Software Profile: Journaling Flash File System, Version 2 (JFFS2)

By Luca Boschetti
Micron Software Engineer

The Journaling Flash File System, version 2 (JFFS2) is a log-structured open-source file system that is distributed under the terms of a General Public License (GPL). JFFS2 was widely designed and used for embedded devices. Originally developed to support NOR Flash memory, JFFS2 has been included with the Linux kernel since version 2.4.10. Currently, it supports NAND and OneNAND™ Flash memory devices.

JFFS2 is developed and maintained within the Memory Technology Device (MTD) layer, which provides support in Linux for Flash devices. For further information about the MTD, refer to the *Enabling a Flash memory device into the Linux MTD* technical note.

Unlike file systems that are not specifically designed to work with Flash memory, JFFS2 does not rely on a glue logic layer to fit Flash needs (such as a Flash Translation Layer that turns Flash devices into standard block devices). Instead, JFFS2 directly uses low-level driver calls, which are provided by the MTD itself in Linux.

- As a Flash file system, JFFS2 provides:
- Block management for “dirty” blocks (garbage collection)
- The use of Flash blocks to preserve endurance (wear leveling)
- Robustness against power loss (power loss recovery)

JFFS2 Log

In JFFS2, all files operations (file creation, write/update requests, file rename and deletion) are represented as a new entry in the log. Physically, data representing these entries are written in Flash sequentially on a current block, until the block is full. Then, a new free block is chosen, and so on. Entries in the log are called nodes. All nodes have a common header that specifies the node type, length and a Cyclic Redundancy Check (CRC) value for the header itself.

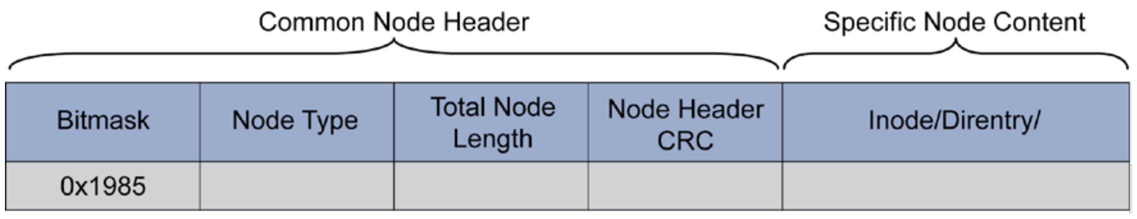


Figure 1. JFFS2 node content

Node payloads differ according to their use. Nodes representing directory entries (files and directories) store the name, inode and parent inode numbers, creation time and CRCs for the node and name. Inode nodes store a slice of file data, information about compression, all file-system-related attributes (such as permissions, user and group) and CRCs for the node and data.

For example, when creating a new 1KB file, the log's organization of data and metadata results in a write in two nodes. One represents the creation of a new file system entry, which stores the file's name, creation time and location within the file system tree. The other node stores the 1KB of data.

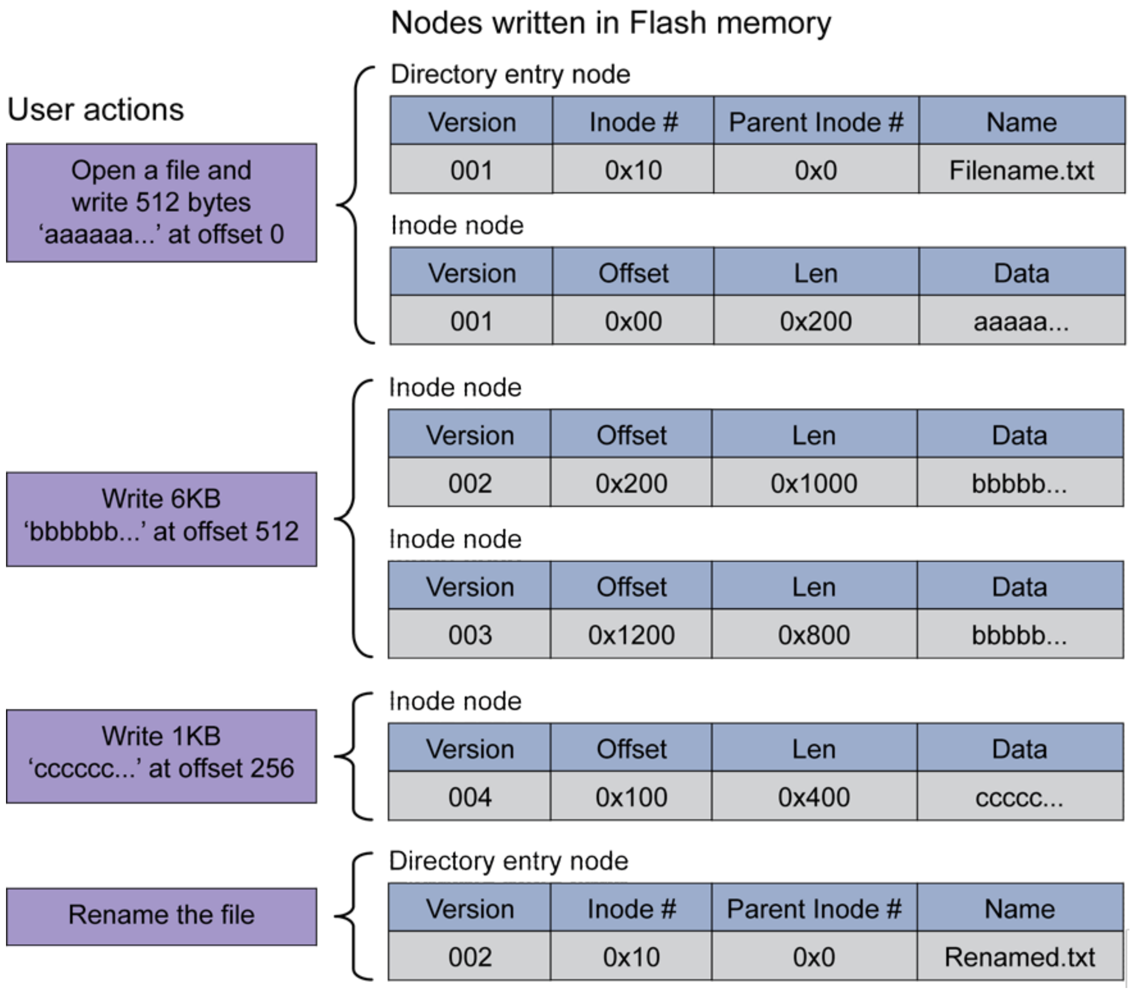
All log entries also have a version number. Each update to the file system creates a new node with a higher version number, which carries the modifications. This means that updating 100 bytes of the 1KB file in our example would make JFFS2 write to Flash a new inode node that stores only the updated bytes and the offset information for locating the new data. Similarly, renaming a file creates a new directory entry node that stores the new name, while moving a file produces a node that updates its parent inode number. Thus, a file is made up of the history stored in its nodes, and modifications borne by nodes with higher version numbers prevail over those of obsolete nodes, as long as their CRCs are correctly verified.

JFFS Log Organization

The robustness of JFFS2 lies in its log organization, which allows for easy recovery if the system fails. Each node is valid if its CRCs are correct. If the CRC is not correct because of a failure while writing, the node is simply discarded and will not contribute to the file history. With this architecture, JFFS2 is immune to system failures where file system integrity is concerned.

A JFFS2 partition will never be unable mount, and a file will never fail to be accessed. However, from user's file I/O point of view, the file system does not provide any transactional capability. Any write operation larger than the page size, which is architecture and machine model dependent, is split up in several nodes in the log. As a result, if power is lost during the write operation, part of user's write request has already been served and no mechanism will invalidate it afterwards. This leaves the file in an intermediate state, and on next power up, the log will be played back and most likely only the node being written will be corrupted, while the other nodes will be considered valid.

The following figure shows an example of file I/O and related nodes created in Flash. Notice how requesting a data write operation whose size is larger than system page size (4KB in this example, where 6KB of data is requested to be written) generates two inode nodes in the log.


Nodes written in Flash memory
User actions
Open a file and write 512 bytes 'aaaaaa...' at offset 0
Write 6KB 'bbbbbb...' at offset 512
Write 1KB 'cccccc...' at offset 256
Rename the file

Directory entry node

Version	Inode #	Parent Inode #	Name
001	0x10	0x0	Filename.txt

Inode node

Version	Offset	Len	Data
001	0x00	0x200	aaaaa...

Inode node

Version	Offset	Len	Data
002	0x200	0x1000	bbbbb...

Inode node

Version	Offset	Len	Data
003	0x1200	0x800	bbbbb...

Inode node

Version	Offset	Len	Data
004	0x100	0x400	ccccc...

Directory entry node

Version	Inode #	Parent Inode #	Name
002	0x10	0x0	Renamed.txt

Figure 2. File I/O and related nodes created in Flash memory

Resource Consumption

The log paradigm provides a high degree of reliability, even though this takes its toll on performance and memory consumption. In fact, references to all nodes of all inodes in the file system are kept in RAM to provide immediate access to all valid fragments needed to access file system entries. Building this table requires scanning at initialization all blocks in the partition, discarding obsolete nodes and allocating the RAM structures needed to keep track of fragments. RAM allocation grows linearly with the number of files in the file system and the degree of fragmentation of these files. However, once it has been built, keeping the table of indexes in RAM allows a throughput for file system I/O that is limited only by underlying Flash bandwidth. The following table shows examples of RAM usage for root partitions of different sizes.

Table 1: RAM Usage for Root Partitions of Different Sizes

Partition Size	# Inodes	RAM After Mount	Max RAM at Mount
10MB	1400	110KB	142KB
27MB	2500	198KB	256KB
64MB	5500	384KB	512KB

In addition to memory consumption, the scan process is expensive in terms of time needed, which also grows linearly. These are the root of a well-known scalability problem of JFFS2, which originally was designed for small Flash memory densities. In fact, while mounting a 64MB partition with hundreds of files and thousands of directories could take tenths of seconds. The same operation on a NAND device larger than 1GB could take up to several minutes. For this reason, using JFFS2 with partitions larger than 128MB requires careful evaluation.

Micron has worked on JFSS2 resource optimizations. Some of this effort has been aimed at removing the maximum fragment size limit that is related to the system page size. This was done to reduce the space overhead due to node metadata, the RAM needed for tables and the time needed to mount and access the file system. For more information, please contact your Micron representative.

Erase Block Summary (EBS)

To overcome the slow mount time, the Erase Bock Summary (EBS) was introduced in version 2.6.15 of the Linux kernel. At the end of each block, a set of nodes summarizes block content, which avoids the need to scan the entire block to create the table of indexes. This summary information is created during write operations, speeding up the mount process – especially in large NAND Flash memory devices. If the summary nodes are missing (for example, as a result of a power low that occurred before its creation), the mount process scans the entire block.

EBS improves mount time. However, it still is not the solution to make JFFS2 a real scalable file system because it does not reduce resource consumption. Additional RAM is needed to hold summary information at run time, and some Flash memory storage is wasted to save new EBS nodes.

Mount time and resource consumption depends on partition size, compression, etc. However, on average, mount time is reduced by a factor of four or five on large partitions, while required space overhead on Flash is around 5%.

JFFS2 Compression Support

Three algorithms are available in JFFS2 to support compression: zlib, rubin and rtime. File data is compressed in chunks as large as the hardware page size. This is done to facilitate quick decompression. The benefits of compression depend on use models. Media files that are already compressed (such as jpg images, mpg videos, zipped documents, etc.) result in wasted CPU cycles when they are recompressed. However, when applicable, compression also improves resource overhead, as the number of fragments is reduced. This reduces the space required for metadata, both on Flash and in RAM.

Garbage Collection

The garbage collection process (also known as reclaim) is the trickiest task performed by JFFS2. Garbage collection creates free space by scanning Flash blocks and determining which nodes can still be considered valid for file history, and which are obsolete (substituted by newer nodes) or invalid (corrupted by a power loss while being written). All valid data is relocated to a new block, until only old invalid data remains, at which point the block can be erased.

Garbage collection activities may start in one of two ways:

- When needed: For example, when there are insufficient spare clean blocks remaining and a write request cannot be served, garbage collection is performed repeatedly until a block can be erased and then used for future writes.
- In the background: For example, when free space goes below a predetermined threshold, garbage collection attempts to free space while the file system is not otherwise being used.

In both cases, the garbage collection process will not start if the file system is mounted in read-only mode or until all nodes read from Flash during the first scan have been checked through their CRC correctness check. The CRC check is delayed to speed up the mount process, but is mandatory before garbage collection starts, as it would be useless to move around corrupted nodes. Garbage collection picks a block and performs steps to prepare the block to be erased by moving valid nodes to the currently active block and marking it to be erased. This way, all wasted (invalid) space is left behind. By each call, only one node is moved at a time and one block is chosen for erasing, if possible.

Essential for garbage collection, JFFS2 constantly updates a series of lists by which it keeps track of the global statistical information listed in the following table:

Table 2: Global Statistical Information

List	Blocks
Clean	Blocks are 100 percent full of clean data.
Very dirty	Blocks contain a large amount of dirty space.
Dirty	Blocks contain some dirty space.
Erasable	Blocks are completely dirty, and need erasing.
Erasable pending wbuf	Blocks require erasing, but only after the current write buffer is flushed.
Erasing	Blocks are currently being erased.
Erase checking	Blocks are being checked and marked.
Erase pending	Blocks must be erased now.
Erase complete	Blocks are erased and need a clean marker written to them.
Free	Blocks are free and ready to be used.
Bad	Blocks are bad.

List	Blocks
Bad used	Blocks are bad, but contain valid data.

An algorithm relies on these lists and some thresholds set beforehand to determine when and how garbage collection should step in to try and free dirty space.

The next step is to choose the block to start from. Blocks from the "bad used" list are taken only when the number of available (Free) blocks is not greater than a conservative threshold. This is done because collecting a bad block does not result in more free space. Aside from the "bad used" list, the garbage collection process chooses blocks randomly. Blocks are taken from the "erasable" list 40% of the time. Because most of the blocks in this list will be erased directly, their chance of being picked for garbage collection is reduced compared to blocks in the "Dirty" and "Very dirty" lists. Blocks in these lists are selected nearly 60% of the time.

Occasionally, (approximately 1% of the time), blocks are selected from the "clean" list. This is done by JFFS2 to ensure Static Wear Leveling among all blocks, including clean blocks that would not typically be selected for garbage collection.

When moving pristine nodes from a block during garbage collection, the garbage collection process also attempts to optimize the space taken by valid data by merging adjacent fragments belonging to the same inode. This strategy leads to reduced Flash use due to the merging of the metadata of the nodes.

Internal node representation is also simplified, thus reducing memory consumption and improving the efficiency of data location. Even compression benefits from this because the compression algorithms take nodes as isolated compression units. This means that the less data they convey, the poorer compression results, whereas larger nodes are more likely to occupy less Flash space. In this case, the number of free blocks available to allow data merging is a threshold that is set beforehand.

JFFS2 Efficiency

With its log structure, JFFS2 can make a Flash memory device appear to be out of space, even when the logical device size has not yet been filled. If the file system changes too rapidly, the garbage collection thread may not be able to keep up.

Typically, logs or databases cause inefficiency, as many small file operations are requested continuously, wasting much space for the metadata related to small chunks of data (space overhead could reach as much as twice the space needed for data only). This worsens when the spare blocks available for the garbage collection process are limited. This means that that the partition has not been allocated enough space over and above space needed for foreseen data. As a result, the garbage collection process could not be performed at all.

Among the thresholds calculated to support garbage collection, one sets when the garbage collection thread should run in the background. Once the number of free blocks goes below this threshold, the garbage collection thread operates. Its value is set to:

$$\frac{2\%FlashSize + 100NrBlocks + BlockSize - 1}{BlockSize} + 2$$

Because this threshold determines when garbage collection starts, it may be wise to consider at least this as the amount of extra space over and above space required for data. For example, a 64MB Flash memory device with 256KB blocks would require an extra space of 8 blocks on the total amount of 256. Writing new data that exceeds the 248th block may result in a “no space left” error being reported.

Of course, the accuracy of this estimate depends on write access granularity (smaller writes result in higher metadata overhead). Nonetheless, it is legitimate to consider that with such extra space, once all available blocks have been filled up with non-critical write operations (too rapid or too small), the garbage collection algorithm can keep up and allow users to continue and operate on the file system normally.

Conclusion

JFFS2 is a good compromise between performance, space overhead and robustness, as long as the choice to use JFFS2 fits your Flash memory device and use cases. The following list provides a discussion of some of the use cases for JFFS2:

- **NOR Flash memory devices**: JFFS2 was originally designed for NOR Flash memory devices and is a good choice when using NOR. JFFS2 is typically not used for boot images, for which read-only eXecute in Place (XiP) file systems are a better choice. However, JFFS2 is a good option for embedded systems whose data is not too fragmented (log files, DBs) or updated frequently. Excluding these latter cases, it is rare to experience mount time or RAM consumption issues, even on resource-constrained systems.
- **Small to medium sized NAND devices**: Typically, this includes NAND devices up to 128/256MB. What is true of JFFS2 for NOR is typically true for NAND. In general, JFFS2 a good choice, even for larger density devices in which the file system does not change too frequently, systems with larger and fewer files and those that do not require critical performance. However, performance must considered when choosingeJFFS2 for slightly larger partitions. Of course, moving far beyond these constraints can make mount time, RAM consumption and performance issues increase dramatically.
- **Very large NAND devices**: Customers should carefully consider JFFS2 mount time, RAM usage and performance for NAND devices with densities beyond 256MB. Other file systems may be a better choice for large NAND devices. For more information about choosing the right file system on Linux, please refer to the article Choosing a Linux file system for Flash memory devices.

Linux Support for Micron Flash Memory Devices

NOR an NAND devices are enabled in the Linux kernel MTD drivers. Micron works to ensure support is available in the mainline kernel for Micron Flash memory devices. Since there are variations in distributions and kernel version, contact your Micron representative for additional information regarding your configuration.