Choosing a Linux File System for Flash Memory Devices

By Kurt Sowa
Numonyx Software Product Manager

When deciding which file system is best suited for a project, Linux provides many options to consider. Each file system has strengths and weaknesses, and it is important to understand what characteristics are most important to the success of your project. Whether it is performance or scalability, power loss tolerance or minimal storage overhead, there is likely a Linux file system that matches your needs.

This article focuses on file systems targeting Flash memory (but not solid state drives) as a storage medium. Flash memory requires additional management (such as wear leveling and garbage collection) that requires consideration in the file system architecture.

This article reviews the constraints that Flash memory places on file systems and the system characteristics to consider. A selection of Linux file systems appropriate for Flash memory is discussed, along with the features that might make them the correct choice for you, or eliminate them from consideration.

The Type of Flash in Your Design Impacts File System Selection

Several types of Flash memory are available, each with different characteristics, costs and impact on the final bill of materials (BOM). The type of Flash memory present in a design will impact the selection of a file system. Requirements for garbage collection (Flash memory must be erased to be rewritten) and wear leveling are common to most Flash memory devices. Other characteristics are specific to the type of Flash, such as NAND bad block management. The next sections describe the key characteristics of common types of Flash memory.

In addition to constraints placed on the file system by the type of Flash selected, the execution model for the system is also impacted by the flash type. For more information regarding the impact of code storage and execution model on Flash selection, refer to the Micron white paper: Demystifying Embedded Code Storage: Optimizing for lower cost and higher performance through balanced XIP.

NAND

NAND Flash memory connects memory cells in series and does not have the capability for random access. This means that programs cannot be executed directly from NAND memory, and must be copied to RAM first. Some NAND Flash devices do support boot capability by guaranteeing the first block or through special mapping at initialization. In this case, the bootstrap code or pre-boot loader is copied to RAM, and is used to start the boot process.

Because NAND cells are connected in series, there is a potential for interaction between the cells. As a result, NAND typically requires Error Correcting Code (ECC) to ensure that any bit errors are corrected, depending on the NAND device and application. ECC support can be implemented in the NAND controller or in software.

With NAND, there is an expectation that some blocks have defects and can fail. This adds a requirement to the file system for bad block management. In some cases, reading data can impact the device, and even static data must be rewritten to guarantee reliable reads.

NAND Flash has faster write and erase performance when compared to NOR flash. With its smaller cell size, NAND is the least expensive nonvolatile memory. It is available with single-level cells (SLC, single data bit in each memory cell) and in multi-level cells (MLC, multiple bits in each memory cell). While MLC devices have increased density at a lower cost, SLC devices provide higher reliability and endurance.

NOR

NOR Flash connects memory cells in parallel. This enables random access, and software can execute directly from NOR (XiP) without needing to be copied to RAM. It also results in faster read times for NOR Flash. NOR devices are available in SLC and MLC configurations. NOR Flash typically has higher endurance than NAND. A NOR-based system using XiP typically requires one-half to one-quarter the RAM of a comparable NAND-based solution.

Managed NAND

Managed NAND includes a built-in controller, which can provide necessary ECC, wear leveling and garbage collection while presenting a sector interface to the user. This can simplify the requirements of system software. However, because of the increased device size for the controller, Managed NAND bears a higher cost.

e·MMC™

e·MMC™ is a joint standard defined by the MultiMediaCard Association (MMCA) and the JEDEC Solid State Technology Association. This standard (currently version 4.41) defines an architecture consisting of embedded storage with an MMC interface, Flash memory and controller. In addition to providing a sector-based interface to the module, the standard extends the functionality, adding features like boot support and reliable writes.

OneNAND™

OneNAND™ is a combination of NAND, RAM and controller logic. It is designed to provide a NOR Flash interface and NOR capability using NAND devices. As with NAND, code stored in OneNAND must be copied to RAM for execution.

Phase Change Memory (PCM)

PCM is a new, nonvolatile memory technology that delivers the best features of RAM, NOR, and NAND in a single device. PCM memory is based on a chalcogenide, which is an alloy containing an element from the oxygen/sulfur family of the periodic table. This material can switch between an amorphous state with high resistance and a crystalline state with low resistance.

Like RAM, PCM memory is bit alterable. This means that they do not need to be erased before being written. PCM has write performance that matches the fast write speeds of NAND and read performance that pairs NOR's fast read times with the read bandwidth of RAM.

The capabilities of PCM make it an excellent choice for both code storage (XiP capable) and data (fast writes). PCM's bit alterability enables new paradigms and PCM can be used for application data (heap and static data). Data that was previously stored in Flash memory

can be moved to PCM for a performance increase. In fact, it is possible (while not entirely practical) to design a system with only PCM and no RAM.

Reliability, Performance, and Longevity Considerations When Selecting a Flash File System

When using Flash memory for file system storage, it is important to manage not only the data in the file system, but the Flash itself. There are several considerations that affect the architecture of a file system. These considerations, which are covered in the next few sections, can impact the reliability, performance and longevity of the file system.

Flash Type and Flash Device

The first consideration for any flash file system is whether or not it supports the flash type (NAND, NOR, *e*·MMC, etc.) planned for the system. Some file systems support only NAND or NOR devices, while some support both. Secondary to supporting the desired flash type is support for the desired flash device. If the desired flash device is not supported, there may be significant effort to add support for another device (if it is even possible).

The second consideration is the cost of the total memory solution of the system. Even in a demand paging system, the RAM requirement may still be substantially increased by the requirements of a NAND file system. Due to the slow read speeds of NAND, a NAND-based file system may buffer more data (to improve performance) than a comparable NOR file system. NOR memory allows for XiP-enabled file systems, which can decrease boot time and application launch time while further decreasing the RAM usage. On the other hand, NOR memory has a higher average cost per bit when compared with NAND densities. Thus, it is important to examine the total memory cost of the system (volatile and non-volatile memory) to see if a NOR, NAND, PCM, or blended memory solution will meet the BOM cost target at the desired performance.

Garbage Collection

Since Flash memory (with the exception of PCM) must be erased before the contents can be updated, a garbage collection process is used to recover dirty space (deleted files, etc.) so that it can be re-used.

Flash memory is segmented into blocks or pages that must be erased as a whole. These page sizes are typically much larger than the sectors (or data elements) used in a file system, and usually contain both valid and obsolete data. To recover the dirty space, valid data is copied to a new location and the block is then erased, recovering it for future use. Because the time required to copy valid data and erase a block can be significant, some file systems support garbage collection in a background thread during idle time.

Wear Leveling

Flash devices are limited in the number of erase cycles that can be performed on each block. Flash file systems track the number of erase cycles in a block and take the count into consideration when determining which block to use for a write operation. In addition, some devices restrict the number of reads allowed between erases, so static data must be rewritten to guarantee reliability. Wear leveling support in a file system manages the distribution of data within a device with the goal of maximizing the device lifespan by locating and moving data so that there is an even distribution of erase counts among the blocks of a device.

Sequential Writes and Partial Page Programming

NAND Flash memory is organized into blocks consisting of multiple pages. When erasing data from a NAND device, the entire block must be erased. However, when programming, NAND Flash is written in pages. Typically, NAND requires that these pages be written sequentially within a block. In addition, the number of write operations within a page (partial page program) is restricted. The number of write operations allowed within a page ranges from one to four writes, depending on the device.

Power Loss Tolerance

Flash file systems are often used in battery-powered portable devices (for example: mobile phones), where reliability and robustness are important factors for customer satisfaction. To this end, most flash file systems offer some degree of power loss reliability. No file system can prevent the loss of data that is in the process of being written when power is lost. However, avoiding corruption and loss of existing data is important.

Most Linux Flash file systems provide this support via journaling. Journaling file systems write a log of changes prior to making the changes in the file system. In the event of a power loss, the journal can be replayed to restore the file system.

Pre-OS Access

During the booting of a system, there are resources required by the boot loader (for example: splash screens, configuration parameters, etc.). It is advantageous to manage these resources in a file system. This allows updates and removes the need for additional code to manage these resources. Since a file system is initialized and mounted by the operating system, normal access to the files is not available early in the bootstrap process. Some file systems provide a pre-OS mode that supports read access prior to completing the OS load process.

File System Efficiency

File systems impose a structure on the data being stored. This structure includes the storage for the data itself and metadata to manage file system information, such as directories and creation time. Overhead will vary depending on the architecture of the file system. A file system designed for small data will have different overhead than one designed for large multimedia files. Memory usage, performance requirements, and even the storage capacity of a volume can also have an impact on the file system overhead.

Performance

Read and write throughput is also constrained by the file system architecture. In addition to throughput, there are many other performance considerations that can impact user satisfaction, such as finding a file, file delete time and initialization and mount time.

ECC

NAND Flash memory requires ECC to ensure that data is valid. The number of bits of ECC required to correct expected read errors changes depending on the device. In the case of 1-bit ECC, software can generally provide the ECC calculation without significant performance degradation. If two or more ECC bits are required, hardware ECC support is desired to maintain performance. How a file system manages ECC (software and/or hardware) can have a bearing on suitability.

Bad Block Management

The architecture of NAND Flash brings with it the expectation that not all blocks in a device are functional when shipped. In addition, there is an expectation that some blocks will fail during the life of the device. A robust NAND file system must be able to manage the usage of blocks within the device and prevent the usage of bad blocks. It must also manage the recovery of valid data from failing blocks, and replace the failing block with a good block from a reserve.

Open Source vs. Proprietary File Systems

In most cases, Flash memory devices are supported in the file systems commonly used in Linux. Open source file systems are widely used in multiple systems from different OEMs using a variety of flash memory devices. The large community of users generally ensures that any issues are quickly resolved and the quality of the file system is high.

Some devices are supported only by proprietary file systems. While this can restrict the file system choices available, it can bring other benefits. File systems targeted at a specific device often outperform general file systems, not only in read and write throughput, but in other areas, such as wear leveling algorithms or file system efficiency.

Selecting File Systems Tailored for Specific Use Cases

With Linux (and other operating systems) there is no reason to limit your design to a single file system. Often, it is desired to have several file systems that are tailored to specific uses. File systems can be used to manage code and/or data, and they may be read-only or modifiable. The next two sections cover two key file system use cases.

Code Management

For security and reliability issues, devices often place the code image in a read-only file system. Depending on the type of Flash memory, this could be a mix of compressed and uncompressed files. Uncompressed code in NOR Flash can be executed directly from the Flash memory device. Code in NAND and compressed code in NOR must be copied to RAM for execution. For efficiency, it is useful to store code (compressed or uncompressed) in sector sizes that match the code page of the processor in the system, allowing a single chunk read to fill a code page.

Data

User data is often a mix of a variety of file sizes and data types. The mix of data (whether predominantly large multimedia files, small files or a mix of both) can impact the overhead. Throughput requirements (for example: playing a movie without dropping frames) are also impacted by the file system architecture.

A Guide to Open-source Linux Flash File Systems

Now that we have covered the basics of Flash memory and considerations that impact file system selection, it is time to look at the file systems themselves to see how they support the requirements of your system. In this article, we look only at open source file systems that are freely available with Linux. We provide a brief description of each file system grouped by file system type (read-only or Read/Write), followed by a table comparing the file systems.

Read-Only File Systems

Read-only file systems are commonly used for code and static system parameters. Most read-only file systems can utilize compression, which requires decompression to read code or data. Some read-only file systems support uncompressed storage and code execution directly from Flash.

CRAMFS

CRAMFS is a compressed read-only file system that is designed for efficiency in resource-constrained designs. CRAMFS uses z-lib to compress and store data on a per-page basis. This file system is typically used to manage the system image that ships on a device. CRAMFS is a good selection for a file system to manage a boot image.

SQUASHFS

SQUASHFS is a compressed read-only NAND file system that can use z-lib or Lembel-Ziv-Markov for improved compression or speed when compared to CRAMFS. Unlike CRAMFS, which is limited to a 4KB compression block size, SQUASHFS uses a 4KB to 128KB compression block size to tailor the file system to your needs. The file system attempts to make up for the slow read speed of NAND with improved compression block buffering, which increases RAM requirements.

Read/Write File Systems

Read/Write file systems serve the needs of system and user storage. They can manage both code and data as required, and allow files to be created or updated. Read/Write file systems typically store files in fragments, making them unsuitable for direct code execution. There are two types of Read/Write file systems that are covered in this article: sector-based file systems and Flash-based Read/Write file systems.

Sector-Based Read/Write File Systems

Sector-based file systems, such as those typically used on a desktop system, are not designed to be used on Flash memory, but instead on a hard drive. A Flash Translation Layer (FTL) is required to handle garbage collection and wear leveling, and provide a sector interface in order to use a “desktop” file system.

EXT2/EXT3

EXT2 is commonly used as the primary file system on desktop implementations of Linux. EXT2 and EXT3 are sector-based file systems that are fairly efficient in their use of space. EXT2 is generally not used as a Flash file system because it is not power loss safe. EXT3 adds journaling to EXT2 and can be safely used on a Flash file system. However, the EXT3 journaling mechanism uses a static location that can lead to wear leveling issues. EXT3 is often used on managed flash such as SD cards or *e*·MMC.

FAT

FAT is a sector-based file system made popular in Microsoft Windows. For devices that need to mount on Windows computers, FAT provides the quickest path to Windows compatibility. FAT provides support for power loss. However, the FAT table used to organize the file system can be susceptible to corruption if power loss occurs.

Flash-Based Read/Write File Systems

Flash-based Read/Write file systems are designed to work well within the constraints imposed by Flash memory, and typically provide support for required elements such as wear leveling, garbage collection, and ECC.

JFFS2

JFFS2 is a popular general file system for Flash. JFFS2 was designed for NOR devices, but also supports NAND devices. JFFS2 is a logging file system that uses i-nodes to store data. The file system treats each flash block separately, maintaining lists of blocks that contain valid nodes (clean), blocks that have some dirty i-nodes (dirty), and blocks that are erased and available for use (free). This provides flexibility to the garbage collection algorithms and allows JFFS2 to support static wear leveling by selecting blocks from the clean list as well as from the dirty list.

Because JFFS2 builds links of nodes and tracks each block, RAM usage increases with the size of the device(s) used in the file system. With large devices, memory usage and mount time become increasingly problematic with JFFS2. For Flash volumes over 256MB, JFFS2 is not recommended due to excessive mount time and RAM usage when compared with other writable Flash file systems.

YAFFS/YAFFS2

YAFFS is a general purpose NAND file system. Unlike JFFS2, YAFFS only supports NAND. YAFFS assigns block sequence numbers to improve initialization and mount time. It also stores a representation of the file system back into Flash from RAM on shutdown. This also improves mount time.

LOGFS

LOGFS is a logging flash file system intended to resolve the mount performance and RAM scalability issues of JFFS2. In addition, LOGFS provides simple versioning capabilities in the form of “snapshots.” LOGFS is in the process of being refined.

UBIFS

UBIFS is a relatively new file system designed to correct some of the shortfalls of JFFS2 and YAFFS2. UBIFS is more predictable in initialization performance and RAM requirements, making it a good choice for a general purpose file system as designs move to larger file system volume sizes.

Conclusion

There are several Linux file systems designed for a wide variety of uses, and Micron has experienced Linux experts who can help with flash and file system selection, integration and customization. Contact your Micron representative for more information.

Table 1: Comparison of each open-source Linux flash file system

File system	Requirements	NOR	NAND	OneNAND™	e•MMC	Bad block mgmt	Garbage collection	Wear leveling	Power loss	ECC	Overhead	Performance	Comments
CRAMFS	–	X	X	–	–	No. Requires FTL or other system to manage blocks	N/A	N/A	N/A	No	Very Good	Good	• Efficient storage • Good for resource-constrained systems • Good for boot images • Primarily used in embedded systems
SQUASHFS	–	X	X	–	–	No. Requires FTL or other system to manage blocks	N/A	N/A	N/A	No	Very Good	Good	• Efficient storage • Good for resource-constrained systems • Good for boot images • Primarily used in embedded systems
AXFS	–	X	X	–	–	No	N/A	N/A	N/A	No	Very Good	Very Good	• Good for code management • Mix NAND and NOR in the same design • Most flexible of the read-only file systems

File system	Requirements	NOR	NAND	OneNAND™	e·MMC	Bad block mgmt	Garbage collection	Wear leveling	Power loss	ECC	Overhead	Performance	Comments
EXT2	Block Driver	X	X	–	X	No. Requires FTL or other system to manage blocks	Block Driver	Block Driver	FTL	Block Driver	Better	Good	• Good for cards • Poor for raw devices
EXT3	Block Drover	X	X	–	X	No. Requires FTL or other system to manage blocks	Block Driver	Block Driver	Yes - Journaling	Block Driver	Better	Better	• EXT2 extended for Power Loss
FAT	Block Driver	X	X	–	X	No. Requires FTL or other system to manage blocks	Block Driver	Block Driver	FTL	Block Driver	Poor	Poor	• PC/USB compatible
JFFS2	MTD	X	X	X	–	Yes	Yes	Active	Yes - Journaling	Yes	Poor	Good	• Init performance and RAM usage increases with volume size
YAFFS	MTD		X	X	–	Yes	Yes	Static	Yes - Journaling	Yes	Poor	Better	• Init performance and RAM usage increases with volume size, but better than JFFS2

File system	Requirements	NOR	NAND	OneNAND™	e·MMC	Bad block mgmt	Garbage collection	Wear leveling	Power loss	ECC	Overhead	Performance	Comments
LOGFS	MTD or Block Driver	–	X	–	X	No	Yes	Yes	No	No	Good	Good	• Designed for large volumes • Poor performance on block devices • Wear leveling is not robust
UBIFS	UBI	X	X	X	X	Yes	Yes	Static	Yes - Journaling	Yes	Good	Better	• Relatively new