Technical Note

A Hardware-Assisted Log-Structured File System with Temporal Locality Optimization Under Resource Constraints

Daisuke Sugisawa *

Posted Date: 24 October 2025

doi: 10.20944/preprints202510.1893.v1

Keywords: embedded systems; IoT devices; log-structured file system; temporal locality; hardware-assisted optimization; resource constraints; non-volatile memory

 doi:10.20944/preprints202510.1893.v1

Technical Note

A Hardware-Assisted Log-Structured File System with Temporal Locality Optimization Under Resource Constraints

Daisuke Sugisawa

Xander, LLC. Shibuya, Tokyo, Japan; daisuke.sugisawa.xander@gmail.com

Abstract

With the rapid advancement of IoT devices, there is an increasing demand for MPU environments that are low in overall system resources while achieving low power consumption, high energy efficiency, and high performance. Such systems must be capable of controlling multiple sensors and wireless communication modules on minimal battery power, while retaining several days to weeks of logs even under unstable wireless conditions. In logging applications, sensor data are continuously generated and written to non-volatile memory. However, conventional file systems are not designed to handle these operations efficiently. This is because they generally lack a ring-buffer–based logical interface, and attempting to simultaneously achieve low power consumption, high performance, and power-failure resilience often introduces significant overhead and complexity, ultimately degrading energy efficiency. In short, traditional file systems inherently face trade-offs between power efficiency, functionality, and performance. In typical designs, achieving low power consumption, high performance, and power-failure resilience without relying on specialized modules or custom hardware components ironically requires more resources—such as CPU cycles, power, circuit size, and component count—leading to increased complexity. Therefore, realizing these properties using only general-purpose resources remains a challenging task.

Compact/logging type: The proposed time-locality-optimized, hardware-cooperative log-structured file system achieves low power consumption, high performance, and resilience to unexpected power loss using only common components and interfaces, without special or custom hardware.

Keywords: embedded systems; IoT devices; log-structured file system; temporal locality; hardware-assisted optimization; resource constraints; non-volatile memory

---

1. Introduction

IoT devices are a category of embedded systems. In embedded system design, the choice of operating system (OS) is a critical consideration. Recently, from a security standpoint, adopting (Embedded) Linux has become the de facto standard—for example, enabling compliance verification through Software Bill of Materials (SBOM)–based security clearance.

However, under strict resource constraints, an approach that does not rely on an operating system can also be a practical and efficient solution.

 doi:10.20944/preprints202510.1893.v1

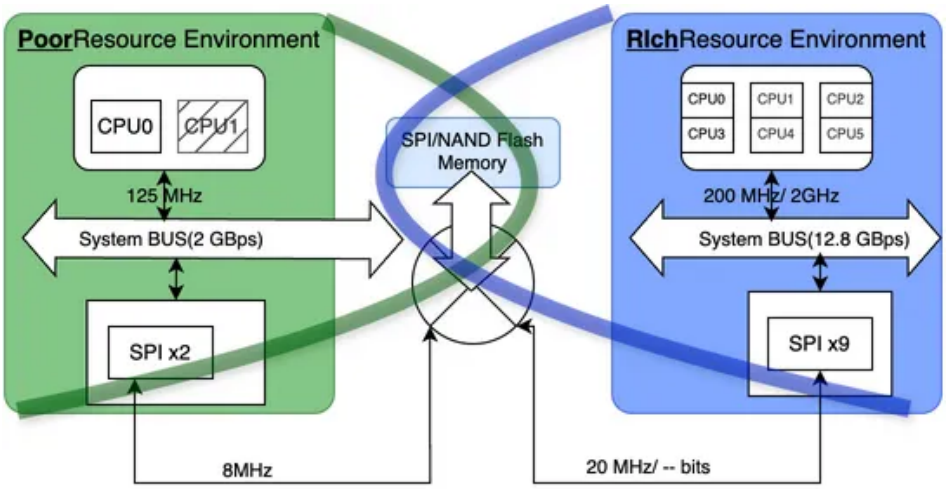


Figure 1. Hardware-Cooperative Architecture between Low-Resource and High-Resource Systems

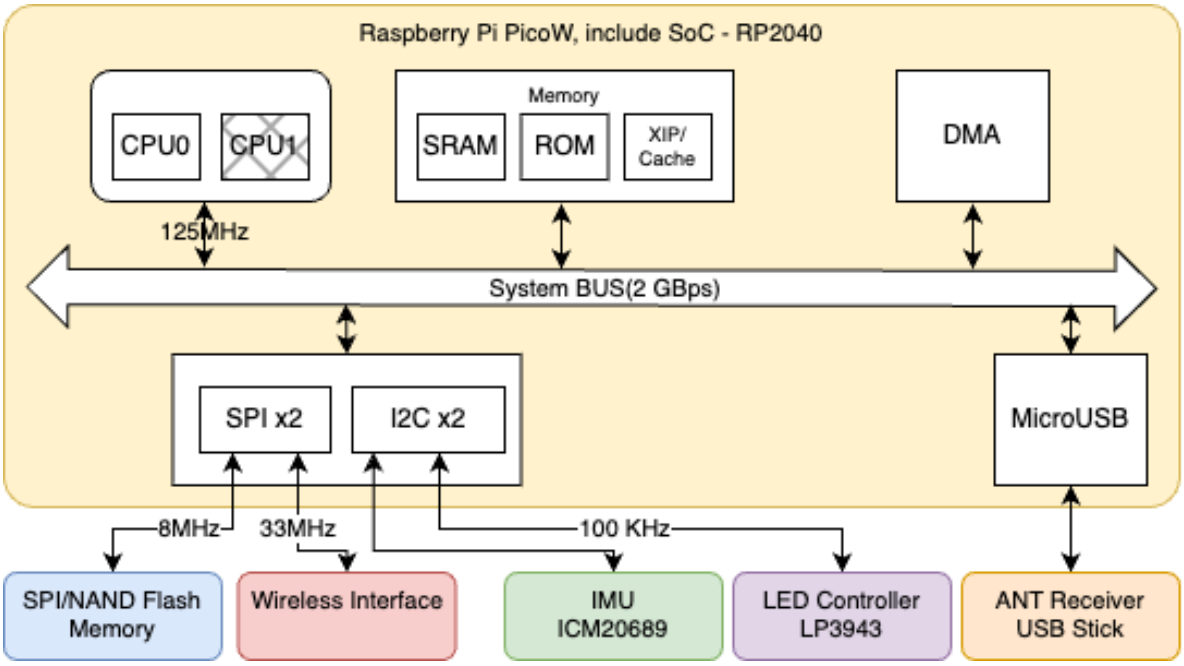


Figure 2. Module blocks

1.1. Systems Without an Operating System

For example, the **pico-sdk** for Raspberry Pi Pico is not an operating system. Instead, it provides a software development kit (SDK) that includes a ported **LWIP** (TCP/IP, UDP/IP protocols), along with hardware FIFO access, memory mapping, linker scripts, and device drivers for standard peripherals of the Pico W board.

However, it does **not** include a ported journaling file system such as **ext4**, nor embedded **FAT-based** file systems with mechanisms for resilience to unexpected power loss. Consequently, while it enables lightweight connectivity and control, it lacks robustness in non-volatile storage management.

1.2. Adoption of Embedded Linux

By contrast, adopting Linux offers significant advantages, such as access to a stable, feature-rich, and versatile file system, a mature and reliable network protocol stack, and stable device drivers. These attributes make Linux the practical standard for many embedded applications.

Nevertheless, as discussed earlier, in scenarios where constraints and requirements—such as power consumption, limited memory space, the number of attached sensors, and network communica-

 doi:10.20944/preprints202510.1893.v1

tion frequency—cannot tolerate the overhead of a full OS, a bare-metal or OS-less approach becomes more effective.

This is particularly true when porting to a target OS cannot satisfy performance or resource limitations inherent to the system design.

1.3. File System in This Use Case

In the IoT devices targeted in this study, adopting a conventional operating system is not feasible due to resource constraints defined by system requirements and hardware limitations. Under such constrained environments, the selection of an appropriate file system becomes a critical design decision.

When an OS cannot be employed, the available options for file system design are generally limited to lightweight, custom, or bare-metal implementations. These must operate directly on hardware interfaces such as SPI- or NAND-based non-volatile memory, while satisfying requirements for data reliability, low power consumption, and tolerance to unexpected power loss.

Table 1. File System Option Groups

Commercial Solutions	
Adopting commercially available file-system products can be a rational choice from an economic standpoint. These solutions often provide proven stability and power-loss resilience.	However, most commercial solutions require high evaluation and licensing costs, and their deployment involves significant time overhead due to contractual and legal processes.
Porting ext4 (Open Source)	
Porting an open-source, general-purpose file system such as ext4 to a low-resource environment. ext4 offers journaling and metadata protection mechanisms that improve resilience to unexpected power loss.	In resource-constrained environments, the overhead of journaling and caching mechanisms typically leads to resource exhaustion. Implementation and validation costs are unpredictable, making this approach impractical.
Porting LOGFS (Open Source)	
Porting an open-source log-structured file system (LOGFS) designed for flash storage. LOGFS inherently supports append-only writes and offers better power-loss protection than ext4.	While suitable for flash memory, LOGFS suffers from excessive mount-time costs proportional to the flash size. Implementation and evaluation overhead remain uncertain.
Designing a Custom Log-Structured File System	
Designing a proprietary log-structured file system fully optimized for the specific constraints and requirements of the target IoT device.	Based on pre-implementation and cost analysis, this approach provides the most economically and technically feasible solution under the given resource constraints.

2. Functional Requirements

2.1. Control of Multiple Sensors

The system is compactly designed to control multiple sensors while maintaining wireless communication capabilities. In scenarios where wireless connectivity is unstable or unavailable, the device must be capable of temporarily storing acquired sensor data in non-volatile memory areas such as

 doi:10.20944/preprints202510.1893.v1

NAND flash. Subsequently, when wireless conditions improve—or when the device switches to a serial wired connection such as USB—the stored data can be retrieved or transferred reliably.

2.2. Resilience to Unexpected Power Loss

The system must also incorporate a file system with strong resilience to unexpected power loss, particularly in low-power, battery-driven environments. Such resilience is essential to prevent data corruption and ensure continuous data logging and recovery without the need for specialized hardware components.

2.3. Focus of This Study

This paper reports on the design and evaluation of the **Data-Access / Temporal-Locality-Optimized / Hardware-Coordinated File System** for logging-type IoT devices. The proposed system addresses the above functional requirements by combining efficient data access, time-locality optimization, and hardware-level coordination, providing both robustness and performance under strict resource constraints.

Table 2. Comparison of File Systems

File System	Unexpected Power Loss	Mount/ Boot	Wear Leveling	Resource Size/ Footprint
ext4	◯	◯	—	◯
FAT (12,16,32,v)	◯	◯/—	—	◯
LogFS	◯/—	◯	◯	—
This Architecture MXFS	◯	—/◯	◯+	◯+

2.3.1. Notes

- "◯" indicates that the item is well supported or fully implemented.
- "—" indicates that the feature is not explicitly supported or not prioritized.
- "◯+" indicates that the feature is enhanced or specially optimized, providing superior performance compared to standard implementations.
- "◯/—" indicates conditional or partial support — for example, LogFS offers power-loss resilience under certain conditions but not universally.

 doi:10.20944/preprints202510.1893.v1

Table 3. Small Footprint Commercial File Systems

Tuxera Rliance Edge	
Log-structured + transactional, combining journal and log-structured approaches	https://www.ubiquitous-ai.com/products/reliance_edge/
Renesas FAT file system	
Implements power-failure support within the scope of the FAT specification	https://www.renesas.com/jp/ja/software-tool/fat-file-system
OSS no-OS FatFS (SD / SPI / RPi Pico)	
Power-failure support implemented within the scope of the FAT specification	https://github.com/carlk3/no-OS-FatFS-SD-SPI-RPi-Pico
HITACHI UltraFile	
Power-failure support implemented within the scope of the FAT specification	https://www.hitachi-solutions-tech.co.jp/embedded/service/middleware/ultra_file/index.html

3. Requirements and Characteristics

The target system is designed to operate in an environment with stringent power and resource constraints, while maintaining high performance and reliability. The key requirements and characteristics are as follows:

- **High data access performance:** The system must achieve fast read and write access to data for real-time or near–real-time processing of sensor information.
- **Strong resilience to unexpected power loss:** Data integrity must be maintained even in the event of sudden power interruptions.
- **Small footprint and low power consumption:** Both hardware and software components should be optimized for compactness and minimal energy usage.
- **Multi-sensor control capability:** The system must be able to manage and synchronize multiple sensor inputs efficiently.
- **Wireless communication support:** Integrated wireless connectivity is required for data transmission and remote monitoring.
- **Over-the-Air (OTA) updates:** Secure OTA firmware updates are supported, though this topic is beyond the scope of this paper.
- **Use of SPI/NAND flash memory:** SPI/NAND flash is advantageous due to its stable supply chain, ease of integration, large capacity, and broad interchangeability and availability.
- **Temporal data locality:** The system should provide access to sensor data stored in non-volatile memory for the past few days, ensuring locality of reference over time.
- **Graceful degradation on power loss:** Loss of unsynchronized sensor data during unexpected power failure is considered acceptable.
- **Low-power operation under poor connectivity:** The system must continue data storage and deferred transmission even in environments with unstable or limited wireless connectivity.

4. Log-Structured File System

The Log-Structured File System (LFS) is a well-known file system architecture that improves robustness against unexpected power loss by appending both payload and metadata as sequential log entries rather than overwriting existing data. This design minimizes the risk of file-system corruption, such as Master Boot Record (MBR) inconsistencies, by maintaining a write-once pattern.

 doi:10.20944/preprints202510.1893.v1

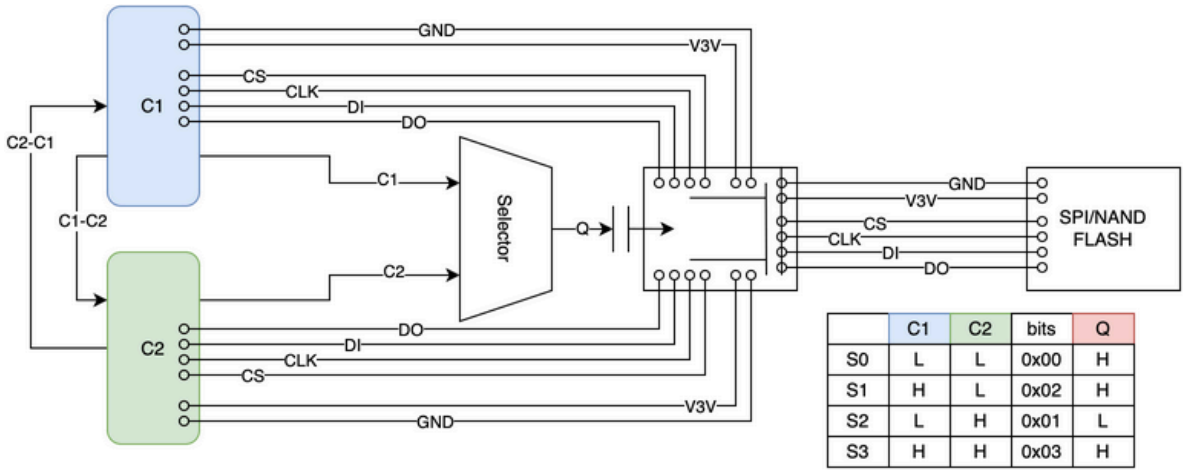
However, a notable drawback of LFS is that the time required to mount the file system increases proportionally with the capacity of the underlying SPI or NAND flash memory. When system design prioritizes high availability, this mounting latency can become significant, leading to degraded boot performance. For many IoT devices, where rapid startup is critical, this behavior represents a serious limitation.

Furthermore, as a general-purpose file system, LFS typically consumes considerably more system resources compared to domain-specific or custom lightweight implementations. This additional overhead is particularly disadvantageous in ultra-constrained environments where an operating system may not be available or where computational and memory resources are strictly limited.

4.1. Proposed Technique

The proposed technique mitigates these issues by introducing **temporally distributed processing** for SPI/NAND flash memory access and by employing **coordinated software–hardware finite-state machine (FSM) control** to manage access permissions across multiple low-resource MPUs.

This coordinated approach effectively eliminates the excessive mount-time problem observed in conventional log-structured file systems, thereby enabling a **practical and efficient file system** architecture suitable for **highly resource-constrained environments**.



	C1	C2	bits	Q
S0	L	L	0x00	H
S1	H	L	0x02	H
S2	L	H	0x01	L
S3	H	H	0x03	H

Figure 3. fsm circuit

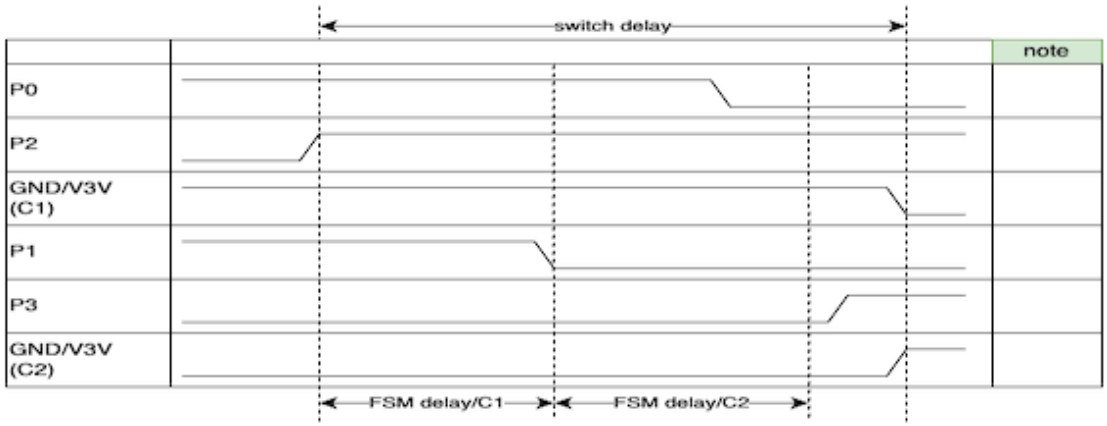


Figure 4. timing-chart

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 October 2025 doi:10.20944/preprints202510.1893.v1

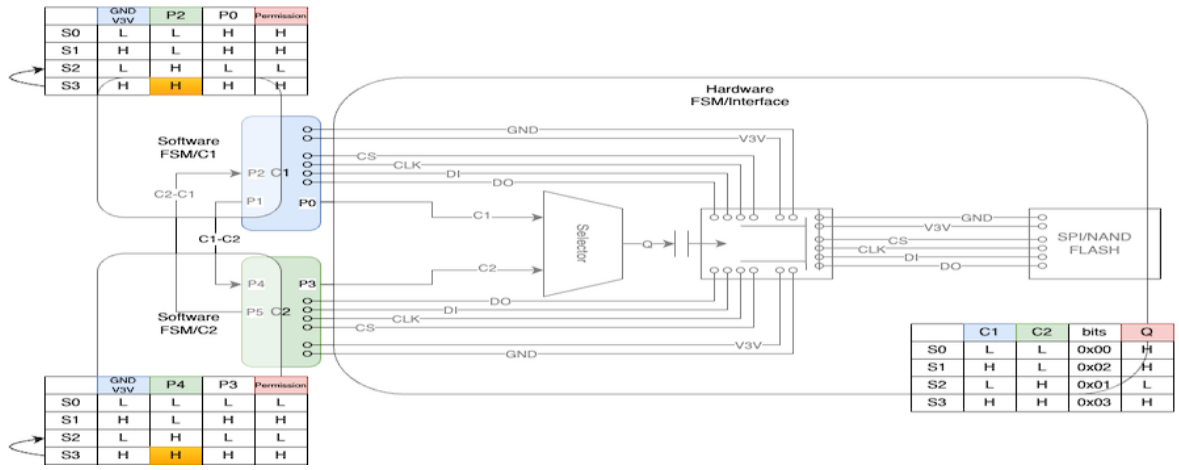


Figure 5. fsm-coordinate

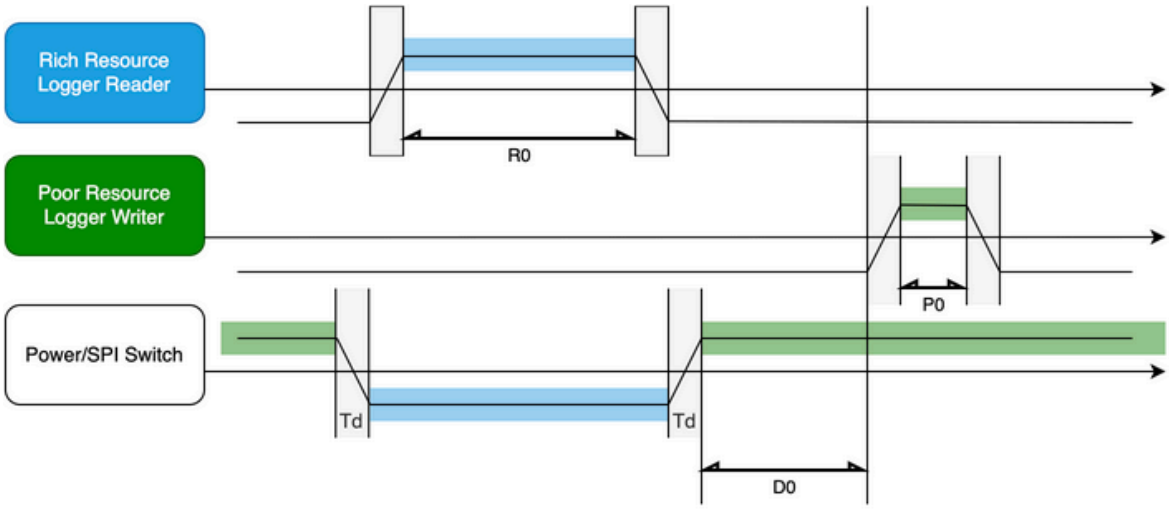


Figure 6. fsm-coordinate-switch

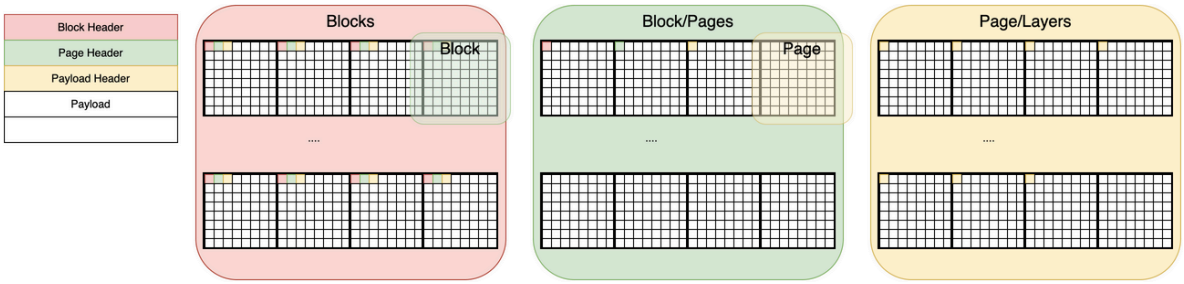


Figure 7. protocol-stack

5. Copy-on-Write (COW)

In the proposed system, the flash write timing for user payloads to the non-volatile storage area is determined by the data size after **differential compression**, with writes performed in **page-sized increments of 2,048 bytes**.

Sensor data that becomes incomplete due to unexpected power failures or other interruptions is simply skipped during subsequent read operations. Consequently, during write processing, interfaces typically required by general-purpose file systems to ensure data persistence—such as **fsync(2)**—are neither utilized nor implemented.

 doi:10.20944/preprints202510.1893.v1

In this paper, notations such as **read(2)** and **write(2)** denote **standard system calls**, whereas notations such as **append(x)**, **flush(x)**, and **read(x)** refer to **custom file-system calls** defined in the proposed implementation.

6. Fast Mount

After an unexpected power loss, the system must be capable of **rapid restart** and **prompt resumption of data streams** to ensure minimal downtime and data continuity.

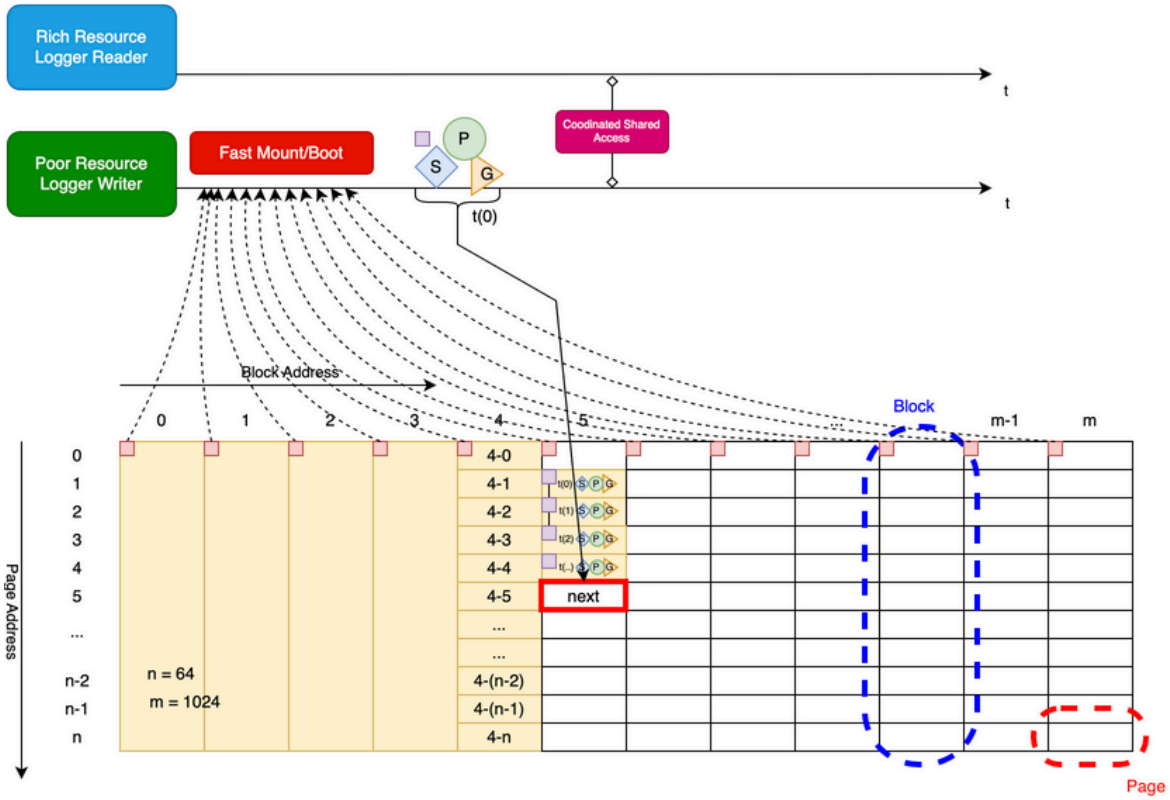


Figure 8. Fast Mount sequence

Scanning of target blocks/pages = mounting process; minimize DMA overhead.

7. Wear Leveling

Wear-leveling design is essential for long-lifetime architecture.

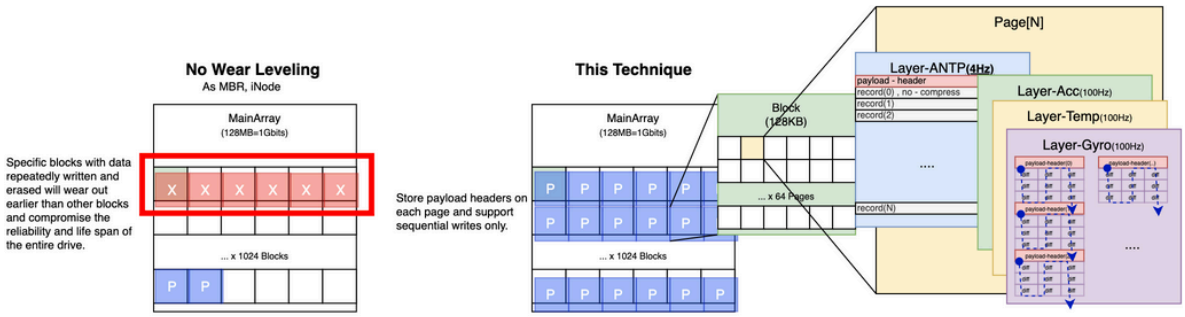


Figure 9. Wear-Leveling

8. Extending Life of NAND/Flash Memory

Delta-compression design is essential for long-lifetime architecture.

 doi:10.20944/preprints202510.1893.v1

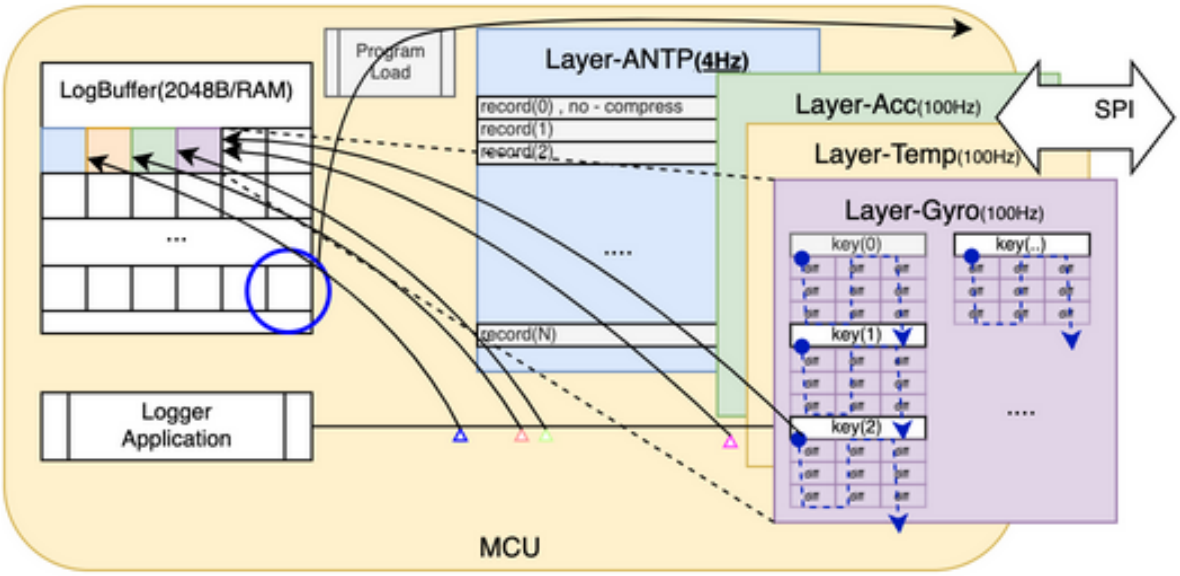


Figure 10. Log Lifetime architecture

9. Unexpected Power Loss

In common open-source file systems (such as FATxx, ext4, and log-structured variants), mechanisms like **journaling** (as in ext4) and **log-structured architectures** are well-established approaches to improve resilience against unexpected power loss. However, neither of these solutions is well-suited for **low-resource MPUs**.

Selecting ext4 often results in **resource exhaustion**, while adopting a log-structured file system incurs significant **mount-time overhead**.

Several commercial or open implementations for RTOS environments claim to provide resilience against unexpected power failures. However, most of these systems are essentially **derivatives of journaling file systems**, typically employing techniques such as **compressed transaction logs** or **distributed writes** across multiple blocks or pages. None of these designs are fundamentally robust against unexpected power loss.

The only file system currently suitable for **low-footprint IoT devices** that offers both compactness and power-failure resistance is **Reliance Edge** (Tuxera Reliance Edge Tiny Power-Failsafe File System). However, it was deemed **economically impractical** for this system due to the **time and cost overhead** associated with its **licensing process**.

 doi:10.20944/preprints202510.1893.v1

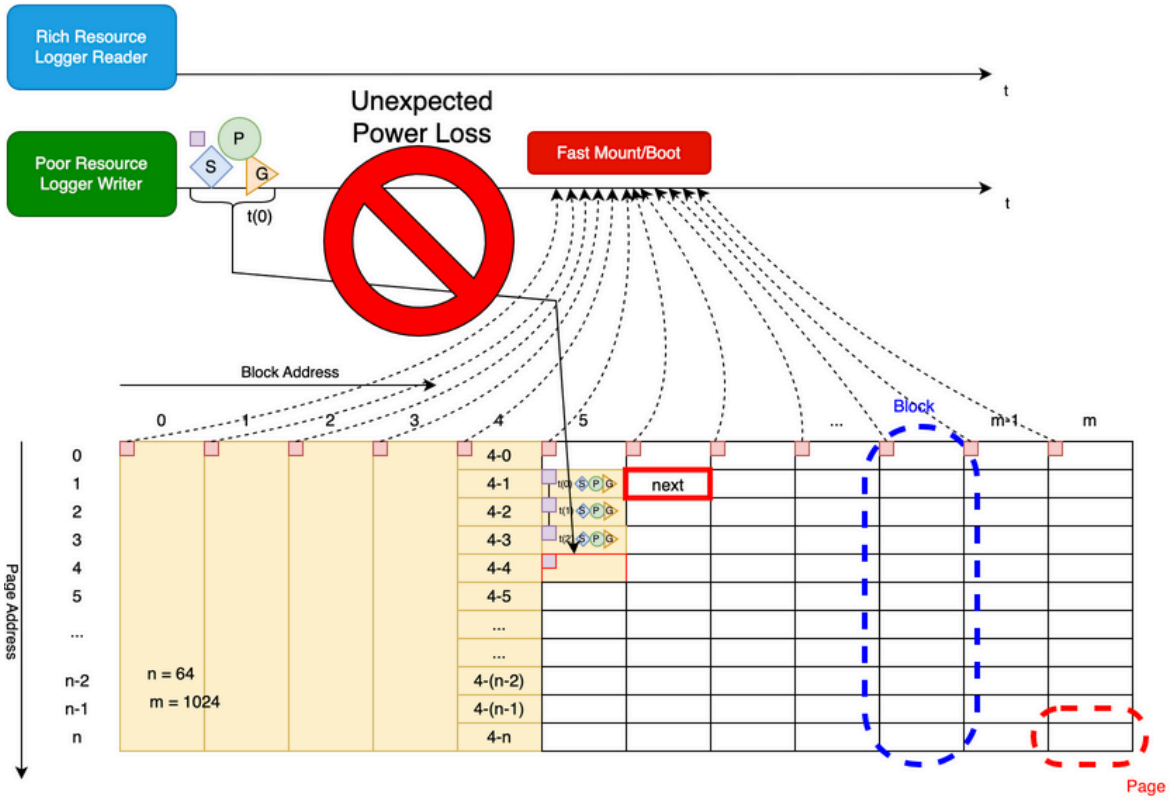


Figure 11. Unexpected Power Loss

10. Interfaces

Processes related to file names and path resolution associated with file descriptors are excluded to eliminate unnecessary overhead. Timestamps are **compressed and stored within the payload header** using an **initial value and incremental step method**, reducing metadata size and access latency.

Table 4. POSIX Interfaces

Interface	Description
open	Generate or open a file.
close	Close a file.
read	Read data from a file descriptor into a buffer.
write	Write data from a buffer to a file descriptor.
create/release	Initialize or release the file system, including mount, unmount, and boot processes.

Table 5. This article Interface

Interface	Description
append	append data to specific layer(flush is automatic).
create/release instance	Initialize/Release File system, mount/unmount, included boot.

11. DMA Performance

As the measurement results indicate, the memory access performance in both **NAND flash read** and **write** operations is **proportional to the DMA transfer size**.

 doi:10.20944/preprints202510.1893.v1

Table 6. SPI performance

AVG	Cmd Bytes	1	2	3	4	5	6	7	8	9	10
2195	write 32	2162	2162	2228	2228	2226	2162	2166	2183	2210	2229
2101	read 32	2067	2055	2192	2104	2116	2086	2092	2045	2133	2129
5632	write 1984	5926	5553	5556	5904	5559	5554	5557	5579	5558	5578
5526	read 1984	5781	5407	5431	5852	5482	5464	5511	5459	5424	5451

12. Related Works

12.1. Log-Structured File Systems Basic

The concept of a Log-Structured File System (LFS) was originally proposed by Rosenblum and Ousterhout in their seminal work, *The Design and Implementation of a Log-Structured File System* [1]. LFS was designed to address the widening imbalance between CPU speed and disk seek latency by treating the entire storage device as a sequential log. All modifications—file data, metadata, and directory updates—are appended sequentially to the log, thereby minimizing random seeks and improving overall write throughput.

The original LFS architecture, as implemented in Sprite LFS, introduced three key mechanisms: (1) segmentation of the log into fixed-size segments, (2) a segment cleaner that compacts live data to maintain large contiguous free areas, and (3) checkpointing combined with roll-forward recovery to restore a consistent state after a crash. This design demonstrated that LFS could utilize up to 70% of the raw disk bandwidth for writes—an order of magnitude higher than traditional Unix FFS—while achieving efficient recovery and high throughput for small-file workloads.

However, the recovery model in Sprite LFS assumes a continuously powered, resource-rich environment, such as workstation-class systems with sufficient CPU and memory resources and stable disk I/O. In contrast, applying the same LFS model directly to embedded or IoT systems is impractical due to (a) limited CPU frequency, (b) constrained RAM, (c) the use of SPI- or NAND-based flash memory with high access latency, and (d) frequent power interruptions. In such power-volatile environments, metadata updates must be deferred until the system enters a stable state, while sensor payloads and other data may still be appended to the log without guaranteeing immediate consistency. Recovery is therefore context-dependent: it occurs only when the system transitions to a stable power domain or is temporarily coupled to a higher-performance controller capable of checkpointing and log reconciliation.

In our implementation, this transition is supported by a hardware design that enables shared and switchable SPI-based access to the origin memory source, allowing a high-performance processor to participate in recovery only when stable power and sufficient computational resources are available.

12.2. Generic FTL

A Flash Translation Layer (FTL) provides a logical abstraction between file systems and raw NAND flash memory, managing logical-to-physical address translation, wear-leveling, bad-block management, and garbage collection [2]. Traditional FTLs, as summarized by Alahmadi and Chung [2], are typically implemented in firmware and optimized for general-purpose SSDs and eMMC

 doi:10.20944/preprints202510.1893.v1

devices under continuous power conditions. Their taxonomy includes page, block, and hybrid mapping schemes, all of which rely on maintaining large mapping tables in volatile memory to ensure consistency during crash recovery.

While these FTL-based systems are effective for enterprise or consumer-grade flash storage, they assume (1) stable power availability, (2) sufficient DRAM resources for address mapping, and (3) a unified hardware controller that performs both garbage collection (GC) and wear-leveling internally. In contrast, our proposed architecture redefines the FTL concept by integrating time-locality and hardware-level cooperation between heterogeneous MCU domains (Mother/Child nodes). The garbage collection and metadata consolidation are offloaded to a stable power domain (Mother board) equipped with sufficient resources, whereas low-power sensor nodes (Child boards) execute only append and read operations without performing GC.

This approach effectively eliminates the dual-overhead problem of FTL and FS layering observed in conventional systems [2], while maintaining consistency through a hardware-assisted finite state machine (FSM) that arbitrates SPI/NAND ownership and ensures deterministic synchronization across nodes. Thus, the proposed system is not merely "FTL-less," but rather introduces a redefinition of the FTL abstraction itself — one that is temporally and spatially distributed, aligning garbage collection and recovery operations with hardware power domains and temporal locality.

The novelty of this work lies in:

- Unifying FTL and FS responsibilities into a single log-structured control plane under severe resource constraints (less than 10 KB RAM).
- Introducing a hardware-cooperative FSM mechanism for safe multi-MCU NAND access.
- Delegating crash recovery and GC to a power-stable domain, rather than an always-on firmware process.
- Achieving high endurance and energy efficiency in IoT-class devices without external controllers or firmware-managed FTL layers.

12.3. A Survey of Data Recovery on Flash Memory

Tran and Park [3] presented a comprehensive survey on data recovery in flash-based storage, covering both Flash Translation Layer (FTL) algorithms and Power Loss Recovery (PLR) mechanisms. Their work categorized prior studies into three principal groups: (1) B-tree–based index buffer recovery methods (e.g., BFTL, IBSF, MR-tree), (2) index-segment log directory schemes (e.g., BISLD, T*-ISLD), and (3) hybrid PLR approaches (e.g., A-PLR, HYFLUR, C-HYFLUR). These solutions collectively form the foundation of FTL/PLR research, aiming to preserve metadata consistency, minimize mapping-table reconstruction latency, and maintain data integrity under power-loss conditions.

However, all of these conventional architectures share a fundamental assumption: a single homogeneous processing domain—typically one CPU clock domain or MCU controlling the entire flash subsystem. As a result, the FTL and PLR operations (such as garbage collection, wear-leveling, and recovery checkpointing) are serialized in time, bounded by the performance and power envelope of a single controller.

In contrast, the proposed system in this study extends beyond the conventional FTL/PLR model. It introduces a *multi-domain cooperative flash management architecture*, where multiple MCUs with distinct frequency and power domains operate on the same SPI/NAND interface under strict ownership arbitration. Specifically:

- **Child boards** (low-resource, low-frequency domain) perform high-speed sensor logging and sequential `append()` operations only when they temporarily own the SPI/NAND bus, ensuring minimal latency during active sensor sampling.
- **Mother boards** (high-resource, stable-power domain) assume SPI/NAND ownership asynchronously to execute computationally intensive tasks such as garbage collection (GC), wear-leveling, and power-loss recovery (PLR).

 doi:10.20944/preprints202510.1893.v1

This division of responsibilities enables **temporal decoupling of FTL and PLR functions across heterogeneous domains**. Unlike prior schemes where GC or PLR must occur synchronously in the same execution context, this architecture allows such operations to occur only when the system is in a stable power and frequency domain. The coordination is governed by a hardware-level finite state machine (FSM) that arbitrates bus access and guarantees deterministic state transitions during ownership switching.

Consequently, the proposed design does not discard FTL/PLR concepts, but rather **extends and redefines them** to operate across asynchronous hardware domains with temporal locality optimization. It achieves a continuum where:

- FTL mapping and PLR recovery are preserved as functional principles,
- but their execution timing and resource domains are decoupled and re-scheduled according to hardware-level temporal locality and power stability.

Thus, this study positions its contribution not as an alternative to FTL/PLR, but as **an evolutionary framework that advances beyond them**—integrating their logical roles into a cooperative, distributed control model that unifies log-structured file semantics, time-locality, and hardware-assisted power-domain coordination.

13. Conclusion

The management-side module writes **bitmap metadata** into the headers of each block and page, serving as structural metadata. These bitmap entries are then utilized by the **device-side module** to enable **fast booting** and efficient block scanning.

The **sensor-side module** minimizes the frequency of write operations to **non-volatile memory**, thereby reducing wear and improving overall system longevity.

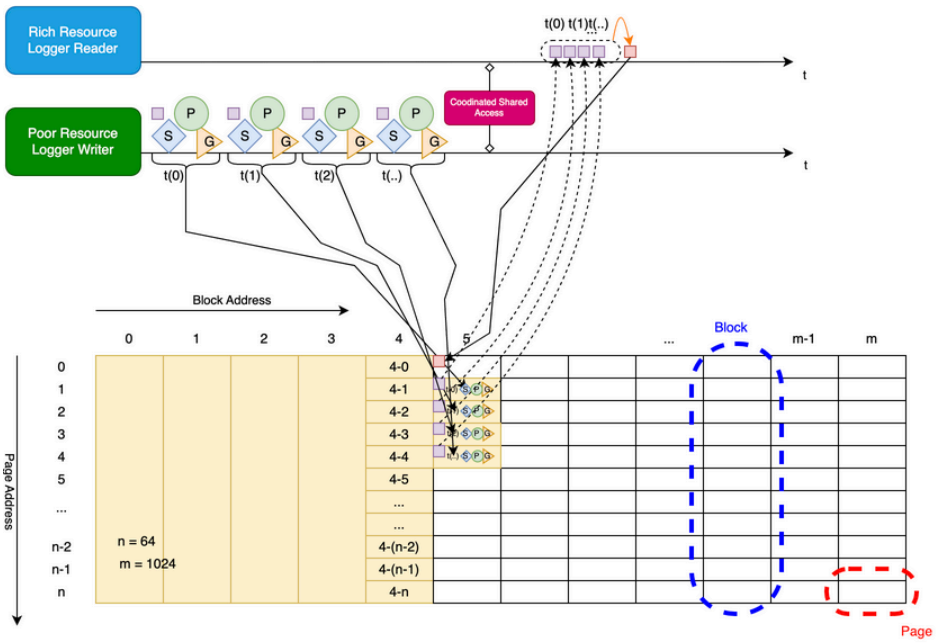


Figure 12. Data-Timing Chart

13.1. Operation Under Resource-Constrained Conditions

The system operates with **less than 10 KB of RAM**, and the file-system code size is constrained to 1 KB or less.

13.2. Deep Sleep Behavior

On the management side, the system spends the majority of its time waiting for flash-memory ownership transfer, being active for only 14.75% of the total cycle.

 doi:10.20944/preprints202510.1893.v1

13.3. Summary

This paper describes a **log-structured file system** optimized for **resource-constrained environments**, employing a shared architecture that physically switches between SPI and NAND flash connections. The design leverages **cooperative control between hardware and software finite state machines (FSMs)** and exploits temporal locality to achieve high efficiency under strict resource limitations.

13.4. Final Thoughts

I believe that a loosely coupled file system for IoT devices—composed of readily available SPI/-NAND flash memory and a generic interface microcontroller, without reliance on specialized instructions—offers competitive advantages in specific domains from an economic standpoint, particularly in terms of resilience to unexpected power loss.

Author Contributions: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data Curation, Writing-Original Draft, Writing-Review and Editing, Visualization, Supervision, Project administration: D.Sugisawa.

Funding: This research received no external funding.

AI Disclosure: The author used a large language model ChatGPT, OpenAI to assist in proofreading, grammar checking, and improving the clarity of expressions. The author reviewed and is responsible for the final content.

Conflicts of Interest: The author declares no conflicts of interest.

References

1. M. Rosenblum and J. K. Ousterhout, The Design and Implementation of a Log-Structured File System, ACM Transactions on Computer Systems, Vol. 10, No. 1, February 1992, pp. 26–52.
2. Alahmadi, A., and Chung, T. S. 2023. Crash Recovery Techniques for Flash Storage Devices Leveraging Flash Translation Layer, A Review. Electronics, 12(6), 1422. https://doi.org/10.3390/electronics12061422
3. V. D. Tran and D. J. Park, A Survey of Data Recovery on Flash Memory, International Journal of Electrical and Computer Engineering (IJECE), vol. 10, no. 1, pp. 360–376, Feb. 2020. https://doi.org/10.11591/ijece.v10i1.pp360-376