Micro Digital

smxFFS™ User's Guide

Flash File System

Version 2.11
October 8, 2021

by Yingbo Hu

Table of Contents

© Copyright 2011-2021

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

All rights reserved.

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

1. Overview

Note: v2 is a complete redesign from v1. There is no compatibility between versions. See section 1.4 Version 2 for more information.

smxFFS is a power fail safe flash file system. Unlike a FAT file system, there is no FAT area for the smxFFS flash file system, so if power fails during a file operation, only those files that are not closed may lose data. Other files and the file system itself will not get damaged.

smxFFS is reentrant (multitasking safe) and requires minimal RAM and ROM (only 4KB RAM + 2KB RAM for each open file (for 512 byte sector size) and 20KB code). Unlike the old version of smxFFS, there is no large FAT table to store in RAM, and RAM usage does not increase with flash memory size.

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

smxFFS consists of these components:

1. **FFS API** provides the standard C library API: fopen(), fread(), fwrite(), fseek(), fclose(), etc. to the application.
2. **FFS Path** implements the FCB structure handler.
3. **FFS Mount/Format** implements the mount/format functions for the flash devices.
4. **FFS Cache** implements the cache functionality for the disk's free clusters.
5. **FFS Driver Interface** uses a unique interface to integrate all the devices into the file system.
6. **FFS Port** implements the OS and compiler-related definitions, macros, and functions.

1.1 Features

A primary goal of the new design was to greatly reduce RAM usage, while supporting very large flash devices up to 256TB. By default it is configured to support up to 32GB but can be easily changed to support larger disks. See section 3.3.1 Supporting Flash Larger than 32GB. The old design was developed at a time when flash memories were small, typically 8 or 16MB. As they grew, the RAM needs grew substantially, making it inappropriate for small SoCs. Another main goal was to support NOR flash in addition to NAND. The following list summarizes smxFFS features:

- Works with NAND, NOR or any block device which can guarantee data consistency within each sector.
- Flash disks up to 256TB.
- Standard C library APIs for most common file operations.
- Subdirectory support (limited to 3 levels of nesting and 254 files per directory)
- Power fail safe.
- Small:
    - RAM: 4KB for the file system. Each open file needs an additional 2KB (when sector size is 512 bytes).
    - ROM: 20 KB for the complete API.
- Can share flash with smxFS, smxFLog, boot code, and application code.
- Supports multiple disks, like smxFS does.

1.2 Limitations

In order to achieve the primary goals discussed in the previous section, it was necessary to put some restrictions on the capabilities of smxFFS.

- Maximum file length is 4GB-2.
- Maximum file name length and number of files per directory is specified at compile time.
- Moderate performance.
- Data cluster size is not a power of 2 such as 1024 or 2048 since some metadata is written there not to the spare area, which does not exist on NOR flash.
- Subdirectories
    - Maximum nesting is 3 subdirectories, for example, A:\\subdir1\\subdir2\\subdir3\\file.
    - Each directory, including the root directory, can only have 254 or fewer files. Smaller flash may only support fewer than 254 files. This is set in a pre-compiled configuration table.
    - There is no current directory, so the full path of a file must be specified each time a file is opened, sff_fopen("A:\\subdir1\\subdir2\\subdir3\\file", "rb")

1.3 Overhead

Not all the space in a cluster will be used to store file data. The first 8 bytes of each cluster is reserved by the file system, so the whole disk has some overhead. Overhead depends on the cluster size. For 8KB clusters, the overhead is only 0.1%.

1.4 Version 2

This manual documents smxFFS v2, which is a complete redesign from v1. There is no compatibility between versions. The new version starts numbering at v2.00. There is a little confusion here because the old design reached v2.00 when it was modified for SMX v4. However, references to v2 indicate the new design. They can be easily distinguished with a preprocessor conditional, as discussed next.

For users using v1 who are sharing application code for a new project that uses v2, the following check can be used to preserve the old v1 case:

```
#if defined(SFF_VERSION)
/* new code for v2 */
#else
/* old code for v1 */
#endif
```

This is because SFF_VERSION is new; it was FFS_VERSION for v1. It was renamed to match the prefixing convention used in all SMX code.

Also, in the master preinclude file or makefile, the main build conditional SMXFFS2 was added to select v2, which was necessary because the name of the main header file changed to be consistent with other SMX modules. Eventually, the old smxFFS will be eliminated.

2. Using smxFFS

2.1 Installation

smxFFS is installed by copying files from the distribution media. When ordered with the SMX® RTOS, it is part of the SMX release and is installed with it.

2.2 Getting Started

smxFFS is configured to support any environment. To support a compiler which is not in our porting file, see Appendix B. Porting Notes, and implement the porting layer for your environment first, before using smxFFS.

You may erase the flash first if it contains any pre-loaded image or data. After you implement your low level NAND or NOR flash driver, use the code provided in nandtest.c (for NAND), nortest.c (for NOR), or flltest.c (both) to verify your driver first. Please see section 3.2 nandio.c in the smxNAND User's Guide or section 4.3 Verify the Driver in the smxNOR User's Guide for details.

2.3 Basic Terms

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 File Node 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.
Disk	In this manual, "disk" and "media" are used interchangeably. Since smxFFS focuses on supporting flash memory devices, the term "media" is correct, but sometimes, it is clearer in the text to use "disk".
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 smxFFS, it is a pointer to a SFF_FILE 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 sff_fseek().
Media	See disk.

2.4 Configuration Settings

If any settings are changed, it is necessary to rebuild the smxFFS library, clean.

2.4.1 ffcfg.h

ffcfg.h contains flash file system configuration constants that allow selecting features and tuning performance, code size, and RAM usage.

SFF_MAX_DEV_NUM

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

SFF_DRV_

These specify which of the smxFFS 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. smxFFS requires the driver to guarantee the data is consistent within each sector. Our smxNAND and smxNOR drivers meets this requirement.

SFS_READONLY

If set to 1, smxFFS becomes a read-only filesystem. All the API functions to modify the contents of the disks are omitted, such as sff_fwrite(), sff_ftruncate(), sff_rename(), sff_mkdir(), sff_rmdir(). sff_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 smxFFS, so they don’t include ffcfg.h. Also, they might be shared by smxUSBD. This is why they have separate defines instead of checking SFF_READONLY. Also set SFD_READONLY (XFD\fdcfg.h.

SFF_PATHSEP

Set the path separator character as desired.

SFF_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 SFF_FIRST_DRIVE. See the section 3.1.1 Drive Lettering for more information.

SFF_FREECLUS_CACHE_SIZE

The free cluster cache size. smxFFS has an internal cache to hold the free clusters of the disk. This is the size of this cache.

SFF_FREECLUS_SCAN_NUM

When the file system cannot find enough free clusters in the free cluster cache, it needs to scan the disk to find more. This setting indicates how many clusters the file system will scan each time to find more free clusters.

SFF_FILENAME_LEN

This is the file name size. When you declare a buffer for a file name, use SFF_FILENAME_LEN + 1. File name does not include any directory name and disk letter.

Please note, whenever you change this, you must also change the FCB structure. Doing so requires you to reformat the disk.

SFF_FULLPATHNAME_LEN

This is the full path name size. It includes all the directory names and disk letter. When you declare the buffer for the path name, use SFF_FULLPATHNAME_LEN + 1. You should always use the full path name when call smxFFS APIs.
Please note, smxFFS only supports up to 3 levels of subdirectories.

SFF_SAFETY_CHECKS

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

2.4.2 ffport.h

smxFFS’s porting layer maps onto smxBase services, for general purpose compiler and OS definitions. See the smxBase User’s Guide for more information.

2.5 Using the API

smxFFS 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 smxFFS 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 SFF_FIRST_DRIVE is ’A’. See the section 3.1.1 Drive Lettering for more information. The lines that register the drivers assume that you have enabled these drivers in ffcfg.h. Also see demo.c or ffdemo.c for more example code.

```
#include "smxbase.h"   /* Porting layer. Includes OS and other header files. */
#include "smxffs.h"    /* smxFFS API header file */

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

   if(sff_init() == SB_PASS)                        /* initialize smxFFS */
   {
      /* Register device drivers. */
      sff_devreg(sff_GetNANDInterface(), 0);        /* A: */
      ...

      /* Do basic file operations. (Should normally check return values.) */
      fh = sff_fopen("A:\\testfile.bin", "w+b");    /* open file */
      sff_fwrite(pData, 100, 1, fh);                /* write some data */
      sff_fseek(fh, 0, SFF_SEEK_SET);               /* rewind to the beginning */
      sff_fread(pData, 100, 1, fh);                 /* read it back */
      sff_fclose(fh);                               /* close file */
   }
}
```

3. Theory of Operation

3.1 Device Drivers

The following is basic information about using device drivers with smxFFS.

3.1.1 Drive Lettering

Drive lettering is simple. It is determined by:

DeviceID + SFF_FIRST_DRIVE

DeviceID is the ID value passed to sff_devreg(), and SFF_FIRST_DRIVE is a letter defined in ffcfg.h, which is 'A' by default.

3.1.2 Registering a Driver

The built-in device drivers supported by smxFFS are registered by smxffs_init() in SMX's initmods.c. For non-SMX systems, call sff_devreg(). See the example in the sff_devreg() call description in section 4. File System API in this manual. Note that the number of drivers that may be registered simultaneously is controlled by SFF_MAX_DEV_NUM in ffcfg.h.

3.1.3 Available Drivers

- NAND flash
- NOR flash

3.2 Rules

3.2.1 File Names

The maximum length of a file name is defined as SFF_FILENAME_LEN in ffcfg.h. File names are case sensitive, so File1 and file1 are two different files for smxFFS. smxFFS does not impose limitations on special characters used in file names. However, it is recommended to avoid use of ? and *, to avoid ambiguity in calls to sff_findfirst() and sff_findnext(). For example, if files existed named file1.txt and file?.txt, findfirst/next searches for file?.txt would return both of these files, not just file?.txt.

3.2.2 Timestamps

Timestamps are like the FAT file system; year is relative to 1980. This could be changed by editing sb_GetLocalTime() in XBASE\bbase.c (or SFF_GET_LOCAL_TIME() in older versions of smxFFS).

3.3 Application Notes

3.3.1 Supporting Flash Larger than 32GB

By default, smxFFS is configured to support up to 32GB flash disks. If you need to support even larger disks, just add items to the array SecPerClus[] in ffmount.c. Sector size should match the flash chip; normally it is the same as page size. The following example adds a line for up to 256GB flash disk support.

```
typedef struct
{
    u32  DiskSize; /* in sectors; DiskSize*BytesPerSector = MediaSize (in bytes) */
    u8   SecPerClusVal;  /* sectors per cluster */
    u8   MaxFileNum;
    u16  SectorSize;
} SFF_SECPERCLUSTABLE;

STATIC const SFF_SECPERCLUSTABLE  SecPerClus[] =
{
       {        256, 2,  7,  256},          /* less than 64K */
       {       1024, 2, 15,  512},          /* 512K */
       {       2048, 2, 31,  512},          /* 1M */
       {       4096, 4, 31,  512},          /* 2M */
       {       8192, 4, 63,  512},          /* 4M */
       {      16384, 8, 63,  512},          /* 8M */
       {     524288, 8,127,  512},          /* 256M */
       {    4194304,16,254,  512},          /* 2G, SD/USB */
       { 67108864,32,254,  512},            /* 32G, SD/USB */
       {       4096, 2, 31, 1024},          /* 4M, internal NOR flash */
       {     131072, 8,127, 2048},          /* 256M */
       {     524288, 8,254, 2048},          /* 1G */
       {    4194304,16,254, 2048},          /* 8G */
       { 33554432,16,254, 8192},            /* 256G */
       {0xFFFFFFFFL, 0,  0,    0}           /* more than 256G */
};
```

3.3.2 Changing the Max File Number

smxFFS can support up to 254 files/subdirectories in each directory, but for smaller flash disks, we may choose to use smaller file names. The Max File Number is also controlled by the above array SecPerClus[] in ffmount. For example, if you need to support more than 15 files, say 63 files, for 512KB flash disk, change the corresponding line to

```
       {       1024, 2, 63,  512},          /* 512K */
```

3.3.3 Improving Read/Write Performance

When you call sff_fread()/sff_fwrite(), passing a buffer that is exactly the cluster size can get better performance than reading/writing one byte. smxFFS provides sff_clustersize() to tell the application the cluster size of the current disk. Here is an example showing use of this function:

```
int testPerformance(uint iParameter)
{
    SFF_FILEHANDLE fHdl;
    int i;
    char *pData;
    u32 filelen;
    u32 iBufSize = sff_clustersize(0);
    pData = (char *)malloc(iBufSize);
    if(pData)
    {
        filelen = sff_filelength("A:\\sffstest.bin");
        fHdl = sff_fopen("A:\\sffstest.bin", "rb");
        if(fHdl)
        {
            for(i = 0; i < filelen/iBufSize; i++)
            {
                if(0 == sff_fread(pData, iBufSize, 1, fHdl))
                    break;
            }
            sff_fclose(fHdl);
        }
        free(pData);
    }
    return 0;
}
```

4. File System API

The smxFFS 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 sff_ prefix gives these their own namespace, and makes it easy to search for calls to this library. 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.

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: They are case-sensitive when creating a file. See the earlier section 3.2.1 File Names for more information.

4.1 API Data Types

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

SFF_FILEHANDLE	Pointer to a SFF_FILE 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.
SFF_FILEINFO	Structure containing various information about a file found with sff_findfirst() or sff_findnext().
SFF_FINDHANDLE	Structure containing various information about a session of sff_findfirst()/sff_findnext().
SBD_IF	Pointer to a structure of pointers to the driver interface functions. Defined in smxBase header file bbd.h
u8, u32, etc.	Unsigned integer types of the size (bits) indicated. Defined in smxBase header file bdef.h

4.2 API Summary

int	**sff_init**(void)
void	**sff_exit**(void)
int	**sff_devreg**(const SBD_IF *dev_if, uint nID)
int	**sff_devunreg**(uint nID)
int	**sff_devstatus**(uint nID)
unsigned long	**sff_freekb**(uint nID)
unsigned long	**sff_totalkb**(uint nID)
int	**sff_ioctl**(uint nID, uint command, void * par)
int	**sff_getlasterror**(uint nID);
SFF_FILEHANDLE	**sff_fopen**(const char *filename, const char *mode)
int	**sff_fclose**(SFF_FILEHANDLE filehandle)
size_t	**sff_fread**(void * buf, size_t size, size_t items, SFF_FILEHANDLE filehandle)
size_t	**sff_fwrite**(void * buf, size_t size, size_t items, SFF_FILEHANDLE filehandle)
int	**sff_fseek**(SFF_FILEHANDLE filehandle, long lOffset, int nMethod)
int	**sff_fflush**(SFF_FILEHANDLE filehandle)
int	**sff_feof**(SFF_FILEHANDLE filehandle)
void	**sff_rewind**(SFF_FILEHANDLE filehandle)
long	**sff_ftell**(SFF_FILEHANDLE filehandle)
void	**sff_ftruncate**(SFF_FILEHANDLE filehandle)
int	**sff_fdelete**(const char * filename)
unsigned long	**sff_filelength**(const char *filename)
int	**sff_findfile**(const char *filename)
int	**sff_mkdir**(const char *path)
int	**sff_rmdir**(const char *path)
int	**sff_setcwd**(const char *path)
char*	**sff_getcwd**(char * buffer, int maxlen)
int	**sff_chkdsk**(uint nID)
unsigned long	**sff_clustersize**(uint nID);
int	**sff_gettimestamp**(const char * filename, DATETIME* datetime)
int	**sff_timestamp**(const char * filename, DATETIME* datetime)
int	**sff_rename**(const char * oldname, const char * newname)
SFF_FINDHANDLE	**sff_findfirst**(const char * filespec, SFF_FILEINFO* fileinfo)
int	**sff_findnext**(SFF_FINDHANDLE handle, SFF_FILEINFO* fileinfo)
int	**sff_findclose**(SFF_FINDHANDLE handle)

4.3 API Reference

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

int **sff_chkdsk** (uint nID)

Summary Checks and/or fixes problems found in the file system.

Details When smxFFS mount a disk, it will check the consistency of the disk to recover from any possible power fail issue. Normally you don't need to call this function in your application. It is mainly for the test purposes.

Pars	nID	The device ID that was specified in the call to sff_devreg().
Returns	SB_PASS	disk checked.
	SB_FAIL	disk IO error

See Also none

unsigned long **sff_clustersize** (uint nID)

Summary Returns the disk's data cluster size.

Details Call this function to get the disk's data cluster size, in bytes. Use the exact cluster size for sff_fwrite() and sff_fread() function for best performance.

Pars	nID	The device ID that was specified in the call to sff_devreg().

Returns Number of bytes per data cluster.

See Also none

int **sff_devreg** (const SBD_IF *dev_if, uint nID)

Summary Registers a device driver with smxFFS.

Details You must call this function to actually add a device driver to smxFFS. You can register as many drivers as specified by the macro MAX_DEV_NUM in ffcfg.h. You can call this function at any time after you call sff_init() and before you call sff_exit(). This function allocates some internal data structures from the heap.

Pars	dev_if	The device driver interface structure pointer.
	nID	The ID number to assign to the disk. You can specify any ID which is less than the macro MAX_DEV_NUM. The macro SFF_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 sff_init(), sff_devunreg(), Alternate Filesystem Access in Chapter 3.

Example

```
void appl_init()
{
    sff_init();
    sff_devreg(sfs_GetNANDInterface(), 0);
    fp = sff_fopen("d:\\test.bin", "wb");
    sff_fclose(fp);
}
```

int **sff_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 sff_devreg().

Returns	SFF_DEVICE_NOT_FOUND	Device ID invalid or not mounted.
	SFF_DEVICE_MOUNTED	Mounting is complete and the device can be used now.
	SFF_DEVICE_UNFORMATTED	The device is inserted but smxFFS could not find the correct format on it.

See Also sfs_devreg()

Example

```
If(SFF_DEVICE_NOT_FOUND  == sff_devstatus(0))
    printf("The disk 0 is not found.");
```

int **sff_devunreg** (uint nID)

Summary Unregisters a registered device driver from smxFFS.

Details Call this function to remove a device driver from smxFFS. When smxFFS is unmounted (by calling sff_exit()), this function will be called automatically so normally you do not need to call it explicitly.

Pars nID The device ID that was specified in the call to sff_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 sff_exit(), sff_devreg(), Alternate Filesystem Access in Chapter 3.

Example

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

int **sff_exit** (void)

Summary	Uninitializes the smxFFS file system.
Details	This is the last smxFFS API call that should be made at exit. This function un-registers all device drivers and stops the media status monitor task.
Pars	none

Returns

SB_PASS	Success.
SB_FAIL	Uninitialization failed.

See Also sff_init()

Example

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

int **sff_fclose** (SFF_FILEHANDLE filehandle)

Summary Closes an open file.

Details Closing a file causes all the data to be flushed to the media. All resources allocated by sff_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 sff_fopen().

Returns

SB_PASS	Success.
SB_FAIL	File cache flush failed or file was already closed.

See Also sff_fopen()

Example

```
SFF_FILEHANDLE fp;
fp = sff_fopen(“a:\\test.bin”, “wb”);
if(fp != NULL)
{
    sff_fwrite(…);
    sff_fclose(fp);
}
```

void **sff_fdelete** (const char * filename)

Summary Deletes a file.

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

Pars

filename	The name of the file to be deleted.

Returns

SB_PASS	Success.
SB_FAIL	File not found, file is open, or device has been removed.

See Also sff_findfile()

Example

```
SFF_FILEHANDLE fp;
sff_fdelete(“a:\\test.bin”);
sff_fdelete(“a:\\test.bin”); // attempting to delete a file that does not exist will not cause any damage.
```

int **sff_feof** (SFF_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 the 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 sff_fopen().

Returns

SB_PASS	EOF
SB_FAIL	not EOF

See Also sff_fopen(), sff_fseek(), sff_fwrite(), sff_fread()

Example

```
SFF_FILEHANDLE fp;
char buf[20]=”Test data”;
fp = sff_fopen(“a:\\data.dat”, “r+b”);
while(!sff_feof(fp))
    sff_fread(buf, 1, 20, fp);
sff_fclose(fp);
```

int **sff_fflush** (SFF_FILEHANDLE filehandle)

Summary Flushes 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 sff_fopen().

Returns

SB_PASS	Success.
SB_FAIL	Device has been removed or there is some other error.

See Also sff_fopen(), sff_fwrite()

Example

```
SFF_FILEHANDLE fp;
char buf[20]="Test data";
fp = sff_fopen("a:\\data.dat", "r+b");
sff_fwrite(buf, 1, 20, fp);
sff_fflush(fp);
sff_fclose(fp);
```

unsigned long **sff_filelength** (const char *filename)

Summary Returns 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 sff_findfirst(), sff_findnext()

Example

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

int **sff_findclose** (SFF_FINDHANDLE * handle)

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 sff_findfirst(), which makes this clear.

Pars	handle	The handle for the sff_findfirst/sff_findnext session.
Returns	0	The internal buffer has been freed.
	-1	Session handle is invalid.

See Also sff_findfirst(), sff_findnext()

Example See sff_findfirst().

int **sff_findfile** (const char *filename)

Summary Tests 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.

Pars	filename	The name of the file or directory to find. Wildcards are not supported.
Returns	SB_PASS	File found.
	SB_FAIL	File not found.

See Also sff_findfirst(), sff_findnext()

Example

```
if(sff_findfile("a:\\test.dat") > 0)
    printf("Found test.dat");
```

SFF_FINDHANDLE **sff_findfirst** (const char * filespec, SFF_FILEINFO * fileinfo)

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 find handle for the session, which can be used in a subsequent call to sff_findnext(). Otherwise, it returns NULL. 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*:
"a:\*.*"
"a:\\path\\*.dat"
"a:\\path\\test?.*"
"a:\\path\\test?2.dat"

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

Returns !NULL File found matching *filespec.*

NULL	No file found or out of memory.

See Also sff_findclose(), sff_findfile(), sff_findnext()

Example

```
SFF_FILEINFO fileinfo;
SFF_FINDHANDLE handle;
int id;
handle = sff_findfirst("a:\\*.*", &fileinfo);
if(handle)
{
    do
    {
        printf("File Name: %s, File Size: %d\n", fileinfo.name, fileinfo.st_size);
        id = sff_findnext(handle, &fileinfo);
    }while(id != -1);
    sff_findclose(handle);
}
```

int **sff_findnext** (SFF_FINDHANDLE handle, SFF_FILEINFO * fileinfo)

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

Details If successful, this function returns 0. Otherwise, returns –1. Check (fileinfo.st_mode & S_IFDIR) to see if it is a directory rather than a file.

Pars

handle	The find handle from the sff_findfirst() call.
fileinfo	The returned file info which includes the file's name and size.

Returns

0	File found matching *filespec.*
-1	No file found.

See Also sff_findclose(), sff_findfile(), sff_findfirst()

Example See sff_findfirst().

SFF_FILEHANDLE **sff_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 sff_fmkdir() 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"**.

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

See Also sff_fclose(), sff_fmkdir()

Example

```
/* single open request */
SFF_FILEHANDLE fp;
fp = sff_fopen("a:\\test.bin", "r");
if(fp != NULL)
{
    sff_fread(…..);
    sff_fclose(fp);
}
```

int **sff_format** (uint nID)

Summary Formats a disk.

Details Formats a disk.

Note that smxFFS will autoformat an unformatted disk during the mount process. You may not need to call this function in your application for the normal use case.

Pars	nID	The device ID that was specified in the call to sff_devreg().
	formatinfo	Pointer to a structure with additional format parameters. If NULL, default values are used.

Returns	SB_PASS	Success.
	SB_FAIL	Some error occurred.

See Also sff_init(), sff_devreg()

Examples

```
sff_format(0);
```

size_t **sff_fread** (void *buf, size_t size, size_t items, SFF_FILEHANDLE filehandle)

Summary Reads some data from an open file.

Details This function reads up to (*items* * *size*) bytes from the current file pointer in the file and stores them in *buf*. The file pointer is increased by the number of bytes actually read. The file pointer position is indeterminate if an error occurs. The value of a partially read item cannot be determined.

Pars

buf	Pointer to the buffer to store the returned data.
size	Item size in bytes.
items	Maximum number of items to be read.
filehandle	File handle returned by sff_fopen().

Returns

value	Number of items read.
0	Error or reached the end of file.

See Also sff_fopen(), sff_fwrite()

Example

```
SFF_FILEHANDLE fp;
char buf[20];
fp = sff_fopen("a:\\test.bin", "rb");
if(fp != NULL)
{
    sff_fread(buf, 1, 20, fp); // if "test.bin" file size is 0, this call will return 0.
    sff_fclose(fp);
}
```

long **sff_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.

Pars

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

Returns

>= 0	Free size (kilobytes) of the disk.
-1	The deviceID is not valid or the device is not inserted.

See Also sff_devreg(), sff_totalkb()

Example

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

int **sff_fseek** (SFF_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 sff_fopen().
offset	Number of bytes from *whence.*
whence	Initial position; three predefined constants are:

SFF_SEEK_CUR	Current position of file pointer
SFF_SEEK_END	End of file
SFF_SEEK_SET	Beginning of file

Returns

0	Success.
!0	Fail.

See Also sff_fopen(), sff_fread(), sff_fwrite()

Example

```
/* normal seek operation */
SFF_FILEHANDLE fp;
char buf[20];
fp = sff_fopen("d:\\test.bin", "rb");
if(fp != NULL)
{
    sff_fseek(fp, 10, SFF_SEEK_SET);
    sff_fread(buf, 1, 20, fp);
    sff_fclose(fp);
}

/* seeking beyond the file area will cause error  if it is Read-Only */
SFF_FILEHANDLE fp;
char buf[20]="This is a test.";
fp = sff_fopen("d:\\test.bin", "rb");
if(fp != NULL)
{
    sff_fseek(fp, 10, SFF_SEEK_END);    // this will move the pointer to the end of file.
    sff_fclose(fp);
}

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

long **sff_ftell** (SFF_FILEHANDLE filehandle)

Summary Returns the current file pointer.

Details This function returns the current file pointer.

Pars

filehandle	File handle returned by sff_fopen().

Returns

value	File pointer position.

See Also sff_fopen(), sff_fseek()

Example

```
SFF_FILEHANDLE fp;
char buf[20]=”Test data”;
fp = sff_fopen(“a:\\data.dat”, “r+b”);
sff_fwrite(buf, 1, 20, fp);
sff_fseek(fp, sff_ftell(fp) -1, SFF_SEEK_SET );
sff_fclose(fp);
```

int **sff_ftruncate** (SFF_FILEHANDLE filehandle)

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 sff_fopen().

Returns

SB_PASS	The file has been truncated successfully.
SB_FAIL	The file was not truncated due to an error.

See Also sff_fopen(), sff_fseek(), sff_fwrite()

Example

```
SFF_FILEHANDLE fp;
char buf[20]=”Test data”;
fp = sff_fopen(“a:\\data.dat”, “r+b”);
sff_fwrite(buf, 1, 20, fp);
sff_fseek(fp, sff_ftell(fp) -10 , SFF_SEEK_SET );
sff_ftruncate(fp); //discard 10 bytes
sff_fclose(fp);
```

size_t **sff_fwrite** (void *buf, size_t size, size_t items, SFF_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", sff_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 sff_fopen().

Returns

value	Number of items written.
0	Error.

See Also sff_fopen(), sff_fread()

Example

```
/* normal write operation */
SFF_FILEHANDLE fp;
char buf[20]="This is a test.";
fp = sff_fopen("a:\\test.bin", "wb");
if(fp != NULL)
{
    sff_fwrite(buf, 1, 20, fp);
    sff_fclose(fp);
}

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

char * **sff_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 sff_setcwd()

Example

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

int **sff_getlasterror** (uint nID)

Summary Gets 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 sff_devreg().

Returns The error code of the last failed file operation. Error codes are:

SFF_ERR_NO_ERROR	No error.
SFF_ERR_DISK_REMOVED	Disk is removed.
SFF_ERR_DISK_IO	Disk driver returned I/O error.
SFF_ERR_INVALID_DIR	Diretory entry contains invalid field.
SFF_ERR_INVALID_MCB	MCB settings are not the same as current configuration.
SFF_ERR_INVALID_PAR	Function got invalid parameter or settings
SFF_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.
SFF_ERR_DISK_FULL	File system cannot find a free data cluster.
SFF_ERR_DISK_WP	Disk is write protected.
SFF_ERR_FILE_EXIST	File already exists. For example, you want to rename a file, but a file with the new name already exists.
SFF_ERR_FILE_NOT_EXIST	File does not exist. For example, you want to rename a file but the file does not exist.
SFF_ERR_OUT_OF_MEM	File system could not allocate required memory.

See Also sff_fopen(), sff_fread(), sff_fwrite()

Example

```
void main()
{
    SFF_FILEHANDLE fp;
    sff_devreg(0, pDevInterface);
    sff_fdelete(“A:\test.bin”);
    fp = sff_fopen(“A:\test.bin”, “rb”);
    if(fp == NULL)
    {
        printf(“Last Error Code is %d\r\n”, sff_getlasterror(0));
    }
}
```

int **sff_gettimestamp** (const char * filename, DATETIME* datetime)

Summary Gets the modification time for a file or directory.

Details Gets the modification time for a file or directory.

Pars

file	The full name of the file or directory whose time you want to check.
datetime	The structure to hold the returned time.

Returns

0	Got timestamp successfully.
!=0	File or directory not found.

See Also None

Example

```
void appl_init()
{
    DATETIME datetime;
    sff_init();
    sff_devreg(sfs_GetNANDInterface(), 0);
    sff_gettimestamp(“A:\\test.bin”, &datetime);
    /* use the time returned */
}
```

int **sff_init** (void)

Summary Initializes the smxFFS internal data structures.

Details This function must be called before calling any other smxFFS API functions. Then sff_devreg() must be called to register each device driver.

Pars none

Returns

SB_PASS	Success.
SB_FAIL	Initialization failed. smxFFS could not start the media status monitor task.

See Also sff_exit()

Example

```
void appl_init()
{
    if(sff_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 **sff_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. smxFFS 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 sff_devreg().
command	Driver-specific command. User commands must be >= SBD_IOCTL_CUSTOM. Values less than this are used internally by smxFFS 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 sff_devreg()

Example

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

int **sff_mkdir** (const char *path)

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, "a:\\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 sff_rmdir()

Example

```
/* create one directory on the root */
sff_mkdir("a:\\path");

/* create one parent directory and two subdirectory */
if(sff_mkdir("a:\\parent"))
{
    sff_mkdir("a:\\parent\\sub1");
    sff_mkdir("a:\\parent\\sub2");
}
```

int **sff_rename** (const char * oldname, const char * newname)

Summary Renames a file or directory.

Details This function renames the file or directory specified by *oldname* to the name given by *newname*. 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). The path must be the same in the two names. It cannot move files between two directories or disks.

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 sff_findfile()

Example

```
SFF_FILEHANDLE fp;
char buf[20]="Test data";
fp = sff_fopen("d:\\data.dat", "w+b");
sff_fwrite(buf, 1, 20, fp);
sff_fclose(fp);
sff_rename("d:\\data.dat", "d:\\newdata.dat");
```

void **sff_rewind** (FILEHANDLE filehandle)

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

Details This is equivalent to sff_fseek(filehandle, 0, SFF_SEEK_SET).

Pars

filehandle	File handle returned by sff_fopen().

Returns none

See Also sff_fopen(), sff_fseek()

Example

```
SFF_FILEHANDLE fp;
char buf[20];
fp = sff_fopen("a:\\data.dat", "rb");
sff_fread(buf, 1, 20, fp);
sff_rewind(fp);
sff_fclose(fp);
```

int **sff_rmdir** (const char *path)

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 sff_fdelete().

Pars	path	The full path name. For example, "a:\\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 sff_mkdir(), sff_fdelete()

Example

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

int **sff_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 call this function from a particular task.

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 sff_getcwd(), sff_mkdir()

Example

```
void appl_init()
{
    sff_init();
    sff_devreg(sfs_ GetNANDInterface (), 0);
    sff_mkdir(“a:\\test”);
    sff_setcwd(“a:\\test”);
}
```

int **sff_timestamp** (const char * filename, DATETIME* datetime)

Summary Sets 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 name of the file or directory whose time you want to modify.
datetime	The structure containing the new modification time.

Returns

SB_PASS	The timestamp has been changed successfully.
SB_FAIL	File or directory not found.

See Also None

Example

```
void appl_init()
{
    DATETIME datetime;
    sff_init();
    sff_devreg(sfs_GetNANDInterface(), 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;
    sff_timestamp(“A:\\test.bin”, &datetime);
}
```

long **sff_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 sff_devreg().
Returns	>= 0	Total size (kilobytes) of the disk.
	-1	The Device ID is not valid or the device is not inserted.
See Also	sff_devreg(), sff_freekb()	

Example

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

A. File Summary

FILE	DESCRIPTION
smxffs.h	Main header file. Include in your application code. Includes all needed smxFFS header files in the proper order.
ffcfg.h	Configuration file for smxFFS.
ffintern.h	Internal main header file. Used only by smxFFS files. It includes other header files in the proper order.
ffconst.h	Internal constant value definitions.
ffstruc.h	Internal data structure definitions.
ffapi.c,h	File I/O API functions such as sfs_fopen(), sfs_fclose().
ffcache.c,h	Data and free cluster cache related functions.
ffind.c,h	Functions used by sff_findfirst() and sff_findnext().
ffmount.c,h	File system mount related functions.
ffpath.c,h	Directory related functions.
ffport.c,h	Porting functions for hardware.
fdnand.c,h ..\xfd\nand*.*	NAND flash driver.
fdnor.c,h ..\xfd\nor*.*	NOR flash driver.
mak.bat, ffs.mak	Makefile for building the smxFFS library for SMX.

B. Porting Notes

B.1 ffcfg.h

ffcfg.h contains file system configuration constants.

B.2 fport.h and fport.c

ffport.* contains porting functions that are specific to smxFFS, such as the interface to get local date/time, debug information output functions, and byte order swapping macros. smxFFS's porting layer maps onto smxBase services, so for general purpose OS, hardware, and compiler porting information, please see the smxBase User's Guide.

void **SFF_API_ENTER**(SFF_MUTEX_HANDLE *handle)
Tests the API mutex or semaphore. Waits if another API function has claimed it.

void **SFF_API_EXIT**(SFF_MUTEX_HANDLE *handle)
Signals the API mutex or semaphore so other API functions can run.

B.3 C Library Function Requirements

This is a list of C library functions that smxFFS calls. If your compiler does not provide some of these, you should implement them in brtl.c in smxBase. Some are already implemented there, so it is just a matter of changing the conditionals to enable them for your compiler.

- memcpy()
- memcmp()
- memset()
- strcpy()
- strlen()
- strstr()
- strcmp()
- strchr()

C. Size and Performance

C.1 Code Size

Code size varies depending upon CPU, compiler, and optimization level. Size does not include the flash driver. See the smxNAND and smxNOR User's Guides for their sizes.

	ARM7/9	**ColdFire**
	IAR	CodeWarrior
smxFFS	20 KB	

C.2 Data Size

smxFFS was designed to minimize RAM use. Size does not include the flash driver. See the smxNAND and smxNOR User's Guides for their sizes.

512 byte sector size, one open file	4KB
2048 byte sector size, one open file	10KB
4096 byte sector size, one open file	18KB

C.3 Performance

The following are performance results for smxFFS on platforms we tested.

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

Platform	**Reading**	**Writing**
AT91SAM9M10G45-EK 256MB NAND	5600 KB/s	1900 KB/s

D. Tested Hardware

D.1 NAND

- K9F1G08U on NXP LPC2468 board.
- MT29F2G08ABD on Atmel AT91SAM9M10G45EVB board.
- K9F2808U on our Avnet Coldfire 5282 add-on board.

D.2 NOR

- 39VF320 on NXP LPC2468 board.