Expertise
and insight
for the future

Hieu Tran

Implementation of High Integrity File System for Serial Flash Memories

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

11/11/2019

Author Title Number of Pages Date	Hieu Tran Implementation of High Integrity File System for Serial Flash Memories 38 pages + 1 appendices 11 November 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Smart Systems
Instructor	Keijo Länsikunnas, Senior Lecturer

The invention of flash memory at Toshiba in 1980 had a great effect on development of computer storages. Accessing performance was improved with flash storage over hard disk drive, and the new type of storage quickly became the popular choice for secondary storage. With the new type of storage device, new file systems were designed specifically to get the most optimization of its characteristics. One important attribute flash technology has is that the number of erase cycles for each memory block is limited; passing this limit, the memory cell is no longer reliable to store data. In embedded systems, the host should have a scheme to deal with this limitation of flash storage, to prolong the lifetime of the device. It is only logical to implement a such feature along with the file system, thus allowing files organization and wear issue awareness in the same design.

The objective of this thesis is to propose a file system design that is not only simple and does not require much resources in embedded systems, but also provide an approach for wear leveling in flash memory. Embedded systems are prone to unexpected power failure, thus another goal the file system design needs to address is ensuring data integrity in such conditions.

The hardware used for the development were Raspberry Pi as computer host and raw NOR flash which can be controlled over SPI communication lines. The file system was written in C programming language, thus a C SPI library for Raspberry Pi was required to provide API for communicate over SPI protocol, which was WiringPi library.

Differences in the design of the file systems can affect the performance and functionality of the underlying storage medium. With the correct choice of design, certain file systems are more suitable for flash storages than others. Therefore, choosing the file system for a system must always comply with the purpose of the application as well as the targeted storage device in use.

Keywords	File system, flash memory, flash file system, NOR flash, SPINFS

Contents

Appendices

Appendix 1. Macro definitions for S25FL164K Flash Memory

List of Abbreviations

API	Application Programming Interface
BSD FFS	Berkley Software Distribution Fast File System
BFS	Be File System
Ext2	Second Extended File System
FAT	File Allocation Table
FS	File System
GUI	Graphical User Interface
IC	Integrated Circuit
MLC	Multi-level Cell
MOSFET	Metal-oxide-semiconductor field-effect transistor
OS	Operating System
P/E	Program/erase
POSIX	Portable Operating System Interface
RAM	Random Access Memory
SLC	Single Level Cell
SPI	Serial Peripheral Interface
SPINFS	SPI NOR File System

1 Introduction

Since the birth of modern computers based on the abstract model of Turing machine in 1936 [1], computers have evolved significantly from the purpose of doing arithmetic computations into general purpose that could solve any of the mankind's daily life problems; and the principle for this leap is the way computers treat input information, including computing, storing and manipulating data. Computers store data in a device called storage, which advances from punch cards, magnetic tapes to hard disks, semiconductor chips as today the main storage device types.

In the early days, when computers only performed one computation, the entire storage operated like one big file; and data started at the beginning of the storage, then filled up in order up to the storage capacity as output was produced. However, as computational power and storage capacity improved, it became possible and useful to store more than one file at a time. Unfortunately, storage devices have no notion of files since they are only mechanisms for storing binary bits, hence the need for a system that can keep track of all files in the storage emerged, and it is known as file system (FS).

There are many types of computer storage invented and each one of them was made from completely different materials with distinctive attributes. Thus, each type of a storage device requires specific file systems that have the best knowledge of the underlying structure to get the most optimal functionalities and performance.

The most recent technology in storage devices is flash memory, which was built on the ground of semiconductor material, and this kind of memory is quickly becoming the most popular computer storage. Much like transistor count and Moore's law, memory and storage technologies have followed a similar exponential trend and as of 2017, NAND and NOR flash jointly accounted for 49.1 billion US dollars in total 110 billion US dollars of Worldwide Memory Market [2][3]. Figure 1 summarizes NAND flash market from 2015 to 2019 in an upward curve, and predicts the continued trend in 2020.

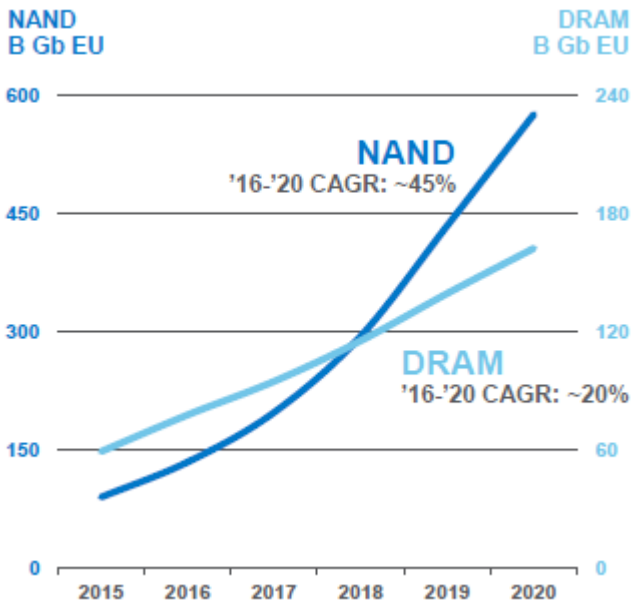


Figure 1. Memory Market Bit Demand Forecast [3]

The motivation for this thesis topic arose from the current work environment of the author at Tuxera Inc. The author's workplace is a software company located in Espoo, Finland who expertizes in embedded file systems and storage solutions [4]. In the path of gaining more knowledge to benefit daily work tasks, the author set the objective to write his own file system with flash memory as the targeted storage. By following the objective, the author would learn about the basics of file system design, which is the main product Tuxera Inc. offers, and understand technology behind the latest generation of storage device, flash memory.

The goal of the study is to program a simple but functional file system software for flash memory that could be used in low resource embedded applications. Two requirements this project design needed to tackle were overcoming the wear limitation of flash technology, discussed in Chapter 2.3.2, and preserving data integrity in any chance of power failure or disruptive communication.

2 Theoretical Background

This chapter introduces the topics required for understand the concept of file system and flash technology. Firstly, a general definition of file system, consisted components and related terminologies in computing must be defined and mentioned. Afterwards, introduction to some existing file systems with their traditional designs is discussed. Lastly, this chapter covers flash storage memory and their effect on the legacy file system designs.

2.1 File System

A File system is a piece of software that allows Operating System (OS) to manage data on permanent storages. These permanent storages, which often are optical disks, hard disks, flash storage drives, etc., are used to store computer data and information for long-term even without electrical power [5]. Therefore, the main function of file systems is to provide a way to organize, store, retrieve, and manipulate data and information on such kind of storages. To bring the most natural experience for end users when storing and retrieving computer data to and from persistent storage devices, hierarchy concept has been adapted to the computer world with two main components: file and directory [6]. Thus, file systems are usually built around what is called file and directory.

2.1.1 File and Directory

A file system is formed from the concept of computer file and directory. From human world, books in libraries are stored with careful thought about ease for later look up and retrieval, hence they are arranged into shelves sorted by family name of the authors or categorized by their subjects. A file system can be compared to a library where a file is a book that has individual piece of information and directory is a shelf that holds multiple books or files that share common attributes, an example of files hierarchy is illustrated in Figure 2. This relationship between file and directory makes organizing and looking for computer data more modular.

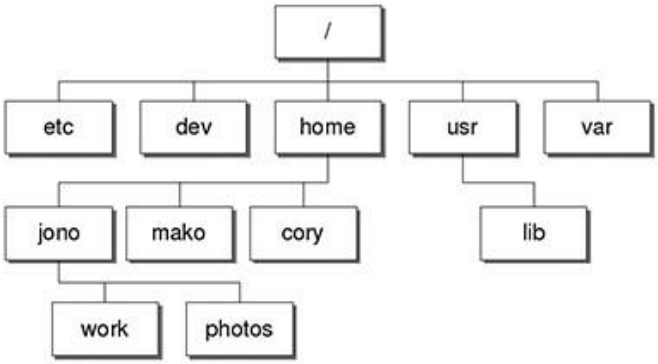


Figure 2. File and Directory hierarchy example in Linux

From the file system point of view, a file has information used by the computer, and in order to query for the exact piece of data stored in a file, a filename must be assigned to the file. As a result, the job of file systems is storing files, which includes the raw data or "stream of bytes" and its names, on the storage device, then should be able to return back the content of the file when computer requests by passing the filename.

Most computer users are familiar with the functionality of a file or a directory from direct experience when using the computers. Computer programs write data permanently to storage drive into files, which can be simply seen as a piece of information. This piece can be texts, graphical images, data structures, any combination of data types or arbitrary bytes only the program can interpret.

Dominic Giampaolo hinted on the important role of the file system when storing a file on a physical medium: "A file appears as a continuous stream of bytes at higher levels, but the blocks that contain the file data may not be contiguous on disk." [7, 11]. File systems are responsible for mapping the file's logical memory as physical memory on a disk, thus when there is a write request, the correct physical blocks will be modified and data on the corresponding blocks needs to be returned in the case of a read request.

Directory, similar to the term Folder in Windows Operating System, is the way a file system organizes multiple files. A directory can be considered a file, but its content is a list of names, these names can be names of files or names of sub-directories (directories living under a directory). The policy of storing this name list in storage plays a crucial part in file lookup performance. For instance, if the list is unsorted, searching for a file name in the list must scan through the whole list and in the worst-case scenario the wanted file

appears to be the last file in the list. Therefore, normal file systems take advantage of sorted data structure when storing the content of the directory for efficiency in the lookup operation.

2.1.2 Terminologies

It is necessary to define some computer terms referring to certain concepts that make up a file system for better understand the relationship between storage devices and file systems in computers. The list below is put down with dependency ordering, the previous definitions are building blocks for the next ones.

- Partitions: are divisions or parts of a real hard disk drive. A partition is a logical separation from the whole drive, but it appears as though the division creates multiple physical drives, which can be managed separately. A partition is a logically independent section of a hard disk drive that contains a single type of filesystem.

- Volume: refer to a disk or partition that has been initialized with a file system.

- File system instance (or filesystem): refer to all the files, directories and file system data structures stored in a partition of a device.

- Metadata: is data information that provides knowledge and description about other data. For example, creation time of a file is a valuable information about a file, but it is not part of the data stored in the file.

- Superblock: a small area on a volume where characteristics of the filesystem currently residing on that volume are stored. Superblock usually contains volume size, block size, layout and counts of empty blocks, with other file system specific metadata such as size and location of i-node tables, disk block map and usage information.

- I-node: is a data structure of a file system implementation, usually on Unix-like operating systems, where all information about a file except its name and its actual data are stored. The i-node also provides the connection to the physical locations on disk which have the file data.

With the definitions above, understanding how a file system is implemented is possible since they are all involved in the ways computers, file systems and storages determine and communicate to each other.

2.1.3 Operations

This chapter introduces some basic operations involved around the file system, how a file system would work based on two basic abstractions of files and directories. These operations are not necessarily corresponding to user-level operations, the former might be executed internally by the OS and do not send any notification to the users.

- Initialization:

  The first operation to attach a file system to any fresh partition in a drive is always creating an empty file system on a given volume. The volume's characteristics such as total size and the user's preferences will be taken into account when creating and placing internal data structures of a file system onto the fresh volume. Most of the time, these non-default options are related to the size of the superblock and the allocating method for storing new entries on the memory layout.

  In the Linux world, initializing a partition is usually done by executing mkfs programs from the command line interface (CLI) as shown in Figure 3. The chosen file system is ext4 and the partition /dev/sdb1 were formatted to have ext4 layout residing on it.

```
[0]hieu@ubuntu:~$ sudo mkfs -t ext4 /dev/sdb1
mke2fs 1.44.1 (24-Mar-2018)
/dev/sdb1 contains a vfat file system
Proceed anyway? (y,N) y
Creating filesystem with 4194304 4k blocks and 1048576 inodes
Filesystem UUID: 8a3cf165-576f-45db-955a-95c47d78ef1f
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208,
        4096000

Allocating group tables: done
Writing inode tables: done
Creating journal (32768 blocks): done
Writing superblocks and filesystem accounting information: done
```

Figure 3. Initializing a USB from mkfs program in Linux

On Windows OS counterpart, the users are more familiar with a graphical interface software and the initialization process is referred as Formatting. Figure 4 shows the initializing of a 16 Gigabytes USB with FAT32 file system format and 16 kilobytes of allocation unit size.

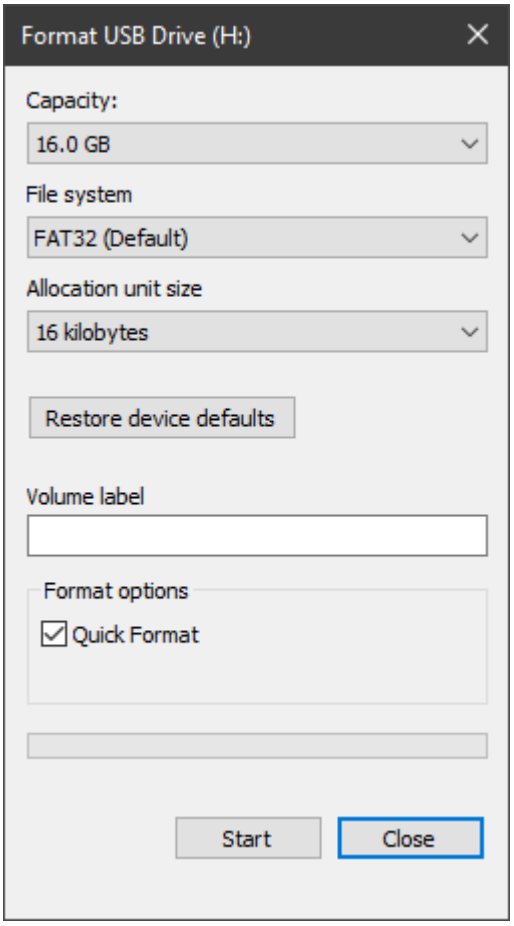


Figure 4. Initializing a USB as Format USB Drive in Windows

During the process of initializing the file system, an empty top-level directory must be created, also known as root directory. All files and directories created after the initialization phase are organized under this root directory in the hierarchy.

- Mounting:

Mounting is the task of accessing the raw storage device, using file system program to read its superblock to compute necessary information for later reading and modifying memory content. Even though mounting is often a user program, the process returns a handle to Operating System for accessing the disk. In

graphical OS environment, mounting is transparent and performed automatically when there is a new storage device connected to computer motherboard. One crucial job when mounting a disk is to check the consistency state of the disk to find out if it was cleanly shutdown last time, or there is a power outage while data was being written to the disk. Such inconsistency can cause corruption to the disk and all data can be lost. In case file system detects unclean shutdown state of the disk when trying to mount, extra tasks of verifying and possibly repairing any damage must take action. These tasks will be handled by another program called file system check, or fsck for short, and it is extremely complex and takes time since the whole storage needs to be checked.

- Unmounting:

  The unmount operation indicates the user no longer has intention to use a particular volume and it should not be available to read or write. The process involves flushing all unwritten data from Random Access Memory (RAM) to physical media, marking the volume to be "clean" indicating that a normal shutdown is performed and removing the accessing handle from Operating System. After unmounting, it should not be possible to access the storage device until the next mount operation.

- File and Directory Manipulation:

  Users should have ability to interact with their data at file level by any actions including creating, opening, writing, reading, deleting, moving and renaming. As mentioned above, a directory is essentially a file where its data is the name list of sub-files and directories. Therefore, most file operations can also be applied to directory.

  After mounting a fresh file system, there is nothing on the volume until a new file is created and some data is stored in the file to be written permanently to disk. Creating a file requires name of the file and which directory the file will reside in. If the volume is empty, a new file will be created under the root directory. File creation only involves allocating i-node and writing metadata of the file, the actual data, stream of bytes in the file will be added later when writing actually happens.

Since having a new file under a directory means the content of the directory has been modified, file system also needs to update the parent directory's content.

When a number of files are presented on the file system, they need to be opened before being able of reading and writing. Similar to mounting a file system, opening a file returns file handle to OS, instead of the volume handle when mounting, and by using this handle, the OS can request file system to read or write to a specific location in the file.

Write operation allows computer programs to store data while read allows extracting information in files. The file system needs a reference to the file, which is returned from the open operation, offset position in the file to begin reading and writing, memory buffer and length of the requested writing or reading data. Writing to a file can increase the file size and, in such case, extra memory blocks will be allocated to append to the end of logical file memory array, which eventually update the file i-node and potentially superblock of the file system depending on its implementation. Writing to a logical address of a file will be translated to a physical address by file system, thus making it permanent. Read operation is much simpler, all the file system needs to do is to map the logical position of the file to the corresponding physical block on disk, which comes down to retrieving data from the storage device and place it to the read buffer.

On the other of creating files, it is only natural to have the option to delete them. The file deletion process first removes the name of the file in parent directory's name list and until there are no more programs with open file handle to the file, file system can free the file's resources by returning occupied blocks to the free block pool and the file i-node to the free i-node list. It is not necessary to set all the memory bits associated with the metadata and data of the deleted file to be zeros. Marking a delete flag in the file metadata and free these blocks in used/free block pool is sufficient for the OS to know previous memory areas are free to be overwritten.

Rename is often regarded as the most complex operation that a file system has to support. There are many checks and validations that must pass before actual the renaming takes place. These checks include whether the source and destination names are in different directories, then the new name can be the same as

the old one, if names refer to directories, the new name cannot be a sub-directory of the old name, etc. Only after all validations are satisfied, the file system will delete the old name entry, create a new name entry in a new destination directory then update the corresponding parent directories.

There are many other features that can enhance the capabilities of a file system, but at least the baseline of functionality was denoted. This basic set of operations provides the minimal amount of functionality needed in a file system. More advanced features such as symbolic links, hard links, attributes and indexing would be required in the implementation of a feature-rich modern computer file system.

2.1.4 Existing File Systems

File systems are usually packed within OS and each OS has their own implementation of file systems. Depending on the underlying physical storage medium, purpose when storing data and host OS, file system type will be chosen for individual media. For instance, CDs use ISO 9660 file system [8], hard disks and managed flash drive such as SSD depend on what is available from the host OS, Microsoft Windows utilizes NTFS [9] while MacOS takes advantage of Apple's APFS with improved Graphical User Interface (GUI) experience [7, 37][10] and Linux has ext4 developed by the Open-source Community [11]. As shown in Figure 5, MacOS has the ability to tag a color to any files or directories thanks to its native file system, which makes querying for those files having the same tag color fast and user friendly.

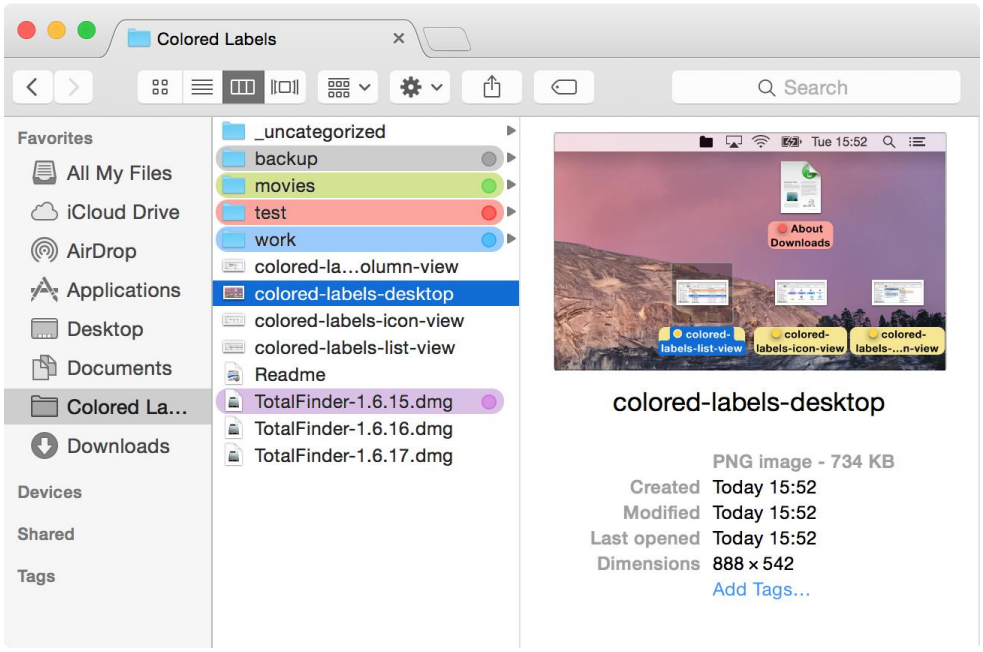


Figure 5. File system with GUI colored labels in Mac OSX

In summary, the file system is a program that enables the Operating System to store, retrieve and manipulate data on a persistent storage device, data here is usually stored as content of files and organized into directories. Each file system targets a specific type of storage on a specific Operating System, though they all must support some basic operations to comply with the existing conventional and modern designs.

2.2 Block-based File System

During the development of modern computers, tech researchers and programmers have experimented with many file system designs, investigated both simple and complex solutions, however, block-based type file system, which is also known as traditional design, stood out both in terms of simplicity in implementation and effectiveness in performance. Block-based file system is straight forward when coming to the explanation of the concept, storage is divided into blocks and file need to be stored in blocks boundary [12]. Any modern file systems based on this central design can trace their inspirations back to the Berkley Software Distribution Fast File System (BSD FFS), who set the robustness and speed standards for Unix file systems in nearly a decade. Traditional design usually consists of a superblock, a block bitmap, an i-node bitmap and a data mapping scheme in form of direct or in-direct blocks. This design can be found in many popular file systems in computer history such as Be File System (BFS), Second Extended File System (ext2), and File Allocation Table (FAT) File System. [7, 33-35]

2.2.1 Internal Components

The following section introduces the underlying basis of the block-based file system. Most of these file systems share the same notion for some internal components which they comprise, for example: superblock, bitmap, i-node, and data stream.

Essentially, there is a known place on a storage device to store the most important block to the file system called superblock. Suitable choices for this place are often the first or the last block of the partition but first block is usually chosen for the ease of seeking (no need to find the size of the partition to calculate the last block number). In superblock, it contains information about block size, total number of blocks, number of used blocks, dirty bit flag, and address of i-node of the root directory. Without this reference from the superblock to the root of the hierarchy of all files and directories, the file system would not able to connect and find any files on the volume.

The following blocks after the superblock can be occupied for bitmap scheme, which is an approach to manage free space of a disk. The bitmap scheme represent each disk block as 1 bit, thus binary value of 0 or 1 in a bit can indicate vacancy status of a block whether it contains invalid data that the file system can freely write or it is not empty and should not be overwritten. The number of blocks used for bitmapping is based on the size of the partition and file system block size, as calculated in Equation 1. Each byte consists of 8 bits, therefore the bitmap for 8GB disk with 1K blocks would requires 1MB of space, or 1024 blocks in this case.

$$block\ bitmap\ size = \frac{disk\ size\ in\ bytes}{file\ system\ block\ size\ \times 8}$$

Equation 1. Block bitmap size equation

Beside metainformation about the file such as the size of a file, access permission information, its creation and modification times, i-node data structure needs to keep track of which locations on disk are belongs to this i-node data stream. This basic structure is the fundamental building block of how data is stored in a file on a file system.

The traditional approach for linking on disk addresses to logical file offsets is storing a list of blocks directly inside i-node, which is called direct blocks. Each entry in this list is a physical block address of the storage device, and since the size of an i-node structure

is limited, it limits the amount of data the file can contain. Generally, about 4 to 16 addresses can be stored directly in i-node, which means the maximum size of a file can be is 16KB with 1KB file system block size. Figure 6 shows the relationship between direct block members of i-node and the physical addresses where those blocks of data are located on physical storage.

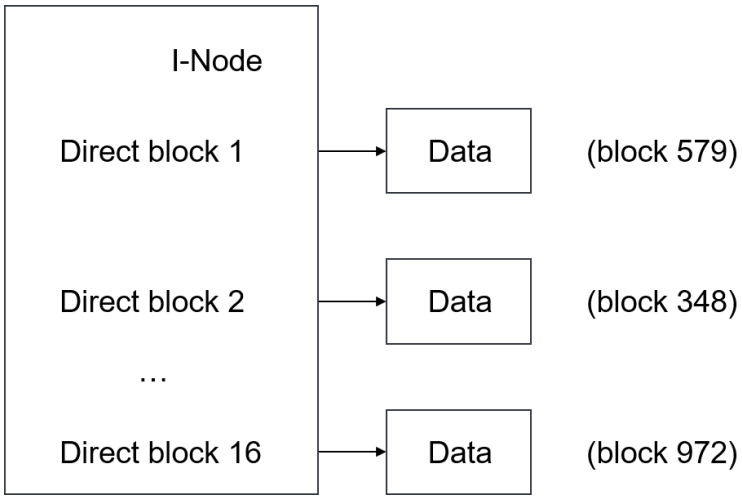


Figure 6. Direct blocks mapping

To address the space constraint of direct block, same concept can still be applied with extra address block in between, called indirect block. Rather than having direct reference to a physical address that has the file data, i-node can carry the block address of this indirectly block. While the data block contains user data, the indirect block has pointers to other data blocks that do have user data in them, which make up the whole stream bytes of file data after concatenating. Therefore, one disk block address can map a much higher number of data blocks, instead of mapping one by one with direct block address. Figure 7 visualizes indirect block mapping and its extended concept, double-indirect block mapping.

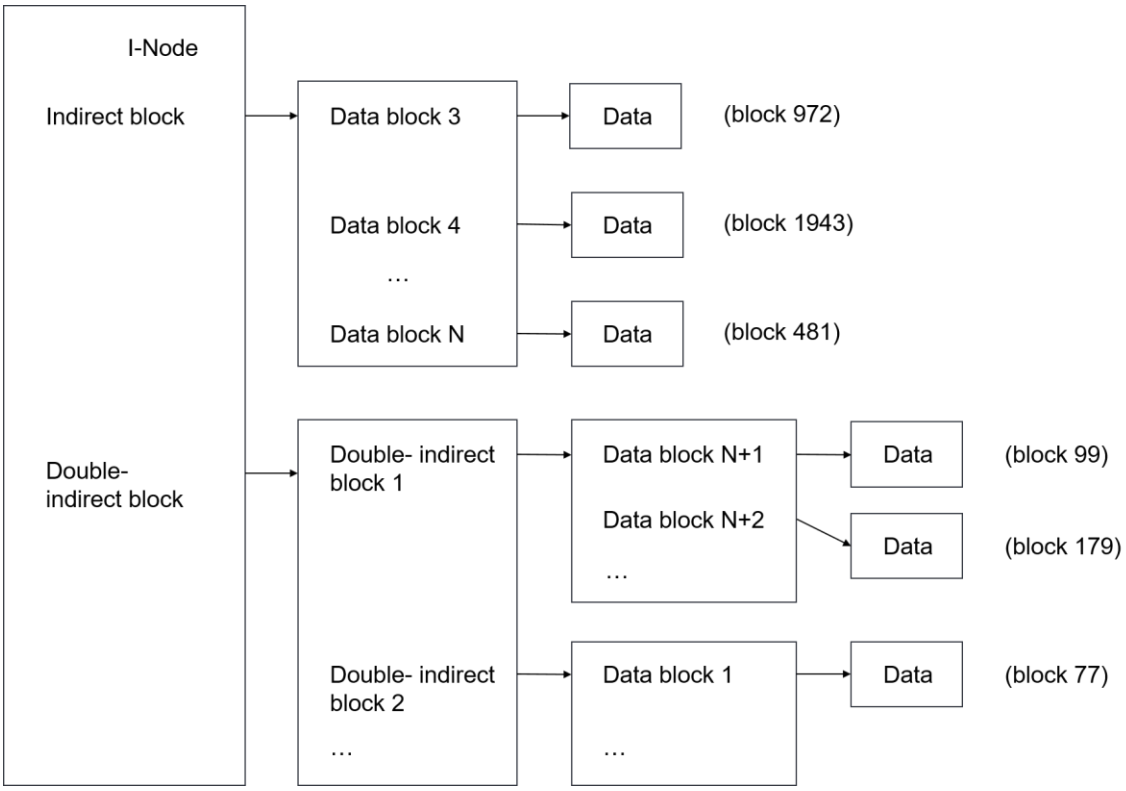


Figure 7. Indirect block and Double indirect block mappings

The indirect block helps increasing the maximum data size of a file an i-node can keep track, however, it is not enough to locate data blocks for a file which is much more than a few hundred kilobytes in size. To overcome this issue to allow an even bigger file, the indirect block technique can be applied a second time, making double-indirect blocks. The same concept and basic idea still hold true for double-indirect block as indirect block. Each double-indirect block address the i-node contains points to a block on a disk whose content is more pointers to indirect blocks and respectively refers to an exponential number of actual data blocks constructing a file.

2.2.2 Advantages

The traditional design of file systems is straight-forward, there are not many abstraction layers between how users see the files and directories on the computer screen and how file system hides its implementation details to store those data on disk. Basically, data in a file is broken down into many chunks of blocks and the file i-node is responsible for keeping track of the locations of the blocks. On top of that, the file system superblock will always have a connection to the address where each i-node resides and return that information whenever there is a request to read/write that file. Subsequently, when accessing a file, in addition to the disk operations of accessing directly the blocks having actual file data, there will be few more reads to the superblock and other blocks to search for the i-node and eventually leading to the data blocks. These additional reads are called file system overhead, and because of the simplicity in the design, traditional file systems' performance suffers little from it and tend to be robust since less complexity means less bugs and corruptions. When putting more optimization features such as bigger block size and caching, performance can even be pushed further.

2.2.3 Disadvantages

Writing any new data to the disk might require updating the file system housekeeping data structure, which are superblock and bitmap, and having these areas to get frequently updated is a big disadvantage in the design. It really depends on the underlying storage device, continually changing the content in a part of a hard disk drive might not be a problem, but doing the same thing in modern flash drive is not at all recommended due to the drive's characteristics and operations involving in writing data to a memory block discussed in Chapter 2.3 on Flash Memory. One key disadvantage of the flash memory is that it can only endure a relatively small number of write cycles in a specific block. Therefore, the block-based file system is not a suitable choice to be placed on top of a flash storage device.

2.3 Flash Memory

Flash memory is a modern type of computer memory constructed from semiconductor components and used primarily in persistent storage mediums. This memory type falls into the category of non-volatile technology meaning that power is not required to retain data in the memory making it a perfect match to use as secondary storage, or long-term persistent storage.

Flash memory was invented by Fujio Masuoka while he was working at Toshiba in 1980 and made commercially the first introduction to the market in late 1980s [14]. The key component that composes of individual flash cells is a floating-gate metal-oxide-semiconductor field-effect transistor (MOSFET), also known as a floating-gate transistor. Figure 8 illustrates the cross-section of a normal MOSFET.

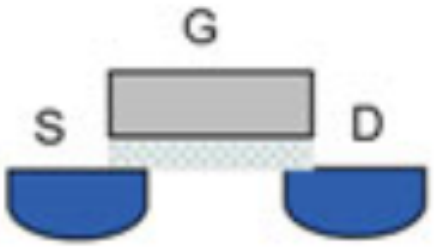


Figure 8. Cross-section of a MOSFET, showing gate (G), source(S), and drain(D) [13, 12]

Different from a normal MOSFET shown in Figure 8, a floating-gate MOSFET has an extra gate called floating gate added between the control gate and the body of the transistor as indicated in Figure 9.

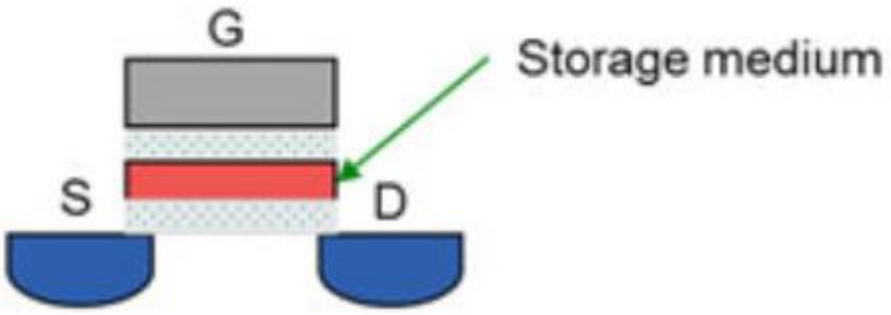


Figure 9. Cross-section of a floating-gate MOSFET, which has a floating gate in between. [13,12]

The floating gate is separated from the control gate and the body by the oxide insulating layers preventing any electrons on floating gate to escape easily. Having this electrical isolated element allows charges on floating gate to stay for long periods of time, and flash memory utilizes this characteristic to store the cell's value on the floating gate. Theoretically, if there is no electron on the floating gate, the cell is in the erased state and the bit value is known to be 1, whereas having electrons on the floating gate presents the bit value of 0. [13]

2.3.1 Operating Principle

How read and write operations in flash storage device are performed is discussed in this section with the principles down to the flash memory cell level. As previously introduced, each memory cell consists of a floating-gate transistor which can have two charge states on the floating gate, either no charge citing bit value of 1 and negative charge implying bit value of 0, and this type of two states cell is called single-level cell (SLC) and capable of storing one bit of data. However, to achieve larger bit density in a single flash integrated circuit (IC), multi-level cell (MLC) technology was developed to allow more than one bit to be encoded per floating gate, resulting in increasing maximum capacity of flash memory significantly. The theory behind MLC is based on the fact that the number of electrons which can be charged into the floating gate is a variable. Therefore, instead of having only charge and no charge states, floating gate can analogously have as many charging states as it wants depending on the amount of electrons presented on floating gate. Even though technology always seeks the greatest number of possible states, it comes with a downside when evaluating state of floating gates. More charge states express smaller tolerance in distinguishing them, as described in Figure 10, leading to higher chance of errors both in determining the amount of charges and injecting correct number of electrons. Considering the size of a typical flash cell is only 40nm [15], measuring any aspects of the cell requires extreme precision. Until today, most common types of MLC flash devices have 2 to 4 bits per cell, which requires 4 to 16 states of the floating gate to be distinguished [16]. Figure 10 represents the shrink in reliability margin due to the decrease in size of flash cell and the increase in number of bits.

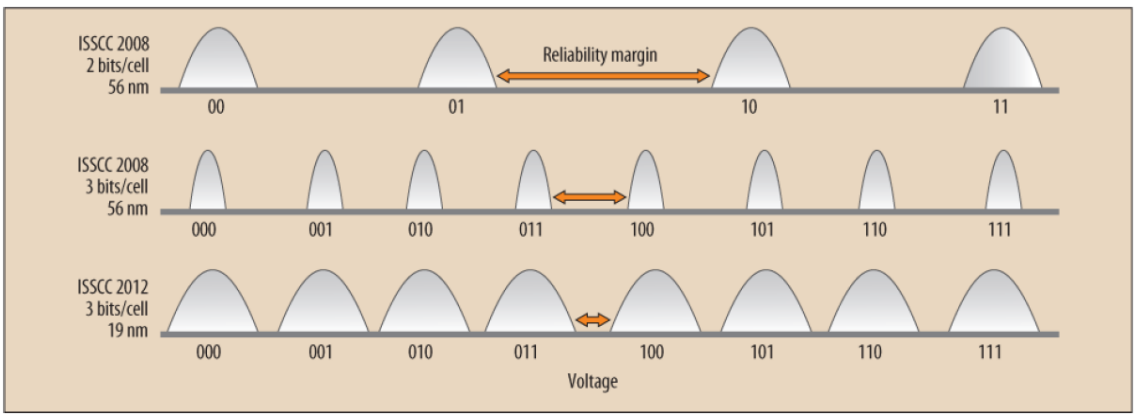


Figure 10. Reliability margin is inversely proportional to number of bits per cell.

Reading the value of a cell is basically detecting which charge state the floating gate is having, and from that charge state, bit value can be translated. Based on the principle

operations of MOSFET, if there is a threshold voltage applied to the gate, the current will flow from source to drain [17]; applying some voltage to the control gate of the floating-gate MOSFET, the existent of current from source to drain can disclose the charge state of the floating gate. Electrons on the floating gate act as a negative mask layer which screen off some positive charges on the control gate, subsequently having an effect on controlling the current. The strength of this influence depends on the amount of negative charges on the floating gate. Therefore, the voltage required for the control gate when the bit value is 0 is higher than when the bit value is 1 (no electrons on the floating gate) in order to have current flows in the drain terminal. Figure 11 demonstrates the relationship between the current flowing from source to drain versus gate to source voltage in an IV (Current versus Voltage) graph with two “turning on” curves starting at V_{T1} and V_{T0}, corresponding to two threshold voltages of different charge states. [13, 13]

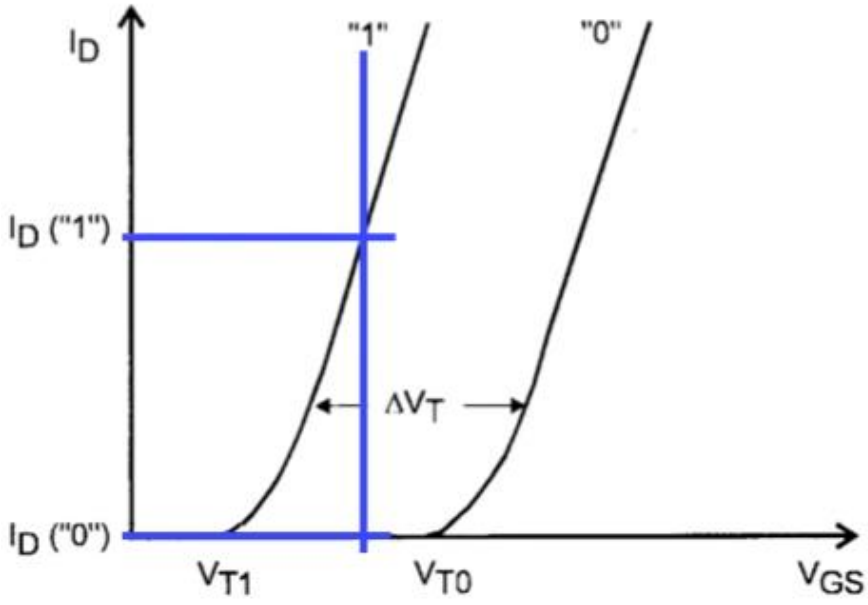


Figure 11. Current vs Voltage graph with two charge states, or 1 bit/cell.

The threshold voltage for erased state is called V_{T1} and threshold voltage for programmed state is called V_{T0}; and according to earlier statement, $V_{T0} > V_{T1}$. An intermediate voltage in the middle of V_{T1} and V_{T0} indicating by the blue line on can be applied to measure the current. When identifying the charge state on the floating gate to determine the bit value of a flash cell, no reading on the current value means the transistor is on “0” curve correlating to bit value 0, while a positive current reading means the other curve and bit value is 1. [13, 14]

The approach for the MLC flash memory is not too different, instead of two curves account for two threshold voltage levels, there are four curves, one for each charge state, present in Figure 12 IV graph if each cell encodes 2 bits of data. On this condition, there are three intermediate voltages and by reading the current value when applying each one voltage difference from gate to source, the bit values can be identified.

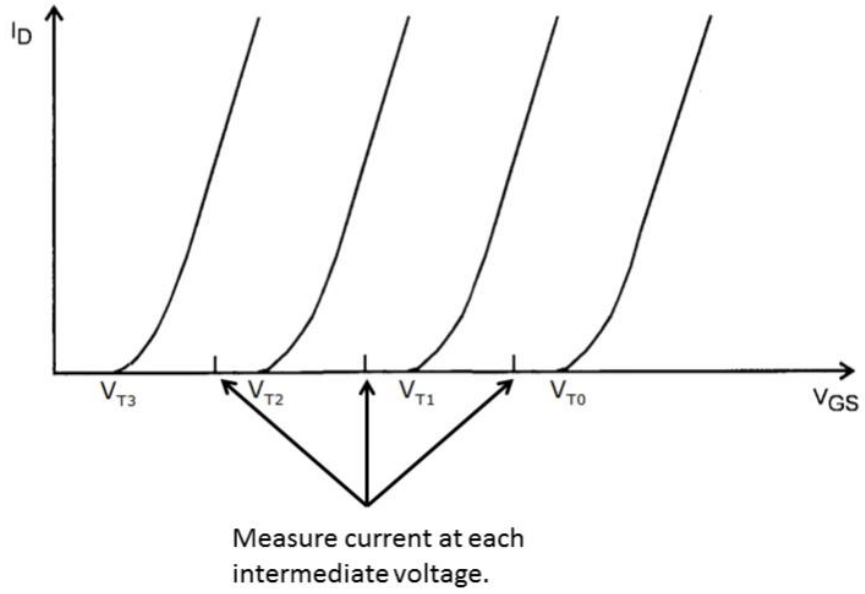


Figure 12. Current vs Voltage graph with four charge states, or 2 bits/cell.

Beside reading, the other fundamental operation on a flash cell is writing, which is the process of alternating the number of elections on the floating gate to reach different charge states. Despite no electrical contact to the floating gate, a few methods of sending charges through the oxide layer have been discovered. The first method is Quantum Tunneling and the second one is Hot Electron Injection. This document does not address these subjects in details; however, in short, Quantum Tunneling is based on quantum physics to shrink the oxide gap between control gate and floating gate to push electrons off from control gate to floating gate and vice versa, whereas, Hot Electron Injection increases kinetic energy of electrons from body substrate to let them hop over the insulating layer onto floating gate. [13, 14-16]

2.3.2 Hardware Limitation

Flash technology is gaining its popularity and quickly becoming the successor to replace hard disk drives as secondary storage due to its faster read and write speed since there is no mechanical part movement associated like in the latter [18], though one major limitation it has is that flash memory cannot stand the test of time. This issue of the technology is called memory wear, since the flash memory has a finite number of program – erase (P/E) cycles before it becomes unreliable for reading. The reason behind memory wear connects directly with the current ways how the memory cell is written. Both writing methods, Quantum Tunneling and Hot Electron Injection, require high voltages and high electric fields to be applied on the transistor, which causes the electrons to gain huge energy then dissipate that energy by colliding with the oxide layer lattice [13, 157]. This damage builds up overtime and oxide insulator wears out eventually, causing electrons leakage on the floating gate, thus charges get altered unintendedly and bit value is encoded wrongly.

On MLC flash memory, the wear problem has even a greater effect due to the small tolerance of the reliability margin. Illustrated in Figure 13, the threshold voltage region of each charge state is much smaller than in SLC memory and potentially crosses over others when worn out, represented by the dotted grey line, hence a tiny error in the number of electrons can cause incorrect reading when applying the same factory defined intermediate V_{THR} voltages for measuring.

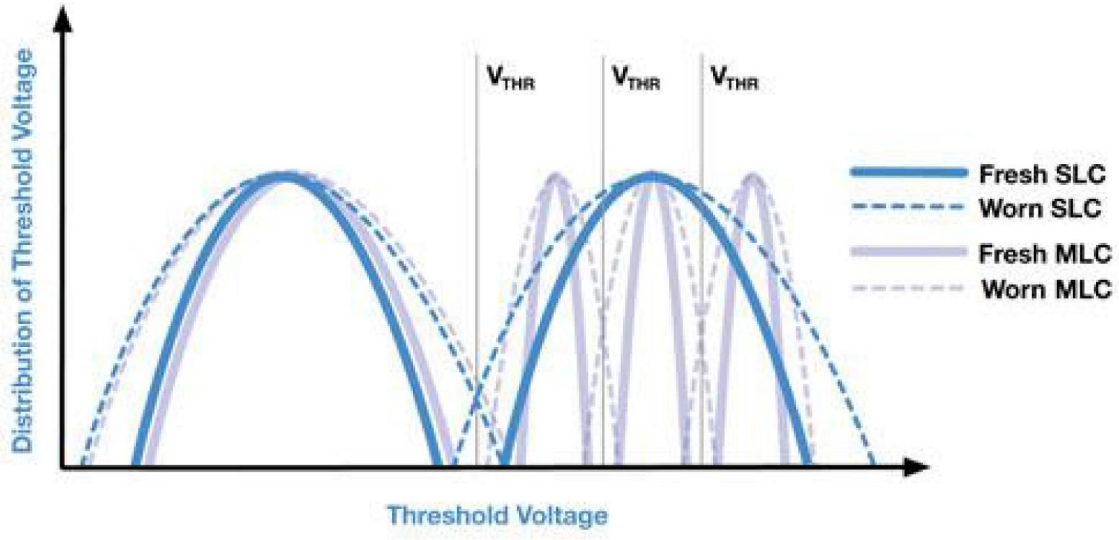


Figure 13. Shifting in Threshold Voltage of worn SLC and MLC flash memory.

Apparently, due to the smaller tolerance involved for MLC, it takes less wear to make MLC flash unusable. A typical maximum number of writes before wearing happens for SLC is 100,000 P/E cycles while this number for MLC only ranges from 1,000 to 10,000 cycles [19].

The consequences of the wearing effect only rise when some blocks in the whole flash memory have significantly higher erase count than the rest of the blocks because their data were constantly focused for rewriting. The heat map in Figure 14 shows an example of such a situation where about 20% random locations in the flash is reaching the maximum P/E cycles and other flash blocks still has considerably low erase cycles.

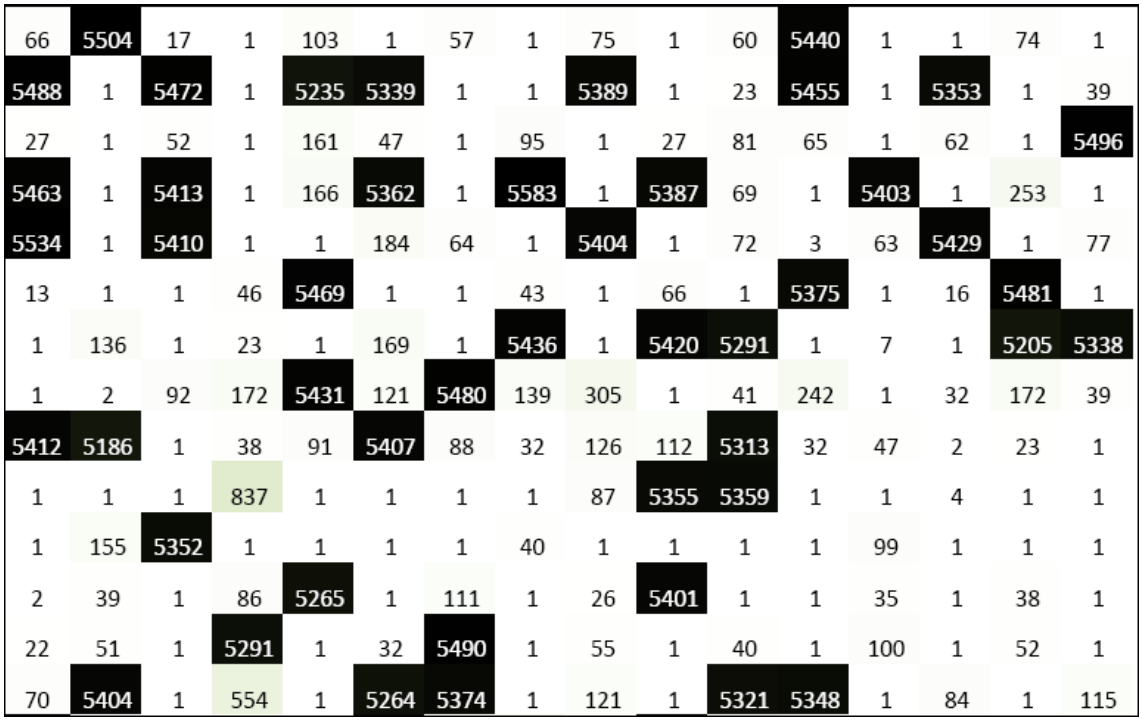
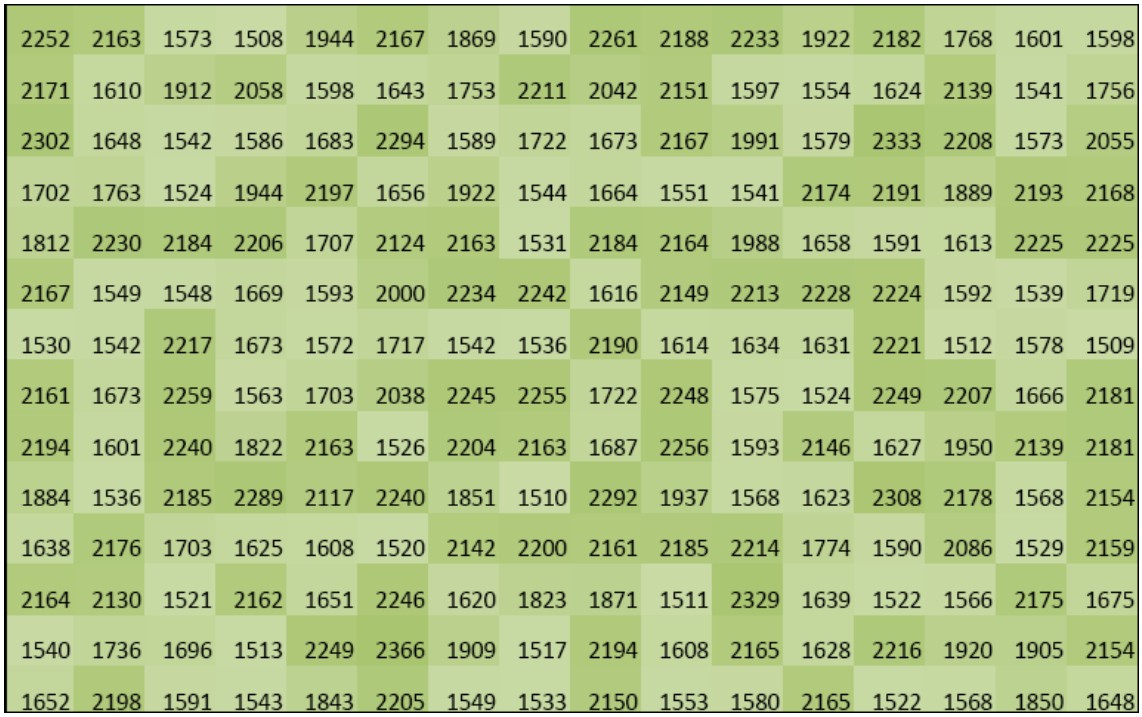


Figure 14. Erase counts on simulated NAND media without wear leveling [20]

In such condition, once those higher rewritten rated blocks wear out, the whole filesystem residing on this flash memory is acknowledged to be corrupted and the flash is not meant for continued use, even though 80% of the flash is still in good shape. Therefore, it is not a good use case of flash memory to get frequently updated and rewritten partially, this will decrease the life span of the flash's tremendously. In order to extend the practical used time of the flash memory, data written to the storage should be distributed equally to all blocks, thus having erase counts across the flash to be closed to each other. Figure

15 portrays what a healthy flash memory array should look like when all blocks have approximately similar P/E cycles.



2252	2163	1573	1508	1944	2167	1869	1590	2261	2188	2233	1922	2182	1768	1601	1598
2171	1610	1912	2058	1598	1643	1753	2211	2042	2151	1597	1554	1624	2139	1541	1756
2302	1648	1542	1586	1683	2294	1589	1722	1673	2167	1991	1579	2333	2208	1573	2055
1702	1763	1524	1944	2197	1656	1922	1544	1664	1551	1541	2174	2191	1889	2193	2168
1812	2230	2184	2206	1707	2124	2163	1531	2184	2164	1988	1658	1591	1613	2225	2225
2167	1549	1548	1669	1593	2000	2234	2242	1616	2149	2213	2228	2224	1592	1539	1719
1530	1542	2217	1673	1572	1717	1542	1536	2190	1614	1634	1631	2221	1512	1578	1509
2161	1673	2259	1563	1703	2038	2245	2255	1722	2248	1575	1524	2249	2207	1666	2181
2194	1601	2240	1822	2163	1526	2204	2163	1687	2256	1593	2146	1627	1950	2139	2181
1884	1536	2185	2289	2117	2240	1851	1510	2292	1937	1568	1623	2308	2178	1568	2154
1638	2176	1703	1625	1608	1520	2142	2200	2161	2185	2214	1774	1590	2086	1529	2159
2164	2130	1521	2162	1651	2246	1620	1823	1871	1511	2329	1639	1522	1566	2175	1675
1540	1736	1696	1513	2249	2366	1909	1517	2194	1608	2165	1628	2216	1920	1905	2154
1652	2198	1591	1543	1843	2205	1549	1533	2150	1553	1580	2165	1522	1568	1850	1648

Figure 15. Erase counts on simulated NAND media with good wear leveling [20]

To guarantee written data is distributed equally across the whole flash, at least one type of wear levelling should be present in the whole system. The wear levelling solution can be accomplished in the form of hardware as an extra IC in the same package of the flash or executed as software directly from the host computer. In summary, if no wear levelling was integrated in a system that use flash storages to even out the load, applications might end up writing to the same block of the memory repeatedly, thus making that block of memory unusable quickly.

2.3.3 Managed Flash vs. Unmanaged Flash

Modern flash storages can be divided into two types, managed flash and unmanaged flash, based on the existence of a component called controller in the same storage package.

Popular memory devices such as SSD, SD, eMMC, UFS found in PCs, laptops, mobile phones nowadays are all managed flashes. There is a controller on the device that do many memory management applications specifically for the current type of flash such as error correction, wear leveling, bad block management, etc. By having a dedicated controller for the flash in the same package, all management operations are handled by the memory internally, lifting the burden from host computer to take extra care for any of these services. Moreover, an integrated controller has a better understanding of the underlying structure and characteristics of the flash than any external controller can possibly support, proving that an internal controller is likely to have better optimizations at doing any above-mentioned operations.

Unmanaged flashes are also known as raw NAND/NOR chips, which essentially consist of only memory array ICs without any kind of dedicated memory management component. This type of storage tends to have smaller footprint and lower cost comparing to its managed counterpart, thus suitable for embedded applications where they are used in often limited resource systems. Since there is no onboard memory management, all responsibilities to provide these instructions rely on host side. Therefore, when implementing the file system targeting to the use for such devices, wear leveling needs to be taken into consideration somehow that data will be written evenly to all blocks of the flash. If the design of the file system is comprehended with the traditional approach described in Chapter 2.2, the first few blocks of the flash where superblock and bitmap are resided will have higher P/E cycles than any other blocks.

The graph below, Figure 16, shows an example of using an FAT file system (which uses some beginning blocks for storing file system metadata, i-node table in this case) directly on a raw flash without wear leveling, which should always be avoided.

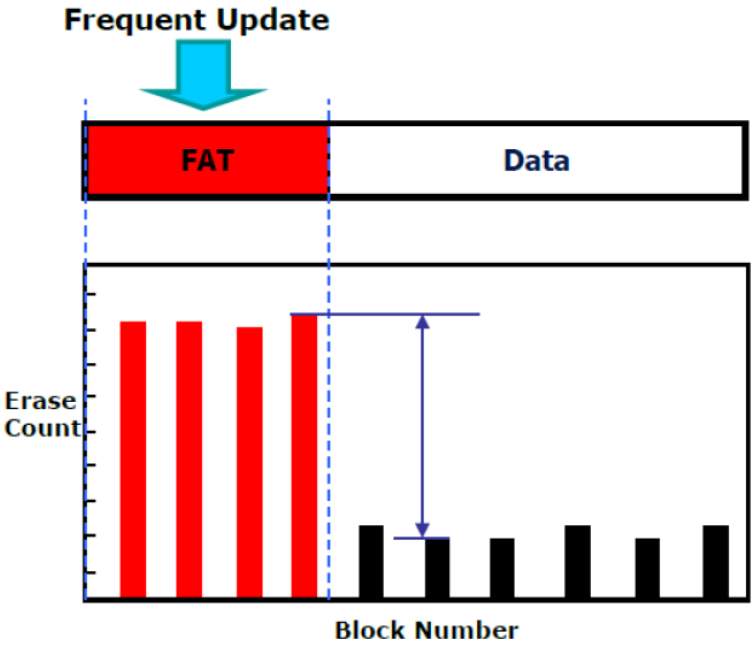


Figure 16. Erase counts of a flash storage using FAT file system

To sum up, traditional file systems are inadvisable choices to manage files on top of raw flashes because of an obvious reason, they are wear-levelling intolerant. Therefore, new file system designs need to be invented to conform with the natural characteristics of flash memory.

3 Design and Implementation

This chapter contains an overview of the design for SPI NOR File System (SPINFS), a file system the author has developed and named according to a specific targeted type of flash memory: raw NOR flash with Serial Peripheral Interface (SPI) communication. As described in the Introduction, the goal was to implement a file system that could overcome wear behavior of flash memory to achieve longest possible lifetime and assure certain level of data integrity. First, there is a brief introduction to the structure of the file system and its internal components. Afterwards, the operations of SPINFS are introduced with their principle based on underlying data structure and concept. Finally, details for a special operation which is fundamental for the design of SPINFS called Garbage Collection are introduced

3.1 Design Overview

Conceptually, SPINFS is a log-structured file system and takes a completely diverse approach to popular file systems on hard disk drive like BSD FFS and FAT. Differentiating from traditional block-based file system that storage location is bound to a piece of data, log-structured file system makes use of entire storage for a circular log which is appended with every change made to the filesystem. Figure 17 and 18 display the different file allocation policies between block-based and log-structured file system designs.

Figure 17. Files location of a block-based file system in memory array

Figure 18. Files location of a log-structured file system in memory array

Unlike their block based counterpart, which has i-node blocks representing a file or directory scatter around the whole disk with address pointer pointing to the blocks that have the actual data, which also spread out; the circular structure consists of multiple nodes, each representing a file or directory and the data inside it, following each other. To simplify, the traditional file system has files laid out in a predetermined way with space between them as shown in Figure 17, while Figure 18 demonstrates the principle operation of a log-structured one, thus the files are written one by one in the ascending order to the end of the storage media.

3.2 Node Data Structure

In SPINFS, the basic building block in the log is a structure known as **struct spinfs_raw_inode** which is also the only type of node in this design of the file system. Because the file and directory are treated equally in SPINFS, only one node structure can be considered to be used for both, which minimizes the complexity of its architecture. Directory is theoretically a file whose data is a name list consisting of files and sub-directories are under this parent directory. Moreover, both share some similarities in metadata footprints, for example, name, i-node number, creation, modification time, size, owner

and access flags are some matching statistics of file and directory. Therefore, detailed members of SPINFS node structure are displayed in Listing 1.

```
struct spinfs_raw_inode {
    char name[32];
    uint32_t inode_num;
    int32_t uid;
    int32_t gid;
    int32_t mode;
    int32_t flags;
    time_t ctime;
    time_t mtime;
    uint32_t parent_inode;
    uint32_t version;
    uint32_t data_size;
    char data[0];
};
```

Listing 1. spinfs_raw_inode data structure

The SPINFS node is very similar to a conventional i-node in that it contains the metainformation about entities that live in the file system with only one exception; instead of storing addresses of that entity's data blocks which live somewhere else, the data part is included directly inside the node structure, right after the metadata segment.

Looking at the raw i-node structure, it is clear to see some basic file information that the SPINFS supports. First of all, the maximum file name length is 32 one-byte ASCII characters, which is said to be the minimum requirement for any interactive system [7, 18]. The next field, **inode_num**, is self-explain, it is the i-node number of the file or directory the current node is referring to. In SPINFS, the 32-bit unsigned integer value is used for storing the i-node number, allowing for 4 milliard files to be existed on the filesystem. However, considering that raw flashes are usually appeared in embedded systems, this maximum number of files is unlikely to be reached due to the simplicity and often single, straight forward application these kinds of systems provide. One thing to noted is that i-node numbers are never reused in SPINFS. In a case when files are deleted, new files

will always have the next highest available i-node number rather than having an obsoleted number from the deleted files.

The **uid** and **gid** fields record ownership information about a file. SPINFS is designed to be run on Linux host machines, thus it follows the convention that was specified in Portable Operating System Interface (POSIX) compliant that any files must have corresponding user id and group id which this file belongs to [21]. Combining with the **mode** field, the file system can provide file access permission check. By following the POSIX specification, the file permission model in SPINFS consists of user, group and "other" classes and three distinct operations that these classes can do to a file system entity: read, write and execute. The checking is done by comparing current logged in user with **uid** and **gid** fields to determine which class file access permission will be checked, then **mode** field will tell if the user can continue doing what he/she intends to do with the file, either read, write to it or execute an executable file. In addition to file access permission, SPINFS also stores information about whether this node is a regular file or a directory along in 32-bit **mode** value.

Moving on to the next **flags** field, it is a record of various bits of i-node states, and at the time of development there are only two other states beside normal state of an i-node in SPINFS. With two extra obsolete and deleted states, this field will only need to provide 2 bits in total of 32 to indicate the status of i-node.

SPINFS maintains some useful timely aspect of files known as creation time (**ctime**) and last modified time (**mtime**) of a file, and with these information users can easily query for files with a timestamp in mind. Unlike other Unix file systems, the author's approach does not attempt to support the last accessed time (**atime**) simply because this information is too expensive to maintain over the small amount of benefit that it provides. Every access to a file will need to update its node structure, and in a log-structured file system, this means the whole new node with all file content will be written to the flash even when only a small part of the file is read, which eventually wears the flash extremely fast.

The **parent_inode** field is effectively the i-node number of the parent directory of current file or directory. With a parent i-node stored directly inside a node structure, traversing backward the filesystem tree can be made efficiently to reconstructing a full path name of the file, while without it, the only way to know the full path name of an i-node is having that information in memory all the time while the file is opened.

The **version** field is a crucial piece of information for any nodes because it is where SPINFS maintains the historical ordering for them. As the spirit of SPINFS is a log-structured file system, every update to a file is actually writing a new node of the file to storage rather than modifying old data blocks, the file system needs a way to identify which is the most recent **version** of a file among many nodes referring to the same file. It is exactly what this field in raw i-node structure is responsible for, each new node is appended with a **version** higher than all previous nodes belonging to the same i-node number. Similar to **inode_num**, version is an 32-bit unsigned integer, hence there is a ceiling limit for the number of nodes can be updated for an i-node during the whole lifetime of a flash chip, however, this number of updates is reasonable so this limitation is deemed to be acceptable.

The last two members in the data structure specify the data segment of the node. The **data_size** field indicates how many bytes the size of the node is, which essentially is the size of a file in case this node is a regular file node. Finally, the last field is a flexible array member, **data[]**, where the actual data stream of a file is located. Since size of the files varies, the length of **data[]** array is determined by the **data_size** field, thus the starting address of the data is always presented in the node structure while the last address can be easily calculated for the file system to seek the next consecutive i-node.

After some considerations and updates to the design of SPINFS, the author decided to implement checksum for each node to improve the integrity of the file system. In C programming, flexible array member has to be the last data member of a struct [22] so a new checksum member value cannot be added to the end of the current data structure, thus leaving the author of SPINFS no choice other than having it in the **data[]** array right after the actual data of the node. Consequently, this increases the size of each node by the size of the checksum, 160 bits for SHA1 sum in this case [23], at the end; hence making **data_size** no longer reflects the size of a file if raw i-node is a regular file node, but the actual size plus 20 bytes.

3.3 Operations

In order to use any file system with a storage device, the very first operation is initializing the medium with a format that file system can understand. According to SPINFS, this step is essentially erasing the whole memory array and write the first node for root directory, which is also the most important entity in any file system. Writing the root directory

can be as simple as transferring all bytes consisting a raw i-node structure with "/" as name, i-node number of 1, parent i-node of 0 expressing no parent, and 0 byte in size of data part to address 0x0 of the flash.

The next usual operation with file system is populating the storage by creating files or sub-directories under root directory. A new file created in the filesystem means a new node structure referring to the file with corresponding i-node will be written to the flash, continuing after the last occupied address of the last node. In the case with a freshly formatted filesystem, new node is added after root directory node. Continuing creating new files and sub-directories will keep appending new nodes for those entities to the flash toward the ending address. At the point when there is not enough space to allocate a new node, an operation called garbage collection will be triggered trying to search for free space, and if it fails to do so, a "Not enough space left" error will be returned. Garbage collection is discussed more in details later in Chapter 3.4.

Due to the design of the log-structured file systems, every update to the content or metadata of a file results in writing a whole new node for that file with modified data to next available space in the raw flash. The new node will have a higher version value than the old ones, and these old nodes are said to be obsoleted, where the content they contain has been outdated by a later node. Space taken by obsoleted nodes is referred to as "dirty space" and will be reclaimed later by garbage collection operation.

File deletion is also considered an update to the file. SPINFS simply marks the file as deleted by setting deleted bit in the node **flags** member and write a new **version** of the file's node with zero data to the flash. After the file is marked as deleted, any operations with the associated i-node number should not be possible and should return error.

When creating or deleting any files and directories, the content of the parent directory is subjected to get modified, hence an update to the parent's node will be performed and another new node will be written. The name list which is the data parent directory carries is changed accordingly, having a new entry in case new file was created, or removing the old entry of a deleted file.

For the reason that there will be many nodes associated with a single i-node number in file system structure, when reading content of an i-node, SPINFS should ignore all obsoleted nodes and only return the latest information about that i-node. Therefore,

SPINFS always keeps an i-node table in memory and updates this table with corresponding changes. The i-node table has columns for i-node number, latest **version** value of the node and physical starting address of that raw i-node structure in storage device. At the mount time, SPINFS scans through the whole flash to populate this table. Upon looking at a new i-node number, a new entry will be appended to the table, while upon a newer version of an entry is found, that entry will be updated with the corresponding data.

3.4 Garbage Collection

At the point when the system is out of space due to continuous writes of new data, it needs to start reclaiming the dirty space which is the result of obsoleted and deleted nodes. To assist this reclaiming operation, the file system always keeps track of the physical offsets of the oldest node and the next erased address for a new node in the flash, these address offsets are called **head** and **tail** respectively. In a filesystem that this cleaning process has never been triggered, the **head** will stay at the very beginning of the flash, whereas the **tail** is reaching the last available address after any update operations.

During the practical work of this project, the test flash chip model had a feature called Security Registers, which are small separate memory regions aside from the main memory region consisting of only 256 Bytes [24, 46], and two of these registers were chosen solely for the purpose of storing the **head** and **tail** pointers' values. In SPINFS, **head** and **tail** are 32-bit unsigned integer, thus each Security Register can store 64 values for them and will be updated according to each write to main memory region. This means any new node written to the end of the flash will also write a new value for both **head** and **tail** in Security Registers 1 and 2. When all 64 slots has been used, SPINFS simply erased these two registers and start writing new **head** and **tail** values from the beginning. The search for correct, or most recent, **head** and **tail** values in Security Registers is straightforward since the registers can be read backward from the end and the first non-erased value, not 0xFFFFFFFF, is the wanted one.

Each time the garbage collection code is executed, the objective is to erase the first flash block pointed by **head**. The operation will begin by iterating **head** toward **tail** in natural direction over each node and examine whether the pointing node is obsolete or valid. In case the node is obsoleted by a later version written with the same i-node number or it is marked with a deleted flag with zero sized data part, it will be skipped, and the **head**

moves on to the next node. On the other hand, when the node is still valid, meaning that it has the latest version of a still-in-use file, the whole node will be copied to the **tail** of the log by writing an exact same node just with bigger version number, hence rendering the current node to obsolete and the **head** continues iterating.

Once the **head** has been progressed to the next erase block boundary of the flash, all nodes in the previous block should not be relevant to the filesystem anymore and it is safe to erase that block without losing any data. As a result, the amount of free space in the flash, indicating by number of bytes counting from the **tail** address to the **head** address, increases by one erase block size. In the instance of insufficient dirty space to have the size of an erase block when added together, the Garbage Collection process will return error.

4 Results

This chapter covers the hardware that were used in the development and the outcome of the programming work. The hardware including a host computer and a flash storage was provided by Metropolia University of Applied Sciences. Based on the available equipment, the implementation of file system was carried out for the specific host platform and storage model.

4.1 Development Hardware

This project was developed on a Raspberry Pi and a NOR flash chip soldered to its breakout circuit board which then is connected to GPIO pins corresponding for SPI communication channel 0 on the Pi. The specific model of the Pi is Raspberry Pi 3 Model B V1.2, and the flash is NOR Flash IC S25FL164K from Spansion. The Pi acts as the host running Raspbian which is a distribution of Linux Operating Systems targeted to Raspberry Pi's hardware. From the breakout board of the flash chip, jumper wires were used to connect directly to GPIO pins on the Pi as shown in Figure 19 and the connections are described in Table 1. Because SPINFS was written with the C programming language, there was a need to interface SPI communication to the code, and the author chose to use the SPI Application Programming Interface (API) from WiringPi library. For example, to send over a SPI packet, the programmer can call the **wiringPiSPIDataRW(SPI_CHANNEL, buf, sizeof(buf))** function. This library can easily be installed

from the apt-get package manager in Raspbian and should be linked with when compiling C source files with “-lwiringpi” gcc option [25].

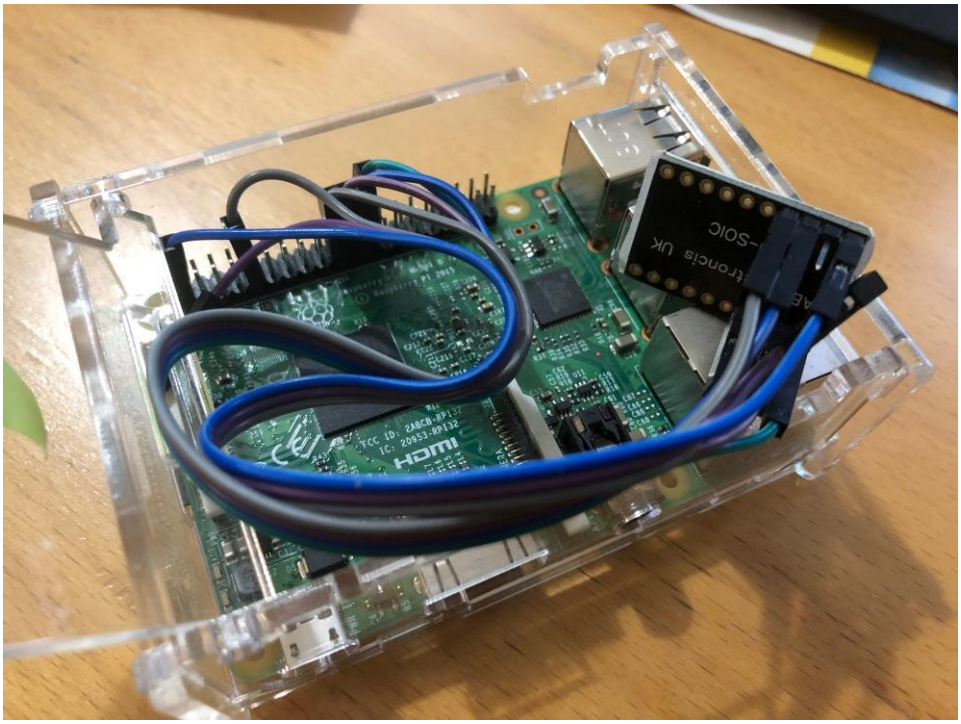


Figure 19. Raspberry Pi and S25FL164K breakout board SPI connection

The SPI protocol consists of 4 communication lines: CE, SCLK, MOSI, and MISO. The chip enable CE line is active low, thus enabling the chip whenever the voltage is drop to 0V. From Table 1, chip enable is connected to Raspberry Pi CE0 pin 24, which is SPI line channel 0. The SCLK is the clock signal and is generated by the Pi. The last two wires in SPI communication are MOSI and MISO. They are data lines and responsible for data transmission from master to slave and vice versa.

Table 1. Details in connection between Raspberry Pi host and SPI flash chip

	Raspberry Pi GPIO pin	**S25FL164K pinout**
MOSI	19	5
MISO	21	2
SCLK	23	6
CE0	24	1
3V3	1	8
GND	9	4

The S25FL164K flash model is a 64 Megabits (8 Megabytes) variant in its flash chip family, thus having address counting from 0x000000 to 0x7FFFFF. All flashes in this family share some similar characteristics, such as 256 bytes programming page size and smallest erased size is 4 Kilobytes. It is a NOR flash meaning any byte in the flash is allowed to get accessed directly for reading and writing; comparing to NAND flash, this has to be performed on the whole page [13, 35, 53]. As being said, a read command can read one byte at any specific address or data in following addresses will also be returned as long as clock line in SPI protocol is still running. However, programming to the flash accepts only 256 bytes at most [24, 74]; and if this maximum value is desired to be written, the address used in the program command should be 0xXXXX00, otherwise the address will be wrapped around at page size and old data will be overwritten. A complete list of C macro definitions used for S25FL164K flash model is provided in Appendix 1. [24]

4.2 Functionalities

SPINFS defines structures on an array of memory enabling file and directory hierarchy to reside there. At the end of the project, the file system is still in its simple form and missing many features for a fully functional file system that could be used natively in an OS. However, there are command line programs were written to support basic file operations on a SPINFS structured flash device. Because the targeted storage communicates on SPI protocol, the tools utilized WiringPi SPI APIs to send commands over SPI line to control the storage, these SPI commands can get the flash information, read,

erase, program or set some settings for the transmit protocol. Those programs and their synopses, functions are listed below:

- spinfs-mkfs: format the flash and write SPINFS structure. As introduced in Chapter 3.3, this program will perform Initialization operation, which erases the whole flash and write the first node for root directory at address 0x0.

- spinfs-ls [PATH]: list information about files and directories. PATH is similar to UNIX path to the file or directory, which has "/" separator between parent directory and its child. In case of directory, this program will list its name list or list of file and sub-directories.

- spinfs-mkdir [PATH]: create a directory at a certain path.

- spinfs-cp [SRC] [DEST]: copy file between host and memory device. Since SPINFS have yet to support modifying file natively, copying a file from host is the only to write to the flash. Using the same tool, a file on the flash can be read by copying it back to host machine.

- spinfs-rm [FILE]: remove a file in the filesystem. Executing this program is not necessarily zero out all bits associated with a file, it only marks the file node as deleted, and will only be erased at the next Garbage Collection run.

- spinfs-free: run Garbage Collection to reclaim dirty space. This operation has been explained in detail in Chapter 3.4

Even though SPINFS is not yet able to perform as a normal file system from OSes, e.g. ext4, NTFS, FAT, etc., making use of six SPINFS tools above can achieve similar effect. The memory flash can be formatted with SPINFS structure, files and directories can be created and stored on devices for later retrieval and fetching metainformation, unused files can be removed, and dirty space can be reclaimed for later use. As a result, SPINFS has most basic functionalities any file system has to support and provide its own set of APIs for users to control raw NOR flash memories, store, organize and retrieve multiple files and directories.

5 Discussion

This chapter compares the results from Chapter 4 to the objectives set in the Introduction. There is assessment to verify that whether the final outcome meets the project's requirements. Moreover, any caveats of the design are also discussed with possible future enhancements which could potentially make the implementation of the file system be ready for public use and industrial standards.

5.1 Design Goals

The design of SPINFS fulfilled the goals of the project. Being a log-structured file system means that the whole memory is written in circular motion, making all block of the flash to have the same P/E cycles. That being said, there are no blocks getting erased more frequently than the others. The logging file system naturally solves the requirement for unmanaged flash memory, it has a perfect wear leveling scheme that distributes wear across the storage equally. With the help of the Garbage Collection process, when the memory is close to full, the blocks at the top of the array will be erased to produce more free space. By continuing to write at **tail** and erase blocks at **head** the log-structured of the file system is preserved, data will always be written in circular motion, keeping wear leveling design present at any time.

Focusing on SHA1 checksum value of each node, this feature has improved data integrity of the flash preventing any chance of power failure during write. Without any fail-safety methods, the writing process to storage device might be interrupted because of disconnection in communication or power outage and a such situation is extremely dangerous for the system. Incomplete writes are the main corruption of file systems, data on the flash which does not follow the format of the file system would not be referenced by the file system and are considered to be lost. The file system might lose some files or in a worse case, lose a whole directory. Having checksum at the end of the node ensures that data in the node is correct and sane, which guarantees the SPINFS that the current node does not have any corruptions. In SPINFS, **tail** value only gets updated after a new node is fully written to the main memory array, hence in a case of power reset during write, the new node can be verified to be valid based on its checksum. If the checksum validation does not pass and indeed the node was incompletely written, using **tail** the file system can roll back to the state before writing the corrupted node, making the whole system become valid.

5.2 Limitations

Utilizing a log-structure file system is an appropriate approach to tackle the wear issue in the flash memory, however, the design also has flaws in other aspects. The operations of SPINFS as discussed in Chapter 3.3 pointed out that any update to the filesystem means a new node will be written to the flash. Based on the size of the updated file, even a small modification ends up writing the whole metadata and data segment of the file, which could take up a lot of space. Consequently, more writes to the flash also means more wears, the storage is easily filled up and the number of erase increases. It is a trade-off that file system designers are often willing to take since the benefit of natural wear leveling in this type of file system is more important than the number of extra writes.

Moreover, SPINFS has not been extensively tested with all possible use cases and in critical conditions to expose to any possible failures due to development time constraints. Though the design of the file system and its associated tools were carefully planned and executed, extra testing phases for finding bugs are necessary for any software products. Practically, a normal use case might not able to expose critical software bugs; only with the help of thoughtful test tools and test cases, the products could go to some conditional states more prone to errors. Based on the test conditions, developers can further debug and improve their software to prevent as many errors as possible.

5.3 Future Development

Chapter 4.2 mentioned command line programs which send requests to the flash and translate SPINFS data structure to files and their stream of bytes; so the current design lacks support for mounting the storage to a native OS path. Supporting this service is one major feature that can be considered for future development of SPINFS. There are two approaches to achieve this idea, either having SPINFS running as a kernel file system program or having it running as userspace program. The former option can bring great outcome both in terms of performance and security. However, implementing a kernel program requires extensive knowledge in kernel developments which is not a strong point of the author. The latter method is more feasible and can be more accessible for other users if SPINFS is open for public use. In order to develop a file system in userspace, Filesystem in Userspace (FUSE) library API can be used to support a fully functional file system. FUSE library provides interface for all file system related similar to what a kernel file system would need to implement, then programmers would have to

define their own implementation of the functions and link them by using callback functions. Examples for some basic file system operations in libfuse are getattr, open, read, write, opendir, readdir, chmod, chown, rename, etc. [26]

Another aspect that could be enhanced in SPINFS is to have specific test tools to evaluate the file system. With test tools, the file system can test its functionality in some rare and critical conditions, potentially exposing bugs that never appear in normal operation sequences. For example, testing to write a large number of files, directories in a deep hierarchy tree is a common test case for any file system; this can really stress the file system to see if it can correctly keep up with the complexity. Because of limited amount of time for development, SPINFS has not been tested much in such cases to guarantee its defined behavior in such situations.

6 Conclusion

With a large variety of storage technology, choosing a suitable file system for a specific storage device should not be overlooked. In embedded applications, an unmanaged raw flash IC is an appropriate choice to store data and information due to their small footprint and power consumption. Therefore, having a file system that works on this kind of flash is needed if there are requirements to store multiple files with complex directory hierarchy in the system. Flash technology has been an improvement for storage devices over hard disk technology due to the fast read and write speed in memory cell. However, flash storage has a limited erase cycles before the cells become unreliable for retrieving and storing data.

The goal of this thesis was to design and create a file system that can tackle the wear problem in flash memory with some methods to ensure data integrity present. By utilizing the log-structure design of file system, wear leveling is guaranteed by nature of the design. The file system entity is called a node, which represents both file and directory, and nodes are written to the memory in circular motion, making all blocks have approximately the same number of P/E cycles. Furthermore, each node also stores its checksum value, which can notify the file system if there is any corruption in its data. Thus, two set goals of the project were successfully achieved based on the design of the file system. The implementation part also provides a set of command line tools to read and write the

memory flash with SPINFS data structure. In future development, the design can be enhanced by linking with the FUSE library to provide a mount point to the OS and would work like any normal, fully functional computer file systems.

From this project, the author has gain extensive knowledge in file system design and flash memory technology. Regarding file system design, the author has learned what is the definition of file system, how a traditional block-based file system works, and which kind of design would be suitable for raw flash memory. Additionally, gaining more understand about flash memory, its operations, limitations, trend in technologies is greatly beneficial for the author's work experience as an employee of a file system and storage company.

References

1 Turing A. On Computable Numbers, with an Application to the Entscheidungsproblem. Cambridge: King's College; 1936.

2 Arrow Electronics, Inc. Memory Trends in the Semiconductor Industry [online]. Arrow Electronics, Inc; 6 April 2018. URL: https://www.arrow.com/en/research-and-events/articles/global-growth-trends-and-forecasts-in-the-semiconductor-industry. Accessed 7 October 2019

3 Alexey Mylnikov. Memory Market 2018 and beyond [online]. Intel Corporation: Micron Technology, Inc.; 21 May 2018. URL: https://www.intel.ru/content/dam/www/public/emea/ru/ru/documents/resellers/micron-about-memory-market-and-2018-trends.pdf. Accessed 7 October 2019

4 Tuxera Inc. About us [online]. Tuxera Inc; 2019. URL: https://www.tuxera.com/about-us/. Accessed 20 October 2019

5 Nishi Y. Advances in Non-volatile Memory and Storage Technology. Sawston: Woodhead Publishing; 2014.

6 Quinlan D, Russell P, Yeoh C. Filesystem Hierarchy Standard [online]. The Linux Foundation; 19 March 2015. URL: https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html. Accessed 16 October 2019

7 Giampaolo D. Practical File System Design with the Be File System. San Francisco: Morgan Kaufmann Publishers; 1999.

8 ISO (International Organization for Standardization). Information processing — Volume and file structure of CD-ROM for information interchange. ISO 9660:1988(en); 1998.

9 Microsoft. NTFS Overview [online]. Microsoft Docs; 17 June 2019. URL: https://docs.microsoft.com/en-us/windows-server/storage/file-server/ntfs-overview. Accessed 30 October 2019

10 Hutchinson L. New file system spotted in macOS Sierra [online]. Ars Technica; 13 June 2016. URL: https://arstechnica.com/gadgets/2016/06/new-apfs-file-system-spotted-in-new-version-of-macos/. Accessed 30 October 2019

11 Calleja D. Ext4 [online]. Kernelnewbies; 30 December 2017. URL: https://kernelnewbies.org/Ext4. Accessed 30 October 2019

12 Haster C. Littlefs Design [online]. Github: ARMmbed; 2 September 2019. URL: https://github.com/ARMmbed/littlefs/blob/master/DESIGN.md. Accessed 25 September 2019

13 Richter D. Flash Memories Economic Principles of Performance, Cost and Reliability Optimization. Dordrecht: Springer; 2014.

14 Matsuoka F. US4531203 (A) - Semiconductor memory device and method for manufacturing the same. United States Patent; 1985.

15 Tkachev Y, Kotov A. A Detailed Analysis of Hot-Electron Programming Efficiency in 40-nm Split-Gate Flash Memory Cells. San Jose: Silicon Storage Technology Inc.; 2017.

16 Jelinek L. Global Semiconductor Market Trends. London: IHS Markit; 2018.

17 Galup-Montoro C, Schneider M. MOSFET Modeling for Circuit Analysis and Design. Singapore: World Scientific Publishing; 2007.

18 Chen B. Hard Disk Drive Servo Systems. London: Springer; 2006.

19 Scan. SSDs – SLC vs MLC [online]. Scan Computers International; 28 September 2011. URL: https://www.scan.co.uk/tekspek/hard-drives/ssds-slc-vs-mlc. Accessed 19 September 2019

20 Denholm T. Tesla memory problem - is the type of wear leveling the real problem? Datalight Inc.; 18 October 2019. URL: https://www.datalight.com/blog/2019/10/18/tesla-memory-problem-wear-leveling/. Accessed 11 October 2019

21 The Open Group. POSIX.1-2017 [online]. The Open Group; 2017. URL: https://pubs.opengroup.org/onlinepubs/9699919799/. Accessed 5 October 2019

22 ISO, IEC. ISO/IEC 9899:1999 (E) Programming Languages – C. ISO/ICE; 1999.

23 Drepper U, Miller S, Madore D. Sha1sum manpage [online]. Linux Programming Interface; March 2019. URL: http://man7.org/linux/man-pages/man1/sha1sum.1.html. Accessed 18 October 2019

24 Spansion. S25FL1-K Datasheet Revision 3. Spansion; 2014.

25 Gordon. WiringPi SPI Library [online]. WiringPi. URL: http://wiringpi.com/reference/spi-library/. Accessed 22 August 2019

26 Szeredi M, Rath N. The reference implementation of the Linux FUSE (Filesystem in Userspace) interface [online]. Github. URL: https://github.com/libfuse/libfuse. Accessed 28 October 2019

Macro definitions for S25FL164K Flash Memory

```
/*
 * NOR Flash info
 * All 4-kB sectors have the pattern XXX000h-XXXFFFh
 * All 64-kB blocks have the pattern XX0000h-XXFFFFh
 */
#define MAIN_FLASH_SIZE         8388608 //B = 8 MiB
#define ADDRESS_BITS            24
#define ADDRESS_BYTES           (ADDRESS_BITS / 8)
#define SECTOR_SIZE             (4 * 1024)       //4 KiB
#define BLOCK_SIZE              (64 * 1024)      //64 KiB
#define SECTOR_COUNT            2048
#define PAGE_SIZE               256     //256 B
#define STARTING_ADDRESS        0x000000
#define ENDING_ADDRESS          0x7FFFFF
/*
 * Security Registers info
 * has size of 256-byte each
 */
```

```
#define SEC_REG_SIZE            256     //256 B

#define SEC_REG_1_START_ADDR    0x001000

#define SEC_REG_1_END_ADDR      0x0010FF

#define SEC_REG_2_START_ADDR    0x002000

#define SEC_REG_2_END_ADDR      0x0020FF

#define SEC_REG_3_START_ADDR    0x003000

#define SEC_REG_3_END_ADDR      0x0030FF

/*

 * Status Registers info

 */

#define BUSY_BIT_MASK           1

/*

 * NOR Flash command macros

 */

#define READ_STATUS_REGISTER_1  0x05

#define READ_STATUS_REGISTER_2  0x35

#define READ_STATUS_REGISTER_3  0x33

#define WRITE_ENABLE            0x06

#define WRITE_DISABLE           0x04
```

```
#define PAGE_PROGRAM            0x02    //1 - 256 B

#define SECTOR_ERASE            0x20    //4 KiB

#define BLOCK_ERASE             0xD8    //64 KiB

#define CHIP_ERASE              0xC7    //8 MiB

#define READ_DATA               0x03

#define READ_JEDEC_ID           0x9F

#define ERASE_SEC_REG           0x44

#define PROGRAM_SEC_REG         0x42

#define READ_SEC_REG            0x48
```