



smxFD™ User's Guide

NOR Flash Driver

Version 1.03
December 12, 2007

by Yingbo Hu

Table of Contents

1. Overview	1
2. Using smxFD	1
2.1 Installation	1
2.2 Getting Started	1
2.3 Basic Terms	1
2.4 Configuration Settings	1
3. NOR Flash Driver API.....	3
3.1 API Data Types.....	3
3.2 API Reference.....	3
4. NOR Flash Hardware IO Routines	8
4.1 IO Routine Data Types	8
4.2 IO Routine Reference	8
4.3 Verify the IO Routines.....	12
A. File Summary.....	13
B. Porting Notes.....	14
B.1 C Library Function Requirements.....	14
C. Tested Hardware	15
D. Preprogramming Flash.....	16

© Copyright 2006-2007

Micro Digital Associates, Inc.
2900 Bristol Street, #G204
Costa Mesa, CA 92626
(714) 437-7333
support@smxrtos.com
www.smxrtos.com

All rights reserved.

smxFD is a Trademark of Micro Digital Inc.
smx is a Registered Trademark of Micro Digital Inc.

1. Overview

smxFD is a Flash Driver designed for embedded systems to have a minimal RAM requirement. Typically 700 bytes is sufficient to achieve moderate performance.

The following restrictions must be applied to make it work:

- Assume the NOR driver uses only regular size blocks. Small blocks will be left for other uses, such as boot, program storage, etc. If smaller blocks are half the size of regular blocks, you can just group two together and treat them as a regular block.
- The logical sector size is fixed at compile time. It will normally be 512 bytes.
- Supported NOR flash types are those that: support block erase (change all cells to 0xFF) and each byte or word can be written at least 3 times (but bits can be changed only from '1' to '0').
- The Sector Map requires space, in bytes, equal to four times the number of sectors per block. Normally this will fit into the first sector of each block. Any space left over is not used.

If you have any question about the above restrictions, please contact us.

2. Using smxFD

2.1 Installation

smxFD is installed by copying files from the distribution media. When ordered with the smxFS, it is part of the smxFS release and is installed with it.

2.2 Getting Started

smxFD is configured to support smxFS and the processors and compilers it supports. If you are using smxFD for another file system, processor, or compiler, see section B Porting Notes, and implement the porting layer for your environment first, before using smxFD.

2.3 Basic Terms

Block	Minimum erasable unit of the flash chip. Some flash chips may use the term <i>sector</i> , instead.
Sector	Minimum unit of data for a file system. It is 512 bytes, by default.
Page	Maximum reading/writing unit of data of the flash chip. It is 256 or 512 bytes.

2.4 Configuration Settings

If you change any settings you should rebuild the smxFD library, clean.

2.4.1 fdcfg.h

fdcfg.h contains flash driver configuration constants that allow you select features and tune performance, code size, and RAM usage.

NOR_MAX_CHIP_NUM

This specifies how many physical NOR flash chips are in your system. Default value is 1.

NOR_PSMC_ENTRY_NUM

This is the internal Flash Driver mapping table cache size. Increasing it will increase the RAM requirement. Each mapping table item will use two bytes or four bytes, depending on the flash size, so if you have enough memory, increase it to 64 or 128. The default setting is 16.

NOR_PSMC_ITEM_NUM

This is the internal Flash Driver mapping table cache number. Increasing it will increase the RAM requirement. It must be greater than 1.

NOR_DEFAULT_SECTOR_SIZE

This is the default File System sector size. It should be 512 or multiple of 512 such as 1024.

NOR_LARGE_FLASH_SUPPORT

Set it to “1” if any of the following statements are true:

- Your flash chip's block size is larger than 64KB or
- Your flash chip's block number is larger than 65535

If you need to change it, contact us for details.

NOR_DATA_READ_BACK_CHECK

Set it to “1” to read back the data after writing to verify it. Read back will increase the reliability but decrease the performance. We recommend enabling it only when you are debugging your code.

NOR_START_BLOCK_NUM

NOR_END_BLOCK_NUM

These are the number of reserved blocks before and after the smxFD flash memory area.

2.4.2 fdport.h

fdport.h contains OS and compiler definitions.

SFD_BIG_ENDIAN_CPU

Set to 1 for a big endian CPU (e.g. ColdFire).

3. NOR Flash Driver API

The smxFD NOR Flash Driver API is defined in `norfd.h`, which contains the functions that are called by the file system

```
int nor_FlashInit (uint iID);
int nor_FlashRelease (uint iID);
int nor_BlockReclaim (uint iID, u32 iExpectedEmptySectorNum);
int nor_SectorRead (uint iID, u8 * pRAMAddr, u32 wLogicalIndex);
int nor_SectorWrite (uint iID, u8 * pRAMAddr, u32 wLogicalIndex);
int nor_SectorDiscard(uint iID, u32 wLogicalIndex);
u32 nor_SectorNum (uint iID);
u32 nor_SectorSize (uint iID);
```

3.1 API Data Types

These are defined in `fdport.h`.

u32, u16, etc Unsigned integer types of the size (bits) indicated.

3.2 API Reference

int **nor_FlashInit** (uint iID)

Summary Initialize smxFD NOR Flash Driver.

Descr This function should be called first before you use the NOR Flash Driver. It calls the NOR Flash Hardware IO Routine to initialize the NOR flash chip and retrieve the basic information of the NOR Flash Chip such as the Block size and Total Block number. It then tries to read the Flash chip to find if this flash chip was used before by smxFD NOR Flash Driver. If so, it retrieves the old information so the File System will get the saved data; otherwise, this function will format it.

Please make sure the Flash chip is empty, meaning all the data bytes are FF before you first use the chip. The driver will NOT automatically erase the whole chip before using it.

Pars nID The device ID you want to use. Valid values are 0 to `NOR_MAX_CHIP_NUM-1`.

Returns 1 Initialization succeeded.
 0 Initialization failed.

See Also `nor_FlashRelease()`

Example

```
if(nor_FlashInit(0))  
    printf("NOR Flash Initialized.");
```

`int` **nor_FlashRelease** (`uint iID`)

Summary Release smxFD NOR Flash Driver.

Descr This function should be called when you are done with the NOR Flash Driver. It releases the internal buffers allocated by `nor_FlashInit()` and calls the NOR Flash Hardware IO Routine to release the hardware resource.

Pars `nID` The device ID you want to use (the same ID you passed to `nor_FlashInit()`).

Returns `1` Release succeeded.
 `0` Release failed.

See Also `nor_FlashInit()`

Example

```
if(nor_FlashRelease(0))  
    printf("NOR Flash Released.");
```

`int` **nor_BlockReclaim** (`uint iID, u32 iExpectedEmptySectorNum`)

Summary Do Block Reclaim for smxFD NOR Flash Driver.

Descr This function will reclaim the used blocks to get more empty sectors that can be used by the file system. The driver will be forced to do Block Reclaim when the number of empty sectors has dropped to certain low level, but we recommend you call this API when your system is idle to improve performance when writing data. smxFS already provides an IOCTL function to do it. If you are using smxFS, please call that function to make sure it is multitasking safe.

Pars `nID` The device ID you want to use (the same ID you passed to `nor_FlashInit()`).
 `iExpectedEmptySectorNum` The expected number of empty sectors you want to get after the block Reclaim returns.

Returns The current number of empty sectors on the NOR flash chip.

See Also `nor_FlashInit()`

Example

```
printf("Current Empty Sector Number is %d\n.", nor_BlockReclaim(0, 0));
```

`int` **nor_SectorRead** (`uint iID, u8 * pRAMAddr, u32 wLogicalIndex`)

Summary Read one sector of data from the Flash chip.

Descr This function reads one sector of data from the flash chip. A sector is normally 512 bytes by default but may be another value so please make sure the memory buffer is big enough.

Pars

<code>nID</code>	The device ID you want to use (the same ID you passed to <code>nor_FlashInit()</code>).
<code>pRAMAddr</code>	The pointer to the memory buffer to hold the data. It should be at least one sector in size.
<code>wLogicalIndex</code>	The logical sector index from the file system's point of view.

Returns 0 The read succeeded.

See Also `nor_SectorWrite()`

Example

```
u8 pData[512];  
nor_SectorRead(0, pData, 0); /* Read the MBR if using FAT file system */
```

`int` **nor_SectorWrite** (`uint iID, u8 * pRAMAddr, u32 wLogicalIndex`)

Summary Write one sector of data to the Flash chip.

Descr This function writes one sector of data to the flash chip. A sector is normally 512 bytes by default.

Pars

<code>nID</code>	The device ID you want to use (the same ID you passed to <code>nor_FlashInit()</code>).
<code>pRAMAddr</code>	The pointer to the memory buffer holding the data. It should be at least one sector in size.
<code>wLogicalIndex</code>	The logical sector index from the file system's point of view.

Returns 0 The write succeeded.

See Also `nor_SectorRead()`

Example

```
u8 pData[512];  
memset(pData, 0xFF, 512);  
nor_SectorWrite(0, pData, 0); /* Empty the first sector of file system */
```

`int` **nor_SectorDiscard** (`uint iID`, `u32 wLogicalIndex`)

Summary Tell the flash driver that a sector is not used by the file system.

Descr This function will mark one sector as discarded, which means the file system does not use the data in this sector so the driver can reclaim it later when it does Block Reclaim.

Pars `nID` The device ID you want to use (the same ID you passed to `nor_FlashInit()`).
 `wLogicalIndex` The logical sector index of the discarded sector.

Returns 0

See Also `nor_SectorRead()`, `nor_SectorWrite()`, `nor_BlockReclaim()`

Example

```
nor_SectorDiscard(0, 100);
```

`int` **nor_SectorNum** (`uint iID`)

Summary Return the total number of sectors on this flash chip.

Descr This function returns the total number of sectors to file system.

Pars `nID` The device ID you want to use (the same ID you passed to `nor_FlashInit()`).

Returns Total number of sectors.

See Also `nor_FlashInit()`, `nor_SectorSize()`

Example

```
printf("Total sector number of device %d is %d\n", 0, nor_SectorNum(0));
```

int **nor_SectorSize** (uint iID)

Summary Return the sector size, in bytes, of this flash chip.

Descr This function returns the sector size to file system.

Pars nID The device ID you want to use (the same ID you passed to `nor_FlashInit()`).

Returns Flash sector size.

See Also `nor_FlashInit()`, `nor_SectorNum()`

Example `printf("Sector size of device %d is %d\n", 0, nor_SectorSize(0));`

4. NOR Flash Hardware IO Routines

The smxFD NOR Flash Hardware IO Routines are defined in norio.h. These must be implemented for your particular flash hardware and hardware platform. We provide some sample code in the norio.c which can be used as a reference.

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 these will be the same as a normal flash block (e.g. 64KB). To handle this, you can either exclude these small blocks from the part of the flash used by smxFD or you can treat them as a single block by adding special handling to nor_IO_SectorRead() and nor_IO_SectorWrite().

```
int nor_IO_FlashInit (uint iID, NOR_DEVINFO * pDevInfo);  
  
int nor_IO_FlashRelease (uint iID);  
  
int nor_IO_SectorRead (uint iID, u8 * pRAMAddr, uint wPhySectorIndex, uint wSectorSize);  
  
int nor_IO_SectorWrite (uint iID, u8 * pRAMAddr, uint wPhySectorIndex, uint wSectorSize);  
  
int nor_IO_BlockErase (uint iID, u32 wBlockIndex);  
  
int nor_IO_InfoWrite (uint iID, void * pInfo, uint wBufSize, u32 wPhyBlockIndex, uint wOffset);  
  
int nor_IO_InfoRead (uint iID, void * pInfo, uint wBufSize, u32 wPhyBlockIndex, uint wOffset);
```

4.1 IO Routine Data Types

```
typedef struct  
{  
    u32 dwTotalBlockNum;  
    u32 dwBlockSize;  
} NOR_DEVINFO;
```

4.2 IO Routine Reference

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

Summary Initialize the flash chip.

Descr This function initializes the NOR flash chip and gets the basic information of it. You need to set the member variables, dwTotalBlockNum and dwBlockSize of structure NOR_DEVINFO, so the driver can initialize its own internal data structures.

Pars nID The device ID you want to use.
 pDevInfo The pointer to the device information structure you need 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);
}
```

`int` **nor_IO_FlashRelease** (`uint iID`)

Summary Release the flash chip.

Descr This function releases the NOR flash chip.

Pars `nID` The device ID you want to use.

Returns 1

See Also `nor_IO_FlashInit()`

Example

```
nor_IO_FlashRelease (0);
```

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

Summary Read one sector of data from the flash chip.

Descr 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 your flash chip cannot read 512 bytes each time, you may need 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.

Pars `nID` The device ID you want to use.
`pRAMAddr` The pointer to the memory buffer to hold the data. The buffer should be at least one sector in size.
`wPhySectorIndex` The physical sector index of the flash chip. You may need 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 wPhySectorIndex, uint wSectorSize)

Summary Write one sector data to the flash chip.

Descr This function writes one sector of data to the flash chip. The sector is normally 512 bytes by default. If your flash chip cannot write 512 bytes each time, you may need to map this function call to multiple flash commands. For example, some serial flash can only write up to 256 bytes in one command. Our sample code already shows how to handle this case.

Pars

<code>nID</code>	The device ID you want to use.
<code>pRAMAddr</code>	The pointer to the memory buffer holding the data to write. It should be at least one sector in size.
<code>wPhySectorIndex</code>	The physical sector index of the flash chip. You may need to map this index to the flash address.
<code>wSectorSize</code>	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 wPhyBlockIndex, uint wOffset)

Summary Read a few bytes from the flash chip.

Descr 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 you want to use.
pInfo	The pointer to the memory buffer pointer holding the data.
wBufSize	The size of the buffer pointed to by pInfo.
wPhyBlockIndex	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 wPhyBlockIndex, uint wOffset)

Summary Write a few bytes to the flash chip.

Descr 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 you want to use.
pInfo	The pointer to the memory buffer pointer to hold the data.
wBufSize	The size of the buffer pointed to by pInfo.
wPhyBlockIndex	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.

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

Pars `nID` The device ID you want 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);  
}
```

4.3 Verify the IO Routines

We provide sample code to verify if your porting of the IO routines is correct. The code is in `fdtest.c`. Please run it first after you complete your porting. Then test the integration with your file system.

A. File Summary

FILE	DESCRIPTION
fdcfg.h	Configuration file for smxFD.
fport.h	Porting definitions, macros, and functions for hardware and OS. Ported to SMX, as shipped.
norfd.c, .h	NOR Flash Driver API.
norio.c, .h	NOR Flash Driver Hardware IO Routines.
fdtest.c	Sample code to verify API and Hardware IO Routines

B. Porting Notes

The porting layer is simple. It does not need any OS system calls. The most important part of smxFD porting is to implement the NOR Flash Hardware IO Routines, according to your specific hardware. We already provide the sample code for M25PXX serial flash driver in `norio.c`. You can use it as a starting point. Please read section 4 NOR Flash Hardware IO Routines for details about how to implement those functions.

B.1 C Library Function Requirements

This is a list of C library functions that smxFD calls. If your compiler does not provide some of these, you should implement them yourself.

- `free()`
- `malloc()`
- `memcpy()`
- `memcmp()`
- `memset()`

C. Tested Hardware

- STMicro M25P10, M25P80 on STR710-Eval board
- STMicro M25P16 on our Avnet Coldfire 5282 add-on board.
- 39VF320 NOR Flash on LPC2468 board
- 28F128K3, 28F256K3, 28F128J3D NOR Flash on MCF5485EVB board

D. Preprogramming Flash

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

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

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

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

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

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

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

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