SMARTFS

This page contains information about the implementation of the NuttX Sector Mapped Allocation for Really Tiny (SMART) FLASH file system, SMARTFS.

Features

This implementation is a full-feature file system from the perspective of file and directory access (i.e. not considering low-level details like the lack of bad block management). The SMART File System was designed specifically for small SPI based FLASH parts (1-8 Mbyte for example), though this is not a limitation. It can certainly be used for any size FLASH and can work with any MTD device by binding it with the SMART MTD layer and has been tested with devices as large as 128MByte (using a 2048 byte sector size with 65534 sectors).

The FS includes support for:

- Multiple open files from different threads.
- Open for read/write access with seek capability.
- Appending to end of files in either write, append or read/write open modes.
- Directory support.
- Support for multiple mount points on a single volume / partition (see details below).
- Selectable FLASH Wear leveling algorithym
- Selectable CRC-8 or CRC-16 error detection for sector data
- Reduced RAM model for FLASH geometries with large number of sectors (16K-64K)

General operation

The SMART File System divides the FLASH device or partition into equal sized sectors which are allocated and "released" as needed to perform file read/write and directory management operations. Sectors are then "chained" together to build files and directories. The operations are split into two layers:

1. The MTD block layer (nuttx/drivers/mtd/smart.c). This layer manages all low-level FLASH access operations including sector allocations, logical to physical sector mapping, erase operations, etc.
2. The FS layer (nuttx/fs/smart/smartfs_smart.c). This layer manages high-level file and directory creation, read/write, deletion, sector chaining, etc.

SMART MTD Block layer

The SMART MTD block layer divides the erase blocks of the FLASH device into "sectors". Sectors have both physical and logical number assignments. The physicl sector number represents the actual offset from the beginning of the device, while the logical sector number is assigned as needed. A physical sector can have any logical sector assignment, and as files are created, modified and destroyed, the logical sector number assignment for a given physical sector will change over time. The logical sector number is saved in the physical sector header as the first 2 bytes, and the MTD layer maintains an in-memory map of the logical to physical mapping. Only physical sectors that are in use will have a logical assignment.

Also contained in the sector header is a flags byte and a sequence number. When a sector is allocated, the COMMITTED flag will be "set" (changed from erase state to non-erase state) to indicate the sector data is valid. When a sector's data needs to be deleted, the RELEASED flag will be "set" to indicate the sector is no longer in use. This is done because the erase block containing the sector cannot necessarily be erased until all sectors in that block have been "released". This allows sectors in the erase block to remain active while others are inactive until a "garbage collection" operation is needed on the volume to reclaim released sectors.

The sequence number is used when a logical sector's data needs to be updated with new information. When this happens, a new physical sector will be allocated which has a duplicate logical sector number but a higher sequence number. This allows maintaining flash consistency in the event of a power failure by writing new data prior to releasing the old. In the event of a power failure causing duplicate logical sector numbers, the sector with the higher sequence number will win, and the older logical sector will be released.

The SMART MTD block layer reserves some logical sector numbers for internal use, including:

```
Sector 0:     The Format Sector.  Has a format signature, format version, etc.
              Also contains wear leveling information if enabled.
Sector 1-2:   Additional wear-leveling info storage if needed.
Sector 3:     The 1st (or only) Root Directory entry
Sector 4-10:  Additional root directories when Multi-Mount points are supported.
Sector 11-12: Reserved
```

To perform allocations, the SMART MTD block layer searches each erase block on the device to identify the one with the most free sectors. Free sectors are those that have all bytes in the "erased state", meaning they have not been previously allocated/released since the last block erase. Not all sectors on the device can be allocated the SMART MTD block driver must reserve at least one erase-block worth of unused sectors to perform garbage collection, which will be performed automatically when no free sectors are available. When wear leveling is enabled, the allocator also takes into account the erase block erasure status to maintain level wearing.

Garbage collection is performed by identifying the erase block with the most "released" sectors (those that were previously allocated but no longer being used) and moving all still-active sectors to a different erase block. Then the now "vacant" erase block is erased, thus changing a group of released sectors into free sectors. This may occur several times depending on the number of released sectors on the volume such that better "wear leveling" is achieved.

Standard MTD block layer functions are provided for block read, block write, etc. so that system utilities such as the "dd" command can be used, however, all SMART operations are performed using SMART specific ioctl codes to perform sector allocate, sector release, sector write, etc.

A couple of config items that the SMART MTD layer can take advantage of in the underlying MTD drivers is SUBSECTOR_ERASE and BYTE_WRITE. Most flash devices have a 32K to 128K Erase block size, but some of them have a smaller erase size available also. Vendors have different names for the smaller erase size; In the NuttX MTD layer it is called SUBSECTOR_ERASE. For FLASH devices that support the smaller erase size, this configuration item can be added to the underlying MTD driver, and SMART will use it. As of the writing of this page, only the drivers/mtd/m25px.c driver had support for SUBSECTOR_ERASE.

The BYTE_WRITE config option enables use of the underlying MTD driver's ability to write data a byte or a few bytes at a time vs. a full page at at time (which is typically 256 bytes). For FLASH devices that support byte write mode, support for this config item can be added to the MTD driver. Enabling and supporting this feature reduces the traffic on the SPI bus considerably because SMARTFS performs many operations that affect only a few bytes on the device. Without BYTE_WRITE, the code must perform a full page read-modify-write operation on a 256 or even 512 byte page.

Wear Leveling

When wear leveling is enabled, the code automatically writes data across the entire FLASH device in a manner that causes each erase block to be worn (i.e. erased) evenly. This is accomplished by maintaining a 4-bit wear level count for each erase block and forcing less worn blocks to be used for writing new data. The code maintains each block's erase count to be within 16 erases of each other, though through testing, the span so far was never greater than 10 erases of each other.

As the data in a block is modified repeatedly, the erase count will increase. When the wear level reaches a value of 8 or higher, and the block needs to be erased (because the data in it has been modified, etc.) the code will select an erase block with the lowest wear count and relocate it to this block (with the higher wear count). The idea being that a block with the lowest wear count contains more "static" data and should require fewer additional erase operations. This relocation process will continue on the block (only when it needs to be erased again).

When the wear level of all erase blocks has increased to a level of SMART_WEAR_MIN_LEVEL (currently set to 5), then the wear level counts will all be reduced by this value. This keeps the wear counts normalized so they fit in a 4-bit value. Note that theoretically, it IS possible to write data to the flash in a manner that causes the wear count of a single erase block to increment beyond it's maximum value of 15. This would have to be a very, very, very specific and un-predictable write sequence though as data is always spread out across the sectors and relocated dynamically. In the extremely rare event this does occur, the code will automatically cap the maximum wear level at 15 an increment an "uneven wear count" variable to indicate the number times this event has occurred. So far, I have not been able to get the wear count above 10 though my testing.

The wear level status bits are saved in the format sector (logical sector number zero) with overflow saved in the reserved logical sectors one and two. Additionally, the uneven wear count (and total block erases if PROCFS is enabled) are stored in the format sector. When the PROCFS file system is enabled and a SMARTFS volume is mounted, the SMART block driver details and / or wear level details can be viewed with a command such as:

```
cat /proc/fs/smartfs/smart0/status
   Format version:    1
   Name Len:          16
   Total Sectors:     2048
   Sector Size:       512
   Format Sector:     1487
   Dir Sector:        8
   Free Sectors:      67
   Released Sectors:  572
   Unused Sectors:    817
   Block Erases:      5680
   Sectors Per Block: 8
   Sector Utilization:98%
   Uneven Wear Count: 0

cat /proc/fs/smartfs/smart0/erasemap
   DDDCGCCDDCDCCDCBDCCDDGBBDBCDCCDDDCDDDDCCDDCCCGCGDCCDBCDDGBDBDCDD
   BCCCDDCCDDDCBCCDGCCCBDDCCGBBCBCCGDCCDCBDBCCCDCDDCDDGCDCGDCBCDBDG
   BCDDCDCBGCCCDDCGBCCGBCCBDDBDDCGDCDDDCGCDDBCDCBDDBCDCGDDCCBCGBCCC
   GCBCCGCCCDDDBGCCCCGDCCCCCDCDDGBBDACABDBBABCAABCCCDAACBADADDDAECB
```

Enabling wear leveling can increase the total number of block erases on the device in favor of even wearing (erasing). This is caused by writing / moving sectors that otherwise don't need to be written to move static data to the more highly worn blocks. This additional write requirement is known as write amplification. To get an idea of the amount of write amplification incurred by enabling wear leveling, I conducted the smart_test example using four different configurations (wear, no wear, CRC-8, no CRC) and the results are shown below. This was done on a 1M Byte simulated FLASH with 4K erase block size, 512 sectors per byte. The smart_test creates a 700K file and then performs 20,000 random seek, write, verify tests. The seek write forces a multitude of sector relocation operations (with or without CRC enabled), causing a boatload of block erases.

Enabling wear leveling actually decreased the number of erase operations with CRC enabled or disabled. This is only a single test point based one testing method ... results will likely vary based on the method the data is written, the amount of static vs. dynamic data, the amount of free space on the volume, and the volume geometry (erase block size, sector size, etc.).

The results of the tests are:

Case		Total Block erases
No wear leveling	CRC-8	6632
Wear leveling	CRC-8	5585
No wear leveling	no CRC	6658
Wear leveling	no CRC	5398

Reduced RAM model

On devices with a larger number of logical sectors (i.e. a lot of erase blocks with a small selected sector size), the RAM requirement can become fairly significant. This is caused by the in-memory sector map which keeps track of the logical to physical mapping of all sectors. This is a RAM array which is 2 * totalsectors in size. For a device with 64K sectors, this means 128K of RAM is required just for the sector map, not counting RAM for read/write buffers, erase block management, etc.

So a reduced RAM model has been added which only keeps track of which logical sectors have been used (a table which is totalsectors / 8 in size) and a configurable sized sector map cache. Each entry in the sector map cache is 6 bytes (logical sector, physical sector and cache entry age). ON DEVICES WITH SMALLER TOTAL SECTOR COUNT, ENABLING THIS OPTION COULD ACTUALLY INCREASE THE RAM FOOTPRINT INSTEAD OF REDUCE IT.

The sector map cache size should be selected to balance the desired RAM usage and the file system performance. When a logical to physical sector mapping is not found in the cache, the code must perform a physical search of the FLASH to find the requested logical sector. This involves reading the 5-byte header from each sector on the device until the sector is found. Performing a full read, seek or open for append on a large file can cause the sector map cache to flush completely if the file is larger than (cache entries * sector size). For example, in a configuration with 256 cache entries and a 512 byte sector size, a full read, seek or open for append on a 128K file will flush the cache.

An additional RAM savings is realized on FLASH parts that contain 16 or fewer logical sectors per erase block by packing the free and released sector counts into a single byte (plus a little extra for 16 sectors per erase block). A device with a 64K erase block size can benefit from this savings by selecting a 4096 or 8192 byte logical sector size, for example.

SMART FS Layer

This layer interfaces with the SMART MTD block layer to allocate / release logical sectors, create and destroy sector chains, and perform directory and file I/O operations. Each directory and file on the volume is represented as a chain or "linked list" of logical sectors. Thus the actual physical sectors that a give file or directory uses does not need to be contiguous and in fact can (and will) move around over time. To manage the sector chains, the SMARTFS layer adds a "chain header" after the sector's "sector header". This is a 5-byte header which contains the chain type (file or directory), a "next logical sector" entry and the count of bytes actually used within the sector.

Files are stored in directories, which are sector chains that have a specific data format to track file names and "first" logical sector numbers. Each file in the directory has a fixed-size "directory entry" that has bits to indicate if it is still active or has been deleted, file permission bits, first sector number, date (utc stamp), and filename. The filename length is set from the CONFIG_SMARTFS_NAMLEN config value at the time the mksmartfs command is executed. Changes to the CONFIG_SMARTFS_NAMLEN parameter will not be reflected on the volume unless it is reformatted. The same is true of the sector size parameter.

Subdirectories are supported by creating a new sector chain (of type directory) and creating a standard directory entry for it in it's parent directory. Then files and additional sub-directories can be added to that directory chain. As such, each directory on the volume will occupy a minimum of one sector on the device. Subdirectories can be deleted only if they are "empty" (i.e they reference no active entries). There are no provision made for performing a recursive directory delete.

New files and subdirectories can be added to a directory without needing to copy and release the original directory sector. This is done by writing only the new entry data to the sector and ignoring the "bytes used" field of the chain header for directories. Updates (modifying existing data) or appending to a sector for regular files requires copying the file data to a new sector and releasing the old one.

SMARTFS organization

The following example assumes 2 logical blocks per FLASH erase block. The actual relationship is determined by the FLASH geometry reported by the MTD driver:

```
ERASE LOGICAL                   Sectors begin with a sector header.  Sectors may
BLOCK SECTOR      CONTENTS      be marked as "released," pending garbage collection
  n   2*n     --+---------------+
     Sector Hdr |LLLLLLLLLLLLLLL| Logical sector number (2 bytes)
                |QQQQQQQQQQQQQQQ| Sequence number (2 bytes)
                |SSSSSSSSSSSSSSS| Status bits (1 byte)
                +---------------+
         FS Hdr |TTTTTTTTTTTTTTT| Sector Type (dir or file) (1 byte)
                |NNNNNNNNNNNNNNN| Number of next logical sector in chain
                |UUUUUUUUUUUUUUU| Number of bytes used in this sector
                |               |
                |               |
                | (Sector Data) |
                |               |
                |               |
      2*n+1   --+---------------+
     Sector Hdr |LLLLLLLLLLLLLLL| Logical sector number (2 bytes)
                |QQQQQQQQQQQQQQQ| Sequence number (2 bytes)
                |SSSSSSSSSSSSSSS| Status bits (1 byte)
                +---------------+
         FS Hdr |TTTTTTTTTTTTTTT| Sector Type (dir or file) (1 byte)
                |NNNNNNNNNNNNNNN| Number of next logical sector in chain
                |UUUUUUUUUUUUUUU| Number of bytes used in this sector
                |               |
                |               |
                | (Sector Data) |
                |               |
                |               |
 n+1  2*(n+1) --+---------------+
     Sector Hdr |LLLLLLLLLLLLLLL| Logical sector number (2 bytes)
                |QQQQQQQQQQQQQQQ| Sequence number (2 bytes)
                |SSSSSSSSSSSSSSS| Status bits (1 byte)
                +---------------+
         FS Hdr |TTTTTTTTTTTTTTT| Sector Type (dir or file) (1 byte)
                |NNNNNNNNNNNNNNN| Number of next logical sector in chain
                |UUUUUUUUUUUUUUU| Number of bytes used in this sector
                |               |
                |               |
                | (Sector Data) |
                |               |
                |               |
              --+---------------+
```

Headers

SECTOR HEADER

Each sector contains a header (currently 5 bytes) for identifying the status of the sector. The header contains the sector's logical sector number mapping, an incrementing sequence number to manage changes to logical sector data, and sector flags (committed, released, version, etc.). At the block level, there is no notion of sector chaining, only allocated sectors within erase blocks.

FORMAT HEADER

Contains information regarding the format on the volume, including a format signature, formatted block size, name length within the directory chains, etc.

CHAIN HEADER

The file system header (next 5 bytes) tracks file and directory sector chains and actual sector usage (number of bytes that are valid in the sector). Also indicates the type of chain (file or directory).

Multiple Mount Points

Typically, a volume contains a single root directory entry (logical sector number 1) and all files and subdirectories are "children" of that root directory. This is a traditional scheme and allows the volume to be mounted in a single location within the VFS. As a configuration option, when the volume is formatted via the mksmartfs command, multiple root directory entries can be created instead. The number of entries to be created is an added parameter to the mksmartfs command in this configuration.

When this option has been enabled in the configuration and specified during the format, then the volume will have multiple root directories and can support a mount point in the VFS for each. In this mode, the device entries reported in the /dev directory will have a directory number postfixed to the name, such as:

```
/dev/smart0d1
/dev/smart0d2
/dev/smart1p1d1
/dev/smart1p2d2
etc.
```

Each device entry can then be mounted at different locations, such as:

/dev/smart0d1 --> /usr
/dev/smart0d2 --> /home
etc.

Using multiple mount points is slightly different from using partitions on the volume in that each mount point has the potential to use the entire space on the volume vs. having a pre-allocated reservation of space defined by the partition sizes. Also, all files and directories of all mount-points will be physically "mixed in" with data from the other mount-points (though files from one will never logically "appear" in the others). Each directory structure is isolated from the others, they simply share the same physical media for storage.

SMARTFS Limitations

This implementation has several limitations that you should be aware before opting to use SMARTFS:

There is currently no FLASH bad-block management code. The reason for this is that the FS was geared for Serial NOR FLASH parts. To use SMARTFS with a NAND FLASH, bad block management would need to be added, along with a few minor changes to eliminate single bit writes to release a sector, etc.

The implementation can support CRC-8 or CRC-16 error detection, and can relocate a failed write operation to a new sector. However with no bad block management implementation, the code will continue it attempts at using failing block / sector, reducing efficiency and possibly successfully saving data in a block with questionable integrity.

The released-sector garbage collection process occurs only during a write when there are no free FLASH sectors. Thus, occasionally, file writing may take a long time. This typically isn't noticeable unless the volume is very full and multiple copy / erase cycles must be performed to complete the garbage collection.

The total number of logical sectors on the device must be 65534 or less. The number of logical sectors is based on the total device / partition size and the selected sector size. For larger flash parts, a larger sector size would need to be used to meet this requirement. Creating a geometry which results in 65536 sectors (a 32MByte FLASH with 512 byte logical sector, for example) will cause the code to automatically reduce the total sector count to 65534, thus "wasting" the last two logical sectors on the device (they will never be used).

This restriction exists because:

The logical sector number is a 16-bit field (i.e. 65535 is the max).

Logical sector number 65535 (0xFFFF) is reserved as this is typically the "erased state" of the FLASH.

Ioctls

BIOC_LLFORMAT

Performs a SMART low-level format on the volume. This erases the volume and writes the FORMAT HEADER to the first physical sector on the volume.

BIOC_GETFORMAT

Returns information about the format found on the volume during the "scan" operation which is performed when the volume is mounted.

BIOC_ALLOCSECT

Allocates a logical sector on the device.

BIOC_FREESECT

Frees a logical sector that had been previously allocated. This causes the sector to be marked as "released" and possibly causes the erase block to be erased if it is the last active sector in the it's erase block.

BIOC_READSECT

Reads data from a logical sector. This uses a structure to identify the offset and count of data to be read.

BIOC_WRITESECT

Writes data to a logical sector. This uses a structure to identify the offset and count of data to be written. May cause a logical sector to be physically relocated and may cause garbage collection if needed when moving data to a new physical sector.

Things to Do

- Add file permission checking to open / read / write routines.
- Add reporting of actual FLASH usage for directories (each directory occupies one or more physical sectors, yet the size is reported as zero for directories).