wp, pi)
Writes the page in the page buffer starting at wp into the page at pi.

int   FFS_Sector_Read (rp, si)
Reads the 512 byte sector at si into the sector buffer starting at rp.

int   FFS_Sector_Write (wp, si)
Writes the 512 byte sector from the sector buffer starting at wp into the sector at si.

bn = block number, pi = page index, si = sector index, rp = read buffer pointer, wp = write buffer pointer.

A software layer in the file system maps the file system's standard driver API to the above API. See the file system data sheet or manual for more information. Read and write operations are performed on the Block Cache (see below) in RAM. Typical block sizes are 8KB for 4MB and 8MB devices, 16KB for 16 to 64MB devices, and 128KB for larger devices.

Operation

Block Cache. smxNAND implements a least-recently-used cache to hold full or partial flash blocks in RAM. These are called *cache blocks*. A cache block can be as small as 1 page or as large as a flash block (it must be a power of 2 ∗ page size and located on a like boundary). The number of cache blocks is user-specified, as is their size. When a cache miss occurs, the least-recently-used cache block is selected and its data, if changed, is flushed to flash (See Block Replace below). Then, the cache block containing the target page or sector is read into the cache from flash and the read or write operation is performed on it. Smaller cache blocks reduce RAM requirements; larger cache blocks and more cache blocks increase performance. Hence, block cache size can be adjusted to optimize RAM vs. performance. Other parts of smxNAND use a block as the physical data read/write unit, since the NAND flash can only be erased by block.

Block Replace. When a cache block is flushed to flash, a free (i.e. empty) flash block is found and the cache block is written to it. The algorithm finds a free block that has not been used recently, in order to provide *dynamic* wear leveling. If multiple cache blocks correspond to the same flash block, all are written to the new flash block. Also, non-cached pages are copied from the old block, in flash to the new block. Thus, the whole block is moved to a new location in flash memory, then the block table (see below) is updated.

Block Table. The block table does logical to physical address translation. 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 block index and a 2-bit status field. The block 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 bits, by default, but can be increased to 32 bits to support large NAND chips or arrays.

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. 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.

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 block 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. Several iterations may be required to move all static data.

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 reused. The user should call the garbage collection function when the file system is idle, after closing a file, or when a data operation is finished. Otherwise, garbage collection may occur during a write, if a free block is needed, which obviously hurts write performance and could cause data overruns.

Power Fail Safety. When a block of data is updated, it must be written to a new block in flash and the Block Table must be updated to reflect this. Before starting this operation, the current Block Table is marked as being in-progress. When the data and the Block Table have been successfully updated, the new Block Table is marked as valid and the old Block Table is marked as discarded. If power is lost at any point in this process, there will always be at least a valid or an 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. Any cache blocks that have not been flushed will be lost, if power is lost, but the file system will remain intact. Cache flushes can be invoked either by closing a file or by explicitly flushing the cache.

Bad Block Handler. If the flash hardware interface returns an error for a write or an erase operation, smxNAND retries a few times. If all retries fail, smxNAND marks the block Bad in the Block Table and uses a different free block for the operation

Error Correction. An ECC algorithm is provided that can detect and correct any single-bit error in 256 bytes. If enabled, the ECC is automatically generated for each 256 bytes of data before writing it to flash and tested before read data is returned. Single bit errors are corrected; 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. MLC requires at least 4-bit ECC. Hardware ECC is needed because it is too slow in software.

Hardware Interface Layer (HIL)

The low-level driver is referred to as the Hardware Interface Layer (HIL) because it creates a hardware-independent abstraction layer for the high-level driver.

API

void**asm_Flash_Reset (id)**
Resets the flash hardware. Normally issues the 0xFF command to the chip.

void**asm_Flash_Init (void)**
Initializes the interface hardware between the microprocessor and the NAND flash chip, such as GPIO and MMU.

uint16**asm_Read_Device_ID (id, pdi)**
Reads the device ID so the flash driver can load hardware information into the DeviceInfo structure.

uint16**asm_Write_Page (id, wp, pi, os, ds)**
Writes data to the NAND flash. The flash driver ensures that the whole block has been erased before writing to it.

uint16**asm_Read_Page (id, rp, pi, os, ds)**
Reads data from the NAND flash.

uint16**asm_Write_Page_Spare (id, wp, pi, os, ds)**
Writes data to the NAND page spare area. The flash driver ensures that the whole block has been erased before writing to it.

uint16**asm_Read_Page_Spare (id, rp, pi, os, ds)**
Reads data from the NAND page spare area.

uint16**asm_Erase_Block (bi)**
Erases one flash block.

id = chip id, bi = block index, ds = data size, os = offset, pdi = device info pointer, pi = page index, rp = read pointer, wp = write pointer

4

Supported Flash

smxNAND has been tested for Samsung, Toshiba, SanDisk, and Fujitsu NAND flash of 4, 8, 16, 32, 64, 128 and 256 MB sizes and for Samsung, STMicro, Micron, Numonyx NAND flash sizes of 64, 128, 256 MB, and 1GB. There should be no problem supporting any standard x8 or x16 NAND flash chip or array of chips with the standard low-level driver. For other NAND chips and NAND controllers the low-level driver will need to be modified or rewritten.

MLC (Multi-Level Cell) support was added in v1.90. However, please read our white paper *MLC vs. SLC NAND Flash in Embedded Systems* www.smxrtos.com/articles/mlcslc.htm before you design your system.

Size and Performance

Code Size

CPU and Compiler	Size KB
Coldfire CodeWarrior 6.3	9
ARM IAR 5.11	12

Data Size (RAM Requirement)

RAM usage is determined by the following formula:
RAM = BlockTableSize + WearCtrSize + CacheSize + TmpBufSize
where:
BlockTableSize = 2 or 4 * number of blocks
WearCtrSize = 2 * number of blocks
CacheSize = cache blocks * cache block size (= pages each * page size)
TmpBufSize = cache block size

RAM usage can be quite large for MLC flash, depending upon the performance needed and the total flash size. The following are some examples:

Flash Size	SLC	MLC
256 MB	13 KB	270 KB
512 MB	21 KB	290 KB
1 GB	38 KB	320 KB
2 GB	70 KB	360 KB

Performance

Measurements were made on a Samsung K9F2G08U using an AT91SAM9263 processor with a clock speed of 240 MHz, a processor bus speed of 120 MHz, a NAND chip speed of 30 MHz, SDRAM. The 9263 has hardware ECC but we did not develop a driver for it, so the ECC was calculated in software.

No DMA, No ECC	Read KB/sec	Write KB/sec
smxNAND raw data	4266	1969
smxFFS with smxNAND	3961	1785
smxFS with smxNAND	3373	1689

DMA, No ECC	Read KB/sec	Write KB/sec
smxNAND raw data	8533	3657
smxFFS with smxNAND	7787	3038
smxFS with smxNAND	5953	2852

DMA, 1-Bit Sfw ECC	Read KB/sec	Write KB/sec
smxNAND raw data	1828	1505
smxFFS with smxNAND	1801	883
smxFS with smxNAND	1719	847

NAND Flash Performance vs. Cache Size

For SLC flash, we are using Second Block Table technology so using a page cache will not decrease performance too much, compared with a whole block cache. But for MLC flash, we cannot use that feature so the cache size is a bigger factor of the write performance. The following 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.

AT91SAM9263 (DMA)

	1	2	4	8	16	32	64
Read without ECC	8533	8533	8533	8533	8533	8533	8533
Write without ECC	62	124	246	492	948	1828	3657
Read with software ECC	1828	1828	1828	1828	1828	1828	1828
Write with software ECC	23	49	91	115	316	673	1505

Performance (KB/s)

Cache Size (Page#)