Micro Digital

smxFLog™ User's Guide

Flash Logger

Version 1.50
February 22, 2024

by Yingbo Hu and David Moore

Table of Contents

Micro Digital Associates, Inc.
(714) 437-7333
support@smxrtos.com
www.smxrtos.com

1. Overview

smxFLog has the simple purpose of logging data efficiently and reliably in flash, using a minimum of RAM. Logging data is a very common operation in embedded systems, and warrants a good solution. It is a sequential operation consisting of appending data to a file. This is not efficient in FAT file systems when writing to flash media. The problem is that writing less than a full cluster of data each time requires the partially-filled last cluster to be moved around in flash each time, and it also requires modifying FAT and directory sectors, which also have to be moved. This hurts peformance and wears the flash. It also means garbage collection is frequently required, which is a lengthy operation and can stall further logging temporarily. smxFLog can append new records without moving data, and it has no tables to move either.

smxFLog is also power-fail-safe because of its simplicity. The DOS/Windows FAT file system is inherently not power-fail-safe, unless it is extended with journaling or some other mechanism, which might be incompatible. Non-standard file systems can be designed to be power-fail-safe, as smxFFS was, but they are more complex and have larger memory footprints. This makes them unsuitable for many low-end SoCs with small on-chip SRAM and no ability to connect to external memory.

File systems are very useful, though, because they allow storing multiple files and directories. Moreover, the DOS/Windows FAT file system allows media to be shared with other computers. Even non-removable media in an embedded target can be read from the file system on a PC if the target is running a USB device stack with mass storage driver, such as smxUSBD. In this case, a disk stored in resident flash will look like a disk to a PC connected by USB cable. File systems are useful because they allow sharing data either using removable media or via a data link such as USB or FTP.

Thus, it is beneficial to use smxFLog and smxFS in the same system. Data can be logged reliably by smxFLog and periodically offloaded in chunks to the file system, which is efficient. API functions are provided, and code examples shown in section 5. Application Examples, to make this easy. They can coexist in the same flash (in separate partitions), and in this case, they will share the same low-level flash driver, so there is only one driver to port and no duplicated code.

1.1 Features

- Works with NAND, NOR, or serial NOR flash.
- Can be used with any size flash memory.
- Supports multi-chip flash arrays.
- Uses the same low-level drivers as smxNAND and smxNOR.
- Supports multiple logs by partitioning the flash.
- Wear leveling is inherent.
- Skips bad blocks.
- Efficient and fast.
- Designed for reliable use.
- ECC mode for NAND flash.
- Read back and verify mode.
- Simple, safe API.
- Power fail safe.
- Small:
    - ROM: 5 KB with ECC, 3 KB without ECC.
    - RAM: 288 bytes with ECC: 32 bytes without ECC, for each log.
- Can share flash with smxFS, smxFFS, boot code, and application code

1.2 Limitations

In order to make smxFLog simple, efficient, and reliable, the following restrictions are imposed:

- All flash records in a log must have the same size, specified at compile time. Size restrictions are different for NAND and NOR flash. **See g_LogConfig[] in section 2.3.2 flcfg.c**.
- New records can only be appended following the old records, and existing records cannot be modified.
- Supported NAND flash types are those that: support block erase (changing all cells to 0xFF), support partial page write at least 3 times, and have a spare area. There is no overhead in the data area. MLC flash is not supported because it has no partial page programming.
- For NOR flash there is overhead in the data area to store the status and ECC bytes since there are no spare areas.

1.3 Overhead

1/4 status byte plus 3 bytes ECC (optional) are stored for each flash record. For NAND flash there is no overhead because these are stored in the spare area of each page. For NOR flash there are no spare areas, so this information must be stored in the data area. Overhead is (1+3*((record_size+255)/256))/record_size if ECC is enabled. The minimum tested flash record size is 32 bytes, for which overhead is 12.5%. For a record size of 256, overhead is 1.56%. See the discussion for g_LogConfig[] in section 2.3.2 flcfg.c.

1.4 Record Size

For NAND and NOR, record size cannot exceed (erase) block size. If you must log more data than the block size, you must use multiple records.

NAND flash: Flash records must be 512 bytes or a power of 2 multiple of 512 bytes (1, 2, 4, 8, ..). This is because of partial programming limitations. (See section 2.4.2 Status and ECC Bytes.) If the record were 256 bytes, there would be 2 per page, requiring partial programming 6 times (3 times for each status byte (ECC is written with first status byte write)), but many flash chips support only 3 times. If the flash chip supports at least 6 times, this can be reduced to 256. Note that some flash chips with page size 2048 actually combine four 512-byte pages together to generate the 2048-byte page so these flash chips can still support a 512-byte record size. Record size does not need to be 2048 bytes.

NOR flash: Flash records can be any size that is a power of 2 because there is no partial programming issue, but the overhead of status and ECC bytes becomes significant as block size decreases. See section 1.3 Overhead. It is important to support smaller record sizes for NOR flash since most systems won't have much. The minimum tested record size is 32 bytes.

For serial NOR (SPI) flash, record size must be the page size of the flash, usually 256 bytes, for the low-level drivers we supply (since they are written to also support our file systems, smxFFS and smxFS which read file system sectors). Also, this is required for flash chips that require writes to start at page boundaries, unless buffering and copying are done. Some allow writing at any address within a page, which should make it easier to use a smaller record size. In any case, when creating a new driver, we recommend supporting record == page size first. Also for smaller record sizes, it's likely the InfoRead/Write() functions will need special handling, as discussed below.

Tips

1. Our low-level flash drivers (nandio*.c and norio*.c) are shared with smxFFS, smxFS, and smxFLog.

2. SectorSize in the low-level drivers means a file system sector size, typically 512 bytes for our file systems. For smxFLog it means a record.

3. For serial NOR, choose a chip that supports writing to any location in a page, if you want to use a record size < page size. See discussion above.

4. nor_IO_InfoRead() and nor_IO_InfoWrite() write the metadata (status + ECC bytes) at the start of each flash erase block for all records in that block. If record size is small or if enabling ECC, this could mean it takes more than 1 page of space. In that case, these routines must have extra logic to read/write the correct page, such as this:

```
xxxx_READ(BlockIndex*PagesPerSector + Offset/PageSize, Buf, PageSize, 0);
memcpy(Buf + Offset%PageSize, pInfo, BufSize);
xxxx_WRITE(BlockIndex*PagesPerSector + Offset/PageSize, Buf, PageSize);
```

1.5 Multiple Logs

Multiple logs are supported by defining fixed-size partitions in g_LogConfig[] (see section 2.3.2 flcfg.c for details). Record size can be set independently for each log. Logs are identified by integer IDs, which could be named constants in the application, for readability.

Typical applications sample data at a constant interval with a constant sample size, so the rate of growth of each log is known in advance, and partition sizes can be set proportionately. Supporting fixed-size logs rather than dynamic logs makes the code simpler and smaller and requires less RAM. smxFLog was designed to run with very little RAM (only 32 bytes with ECC disabled), unlike flash file systems. See Appendix B. Size and Performance for code and data sizes.

2. Using smxFLog

2.1 Getting Started

You must erase the flash first if it contains any pre-loaded image or data. After you implement your low level NAND or NOR flash driver, use the code provided in ffstest.c (for NAND), fdtest.c (for NOR), or flltest.c (both) to verify your driver first. Please see section 3.2 nandio.c in the smxNAND User's Guide or section 5.3 Verify the Driver in the smxNOR User's Guide for details.

2.2 Basic Terms

Block	Minimum erasable unit of the flash chip. Some NOR flash chips use the term *sector*, instead.
Page	Maximum read/write unit of data of a NAND flash chip. It is 512 or 2048 bytes.
Flash Record	A unit of flash data. All in a log must be the same size and a power of 2.
Data Record	A unit of application data. May not be the same size as a flash record, in which case, several might occupy a single flash record, or one may span several flash records. smxFLog is not aware of data record size or structure.

Pointer/Ptr Record index not address. For example, 1, 2, 3, ….

Note: It is cumbersome to specify "flash record" and "data record" in all places. When not specified, context should make it clear. Most often, record means flash record, since that is what smxFLog is concerned with, not data records.

2.3 Configuration Settings

2.3.1 flcfg.h

flcfg.h contains flash logger configuration constants that allow selecting features and tuning performance, code size, and RAM usage.

Note: Other settings are in the low-level driver configuration file, flashcnf.h (NAND) or fdcfg.h (NOR).

SFL_NAND
Set to "1" to use NAND flash to log data.

SFL_NOR
Set to "1" to use NOR flash to log data.

SFL_USE_ECC
Set to "1" to enable software ECC code to check and correct data consistency. Default setting is "0". If you have hardware ECC support for your processor, set SFL_USE_ECC to 0 and implement it in the low level driver. Ensure the hardware generated ECC code won't overwrite the 4-byte status in the spare area.

SFL_READBACK_VERIFY
Set to "1" to enable read back verification to check data consistency. It will allocate an additional flash record-sized read back buffer in RAM. Default setting is "0". Normally this is only used for testing.

Operation if enabled: After writing the data, smxFLog reads it back into another buffer and compares this buffer to the original data buffer. If they are not the same, it marks the current record bad and goes to the next empty record to write it again. The process repeats until it is written successfully, or it will stop if SFL_RECYCLE_FLASH is 0 and there are no more empty records. The retries are transparent to the application, and the bad block is not marked because we cannot guarantee we can still update the status byte to the desired value. The next time we encounter it, we will skip it again.

SFL_RECYCLE_FLASH
Set to "1" to enable auto reclaim of the oldest block when the flash is full. Default setting is "1". If 0, smxFLog will stop logging when the flash is full.

Operation if enabled: Immediately after writing a record, smxFLog checks to see if the flash is full. If so, it reclaims the oldest block(s), so empty records are ready for new data.

CAUTION: Erasing blocks is a slow and variable-length operation. See section 2.8 Erasing Blocks.

SFL_SAFETY_CHECKS

Set to "1" to enable extra safety checking code to check internal data structures and parameters passed to the APIs. The safety checks are not guaranteed to catch all problems, such as a particular memory corruption pattern or corrupted record data buffer pointer.

SFL_BADREC_BITMAP

If 1, a bitmap is used for the bad record array for sfl_Read() rather than a byte array. This allows reducing the size of the buffer that must be passed to hold bad record information, but it is less efficient. If the application needs to read many records at once, you may want to enable this.

SFL_MAX_RECORD_SIZE

Maximum record size of all logs. It is used to allocate memory if **SFL_READBACK_VERIFY** is enabled or to calculate ECC bytes if **SFL_USE_ECC** is enabled. It can be ignored it if both **SFL_READBACK_VERIFY** and **SFL_USE_ECC** are disabled.

Note: Record size for each log is set in the configuration tables in flcfg.c. By default, they use this value, but it can be specified as any value <= SFL_MAX_RECORD_SIZE in those tables, and it can be different for each log.

SFL_MAX_LOG_NUM

Maximum number of logs smxFLog should support. It is also necessary to set the log configuration table in flcfg.c. See section 2.3.2 flcfg.c for details.

2.3.2 flcfg.c

g_LogConfig[]

This is the multiple log configuration table used to tell smxFLog the properties of each log. Configure this table as desired. This is an array of structure SFL_LOG_CONFIG, which is defined as:

```
typedef struct
{
   u32 iStartBlock;
   u32 iBlockNum;
   uint iRecordSize;
   uint iRecycleBlockNum;
} SFL_LOG_CONFIG;
```

iStartBlock

Start block index of this log. When using multiple logs or other file systems on the same flash chip, ensure this log's area does not overlap other partitions.

iBlockNum

Total number of blocks reserved by this log. Use ((u32)-1) to use the remaining blocks to the end of flash. When using multiple logs or other file systems on the same flash chip, ensure this log's area does not overlap other partitions.

iRecordSize

Flash record size. Must be a power of 2, subject to the characteristics of the flash, as discussed in section 1.4 Record Size. All flash records in the same log must be the same size, so for small data records, we recommend buffering data, then writing multiple data records into each flash record.

iRecycleBlockNum
How many block(s) smxFlog needs to erase each time, when the SFL_RECYCLE_FLASH is set to 1. Normally it should be set to 1. If you want to group multiple records into one big virtual record, you may need to change this setting. For example, if your application needs to use a 256KB record size but your flash's block size is only 128KB then you cannot set iRecordSize to 256KB but your application can read/write two 128KB record to get a virtual 256KB application record. To avoid the problem of having a partial record due to smxFLog recycling one block, you need to set this value to 2 so smxFlog will erase two blocks each time.

2.4 Implementation Details

This manual does not document the theory of operation of smxFLog. However, we explain here a few key concepts that will aid in understanding how to use the API.

2.4.1 Record Pointers and Marks

smxFLog maintains pointers to the oldest record, next record to read, and the next record to write. These are initialized by scanning the whole flash at startup. The oldest record pointer is advanced as old flash blocks are reclaimed. The read pointer is advanced after each sfl_Read() to point at the next record to read. The write pointer is advanced after each sfl_Write() to point at the next empty slot to write to. The flash is treated as a circular queue and the pointers go round and round, unless the application never erases old blocks and SFL_RECYCLE_FLASH is 0.

sfl_ReadPtrMark() allows marking the record in flash that the read pointer currently points to so that if a power fail occurs, the read pointer will start there after restart. This API will erase all the old records that were already read, in the blocks preceeding the one the mark is written to.

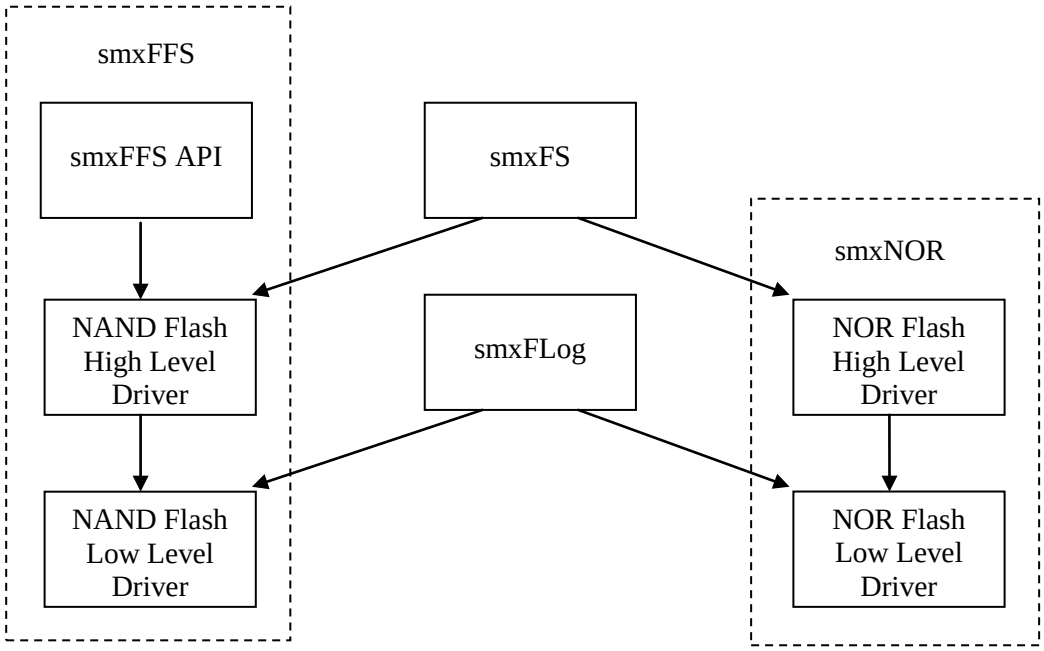
2.4.2 Status and ECC Bytes

The status byte is used to indicate if a write is in progress or completed, or to mark the record bad. It is also used to mark the current read pointer when blocks are erased or it is moved by sfl_ReadPtrMark(). The status byte is also used as an erase mark, so an erase operation can be resumed if power fails. The byte starts at 0xFF, which is the flash erased value. It is first changed to the value to mean write in progress. At that same time, the ECC is written. When the write is done, it is either changed to the write completed value or to the bad record value. In the first case, it might be changed later by clearing a bit to indicate it is the current read pointer. At most 3 writes are done to the combined status + ECC bytes, so that the 3-times rule for partial programming NAND flash is not violated. An erase mark is never written to the same page as the read mark, to avoid exceeding the partial programming limit.

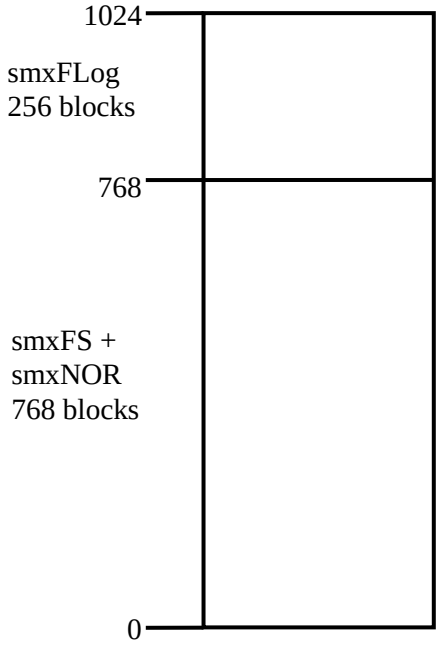
2.5 Partitioning the Flash

By default smxFLog uses the entire flash. However, it is easy to reserve blocks before it and after it in the flash, for use by other software, such as our FAT file system smxFS, as is discussed in the Overview of this manual. In order to understand how to configure this, the following diagrams are helpful.

The following diagram shows how the different SMX file systems relate to each other.



Notice that smxFLog is at the same level as the NAND and NOR high level drivers. The start and end block numbers in flashcnf.h (NAND) and fdcfg.h (NOR) are comparable to those in flcfg.h (smxFLog). The following diagram shows how smxFS + smxNOR and smxFLog can share NOR flash. This example shows a flash chip with 1024 flash blocks.



To achieve this, configure as follows:

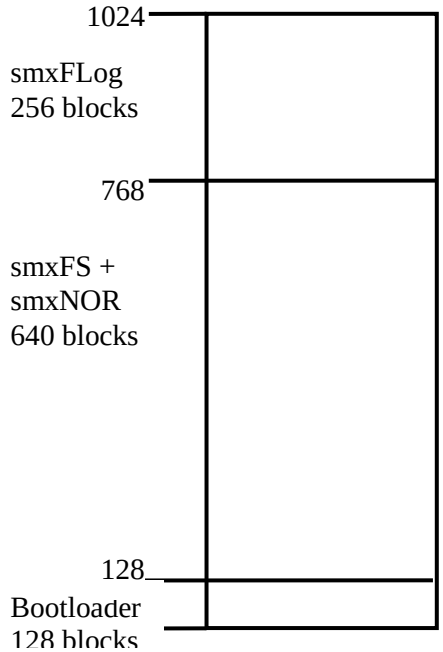
XFD**fdcfg.h** (smxNOR; NOR driver for smxFS)

```
#define NOR_START_BLOCK_INDEX   0
#define NOR_BLOCK_NUM           768
```

XFL**flcfg.c**

```
const SFL_LOG_CONFIG g_LogConfig[SFL_MAX_LOG_NUM] =
{
   {768, 256, 2048},
};
```

Note that additional blocks could be reserved at the start and end of flash for other purposes by changing NOR_START_BLOCK_INDEX to non-zero values and SFL_BLOCK_NUM to be smaller than 256. For example, if additional 128 blocks need to be reserved for boot loader then the configuration will be:



XFD**fdcfg.h**

```
#define NOR_START_BLOCK_INDEX   128
#define NOR_BLOCK_NUM           640
```

XFL**flcfg.c**

```
const SFL_LOG_CONFIG g_LogConfig[SFL_MAX_LOG_NUM] =
{
   {768, 256, 2048},
};
```

Partitioning the NAND flash is similar.

2.6 Mixed and Small Block Sizes

For NAND flash, all blocks are the same size, but it is common for NOR flash to have smaller block sizes at the start or end of the flash (i.e. the boot block). The total size of the small blocks is the same as a normal flash block (e.g. 64KB). To handle this, either exclude these small blocks from the partitions of the flash used by smxNOR and smxFLog, or treat them as a single block by adding special handling to nor_IO_SectorRead() and nor_IO_SectorWrite() in the low-level NOR driver.

SoCs with on-chip flash typically have small blocks, such as 4KB instead of the typical 64KB. The low-level NOR driver can support this.

2.7 Wear Leveling

Wear leveling is guaranteed because data is written to the flash sequentially.

2.8 Erasing Blocks

Erasing blocks is a slow operation, especially for NOR flash. It can take 0.1s to 1s or even more. The amount of time is variable and can increase for a particular block the more times it has been erased. Be careful about any assumptions about how long an erase or logging operation should take. smxFLog provides an option to recycle flash and two API functions that erase one or more flash blocks, but that doesn't mean they should be used. If the system must continue logging data without interruption, it may be necessary to choose a large enough flash chip (or array) to store a complete log without erasing. In the application, it may only be safe to wait until a sampling session is complete, and then offload the data, erase the whole flash, and start over.

The two API functions that erase blocks are sfl_Erase() and sfl_ReadPtrMark(). It might be good to use them only to erase one block at a time. For sfl_Erase(), pass SFL_ERASE_ONE_BLOCK. For sfl_ReadPtrMark(), call it every time a block worth of records has been read.

2.9 Power Fail Safety

Power fail safety is easily achieved because there are no data structures, such as mapping tables, FATs, or directories, to keep consistent with the data. A status byte for each flash record indicates whether a write operation is pending or completed. Partially written records are simply skipped, when reading.

2.10 Error Correction

After a NAND flash chip has been used for a long time, it may develop some bad bits. smxFLog implements a software ECC algorithm capable of detecting a 2-bit error and fixing a 1-bit error. The ECC code is 3 bytes per 256 bytes. smxFLog uses ECC only for NAND because NOR flash tends to be more reliable. Also, when it does fail, many bits tend to go bad, not just one or two.

The ECC is generated before the data is actually written to the 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 SFL_ERROR_BAD_REC is reported but the record data will still be returned.

Our ECC algorithm can only process 256 bytes, but a flash record may be larger or smaller. For a record of size 512 bytes or a multiple of 512 bytes, two ECC codes are generated for each 512 bytes of data—

one for the first 256 bytes and the other for the second 256 bytes. The ECC codes are stored in the spare area of each page for NAND flash or in the area smxFLog reserves in NOR flash. For NAND flash, if the record (page) size is a multiple of 512 bytes, several ECC codes may be created in the NAND spare area, so the flash chip must have spare areas larger than 16 bytes per page. The ECC codes require 3 bytes per 256 bytes, so ECC uses 3 * page_size/256 bytes, and 1 byte is required for the status byte. For example a 2048-byte page size requires 1 + 3 * 2048/256 == 25 bytes in each spare area. Flash chips with page size 2048, normally have a spare area of 64 bytes instead of the normal 16 bytes, which is plenty to store the ECC and status byte information. For NOR flash, this is not an issue because the status byte plus three ECC bytes is written after each data record or 256 data bytes, whichever is smaller. When a NOR flash record is less than 256 bytes, it is treated as if it were padded out to 256 bytes with 0's, and a 3-byte code is still generated.

If the record size is multiple of NAND flash page size, ECC of each page will be stored in the spare area of each page.

2.10.1 ECC Code Generation

We use a Hamming code to implement 1-bit ECC. It can detect a 2-bit error and correct a 1-bit error.

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

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

…….

253th byte	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	LP00	LP0 2	LP05
254th byte	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	LP01		
255th byte	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	LP00	LP0 3	
256th 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:

CP00 = bit7 XOR bit5 XOR bit3 XOR bit1 XOR CP00
CP01 = bit6 XOR bit4 XOR bit2 XOR bit0 XOR CP01
CP02 = bit7 XOR bit6 XOR bit3 XOR bit2 XOR CP02
CP03 = bit5 XOR bit4 XOR bit1 XOR bit0 XOR CP03
CP04 = bit7 XOR bit6 XOR bit5 XOR bit4 XOR CP04

CP05 = bit3 XOR bit2 XOR bit1 XOR bit0 XOR CP05

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 = 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.

2.11 Application Development

The main operations of smxFLog are: initialize flash, write records, read records, and (optionally) erase old blocks of records. The read pointer can be moved in limited ways, and a read mark can be set to revert to. smxFLog can be configured to write once through the flash or to recycle the flash and wrap around to the beginning when the end is reached. The API is documented in section 3. smxFLog API.

The read and write pointers are not written to flash due to partial programming limitations. Instead, at power-up they are determined by scanning the flash. Their positions are determined based on unused space (empty blocks). Unfortunately, on some flash chips, bad blocks cannot be distinguished from empty blocks, so the locations of these pointers is ambiguous. To make their locations certain, smxFLog writes a read mark at the start of the flash when it is initialized so it is guaranteed that there will be one, and when writing a new read mark (with sfl_ReadPtrMark()), all blocks before it are erased. Erasing flash can be slow, especially for NOR flash, so it may be best to erase one block at a time. See section 2.8 Erasing Blocks for more information. In a multitasking environment, a loop could be used that suspends the task briefly after erasing each block, to allow others to run.

smxFLog prevents loss of data, but if a power fail occurs, the read pointer may be restored to an earlier position, so that data being transcribed or transmitted may be repeated. As a consequence, the application should store a sequence number or timestamp in each record, so that it is possible for the recipient to determine where to continue processing the data.

3. smxFLog API

The smxFLog API is defined in smxflog.h, which contains the functions that are called by the application.

int **sfl_Init**(uint iFlag);

int **sfl_Release**(void);

int **sfl_Read**(uint iLogID, u8 *pRecord, uint iNum, uint *piNumRead, u8 *pBadRecArray, uint iNumBadRecArray);

int **sfl_ReadPtrMark**(uint iLogID);

int **sfl_ReadPtrRestore**(uint iLogID);

int **sfl_ReadPtrSkipTo**(uint iLogID, u32 dwNewReadPtr);

int **sfl_ReadPtrSkipToRel**(uint iLogID, u32 dwNum);

int **sfl_Write**(uint iLogID, u8 *pRecord, uint iDataSize);

int **sfl_Erase**(uint iLogID, uint iFlag);

u32 **sfl_GetMaxRecords** (uint iLogID);

u32 **sfl_GetMaxFreeRecords** (uint iLogID);

u32 **sfl_GetOldestReadPtr**(uint iLogID);

u32 **sfl_GetReadMarkPtr**(uint iLogID);

u32 **sfl_GetReadPtr**(uint iLogID);

u32 **sfl_GetWritePtr**(uint iLogID);

3.1 API Data Types

These are defined in **flport.h**.

3.2 API Reference

Each smxFLog API will return a result to the application. The application should check the return value carefully for any potential error case. SFL_ERROR_NONE is the return value for success.

int **sfl_Init** (uint iFlag)

Summary Initialize hardware and internal data structures.

Details This function should be called first before using smxFLog. It calls the NAND or NOR Flash Hardware IO Routine in the low-level driver to initialize the flash chip and retrieve the basic information about it such as the block size and total number of blocks. It then will perform the following:

Pars

iFlag	One of the following: **SFL_INIT_CHECK** Do normal power up and scan the whole flash to initialize the record pointers. Preserves existing records in the flash. This delays the power up procedure due to the scanning. **SFL_INIT_SKIP_CHECK** Do not scan the flash to initialize the record pointers. This assumes the application has erased the whole flash partition used by smxFLog. It is the fastest way to initialize smxFLog. **SFL_INIT_ERASE_ALL** Erase the whole smxFLog partition before using it. All records will be lost.

Returns

SFL_ERROR_NONE	Initialization succeeded.
SFL_ERROR_HARDWARE	Initialization failed because the low level hardware init function returned an error.
SFL_ERROR_NO_MUTEX	Initialization failed because smxFLog could not allocate mutex resource from the OS.
SFL_ERROR_INTERNAL	Internal data structure(s) are corrupted. smxFLog will stop logging. Please call sfl_Release() and sfl_Init() again to try to recover from it.

See Also sfl_Release()

Example

```
if(SFL_ERROR_NONE == sfl_Init(SFL_INIT_CHECK))
    printf("Flash Logger Initialized.");
```

int **sfl_Release** (void)

Summary Release smxFLog resources.

Details This function should be called when the application is done with smxFLog. It resets the internal record pointers set by sfl_Init() and calls the NAND or NOR Flash Hardware IO Routine to release the hardware resources.

Pars none

Returns

SFL_ERROR_NONE	Release succeeded.

See Also sfl_Init()

Example

```
if(SFL_ERROR_NONE == sfl_Release())
    printf("Flash Logger Released.");
```

int **sfl_Read**(uint iLogID, u8 *pRecord, uint iNum, uint *piNumRead, u8 *pBadRecArray, uint iNumBadRecArray)

Summary Read one or more flash records from the flash memory. Bad records are still returned but will be marked in the BadRecArray.

Details This function reads flash records from the flash memory starting at the record pointed to by the read pointer. The buffer allocated by the application must be large enough to hold the specified number of records. The read pointer (in RAM) is advanced to point to the next record to read.

If bad records are encountered, the data is still read into the destination buffer *pRecord as is, and pBadRecArray[] indicates which records are bad. For example, if the application needs to read 4 records but the second one is bad because the ECC check failed, then pBadRecArry[] = {0, 1, 0, 0}. To reduce the size of this array for large reads, SFL_BADREC_BITMAP can be enabled so it is a bitmap rather than a byte array. For more information, see section 2.3 Configuration Settings.

Pars

iLogID	Log ID. Should be 0 to SFL_MAX_LOG_NUM–1.
pRecord	The destination buffer to hold the records read from flash.
iNum	The requested number of records to read.
piNumRead	The actual number of records read. This number may be smaller than the number requested.
pBadRecArray	Array to indicate bad records. If NULL is passed for this parameter, smxFLog will not return information about bad records, and it will ignore parameter iNumBadRecArray.
iNumBadRecArray	The size of the bad records array.

Returns

SFL_ERROR_NONE	Records were read from the flash, but the actual number read may be smaller than the number requested, so the application needs to check the value of *piNumRead.
SFL_ERROR_BAD_REC	One or more of the records read was bad. The application needs to check pBadRecArray to find out which records are bad.
SFL_ERROR_FLASH_EMPTY	All records have been read.
SFL_ERROR_INTERNAL	Internal data structure(s) are corrupted. smxFLog will stop logging. Please call sfl_Release() and sfl_Init() again to try to recover from it.
SFL_ERROR_INVAL_PARAM	A parameter passed to this API is invalid. For example, pRecord is a NULL pointer or iNumBadRecArray is smaller than the iNum, the number of records requested.

See Also sfl_Init(), sfl_ReadPtrMark(), sfl_ReadPtrRestore(), sfl_Write()

Example

```
u8 RecordBuf[10*512];
uint iNumRead
u8 BadRecArray[10];
int result;
uint I;
result = sfl_Read(0, RecordBuf, 10, &iNumRead, BadRecArray, 10);
if(result == SFL_ERROR_NONE)
{
    printf("There are no bad records. Actual number of records read is %d.", iNumRead);
}
else if(result == SFL_ERROR_BAD_REC)
{
    printf("Returned records contains bad records, total record number is %d", iNumRead);
    for(i = 0; i < iNumRead; i++)
    {
        If(BadRecArray[i] == 1)
        {
            printf("Record %d is bad. You may need to ignore it.", i);
        }
    }
}
```

int **sfl_ReadPtrMark**(uint iLogID)

Summary Mark the read pointer in the flash chip.

Details This function marks the flash record to which the current read pointer (stored in RAM) points, by clearing one of its status bits, so after system restart, it is not necessary to re-read the old records that were already read from the flash. The old blocks of records before it will be erased, starting at the block pointed to by the oldest record pointer. It erases up to the nearest physical block boundary below the new read pointer.

CAUTION: Erasing blocks is a slow and variable-length operation. See section 2.8 Erasing Blocks. To ensure only one block is erased, this function should be called every time a block worth of records is read.

Pars

iLogID	Log ID. Should be 0 to SFL_MAX_LOG_NUM–1.

Returns

SFL_ERROR_NONE	Mark succeeded.
SFL_ERROR_FLASH_EMPTY	All records have been read.
SFL_ERROR_FLASH_ERASE	Mark succeeded but erasing the old blocks failed.
SFL_ERROR_FLASH_WRITE	Mark failed because the flash write operation failed.
SFL_ERROR_INTERNAL	Internal data structure(s) are corrupted. smxFLog will stop logging. Please call sfl_Release() and sfl_Init() again to try to recover from it.

See Also sfl_Read()

Example

```
while (data_to_send)
{
    sfl_Read(0, &sbuf, num_records, &num_records, NULL, 0);
    SendData(&sbuf);
    if (data_sent_ok)
        sfl_ReadPtrMark(0, TRUE);
    else
        sfl_ReadPtrRestore(0);
}
```

int **sfl_ReadPtrRestore** (uint iLogID)

Summary Restore the read pointer to the last marked position.

Details This function restores the read pointer to the last position stored in the flash so if anything goes wrong, the application can restart reading from the last known read pointer. Useful if data is lost during transmission.

Pars

iLogID	Log ID. Should be 0 to SFL_MAX_LOG_NUM–1.

Returns

SFL_ERROR_NONE	Restore succeeded.
SFL_ERROR_INTERNAL	Internal data structure(s) are corrupted. smxFLog will stop logging. Please call sfl_Release() and sfl_Init() again to try to recover from it.

See Also sfl_Read(), sfl_ReadPtrMark()

Example See the example for sfl_ReadPtrMark().

int **sfl_ReadPtrSkipTo** (uint iLogID, u32 dwNewReadPtr)

Summary Move the read pointer forward to skip unwanted records.

Details This function allows moving the read pointer forward to a certain position to skip unwanted records. It is necessary to know the new read pointer position before calling this function. Typically this is a write pointer that was saved at some meaningful location, by sfl_GetWritePtr(). It is not possible to move the read pointer backward to an older record to retrieve the skipped records using this function. (Use sfl_ReadPointerRestore().)

Pars

iLogID	Log ID. Should be 0 to SFL_MAX_LOG_NUM–1.
dwNewReadPtr	The new read pointer location.

Returns

SFL_ERROR_NONE	Skip succeeded.
SFL_ERROR_INTERNAL	Internal data structure(s) are corrupted. smxFLog will stop logging. Please call sfl_Release() and sfl_Init() again to try to recover from it.

SFL_ERROR_INVAL_PARAM The new read pointer is not valid. For example, an attempt was made to move it backward to an older record.

See Also sfl_GetReadPtr(), sfl_GetWritePtr(), sfl_Read(), sfl_ReadPtrMark()

Example

```
dwSavedPtr = sfl_GetWritePtr(iLogID);
//…write some records with sfl_Write()
//…
sfl_ReadPtrSkipTo(iLogID, dwSavedPtr);
```

int **sfl_ReadPtrSkipToRel** (uint iLogID, u32 dwNum)

Summary Move the read pointer forward to skip unwanted records, skipping the specified number of records.

Details This function allows moving the read pointer forward to skip specific number of unwanted records. It is not possible to move the read pointer backward to an older record to retrieve the skipped records using this function. (Use sfl_ReadPointerRestore().)

Pars

iLogID	Log ID. Should be 0 to SFL_MAX_LOG_NUM–1.
dwNum	Number of records to skip.

Returns

SFL_ERROR_NONE	Skip succeeded.
SFL_ERROR_INTERNAL	Internal data structure(s) are corrupted. smxFLog will stop logging. Please call sfl_Release() and sfl_Init() again to try to recover from it.
SFL_ERROR_INVAL_PARAM	The new read pointer is not valid. For example, an attempt was made to move it backward to an older record.

See Also sfl_GetReadPtr(), sfl_GetWritePtr(), sfl_Read(), sfl_ReadPtrMark()

Example

```
sfl_ReadPtrSkipToRel(iLogID, 10);    /* skip 10 records from current read pointer */
```

int **sfl_Write**(uint iLogID, u8 *pRecord, uint iDataSize)

Summary Append a new flash record to the flash log.

Details This function adds a new flash record to the flash log following the previous one. After writing the record, the write pointer is advanced to the next empty record. If SFL_READBACK_VERIFY is 1 and the verify fails, the record is marked bad and it is written at the next location and checked again. If the end of flash has been reached and recycle mode has not been specified, no further writes will be allowed. If recycle mode has been specified and the last free block is about to be used, then the next block is erased.

Pars

iLogID	Log ID. Should be 0 to SFL_MAX_LOG_NUM–1.
pRecord	Pointer to the record data buffer to write.
iDataSize	Actual data record size. If the actual size is smaller than iRecordSize in the configuration table, performance can be improved by passing the actual size to this

API, and then padding is not written. If the data record size is very small, it is recommended to pack multiple small records into one big flash record (i.e. iRecordSize) to improve the performance and flash usage efficiency. For example, if the data record size is 120 and iRecordSize is 512, then pack 4 data records into one 512 byte buffer and call sfs_Write(pRecord, 480) to write them at once.

Returns

SFL_ERROR_NONE	Flash record data appended.
SFL_ERROR_FLASH_FULL	Flash is full and automatic reclaim of the oldest block is disabled.
SFL_ERROR_INTERNAL	Internal data structure(s) are corrupted. smxFLog will stop logging. Please call sfl_Release() and sfl_Init() again to try to recover from it.
SFL_ERROR_INVAL_PARAM	A parameter passed to this API is invalid. For example, pRecord is a NULL pointer or iDataSize is larger than iRecordSize.

See Also sfl_Read(), sfl_Erase()

Example

```
u8 RecordBuf[512];
sfl_Write(0, RecordBuf, 512);
```

int **sfl_Erase**(uint iLogID, uint iFlag)

Summary Erase one or more blocks of the oldest records

Details This function erases one or more blocks of the oldest records, according to the flag specified.

CAUTION: Erasing blocks is a slow and variable-length operation. See section 2.8 Erasing Blocks. It may be desirable to erase one block at a time (pass SFL_ERASE_ONE_BLOCK for iFlag).

Pars

iLogID	Log ID. Should be 0 to SFL_MAX_LOG_NUM–1.
iFlag	One of the following flags to indicate how many of the oldest blocks should be erased: **SFL_ERASE_ONE_BLOCK** Erase only the oldest block of records each time. Erasing one block at a time from the idle task/loop will minimize the impact on performance if logging is occurring simultaneously. Moves the marked read pointer up to the first record in the next block if it was pointing at the block that was deleted. **SFL_ERASE_OLD_BLOCKS** Erase all the old blocks of records up to the one the read pointer is in. At least one block will be kept. Moves the marked read pointer up to the first record in the next block if it was pointing at a block that was deleted. **SFL_ERASE_ALL_BLOCKS** Erase all data records. After this call the flash partition is just like a new one; all the blocks within that partition are erased. smxFLog will write new records starting at the beginning of that flash partition. Resets all record pointers.

Returns

SFL_ERROR_NONE	Erase succeeded.
SFL_ERROR_FLASH_ERASE	Erase failed.
SFL_ERROR_INTERNAL	Internal data structure(s) are corrupted. smxFLog will stop logging. Please call sfl_Release() and sfl_Init() again to try to recover from it.
SFL_ERROR_INVAL_PARAM	The parameter iFlag is not valid.

See Also sfl_Init()

Example

```
sfl_Erase(0, SFL_ERASE_ONE_BLOCK);
```

u32 **sfl_GetMaxFreeRecords** (uint iLogID)

Summary Get the maximum number of free records in this smxFLog partition.

Details Call this function to get the maximum number of free records in this smxFLog partition. If there is any bad blocks, the real number of records that can store data may be less than this value.

Pars

iLogID	Log ID. Should be 0 to SFL_MAX_LOG_NUM–1.

Returns The maximum number of free records.

See Also sfl_GetMaxRecords()

Example

u32 **sfl_GetMaxRecords** (uint iLogID)

Summary Get the maximum total number of records this smxFLog partition can store.

Details Call this function to get the maximum number of records that can be stored in this smxFLog partition. If there are any bad blocks there the actual number of records you can store may be less than this value.

Pars

iLogID	Log ID. Should be 0 to SFL_MAX_LOG_NUM–1.

Returns The maximum total number of records.

See Also sfl_GetReadPtr(), sfl_GetWritePtr()

Example See the example for sfl_ReadPtrSkipTo().

u32 **sfl_GetOldestReadPtr** (uint iLogID)

Summary Get the oldest read pointer of this smxFLog partition.

Details Call this function to get the oldest read pointer of this smxFLog partition.

Pars iLogID Log ID. Should be 0 to SFL_MAX_LOG_NUM–1.

Returns The oldest read pointer.

See Also sfl_GetReadPtr()

Example See the example for sfl_ReadPtr().

u32 **sfl_GetReadMarkPtr** (uint iLogID)

Summary Get the current read mark pointer of this smxFLog partition.

Details Call this function to get the current read mark pointer of this smxFLog partition.

Pars iLogID Log ID. Should be 0 to SFL_MAX_LOG_NUM–1.

Returns The current read mark pointer.

See Also sfl_ReadPtrMark ()

Example See the example for sfl_ReadPtrMark().

u32 **sfl_GetReadPtr** (uint iLogID)

Summary Get the current read pointer of this smxFLog partition.

Details Call this function to get the current read pointer of this smxFLog partition.

Pars iLogID Log ID. Should be 0 to SFL_MAX_LOG_NUM–1.

Returns The current read pointer.

See Also sfl_GetWritePtr()

Example See the example for sfl_ReadPtrSkipTo().

u32 **sfl_GetWritePtr** (uint iLogID)

Summary Get the current write pointer of this smxFLog partition.

Details Call this function to get the current write pointer of this smxFLog partition. The only way to change the write pointer is by calling sfl_Write()..

Pars iLogID Log ID. Should be 0 to SFL_MAX_LOG_NUM–1.

Returns The current write pointer

See Also sfl_GetReadPtr()

Example See the example for sfl_ReadPtrSkipTo().

4. Low-Level Flash Drivers

smxFLog uses the same low-level drivers as smxNAND and smxNOR.Low-level drivers normally need some customization to the specific hardware on which they are to run. Below is the summary of those low level functions

4.1 NAND flash

void **nand_IO_Flash_Reset** (uint iChipID);

Summary Reset the flash chip.

Details Reset the flash hardware. Normally issues the 0xFF command to the chip. Refer to the hardware spec for details.

Pars iChipID The device ID to use.

Returns none

See Also asm_Flash_Init()

Example

```
nand_IO_Flash_Init();
nand_IO_Flash_Reset(0);
```

void **nand_IO_Flash_Init** (void);

Summary Reset the flash chip.

Details Initialize the interface hardware between the processor and the NAND flash chip, such as GPIO and MMU. This is the first function that must be called for the NAND flash driver.

Pars none

Returns none

See Also nand_IO_Flash_Release()

Example

```
nand_IO_Flash_Init();
```

void **nand_IO_Read_Device_ID (**uint iChipID, DEVICE_INFO *pDeviceInfo);

Summary Get the flash chips information.

Details 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.

Pars

iChipID	The chip index to use. Currently only pass 0.
pDeviceInfo	The NAND flash device information structure to fill.

```
typedef struct
{
   uint32 wDeviceMaker;        //0xec:samsung, 0x98:toshiba
   uint32 wDeviceType;         //1:1M, 2:2M, 4:4M, 8:8M, 16:16M,  ..bytes
   BLOCKNODE wBlockNum;        //blocks in a disk
   PAGENUMTYPE wPagesPerBlock;  //pages in a block
   PAGESIZETYPE wPageSize;     //page_size in bytes
   PAGESIZETYPE wPageDataSize;  //data_size in bytes
   PAGESIZETYPE wPageSpareSize; //spare_size in bytes
   BLOCKNODE wDataBlockNum;     //data block number which can be
            used
   BLOCKSIZETYPE wBlockSize;    //block size in bytes
   BLOCKSIZETYPE wBlockDataSize;//block data size in bytes
}DEVICE_INFO;
```

Returns none

See Also nand_IO_Flash_Init(),nand_IO_Flash_Reset()

Example

```
nand_IO_Flash_Init();
nand_IO_Flash_Reset(0);
nand_IO_Flash_Read_Device_ID(0, &DevInfo);
```

uint16 **nand_IO_Write_Page** (uint iChipID, byte * write_data, uint32 page_index, uint offset, uint32 data_size);

Summary Write data to the main area of one flash page.

Details 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 to write to.

Pars

iChipID	The chip index 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.

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

uint16 **nand_IO_Read_Page** (uint iChipID, byte * read_data, uint32 page_index, uint offset, uint32 data_size);

Summary Read data from the main area of the flash.

Details Read some data from the NAND flash.

Pars		
	iChipID	The chip index 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.

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

uint16 **nand_IO_Write_Page_Spare** (uint iChipID, byte * write_data, uint32 page_index, uint offset, uint32 data_size);

Summary Write data to the spare area of one flash page.

Details 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.

Pars		
	iChipID	The chip index 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 using 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.

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

uint16 **nand_IO_Read_Page_Spare** (uint iChipID, byte * read_data, uint32 page_index, uint offset, uint32 data_size);

Summary Read data from the spare area of the flash.

Details Read some data from the NAND flash spare area.

Pars

iChipID	The chip index 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 using 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.

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

uint16 **nand_IO_Erase_Block** (uint32 block_index);

Summary Erase one flash block

Details Erase one flash block. All the data of that block will be reset to 0xFF.

Pars

block_index	Block index. May be necessary to generate the block address by multiplying it by block size.

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

4.2 NOR flash

int **nor_IO_FlashInit** (uint iID, NOR_DEVINFO * pDevInfo)

Summary Initialize the flash chip.

Details This function initializes the NOR flash chip and gets the basic information of it.
It is necessary to set the member variables, dwTotalBlockNum and dwBlockSize of structure NOR_DEVINFO, so the driver can initialize its own internal data structures. Also set dwSectorSize and set NOR_FORCE_SECTOR_SIZE to 1 if you need to use a certain sector size, or else it will be calculated.

Pars

nID	The device ID to use.
pDevInfo	The pointer to the device information structure to fill.

Returns 1 Initialization succeeded.
0 Initialization failed.

See Also nor_IO_FlashRelease()

Example

```
NOR_DEVINFO DevInfo
If(nor_IO_FlashInit (0, &DevInfo))
{
    printf("Total Block number is %d\n", DevInfo.dwTotalBlockNum);
    printf("Block size is %d\n", DevInfo.dwBlockSize);
    printf("Sector size is %d\n", DevInfo.dwSectorSize);
}
```

int **nor_IO_FlashRelease** (uint iID)

Summary Release the flash chip.

Details This function releases the NOR flash chip.

Pars nID The device ID to use.

Returns 1

See Also nor_IO_FlashInit()

Example

```
nor_IO_FlashRelease (0);
```

int **nor_IO_SectorRead** (uint iID, u8 * pRAMAddr, u32 wSectorIndex, uint wSectorSize)

Summary Read one sector of data from the flash chip. (This is a file system sector or log record of specified size, not a flash erase block.)

Details This function read one sector of data from the flash chip. The sector is normally 512 bytes by default but may be another size so please make sure the memory buffer is big enough.
If the flash chip cannot read 512 bytes each time, it may be necessary to map this function call to multiple flash commands. For example, some serial flash can only read up to 256 bytes in one command. Our sample code already shows how to handle this case. For smxFLog a sector is one record.

Pars

nID	The device ID to use.
pRAMAddr	The pointer to the memory buffer to hold the data. The buffer should be at least one sector in size.
wSectorIndex	The physical sector index of the flash chip. It may be necessary to map this index to the flash address.
wSectorSize	The size of the buffer.

Returns 1 The read succeeded.

See Also nor_IO_SectorWrite()

Example

```
u8 pData[512];
memset(pData, i, NOR_DEFAULT_SECTOR_SIZE);
nor_IO_SectorWrite(0, pData, j + i * iPagePerSec, NOR_DEFAULT_SECTOR_SIZE);
memset(pData, 0xFF, NOR_DEFAULT_SECTOR_SIZE);
nor_IO_SectorRead(0, pData, j + i * iPagePerSec, NOR_DEFAULT_SECTOR_SIZE);
for(k = 0; k < NOR_DEFAULT_SECTOR_SIZE; k++)
{
    If(pData[k] != (u8)i)
    {
        printf("IO Sector Write/Read mismatch at %d, %d, %d \r\n", i, j, k);
    }
}
```

int **nor_IO_SectorWrite** (uint iID, u8 * pRAMAddr, u32 wSectorIndex, uint wSectorSize)

Summary Write one sector data to the flash chip. (This is a file system sector or log record of specified size, not a flash erase block.)

Details This function writes one sector of data to the flash chip. A sector is normally 512 bytes by default. If the flash chip cannot write 512 bytes each time, it may be necessary to map this function call to multiple flash commands. For example, some serial flash can only write up to 256 bytes in one command. For smxFLog a sector is one record.

Pars

nID	The device ID to use.
pRAMAddr	The pointer to the memory buffer holding the data to write. It should be at least one sector in size.
wSectorIndex	The physical sector index of the flash chip. It may be necessary to map this index to the flash address.
wSectorSize	The size of the buffer.

Returns 1 The write succeeded.

See Also nor_IO_SectorRead()

Example

```
u8 pData[512];
memset(pData, i, NOR_DEFAULT_SECTOR_SIZE);
nor_IO_SectorWrite(0, pData, j + i * iPagePerSec, NOR_DEFAULT_SECTOR_SIZE);
memset(pData, 0xFF, NOR_DEFAULT_SECTOR_SIZE);
nor_IO_SectorRead(0, pData, j + i * iPagePerSec, NOR_DEFAULT_SECTOR_SIZE);
for(k = 0; k < NOR_DEFAULT_SECTOR_SIZE; k++)
{
    If(pData[k] != (u8)i)
    {
        printf("IO Sector Write/Read mismatch at %d, %d, %d \r\n", i, j, k);
    }
}
```

int **nor_IO_InfoRead** (uint iID, void * pInfo, uint wBufSize, u32 wBlockIndex, uint wOffset)

Summary Read a few bytes from the flash chip.

Details This function reads a few bytes of information from the flash chip. Information will only be stored in the first few pages of each block.

Pars

nID	The device ID to use.
pInfo	The pointer to the memory buffer pointer holding the data.
wBufSize	The size of the buffer pointed to by pInfo.
wBlockIndex	The physical block index of the flash chip in which the information is stored.
wOffset	The byte offset of the information from the beginning of this block.

Returns 1 The read succeeded.

See Also nor_IO_InfoWrite()

Example

```
nor_IO_InfoWrite(0, &dwInfo, sizeof(u32), i, j*sizeof(u32));
nor_IO_InfoRead(0, &dwInfoTemp, sizeof(u32), i, j*sizeof(u32));
if(dwInfo != dwInfoTemp)
{
    printf("IO Info Write/Read mismatch 1 at %d, %d \r\n", i, j);
}
```

int **nor_IO_InfoWrite** (uint iID, void * pInfo, uint wBufSize, u32 wBlockIndex, uint wOffset)

Summary Write a few bytes to the flash chip.

Details This function writes a few bytes of information to the flash chip. Information will only be stored in the first few pages of each block.

Pars

nID	The device ID to use.
pInfo	The pointer to the memory buffer pointer to hold the data.
wBufSize	The size of the buffer pointed to by pInfo.
wBlockIndex	The physical block index of the flash chip in which the information is stored.
wOffset	The byte offset of the information from the beginning of this block.

Returns 1 The write succeeded.

See Also nor_IO_InfoRead()

Example

```
nor_IO_InfoWrite(0, &dwInfo, sizeof(u32), i, j*sizeof(u32));
nor_IO_InfoRead(0, &dwInfoTemp, sizeof(u32), i, j*sizeof(u32));
if(dwInfo != dwInfoTemp)
{
    printf("IO Info Write/Read mismatch 1 at %d, %d \r\n", i, j);
}
```

int **nor_IO_BlockErase** (uint iID, u32 wBlockIndex)

Summary Erase a block of the flash chip.

Details This function erases the whole block at the specified index.

Pars

nID	The device ID to use.
wBlockIndex	The physical block index.

Returns 1 Erase succeeded.

See Also nor_IO_SectorRead(), nor_IO_SectorWrite(), nor_IO_InfoRead(), nor_IO_InfoWrite()

Example

```
for(i = 0; i < DevInfo.dwTotalBlockNum; i++)
{
    nor_IO_BlockErase(0, i);
}
```

5. Application Examples

5.1 Offload Log Data to smxFS

The data records can be written to a file in a file system by smxFS, so the user can offload the log data to removable media such as a thumb drive, or it can be retrieved from the file system later via data link such as USB, FTP, etc.

```
#include "smxflog.h"
#include "smxfs.h"

void SendRecordsToFile(void)
{
    FILEHANDLE fp;
    uint iNumRead;
    u8 RecordBuf[4*512];

    fp = sfs_fopen("A:\\DataRecord.bin", "wb");
    if(fp)
    {
        do
        {
            sfl_Read(0, RecordBuf, 4, &iNumRead, NULL, 0);      /* read 4 records each time */
            sfs_fwrite(RecordBuf, iNumRead, 512, fp);
        } while(iNumRecords > 0);
        sfs_fclose(fp);

        /* Done offloading, so set the read pointer in flash to the current record and erase all the records
           before it (up to the current block). */
        sfl_ReadPtrMark(0, TRUE);
    }
}
```

5.2 Offload Log Data to smxUSBD Serial Device

Data records can be retrieved through the smxUSBD serial port emulator, so the user can use a laptop to get the log data.

```
#include "smxflog.h"
#include "smxusbd.h"

void SendRecordsToUSBSerial(void)
{
    u8 RecordBuf[10*512];
    uint iNumRead;
    while(sfl_Read(0, RecordBuf, 10, &iNumRead, NULL, 0) == SFL_ERROR_NONE)
    {
        sud_SerialWriteData(0, RecordBuf, iNumRead*512);
    }
}
```

5.3 Erase Oldest Record When System is Idle

The application should erase old blocks when idle so that logging is never delayed. Only erase 1 block at a time if the application needs to continue logging.

```
void idle_task_main(void)
{
    /* do other idle jobs here */
    …
    /* now try to reclaim the oldest block of records */
    sfl_Erase(0, SFL_ERASE_ONE_BLOCK);
}
```

5.4 NAND Flash Array

To get a bigger capacity, a NAND flash array can created. For details please refer to the smxFFS User's Guide, Appendix B: Flash Chip Array.

A. File Summary

FILE	DESCRIPTION
flcfg.h flcfg.c	Configuration file for smxFLog.
flport.h flport.c	Porting definitions.
flog.c	Flash Logger API Implementation.
smxflog.h	Flash Logger API.
flhdw.h	NAND Flash Hardware IO routines.
norio.h	NOR Flash Hardware IO routines.

B. Size and Performance

B.1 Code Size

Code size varies depending upon CPU, compiler, and optimization level.

	ARM7/9	**ColdFire**
	IAR	CodeWarrior
smxFLog (without ECC)	3 KB	3 KB
smxFLog (with ECC)	5 KB	5 KB

B.2 Data Size

smxFLog was designed to minimize RAM use.

smxFLog core	32 B
smxFLog ECC (disabled by default)	256 B
smxFLog readback verify (disabled by default)	flash record size

B.3 Performance

The following are performance tables for smxFLog on platforms we tested. Raw read/write speeds are for the low-level driver only (no logging or file system), and are shown for comparison.

Performance highly depends upon the flash chip, bus speed, microprocessor speed, and RAM speed. It is recommended that you do measurements on your hardware before making final design decisions, if performance is critical. The results here are intended only to provide guidance. Also, keep in mind that smaller record sizes generate more overhead.

NAND: LPC2468

Record Size, Bytes	**Raw Data Read/Write KB/s**	**smxFLog Read/Write KB/s**
512	1795/1638	1438/851
1024	2184/1956	1657/1352

NAND: MCF5282

Record Size, Bytes	**Raw Data Read/Write KB/s**	**smxFLog Read/Write KB/s**
512	2730/1260	2338/712
1024	2730/1260	2338/910

NOR: LPC2468

Record Size, Bytes	Raw Data Read/Write KB/s	smxFLog Read/Write KB/s
64	2048/195	793/155
128	2048/195	1333/169
256	2048/195	1338/185
512	2048/195	2000/185

NOR: MCF5485

Record Size, Bytes	Raw Data Read/Write KB/s	smxFLog Read/Write KB/s
64	5461/264	4000/215
128	5461/264	5376/232
256	5461/277	5397/240
512	5461/282	5397/247

ECC calculation takes significant processor time. The times below show how much performance is reduced, especially for slower processors.

NAND: LPC2468

Record Size, Bytes	smxFLog Read/Write with ECC KB/s	smxFLog Read/Write without ECC KB/s
512	819/569	1438/851

NOR: MCF5485

Record Size, Bytes	smxFLog Read/Write with ECC KB/s	smxFLog Read/Write without ECC KB/s
64	1524/178	4000/215

C. Tested Hardware

C.1 NAND

- K9F1G08U on NXP LPC2468 board. Flash record size: 512, 1024.
- K9F2808U on our Avnet Coldfire 5282 add-on board. Flash record size: 512, 1024.

C.2 NOR

- 39VF320 on NXP LPC2468 board. Flash record sizes: 32, 64, 128, 256, 512.
- 28F128K3, 28F256K3, 28F128J3D on MCF5485EVB board. Flash record sizes: 32, 64, 128, 256, 512, 1024.