



# smxFS™ User's Guide

FAT12/16/32 File System

Version 2.30  
February 23, 2024

by Yingbo Hu and David Moore

## Table of Contents

<b>1. Overview .....</b>	<b>1</b>
1.1 Relationship to Other SMX Filesystems .....	1
1.2 Lite Configuration .....	2
<b>2. Using smxFS.....</b>	<b>3</b>
2.1 Getting Started.....	3
2.2 Basic Terms .....	3
2.3 Configuration Settings.....	3
2.4 Using the API .....	8
2.5 Error Handling.....	8
<b>3. Theory of Operation.....</b>	<b>10</b>
3.1 Device Drivers .....	10
3.2 File Names.....	11
3.3 FAT12/16/32 .....	11
3.4 FAT Management.....	11
3.5 Directory Management .....	12
3.6 Data Structures.....	12
3.7 Memory Management.....	13
3.8 Reentrancy Protection.....	14
3.9 Media Change and Mounting .....	14
3.10 Multiple Drives / Sockets and Partitions .....	15
3.11 Power Fail Safety.....	16
3.12 Check Disk .....	17
3.13 Clean Shutdown Flag.....	19
3.14 Dual FATs .....	19
3.15 Alternate Filesystem Access.....	20
3.16 Safety Checks .....	21
3.17 Fast Cluster Allocation .....	21
3.18 Accessing a File with Multiple File Handles.....	22
3.19 Pass Through Mode.....	22
<b>4. File System API .....</b>	<b>24</b>
4.1 API Data Types.....	24
4.2 API Summary .....	25
4.3 API Reference.....	27

<b>5. Device Driver Details .....</b>	<b>60</b>
5.1 Block Device Interface .....	60
5.2 Test Code for New Drivers.....	60
5.3 Driver-Specific Notes .....	61
<b>A. File Summary.....</b>	<b>65</b>
<b>B. Multiple Language File Name Support .....</b>	<b>66</b>
<b>C. FAT Format .....</b>	<b>67</b>
C.1 Main Regions .....	67
C.2 Directories and Files .....	67
<b>D. Size and Performance .....</b>	<b>68</b>
D.1 Code Size .....	68
D.2 Data Size (RAM Requirement).....	68
D.3 Performance.....	69
<b>E. Tested Hardware .....</b>	<b>75</b>
E.1 CompactFlash Devices.....	75
E.2 MMC/SD Devices.....	75
E.3 NAND Flash Devices.....	75
E.4 NOR Flash Devices.....	75
E.5 USB Mass Storage Devices.....	75
<b>F. Glossary .....</b>	<b>77</b>

© Copyright 2004-2024

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

All rights reserved.

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

# 1. Overview

smxFS is a FAT12/16/32 file system designed for embedded systems. It supports fixed and removable media, and offers drivers for the media typically used in embedded systems<sup>1</sup> such as NAND/NOR Flash, USB flash disks, MMC/SD cards, and CompactFlash. It is DOS/Windows-compatible, so media written by smxFS are interchangeable with these OSes and many others that support the FAT filesystem.

smxFS is reentrant (multitasking safe) and requires minimal ROM and RAM (17KB for code and 4KB for RAM). It supports VFAT<sup>2</sup> (long file names), which is compatible with Win32 operating systems.

smxFS has the standard C library file API (fopen(), fread(), etc.), which is commonly known.

smxFS consists of these components:

1. **FS API** provides the standard C library API: fopen(), fread(), fwrite(), fseek(), fclose(), etc. to the application.
2. **FS Path** implements the Directory Entry and FAT table structure handler. It supports FAT12/16/32 and Long File Names.
3. **FS Mount/Format** implements the mount/format functions for inserted devices.
4. **FS Cache** implements the Cache functionality for Data, FAT, and Directory entries.
5. **FS Driver Interface** uses a unique interface to integrate all the devices into the file system.
6. **FS Port** implements some compiler and hardware related definitions, macros, and functions. It also includes Multi-language related functions, such as functions to convert Chinese characters to Unicode.
7. **FS Utility** implements the check disk and fix functions of the file system.

## 1.1 Relationship to Other SMX Filesystems

**smxFFS** is a proprietary filesystem we developed that is not DOS compatible. It is more efficient for use with flash memory than a DOS FAT filesystem, such as smxFS. It is incompatible with DOS/Windows media, cannot be extended with other disk drivers, and has a smaller API than smxFS. It offers power-fail safety, unlike smxFS. See section 3.11 Power Fail Safety for more information.

**smxFLog** is a simple flash logger we designed for logging data sequentially to flash media, at high or low speed. It can coexist with another filesystem, in a different region of the flash. It maximizes performance, minimizes wear, and offers power-fail safety. Data can be retrieved and written into another filesystem or sent to a host via communication link.

---

<sup>1</sup> Use smxFile for mechanical devices such as hard disks and removable magnetic media.

<sup>2</sup> VFAT is patented by Microsoft. US Patent #5,758,352. Microsoft may require a license fee to use it. Setting SFS\_VFAT\_SUPPORT to 0 will avoid potential patent infringement problems.

## 1.2 Lite Configuration

smxFS can be configured to a subset of functionality to reduce the code and data size significantly, to allow it to be used on smaller SoCs. Set `SFS_FULL_FEATURES_SUPPORT` to 0 in `fcfg.h` to omit advanced features.

Section 4.2 API Summary indicates which API functions are available in the Lite configuration. As the notes following that list explain, some require changing a config setting to enable.

Lite limitations:

- Long File Name support is omitted. You must use 8.3 format names to access files and paths.
- FAT32 is disabled by default but you can set `SFS_FAT32_SUPPORT` to 1 to enable it again
- Current Working Directory is disabled. You must use full path names such as `A:\dir\subdir\file1` to access files. Related APIs are omitted from the source code.
- The following APIs are omitted:
  - `chkdsk()`
  - `findfirst()/findnext()/findclose()`
  - `rename()`
  - `setprop()/getprop()/stat()/chmod()/timestamp()`
  - `setvolname()/getvolname()`
- The following APIs are disabled by default:
  - `mkdir()/rmdir()`. Can be enabled by setting `SFS_MKDIR_SUPPORT` to 1.
  - `format()/partition()`. Can be enabled by setting `SFS_FORMAT_SUPPORT` to 1. These functions are provided primarily to support fixed media, since removable media are preformatted and can be reformatted on a PC.

## 2. Using smxFS

### 2.1 Getting Started

#### 2.1.1 Running the Demo

DEMO\fsdemo.c is a simple demonstration program of basic smxFS operations. It first creates a large file to test write performance, then reads it back to determine read performance. Then it creates a directory to hold test files it generates. These files are of random lengths (up to a configurable size). Basic operations are performed on these randomly and the demo checks the results. The demo has configuration settings near the top of the file. If you are using the default settings, ensure you have about 30 MB free space on the media.

### 2.2 Basic Terms

If you are unfamiliar with terms such as FAT, sector, cluster, file handle, and file pointer, please take a moment now to review the Glossary.

### 2.3 Configuration Settings

#### fcfg.h

fcfg.h contains file system configuration constants that allow you select features and tune performance, code size, and RAM usage.

#### SFS\_DRV\_

These specify which of the smxFS drivers are present. Drivers are available optionally. Note that if you add a new driver, you do **not** need to add a new setting here. Simply link it and register it. See the section 3.1.4 Adding a New Driver for more information.

#### SFS\_DATA\_CACHE\_SIZE

The default data cache size. Increasing it will increase the file read/write speed, especially for large files and continuous read/write operation, but it will use more RAM. The device driver may overwrite the default settings. For example, a RAM disk does not need a big data cache so the driver will overwrite it to 512 bytes. The size should be a multiple of the disk's sector size. For disks whose sector size is 512 bytes, the minimum cache size is 512. Some disks may use 2048 bytes sector size, in which case the minimum size is 2048.

#### SFS\_DIR\_CACHE\_SIZE

The default directory cache size. Increasing it will increase the file open/find/delete speed, especially in a directory that has a lot of files, but it will use more RAM. The device driver may overwrite the default settings. See the discussion for SFS\_DATA\_CACHE\_SIZE.

#### SFS\_FAT\_CACHE\_SIZE

The default FAT cache size. Increasing it will increase the file read/write speed, especially for large

files, but it will use more RAM. The device driver may overwrite the default settings. See the discussion for `SFS_DATA_CACHE_SIZE`.

### **SFS\_CACHE\_BUF\_ALIGNMENT**

The alignment required for cache data buffers. Needed by some drivers.

### **SFS\_MAX\_DEV\_NUM**

The maximum number of device drivers that can be registered with smxFS at the same time. (Device drivers are registered by calling `sfs_devreg()` and can be unregistered with `sfs_devunreg()`.) Increasing this setting has very little impact on RAM usage. smxFS uses it to size an array of pointers, so each increment only adds 4 bytes of BSS data. Only when smxFS actually registers a device, does it `malloc()` a buffer for the `DEVICEHANDLE` structure for that driver.

### **SFS\_CWD\_MAX\_ENTRIES**

Maximum number of entries in the current working directory table. In a multitasking environment, this should be set to the number of tasks that call `sfs_chdir()`. Note that each increment of this setting allocates only space for a pointer. Buffers for each entry (handle and path string) are malloced as needed.

### **SFS\_PATHSEP**

Set the path separator character as desired.

### **SFS\_FIRST\_DRIVE**

The first logical drive letter to be assigned. Each registered device is a logical disk and its letter is the device ID plus `SFS_FIRST_DRIVE`. See the section 3.1.1 Drive Lettering for more information.

### **SFS\_READONLY**

If set to 1, smxFS becomes a read-only filesystem. All the API functions to modify the contents of the disks are omitted, such as `sfs_fwrite()`, `sfs_ftruncate()`, `sfs_rename()`, `sfs_mkdir()`, `sfs_rmdir()`, `sfs_fopen()` will return an error if you try to create a file or open a file for writing. Each driver (`XFS\fd*.c`) also has a `READONLY` setting. If you want to ensure that it is impossible to write to the disk and keep out as much unnecessary code as possible, enable that setting at the top of each driver (`.c`). The drivers are considered to be independent of smxFS, so they don't include `fcfg.h`. Also, they might be shared by smxUSB. This is why they have separate defines instead of checking `SFS_READONLY`. Also set `SFD_READONLY` (`XFD\fdcfg.h` if using the NAND or NOR driver).

### **SFS\_FULL\_FEATURES\_SUPPORT**

If set to 1, additional functions will be enabled such as `sfs_format()`, `sfs_rename()`, `sfs_findfirst()`, and `sfs_findnext()`. Setting this to 0 reduces code space if you don't need these functions. Of course, you can also comment out functions you don't need, individually, but this is unnecessary if your linker can deadstrip unused functions rather than just whole object files. Alternatively, you can enable parts of the extended API by enabling the `_SUPPORT` settings below. This is set to 0 for the Lite version.

### **SFS\_FAT32\_SUPPORT**

If set to 1, FAT32 code is enabled. This setting can be disabled to eliminate extra code for FAT32 support, if you know you only need to support FAT12 and FAT16 disks. Setting this to 0 is probably only appropriate if you are using only fixed media such as resident NAND/NOR flash. Removable media could be formatted to any FAT type so this setting should be 1.

### **SFS\_VFAT\_SUPPORT**

If set to 1, long file name (LFN) code is enabled. VFAT is patented by Microsoft. US Patent

#5,758,352. Microsoft may require a license fee to use it. Setting SFS\_VFAT\_SUPPORT to 0 will avoid potential patent infringement problems.

### **SFS\_CWD\_SUPPORT**

If set to 1, current working directory support is enabled. This means you can set the current working directory once and then just use relative file names to access files, in the same task; it is not necessary to pass the full path of files to the APIs.

### **SFS\_FINDFIRST\_SUPPORT**

If set to 1, you can use findfirst/findnext to find files in a directory. This feature is needed if you want to implement a dir feature or to search for files with names matching a pattern (i.e. wildcards).

### **SFS\_MKDIR\_SUPPORT**

If set to 1, you can use sfs\_mkdir/rmdir() to create and delete directories in the file system.

Note: Even if this is 0, smxFS can still read and write files to directories that already exist; it just cannot create new directories or remove directories.

### **SFS\_VOLUME\_SUPPORT**

If set to 1, you can use sfs\_getvolname/setvolname() to get or set the volume name of a disk.

### **SFS\_PROPERTY\_SUPPORT**

If set to 1, you can use sfs\_chmod/stat/timestamp() to read or modify the file's properties, such as file length (read only), access permissions, and timestamp.

### **SFS\_CHKDSK\_SUPPORT**

If set to 1, you can use sfs\_chkchk() to check and fix errors on the disk.

### **SFS\_BIG5\_SUPPORT**

If set to 1, smxFS will convert any Traditional Chinese file name to Unicode so Windows can display this file name correctly. You must also enable SFS\_VFAT\_SUPPORT if you want to enable this feature. Also see Appendix B. Multiple Language File Name Support.

### **SFS\_GB2312\_SUPPORT**

If set to 1, smxFS will convert any Simplified Chinese file name to Unicode so Windows can display this file name correctly. You must also enable SFS\_VFAT\_SUPPORT if you want to enable this feature. Also see Appendix B. Multiple Language File Name Support.

### **SFS\_VFAT\_ALWAYS\_USE\_LFN**

Set to 1 to always generate the long file name entry even for short (8.3) file names. The advantage is the name will be stored in the specified upper/lower case (e.g. TheFile.txt not THEFILE.TXT). The disadvantage is it uses 2 directory entries instead of 1, so for FAT16 this reduces the maximum size of the root directory by half. When set to 1, it mimics the behavior of Windows; when 0 it reduces directory size.

### **SFS\_SAFETY\_CHECKS**

If set to 1, extra code is enabled to do safety checks. Normally this should be enabled during development to help you troubleshoot a problem and disabled in the final release if the performance and code size is critical.

### **SFS\_HANDLE\_BAD\_SECTOR**

If set to 1, extra code is enabled to handle bad sectors. When writing and a bad sector is encountered,

smxFS will try to replace the cluster with a new one. Bad sectors will only be replaced if the disk driver reports SB\_BD\_BAD\_BLOCK.

### **SFS\_CLN\_SHUTDOWN\_SUPPORT**

If set to 1, smxFS will set a flag in the FAT table if the file system is not shut down cleanly. For example, it will remain set if a power fail or system crash occurs when some files are still open. See section 3.13 Clean Shutdown Flag for discussion.

**Note: We have not fully studied/tested this option to ensure it is reliable for power loss at every point in the code.**

### **SFS\_FAT\_FSINFO\_SUPPORT**

If set to 1, smxFS will reserve one special sector to save the total number of free clusters, for FAT12/16. This feature is normally used only for FAT32. Enabling it will greatly reduce the time to get the free size of a disk, because smxFS does not need to scan the whole FAT to determine this number. However, see the discussion of SFS\_USE\_FAT32\_INFO below. This is ignored for removable media when formatting a disk because it is not supported by other file systems. The problem is that they will not update the total number of free clusters as files are created and deleted, so then smxFS will use the old (wrong) number.

### **SFS\_USE\_FAT32\_FSINFO**

Set to 1 to use the FAT32 FSInfo sector to store the number of free clusters and next free cluster information. If 1, pc\_freekb() will read the values from the FSInfo sector; otherwise, it will scan the whole FAT the first time, which can be very slow. The FSInfo sector can be wrong, though, causing sfs\_freekb() to report the wrong size. For example, Windows chkdsk does not update this sector when lost chains are recovered. Also, with this option on, the FSInfo data is updated as the disk changes, which reduces performance.

### **SFS\_2NDFAT\_SUPPORT**

If set to 1, smxFS will write the FAT information to the second FAT area. Because Windows normally does not check the second FAT, we recommend disabling this feature (set to 0) to improve performance. See section 3.14 Dual FATs for discussion.

### **SFS\_UPDATE\_WHOLE\_FATDIR\_CACHE**

If set to 1, smxFS will write back the whole FAT/Dir cache when a cache miss happens. Set to 0 to write only the cache sector being swapped out.

### **SFS\_FREECLUS\_SUPPORT**

If set to 1, smxFS will allocate an additional cache to store a list of free clusters. This feature is useful when streaming large data files such as video, to avoid long delays while scanning for free clusters that could cause buffer overflow of the incoming data stream. For details, please see the discussion in section 3.17.2 Free Cluster Cache. The following two additional settings are used to control the memory allocated for this free cluster cache.

#### **SFS\_FREECLUS\_CACHE\_SIZE**

This is the free cluster cache size.

#### **SFS\_FREECLUS\_SCAN\_CACHE\_SIZE**

This is the buffer size used to scan (find) the free cluster within the FAT.

#### **SFS\_FREECLUS\_TRIGGER**

Sets how often the free cluster scanning is triggered. If the number of free clusters is lower than this trigger, smxFS will scan the FAT to find more free clusters and store them in the free cluster cache.

**This feature is disabled if the disk is formatted by FAT12 because the FAT is only a few sectors.**

### **SFS\_NUMERICTAIL\_CACHE\_SUPPORT**

Set to 1 to enable the short file name numeric-tail cache. If you are creating a lot of similar long file names within a directory, such as Log-2010-11-8.dat, Log-2010-11-9.dat, etc. smxFS must create corresponding short file names like LOG-20~1.DAT, LOG-20~2.DAT, etc. and verify that short file name is unique in that directory. When you are creating hundreds of this kind of file name, the compare procedure will become very time-consuming without this cache.

### **SFS\_NUMERICTAIL\_CACHE\_SIZE**

This is the size of the numeric-tail cache. This is the number of directories to cache.

### **SFS\_FORMAT\_SUPPORT**

If set to 1, you can use `sfs_format()` to format disks. This also enables smxFS to automatically format a disk that is unformatted (or formatted with something other than the FAT12/16/32 format). Whether a device will be autoformatted depends on how the driver sets `pDeviceInfo->wAutoFormat` in its `ioctl` routine. This feature is disabled by default for removable disks such as USB thumb drive.

### **SFS\_COPY\_BUF\_SZ**

Size of buffer allocated by `sfs_copy()` to copy data from one file to another. Set to a multiple of sector size. 4KB is a good default value to use, unless RAM is limited.

### **SFS\_LONGFILENAME\_LEN**

Maximum length for long file names. 255 is the maximum value. See 3.2 File Names for more discussion.

### **SFS\_PATHFILENAME\_LEN**

Maximum total length for path and filename. 260 is the maximum value. See 3.2 File Names for more discussion.

### **SFS\_FILENAME\_IN\_HANDLE**

Set to 1 to save a copy of full path and filename in file handle structure. Only used for debug purposes. Allows smxAware to display opened filenames.

### **SFS\_USE\_C\_HEAP**

Set to 1 for smxFS to use C library functions `malloc()/free()` to allocate cache and data buffers. Also set to 1 for smx since smx maps these onto smx heap functions. If your compiler does not provide `malloc()/free()` functions, set it to 0 so smxFS will use built-in simple heap functions in `fport.c`. They only work for 32-bit systems.

### **SFS\_HEAP\_SIZE**

The built-in heap size if you set `SFS_USE_C_HEAP` to 0. Heap size depends on the cache size and the number of files you will open simultaneously. The default size is 64KB, which is big enough for two disks with the default cache size settings.

### **SU\_DEBUG\_LEVEL**

Specifies the debug level. The following values are supported:

- 0 disables all debug output and debug statements are null macros
- 1 only output fatal information
- 2 output additional warning information
- 3 output additional status information
- 4 output additional device change information

- 5 output additional data transfer information
- 6 output interrupt information

## 2.4 Using the API

smxFS uses the standard C library API, which many programmers are familiar with. A few additional calls were added. The API is documented in section 4. File System API.

Below is a simple example that shows basic smxFS operations. For simplicity, the code does not test return values of the calls to see if they are successful, but you should do so in your code. Also, note that the drive letters indicated are correct if SFS\_FIRST\_DRIVE is 'A'. See section 3.1.1 Drive Lettering for more information. The lines that register the drivers assume that you have enabled these drivers in fcfg.h. Also see demo.c or fsdemo.c for more example code.

```
#include "smx.h"    /* SMX RTOS */
#include "smxfs.h" /* smxFS API header file */

void main(void)
{
    FILEHANDLE fh;
    u8 pData[100]; /* fill pData with some values (not shown) */

    if(sfs_init() == SB_PASS) /* initialize smxFS */
    {
        /* Register device drivers. */
        sfs_devreg(sfs_GetRAMInterface(), 0); /* A: */
        sfs_devreg(sfs_GetUSBInterface(), 1); /* B: */
        ...

        /* Do basic file operations. (Should normally check return values.) */
        fh = sfs_fopen("A:\\testfile.bin", "w+b"); /* open file */
        sfs_fwrite(pData, 100, 1, fh); /* write some data */
        sfs_fseek(fh, 0, SFS_SEEK_SET); /* rewind to the beginning */
        sfs_fread(pData, 100, 1, fh); /* read it back */
        sfs_fclose(fh); /* close file */
    }
}
```

## 2.5 Error Handling

All smxFS APIs will return an error if the low level disk driver reports any IO problem or smxFS itself finds there are some internal errors in the system file data structures such as a FAT node or directory entry. You should check API return values and after any error, do the following operations. Note that sometimes the error may not be caused by the file you are accessing, so it is best to check the whole disk with sfs\_chkdisk() or on a PC.

1. Close all open files, and do not read/write any more data from/to the file system.
2. For a removable disk such as USB thumb drive or SD card, call sfs\_devstatus() to check if the disk is removed.
3. Your application should warn the user that there is something wrong when accessing the media, and if the disk is removable, suggest he check the disk on a PC to fix possible problems.
4. Your application could call sfs\_chkdisk() to fix possible problems and try the file operation again.
5. smxFS outputs critical error messages via sfs\_DebugL() and sfs\_DumpHexL(). We recommended you set SFS\_DEBUG\_LEVEL to 2 and save the output error or warning message to your system log.

See section 3.16 Safety Checks for information about the conditions smxFS checks if SFS\_SAFETY\_CHECKS is 1.

## 3. Theory of Operation

### 3.1 Device Drivers

The following is basic information about using device drivers with smxFS. For more detailed information and information about the interface functions, see section 5. Device Driver Details and section Block Device Interface in the smxBase User's Guide.

#### 3.1.1 Drive Lettering

Drive lettering is simple. It is determined by:

DeviceID + SFS\_FIRST\_DRIVE

DeviceID is the ID value passed to `sfs_devreg()`, and `SFS_FIRST_DRIVE` is a letter defined in `fcfg.h`, which is 'A' by default.

#### 3.1.2 Registering a Driver

The built-in device drivers supported by smxFS are registered by calls to `sfs_devreg()` in `smxfs_init()` in `smxmods.c`. See the example in the `sfs_devreg()` call description in section 4. File System API. To register your own driver, do it the same way. Note that the number of drivers that may be registered simultaneously is controlled by `SFS_MAX_DEV_NUM` in `fcfg.h`.

For information about registering drivers for multiple devices of the same type, see the section 3.10 Multiple Drives / Sockets and Partitions.

#### 3.1.3 Available Drivers

- ATA (IDE)
- CompactFlash
- MMC/SD card
- NAND flash
- NOR flash
- RAM disk
- USB disk

The RAM disk driver is included with smxFS. The others are available optionally. Please contact us if the driver you need is not listed. It may be recently implemented or under development.

#### 3.1.4 Adding a New Driver

To add a new device driver to smxFS, it is only necessary to implement the 7 device driver interface functions and `GetDriverInterface()`, and register the driver with smxFS, using a call to `sfs_devreg()`. ("Driver" means the name of the driver, such as "RAM" or "USB".) Use the RAM disk driver as a guide. *It is not necessary to make changes to any smxFS files.*

See section Block Device Interface in the smxBASE User's Guide for more information about the driver interface functions.

## 3.2 File Names

smxFS supports DOS-style 8.3 names and Win32-style long file names (VFAT). (8.3 means 8 characters for the name and 3 for the extension.) When creating a file, if a name is 8.3 characters or shorter, it is converted to upper case and only a short directory entry is created. If a name is longer than 8.3 characters, a long directory is created that preserves the case of the name passed to `sfs_fopen()` and a short 8.3 alias is created that is all upper case. The alias is created using the same method Windows uses.

When searching for an existing file or subdirectory, the case of the name (upper or lower) is ignored when comparing the filename parameter in API calls to the filename on disk.

The maximum long file name length is set by `SFS_LONGFILENAME_LEN` (255) in `fcfg.h`. The maximum length of the path plus file name is specified by `SFS_PATHFILENAME_LEN` (260) in `fcfg.h`. The maximum values for these are indicated in parentheses; they can be set smaller. As specified by Microsoft, the drive letter, colon, path, name, and NUL must total  $\leq 260$  chars. This does not mean just what is specified in a string passed to an API function, but the actual full path and name of the file (i.e. it does not matter if you have changed into a subdirectory; it does not shorten the path).

## 3.3 FAT12/16/32

The FAT type (12/16/32) is set when the media is formatted. It cannot be changed without reformatting the media. If it was pre-formatted by another OS, smxFS determines the FAT type from the boot sector and uses that. Otherwise, if smxFS is used to format the media (with `sfs_format()`), the FAT type depends on the size of the media. smxFS uses the following simple rules:

```
size <= 8MB  —> FAT12
size <= 2GB  —> FAT16
size > 2GB  —> FAT32
```

The number of sectors per cluster is specified by the `SecPerClusArrXX[]` tables in `fmount.c`. You can modify these default settings as desired. For example, if your application will only create a few big files then you can increase the cluster size to reduce the FAT size to improve the performance. The settings in this table are only used when you format the disk by calling `sfs_format()`. The cluster size of a pre-formatted disk will not be changed unless you re-format it.

These assignments are arbitrary. Other possibilities could be used. There is a certain maximum disk size that is possible for each FAT type, but otherwise, by using different values of sectors per cluster, you can use a different FAT type. For example, if you wish to make a FAT16 disk that is larger than 512MB, you would need to modify the code in `FormatDevice()` and possibly the values in `SecPerClusArr16[]`, both in `fformat.c`. We recommend you leave this alone, but the reason we point it out is because media formatted by a different OS or with a special disk utility might use different ranges than smxFS does for media it formats. smxFS handles these fine; just don't be confused.

## 3.4 FAT Management

smxFS manages 1 or 2 FATs depending on the setting of `SFS_2NDFAT_SUPPORT`. By default this setting is 0 so only 1 FAT is maintained, for better performance. Many filesystems assume 2 FATs.

Microsoft claims all of their operating systems work for any number of FATs  $\geq 1$ , but other operating systems may not. They recommend using 2 for best compatibility.

For discussion of how the FAT works, see the Appendix C. FAT Format.

## 3.5 Directory Management

There are two status values for a directory entry, stored in the first byte. 0xE5 means the entry is free, and 0x00 means all entries below it are free. When files are deleted smxFs puts the 0x00 at the lowest offset it can, to eliminate unnecessary checks of the following unused entries. If many files in a directory are deleted, such that a whole cluster of the subdirectory entries becomes free, smxFs releases the cluster for use for data or other directories. Consider this example:

1. Assume the directory entries look like this:

```
FILE1
FILE2
FILE3
FILE4
<FreeBelow (0x00)>
```

2. After smxFs deletes FILE3, it becomes:

```
FILE1
FILE2
<FreeThis (0xE5)>
FILE4
<FreeBelow (0x00)>
```

3. After smxFs deletes FILE4, it changes the status of the 3<sup>rd</sup> entry to 0x00:

```
FILE1
FILE2
<FreeBelow (0x00)>
<FreeThis>
<FreeBelow>
```

Now when searching for a file name, the search stops at the 3<sup>rd</sup> entry.

## 3.6 Data Structures

Most FAT file system structures such the BPB and Dir Entry follow the Microsoft FAT white paper. There are also some internal data structures that are only related to the implementation. The following three are the most important:

### 3.6.1 FILEHANDLE

This structure is allocated for each file, just like the FILE \* of the standard C library. It maintains the important information of a file, such as the file attribute, file size, read/write pointer, first data cluster index, path cluster index, and a file-level data cache. Normally the user should **not** directly access or modify the fields of this structure.

### 3.6.2 DEVICEHANDLE

This structure is used to maintain the important information for each disk, such as FAT type, start/end cluster, sector size, cluster size, root directory size, first data cluster index, total free cluster number and disk-level data cache for directory, file allocate table, and data. This data structure is only used within smxFS. The user cannot access it directly through any API function call.

### 3.6.3 FILEINFO

This structure is used when the user wants to find a file, retrieve/modify file properties such as permission settings, and timestamp. smxFS does not maintain this data structure globally so it is always created as a local variable. The user can access/modify some fields of this structure to get/set the file properties. Those fields are:

st_mode	File permission mode.
st_size	File size. Read-only; user should not change it.
st_ctime	File creation time.
st_mtime	File modification time.
st_atime	File last access time.
st_dev	File's device ID. Read-only; user should not change it.
name	File name. Read-only; user should not change it.
bAttr	File attribute byte. Can be changed to add/remove HIDDEN or SYSTEM attributes.

## 3.7 Memory Management

Most smxFS memory is allocated dynamically by the malloc() function. Required memory will only be allocated when the device is successfully mounted. Each registered disk will allocate one device handler data structure to maintain the important properties and attributes of this disk and the caches for directory, FAT, and data sectors. One additional sector buffer is allocated to handle the MBR and boot sector. This buffer is also used as a global temporary buffer when processing long file names in some APIs, to minimize stack size.

The cache is separated into 3 parts: directory, file allocation table, and file data. The user can specify the size for each, in fcfg.h:

```
SFS_DIR_CACHE_SIZE
SFS_FAT_CACHE_SIZE
SFS_DATA_CACHE_SIZE
```

smxFS pre-v1.36 allocated data cache in cluster-sized blocks. It was changed to sectors to support minimal RAM SoCs since a cluster could be up to 32KB, which might be more than the amount of available RAM. Also, this gives better control of RAM usage since cluster size can vary significantly but sector size is usually the same (512 bytes). If your system supports removable media, there is no guarantee what cluster-size media your user will plug in. For example, the Windows format utility has a switch to allow specifying the cluster size, and disks can come preformatted to any cluster size. Now, if two disks have the same sector size, the RAM usage will be the same, regardless of disk size, cluster size, or FAT type.

smxFS v1.39 changed the cache size to the number of bytes instead of the number of sector because we found some disks are using 2KB sector size. It would use 4 times as much RAM compared with the 512 bytes sector size disk. We also added a parameter to the Block Device Interface to allow the device driver to overwrite the default settings for the cache size. For example, the RAM disk only needs the minimum one-sector cache but a USB flash disk may need 16 or 32 sectors of cache to improve performance. Our recommendation is to set the cache to at least 2048 byte so any kind of disk can be supported.

If you will open/list a lot of small files simultaneously, increase the Dir and FAT cache size to improve performance. But if you only open one file at a time and need high performance, increasing the FAT and Data cache sizes is better. If you are using large files, such as for audio or video, you should increase the FAT cache size even more.

When the user opens a file, some additional memory is also allocated to maintain information about it, such as the current read/write pointer. A file-level cache is also created to cache one sector of data to improve the performance of small data access.

The current working directory array is allocated statically. The size is specified in fcfg.h. It is only allocated when SFS\_CWD\_SUPPORT is set to "1".

sfs\_findfirst()/sfs\_findnext()/sfs\_findclose() will allocate an additional small amount of RAM, less than SFS\_PATHFILENAME\_LEN, for wildcard find functions (even if there are no wildcards in the string passed). sfs\_findfirst() allocates the memory and sfs\_findclose() frees it.

sfs\_chkdsk() also needs extra memory to check the disk's entire directory and FAT table. It allocates a memory flag buffer to record which clusters are used by a file and to check if there are any cross-linked clusters. The total size of this buffer is  $2 * \text{TotalClusterNumber} / 8$ . This value depends upon the total disk size and the cluster size to which it is formatted. For example, a 1GB flash disk formatted by FAT32 has 246776 4KB clusters, so the total required flag buffer is  $2 * 246776 / 8 = 61694$  bytes. If the same flash disk is formatted by FAT16, then the cluster size is the same so the flag buffer size is also the same. sfs\_chkdsk() also uses a recursive function to check subdirectories. Each recursive function call needs 24 bytes of stack so we recommend the directory depth should be less than 20.

## 3.8 Reentrancy Protection

Each API call is protected by a mutex or semaphore for each device. If one task is accessing the device, other tasks must wait until the API function completes. This is the purpose of the SFS\_API\_ENTER() and SFS\_API\_EXIT() macros.

## 3.9 Media Change and Mounting

Media change detection and mounting is automatic. All smxFS API functions call CheckMedia() to see if media has been inserted or if a media change occurred.

smxFS has no callback function from the driver to indicate new media is plugged in or removed. You need to create a task in your application to monitor media changes if your media is removable, such as USB or MMC/SD card. smxUSBH has a callback function to notify for media changes so you can use that to register/unregister the disk driver for smxFS. MMC/SD driver does not have a callback function so you need to use the following code to monitor media changes:

```

sfs_devreg(sfs_GetMMCSDBInterface(), SD_DISK);
int SDOldStatus = sfs_devstatus(SD_DISK);
int SDCurrentStatus;
while(1)
{
    SDCurrentStatus = sfs_devstatus(SD_DISK);
    if(SDOldStatus != SDCurrentStatus)
    {
        printf("SD Status changed");
        SDOldStatus = SDCurrentStatus;
    }
    smx_DelayMsec (1000);
}

```

A diagram showing the media change checking and mounting process is shown in section Block Device Interface in the smxBase User's Guide, where it discusses the IOCTL commands.

### 3.10 Multiple Drives / Sockets and Partitions

smxFS supports multiple drives/sockets, but this depends upon the device driver. Currently the RAM disk, and USB flash disk drivers support it. The RAM disk driver serves as an example of how to do it. The idea is to create a separate driver interface structure for each drive and a thin layer of functions that calls the main driver functions passing the relative drive ID to it. For example, RAM0SectorRead() and RAM1SectorRead() call RAMSectorRead() passing 0 or 1 for the drive ID, respectively. The ID is relative to the first of its kind, not the system-wide drive ID. Then call sfs\_devreg() for each drive. For example:

```

sfs_devreg(sfs_GetRAM0Interface(), 0);
sfs_devreg(sfs_GetRAM1Interface(), 1);

```

Do this in sfs\_init() in smxmods.c.

Since smxFS supports FAT32, there is not much need for partitions. However, if you want to use multiple partitions, this can be easily supported. In the case for SBD\_IOCTL\_GETDEVINFO in the driver's IOCTL() function, specify the partition number for each relative drive ID, like this:

```

if (iID == 0)
    pDeviceInfo->wPartition = 0;
else if (iID == 1)
    pDeviceInfo->wPartition = 1;

```

Then call sfs\_devreg() to assign a different drive ID to each, as shown above. When formatting media, sfs\_partition() can be used to create multiple partitions on the disk.

Starting with v2.10, smxFS supports extended and logical partitions. (An extended partition contains logical partitions.) However, note that smxFS can only create primary partitions, but it can use extended and logical partitions that were already created by another OS or utility. sfs\_partition() can create up to four primary partitions, which should be plenty for most embedded systems.

A field has been added to the SBD\_DEVINFO data structure to indicate which logical disk you want to use. For example, if you want to mount the first logical partition (disk), set wLogicalPartition to 0 in the case for SBD\_IOCTL\_GETDEVINFO in the driver's IOCTL() function. For example, if your disk has one primary partition and one extended partition and two logical disks in that extended partition, you may need to implement your IOCTL() interface SBD\_IOCTL\_GETDEVINFO case as:

```

if (iID == 0)
{
    pDeviceInfo->wPartition = 0;
}
else if (iID == 1)
{
    pDeviceInfo->wPartition = 1;
    pDeviceInfo->wLogicalPartition = 0;
}
else if (iID == 2)
{
    pDeviceInfo->wPartition = 1;
    pDeviceInfo->wLogicalPartition = 1;
}

```

In the above example, disk 0 (A:) is a primary partition, disk 1 (B:) is the first logical disk on the extended partition, and disk 2 (C:) is the second logical disk of the extended partition.

wLogicalPartition is **ignored** if the partition is a primary partition. smxFS only checks wLogicalPartition for an extended partition.

If you are using smxFS v2.09 or an earlier version, only the primary partition is supported.

### 3.11 Power Fail Safety

The DOS/Windows FAT filesystem is inherently not power fail safe. The disk data structures (FAT and directories) and file data cannot be modified atomically. For example, the FAT chain for a file can span multiple sectors, and it is possible only one sector was written before power failed. Many clusters of the file may then be missing and the file not terminated. Similarly, directory entries may not have matching size information.

Also, smxFS caches data for efficiency, and the caches can contain various sectors of the FAT, directories, and data. Calling sfs\_fflush() to flush cache data will reduce the problem. Another possibility is to make the caches each only 1 block each, but that hurts performance and cannot totally solve the problem.

One solution, which smxFS does not currently offer, is journaling. In such a system, information about the next operation is written to another area of the disk indicating the operations about to be done. When the operation finishes, this journaling information is cleared. If a power fail occurs during the operation, the recovery code consults the journal and performs the pending operations or undoes the partial operations and discards the pending ones.

In addition to these issues with the high-level filesystem, there may be issues at the driver level for flash media. Our NAND and NOR drivers are power fail safe; they maintain a consistent state in their internal data structures. However, other flash media may not be. For example the SD cards each have an internal controller and driver, and the SD specification does not require it, so it is possible that some cards are power fail safe and others are not. (We don't know of any examples of this, but it is possible.)

If your application requires a high level of power fail safety, you should consider using smxFLog with smxFS, or use smxFFS.

Because damage can occur, smxFS provides sfs\_chkdisk() to allow detecting and fixing many possible file system errors. It also can use a flag to detect whether the file system was cleanly shut down or not. These are discussed in the following sections.

## 3.12 Check Disk

smxFS provides a disk check/fix function `sfs_chkdsk()` to allow you to do some basic checks of file system consistency. This function can also try to fix problem if you pass flags to the `iFixFlag` parameter. Not all problems it finds can it fix, so if you are using a removable disk, such as USB flash disk, MMC/SD, or CompactFlash, we recommend you only use this function to check if there are any problems on the disk, and let Windows fix the problem. That is, pass 0 for `iFixFlag`.

`sfs_chkdsk()` allows passing a pointer to a buffer to hold results of the operation. These are written in text format, similar to the results of a DOS/Windows `chkdsk` utility. The idea is that a person could possibly fix the problems if there is some type of remote management console implemented in your application. Examples of the output are shown at the end of this section.

`sfs_chkdsk()` will check the following fields of all file entries in the root and all subdirectories and the FAT table:

- If this disk is still used by some application (there is any opened file) then it will return `SFS_CHKERR_STILL_IN_USE`.
- This function will try to allocate the FAT flag buffer first, according to the total number of clusters. If there is not enough memory, it will report `SFS_CHKERR_OUT_OF_MEM`.
- When a file entry is marked as `FreeBelow`, the following file entries should all be `FreeBelow` or `FreeThis`. Otherwise, it will report `SFS_CHKERR_INV_FILEENTRY`. When `iFixFlag` is set to `SFS_FIX_AUTO`, it will also erase all the invalid file entries.
- When a file entry is marked as a long file name, the first long file name entry should have an end flag. Otherwise, it will report `SFS_CHKERR_INV_FILEENTRY`. When `iFixFlag` is set to `SFS_FIX_AUTO`, it will also erase that invalid long file name file entry.
- For long file name directory entries, the order of each long file name entry and checksum should be correct. Otherwise it will report `SFS_CHKERR_INV_FILEENTRY`. It will do nothing to fix this problem even if `iFixFlag` is set to `SFS_FIX_AUTO` but we will still output the invalid long file name and short 8.3 name to the results buffer, so a person could scan through it and decide what operations to do to correct it, via a remote console, for example, if implemented by the application. If the file will be kept, it should be renamed by passing the 8.3 name as the source name. The bad directory entries will be freed and new ones created for the new name. The purpose of printing the long name is to help identify the file, even if it is out of order.
- If a file name entry contains any invalid characters, it will report `SFS_CHKERR_INV_FILENAME`. When `iFixFlag` is set to `SFS_FIX_AUTO`, it will also change the invalid characters to an underscore.
- For each file the timestamp should be valid. That is, the month should be 1 to 12, day should be 1 to 31, hour should be 0 to 23, and minute and second should be 0 to 59. It will not check if year is valid. Otherwise it will report `SFS_CHKERR_INV_FILETIME`. When `iFixFlag` is set to `SFS_FIX_AUTO`, it will change the timestamp to the current date/time.
- For each file, the first cluster entry should be valid. It should be within the range of the total number of clusters. Otherwise it will report `SFS_CHKERR_INV_FIRSTCLUS`. When `iFixFlag` is set to `SFS_FIX_AUTO`, it will change the first cluster entry to 0, which means the file length is 0.
- For each file, this function will scan the whole cluster chain in the FAT table, and check that all cluster numbers are within the range of the total number of clusters. If not, it will report

SFS\_CHKERR\_INV\_FATNODE. When iFixFlag is set to SFS\_FIX\_AUTO, it will change that cluster entry to EOC which means the file will be truncated.

- For each file, this function will scan the whole cluster chain in the FAT table, and check that every cluster is not used by another file or directory, that is, that the file is not cross-linked with another file. If so it will report SFS\_CHKERR\_FAT\_CROSSLINK. When iFixFlag is set to SFS\_FIX\_COPY\_CROSS\_CLUS, the files will be split by copying the shared clusters into a new set of free clusters and linked to one of the files. Then, both files will have duplicate data beyond the cross point, but now each will have its own set of clusters and unique FAT chain. If a data buffer is passed to sfs\_chkdsk(), it will list all files that are cross-linked. This way you are warned about which files you must suspect of having wrong data. If there is a remote link or other way to get the data from the system, the files could be inspected, fixed, and copied back to the system, and the original damaged ones could be deleted.
- For each file, this function will also check the file size in the directory entry. It should match the length according to the linked FAT nodes. Otherwise it will report SFS\_CHKERR\_INVALID\_FILELEN. When iFixFlag is set to "1", the file length in the directory entry will be changed to the linked FAT node size.
- For each directory, this function will also check "." and ".." entries to make sure the directory flag is set, time stamp is valid, and first cluster and file length are correct. Otherwise, it will report SFS\_CHKERR\_INV\_DOTDIR. When iFixFlag is set to SFS\_FIX\_AUTO, wrong fields will be fixed to the correct value.
- After all the file entries are checked, this function will scan the whole FAT flag table to make sure there are no orphan FAT nodes. Otherwise, it will report SFS\_CHKERR\_FAT\_LOSTCHAIN. When iFixFlag is set to SFS\_FIX\_AUTO, the lost FAT nodes will be converted to free nodes. When iFixFlag is set to SFS\_FIX\_SAVE\_LOST\_CHAIN, the lost FAT nodes will be copied to the files located in the \FOUND.00x\FILE000x.CHK.

The following shows some examples of the results that might be reported in the results buffer passed to sfs\_chkdsk():

```
smxFS Check Disk Results for Disk A:  
Lost chains 0x13 - 0x1A00 converted to file A:\FOUND.000\FILE0000.CHK  
Lost chains 0x1A01 - 0x1A01 converted to file A:\FOUND.000\FILE0001.CHK
```

```
smxFS Check Disk Results for Disk A:  
Cross-Linked Files:  
A:\Dir1\FileA.txt offset at 0x0007000  
A:\Dir2\Dir3\FileB.txt offset at 0x00928000  
A:\FileC.txt offset at 0x00071000  
A:\Dir3 offset at 0x00001000
```

```
smxFS Check Disk Results for Disk A:  
sfs_chkdsk() did not find any problems
```

For cross-linked files, the number on each line is the byte offset into the file where the first crossed cluster was encountered. Everything up to that point should be ok, but everything following it is suspect. The last line showing just Dir3 means that the directory itself is cross-linked. Directories are files (except the root directory on FAT12/16).

### 3.13 Clean Shutdown Flag

**Note: We have not fully studied/tested this option to ensure it is reliable for power loss at every point in the code.**

If you are using FAT16 or FAT32, according to the Microsoft FAT white paper, the file system may use the highest bit of the FAT[1] entry as a dirty volume flag.

For FAT16:

```
ClnShutBitMask = 0x8000;
```

For FAT32:

```
ClnShutBitMask = 0x08000000;
```

Bit ClnShutBitMask – If bit is 1, volume is “clean”.

If bit is 0, volume is “dirty”. This indicates that the file system driver did not dismount the volume properly the last time it had the volume mounted. It would be a good idea to run a chkdsk/scandisk disk repair utility on it, because it may be damaged.

If you set `SFS_CLN_SHUTDOWN_SUPPORT` in `ucfg.h` to 1, then `smxFS` will mark the file system as dirty (0) when one or more files are open for writing. `smxFS` will set this flag to clean (1) after all such files are closed. If the application aborts or a power fail occurs when some files are still open for writing, the next time `sfs_devstatus()` runs, it will return the status `SFS_DEVICE_NOT_SHUT_DOWN`. In this case should call `sfs_chkdsk()` to check and fix the disk.

Enabling this feature will decrease performance of the file system because it generates extra overhead to check and mark the Clean Shutdown Flag. For flash media, it will also cause additional wear, since the first FAT sector has to be moved every time the flag is changed to 1.

### 3.14 Dual FATs

Most disks have two FATs. The second FAT is intended to be a backup in case the first FAT has any problem. But Windows does not seem to use second FAT that way, based on our testing. We modified the FAT contents of a disk and then ran the Windows `chkdsk` utility to check and fix it. Here are our results:

1. We damaged FAT1 totally and left FAT2 alone. After running `chkdsk`, all the wrong FAT nodes in FAT1 were treated as lost FAT chains and emptied. FAT2 was modified to the same value of FAT1, so Windows did not use FAT2 to correct FAT1.
2. We damaged FAT2 totally and left FAT1 alone. `chkdsk` did not even report any error.

Thus, we believe Windows does not even check whether FAT1 and FAT2 are the same, and it does not use one to repair the other.

`SFS_2NDFAT_SUPPORT` is used to indicate whether `smxFS` should also write FAT data to the second FAT. This feature is disabled by default to improve performance. Currently, `smxFS` does not make use of the copy, so this feature is not useful. In the future we may improve `sfs_chkdsk()` to use it to help repair the disk.

### 3.15 Alternate Filesystem Access

It would be unusual to have another filesystem in your target besides smxFS that can access the same media smxFS does. However, one case where this occurs is if you are also using smxUSB and the mass storage driver. This technique allows plugging your target into a USB host such as Windows, and then operating on your target's media like a USB flash disk. In this case, Windows has its own filesystem that has its own view of the disk. smxUSB only uses a block device driver to access sectors. The problem is that with two filesystems accessing the disk, errors will be introduced because neither is aware of changes the other is making. For example, one may have part of the FAT modified in cache but not yet written to the media. The other may make changes to the same sector of the FAT.

The solution is to permit access to only one at a time and each unmounts before letting the other access the media. Before plugging in the USB cable, you should **close all open files and then call `sfs_devunreg()`** to unmount the device. When done with USB access, **`sfs_devreg()`** should be called again to restore it. Before unplugging the USB cable, the user should do Safely Remove Hardware to shut down the disk just like he would before unplugging a USB flash disk or you can force smxUSB to shut down the mass storage device.

The following code shows an example of how to share a RAM disk between smxFS and smxUSB.

```
sfs_init();
sud_Initialize();

/* Now register RAM disk so smxFS can use it. But smxUSB cannot access mass storage device at this time. */
sfs_devreg(sfs_GetRAM0Interface(), 0);

/* Create a sample file */
fp = sfs_fopen("A:\\fscreate.txt", "wb");
if(fp)
    sfs_fclose(fp);

/* Do other smxFS operations here. */

/* Shut down the disk for smxFS. */
sf_devunreg(0);

/* Now register the device driver with smxUSB. */
sud_MSRegisterDisk(sfs_GetRAM0Interface(),0);

/* Tell the user to plug in the USB cable so USB host can access the disk */

/* Do other smxUSB operations here by USB Host through USB link. */

/* Force shutdown of the USD mass storage device. Then the USB host will not access it. */
sud_MSRegisterDisk(NULL,0);

/* Register it with smxFS again. If the USB host changed anything on the disk you will see it now. */
sfs_devreg(sfs_GetRAM0Interface(), 0);
```

This same technique should be applied if, for some reason, another filesystem is able to access the media (while mounted in your target).

## 3.16 Safety Checks

smxFS does run-time safety checks if `SFS_SAFETY_CHECKS` is set to 1 in `fcfg.h`. When enabled, smxFS will do additional checks to ensure the basic file system structure on the media is ok and that parameters passed by the application are valid. It may not catch every problem, but it can provide more protection for your system. Of course, enabling safety checks may reduce the file system performance and increase code size. You may want to enable it during development and disable it after your system is fully tested. These are the things it checks:

1. File name length is valid and does not overflow buffers used to hold it. The maximum length of long file names is set by `SFS_LONGFILENAME_LEN` (255) in `fcfg.h`. The maximum length of the path plus file name is specified by `SFS_PATHFILENAME_LEN` (260) in `fcfg.h`.
2. The file handle is valid whenever passed as a parameter, such as in calls to `fread()/fwrite()/fseek()`.
3. Cache size set in `fcfg.h` is an exact multiple of sector size for that disk.
4. The BPB structure is valid. For example the sectors per cluster field must be power of 2, and number of FATs must be 1 or 2. This is checked when the disk is mounted.
5. Generated BPB is valid when the disk is formatted.
6. Each FAT node is valid when it is read from the disk.
7. Each directory entry is valid when it is read from the disk. It will also check if the `DateTime` is valid.
8. No NULL pointer is passed to smxFS as parameter to `fwrite()/fread()`.
9. Internal open file handle list is valid whenever a file handle is passed as a parameter.

## 3.17 Fast Cluster Allocation

After repeatedly creating and deleting files on a disk, the FAT may get fragmented, which slows down accesses to files, since many different sectors of the FAT may need to be read into the cache. One thing smxFS does to reduce search time is it keeps a pointer to the last allocated FAT node for each disk. When a new cluster is needed, smxFS will only search the FAT from the last allocated position instead of searching it from the beginning. The following sections discuss other ways to improve cluster allocation speed.

### 3.17.1 Preallocated Contiguous Files

Especially when streaming data, it can be useful to preallocate a large file, so it is not necessary to allocate clusters while writing data. To guarantee it is contiguous, create it soon after you format the disk. Use `fseek(fp, size, SFS_SEEK_SET)` function to preallocate the as many as clusters as you need, without the need to actually write data to it. Then write data to the file as usual.

### 3.17.2 Free Cluster Cache

The free cluster cache is an optional feature that creates a separate cache to store a list of free data clusters, to improve performance for large file writes. This feature can only improve the performance under some specific circumstance so it is disabled by default. To enable it, set `SFS_FREECLUS_SUPPORT` to 1 in `fcfg.h`. The following gives an example showing the need for this feature.

If your system will only store a few very large files, such as the recorded video files, and you delete one of those files, it will generate a big gap within the FAT of free cluster nodes. For example:

You start with a 1GB flash disk and create file1 that is 100MB then file2 is 500MB. Then you delete file1. In this case, the FAT table will have 100MB free clusters, 500MB used clusters and then 400MB free cluster.

Later you want to create file3, which may be 300MB. The first 100MB should be OK, smxFS will find the free cluster pretty easily at the beginning of FAT, but to find additional free clusters, it has to scan through the whole 500MB of used clusters to find the next free one. If the cluster size is 4KB, 500MB needs 128,000 FAT16 nodes whose size is 256KB (512 sectors). That is, smxFS needs to read and scan 256KB of FAT data to skip all the used clusters. This may take a couple seconds, so the file system may take a few seconds to complete that `sfs_fwrite()` operation. For streaming video, that delay may be too long and cause the video buffer to overflow.

The solution we implemented in smxFS is to store a list of free clusters in a special cache whose size is configurable and should be set appropriately for the application. When the disk is mounted, smxFS fills the cache. Then during normal operation, whenever smxFS attempts to allocate some free clusters, it will try to find more if the free cluster number in the cache is less than `SFS_FREECLUS_TRIGGER`. Each time it will scan at most `SFS_FREECLUS_SCAN_CACHE_SIZE` FAT area. The application can call `sfs_scanfreeclus()` to force it to cache more free clusters in the system's idle time.

In smxFS, we assume most free clusters are contiguous. This is true if you only store a few big files on that flash disk. So it stores the free cluster list in compressed format: start Index and number of contiguous clusters.

This feature is disabled for FAT12 disks because FAT12 only has a few sectors of FAT, and using this cache decreases performance.

### 3.18 Accessing a File with Multiple File Handles

We recommend that you access a file by only one file handle. If you need to access a file by multiple tasks, you can share the same file handle among those tasks using a global variable. This saves resources and provides the best performance.

However, if you really do need to have multiple file handles for a file, for example to have separate file pointers for a task that is reading while another is writing to it, this is supported starting with smxFS v2.10. (Previous versions of smxFS were only able to open a file multiple times for reading, and there was only one file pointer.)

You must be careful if you use this feature to write the same open file from multiple tasks! If the tasks attempt to write (update) data to the same sector of a file, the first `sfs_fwrite()` data will be overwritten by the second. For example, if the first `sfs_fwrite()` wrote all 0xFF to a sector and the second wrote all 0x00, the final result of those two `sfs_fwrite()` operations will set the data of that sector to 0x00. You must write your application carefully to avoid this.

### 3.19 Pass Through Mode

smxFS has an internal data buffer. This buffer can improve performance of small data accesses, but extra memory copies may cause extra overhead for high speed streaming data access. smxFS provides an API to enable/disable pass through mode, which will directly pass the application buffer to the disk driver to avoid extra memory copies of the data.

int **sfs\_setpassthrough**(FILEHANDLE filehandle, BOOLEAN bEnable);

To use pass through mode:

1. The current file pointer must be at a cluster boundary.
2. For each subsequent file read/write operation, the data buffer size must be one or multiple clusters. If not, pass through mode will be automatically disabled.
3. The application must ensure the buffer can be used directly by the disk driver. Some drivers may need non-cacheable memory if the processor's data cache is enabled.

## 4. File System API

The smxFS API follows the standard C library file i/o API. Any limitations or differences from the standard are noted in the call descriptions below. The `sfs_` prefix gives these their own namespace, and makes it easy to search for calls to this smxFS. A few non-standard calls were added for additional capabilities such as initializing the filesystem, registering device drivers, and indicating free space on the media.

In order to minimize code space, some of the less-common functions can be omitted by setting `SFS_FULL_FEATURES_SUPPORT` to 0.

Notes about using the API:

1. In paths, use two backslashes `\\` instead of one. This is necessary for C because a single backslash is used to quote the next character or to specify special characters (e.g. `\n` is newline; `\0` is NUL).
2. Drive letters can be specified upper and lower case.
3. File and path names: These are only case-sensitive when creating a file. Case does not matter when operating on an existing file. See section 3.2 File Names for more information.

### 4.1 API Data Types

These are defined in **fapi.h** unless otherwise noted.

<code>FILEHANDLE</code>	Pointer to a <code>FILESTRUCT</code> structure which contains information about an open file, such as its current file pointer. A file handle uniquely identifies an open file, and is passed as a parameter to all API calls to operate on the file. The file handle is released when the file is closed.
<code>FILEINFO</code>	Structure containing various information about a file found with <code>sfs_findfirst()</code> or <code>sfs_findnext()</code> .
<code>FORMATINFO</code>	Structure containing various information about formatting a volume with <code>sfs_format()</code> .
<code>PARTITIONINFO</code>	Structure containing various information about partitioning a disk with <code>sfs_partition()</code> .
<code>SBD_IF</code>	Pointer to a structure of pointers to the driver interface functions.
<code>u32, u16, etc</code>	Unsigned integer types of the size (bits) indicated.

## 4.2 API Summary

Calls marked + are the only ones included in the Lite configuration.

### Basic API Calls

```
+ int          sfs_init(void)
+ int          sfs_exit(void)

+ int          sfs_devreg(const SBD_IF *dev_if, uint nID)
+ int          sfs_devunreg(uint nID)
+ int          sfs_devstatus(uint nID)
+ const SBD_IF * sfs_getdev(uint nID)
+ unsigned long sfs_freekb(uint nID)
+ unsigned long sfs_totalkb(uint nID)
+ int          sfs_ioctl(uint nID, uint command, void * par)
+ int          sfs_writeprotect(uint nID)
+ int          sfs_flushall(uint nID)
+ int          sfs_getlasterror(uint nID);

+ FILEHANDLE   sfs_fopen(const char *filename, const char *mode)
+ int          sfs_fclose(FILEHANDLE filehandle)
+ size_t       sfs_fread(void * buf, size_t size, size_t items, FILEHANDLE filehandle)
+ size_t       sfs_fwrite(void * buf, size_t size, size_t items, FILEHANDLE filehandle)
+ int          sfs_fseek(FILEHANDLE filehandle, long lOffset, int nMethod)
+ int          sfs_fflush(FILEHANDLE filehandle)
+ int          sfs_feof(FILEHANDLE filehandle)
+ void         sfs_rewind(FILEHANDLE filehandle)
+ long         sfs_ftell(FILEHANDLE filehandle)

+ int          sfs_fdelete(const char * filename)
+ unsigned long sfs_filelength(const char *filename)
+ int          sfs_findfile(const char *filename)

+ int          sfs_mkdir(const char *path)
+ int          sfs_rmdir(const char *path)

+ int          sfs_setpassthrough(FILEHANDLE filehandle, BOOLEAN bEnable)
```

### Extended API Calls

```
int          sfs_chkdisk(uint nID, uint iFixFlag, char *pResultBuf, uint iBufLen)

int          sfs_chdir(const char *path)
int          sfs_setcwd(const char *path)
char *       sfs_getcwd(char * buffer, int maxlen)

+ int          sfs_partition(uint nID, PARTITIONINFO * partitioninfo)
+ int          sfs_format(uint nID, FORMATINFO * formatinfo)
```

int	<b>sfs_getvolname</b> (uint nID, char * name)
int	<b>sfs_setvolname</b> (uint nID, const char * name)
int	<b>sfs_getprop</b> (const char * filename, FILEINFO* fileinfo)
int	<b>sfs_setprop</b> (const char * filename, FILEINFO* fileinfo, uint flag)
int	<b>sfs_chmod</b> (const char * filename, uint pmode)
int	<b>sfs_stat</b> (const char * filename, FILEINFO* fileinfo)
int	<b>sfs_timestamp</b> (const char * filename, DATETIME* datetime)
int	<b>sfs_rename</b> (const char * oldname, const char * newname)
int	<b>sfs_move</b> (const char * oldname, const char * newname)
int	<b>sfs_copy</b> (const char * src, char * dest)
int	<b>sfs_delmany</b> (const char * filelist, int num)
int	<b>sfs_findfirst</b> (const char * filespec, FILEINFO* fileinfo)
int	<b>sfs_findnext</b> (int id, FILEINFO* fileinfo)
int	<b>sfs_findclose</b> (FILEINFO* fileinfo)
FILEHANDLE	<b>sfs_fastfopen</b> (FILEINFO* fileinfo, BOOLEAN bReadOnly)
void	<b>sfs_ftruncate</b> (FILEHANDLE filehandle)

## Notes

1. Configuration settings in fcfg.h allow enabling/disabling portions of the API.
2. `sfs_mkdir()` and `sfs_rmdir()` are enabled if `SFS_MKDIR_SUPPORT` is “1” or `SFS_FULL_FEATURES_SUPPORT` is “1”.
3. `sfs_partition()` and `sfs_format()` are enabled if `SFS_FORMAT_SUPPORT` is “1” or `SFS_FULL_FEATURES_SUPPORT` is “1”.

## 4.3 API Reference

*Note: This section is alphabetized. For a functional organization, see the API Summary above.*

**Extended API calls are marked “[xxx\_SUPPORT]” below** (on the first line of each call description). These are enabled by setting SFS\_XXX\_SUPPORT to 1 in fcfg.h.

int           **sfs\_chdir** (const char \*path)   [CWD\_SUPPORT]

Alias for sfs\_setcwd(). See its call description below.

int           **sfs\_chkdsk** (uint nID, uint iFixFlag, char \*pResultBuf, uint iBufLen)   [CHKDSK\_SUPPORT]

**Summary**    Check and/or fix problems found in the file system.

**Details**    Like the DOS/Windows chkdsk utility, this function checks all directory entries and the FAT table for the partition to determine if the information is valid. If iFixFlag is non-zero, it will also try to fix the problems automatically, if possible. The function needs extra stack and heap so make sure your system has enough RAM to support it. See section 3.7 Memory Management for discussion.

**See section 3.12 Check Disk for full details about this function.**

You must close all the files on the disk prior to calling this function. Otherwise it will do nothing but return SFS\_CHKERR\_STILL\_IN\_USE.

You must call sfs\_devstatus() first to check if the disk is mounted. (Only FAT12/16/32 formats are supported by smxFS. It does not support NTFS or exFAT.) Calling sfs\_chkdsk() on an unmounted disk will cause it to return SFS\_ERR\_DISK\_NOT\_MOUNTED.

Note that this function uses some global variables (to reduce stack usage due to recursion), so even if it is used in read-only mode, it is still non-reentrant and is protected by a mutex/semaphore like the rest of the API.

**Pars**

nID	The device ID number of the device driver. You can specify any ID that is less than MAX_DEV_NUM. The macro SFS_FIRST_DRIVE plus this device ID is the drive letter.
iFixFlag	Flag if this function will also try to fix the found problem. Not all problems that are found can be fixed by this API. Valid flags include combinations of the following values:

SFS\_FIX\_AUTO

Automatically fix problems according to the description in section 3.12 Check Disk.

SFS\_FIX\_SAVE\_LOST\_CHAIN

Convert lost clusters to files \FOUND.00x\FILE000x.CHK.

SFS\_FIX\_COPY\_CROSS\_CLUS

Copy all shared clusters of a file so that beyond the cross point, the files have the same data but in separate cluster chains. See section 3.12 Check Disk for more discussion.

These flags are independent. For example, if only

SFS\_FIX\_SAVE\_LOST\_CHAIN is passed, the only problem that will be fixed is saving lost chains to files. Other problems will only be reported. Typically, you will use SFS\_FIX\_AUTO and possibly the other flags.

**pResultBuf** Pointer to a text buffer to hold messages with the results of the check disk operation. See section 3.12 Check Disk for an example of the results buffer.

**iBufLen** The length of this result buffer.

**Returns** 0 No problems were found in the file system  
!= 0 Combination of the following flags means there are some errors in the file system. See section 3.12 Check Disk for detailed explanations of each error.

SFS\_CHKERR\_NO\_ERROR

SFS\_CHKERR\_INV\_FILENAME

SFS\_CHKERR\_INV\_FILETIME

SFS\_CHKERR\_INV\_FILELEN

SFS\_CHKERR\_INV\_FATNODE

SFS\_CHKERR\_INV\_FILEENTRY

SFS\_CHKERR\_INV\_DOTDIR

SFS\_CHKERR\_INV\_FIRSTCLUS

SFS\_CHKERR\_FAT\_CROSSLINK

SFS\_CHKERR\_FAT\_LOSTCHAIN

SFS\_CHKERR\_BUF\_OVERFLOW

SFS\_CHKERR\_STILL\_IN\_USE

SFS\_CHKERR\_OUT\_OF\_MEM

SFS\_ERR\_DISK\_NOT\_MOUNTED if the disk is not mounted yet (not formatted or has an unsupported file format such as NTFS or exFAT)

## Example

Check disk at the beginning of the application.

```
void appl_init()
{
    int result;
    sfs_init();
    sfs_devreg(sfs_GetRAM0Interface(), 0);

    /* Check only. Don't fix. */
    result = sfs_chkdisk(0, 0, NULL, 0);
    if(result == SFS_CHKERR_NO_ERROR)
        printf("File system has no problems");
    if(result & SFS_CHKERR_INV_FILENAME)
        printf("File system contains invalid file name");
    if(result & SFS_CHKERR_INV_FILETIME)
        printf("File system contains invalid file time");
    if(result & SFS_CHKERR_INV_FILELEN)
        printf("File system contains invalid file length");
    if(result & SFS_CHKERR_INV_FATNODE)
        printf("File system contains invalid FAT Node");
    if(result & SFS_CHKERR_INV_FILEENTRY)
        printf("File system contains invalid file entry");
    if(result & SFS_CHKERR_INV_DOTDIR)
        printf("File system contains invalid directory entry");
    if(result & SFS_CHKERR_INV_FIRSTCLUS)
        printf("File system contains invalid first cluster");
    if(result & SFS_CHKERR_FAT_CROSSLINK)
        printf("File system contains cross-linked files (FAT chains)");
    if(result & SFS_CHKERR_FAT_LOSTCHAIN):
        printf("File system contains lost FAT chain");
    if(result & SFS_CHKERR_BUF_OVERFLOW):
        printf("Result Buffer overflowed");
    if(result & SFS_CHKERR_STILL_IN_USE):
        printf("Some files are still open");
    if(result & SFS_CHKERR_OUT_OF_MEM)
        printf("Not enough memory to run sfs_chkdisk()");
}
```

Check disk when you are running your application.

```
void app_task_main()
{
    FILEHANDLE fp;
    int result;
    fp = sfs_fopen("A:\\Test.bin", "r");
    if(fp)
    {
        result = sfs_chkdisk(0, 0, NULL, 0);
        if(result & SFS_CHKERR_STILL_IN_USE)
        {
            printf("Check disk failed because a file is still open.");
            sfs_fclose(fp);
        }
    }

    /* Check and Auto fix. */
    result = sfs_chkdisk(0, SFS_FIX_AUTO, NULL, 0);
    if(result == SFS_CHKERR_NO_ERROR)
        printf("File system has no problems");
    if(result & SFS_CHKERR_INV_FILENAME)
        printf("File system contains invalid file name");
    if(result & SFS_CHKERR_INV_FILETIME)
        printf("File system contains invalid file time");
    if(result & SFS_CHKERR_INV_FILELEN)
        printf("File system contains invalid file length");
    if(result & SFS_CHKERR_INV_FATNODE)
        printf("File system contains invalid FAT Node");
    if(result & SFS_CHKERR_INV_FILEENTRY)
        printf("File system contains invalid file entry");
    if(result & SFS_CHKERR_INV_DOTDIR)
        printf("File system contains invalid directory entry");
    if(result & SFS_CHKERR_INV_FIRSTCLUS)
        printf("File system contains invalid first cluster");
    if(result & SFS_CHKERR_FAT_CROSSLINK)
        printf("File system contains cross-linked files (FAT chains)");
    if(result & SFS_CHKERR_FAT_LOSTCHAIN)
        printf("File system contains lost FAT chain");
    if(result & SFS_CHKERR_BUF_OVERFLOW)
        printf("Result Buffer overflowed");
    if(result & SFS_CHKERR_STILL_IN_USE)
        printf("Some files are still open");
    if(result & SFS_CHKERR_OUT_OF_MEM)
        printf("Not enough memory to run sfs_chkdisk()");
}
```

int **sfs\_chmod** (const char \* filename, uint pmode) [PROPERTY\_SUPPORT]

**Summary** Change permission settings of a file or directory.

**Details** Change the permission settings of the file or directory specified by filename to control read and write access to the file.

**Pars** filename The file or or directory whose permission settings you want to change.  
flag New permission(s) which may be OR'd from the following values:  
S\_IWRITE  
S\_IREAD

**Returns** 0 The permission settings have been changed successfully.  
-1 File or directory not found.

**See Also** sfs\_getprop(), sfs\_setprop(), sfs\_stat(), sfs\_timestamp()

**Example**

```
void appl_init()
{
    sfs_init();
    sfs_devreg(sfs_GetRAM0Interface(), 0);
    /* set permission to Read Only */
    sfs_chmod("A:\\test.bin", S_IREAD);
}
```

int **sfs\_copy** (const char \* src, char \* dest) [FULL\_FEATURES\_SUPPORT]

**Summary** Copy one file to another place on the same or different volume.

**Details** Copy one file to another one. It creates a new file and then copy all the data to it. The source file must exist. If the destination file already exists, this function will overwrite it. The parent directory of the destination file must also exist. The function simply calls sfs\_fread() and sfs\_fwrite() to do the file copy. It allocates a buffer to copy the data so using this function requires additional RAM (SFS\_COPY\_BUF\_SZ in fcfg.h).

**Pars** src The source file name.  
dest The destination file name.

**Returns** SB\_PASS File copy succeed.  
SB\_FAIL File copy failed.

**See Also** sfs\_rename(), sfs\_fopen(), sfs\_fwrite(), sfs\_fread(), sfs\_fclose()

**Example**

```
void appl_init()
{
    sfs_init();
    sfs_devreg(sfs_GetRAM0Interface(), 0);
    /* copy file */
    sfs_copy("d:\\src.bin", "d:\\dest.bin");
}
```

int **sfs\_delmany** (const char \* filelist, int num) [FULL\_FEATURES\_SUPPORT]

**Summary** Delete multiple files at once.

**Details** This function deletes many files in one operation. It is much faster than making multiple calls to sfs\_fdelete(). It avoids flushing the directory and FAT table for each file delete operation.

The file list passed in is a string of file names separated by the NUL character. Each file name includes the drive and path. It is possible to delete files from different directories in one operation, but it is more efficient to break the operation into a separate call per directory. Also, note that after the files are deleted, it does free directory entry collection only on the last directory specified in the list. (This means it sets the marker for end of directory on the next entry after the last used entry in the directory.) All files must be on the same volume.

**Pars**

filelist	The list of files to be deleted.
num	The number of file names.

**Returns**

>= 0.	Number of files deleted.
< 0	Device is not mounted.

**See Also** sfs\_fdelete(), sfs\_findfirst(), sfs\_findnext()

### Example

```
void appl_init()
{
    sfs_init();
    sfs_devreg(sfs_GetRAM0Interface(), 0);
    if (SFS_DEVICE_MOUNTED & sfs_devstatus(id))
    {
        buf[0] = SFS_FIRST_DRIVE+id;
        id = sfs_findfirst(buf, &fileinfo);
        while(id != -1)
        {
            pFileName = (char *)&pDataBuf[position];
            strcpy(pFileName, "a:\\sfstest\\");
            strcat(pFileName, (char *)fileinfo.name);
            position += strlen(pFileName) + 1;
            if(position > DATA_SIZE - 100)
                break;
            fileNum++;
            id = sfs_findnext(id, &fileinfo);
        }
        if(fileNum > 0)
        {
            fileNum = sfs_delmany(pDataBuf, fileNum);
        }
    }
}
```

int **sfs\_devreg** (const SBD\_IF \*dev\_if, uint nID)

**Summary** Register a device driver with smxFS.

**Details** You must call this function to actually add a device driver to smxFS. You can register as many drivers as specified by the macro MAX\_DEV\_NUM in fcfg.h. You can call this function at any time after you call sfs\_init() and before you call sfs\_exit(). This function allocates some internal data structures from the heap.

**Pars**

dev_if	The device driver interface structure pointer. See section Block Device Interface in the smxBase User's Guide for the details of the requirement.
nID	The device ID number of the device driver. You can specify any ID which is less than the macro MAX_DEV_NUM. The macro SFS_FIRST_DRIVE plus this device ID is the disk letter.

**Returns**

SB_PASS	The device driver has been registered successfully
SB_FAIL	The device ID is not valid or this ID has been registered by another device driver.

**See Also** sfs\_init(), sfs\_devunreg(), Alternate Filesystem Access in Chapter 3.

**Example**

```
void appl_init()
{
    sfs_init();
    sfs_devreg(sfs_GetRAMInterface(), 0);
    fp = sfs_fopen("d:\\test.bin", "wb");
    sfs_fclose(fp);
}
```

int **sfs\_devstatus** (uint nID)

**Summary** Returns the current status of the device/disk.

**Details** This function returns the status of the device/disk specified by nID.

**Pars**

nID	The device ID that was specified in the call to sfs_devreg().
-----	---

**Returns**

SFS_DEVICE_NOT_FOUND	The device is not inserted.
SFS_DEVICE_MOUNTING	The device is inserted and smxFS is mounting it.
SFS_DEVICE_MOUNTED	Mounting is complete and the device can be used now.
SFS_DEVICE_UNFORMATTED	The device is inserted but smxFS could not find the correct FAT12/16/32 format on it.
SFS_DEVICE_NOT_SHUT_DOWN	The device is formatted but was not cleanly shut down by the application. That is, files that were open for writing were not closed before the application aborted or power was lost. The flag maybe OR'd with SFS_DEVICE_MOUNTED.

**See Also** sfs\_devreg()

**Example**

```
if(SFS_DEVICE_NOT_FOUND == sfs_devstatus(0))
    printf("The disk 0 has not been inserted.");
```

int **sfs\_devunreg** (uint nID)

**Summary** Unregister a registered device driver from smxFS.

**Details** You can call this function to remove a device driver from smxFS. When smxFS is unmounted (by calling sfs\_exit()), this function will be called automatically so normally you do not need to call it explicitly. One exception would be before granting another filesystem access to the same media, such as with smxUSB. See 3.15 Alternate Filesystem Access.

**Pars** nID The device ID that was specified in the call to sfs\_devreg().

**Returns** SB\_PASS The device driver has been removed successfully.  
SB\_FAIL The device ID is not valid or this ID has not been registered.

**See Also** sfs\_exit(), sfs\_devreg(), Alternate Filesystem Access in Chapter 3.

**Example**

```
void appl_exit()
{
    sfs_devunreg(0);
}
```

int **sfs\_exit** (void)

**Summary** Uninitializes the smxFS file system.

**Details** This is the last smxFS API call that should be made at exit. This function un-registers all device drivers.

**Pars** none

**Returns** SB\_PASS Success.  
SB\_FAIL Uninitialization failed.

**See Also** sfs\_init()

**Example**

```
void appl_exit()
{
    sfs_exit();
}
```

FILEHANDLE **sfs\_fastfopen** (FILEINFO\* fileinfo, BOOLEAN bReadOnly) [FINDFIRST\_SUPPORT]

**Summary** Fast open a file after you already have the file information. Normally you get the file information from the sfs\_findfirst()/sfs\_findnext() function calls.

**Details** If your disk has hundreds or thousands of files, and you want to search for some particular file and then open it, you may need to call findfirst()/findnext() to find a file first and then call fopen() to open it. But doing it this way, sfs\_fopen() will need to search (scan) that directory again. This function allows you to use the file information you already get from sfs\_findfirst()/sfs\_findnext() and open the file quickly.

**Pars** fileinfo File information returned by sfs\_findfirst()/sfs\_findnext().  
bReadOnly Indicates whether to open the file read only.

**Returns** file handle Success.  
NULL File not found or other error. Do not pass a NULL handle to other API calls.

**See Also** sfs\_fopen()

#### Example

```
FILEHANDLE fp;
FILEINFO fileinfo;
int id;
id = sfs_findfirst("d:\\*.img", &fileinfo);
while(id != -1)
{
    fp = sfs_fastfopen(&fileinfo, TRUE);
    if(fp)
    {
        sfs_fclose(fp);
    }
    id = sfs_findnext(id, &fileinfo);
}
sfs_findclose(&fileinfo);
```

int **sfs\_fclose** (FILEHANDLE filehandle)

**Summary** Close an open file.

**Details** Closing a file causes all the data to be flushed to the media. All resources allocated by sfs\_fopen() are released. Once the file is closed, the file handle is no longer valid, so do not use it in another API call.

**Pars** filehandle File handle that was returned by sfs\_fopen().

**Returns** SB\_PASS Success.  
SB\_FAIL File cache flush failed or file was already closed.

**See Also** sfs\_fopen()

## Example

```
FILEHANDLE fp;
fp = sfs_fopen("d:\\test.bin", "wb");
if(fp != NULL)
{
    sfs_fwrite(...);
    sfs_fclose(fp);
}
```

int        **sfs\_fdelete** (const char \* filename)

**Summary**    Deletes a file.

**Details**    This function deletes the file indicated by *filename*. If the file does not exist, this function does nothing and returns. If the file is currently open, this function will return an error.

**Pars**        filename    The name of the file to be deleted.

**Returns**    SB\_PASS    Success.  
              SB\_FAIL    File not found, device has been removed, or file open.

**See Also**    sfs\_findfile()

## Example

```
FILEHANDLE fp;
sfs_fdelete("d:\\test.bin");
sfs_fdelete("d:\\test.bin"); // attempting to delete a file that does not exist will not cause any damage.
fp = sfs_fopen("d:\\data.dat", "rb");
sfs_fdelete("d:\\data.dat"); // attempting to delete an open file does nothing; just returns.
sfs_fclose(fp);
```

int        **sfs\_feof** (FILEHANDLE filehandle)

**Summary**    Tests for end-of-file for a file.

**Details**    This function returns a non-zero value if the file pointer is at the end of file. It returns 0 if the current position is not end of file. EOF means the pointer is at the offset == file size. This means it is the index of the next byte following the last byte of the file.

**Pars**        filehandle File handle returned by sfs\_fopen().

**Returns**    SB\_PASS    EOF  
              SB\_FAIL    not EOF

**See Also**    sfs\_fopen(), sfs\_fseek(), sfs\_fwrite(), sfs\_fread()

## Example

```
FILEHANDLE fp;
char buf[20]="Test data";
fp = sfs_fopen("d:\\data.dat", "r+b");
while(!sfs_feof(fp))
    sfs_fread(buf, 1, 20, fp);
sfs_fclose(fp);
```

int       **sfs\_fflush** (FILEHANDLE filehandle)

**Summary**   Flush all data associated with the file to the storage media.

**Details**    The file system uses a memory cache to store file data to minimize writes to the storage media. This function forces all cached data for this file to be written to the storage media.

**Pars**       filehandle   File handle returned by sfs\_fopen().

**Returns**    SB\_PASS    Success.  
              SB\_FAIL    Device has been removed or there is some other error.

**See Also**   sfs\_fopen(), sfs\_fwrite()

## Example

```
FILEHANDLE fp;
char buf[20]="Test data";
fp = sfs_fopen("d:\\data.dat", "r+b");
sfs_fwrite(buf, 1, 20, fp);
sfs_fflush(fp);
sfs_fclose(fp);
```

unsigned long   **sfs\_filelength** (const char \*filename)

**Summary**    Return the length of a file, in bytes.

**Details**    This function returns the length of the file specified by *filename* if the file exists. If it does not exist, -1 (0xFFFFFFFF) is returned. If the file is currently open, the current file length is returned.

**Pars**       filename    the name of the file whose length will be determined

**Returns**    (unsigned long)-1   File not found.  
              other            Length of file or 0 for a directory.

**See Also**   sfs\_findfirst(), sfs\_findnext()

## Example

```
#define FN "d:\\test.dat"
If(sfs_findfile(FN) == SB_PASS)
    printf("File length = %d", sfs_filelength(FN));
else
    printf("File not found");
```

int **sfs\_findclose** (FILEINFO \* fileinfo) [FINDFIRST\_SUPPORT]

**Summary** Cleans up after the findfirst/findnext operation.

**Details** Call this function after you are finished with a findfirst/findnext operation to free the internal buffer that was used for it. See the example for sfs\_findfirst(), which makes this clear.

**Pars** fileinfo The returned file info which includes the file's name and size.

**Returns** 0 The internal buffer has been freed.  
-1 Device is not mounted.

**See Also** sfs\_findfirst(), sfs\_findnext()

**Example** See sfs\_findfirst().

int **sfs\_findfile** (const char \*filename)

**Summary** Test if a file exists.

**Details** This function searches for the file or directory specified by *filename*. If the file exists, a positive value is returned; otherwise 0 is returned. This function returns the correct result even if the file is open. If you only want to find a file and not a directory, use sfs\_stat() and check (fileinfo.st\_mode & S\_IFDIR) to see if it is a directory rather than a file.

**Pars** filename The name of the file or directory to find.

**Returns** SB\_PASS File found.  
SB\_FAIL File not found.

**See Also** sfs\_findfirst(), sfs\_findnext()

## Example

```
if(sfs_findfile("d:\\test.dat") > 0)
    printf("Found test.dat");
```

int **sfs\_findfirst** (const char \* filespec, FILEINFO \* fileinfo) [FINDFIRST\_SUPPORT]

**Summary** Provides information about the first instance of a file or directory whose name matches the name specified by the *filespec* argument.

**Details** If successful, this function returns a unique search ID identifying the file or directory matching the *filespec* specification, which can be used in a subsequent call to *sfs\_findnext*(). Otherwise, it returns -1. Check (fileinfo.st\_mode & S\_IFDIR) to see if it is a directory rather than a file.

**Pars** filespec The search string, which may include wildcards '\*' and '?'. These must only appear in the filename and not in the path. The following are valid *filespec*:  
"d:\\\*.\*)"   
"d:\\path\\\*.dat"   
"d:\\path\\test?.\*)"   
"d:\\path\\test?2.dat"

fileinfo The returned file info which includes the file's name and size.

**Returns** id File found matching *filespec*.  
-1 No file found.

**See Also** *sfs\_findclose*(), *sfs\_findfile*(), *sfs\_findnext*()

### Example

```
FILEINFO fileinfo;  
int id;  
id = sfs_findfirst("d:\\*.*", &fileinfo);  
while(id >= 0)  
{  
    printf("File Name: %s, File Size: %d\n", fileinfo.name, fileinfo.st_size);  
    id = sfs_findnext(id, &fileinfo);  
}  
sfs_findclose(&fileinfo);
```

int **sfs\_findnext** (int id, FILEINFO \* fileinfo) [FINDFIRST\_SUPPORT]

**Summary** Finds the next file or directory, if any, whose name matches the *filespec* argument in a previous call to *sfs\_findfirst*(), and returns information about it in the *fileinfo* structure.

**Details** If successful, this function returns a unique search id identifying the next file or directory it finds that matches the *filespec* specification that was passed to *sfs\_findfirst*(). Otherwise, returns -1. Check (fileinfo.st\_mode & S\_IFDIR) to see if it is a directory rather than a file.

**Pars** id The search ID returned by the last *sfs\_findnext*() or *sfs\_findfirst*() call.  
fileinfo The returned file info which includes the file's name and size.

**Returns** id File found matching *filespec*.  
-1 No file found.

**See Also** *sfs\_findclose*(), *sfs\_findfile*(), *sfs\_findfirst*()

**Example** See *sfs\_findfirst*()

int **sfs\_flushall** (uint nID)

**Summary** Flush all data associated with the files opened on the specific disk to the storage media.

**Details** The file system uses a memory cache to store file data to minimize writes to the storage media. This function forces all cached data of all open files on that specific disk to be written to the storage media.

**Pars** nID The device ID that was specified in the call to sfs\_devreg().

**Returns** SB\_PASS Success.  
SB\_FAIL Device has been removed or other error.

**See Also** sfs\_devreg()

**Example**  
`sfs_flushall(0);`

FILEHANDLE **sfs\_fopen** (const char \*filename, const char \*mode)

**Summary** Opens a file for read/write access.

**Details** This function must be called before any file access operations. This function will open the file specified by filename with the specified access mode. It returns the file handle. Do not directly access the fields of the structure pointed to by the file handle.

*The file is opened in binary mode. There is no text mode support.* It is fine to pass “rb” instead of “r”, for example, but it is not necessary. If other characters are passed in addition to the characters below, they are ignored (e.g. “rt”).

**Pars**

filename	The file name, which must include the full pathname. For example, d:\path\file.ext. The path must exist before the file is opened. Otherwise, please call sfs_mkdir() first to create the directories in the path.
mode	Access mode. Supported modes are as follows (other characters are ignored):
"r"	Opens for reading only. If the file does not exist or cannot be found, this call fails. The file pointer starts at the beginning of the file.
"w"	Opens an empty file for reading and writing. If the given file exists, its contents are destroyed.
"a"	Opens a file for appending (allows reading and writing). The file pointer starts at the end of the file.
"r+"	Opens for both reading and writing. (The file must exist.) The file pointer starts at the beginning of the file.
"w+"	Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.
"a+"	Same as “a”.

Before smxFS v2.10, a file can be opened for reading with mode “r” by multiple tasks simultaneously, as long as there are enough memory resources (i.e. file handle structures and

file cache memory). If one file is opened with the “r+” mode, the second open request of “r+” mode will be refused. If you are using v2.10 or later, this limitation is removed.

**Returns** file handle Success.  
NULL File not found or other error. Do not pass a NULL handle to other API calls.

**See Also** sfs\_fclose(), sfs\_mkdir()

**Example**

```
/* single open request */
FILEHANDLE fp;
fp = sfs_fopen("d:\\test.bin", "r");
if(fp != NULL)
{
    sfs_fread(...);
    sfs_fclose(fp);
}
```

int **sfs\_format** (uint nID, FORMATINFO \* formatinfo) [FORMAT\_SUPPORT]

**Summary** Formats a disk.

**Details** Formats a disk, according to the tables in fmount.c. See section 3.3 FAT12/16/32 for discussion of the FAT type and number of sectors per cluster used. This function does not touch the partition table or create one.

Note that smxFS can be set to autoformat an unformatted disk during the mount process. This is controlled by setting pDeviceInfo->wAutoFormat in the IOCTL() routine of the driver. See section Block Device Interface in the smxBase User’s Guide for details.

Some media require a low-level format. If the media is blank or corrupted, it is necessary to first call the driver’s IOCTL() function to do that, then call sfs\_partition() if it must have a partition table, and then call sfs\_format() to do the FAT format.

You can also pass format parameters via the formatinfo structure. See the FORMATINFO structure in fapi.h. The fields of this structure are:

uint	wFATNum	Number of FAT tables to create. Normally 2.
uint	wRootDirNum	Number of root directory entries to create. Must be 0 for FAT32. For FAT12/16, 512 is recommended for best compatibility. Otherwise it should be set to a multiple of (sector size / 32) so it fully fills some number of sectors.
u32	dwVolumeID	The disk’s volume ID, such as 1234ABCD.
char *	VolumeLabel	VolumeLabel such as “MYDISK”. If set to 0 or an empty string, the default NO_VOLUME_LABEL (“NO NAME”) will be used. Max 11 chars (not including NUL). Is automatically padded with spaces if shorter or truncated if longer.
u8	bMediaType	0xF0 removable media, 0xF8-0xFF fixed media. If it is 0, the media type is set to 0xF0 for removable media or 0xF8 for fixed media. This is determined by the wRemovable flag set in the driver’s IOCTL() function, for SBD_IOCTL_GETDEVINFO.



**Returns** value      Number of items read.  
0              Error or reach the end of file.

**See Also**    sfs\_fopen(), sfs\_fwrite()

**Example**

```
FILEHANDLE fp;  
char buff[20];  
fp = sfs_fopen("d:\\test.bin", "rb");  
if(fp != NULL)  
{  
    sfs_fread(buf, 1, 20, fp); // if "test.bin" file size is 0, this call will return 0.  
    sfs_fclose(fp);  
}
```

unsigned long    **sfs\_freekb** (uint nID)

**Summary**    Returns the size of the free space on the disk, in kilobytes.

**Details**    This function returns the amount of free space on the disk specified by nID. The first time it is called, it either reads it from the FSInfo sector or scans the whole FAT. Scanning the FAT is slow, and reading FSInfo is fast, but the FSInfo values can be wrong. The FSInfo sector is a feature of FAT32. We implemented it also for FAT12/16 if SFS\_FAT\_FSINFO\_SUPPORT is 1. Another setting, SFS\_USE\_FAT32\_FSINFO, can disable it for FAT32 if 0. In addition to the possibility that it can be unreliable, it also reduces performance to have to modify that sector every time the disk changes, especially for flash media. But if the time to scan the FAT is too long, you may have no other choice than to enable these settings. Note also that for removable FAT12/16 disks the FSInfo sector is not used because the media could be plugged into a system running a different OS and it would not modify the values in the FSInfo sector, which would make it unreliable. See section 2.3 Configuration Settings for more discussion of these settings.

**Pars**        nID            The device ID that was specified in the call to sfs\_devreg().

**Returns**    size            Free size (kilo-bytes) of the disk.  
0xFFFFFFFF    The deviceID is not valid or the device is not inserted.

**See Also**    sfs\_devreg(), sfs\_totalkb()

**Example**

```
printf("The free size of disk 0 is %dKB", sfs_freekb(0));
```

int **sfs\_fseek** (FILEHANDLE filehandle, long offset, int whence)

**Summary** Moves the file pointer to the specified location in the file.

**Details** This function moves the file pointer associated with *filehandle* to a new location that is *offset* bytes from the origin, *whence*. The next read/write operation on the file takes place at this new location. You can NOT use this function to reposition the pointer anywhere in a file. Attempting to move the pointer before the beginning of file is an error; the pointer is moved to the beginning of file and the return value is 0. If the file is open for read/write mode, moving the pointer beyond the end of file will extend the file but the data in this new area is unpredictable until you write data there.

**Pars** filehandle File handle returned by sfs\_fopen().  
offset Number of bytes from *whence*.  
whence Initial position; three predefined constants are:

**SFS\_SEEK\_CUR** Current position of file pointer  
**SFS\_SEEK\_END** End of file  
**SFS\_SEEK\_SET** Beginning of file

**Returns** 0 Success.  
!0 Fail.

**See Also** sfs\_fopen(), sfs\_fread(), sfs\_fwrite()

### Example

```
/* normal seek operation */
FILEHANDLE fp;
char buff[20];
fp = sfs_fopen("d:\\test.bin", "rb");
if(fp != NULL)
{
    sfs_fseek(fp, 10, SFS_SEEK_SET);
    sfs_fread(buff, 1, 20, fp);
    sfs_fclose(fp);
}

/* seeking beyond the file area will cause error if it is Read-Only */
FILEHANDLE fp;
char buff[20]="This is a test.";
fp = sfs_fopen("d:\\test.bin", "rb");
if(fp != NULL)
{
    sfs_fseek(fp, 10, SFS_SEEK_END); // this will move the pointer to the end of file.
    sfs_fclose(fp);
}

/* seeking beyond the file area will increase the files size if it is Read/Write */
FILEHANDLE fp;
char buff[20]="This is a test.";
fp = sfs_fopen("d:\\test.bin", "wb");
if(fp != NULL)
{
    sfs_fseek(fp, 10, SFS_SEEK_END); // file size is 10 bytes now but the contents are unpredictable.
    sfs_fclose(fp);
}
```

long **sfs\_ftell** (FILEHANDLE filehandle)

**Summary** Returns the current file pointer.

**Details** This function returns the current file pointer.

**Pars** filehandle File handle returned by sfs\_fopen().

**Returns** value File pointer position.

**See Also** sfs\_fopen(), sfs\_fseek()

### Example

```
FILEHANDLE fp;
char buff[20]="Test data";
fp = sfs_fopen("d:\\data.dat", "r+b");
sfs_fwrite(buf, 1, 20, fp);
sfs_fseek(fp, sfs_ftell(fp) -1, SFS_SEEK_SET );
sfs_fclose(fp);
```

int **sfs\_ftruncate** (FILEHANDLE filehandle) [FULL\_FEATURES\_SUPPORT]

**Summary** Truncates a file at the current file pointer.

**Details** This function discards all data at and beyond the current file pointer. All bytes before the file pointer are kept. The file size is then set to the current file pointer. This means that the value of the file pointer indicates how many bytes to keep. Also, it means that after this operation, the file pointer is at EOF (1 byte past the end of the data).

**Pars** filehandle File handle returned by sfs\_fopen().

**Returns** SB\_PASS The file has been truncated successfully.  
SB\_FAIL The file was not truncated due to an error.

**See Also** sfs\_fopen(), sfs\_fseek(), sfs\_fwrite()

### Example

```
FILEHANDLE fp;
char buff[20]="Test data";
fp = sfs_fopen("d:\\data.dat", "r+b");
sfs_fwrite(buf, 1, 20, fp);
sfs_fseek(fp, sfs_ftell(fp) -10 , SFS_SEEK_SET );
sfs_ftruncate(fp); //discard 10 bytes
sfs_fclose(fp);
```

size\_t      **sfs\_fwrite** (void \*buf, size\_t size, size\_t items, FILEHANDLE filehandle)

**Summary**    Writes some data to an open file.

**Details**    This function writes up to (*items \* size*) bytes from *buf* to the file starting at the current file position in the file. The file pointer is increased by the number of bytes actually written. The file pointer position is indeterminate if an error occurs. The value of a partially written item cannot be determined.

If the file was opened in read-only mode “r”, `sfs_fwrite()` will return 0 and no data will be written to the file.

**Pars**

buf	Pointer to the data to be written.
size	Item size in bytes.
items	Maximum number of items to be written.
filehandle	File handle returned by <code>sfs_fopen()</code> .

**Returns**

value	Number of items written.
0	Error.

**See Also**    `sfs_fopen()`, `sfs_fread()`

### Example

```
/* normal write operation */
FILEHANDLE fp;
char buf[20]="This is a test.";
fp = sfs_fopen("d:\\test.bin", "wb");
if(fp != NULL)
{
    sfs_fwrite(buf, 1, 20, fp);
    sfs_fclose(fp);
}

/* write to a read-only file will return error */
FILEHANDLE fp;
char buf[20]="This is a test.";
fp = sfs_fopen("d:\\test.bin", "rb");
if(fp != NULL)
{
    sfs_fwrite(buf, 1, 20, fp); /* returns 0 and no data is written */
    sfs_fclose(fp);
}
```

char \* **sfs\_getcwd** (char \* buffer, int maxlen) [CWD\_SUPPORT]

**Summary** Get the current working directory.

**Details** Saves the current working directory for the current task into \*buffer. The directory is the full path including drive letter.

**Pars** buffer The memory pointer to store the current working directory.  
maxlen The maximum length of the buffer.

**Returns** Pointer to the current working directory string.  
NULL There is no CWD for the current task, buffer par is NULL, or the path string including NUL is longer than maxlen.

**See Also** sfs\_setcwd() (or sfs\_chdir())

### Example

```
void main()
{
    char buf[128];
    sfs_setcwd("a:\\test");
    sfs_getcwd(buf, 128);
    printf("Current Working Directory is %s", buf);
}
```

const SBD\_IF \* **sfs\_getdev** (uint nID)

**Summary** Get the device driver of the registered device.

**Details** You can call this function to get the registered device driver.

**Pars** nID The device ID that was specified in the call to sfs\_devreg().

**Returns** DevInterface The registered device driver.  
NULL The device driver does not exist.

**See Also** sfs\_devreg()

### Example

```
void main()
{
    sfs_devreg(0, pDevInterface);
    pDevInterface = sfs_getdev(0);
}
```

int **sfs\_getlasterror** (uint nID)

**Summary** Get the last error code on the specified disk.

**Details** When any file system operation fails, you can call this function to get more detailed failure information. This error code will NOT be reset unless you call this function or a new error occurs.

**Pars** nID The device ID that was specified in the call to sfs\_devreg().

**Returns** The error code of the last failed file operation. Error code is defined as:

SFS_ERR_NO_ERROR	No error.
SFS_ERR_DISK_REMOVED	Disk is removed.
SFS_ERR_DISK_IO	Disk driver return I/O error.
SFS_ERR_INVALID_BOOTSEC	Boot sector of that disk is invalid
SFS_ERR_INVALID_DIR	Directory entry contains invalid field.
SFS_ERR_INVALID_FAT	File allocation table contains invalid value.
SFS_ERR_INVALID_PAR	Function got invalid parameter or settings
SFS_ERR_FAT32_NOT_SUPP	Disk is FAT32 but your system does not enable it.
SFS_ERR_DIR_FULL	Directory entry is full and file system cannot allocate more clusters for it. The disk may be full or it is FAT12/16 and the root directory is full.
SFS_ERR_DISK_FULL	File system cannot find free data cluster.
SFS_ERR_DISK_WP	Disk is write-protected.
SFS_ERR_FILE_EXIST	File already exists. For example, you want to rename a file, but a file with the new name already exists.
SFS_ERR_FILE_NOT_EXIST	File does not exist. For example, you want to rename a file but the file does not exist.
SFS_ERR_FILE_WP	File is write-protected.
SFS_ERR_FILE_OPEN	File is still open when trying to delete it.
SFS_ERR_OUT_OF_MEM	File system could not allocate required memory.

**See Also** sfs\_fopen(), sfs\_fread(), sfs\_fwrite()

### Example

```
void main()
{
    FILEHANDLE fp;
    sfs_devreg(0, pDevInterface);
    sfs_delete("A:\test.bin");
    fp = sfs_fopen("A:\test.bin", "rb");
    if(fp == NULL)
    {
        printf("Last Error Code is %d\r\n", sfs_getlasterror(0));
    }
}
```

int **sfs\_getprop** (const char \* filename, FILEINFO\* fileinfo) [PROPERTY\_SUPPORT]

**Summary** Retrieve the properties for a file or directory.

**Details** Obtains the information about the file or directory specified by filename and stores it in the structure pointed to by fileinfo. The FILEINFO structure has the following fields, which the user can read:

**st\_atime** Time of last access of file.  
**st\_ctime** Time of creation of file.  
**st\_mtime** Time of modification of file.  
**st\_mode** Bit mask for file-mode information. The **S\_IFDIR** bit is set if filename specifies a directory; the **S\_IFREG** bit is set if filename specifies an ordinary file. User read/write bits are set according to the file's permission mode.  
**st\_size** Size of the file in bytes.

**Pars** filename The full or relative file name of an existing file.  
fileinfo The pointer to a structure that receives the result.

**Returns** 0 Got properties.  
!=0 File or directory not found.

**See Also** sfs\_chmod(), sfs\_setprop(), sfs\_stat(), sfs\_timestamp()

### Example

```
void appl_init()
{
    FILEINFO fileinfo;
    sfs_init();
    sfs_devreg(sfs_GetRAMInterface(), 0);
    if(sfs_getprop("A:\\test.bin", &fileinfo) == 0)
    {
        printf("File size is %d\n", fileinfo.st_size);
        if(!(fileinfo.st_mode & S_IWRITE))
            printf("File is read-only\n");
    }
}
```

int **sfs\_getvolname** (uint nID, char \* name) [VOLUME\_SUPPORT]

**Summary** Get a disk's volume name.

**Details** This function gets a disk's volume name. This is the volume name displayed by Windows Explorer.

**Pars** nID The device ID that was specified in the call to sfs\_devreg().  
name The buffer pointer to store the volume name. The length of the buffer must be at least 12 bytes.

**Returns** 0 Got the volume name.  
!=0 Volume name not found.

**See Also** sfs\_setvolname(), sfs\_devreg()

**Example**

```
char volname[16];
if (sfs_getvolname(0,volname) == 0)
    printf("This disk's volume name is %s", volname);
```

int **sfs\_init** (void)

**Summary** Initializes the smxFS internal data structures.

**Details** This function must be called before calling any other smxFS API functions. Then you must call sfs\_devreg() to register each device driver.

**Pars** none

**Returns** SB\_PASS Success.  
SB\_FAIL Initialization failed. smxFS could not allocate the required resources.

**See Also** sfs\_exit()

**Example**

```
void appl_init()
{
    if(sfs_init() == SB_FAIL)
        wr_string(0,0,WHITE,BLACK,!BLINK,"Error initializing file system.");
    else
        wr_string(0,0,WHITE,BLACK,!BLINK,"File system initialized.");
}
```

int **sfs\_ioctl** (uint nID, uint command, void \* par)

**Summary** Runs the specified driver-specific command.

**Details** This function allows a device driver to do some special operations that are only related to that particular driver. smxFS directly passes the command and parameter to the device driver's IOCTL() function.

**Pars** nID The device ID that was specified in the call to sfs\_devreg().  
command Driver-specific command. User commands must be >= SBD\_IOCTL\_CUSTOM. Values less than this are used internally by smxFS functions for media change, write protect, and similar common operations.  
param Command-specific parameter. See driver implementation.

**Returns** SB\_PASS Operation succeeded.

SB\_FAIL Operation failed or command is not supported by the driver.

**See Also** sfs\_devreg()

**Example**

```
sff_ioctl(0, SB_BD_IOCTL_NOR_BLKRECLAIM, 10) /* reclaim at least 10 sectors */
```

int **sfs\_mkdir** (const char \*path) [MKDIR\_SUPPORT]

**Summary** Creates a directory on the disk.

**Details** If the directory already exists, this function will do nothing and just return success. To create a subdirectory, it is necessary to create the parent directory first. For example, if you want to create d:\parent\sub, first create parent, then sub. See the example below.

**Pars** path The full path name. For example, “d:\parent\sub”, do not add a backslash ‘\’ at the end of the path name.

**Returns** SB\_PASS The directory has been created successfully.  
SB\_FAIL The parent directory does not exist or there is no free space to create the directory.

**See Also** sfs\_rmdir()

**Example**

```
/* create one directory on the root */  
sfs_mkdir("d:\path");  
  
/* create one parent directory and two subdirectory */  
if(sfs_mkdir("d:\parent"))  
{  
    sfs_mkdir("d:\parent\sub1");  
    sfs_mkdir("d:\parent\sub2");  
}
```

int **sfs\_move** (const char \* oldname, const char \* newname) [RENAME\_SUPPORT]

Alias for sfs\_rename(). See its call description below.

int **sfs\_partition** (uint nID, PARTITIONINFO \* partitioninfo) [FORMAT\_SUPPORT]

**Summary** Write the partition table of a disk.

**Details** Write the partition table of a disk according to the information provided in the PARTITIONINFO structure. It can only create primary partitions, not an extended or logical partition. Up to four primary partitions can be created, which should be plenty for most embedded systems. smxFS can use extended and logical partitions on a disk that was already partitioned by another OS or utility.

Some media require a low-level format. If the media is blank or corrupted, it is necessary to first call the driver's IOCTL() function to do that, then call sfs\_partition() if it must have a partition table, and then call sfs\_format() to do the FAT format.

You must pass a partition information structure to indicate data used to partition the disk. The PARTITIONINFO structure is defined in fapi.h. The fields are:

uint	ActivePartition	Which partition is active, from 0 - 3.
uint	ForceCreate	Set to 1 to force creating partition table even if the first sector is the boot sector (special case) or blank.
uint	SecPerTrack	Used to convert the LBA sector to CHS. Set to 0 if you do not care about CHS fields.
uint	HeadPerCyl	Used to convert the LBA sector to CHS. Set to 0 if you do not care about CHS fields.
uint	ReservedSectors	Number of reserved sectors between the partition table and the FIRST partition.
u8	IDNumber[SFS_MAX_PARTITION_NUM]	ID numbers for each of the partitions.
u32	Size[SFS_MAX_PARTITION_NUM]	Size of each partition (number of sectors).
u8 *	pMBRProg	MBR boot code or NULL. It is copied to the beginning (1st byte) of the MBR. If you do not care about the boot code, set this to NULL.
uint	MBRProgSize	The size of the MBR boot code. Must be <= 446 bytes to fit before the partition table.

**Pars** nID The device ID that was specified in the call to sfs\_devreg().  
partitioninfo Pointer to structure with partition parameters.

**Returns** SB\_PASS Success.  
SB\_FAIL Some error occurred.

**See Also** sfs\_init(), sfs\_devreg(), sfs\_format()

### Example

```
PARTITIONINFO partinfo;  
memset(&part_info, 0, sizeof(PARTITIONINFO)); /* clear all unused fields */  
partinfo.ActivePartition = 0;  
partinfo.ForceCreate = 1;  
partinfo.ReservedSectors = 15; /* The first partition will start at 16th sector */  
/* only has one partition so the other three of each of the following fields are empty */  
partinfo.IDNumber[0] = 0x4;  
partinfo.Size[0] = 0; /* The first partition will occupy the whole media size */  
  
sfs_partition(0, &partinfo);
```

If you want to add boot code to this sector, set the pMBRProg and MBRProgSize fields. See how it is done in the example for sfs\_format(), which is similar.

int **sfs\_rename** (const char \* oldname, const char \* newname) [RENAME\_SUPPORT]

**Summary** Renames a file or directory or moves a file.

**Details** This function renames the file or directory specified by *oldname* to the name given by *newname*. It can also move a single file elsewhere on the same volume or to another volume. It can move a directory only to the same volume. The old name must be an existing file or directory. The new name must not be the name of an existing file or directory, and its path must exist (see example below). If the path is different in the two names, the file is moved. If the destination is on the same volume, this is done by simply moving the directory entry, but if it is on a different volume, then this function calls sfs\_copy() to copy the data from the old file to the new file and then deletes the old file.

Note: This function cannot move a directory tree to another volume, but it can do this on the same volume. Moving to a different volume requires a recursive copy operation, which is not implemented. Moving on the same volume is only a matter of moving the directory entry for the root of the tree. The second part of the example below illustrates moving a subdirectory (which may contain other subdirectories) to another directory on the same volume.

**Pars** oldname The old file name.  
newname The new file name.

**Returns** SB\_PASS File or directory renamed or moved.  
SB\_FAIL *oldname* does not exist or *newname* is used by another file.

**See Also** sfs\_findfile()

### Example

```
FILEHANDLE fp;
char buff[20]="Test data";
fp = sfs_fopen("d:\\data.dat", "w+b");
sfs_fwrite(buf, 1, 20, fp);
sfs_fclose(fp);
sfs_rename("d:\\data.dat", "d:\\newdata.dat");
...
/* Example of move. Assumes files exist. */
sfs_mkdir("d:\\target");
sfs_rename("d:\\source\\subdir1", "d:\\target\\subdir1");
```

void **sfs\_rewind** (FILEHANDLE filehandle) [Extended API]

**Summary** Moves the file pointer to the beginning of the file.

**Details** This is equivalent to sfs\_fseek(filehandle, 0, SFS\_SEEK\_SET).

**Pars** filehandle File handle returned by sfs\_fopen().

**Returns** none

**See Also** sfs\_fopen(), sfs\_fseek()

**Example**

```
FILEHANDLE fp;  
char buff[20];  
fp = sfs_fopen("d:\\data.dat", "rb");  
sfs_fread(buff, 1, 20, fp);  
sfs_rewind(fp);  
sfs_fclose(fp);
```

int **sfs\_rmdir** (const char \*path) [MKDIR\_SUPPORT]

**Summary** Deletes a directory and all files and subdirectories in it from the disk.

**Details** All files and subdirectories in this directory are removed. To delete a single file, call sfs\_fdelete().

**Pars** path The full path name. For example, "d:\\parent\\sub". Do not add a backslash '\ ' at the end of the path name.

**Returns** SB\_PASS The directory has been removed successfully.  
SB\_FAIL The directory does not exist.

**See Also** sfs\_mkdir(), sfs\_fdelete()

**Example**

```
/* delete one directory on the root */  
sfs_rmdir("d:\\path");
```

unsigned long **sfs\_scanfreeclus** (uint nID, uint iScanSectorNum) [FREECLUS\_SUPPORT]

**Summary** Force smxFS to scan and store more free cluster numbers in the free cluster cache.

**Details** The application can call this API during idle time to get more free cluster numbers from the FAT and store them in the free cluster cache. When **sfs\_fwrite()** need to allocate more data cluster, it won't need to scan the FAT and can get the free cluster directly from the free cluster cache. See section 3.17.2 Free Cluster Cache for details about this feature.

**Pars** nID The device ID that was specified in the call to sfs\_devreg().  
iScanSectorNum The number of FAT sectors you want smxFS to scan.

**Returns** The number of free clusters in the free cluster cache.

**See Also** sfs\_devreg(), sfs\_fwrite()

**Example**

```
/* scan 8 sectors of the FAT to get more free clusters */  
printf("free cluster number in the cache is %d", sfs_scanfreeclus(0, 8));
```

int **sfs\_setcwd** (const char \*path) [CWD\_SUPPORT]

**Summary** Set the current working directory.

**Details** Sets the current working directory for the current task. Each task may have its own working directory. This function fails if the directory does not exist. You must specify the full path name when you first call this function from a particular task and then you can use relative path if you change the directory within the same device.

Note that `sfs_chdir()` is an alias for this function. This is the standard C library name.

**Pars** path The full or relative path name of your new working directory.

**Returns** SB\_PASS The working directory has been changed.  
SB\_FAIL The device is not valid or there is no free working directory entry in the CWD table.

**See Also** `sfs_chdir()`, `sfs_getcwd()`, `sfs_mkdir()`

**Example**

```
void appl_init()
{
    sfs_init();
    sfs_devreg(sfs_GetRAMInterface(), 0);
    sfs_mkdir("a:\\test");
    sfs_mkdir("a:\\test\\dir1");
    sfs_setcwd("a:\\test");
    sfs_setcwd ("dir1");
    sfs_setcwd ("."); /* return to a:\\test */
}
```

int **sfs\_setpassthrough**(FILEHANDLE filehandle, BOOLEAN bEnable)

**Summary** Enables or disables passthrough mode for read and write.

**Details** This function enables or disables passthrough mode for subsequent read and write operations, provided the conditions are met that are listed in section 3.19 Pass Through Mode.

**Pars** filehandle File handle returned by `sfs_fopen()`.  
bEnable Enables if TRUE; disables if FALSE.

**Returns** SB\_PASS Pass through mode enabled.  
SB\_FAIL A required condition is not met so pass through mode not enabled.

**See Also** `sfs_fread()`, `sfs_fwrite()`

## Example

```
FILEHANDLE fp;
char buf[4096];
fp = sfs_fopen("d:\\test.bin", "wb");
if(fp != NULL)
{
    sfs_setpassthrough(fp, TRUE);
    sfs_fwrite(buf, 1, 4096, fp);
    sfs_fclose(fp);
}
```

int **sfs\_setprop** (const char \* filename, FILEINFO\* fileinfo, uint flag) [PROPERTY\_SUPPORT]

**Summary** Set the properties for a file or directory.

**Details** Sets the attributes and timestamps for a file or directory. It is the application's responsibility to make sure no other task has this file open at the same time. The file modification time will be changed by the fclose() function call so if another task has this file open, this date will be lost after that task closes the file.

**Pars**

file	The full or relative file name of the file or directory whose properties you want to modify.
fileinfo	The structure containing the new properties to set for the file or directory.
flag	Which properties should be modified. Valid flags include: SFS_SET_ATTRIBUTE SFS_SET_CREATETIME SFS_SET_WRITETIME

**Returns** 0 The properties have been changed successfully.  
> 0 File or directory not found.

**See Also** sfs\_chmod(), sfs\_getprop(), sfs\_stat(), sfs\_timestamp()

## Example

```
void appl_init()
{
    sfs_init();
    sfs_devreg(sfs_GetRAMInterface(), 0);
    /* Only change the modification time to 06/08/2006. Windows displays this time in File Explorer. */
    fileinfo.st_mtime.wYear = 26; /* year 2006 */
    fileinfo.st_mtime.wMonth = 6;
    fileinfo.st_mtime.wDay = 8;
    fileinfo.st_mtime.wHour = 6;
    fileinfo.st_mtime.wMinute = 30;
    fileinfo.st_mtime.wSecond = 59;
    fileinfo.st_mtime.wMilliseconds = 0;
    sfs_setprop("A:\\test.bin", &fileinfo, SFS_SET_WRITETIME);

    /* Set the file as Read Only, System, and Hidden file. */
    fileinfo.bAttr = SFS_ATTR_READ_ONLY|SFS_ATTR_HIDDEN|SFS_ATTR_SYSTEM;
    sfs_setprop("A:\\test.bin", &fileinfo, SFS_SET_ATTRIBUTE);
}
```

```

    /* Change the file's creation time to 06/09/2006, and set Read Only property. */
    fileinfo.bAttr = SFS_ATTR_READ_ONLY;
    fileinfo.st_ctime.wYear = 26;          /* year 2006 */
    fileinfo.st_ctime.wMonth = 6;
    fileinfo.st_ctime.wDay = 9;
    fileinfo.st_ctime.wHour = 5;
    fileinfo.st_ctime.wMinute = 24;
    fileinfo.st_ctime.wSecond = 24;
    fileinfo.st_ctime.wMilliseconds = 0;
    sfs_setprop("A:\\test.bin", &fileinfo, SFS_SET_CREATETIME|SFS_SET_ATTRIBUTE);
}

```

int **sfs\_setvolname** (uint nID, const char \* name) [VOLUME\_SUPPORT]

**Summary** Set a disk's volume name.

**Details** This function sets a disk's volume name. This is the volume name displayed by Windows Explorer. You can also change this volume name using the Windows Format utility.

**Pars**

nID	The device ID that was specified in the call to sfs_devreg().
name	The new volume name. The maximum volume length is 11 bytes and only allows a-z and 0-9.

**Returns**

0	Set the volume name.
!=0	Volume name could not be set.

**See Also** sfs\_getvolname(), sfs\_devreg()

### Example

```

char volname[16] = "TestVol1";
if (sfs_setvolname(0, volname) == 0)
    printf("Set the volume name to %s", volname);

```

int **sfs\_stat** (const char \* filename, FILEINFO\* fileinfo) [PROPERTY\_SUPPORT]

Alias for sfs\_getprop(), but changes the return value as follows:

**Returns**

0	File status information is obtained.
-1	File not found.

int **sfs\_timestamp** (const char \* filename, DATETIME\* datetime) [PROPERTY\_SUPPORT]

**Summary** Set the modification time for a file or directory.

**Details** Sets the modification time for a file or directory. It is the application's responsibility to make sure no other task has this file open at the same time. The file modification time will be changed by the fclose() function call so if another task has this file open, this date will be lost after that task closes the file.

**Pars** file The full or relative name of the file or directory whose time you want to modify.  
datetime The structure containing the new modification time.

**Returns** 0 The timestamp has been changed successfully.  
!=0 File or directory not found.

**See Also** sfs\_chmod(), sfs\_getprop(), sfs\_setprop(), sfs\_stat()

### Example

```
void appl_init()
{
    DATETIME datetime;
    sfs_init();
    sfs_devreg(sfs_GetRAMInterface(), 0);
    /* only change the written time to 2005/09/22, Windows will display this time in File Explorer */
    datetime.wYear = 25; /* year 2005 */
    datetime.wMonth = 9;
    datetime.wDay = 22;
    datetime.wHour = 8;
    datetime.wMinute = 11;
    datetime.wSecond = 42;
    datetime.wMilliseconds = 0;
    sfs_timestamp("A:\\test.bin", &datetime);
}
```

unsigned long **sfs\_totalkb** (uint nID)

**Summary** Returns the total size of the disk, in kilobytes.

**Details** This function returns the total size of the disk specified by nID.

**Pars** nID The device ID that was specified in the call to sfs\_devreg().

**Returns** size Total size (kilo-bytes) of the disk.  
0xFFFFFFFF The Device ID is not valid or the device is not inserted.

**See Also** sfs\_devreg(), sfs\_freekb()

### Example

```
printf("The total size of disk 0 is %dKB", sfs_totalkb(0));
```

int        **sfs\_writeprotect** (uint nID)

**Summary**    Returns the current status of whether the disk is write protected.

**Details**     This function returns the write protecte status of the disk specified by nID.

**Pars**        nID            The device ID that was specified in the call to sfs\_devreg().

**Returns**    SB\_PASS    Disk is write protected.  
              SB\_FAIL    Disk is not write protected.

**See Also**    sfs\_devreg()  
              sfs\_devstatus()

**Example**

```
if(SB_PASS == sfs_writeprotect(0))  
    printf("The disk 0 is write protected.");
```

## 5. Device Driver Details

### 5.1 Block Device Interface

smxFS does all interaction with devices using only by the Block Device Interface (`\SMX\XBASE\bbd.h`). This makes it easy to add new device drivers. All that is necessary is to implement the interface functions and then call `sfs_devreg()` to register the device. It is not necessary to make any changes to smxFS files to add a new device driver. This is a key element of the design of smxFS.

In addition to the API functions, the driver must provide the following function which simply returns a pointer to the block device interface structure, which holds pointers to the interface functions listed below. Note that “Device” in the name is replaced by the name of the driver (e.g. “RAM”, “USB”, etc.).

```
const SBD_IF  sfs_GetDeviceInterface(void)
```

The pointer returned is passed as the first parameter of `sfs_devreg()`, like this:

```
sfs_devreg(sfs_GetRAMInterface(), 0);
```

For details of the Block Device Interface, please see the smxBase User’s Guide.

### 5.2 Test Code for New Drivers

You can use the following routine to test your block device driver.

```

void testDriver()
{
    u32 i;
    u8 *pBuf;
    uint Status;
    SBD_DEVINFO DevInfo;
    const SBD_IF *plf = sfs_GetXXXInterface();

    if(plf)
    {
        if(SB_PASS == plf->IOctl(SBD_IOCTL_INSERTED, &Status) && SB_PASS == Status)
        {
            if(SB_PASS == plf->DiskOpen())
            {
                plf->IOctl(SBD_IOCTL_GETDEVINFO, &DevInfo);
                pBuf = (u8 *)malloc(3*DevInfo.dwSectorSize);
                if(pBuf)
                {
                    for(i = 0; i < DevInfo.dwSectorsNum; i++)
                    {
                        memset(pBuf + DevInfo.dwSectorSize, i, DevInfo.dwSectorSize);
                        plf->SectorRead(pBuf, i, 1);
                        plf->SectorWrite(pBuf + DevInfo.dwSectorSize, i, 1);
                        plf->SectorRead(pBuf + 2*DevInfo.dwSectorSize, i, 1);
                        if(memcmp(pBuf + DevInfo.dwSectorSize,
                                pBuf + 2*DevInfo.dwSectorSize,
                                DevInfo.dwSectorSize) != 0)
                        {
                            printf("Sector %d Read/Write Check failed\n", i);
                        }
                        plf->SectorWrite(pBuf, i, 1);
                    }
                    free(pBuf);
                }
                plf->DiskClose();
            }
        }
    }
}

```

### 5.3 Driver-Specific Notes

**smxFS disk drivers are considered to be part of the hardware porting layer not the core file system. It is your responsibility to check them carefully to ensure the implementation is correct for your hardware. You may need to modify a disk driver if your system has some special requirement. You may also need to change the disk driver to support different partitions or number of disks.**

**NAND, NOR, and USB** drivers may require modification of the porting layer of their respective modules (i.e. smxNAND, smxNOR, smxUSBH).

Keep in mind that the only thing the disk drivers need to follow is the block device interface defined in bbd.h (formerly xbd.h). If you have any questions about the details of a disk driver, discuss them with Micro Digital.

### 5.3.1 ATA Driver

The ATA driver supports only the basic ATA operations. It does not support ATAPI or DMA. You can add this support by yourself. Always check the source code carefully to understand how it works and make sure it is the correct implementation for your system.

### 5.3.2 CompactFlash Driver

The CompactFlash driver has been tested on MCF5485EVB and MCF5329EVB. On MCF5485EVB, the LogicPD CPLD code adds CompactFlash memory mode storage card support. To port it to another hardware platform, you should implement the following functions for your system:

`void CFClearRegPin()`

Sets the REG pin to low so we can access Attribute Memory.

`void CFSetRegPin()`

Sets the REG pin to high so we can access ATA registers.

`u16 CFGetReg(int iOffset)`

Gets an ATA or Attribute Memory register at the offset iOffset.

`void CFSetReg(int iOffset, u16 val)`

Sets the value of an ATA or Attribute Memory register at the offset iOffset.

`void CFCardInit()`

Initializes the CompactFlash slot.

Always check the source code carefully to understand how it works and make sure it is the correct implementation for your system.

### 5.3.3 MMC/SD Card Driver

MMC/SD performance is greatly improved by doing multi-block reads/writes, so you should configure smxFs for a data cache size of about 16KB or more. Above this, the gain is not as significant.

Performance of bus mode usually should be faster than SPI, but we have found a lot of variance in this. Some on-chip bus mode controllers don't support multiple block mode; some are slow; and some have errata. If a key requirement of your system is high performance access to MMC/SD, you should test various processors. Also, from our testing, it seems that some SD cards do not support SPI mode, so you may be forced to use bus mode, if it is unacceptable to limit the brands/models of SD cards your users can use.

The MMC/SD card driver has been tested on various boards for SD bus mode and SPI mode. Check the configuration in `fdmsdio.h` for the list of tested environments. For your hardware you will have to write similar code to interface it to your SPI controller or host controller using the MMC/SD bus interface. Always check the code carefully to make sure it is correct for your system.

The driver has been designed to separate the higher-level code from the hardware interface code. You need to re-implement only **fdmsdio.c** or **add SPI interface function in fdmsdio\_spi.c**. Please refer to the data sheet for your CPU to learn how to implement the hardware interface. There is no standard for the SPI or bus mode controllers but we have tried to define a general API that can support them all. This API is defined in `fdmsdio.h`. Most differences between SPI and MMC/SD bus are handled in `fdmsdcmnd.c`.

The setting `MMCSA_SPI_BUS` is used to enable SPI mode or MMC/SD Bus mode. You need to set it in `fdmsdio.h`, according to your hardware design. You need to implement the SPI interface functions yourself unless you are using exactly the same hardware Micro Digital already ported it to.

The setting `MMCSA_INTERRUPT_MODE` is used to enable code for the SPI controller to generate an interrupt when the FIFO is full. However, the FIFO is small so this generates a lot of interrupts, so we feel it is better to leave it set to 0 for polled operation. For RM9200/SAM926x MMC/SD bus mode, we are using the onchip PDC (DMA) controller to transfer data and hook MCI interrupt to get the event that the transfer is done.

The setting `MMCSA_STREAMING_MODE` is used to control if the driver will access the data by single block mode or multiple block mode. Some MMC/SD controllers cannot support multiple block mode properly so we must access the data block by block.

The setting `MMCSA_4BIT_BUS` is used to control whether the MMC/SD controller will use 4-bit bus for MMC/SD card. Some MMC/SD controllers have a problem using 4-bit bus mode.

The setting `MMCSA_8BIT_BUS` is used to control if MMC/SD controller will use 8-bit bus for MMC card. Not all MMC/SD controllers support 8-bit bus mode.

The driver automatically detects whether an inserted card is MMC or SD during the Identification phase. This detail is only known to the driver. If the application needs to know if it is MMC or SD, add a custom IO control to the driver to get it.

### 5.3.4 NAND Flash Driver

The bulk of this driver is contained in files in the XFFS directory. Please refer to the `smxNAND User's Guide` for more information about this driver. You must implement the routines in `flhdw.c` for your flash hardware.

Emulation routines are provided to allow you to run this driver on a PC as a confidence test, before you implement the low-level code for your flash hardware. Uncomment the `_EMU` define in `XFFS\flashcnf.h` to do this. The emulator code is in `XFFS\EMU\flashemu.c`.

For 16-bit processors (e.g. x86 real mode), there is a problem if your flash chip's block size is  $\geq 64\text{KB}$  because of the 16-bit addressing. The driver caches full blocks including the spare area after each page, so a flash block size of 64KB would require a bigger cache block, which is not supported because the whole block cannot be addressed.

Although NAND flash is much faster than NOR, we recommend using `smxFLog` for continuous data logging. (See the discussion in section 5.3.5 NOR Flash Driver.) You can partition your flash to run `smxFS` in one area and `smxFFS` or `smxFLog` in another. See their user's guides for more information.

If you need to use multiple partitions, for example, two, you need to change the following configuration:

1. Change `NANDDISK_NUM` in `fdnand.h` to 2
2. Set each partition's start block index and size. -1 means to use the rest of the flash. For example, if the first partition size is 1024 blocks and the second partition will use the rest of the flash chip, the settings should be:

```
#define NAND0_START_BLOCK_INDEX (START_BLOCK_NUM)
#define NAND0_BLOCK_NUM      (1024)
#define NAND1_START_BLOCK_INDEX (START_BLOCK_NUM + NAND0_BLOCK_NUM)
```

```
#define NAND1_BLOCK_NUM    ((u32)-1)
```

If the file system used for the other partition is smxFFS it must use NAND flash driver 0 so smxFS can only register the NAND flash driver 1. That is, when registering the driver in smxFS, you must use driver 1, as follows: `sfs_devreg(sfs_GetNAND1Interface(), 0)`. Also, the NAND0 settings above must match the smxFFS settings `START_BLOCK_INDEX` and `PARTITION_BLOCK_NUM` in `XFFS\vfifcnf.h`.

### 5.3.5 NOR Flash Driver

The bulk of this driver is contained in files in the XFD directory. These files are part of smxNOR. Please refer to the smxNOR User's Guide for more information. You must implement the `nor_IO_routines` for your flash hardware, such as `nor_IO_SectorRead()`.

Emulation routines are provided to allow you to run this driver on a PC as a confidence test, before you implement the low-level code for your flash hardware. Uncomment the `_EMU` define in `XFD\fdcfg.h` to do this. The emulator code is in `XFD\norio.c`.

smxFS + NOR driver is not intended for frequent file operations. Even writing a small amount of data to a log file is a problem if it is done frequently, such as continuously every 5 seconds. The problem is that a DOS FAT filesystem is not well-suited to flash media because the data structures were not designed with flash limitations in mind. Changing one sector of data requires updating disk structures that must be moved to new blocks in the flash, and sometimes erasing those blocks first. Erasing NOR flash is very slow. It can take 1 sec to erase a flash block vs. 50 msec for NAND. For frequent and high performance filesystem operations, you should use NAND flash. For logging status data, we recommend smxFLog, which was designed for this purpose. You can partition your flash to run smxFS in one area and smxFLog in another. See the smxFLog User's Guide for more information.

### 5.3.6 RAM Disk Driver

The number of RAM disks is specified by `RAMDISK_NUM` in `fdram.h`, and the size of each is controlled by `RAMDISK_SIZE0` and 1 in `fdram.c`. Each RAM disk is automatically formatted when it is mounted because the `wAutoFormat` flag is set to 1 in its `SBD_DEVINFO` structure. See where this is done in `RAMIOctl()`. Note that if you are using battery-backed RAM, the RAM disk will not be auto-reformatted the next time you run. The mounting routine first checks to see if the disk is already formatted before it auto-formats it.

### 5.3.7 USB Disk Driver

The bulk of this driver is contained in the smxUSBH USB Host Stack mass storage driver in the XUSB directory. Please refer to the smxUSBH User's Guide for more information.

## A. File Summary

FILE	DESCRIPTION
smxfs.h	Main header file. Include in your application code. Includes all needed smxFS header files in the proper order.
fcfg.h	Configuration file for smxFS.
fintern.h	Internal main header file. Used only by smxFS files. It includes other header files in the proper order.
fconst.h	Internal constant value definitions.
fstruc.h	Internal data structure definitions.
fapi.c,h	Basic File I/O API functions such as sfs_fopen(), sfs_fclose().
fapiext.c	Extended File I/O API functions such as sfs_rename(). (Not Lite)
fcache.c,h	Data, FAT, and Directory cache related functions.
fchkdsk.c,h	sfs_chkdsk() and related functions. (Not Lite)
ffind.c,h	Functions used by sfs_findfirst() and sfs_findnext(). (Not Lite)
fformat.c,h	File system format related functions.
ffreeclus.c,h	Free cluster cache related functions. (Not Lite)
fmount.c,h	File system mount related functions.
fpath.c,h	Basic Directory Entry and FAT related functions.
fpathext.c,h	Extended directory related function such as rename(). (Not Lite)
fpathlfn.c,h	Long File Name related functions. (Not Lite)
fport.c,h	Porting files for compiler and hardware.
funicode.c,h, f*2uni.*, funi2*.*	Multiple language file name support functions and tables. (Not Lite)
fdata.c,h	ATA driver.
fdcf.c,h	CompactFlash driver.
fdmsd.c,h fdmsdio_xx.c,h fdmscmd.c,h	MMC/SD/SDHC card device driver (MMC/SD/SDHC bus and SPI mode).
fdnand.c,h ..\xffs\*.*	NAND flash driver.
fdnor.c,h ..\xfd\nor*.*	NOR flash driver.
fdram.c,h	RAM disk device driver.
fdusb.c,h	USB mass storage driver. Uses smxUSB.

## B. Multiple Language File Name Support

To support multiple language file names, smxFS needs to convert your language encoded file name to a Unicode file name. For example, Simplified Chinese uses GB2312 to represent Chinese characters, but the FAT filesystem use Unicode to save the file name. When you call `sfs_fopen()`, you pass a GB2312 Chinese string as the file name, so smxFS needs to convert this string to Unicode.

Two porting functions are provided for converting it to and from Unicode:

```
uint Unicode2String(u8 *string, u16 unicode);  
uint String2Unicode(u8 *string, u8 *unicode);
```

If you want to support a new language, such as Japanese or Korean, you need to implement these two functions. `Unicode2String()` converts a Unicode encoded string to your language string, `String2Unicode()` converts your language string to a Unicode encoded string.

## C. FAT Format

In order to use smxFS, it is not necessary to know the details of how the FAT filesystem is organized on the media. But if you're curious, see the Microsoft whitepaper *FAT32 File System Specification*. Search [www.microsoft.com](http://www.microsoft.com) for the title to find it quickly. Here is a brief overview.

### C.1 Main Regions

There are four main regions on a FAT disk:

1. Reserved
2. FAT
3. Root Directory (not for FAT32)
4. Data

The Reserved area contains the boot sector and BIOS Parameter Block (BPB) and possibly some additional sectors that are unused or possibly used by disk utilities.

The FAT (File Allocation Table) area indicates which clusters are in free or in use, and by what file. Each file is represented by a linked list of cluster numbers in the FAT. Each entry has the index of the next cluster of the file. The end cluster is marked by 0xF...FF. An entry with value 0 indicates a free cluster. The first cluster number is 2. A few other values at the high end (i.e. before 0xF...FF) are reserved. See the Microsoft whitepaper referenced above for more information.

The Root Directory is the list of files (and directories) in the top-level path. The size is fixed, so there is a limit on how many files can be in the root. This only exists for FAT12 and FAT16. For FAT32, the root directory is stored in the data area just like subdirectories and can grow to any size.

The Data area stores files and directories (which are just special files). It also stores the root directory for FAT32.

### C.2 Directories and Files

The root directory and subdirectories have the same format. Each is a table of information about the files on disk. Each directory entry indicates the file name, size, timestamp, and other characteristics. Directory entries are a fixed size. In order to support long file names, multiple directory entries are used for a single file. Otherwise, for the old 8.3 naming in DOS, each directory entry was for a different file.

One of the fields in the directory indicates the starting cluster number for the file. This is the head of the linked list of clusters. Each entry in the FAT gives the index of the next cluster of the file. The chain is terminated with 0xF...FF.

The root directory is a special area at the beginning of the disk (see above), and is only present in FAT12 and FAT16. A subdirectory is a file just like any data file, except that its contents are directory entries, and one bit in its own directory entry indicates that it is a directory. For a FAT32 disk, the root directory is a file just like a subdirectory.

## D. Size and Performance

### D.1 Code Size

Code size will vary depending upon CPU, compiler, and optimization level.

	<b>ARM Thumb-2</b>	<b>ARM</b>	<b>ColdFire</b>
	<u>IAR 6.10</u>	<u>IAR 6.10</u>	<u>CodeWarrior</u>
API and core files (Full <sup>1</sup> )	25 KB	42.0 KB	50.0 KB
API and core files (Lite <sup>2</sup> )	12 KB	17.0 KB	23.0 KB
FAT32_SUPPORT	1 KB	1.5 KB	2.0 KB
VFAT_SUPPORT	2.5 KB	4.0 KB	4.0 KB
CWD_SUPPORT	1 KB	1.0 KB	1.0 KB
FINDFIRST_SUPPORT	1 KB	2.0 KB	2.0 KB
MKDIR_SUPPORT	1.5 KB	2.0 KB	2.0 KB
VOLUME_SUPPORT	0.5 KB	1.0 KB	1.0 KB
PROPERTY_SUPPORT	0.5 KB	1.0 KB	1.0 KB
CHKDSK_SUPPORT	4 KB	6.5 KB	7.0 KB
RENAME_SUPPORT	1 KB	1.5 KB	1.5 KB
SAFETY_CHECK	0.5 KB	1.0 KB	1.0 KB
FORMAT_SUPPORT	2 KB	2.0 KB	3.5 KB
ATA driver	N/A	1.0 KB	1.5 KB
CompactFlash driver	N/A	1.0 KB	1.5 KB
MMC/SD/SDHC driver	4 KB (SD bus)	6.5 KB (SD bus)	7.5 KB (SPI)
NAND flash driver	N/A	14.0 KB	15.0 KB
NOR flash driver <sup>4</sup>	N/A	5.0 KB	7.5 KB
RAM disk driver	0.5 KB	0.5 KB	0.5 KB
USB disk driver <sup>3</sup>	18 KB	28 KB	34.0 KB

Notes:

1. Full version with nearly everything enabled. Unused API functions are dead-stripped by the linker, so the size used by your application is likely to be much smaller.
2. Lite version. SFS\_FULL\_FEATURES\_SUPPORT is set to "0". Lite plus indented options plus functions not marked "+" in the API Summary gives Full size.
3. The USB disk driver code is part of smxUSBH. The size can vary widely depending upon which Host Controller driver you are linking. The size above includes the ISP1362 driver. See the smxUSBH User's Guide for other sizes.
4. The NOR flash driver code is part of smxNOR. It includes two layers, STL and HIL. HIL is used to port to different hardware and flash chip so the size can vary widely. The above size is only for SPI mode STMicro M25P16 NOR flash chip.
5. Driver sizes vary a little because they have porting code that varies for different hardware.

### D.2 Data Size (RAM Requirement)

RAM is allocated to cache data sectors, portions of the FAT table, and directory entries (for the root directory and subdirectories). The cache size depends upon the Sector size. For most disks, the sector size

is 512 bytes, but smxFS supports other sector sizes such as 1024 bytes. You can adjust the cache settings in fcfg.h. The default setting is 44 sectors total for all the three caches. If the sector size is 512 bytes then the cache is 22KB RAM. Additional 1.0 KB of RAM is needed for the long file name buffer and some global variables. Each file needs 1KB RAM for the data buffer and file handle. Stack size is about 1KB. A total of 25KB RAM is required to open one file and do some file access. If your system has limited RAM, you can reduce all the cache settings to 1 sector, and then 1.5KB is enough for the cache and 4 KB to open one file to do basic file operations such as fread() and fwrite(). But, this minimal setting will greatly decrease the performance of smxFS. We tested and found that performance is only 20% as fast for 1.5 KB cache as the default 22KB cache.

RAM is also needed for the stack. For a single task (or non-multitasking), about 600 bytes are needed. Also for the USB disk driver, another 400 bytes are needed.

Current Working Directory also needs a buffer. Each CWD needs 264 bytes of RAM. This RAM is only allocated from the heap when you call sfs\_setcwd();

sfs\_chkdisk() needs additional RAM and static data. See section 3.7 Memory Management for details.

The filesystem and drivers also have some static data. The following is a summary. Sizes are shown only for one processor/compiler because they should not vary much for others.

	<u>Static Data</u>
smxFS API and core files	76 B
smxFS check disk utility	352 B
ATA	36 B
CompactFlash	36 B
MMC/SD/SDHC	825 B
NAND Flash	28 B
NOR Flash	700 B
RAM Disk	disk size
USB Flash Disk	0

## D.3 Performance

### D.3.1 Performance for Various Drivers

#### D.3.1.1 USB disk

The following is a table for performance testing of smxFS and the **USB flash disk driver**. smxFS reads/writes a big file whose size is 20MB from/to a USB flash disk. smxFS is configured to use 22KB RAM as cache. The table also records the read/write speed of 20MB raw flash disk data. Comparing these two speeds shows that smxFS's overhead is very small.

\*The hardware environment for this testing is:

Celeron 300MHz CPU; 32MB 100M SDRAM; PC motherboard; Host Controller connects to System by 33MHz PCI bus.

\*\*Flash Disk is Lexar JumpDrive USB 2.0 512MB

<u>NEC EHCI Host Controller</u>	<u>Reading</u>	<u>Writing</u>
USB driver raw data	12684 KB/s	8320 KB/s
USB driver and smxFS	10556 KB/s	7787 KB/s
<u>NEC OHCI Host Controller</u>	<u>Reading</u>	<u>Writing</u>
USB driver raw data	891 KB/s	832 KB/s
USB driver and smxFS	885 KB/s	817 KB/s
<u>VIA UHCI Host Controller</u>	<u>Reading</u>	<u>Writing</u>
USB driver raw data	639 KB/s	611 KB/s
USB driver and smxFS	611 KB/s	590 KB/s
<u>ISP116x Host Controller (ISA)</u>	<u>Reading</u>	<u>Writing</u>
USB driver raw data	352 KB/s	334 KB/s
USB driver and smxFS	336 KB/s	328 KB/s
<u>ISP1362 Host Controller</u>	<u>Reading</u>	<u>Writing</u>
USB driver raw data	621 KB/s	493 KB/s
USB driver and smxFS	591 KB/s	476 KB/s
<u>ISP176x Host Controller</u>	<u>Reading</u>	<u>Writing</u>
USB driver raw data	7425 KB/s	3214 KB/s
USB driver and smxFS	7023 KB/s	3072 KB/s

The following is a table for performance testing of smxFS and the **USB flash disk driver** on some ARM chip by using the embedded OHCI controller. Flash Disk is SanDisk USB 2.0 512MB

<u>Atmel SAM9260 OHCI Host Controller (AHB is 105 MHz)</u>	<u>Reading</u>	<u>Writing</u>
USB driver and smxFS	555 KB/s	505 KB/s
<u>Atmel SAM9261 OHCI Host Controller (AHB is 60 MHz)</u>	<u>Reading</u>	<u>Writing</u>
USB driver and smxFS	458 KB/s	414 KB/s
<u>Cirrus Logic EP9315 OHCI Host Controller</u>	<u>Reading</u>	<u>Writing</u>
USB driver and smxFS	575 KB/s	498 KB/s

The following is a table for performance testing of EHCI controller and USB hard disk.

\*The hardware environment for this testing is:

Celeron 300MHz CPU; 32MB 100M SDRAM; PC motherboard; Host Controller connects to System by 33MHz PCI bus.

\*\* Disk is LACIE USB 2.0 40GB

<u>VIA EHCI Host Controller</u>	<u>Reading</u>	<u>Writing</u>
USB driver raw data	24966 KB/s	19784 KB/s
USB driver and smxFS	20078 KB/s	16786 KB/s

### D.3.1.2 MMC/SD/SDHC card

The following is a table for performance testing of the **MMC/SD/SDHC (SPI) driver**. The test is the same as for the USB driver, above. The testing was done on an Avnet 5282 Eval Board using a memory add-on card we developed and the IAR STR912 KickStart board.

For the Avnet 5282 board, the QSPI clock was set to 16MHz and the driver was set to poll rather than use interrupts, because the QSPI FIFO is only 16 bytes, so it interrupted often (125K times per second). Consider the number below to be a best-case time. In a multitasking environment, drivers should be used in interrupt-driven mode to allow tasks to run, so if the CPU cannot handle the high interrupt rate, the QSPI should be run at a slower clock rate. Of course, your hardware may be totally different — it may not have QSPI or it may use DMA or have other characteristics, so this discussion is only given for reference.

SanDisk 256MB MMC/SD driver and smxFS	<u>Reading</u> 399 KB/s	<u>Writing</u> 441 KB/s
SanDisk UltraII 512M MMC/SD driver and smxFS	<u>Reading</u> 397 KB/s	<u>Writing</u> 446 KB/s
SanDisk 1GB MMC/SD driver and smxFS	<u>Reading</u> 398 KB/s	<u>Writing</u> 446 KB/s
SanDisk Extreme III 2GB MMC/SD driver and smxFS	<u>Reading</u> 398 KB/s	<u>Writing</u> 454 KB/s

For the STR912 KickStart board, the SSP clock was set to 24MHz. The SSP FIFO is only 8 bytes so the driver was set to poll rather than use interrupts.

SanDisk 256MB MMC/SD/SDHC driver and smxFS	<u>Reading</u> 1067 KB/s	<u>Writing</u> 1166 KB/s
SanDisk UltraII 512M MMC/SD/SDHC driver and smxFS	<u>Reading</u> 1003 KB/s	<u>Writing</u> 1153 KB/s
SanDisk 1GB MMC/SD/SDHC driver and smxFS	<u>Reading</u> 1015 KB/s	<u>Writing</u> 1153 KB/s
SanDisk Extreme III 2GB MMC/SD/SDHC driver and smxFS	<u>Reading</u> 1080 KB/s	<u>Writing</u> 1210 KB/s
SanDisk SDHC 8GB Class 2 MMC/SD/SDHC driver and smxFS	<u>Reading</u> 1192 KB/s	<u>Writing</u> 1023 KB/s

The following is a table for performance testing of the **MMC/SD/SDHC Bus driver**. The test is the same as for the USB driver, above. The testing was done on AT91SAM9RL64-EK Board.

SanDisk 256MB MMC/SD/SDHC driver and smxFS	<u>Reading</u> 5580 KB/s	<u>Writing</u> 2329 KB/s
---	-----------------------------	-----------------------------

SanDisk UltraII 512M	<u>Reading</u>	<u>Writing</u>
MMC/SD/SDHC driver raw data	6554 KB/s	5650 KB/s
MMC/SD/SDHC driver and smxFS	5094 KB/s	3696 KB/s
SanDisk 1GB	<u>Reading</u>	<u>Writing</u>
MMC/SD/SDHC driver and smxFS	5251 KB/s	3335 KB/s
SanDisk 2GB	<u>Reading</u>	<u>Writing</u>
MMC/SD/SDHC driver raw data	10240 KB/s	3724 KB/s
MMC/SD/SDHC driver and smxFS	5132 KB/s	2348 KB/s
SanDisk Extreme III 2GB	<u>Reading</u>	<u>Writing</u>
MMC/SD/SDHC driver and smxFS	5389 KB/s	4471 KB/s
SanDisk SDHC 8GB Class 2	<u>Reading</u>	<u>Writing</u>
MMC/SD/SDHC driver and smxFS	5120 KB/s	2417 KB/s

### D.3.1.3 CompactFlash card

The following is a table for performance testing of the **CompactFlash driver**. The test is the same as for the USB driver. The testing was done on a M5485EVB using LogicPD CPLD code and M5329EVB, which supports only memory mapped mode CompactFlash.

CF driver and smxFS	<u>Reading</u>	<u>Writing</u>
M5485EVB	919 KB/s	695 KB/s
M5329EVB	1258 KB/s	1090 KB/s

### D.3.1.4 NAND Flash

The following is a table for performance testing of the **NAND Flash driver**. The test is the same as for the USB driver. The test was done on Atmel AT91SAM9263-EK using Samsung K9F2G08U0M using DMA transfers

NAND Flash driver and smxFS	<u>Reading</u>	<u>Writing</u>
Samsung K9F2F08U0M	5953 KB/s	2852 KB/s

Note that performance is better for larger cluster sizes, so you will see different results for the same media depending upon how it is formatted.

### D.3.1.5 NOR Flash

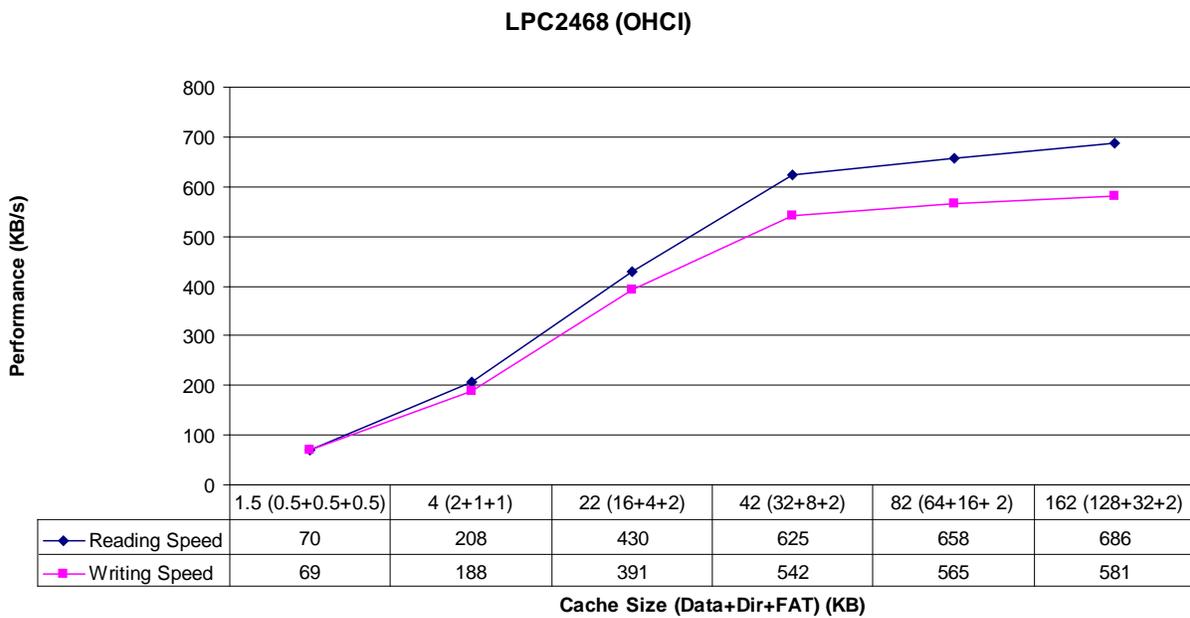
The following is a table for performance testing of the **NOR Flash driver**. The test is the same as for the USB driver but the file size is only 1MB. The tests were done on a Freescale M5485EVB using Intel 28F128K3 StrataFlash and on an Avnet 5282 board using SPI mode STMicro M25P16 serial flash.

NOR Flash driver and smxFS	<u>Reading</u>	<u>Writing</u>
Intel 28F128K3 StrataFlash	950 KB/s	110 KB/s
STMicro M25P16 serial flash	150 KB/s	30 KB/s

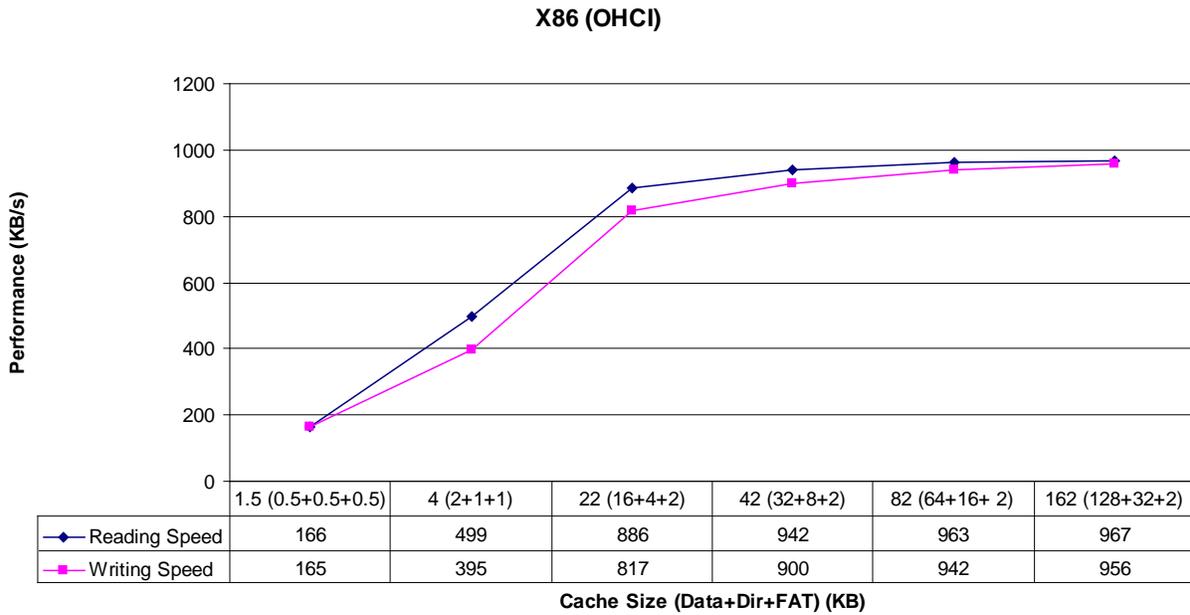
### D.3.2 Cache Size vs. Performance

The following graphs show the performance of a USB flash disk reading/writing operation when the cache sizes are set to different values. These are intended to help you see the tradeoff between RAM usage and performance, especially if you are using an SoC with only internal memory, so you can decide how much of the precious SRAM to designate for this purpose. Keep in mind that actual performance values will vary a lot depending upon the media you are writing to. For example, USB flash disks have software overhead from the USB stack and overhead due to the controller built into the USB flash disk that handles wear leveling, garbage collection, etc behind the scenes. Similarly, SD and CompactFlash cards have similar controllers built in. Probably the best media to test would be ATA hard drives, but these cannot be connected to the ARM and ColdFire eval boards.

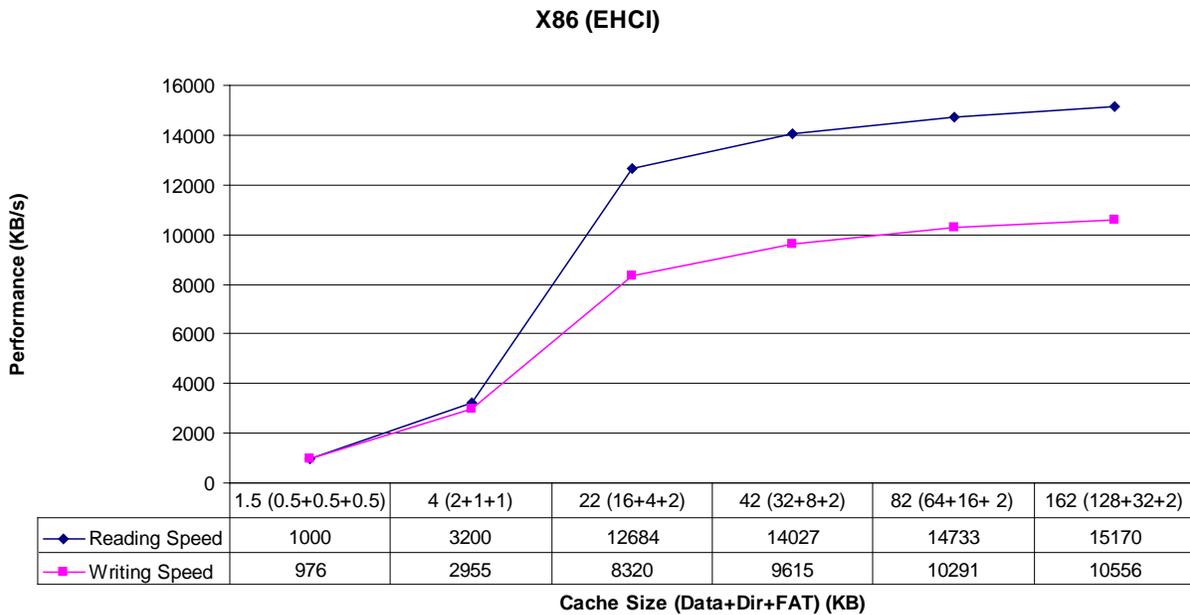
LPC2468EA OEM Board 72MHz (USB OHCI):



x86 Motherboard 300MHz (USB OHCI):



x86 Motherboard 300MHz (USB EHCI):



## **E. Tested Hardware**

### **E.1 CompactFlash Devices**

- Kingston 64MB
- PNY 1GB
- SanDisk Extreme III 2GB

### **E.2 MMC/SD Devices**

- Adata Speedy 2GB SD (MMC/SD/SDHC and SPI bus)
- Kingston 512MB SD (MMC/SD/SDHC and SPI bus)
- Kingston 2GB SD (MMC/SD/SDHC and SPI bus)
- Kingston 8GB SDHC Class6 (MMC/SD/SDHC and SPI bus)
- PQI Hi-Speed 60 2GB SD (MMC/SD/SDHC and SPI bus)
- SanDisk 32MB MMC (MMC/SD/SDHC bus only)
- SanDisk 64MB MMC (MMC/SD/SDHC and SPI bus)
- SanDisk 256MB SD (MMC/SD/SDHC and SPI bus)
- SanDisk 1GB SD (MMC/SD/SDHC and SPI bus)
- SanDisk 8GB SDHC Class2 (MMC/SD/SDHC and SPI bus)
- SanDisk Ultra II 512MB SD (MMC/SD/SDHC and SPI bus)
- SanDisk Extreme III 2GB SD (MMC/SD/SDHC and SPI bus)
- Transcend 2GB SD (MMC/SD/SDHC and SPI bus)
- Transcend 4GB SDHC Class 6 (MMC/SD/SDHC and SPI bus)

### **E.3 NAND Flash Devices**

- Micron 29F2G08A on Atmel AT91SAM9RL64-EK/ AT91SAM9261-EK board
- Samsung K9F1G08U on Embedded Artists LPC2468 OEM board.
- Samsung K9F2G080U on Atmel AT91SAM9260-EK/ AT91SAM9263-EK/ AT91SAM9XE-EK
- Samsung K9F2808U on our Avnet Coldfire 5282 add-on board.

### **E.4 NOR Flash Devices**

- Intel 28F128K3, 28F256K3, 28F128J3D on MCF5485EVB board.
- SST 39VF320 on Embedded Artists LPC2468 OEM board.
- STMicro M25P16 (2MB)

### **E.5 USB Mass Storage Devices**

Tested by Micro Digital:

- ADISK USB 1.1 32MB flash disk
- Aigo USB 1.1 64M flash disk
- Crucial 1G flash disk
- FPT-D US5B2H01 18-in-1 USB card reader/writer

- HP 1G flash disk
- IBM Portable Diskette Drive (floppy drive)
- Integral USB 2.0 2GB flash disk
- Kingston DataTraveler 1GB flash disk
- Kingston DataTraveler 100 2GB flash disk
- Kingston DataTraveler 16GB flash disk
- LACIE USB 2.0 40GB mobile hard drive
- Lexar Media JumpDrive Secure USB 2.0 512MB flash disk
- Memorex 2GB flash disk
- NCP XDrivePlus MMC/SD reader
- Newman USB 1.1 64MB flash disk
- PNY Attache USB 1.1 64MB flash disk
- PNY Attache (U3) 1GB
- PNY Attache 2GB
- PNY Attache 8GB
- PQI MMC/SD reader
- RedLeaf USB 2.0 256MB flash disk
- SanDisk Cruzer USB 2.0 256MB flash disk
- SanDisk Cruzer Micro 2GB flash disk
- SanDisk Cruzer Micro (U3) 2GB flash disk
- SanDisk Cruzer Micro (U3) 4GB flash disk
- SONY MICROVAULT USM256U2 USB 2.0 256MB flash disk
- Transcend JF V30 4GB flash disk

Tested by others:

- Edge DiskGO™ 1GB USB Flash Drive Enhanced for ReadyBoost™
- Edge DiskGO™ 2GB USB Flash Drive Enhanced for ReadyBoost™
- Imation 1GB Swivel USB Flash Drive
- Imation 2GB Swivel USB Flash Drive
- Integral 1GB USB Memory Stick
- MARKEM 1GB USB Memory Stick
- Memorex 1GB TravelDrive™ USB Flash Drive
- Memorex 2GB TravelDrive™ USB Flash Drive
- PNY 1GB Attache USB Flash Drive
- SanDisk 2GB Cruzer® Crossfire USB Flash Drive
- SanDisk 512MB Cruzer® Micro USB Flash Drive
- SanDisk 2GB Cruzer® Micro USB Flash Drive
- SanDisk 4GB Cruzer® Micro USB Flash Drive (U3 function not initialized)
- Sony 512MB Micro Vault Tiny USB Flash Drive
- Sony 2GB Micro Vault Tiny USB Flash Drive
- Sony 1GB Micro Vault Classic USB Flash Drive
- Sony 4GB Micro Vault Classic USB Flash Drive
- X Digital Media 1GB Itty Bit USB Flash Drive
- X Digital Media 1GB Poker Chip USB Flash Drive
- X Digital Media 2GB Itty Bit USB Flash Drive

## F. Glossary

### **cluster**

The minimum allocation unit on a disk. It is some integral number of sectors. The reason this is necessary is because large media have too many sectors to manage individually. The FAT would have to be enormous to map each sector. Instead it maps clusters. The down-side is that even if a file is only 1 byte in size, it still needs a whole cluster, so the extra sectors are wasted. See the section 3.3 FAT12/16/32 for discussion of how the cluster size is determined.

### **disk**

In this manual, “disk” and “media” are used interchangeably. Since smxFS focuses on supporting flash memory devices, the term “media” is correct, but sometimes, it is clearer in the text to use “disk”.

### **DOS-compatible file system**

See FAT file system.

### **drive**

A device or socket that contains media or that media can be plugged into. See also socket.

### **EOF**

End of File. For some filesystems there is an EOF character, but not for smxFS. For smxFS, EOF means that the file pointer points to the next byte following the last byte of the file. That is, file pointer == file size.

### **FAT**

File Allocation Table. This is the data structure used to map the clusters on the disk that are used by each file. It is a simple singly-linked list data structure, allowing files to grow to any length and clusters to be non-contiguous. The FAT file system is named for this data structure.

### **FAT file system**

The file system developed by Microsoft for DOS, which has been extended for 32-bit versions of Windows to support FAT32 for large media and long file names. Media which are formatted for FAT12 or FAT16 are compatible with DOS. All FAT media are compatible with Windows. There are many other types of file systems, which each have their own unique way of organizing data on the media. smxFS could have been implemented to be compatible with one of these, but FAT is a good choice for embedded systems because it is relatively simple, works well, and is widely supported so media can be interchanged between systems.

### **file handle**

A unique ID assigned to an open file. This is used in subsequent API calls that operate on files to specify to operate on this file. In some file systems, it might be an integer, but in smxFS, it is a pointer to a FILESTRUCT structure. This structure holds information about the file such as its current file pointer.

### **file pointer**

The current index into the file. When a file is opened, the file pointer starts at 0. When data is read or written, the file pointer is advanced to the index of the next byte following what was read or written. The file pointer can be forced to a new location with `sfs_fseek()`.

**LFN**

Long file name. The method used in smxFS is the method Microsoft uses in its Win32 operating systems, known as VFAT. Before VFAT, many people developed ways to make long file names, to avoid the DOS 8.3 limitation and these could be used, except then the media would not be interchangeable with Windows. VFAT is patented by Microsoft. See the discussion of configuration option SFS\_VFAT\_SUPPORT.

**media**

See disk.

**mount**

Initialize any data structures and do any necessary operations necessary before a disk can be accessed. In smxFS, this consists of registering a device with sfs\_devreg() and calling MountDevice(). (MountDevice() is called internally.)

**partition**

A logical division of a disk or media into different volumes. (See volume.) See the discussion of multiple partitions in section 3. Theory of Operation.

**sector**

The smallest writeable unit on a disk, usually 512 bytes, but can vary for different devices. Sector size <= cluster size.

**socket**

Analogous to a disk drive, for memory-based media. For example, a USB port, or MMC/SD card socket. Typically an embedded system will have only 1 socket for each type of removable media.

**volume**

A complete FAT filesystem on the media. In the simplest case, there is one volume on a disk. When a disk is partitioned, each partition is a volume. Each volume is assigned a unique drive letter.