NAND Software Working Group

Invitation for Collaboration on Software Interface Standardization

Version 1.0 – 04 13 June 2006

Questions pertaining to this document, or the terms or conditions of its provision, should be addressed to:

MIPI Alliance, Inc.
c/o IEEE-ISTO
445 Hoes Lane
Piscataway, NJ 08854
Attn: Board Secretary

Introduction

Contributors List

COMPANY	NAME
M-Systems Flash Disk Pioneers	Avraham Shimor
Micron Technology, Inc.	Wanmo Wong
Samsung Electronics	Yejin Moon
STMicroelectronics N.V.	Antonio Furno
STMicroelectronics N.V.	Thierry Vuillaume
Texas Instruments Incorporated	Leonardo Estevez
Toshiba Corporation	Shuji Takano
Toshiba Corporation	Patrick Elzer

Contents

Invitation for Collaboration on Software Interface Standardization

1 Overview

1.1 Scope

What is MIPI ?

The Mobile Industry Processor Interface (MIPI) Alliance (https://www.mipi.org/) is an open membership organization that includes over 90 leading companies in the mobile industry that share the objective of defining and promoting open specifications for interfaces to mobile application processors. Through these specifications, the MIPI Alliance intends to speed deployment of new services to mobile users by establishing specifications for standard hardware and software interfaces to mobile application processors and encouraging the adoption of those standards throughout the industry value chain.

Founded by ARM, Nokia, STMicroelectronics and Texas Instruments, the MIPI Alliance is intended to complement existing standards bodies such as the Open Mobile Alliance and 3GPP, with a focus on microprocessors, peripherals and software interfaces.

What is the MIPI NAND SW Working Group ?

The NAND SW Working Group has been established by the MIPI Alliance in the end of 2005 to investigate on SW interfaces standardization for a better and faster integration of NAND products into SW platforms. The group is composed of leading NAND flash industry players and in particular mains NAND manufacturers.

The Working Group will focus on:

- Identifying a state-of-the-art SW architecture comprising the embedded *NAND-like Flash* storage management and anticipating coming NAND flash usage.
- Standardizing the SW interfaces (functions, parameters, return codes...) between the identified architecture components that would be under responsibility of NAND flash manufacturer, and publishing corresponding specification(s).
- Developing additional documentation to help to use and to advertise this interface (white paper, examples...).

As a necessary factor for success, the architecture and interfaces defined by the NAND SW Working Group will aim to stay compatible with existing and future Operating Systems architecture.

What are the problems ?

Traditionally, as the de-facto standard industry practice, NAND Flash devices in embedded and mobile platforms (much like in PC platforms) are used to emulate Hard Disks or block devices. There are two types of problems associated with this contemporary approach:

- The current Hard Disk emulation paradigm is problematic in the sense that it creates performance and reliability problems originating from mismatch between this block device emulation and the unique characteristics of the NAND Flash technology;

- But maybe even more significantly, there is a growing awareness in the Flash industry that certain functional topics like security, DRM, compression, etc., are very difficult to implement over the block-device, sector oriented operation of the Hard Disk emulation.

The NAND SW WG is seeking solutions to improve the usage of NAND flash on file systems, to fasten integration of NAND into platforms and to integrate emerging mass storage technologies.

Scope and Document overview

This paper provides a summary/review of the topics related the interface between NAND Flash storage devices and the embedded/mobile computing platforms and the corresponding Operating Systems:

- Elaboration of the problems in the current block-device emulation (FAT/FTL) approach.
- Overview of the new functional topics of security, DRM, compression and data portability.
- Description of associated Performance Issues.
- Review of alternative (non block-device or non FAT oriented) storage management methods, focusing on their interfaces with the Operating Systems.
- Outline of a "whish list" requirements specification for a new "ideal" solution.

.

1.2 Purpose

The purpose and the intended use of this document is to serve as a **background material**, to facilitate the interaction between leading Operating System vendors and mobile/embedded computing platform vendors with the MIPI NAND Software Workgroup. The objective of the interaction is to **jointly investigate** the **need and possibility** to extend functionality of the NAND Flash management and interface software beyond the current de-facto industry standard practice of block-device disk emulation**.** The result of this investigation will be used to align future NAND SW WG standardization efforts with OS and platform vendor's roadmap.

Clarification. This document is highlighting some problems related to the current software approach for NAND devices; however **this document doesn't impose any solutions,** it's a starting point for **collaboration with leading OS vendors** and mobile/embedded computing platform vendors in definition of NAND Flash software interface with the objective of maximizing generality and industry acceptance.

2 Terminology

2.1 Definitions

Module: Coherent group of functions, classes or other software definition handling a specific subject of the software framework. Each Module is identified by a single name and acronym.

2.2 Abbreviations

e.g. For example

2.3 Acronyms

DRM	Digital Right Management
INCITS	InterNational Committee for Information Technology Standards
JFFS	Journaling Flash File System
MIPI	Mobile Industry Processor Interface
OSD	Object-based Storage Device
SCSI	Small Computer System Interface
FTL	Flash Translation Layer
FAT	File Allocation Table
LBA	Logical Block Address
PBA	Physical Block Address
SW	Software
HW	Hardware
API	Application Programming Interface
LFN	Long File Name
MMC	Multi Media Card
SD	Secure Digital [Card]

3 References

[JFFS] **The Journaling Flash File System, David Woodhouse,** , Red Hat, Inc., dwmw2@cambridge.redhat.com

[JFFS2] **JFFS : The Journalling Flash File System**, **David Woodhouse**, Red Hat, Inc., dwmw2@cambridge.redhat.com

[JFFS3] **JFFS3 design issues, Artem B. Bityutskiy,** dedekind@infradead.org

[OSD2] **Object-Based Storage Devices - 2 (OSD-2)** INCITS T10 Technical Committee,: 2004/10/04, Rev: 00, Status: Development, Project: 1729-D, File: osd2r00.pdf (1313078 bytes)

[**OSD DEMO] SEAGATE, EMULEX AND IBM TEAM UP TO DEMONSTRATE INDUSTRY'S FIRST STANDARDS-COMPLIANT OBJECT-BASED STORAGE DEVICES**

[OSD PROJ1] Seagate Object Based Storage Project

[OSD PROJ2] IBM Haifa Labs – Object Store Project

[PTP] ISO 15740:2005: Electronic still picture imaging. Picture transfer protocol (PTP) for digital still photography devices.

4 Problems of the current Hard Disk or block device emulation

Traditionally, as the de-facto standard industry practice, NAND Flash devices in embedded and mobile platforms (much like in PC platforms) are used to emulate Hard Disks or block devices. The obvious advantage in this practice is that it is leveraging on existing Operating System software components: as far as either the Operating System or even the application software is concerned the actual technology implementation of the NAND Flash devices is totally transparent: for all practical purposes it behaves like yet another Hard Disk Drive.In the context of this paper, by *Hard Disk* or *block device emulation* we refer to the abstraction whereby a storage device, independent of the underlying physical storage technology employed (such as magnetic, optical, or flash) is presenting or exporting a simple, yet powerful, atomic interface of allowing the reading and writing a fixed sized information unit or sector, usually addressable by a linear index (so the storage device is logically represented as a linear array of sectors). On top of this simple scheme, the Operating Systems and its storage and file management components implement additional software based layers of abstractions like for example the FAT type file system which was introduced in the 1980s. (FAT is extant in small systems. Microsoft chose FAT as the file system in Windows CE, and FAT is used in Symbian. Furthermore, FAT is the standard file system format for flash cards like MMC and SD.).

Deleted: ¶
¶

In all implementations of file systems over block devices some of the sectors are used to store the actual information content associated with files, whereas additional sectors are used to store supporting, addressing and organization information such as allocation tables, directory entries, etc. The important implication of this method for the context of this paper is that most application level storage/disk related operations (transactions) require several sector (or device block) level operations; from the device "point of view" these are unrelated and independent "atomic" operations.

NAND Flash storage devices exhibit several functional characteristics which are originating from the essence of this technology: information can be read and written in the form of *pages* (currently of typically 512 or 2048 bytes, with a tendency to increase as the capacity of the devices increases), once a page is written, it cannot be re-written or updated, unless the *erase block* (consisting a large number of pages) containing the page in question is erased first. Obviously, erasing a block implies moving/copying of all or at least some of the pages to another ("fresh", yet unwritten) location in order to avoid information loss. Yet another important characteristics of the flash technology is that the number of erase cycles for a given erase block is limited, thus the number of erase operations needs to be minimized in order to prolong the useful life of the device.

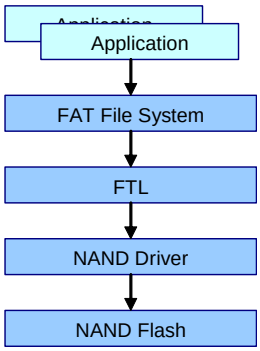

Figure 1: FAT and FTL Architecture

Due to the above technology specific limitations the block device emulation cannot be directly facilitated by a flash device, thus the common technique is use an intermediate software layer between the Operating System component (usually: the file system) "expecting" to interact with a sector based block device and the physical flash memory (see figure 1). A common name for this block device or disk emulation layer is *Flash Translation Layer* (FTL).

The current practice of the Hard Disk emulation paradigm is problematic in the sense that it creates performance and reliability problems originating from mismatch between this block device emulation and the unique characteristics of the NAND Flash technology; The following sections outline some of these problems.

4.1 Power Loss Robustness

By Power Loss Robustness in the context of this paper we refer to the capability of a NAND Flash based device with its corresponding FTL to ensure the integrity of the information stored (primarily the file content "payload" information, but also the flash management related metadata) even in the presence of power loss failure events.

The root cause of this problem is the fact that in file system implementation over any **Module** block device most application level transactions (e.g. create a file, write a new record or data extent to a file, etc) require more than one device/sector level atomic operations (e.g. in order to add a new data extent at least two sector write operation are required: one to update the Allocation table and one to write the data content). These operations cannot be "recognized" by the device as belonging to the same transaction. Consequently, a power failure event encountered when only a part of the atomic operations pertaining to such a transaction were actually executed by the device, can lead to an inconsistency, which may result in an irrecoverable data lost.

While, as mentioned, this problem may exist in **any** block device, it can be more significant and problematic in a NAND Flash device. The reason for this situation is that the management of the NAND Flash device (by FTL) requires additional mapping mechanisms (in addition to the Allocation Tables and directory structures used by the File System) which in turn imply storage and manipulation of additional metadata on the flash media. Furthermore, some of the flash management operations (for example the copying and relocation of sectors mentioned in the previous section, and "garbage collection" operations resulting from the need to prepare for reuse "used" but no longer valid pages and blocks) are significantly time consuming. This might result in increasing the time separation between the actual execution of the composite atomic operations pertaining to an application level transaction, and thus increasing the sensitivity of the process to an eventual power failure.

4.2 Longevity

NAND flash devices usually wear out as the result of repeated programming and erasing; with the contemporary device technologies the typical device lifetime is limited to about 100,000 program/erase cycles per erase block (even lower for MLC device).

To maximize the useful life of the flash, care must be taken by the Flash management algorithms to spread the wear evenly over the device, and in fact the different algorithms employed by different vendors invest a significant effort (and contain significant amount of related Intellectual Property) to address this issue.

While this longevity problem is an inherent result of the nature of Flash device technology, its importance and possible severity are intensified by the block device emulation practice: the need to repeatedly update the File Allocation Tables create and represent a potential "hot spot" for the extended wear of the file system related sectors of the flash media.

4.3 Flash Media Reliability

An additional peculiarity of the NAND Flash device technology is a problem usually referred to as "bit flipping", posing a significant reliability problem if not properly handled by the Flash Management algorithms.

While in a magnetic disk for example imperfections of the recording surface/media are usually of permanent nature, thus "bad blocks" can be detected, marked and avoided a-priori while formatting a magnetic storage device, in NAND Flash devices there is a phenomenon called bit flipping (manifesting itself by an inconsistency between the content intended to be written and the content actually read later) which can evolve, rather than being permanent. The root cause of the problem may be not only marginal imperfections on the chip but may also be wear resultant. This necessitates further, dynamic measures to be exercised by the Flash management algorithms, implying both additional mapping for dynamic bad block detection and avoidance.

Once again, although the bit flipping problems are not a direct result of the block device emulation method, nevertheless, in the situation where application level transactions are split into several independent atomic device/sector level operations, this creates yet another problem spot as a result of the management operations required to handle this problem

5 Possible new application domain requirements

The practical use of storage devices in complex applications involves the necessity to handle several storage related functional requirements which are of higher level than the basic input/output or read/write/delete transactions. Closer examination of several such functional requirements in the following subsections reveals that the feasibility and/or efficiency of implementing these are dependent and interrelated to the nature and functionality of the interface between the storage device and the host side Operating System.

5.1 Digital Rights Management (DRM)

There is an increasing need in the industry for implementing controlled access to information content stored on memory/storage devices, i.e. enforcing a DRM scheme with respect to the protected content. The ideal solution for DRM would be the enforcement/implementation of the DRM access rules internally at the storage device itself.

The nature of the protected data entity is a "content" object, e.g. a song, a picture, a document – these are usually translatable to data objects, and most frequently to files. Clearly, the "sectors" as used the interface with a block-device (or block-device emulation) based storage device, are well below the natural granularity of the associated "rights" in any DRM scheme, thus the device implementation of **trusted and reliable** DRM in these type of devices is practically impossible because it's always possible to by-pass the highest software module (e.g. FAT File System) and access the device trough the FTL layer which has no "knowledge" of DRM rights (figure 2).

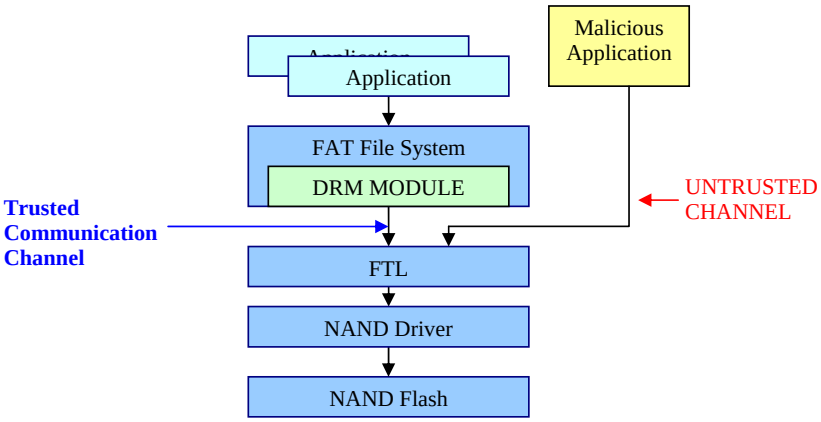

Figure 2 Possible attack on FTL based DRM

However other, higher level abstraction interface schemes, where the content exchanged between a host and a device represents files or data objects, implementing/enforcing DRM at the device level is feasible.

5.2 Compression

In order to increase the effective capacity of a storage device, there is a requirement in some mass storage applications to implement transparent compression at the storage device level. This means that the volume of information actually stored/recorded onto the physical storage media is reduced by a compression algorithm implemented at the device on the information to be written, and a complementary decompression is performed on the read from the storage media before transferring it to the host.

A possible use case or application example: a handheld computer for use in a “mobile office” type application environment, where the flash disk in the device has to store a heterogeneous mix of file types and sizes, such as word processing documents, presentations, spreadsheets, messages, notes, etc. Obviously, the effective capacity of the flash drive is a factor of critical importance for the usefulness of such a device, so it is expected to be deemed advantageous by the end users if the effective capacity could be increased without paying for extra flash, or requiring increased size and/or power consumption, by employing data compression. File type dependent data compression (for example those well established compressed file types for pictures/video/music) might not be applicable or sufficient in this case, due to the heterogeneous nature of the file types as mentioned above, and the associated need to be compliant (i.e. readable, transparent, usable) with a diverse set of applications. Hence, if such a compression could be provided transparently and seamlessly by the storage device itself, this could be the optimal solution. (By transparency we mean for example that the application program is totally unaware that the 2.3 megabyte document it had just written to the flash disk is actually occupying only 0.68 megabytes of space).

In a block-device emulation environment, where the atomic transactions are based upon transparent sectors directly addressable by the host, there is practically very little which can be accomplished by compression decompression, since sectors cannot/must-not be concatenated, thus the efficiency of the is very limited. Another limiting factor is the mapping mechanism from file system sectors to the flash pages.

Another problem related to the compression is that usually multimedia data (e.g. MP3 files) are already compressed and applying the compression algorithm don't cause a real benefit. With a block emulation schema it's impossible to avoid this situation because (again) the FTL cannot know what kind of data it's storing into the device. With a different software architecture that is not based on the Block Emulation approach it could be possible to implement algorithm that (knowing the "nature" of the data) could compress only that data that should be compressed.

5.3 Encryption/security

This subject is somewhat similar to the DRM issue described above: the implementation of some DRM schemes involves encryption; however the topic of encryption is useful even out of the context DRM. It is sometimes necessary to ensure that the content on a storage device is protected from unauthorized access and use. As an extreme use case, for example, sometimes it is necessary to ensure that actual content stored on the flash is encrypted, so it is incomprehensible without the possession of an encryption key which is not stored on the media itself. The encryption and the corresponding keys are clearly content related: using the traditional directory/file terminology, different files must be able to be encrypted different, user specific keys, some files and/or directory entries needs to remain visible (non-protected, non-encrypted) while others needs to be hidden (protected, encrypted).

In a block-device emulation environment, where the atomic transactions across the interface between the storage device and the host are based upon sectors, and where the interpretation of the sectors content as either net, "payload" data or file system structural/support info is only visible to the File System software on the host, there is practically very little possibility to implement the necessary encryption/protection provisions on the device itself, therefore the encryption/decryption must be inevitably done by the application.

If however the block-device scheme would be replaced by some alternative scheme, where the interface transactions are based on data objects or complete files (see below), then these schemes might **enable** implementation of the encryption/decryption, if required, at the (future NAND-like) device level, transparent to the application.

(The assessment whether device level versus application level implementation of encryption/decryption is required/desirable/beneficial is left open for the discussion).

5.4 Data portability

One of the most important use cases for NAND Flash devices (and removable storage media and storage devices in general) is their use as vehicles for data/information/content transfer among different platforms (PC to PC, camera to PC, mobile phone to PDA, etc.). Obviously, in order to ensure a seamless flow of content among different platforms, sometimes involving different Operating Systems, both types of the software components involved need to be compatible: the File System on the host side and the data representation on the media side. Within the contemporary block-device emulation scheme the device stores and handles transparent sectors, some of which are data sectors (i.e. carrying the "net" content) while some others are used solely by the host File System (like allocation tables, directory entries etc.). In order to ensure that a computing platform, other than the one used in creating/writing the content, is capable of reading and "comprehending" the stored content, the reading platform must contain a File System which is binary compatible (with respect to the sectors stored on the device/media) with the writing platform. This requirement is a severe and limiting one: while most contemporary platforms support the FAT file system for cross platform data portability, this is obviously sub-optimal, since in this way the advantages of the more advanced native file systems (e.g. NTFS or advanced UNIX/LINUX file systems) are lost.

On the other hand, if an advanced/alternative interface technology is employed, based on data object transactions rather than transparent sectors (see Section 7), then the problem of data portability is automatically solved (or at least significantly diminishes).

While the primary focus of the MIPI NAND Software Workgroup is on embedded, i.e. non-removable, applications of NAND Flash storage devices, nevertheless, this subject of data portability is relevant due to the fact that the mobile/embedded hosts including the NAND Flash storage (as an integral non-removable component) are frequently used as detachable/removable storage devices in other computing platforms: for example a mobile phone (with embedded NAND Flash storage device) may act as a removable storage device for a PC host when connected to a docking/sync station.

6 Performance issues

There are emerging streaming multimedia use case scenarios which may highlight performance issues with NAND devices such as simultaneously recording a DVB-H stream at one point while rendering from the same stream at another, advanced video trick plays, and advanced video telephony scenarios. For these use cases, some concurrencies and sequences should be considered.

Because the relative performance overhead of the file system for NAND Flash is so small compared to other activities within the system, the performance of the file system is most critical when considering time extended use cases which require very little power for short periods of time. The most common use case today falling into this category is the rendering of MP3/AAC music or audio books with competitive playback times in excess of a hundred hours on a single charge. For use cases like this one, file system performance impact to playback time is largely a function of the data rate.

Larger data rates with extended write scenarios might highlight file system for NAND Flash performance even more. An emerging use case for consideration is that of recording a series of digital TV broadcasts. Here again, the amount of time this scenario can be executed is a function of concurrencies implemented between receive and write subsystems. If writes can be successfully executed concurrently with received bursts, the operating frequency required to enable this concurrency may be a minor discriminating factor. If the write subsystem is executed sequentially after reception, the amount of power consumed by the receiver and the relative on time of the application processor in order to complete the write may become discriminating factors.

The longevity of the recorded TV series scenario may also be compromised if a limited amount of memory is also used as a circular buffer for data with a short life expectancy (such as concurrent commercial offers).

The block emulation software solution (File System plus FTL) limits the possibility to achieve the best performance possible and to guarantee a constant or at least a minima value of write and read throughput. In fact the FS plus FTL solution is not a “NAND friendly” solution and it’s not possible to implement dedicated algorithms to manage the write or read operations. As usual FTL is not aware of the object (e.g. file, directory) that the application is managing and it can only manage sectors (usually 512 byte size). In this situation is almost impossible to implement (in a safe a robust way) algorithms that can use all the potentiality of NAND throughput: read/write caching, multiple operations, large page architecture (e.g. 2KB, 4KB).

7 Review of non Block Device oriented alternatives

There are numerous studies conducted by various research, educational and industrial organizations proposing various alternative (non block-device or non FAT oriented) storage organization and management methods. The following sections include overviews of four of these alternatives which are well known and/or covered by publicly available documents.

7.1 Journaling Flash File System (JFFS2 and JFFS3)

The Journaling Flash File System [JFFS], especially its advanced derivatives [JFFS2 and JFFS3] represent an industry wide well known attempt to create a non block device emulation based NAND Flash File System, with special attention to LINUX based embedded systems.

JFFS is a log-structured file system designed by Axis Communications AB in Sweden specifically for use on flash devices in embedded systems, which is aware of the restrictions imposed by flash technology and which operates directly on the flash devices, thereby avoiding the inefficiency of having two file systems on top of each other.

The log-structured design in JFFS/JFFS2 basically means that the whole file system content, as stored on the flash, may be regarded as one large log of changes: any modification to an entity of the host file system (i.e., file change, directory creation, changing file's attributes, etc) is appended to the log. The log is the only data structure on the flash media. Modifications are encapsulated into small data structures called nodes. In principle, each node in the log contains a data object (file, directory, link) name and meta-data, along with the (variable length) related data content.

The technical details related to the implementation and representation of the JFFS2 on the flash are beyond the scope of this paper, however, the importance of this solution and its differentiating characteristics from a block-device emulation based flash file system lays in the semantics difference in the interface with the host: while in the case of the block-device emulation all the high level information regarding the data objects is transparent to the device (i.e. from the device point of view the interaction is based on reading and writing addressed sectors, all sectors are indistinguishable), in the case of the JFFS2, the atomic transactions are changes related to named data objects (e.g. File-name and additional meta-data, data segment offset, data segment length, data segment content).

7.2 SCSI Object-based Storage Device model (OSD)

The Technical Committee T10 of ANSI INCITS (responsible for the SCSI and related standards) had created a published a working draft specification for Object-Based Storage Device Commands [OSD2]. The **Object-Based Storage Device (OBSD)** is defined as a device that stores **data objects** instead of blocks of data. The purpose of this abstraction is to assign to the storage device more responsibility for managing the location and organization of the data within the storage media.

Some important definitions, used in the following description, are:

> **OSD object**: An ordered set of bytes within an object-based storage device that is associated with a unique identifier. Data in the object is referenced by the identifier and offset information within the object. Objects are allocated and placed on the media by the OSD logical unit.
> **user object**: An OSD object that contains user data that is referenced by byte offset within the OSD object.
> **root object:** An OSD object that is always present whose attributes contain global characteristics for the OSD logical unit. Each OSD logical unit has one and only one root object.
> **collection:** An OSD object in which references to one or more user objects from a single partition may be collected.
> **partition:** An OSD object used for creating distinct management domains (e.g., for naming, security, quota management).

OSD object abstraction as described in [OSD2]:

> The OSD object abstraction is designed to re-divide the responsibility for managing the access to data on a storage device by assigning to the storage device additional responsibilities in the area of

space management. Figure 3 shows the relationship between the OSD model and a traditional SBC-based (SCSI Block Commands) model for a file system.

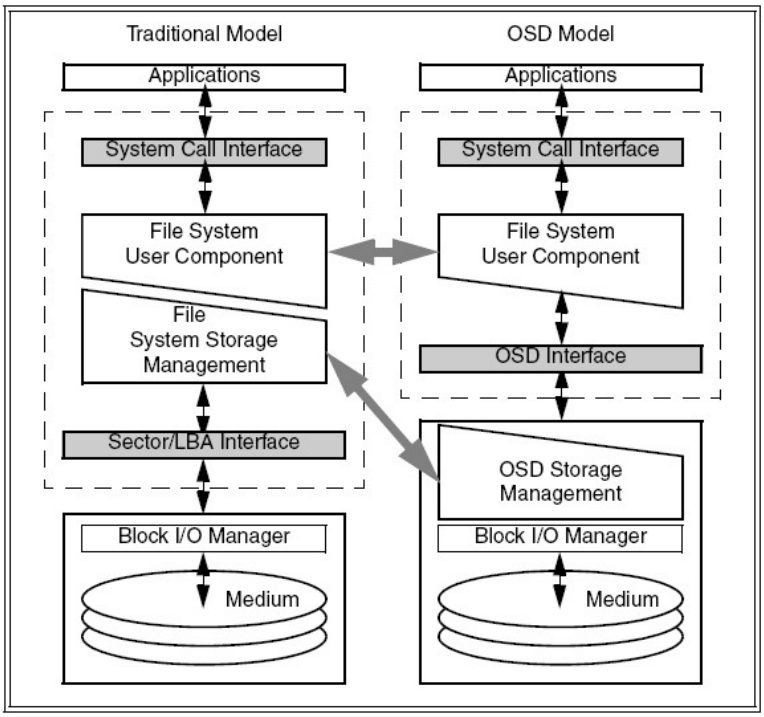

Figure 3: Comparison of traditional and OSD storage models

The user component of the file system contains such functions as:

a) Hierarchy management;
b) Naming; and
c) User access control.

The storage management component is focused on mapping logical constructs (e.g., files or database entries) to the physical organization of the storage media. In the OSD model, the logical constructs are called user objects. The root object, partitions, and collections provide additional navigational aids for user objects.

In addition to mapping data, the storage management component maintains other information about the OSD objects that it stores (e.g., size, and usage quotas, and associated username) in attributes. The user component may have the ability to influence the properties of object data through the specification of attributes (e.g. directing that the location of an object to be in close proximity to another object or to have some higher performance characteristic) via mechanisms that are outside the scope of this standard.

In this model, the OBSD makes the decisions as to where to allocate storage capacity for individual data entities and managing free space.

While the OSD model as defined by the T10 Committee applies to storage devices in general, it seems exceptionally well adapted to the specific case NAND Flash storage devices. The necessity and complexity

of managing a NAND Flash storage device usually requires the use of an intelligent device controller (the "intelligence" can be software or hardware implemented) significantly more "smart" than what is required to manage a "natural" block device (e.g. a rotating magnetic disk). This SW/HW controller can also be used to implement the added functionality (or intelligence) required for the OSD functions.

Similarly to the case of the JFFS above, the interface with the host is based upon the atomic transactions are related to named data objects as opposed to addressed sectors as used in the context of the traditional block-devices.

Additional material related to initial industrial OSD projects and implementations can be found in **[OSD PROJ1]** , **[OSD PROJ2**] and **[OSD DEMO]**. [Check for the board approval]

7.3 FTP Storage

An intelligent storage device can in principle be configured with a functionality equivalent to an ftp (File Transfer Protocol) server: the atomic interface transfer operations on such a device would be file related, i.e. read/write (or get/put) and delete single or multiple files. In a sense this could be considered as a private case of Object-based Storage Device with the "file" being the only object type.

For example a related implementation to this concept is the [PTP].

8 Attributes of the "ideal solution"

As stated above, the main objectives of the MIPI NAND Software Workgroup, in collaboration with leading Operating Systems vendors and mobile/embedded computing platform vendors is to define/specify a Standardized Software Interface between storage devices based upon the NAND Flash technology, and the OS in embedded/mobile computing platforms.

This final paragraph attempts to characterize the intended end product (i.e. the Standardized NAND Software Interface) in terms of a list of desirable attributes ("wish list"). Since the satisfaction of some of these requirements may be mutually contradictory, this list should be considered as representing an "ideal solution", whereas the actual specification resulting, might involve certain compromises and/or options and/or variations.

Thus, the Standardized NAND Software Interface should be:

- **Technologically Scalable, Extensible and Future Proof**: it should take into consideration the foreseeable evolution path of both the NAND-like storage device technology and the evolution of the storage related software technology;

- **Providing a "soft" migration path**, both for Operating Systems currently employing contemporary and pre-standard block-device emulation, and for NAND Flash vendors and their current devices;

- **Capable of continuing to support "legacy" systems and solutions,** employing pre-standard devices and/or Operating Systems (e.g. retain support for the "good old FAT" and bootability feature)

  NOTE by bootability we mean: enable the loading of the executable Operating System image from the storage device/media to the memory of the embedded application system processor main memory, and facilitate secure and controlled means and methods for storing/updating/replacing the OS image to the storage device/media).

- **Adequately address the new application domain requirements** (like DRM, compression, security/encryption, data portability) as listed in above, and also considering new/additional requirement resulting from the collaboration with the OS vendors;

- **Versatile,** i.e. compatible with, or at least adaptable to, all embedded computing platforms and Operating Systems.

- **Robust and low-risk, by** relying on well founded and established software engineering principles and practices.