

smxFFS™

Flash File System

smxFFS is a flash file system for use with board-resident arrays of NAND flash memory. It has the standard C library file API and it is power fail-safe.

General

smxFFS is a simple, power fail-safe file system for use with the smxNAND flash driver. It provides the standard C library API: `fopen()`, `fread()`, `fwrite()`, `fseek()`, `fclose()`, etc. for the application. Unlike the smxFS FAT file system, smxFFS is not intended to work with media other than board-resident NAND flash. It does not support media such as SD or Compact Flash cards. Also, its file structure is not DOS- or Windows-compatible and it does not support subdirectories. smxFFS takes advantage of knowing that the media is NAND flash. To support other media, smxFS should be used. For simple logging to NAND or NOR flash, smxFLog is recommended.

Operation

smxFFS assumes a linear address space for the flash storage media. It also assumes that the media can only be accessed by sector (e.g. 512 bytes at a time). The structure of the flash address space is as follows:

- **Reserved Area** is divided into 2 parts: an area used by smxNAND and a user area that can store a boot sector or other data.
- **File System Signature Area** stores a string to identify the flash file system. It is located at the beginning of the next flash block following the reserved area.

Features

- Supports moderate to large flash memories.
 - Standard C library file API without subdirectory support.
 - Power fail-safe.
 - Integrated with smxNAND driver.
 - Multitasking support.
 - Optimized for SMX® RTOS.
 - Easily portable or runs standalone.
 - Can coexist with smxFS and smxFLog.
 - Source code included.
 - PC emulator included.
-
- **File Control Block (FCB) Area** acts like a root directory. Each FCB represents one file and contains the following information: status, file name, file extension, first cluster index, and file length. FCB area size depends upon how many files may be stored on the flash. Each FCB requires 20 bytes and the FCB area must be a multiple of the flash block size. For a 16 KB block size, up to 400 files can be handled.
 - **File Allocation Table (FAT) Area** contains the FAT nodes. Each node represents one data cluster. 16-bit and 32-bit FAT nodes are supported. FAT size can be minimized by increasing the cluster size. The FAT must also be a multiple of the flash block size
 - **File Data Area** stores the file data. It is aligned on a flash block boundary.

The first cluster index in a file's FCB is the index of the file's first FAT node. If it contains END_CLUSTER, then the file has no data. Each FAT node, by its location in the FAT, corresponds to a data cluster in the linear address space of the flash data area. Each FAT node contains a pointer to the next FAT node of the file. The END_CLUSTER flag marks the last node and the last cluster of the file.

The FAT node size is 16 bits, by default, so smxFFS can support up to $(65534 * \text{SectorsPerCluster} * \text{BytesPerSector})$ bytes of data. (Note that two values are reserved for END_CLUSTER and BAD_CLUSTER). For example: 16-bit FAT nodes, 8 sectors per cluster, and 512 bytes per sector can support media up to 256 MB, in size. Media size includes the total size of all NAND chips in the array. In this case, the FAT table requires 32 KB of flash and RAM. Doubling the cluster size will halve the FAT size, assuming that the media size stays the same.

Larger media can be supported by increasing the number of sectors per cluster or by increasing the FAT node size to 32-bits. This is controlled by the configuration file. For FAT32, the FAT size can become quite large. For example, for a 1 GB media size with 16 sectors per cluster, the FAT requires 64 KB of flash and RAM. Because the FAT is written to flash every time a file is closed (see the power fail-safe discussion that follows), this not only increases the RAM requirement, but it also may hurt performance

When the file system is *mounted*, the FCB and FAT areas are cached in RAM to improve performance. For each open file, the data cluster that is currently being read or written is also cached in RAM. When another cluster needs to be accessed, all altered sectors of the current cluster are first written to flash, then the new cluster is loaded into the cache. Altered sectors are also written to flash whenever `fclose()` is called.

Power-Fail-Safe Operation

As noted above, the FCB and FAT are copied to RAM where they are accessed and may be modified while a file is open. When a file is closed, all data sectors and the updated FCB and FAT are written to flash. When this process is complete, smxFFS calls the whole cache writeback function of the flash driver, which updates the Block Table in flash and discards the old Block Table. (See discussion in the smxNAND data sheet.) At this point the new file data, the new FCB, and the new FAT are all saved in flash. No data will be lost if power is then lost. However, if power is lost before the full operation is completed, then the file system will revert back to the original file, FCB, and FAT. The new data will have been lost, but file system integrity will have been preserved. See the smxNAND data sheet for more information.

In environments where power outages are frequent, files should be closed when not needed in order to protect as much data from loss as possible. However, over-use of this feature may cause excessive NAND flash wear.

Porting Layer

Two files, `ffsport.h` and `ffsport.c`, contain definitions, macros, and functions to port smxFFS to any OS, compiler, or processor. Currently, they support SMX[®] and Windows. (A NAND flash emulator is provided under Windows, that uses the hard disk for testing FFS.) These files implement semaphore protection of the smxFFS API from reentrancy problems in multitasking environments.

smxFFS API

Basic Calls

<code>v_ffs_fmout ()</code>	Mount the NAND Flash File System.
<code>v_ffs_funmount ()</code>	Unmount the NAND Flash File System.
<code>v_ffs_fopen (filename, mode)</code>	Open one file for read/write access.
<code>v_ffs_fclose (filehandle)</code>	Close an open file.
<code>v_ffs_fread (buf, size, items, filehandle)</code>	Read data from an open file.
<code>v_ffs_fwrite (buf, size, items, filehandle)</code>	Write data to an open file.
<code>v_ffs_fseek (filehandle, offset, whence)</code>	Move the file pointer to the specified location.
<code>v_ffs_fdelete (filename)</code>	Delete a file.
<code>v_ffs_findfile (filename)</code>	Test if a file exists.
<code>v_ffs_filelength (filename)</code>	Return the length of a file, in bytes.
<code>v_ffs_fgetversion ()</code>	Return the file system version number.

Extended Calls

<code>v_ffs_rewind (filehandle)</code>	Move the file pointer to the beginning of the file.
<code>v_ffs_fflush (filehandle)</code>	Flush all data associated with the file handle to the storage media.
<code>v_ffs_ftell (filehandle)</code>	Determine the current file pointer position.
<code>v_ffs_fsetend (filehandle)</code>	Truncate a file at the current file pointer.
<code>v_ffs_feof (filehandle)</code>	Test for end-of-file on a file.
<code>v_ffs_rename (oldname, newname)</code>	Rename a file.
<code>v_ffs_findfirst (filespec, fileinfo)</code>	Provide information about the first instance of a file whose name matches the name specified by the <i>filespec</i> argument.
<code>v_ffs_findnext (id, fileinfo)</code>	Find the next file, if any, whose name matches the <i>filespec</i> argument in a previous call to <code>v_ffs_findfirst()</code> , and return information about it in the <i>fileinfo</i> structure.
<code>v_ffs_fgetfilenum (filespec)</code>	Determine the total number of files, if any, whose names match the <i>filespec</i> argument.
<code>v_ffs_flushall ()</code>	Flush all data to the storage media.
<code>v_ffs_freespace ()</code>	Determine the percentage of free space on the storage media.
<code>v_ffs_freesize ()</code>	Determine the number of free bytes on the storage media.
<code>v_ffs_devicemounted ()</code>	Determine the percentage of free space on the storage media.
<code>v_ffs_formaterror ()</code>	Determine if the file system structure has an error.

Size and Performance

Code Size (excluding smxNAND)

Code size varies depending upon the CPU, compiler, and optimization level. Below are some examples. The Mini Library includes only the basic calls.

CPU and Compiler	Full Library (KB)	Mini Library (KB)
Coldfire & CodeWarrior 6.3	12	7
ARM & IAR 5.11	12.5	7.5
x86 & Borland C++ 32-bit	11	7

Data Size (excluding smxNAND)

Data size depends upon the flash size, maximum number of files, and the cluster size. For large flash, the maximum number of files is not very important; cluster size is the most important factor.

For a 16 MB flash chip, with a 16 KB block size:

Files	Cluster Size (Bytes)	RAM (Bytes)
100	4096	20398
100	1024	51886
1024	512	116971

For a 256 MB flash chip, with 128 KB block size:

Files	Cluster Size (Bytes)	RAM (Bytes)
32	16384	66208
32	8192	131744

For a 512 MB flash chip, with 128 KB block size:

Files	Cluster Size (Bytes)	RAM (Bytes)
32	16384	155176
32	8192	204320

For a 1 GB flash chip, with 128 KB block size:

Files	Cluster Size (Bytes)	RAM (Bytes)
32	16384	311312

For a 2 GB flash chip, with 256 KB block size:

Files	Cluster Size (Bytes)	RAM (Bytes)
32	32768	327696

Performance

See the smxNAND data sheet for performance.