A Fat Filesystem for NAND Flash

Jens Kager
uni@j-k.at
0026337
033 535

Fritz Praus
fritz@praus.at
0025854
086 881

September 16, 2009

There already exists an (incomplete) implementation of the FAT file system for the JOP (written by Rainhard Raschbauer) which has been used with pre-formatted SD cards.

We decided to make it usable with the NAND flash, by adding an interface layer for reading/writing into the NAND and a method for creating a new file system in an empty NAND.

Section 1 gives an overview of the implementation. Section 2 describes the modifications to the existing code and Section 3 explains the added classes. Finally, Section 4 lists the open issues and Section 5 concludes.

Contents

1 Introduction

The goal of this project was to produce a simple and small filesystem for the NAND flash which is able to handle just a few files and directories. A FAT filesystem [Wik09a] for SD cards already exists for the JOP [Ras09]. We decided to adapt it to work with the NAND flash, due to the follwing reasons:

- Only slight adaptions for existing applications will be necessary to read/write the NAND instead of the SD card. In particular, the interface to the FAT filesystem has nearly been untouched. Only the constructor of `FileInputStream` and `FileOutputStream` have been extended to switch between writing to SD card and the NAND (see Section 2).
- Modifications to the FAT filesystem can be tested on SD card and standard filesystem check tools can be used to verify the correctness. This eases the development effort.
- The FAT layout is quite similar to the NAND layout. Within FAT, a file consists of at least one cluster, which itself consists of multiple sectors. The NAND consists of a number of 16 KByte blocks which are separated into 32 pages of 512 Bytes each. In our implementation, we set the sector size of the FAT to 512 Bytes to coincide with the flash granularity. The sectors per cluster are adjustable to allow the user to decide between smaller but more files or bigger but less files. Currently we defined a cluster to contain 4 sectors which results in a minimum file size of 2 KBytes.

Of course, care has to be taken with respect to the wearout of the NAND flash (see Section 3.2.

Figure 1 describes the implementation from a top down view. `FatTest` contains the main routine and small test methods to create the filesystem and perform a read-/write test. `FatTest` creates a `FileInputStream` and `FileOutputStream` object, which in turn create a `FatLowLevel` instance and initialize it. The `FileInputStream` and `FileOutputStream` classes statically access the `FatIts`, which calls the corresponding methods of the existing `FatLowLevel` interface.

2 Modifications

The existing code has been copied from `joptarget/app/sdcard` to `joptarget/common/fat` in order to better fit the directory structure and allow manipulations.

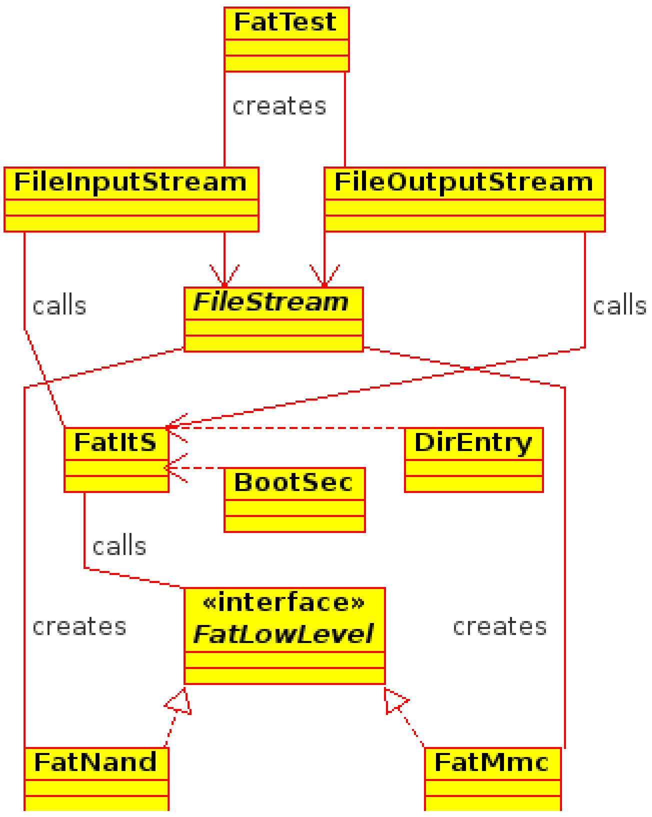

Figure 1: Implementation overview

2.1 joptarget/common/util/Nand.java

- The `nand.read()` was not public and could not be called. Fixed.

2.2 joptarget/app/sdcard/BootSec.java

- This class has been extended with a `construct` method which creates a bootsector compatible with this implementation. For details see comments in the code and [Wik09a]#Boot_Sector.

2.3 joptarget/app/sdcard/DirEntry.java

No modifications.

2.4 joptarget/app/sdcard/FatItS.java

- All methods have been extended with a `FatLowLevel` parameter which specifies where the FAT accesses go (SD/NAND).
- A `fat_mkfs` method has been added which creates a minimal filesystem compatible with this implementation. This method first clears the medium, constructs a minimal master boot record [Wik09b] and a bootsector and finally writes them to the medium.
- A `fat_unlink` dummy has been added which deletes a file.
- A `dump_sector` method has been added which dumps a Sector to `System.out`.

2.5 joptarget/app/sdcard/FatMmc.java

- The class has been adapted to fit the `FatLowLevel` interface.
- A minimal `ClearMedium` method has been added to erase the medium.

2.6 joptarget/app/sdcard/FatTest.java

- A static variable `TESTTYPE` has been added which specifies which medium is used in the test.
- A method `dumpMedium` has been added to dump the first 10 sectors to System.out;
- A method `createfs` has been added to create the filesystem.
- Some minor `System.outs` for better debugging.

2.7 joptarget/app/sdcard/FileInputStream.java

- The class has been altered to extend FileStream.
- The constructor has been extended with an additional parameter `streamtype` (`FileStream.STREAM_TYPE_MMC` or `FileStream.STREAM_TYPE_NAND`).

2.8 joptarget/app/sdcard/FileOutputStream.java

- The class has been altered to extend FileStream.
- The constructor has been extended with an additional parameter `streamtype` (`FileStream.STREAM_TYPE_MMC` or `FileStream.STREAM_TYPE_NAND`).
- The `close` method now flushes the caching buffer of the underlying medium.

3 Extensions

3.1 joptarget/common/fat/FatLowLevel.java

- The interface defines the minimum required methods to work with the FatItS class. See source code for documentation.

3.2 joptarget/common/fat/FatNand.java

This class implements the necessary steps in order to make the FAT filesytem work on NAND flashes. These treatments are:

- Implementation of the `FatLowLevel` interface.
- Translation of the FAT byte address to block and page address of the NAND. The NAND is filled page by page and block by block from the configurable constant `block_offset` (typically block 0, page 0).
- Bitwise invertion of the FAT data which is being written. FAT relies on empty blocks being equal to 0x00 and full blocks being equal to 0xFF. Erasing the NAND, however, sets all pages to 0xFF and thus the FAT filesystem would consider the medium being full. Simple bitwise invertion solves this problem and also reduces the wearout for growing or adding files to the filesystem, since the corresponding bytes in the FAT can be written directly (without erasing the block before and then writing the altered bytes). Only deletion of files would require a erase and rewrite of the block.
- The FAT is normally read and written quite often when accessing files. To reduce the wearout a caching of the first 5 NAND blocks has been implemented. These blocks will be written to the medium when `FatLowLevel.Flush` is called. This behavior can be en-/disabled with constant `useBuffer`.

- Implementation of a Read-before-Write mechanism, which might help mitigate NAND wearout (data is not programmed again if it has already been written). This behavior can be en-/disabled with constant `readBeforeWrite`.

3.3 joptarget/common/fat/FileStream.java

- This abstract class creates the `FatLowLevel` objects to access the medium.

4 Open Issues

- Fragmentation of the medium is not handled.
- The deletion of files has to be implemented.

5 Conclusion

Although not being suitable for use on NAND flashes, the adapted FAT filesystem allows easy treatment of files. The wearout of the flash (especially with the location of the FAT) has been minimized with caching and inverting functions.

References

[Ras09] Rainhard Raschbauer. Jop sd card, 2009. http://www.jopwiki.com/SD_Card.

[Wik09a] Wikipedia. File allocation table, 09 2009. http://en.wikipedia.org/wiki/File_Allocation_Table.

[Wik09b] Wikipedia. Master boot record, 09 2009. http://en.wikipedia.org/wiki/Master_boot_record.