Filesystems for Embedded Devices-LittleFS

Emna Bouguerra

Abstract

This project presents a forensic analysis of the LittleFS filesystem. The goal was to explore the internal structure of LittleFS, understand how files and metadata are stored, and build a custom tool to inspect and recover data from a raw binary image. Through manual hex inspection and Python scripting, I identified key structures such as the superblock, metadata tags, and inline file entries. A script was developed to recover a deleted file successfully.

■ Contents

■ INTRODUCTION

In this project, our goal is to study how the LittleFS filesystem works and understand why it is suitable for embedded systems. These systems usually run on small microcontrollers with limited RAM and ROM, and are often connected to SPI NOR flash chips with a few megabytes of storage. Because of these constraints, using traditional filesystems like FAT is not possible, and a lightweight and reliable alternative is needed.

Flash memory comes with its own challenges. You can't just overwrite data writing to flash requires two steps: programming (which turns bits to 0) and erasing (which resets bits back to 1). Erasing is slower and requires an expensive and destructive operation, which gives flash its name. On top of that, power loss is very common in embedded devices, which can happen at any time without warning. This makes it hard to keep data safe and consistent.

LittleFS was created to solve these problems. It is designed to be safe against power cuts, to spread writes across the memory to avoid wear, and to work with a small, fixed amount of memory. It also offers a simple interface similar to POSIX that makes it easier to use.

1. Design Principles of LittleFS

1.1. Power-Loss Resilience:

LittleFS is copy-on-write and log-structured meaning it never overwrites live metadata or file data, instead it writes to a new block then commits the changes atomically , so if the power fails during a write , the old version remains untouched.

How does it work internally?

- **Writes are buffered:**Data and metadata changes are staged in RAM before being written, which allows the littleFS group changes and commits them all at once.
- **Copy-on-write(COW):**Every update (rename, write,mkdir, delete) creates a new version of structures(directories, files) For example, modifying a file creates a new version of its metadata, written to a new block.
- **Atomic commit:**After all new blocks are written, LittleFS performs a final commit: it updates a reference pointer(e,g, head of the directory) to point to the version; this update is atomic. If a crash occurs before the commit, the new data is ignored; if it happens after, the old version is garbage.
- **Rollback is automatic:**On mount, LittleFS scans the metadata and picks the most recent valid version of the directory tree. Broken or half-written blocks are ignored.

Garbage collection: Orphaned blocks from old versions are not immediately erased. LittleFS reclaims them later via background garbage collection, This delay means deleted files or overwritten data can still be recovered if GC hasn't run.

Forensics Insights:

- can scan raw blocks for: Orphaned but valid file data, Old metadata blocks (directories, headers)
- Deleted or overwritten files may persist for a while

1.2. Dynamic wear leveling:

Flash memory has a limited number of erase (write cycles per block, 10,000 to 100,000), so without wear leveling, some blocks get used too much and others stay untouched, which causes premature failure of memory. Wear leveling ensures that all blocks are used more evenly over time.

Static vs Dynamic wear leveling:

- **Static:**Tracks active,inactive blocks. Periodically move static(unchanging) data to new locations, which ensures that even blocks that store rarely updated files are erased and rewritten.
- **Dynamic(used by LittleFS):**(Optimal for embedded systems with limited flash cycles.)Tracks only blocks that change and ensures new writes go to a different physical block.

How LittleFS does it?

- **Copy-on-write design:**Data is never overwritten in place , every change get written to a new block.
- **Free block pool**: LittleFS keeps track of free and available blocks, it rotates through them over time, choosing a new one each update.
- **Garbage collection(GC):** When space runs low, GC is triggered to reclaim orphaned or outdated blocks.
- **Bad block detection:**If a block fails, LittleFS marks it as bad and avoids it in the future.

Forensics Insights:

- Since blocks are rotated frequently, deleted or outdated data might survive in multiple locations.
- A forensic tool can scan blocks outside the current FS to recover deleted content
- Bad blocks might contain partial data, but are still detectable.

1.3. Bounded RAM/ROM in LittleFS:

The problem in embedded systems

They have just tens of kilonytes of RAM, and minimal ROM space. Other filesystems like FAT requires:

- Dynamic memory allocation
- Structures that grow with the FS size.
- Recursion, which can crash on small stacks
  ==> Not acceptable on microcontrollers

How LittleFS solves it

- **Fixed RAM usage:**It uses a constant amount of RAM, no matter how many files or directories exist. You can mount and read a 1GB LittleFS volume on a device with 4KB of RAM. This is possible because:
  - LittleFS never holds the full FS structure in memory(unlike FAT)
  - It only reads the part it needs from the block device-flash)
  - It uses a small set of pre-allocated buffers for:
    * **read cache:**For data read from flash(16–256 bytes)
    * **Program cache:**For data being written to flash(16–256 bytes)
    * **Lookahead buffer:**to manage free block allocation(64–512 bytes)
- **No Recursion:**
  - Many FS use recursive functions to walk directory trees or copy files
  - In LittleFS, directory traversal is done iteratively, not recursively.
  - This avoids blowing the call stack when listing deep directories or walking trees.

- **No hidden malloc():**
  - LittleFS is designed to work without dynamic memory allocation at all, you can provide a statically allocated buffer at lfs_config init
  - Or use malloc() , but it's optional.

2. Filesystem Interface and Usage

2.1. Core Filesystem Structures

Before using any LittleFS function, the user must define and manage three core structures:

- **lfs_t** Filesystem instance:Holds internal filesystem state. Passed to all API calls like mount, unmount, file_open
- **lfs_file_t** File handle:Represents an open file. Used with lfs_file_open, read, write, and close.
- **lfs_dir_t** Directory handle:Represents an open directory. Used with lfs_dir_open, read, and close to list directory contents.

2.2. The `lfs_config` Structure

LittleFS is configured through a user-defined structure, `lfs_config`, which determines how the filesystem interacts with the underlying block device. This configuration includes low-level device operations (such as read, write, erase), storage geometry (block size and count), memory management parameters, and optional static buffers to avoid dynamic memory allocation. These parameters allow developers to tune the filesystem's performance and memory usage to suit specific embedded environments.

Block Device Operations

These are mandatory user-provided functions that allow LittleFS to interact with the underlying storage device (e.g., SPI flash, SD card in SPI mode, etc.)

- **read()**: Reads a sequence of bytes from a specified offset within a block into a RAM buffer. This function allows LittleFS to request arbitrary reads at byte granularity, though the read_size parameter defines the minimum alignment.
- **prog()**:Writes data to a block at a given offset. Flash memory must be erased before being programmed. This function must enforce that the target region has been erased. If a write error or corruption is detected, it may return LFS_ERR_CORRUPT, which will trigger relocation logic in LittleFS.
- **erase()**: Erases a full block, resetting it to its default erased state.
- **sync()**: is called by LittleFS to ensure that all write operations are truly stored in flash , especially before closing a file, doing a commit, or unmounting the filesystem.It's needed because On many storage systems, writing data doesn't instantly reach non-volatile memory. It may be held in a cache or buffer so sync() makes sure all pending operations are flushed and the data is physically stored.
- **context**: is just a pointer that you can use to pass your data or configuration to the block device functions (read, prog, erase, sync), Suppose you're using multiple flash devices. You can store which one to use inside the context. Every time LittleFS calls your read() or write() function, it passes this context along, so you know what to operate on.

Block Geometry and Limits

These fields in the lfs_config structure define the physical layout and size constraints of the block device.

- **block_size**: Size (in bytes) of each erasable block.Must be a multiple of both read_size and prog_size.Large block sizes (e.g., 4096B or 128KB) mean each file or directory uses at least one full block
- **block_count**: Total number of blocks available on the device.

- **read_size**: Minimum size (in bytes) for read operations.
- **prog_size**: Minimum size (in bytes) for program (write) operations.
- **block_cycles**: Controls dynamic wear leveling. After block_cycles writes to a metadata block, LittleFS will relocate it to a new block.

RAM Buffers and Memory Usage

LittleFS is designed to operate with a minimal and deterministic memory footprint, especially important for embedded systems with very limited RAM. These fields configure internal caches and optional statically allocated buffers to control memory use precisely.

- **cache_size**:Size of each cache used for reading and writing blocks.Must be a multiple of both read_size and prog_size.
- **lookahead_size**:Used for dynamic block allocation.LittleFS stores a bitmap to track which blocks are free,each byte can track 8 blocks, so a lookahead_size of 16B can track 128 blocks.
- **read_buffer**:Optional static buffer used for read cache.
- **prog_buffer**:Optional static buffer used for write cache.
- **lookahead_buffer**:Optional static buffer used for block allocation tracking.

Filesystem Metadata and File Limits

These optional configuration fields allow developers to fine-tune the limits and behavior of metadata compaction, filename lengths, file sizes, and how much space is reserved for inlined or structured data. They are especially useful for bounding performance, preventing resource exhaustion, and adapting the filesystem to constrained environments.

- **compact_thresh**: Threshold for metadata compaction during garbage collection. It helps reduce log growth and improve performance.
- **name_max**: Limits the length of filenames in bytes. This helps reduce RAM usage and metadata overhead when working with large directories.
- **file_max**: Maximum size (in bytes) for individual files.Useful for enforcing storage quotas or preventing oversized file writes.
- **attr_max**: Limits the size of custom attributes stored with files or directories.
- **metadata_max**: Maximum size allocated for metadata.
- **inline_max** : Maximum size for files stored inline (inside metadata block).

2.3. Filesystem Lifecycle Functions

Before any file or directory operations can be performed, the filesystem must be formatted (if new) and mounted. These lifecycle functions are used to initialize, attach, and detach the filesystem at runtime.

- **lfs_format()**

```
int lfs_format(lfs_t *lfs, const struct lfs_config *config);
```

Figure 1. Function prototype of `lfs_format()`.

- – This erases and initializes the filesystem layout.
- – Must be called only once on a new or raw block device.
  - * **lfs**:Pointer to the filesystem context object.
  - * **config**:Pointer to the filesystem configuration.
- **lfs_mount()**

```
int lfs_mount(lfs_t *lfs, const struct lfs_config *config);
```

Figure 2. Function prototype of `lfs_mount()`.

- – Validates the configuration against the on-disk superblock.
- – Reads and initializes internal state.
- – After mounting, you can perform file and directory operations.
- – lfs_t and lfs_config must both remain allocated while mounted.

- **lfs_unmount()**

```
int lfs_unmount(lfs_t *lfs);
```

Figure 3. Function prototype of `lfs_unmount()`.

- – Must be called before deallocating or reusing the lfs_t object.
- – Ensures proper cleanup of buffers and cached state.

2.4. General Filesystem Operations

- **lfs_remove()**

```
int lfs_remove(lfs_t *lfs, const char *path);
```

Figure 4. Function prototype of `lfs_remove()`.

- – Removes the file or directory at the specified path.
- – Directories must be empty before removal.
- – Returns 0 on success or a negative error code on failure.

- **lfs_rename()**

```
int lfs_rename(lfs_t *lfs, const char *oldpath, const char *newpath);
```

Figure 5. Function prototype of `lfs_rename()`.

- – Moves the object at oldpath to newpath.
- – The destination must not conflict in type (i.e., file vs dir).
- – Returns 0 on success or a negative error code.

- **lfs_stat()**

```
int lfs_stat(lfs_t *lfs, const char *path, struct lfs_info *info);
```

Figure 6. Function prototype of `lfs_stat()`.

- – Fills the info structure with type, name, and size.
- – Useful for listing contents or inspecting individual paths.
- – Returns 0 on success or a negative error code.

2.5. File and Directory Information:`lfs_info`

The lfs_info structure is used to get information about files and directories in LittleFS. It is filled automatically when reading a directory, and tells you the name, type, and size (for files only).

- **type**:Indicates the object type: LFS_TYPE_REG (file) or LFS_TYPE_DIR (directory).
- **size**:File size in bytes (only valid if type == LFS_TYPE_REG).
- **name**:Null-terminated name of the file or directory (max LFS_NAME_MAX + 1 bytes).

For Forensics

This structure is also very useful when writing forensic tools to:

- List all files and directories
- Find their names and sizes
- Check what type each item is (file or folder)

2.6. Files Operations

LittleFS implements a POSIX-inspired interface, meaning its file and directory operations are named and behave similarly to traditional UNIX file APIs (such as open, read, write, and close).

2.6.1. Opening Files

There are two ways to open a file in LittleFS:

- **lfs_file_open()**

```
int lfs_file_open(lfs_t *lfs, lfs_file_t *file,
                  const char *path, int flags);
```

Figure 7. Function prototype of `lfs_file_open()`.

- Opens a file at the given path with the specified flags (see lfs_open_flags)
- If LFS_NO_MALLOC is defined, this function fails with LFS_ERR_NOMEM(because lfs_file_open() dynamically allocates memory (using malloc()) for file-related buffers but Some embedded systems forbid or avoid dynamic memory allocation , so we use as an alternative lfs_file_opencfg() instead)

- **lfs_file_opencfg()**

```
int lfs_file_opencfg(lfs_t *lfs, lfs_file_t *file,
                     const char *path, int flags,
                     const struct lfs_file_config *config);
```

Figure 8. Function prototype of `lfs_file_opencfg()`.

- Same behavior as lfs_file_open(), but allows fine-tuned control via lfs_file_config.
- Needed when dynamic allocation is disabled (e.g., real-time embedded systems).

2.6.2. Reading from a File

- **lfs_file_read()**

```
lfs_ssize_t lfs_file_read(lfs_t *lfs, lfs_file_t *file,
                          void *buffer, lfs_size_t size);
```

Figure 9. Function prototype of `lfs_file_read()`.

- Reads up to size bytes from the current file position into buffer.
- Returns: Number of bytes read on success,0 on end-of-file, Negative error code on failure.

2.6.3. Closing and Syncing Files

LittleFS separates closing and synchronizing (sync) to give developers control over when file data is committed to flash memory. This is important in embedded systems where power loss, wear leveling, and performance are all concerns.

- **lfs_file_close()**

```
int lfs_file_close(lfs_t *lfs, lfs_file_t *file);
```

Figure 10. Function prototype of `lfs_file_close()`.

- Closes the file and ensures all pending data (writes or metadata) is committed to the storage device.
- It also frees any resources (buffers, caches) used for that file.

- Closing a file implies a full sync, so no separate lfs_file_sync() is needed afterward.

- **lfs_file_sync()**

```
int lfs_file_sync(lfs_t *lfs, lfs_file_t *file);
```

Figure 11. Function prototype of `lfs_file_close()`.

- Flushes any pending writes without closing the file.
- This is useful for:
  - ∗ Long-running logs: You want to flush regularly but keep the file open
  - ∗ Power-loss resilience: Sync after critical data is written
  - ∗ Incremental saving: Flush after every N writes to reduce risk

2.6.4. File Manipulation and Positioning

This section describes how LittleFS allows users to write to files, move the file pointer, and query or modify the file's size.

- **lfs_file_write()**

```
lfs_ssize_t lfs_file_write(lfs_t *lfs, lfs_file_t *file,
                           const void *buffer, lfs_size_t size);
```

Figure 12. Function prototype of `lfs_file_write()`.

- Writes size bytes from the buffer to the file at the current position.
- Returns the number of bytes written or a negative error code.
- The data is not immediately written to storage , it stays in RAM until lfs_file_sync() or lfs_file_close() is called.

- **lfs_file_seek()** and **lfs_whence_flags**

```
lfs_soff_t lfs_file_seek(lfs_t *lfs, lfs_file_t *file,
                         lfs_soff_t off, int whence);
```

Figure 13. Function prototype of `lfs_file_seek()`.

lfs_file_seek() moves the file pointer to a new position relative to the current, start, or end of the file. The exact behavior is controlled by a whence flag from the `lfs_whence_flags` enum:

- LFS_SEEK_SET (0): Set position relative to the beginning of the file.
- LFS_SEEK_CUR (1): Set position relative to the current position.
- LFS_SEEK_END (2): Set position relative to the end of the file.

```
1 lfs_file_seek(&lfs, &file, 10, LFS_SEEK_SET);
      // Go to byte 10
2 lfs_file_seek(&lfs, &file, -4, LFS_SEEK_CUR);
      // Move 4 bytes back
3 lfs_file_seek(&lfs, &file, 0, LFS_SEEK_END);
      // Jump to end of file
```

- **lfs_file_truncate()**

```
int lfs_file_truncate(lfs_t *lfs, lfs_file_t *file, lfs_off_t size);
```

Figure 14. Function prototype of `lfs_file_truncate()`.

- Truncates the file to the specified size.

– If size is less than current length, the file is shortened.
– Only available if LFS_READONLY is not defined.

• **lfs_file_tell()**

```
lfs_soff_t lfs_file_tell(lfs_t *lfs, lfs_file_t *file);
```

Figure 15. Function prototype of `lfs_file_tell()`.

– Returns the current file pointer (equivalent to lfs_file_seek(..., 0, LFS_SEEK_CUR))
– Useful for tracking write progress or read position.

• **lfs_file_rewind()**

```
int lfs_file_rewind(lfs_t *lfs, lfs_file_t *file);
```

Figure 16. Function prototype of `lfs_file_rewind()`.

– Resets the file pointer to the beginning.
– Equivalent to lfs_file_seek(..., 0, LFS_SEEK_SET)

• **lfs_file_size()**

```
lfs_soff_t lfs_file_size(lfs_t *lfs, lfs_file_t *file);
```

Figure 17. Function prototype of `lfs_file_size()`.

– Returns the current size of the file in bytes.
– Internally, this is implemented via lfs_file_seek(..., 0, LFS_SEEK_END).

2.6.5. *File Open Flags* `lfs_open_flags`

Flag	Value	Description
LFS_O_RDONLY	0x0001	Open file for read-only access.
LFS_O_WRONLY	0x0002	Open file for write-only access.
LFS_O_RDWR	0x0003	Open file for both reading and writing.
LFS_O_CREAT	0x0100	Create the file if it does not exist.
LFS_O_EXCL	0x0200	Fail if the file already exists).
LFS_O_TRUNC	0x0400	Truncate the file to zero size if it already exists.
LFS_O_APPEND	0x0800	Move file pointer to the end before each write (append mode).

2.7. Directory Operations

LittleFS allows users to create, open, iterate over, and navigate directories using a simple, POSIX-inspired API. These functions operate on a lfs_dir_t object, which tracks directory state during iteration.

2.7.1. *Creating and Navigating Directories*

Table 1. Directory Operation Functions in LittleFS

Function	Description
lfs_mkdir()	Creates a new directory at the specified path.
lfs_dir_open()	Opens an existing directory for iteration.
lfs_dir_close()	Closes a previously opened directory and frees resources.
lfs_dir_rewind()	Resets directory position to the beginning.
lfs_dir_seek()	Moves directory pointer to a previously stored offset.
lfs_dir_tell()	Returns the current position in the directory iteration.

2.7.2. *Reading Directory Entries*

```
int lfs_dir_read(lfs_t *lfs, lfs_dir_t *dir, struct lfs_info *info);
```

Figure 18. Function prototype of `lfs_dir_read()`.

• Fills in a lfs_info struct with:The name of the entry,Type (LFS_TYPE_REG or LFS_TYPE_DIR),Size (if it's a file)
• We Use this function inside a while() loop to iterate through all entries in a directory.

Example:

```
struct lfs_info info;
while (lfs_dir_read(&lfs, &dir, &info) > 0) {
    printf("Found: %s\n", info.name);
}
```

2.8. Filesystem-Level Operations

These functions expose global information about the entire filesystem.

2.8.1. *Filesystem Info and Size*

• **lfs_fs_stat()**Retrieves metadata about the mounted filesystem (block size, version, etc.).
• **lfs_fs_size()**Returns the number of allocated blocks (estimated used space).

2.8.2. *Traversing All Used Blocks*

```
int lfs_fs_traverse(lfs_t *lfs, int (*cb)(void*, lfs_block_t), void *data);
```

Figure 19. Function prototype of `lfs_fs_traverse()`.

• **cb**:A pointer to a callback function you define.
• **data**:A pointer to user-defined context that will be passed to the callback.

The callback is called for every block currently allocated, including:

• Metadata blocks
• File content blocks
• Inlined file structures (if they consume blocks)
• Superblocks and tail structures

2.9. Filesystem Summary Information:

`lfs_fsinfo` The lfs_fsinfo structure stores general information about the filesystem, such as block size, version, and filenames and file sizes limits. It is part of the filesystem metadata and can be accessed from the superblock or runtime state.

- **block_size**:Size of a logical block in bytes (e.g., 4096B). All file and metadata operations are based on this size.
- **block_count**:Total number of blocks in the filesystem. Defines the maximum storage capacity.
- **name_max**:Maximum allowed length of filenames (in bytes).
- **file_max**:Maximum file size allowed (in bytes).
- **attr_max**:Maximum size for custom attributes attached to files or directories.
- **disk_version**:The ondisk format version used by the filesystem. Useful for compatibility.

Observation: block_size and block_count fields appear in lfs_config and lfs_fsinfo structures.

- In `lfs_config`, they describe the physical device and are provided by the user(What you say the hardware looks like)
- In `lfs_fsinfo`, they reflect the on-disk filesystem configuration and are stored during formatting.(What the filesystem was actually created with.)
  ==> Consistency between both is required for a valid mount.

3. Internal Design and Implementation

This section provides an overview of the high-level architecture and core implementation decisions behind LittleFS. It focuses on how the filesystem represents data internally, handles updates safely, and manages flash memory efficiently, especially under the constraints typical of embedded systems.

3.1. Existing Filesystems

Before starting , let's talk about what we already have..What's already out there ? There are, of course,many filesystems out there, but not all of them are suited for embedded systems. To understand where LittleFS fits, it's helpful to look at other designs, especially in terms of power-loss resilience and flash wear management.

- **Block-based filesystems**, like FAT and ext2, are some of the oldest and simplest. They divide storage into blocks and store files directly within them. These systems are fast and small, but they don't handle power cuts well. If power is lost while writing, the filesystem can be corrupted. They also don't do wear leveling data is always written to the same place, which wears out flash memory faster.
- **Logging filesystems**, like JFFS, YAFFS, and SPIFFS, take a different approach. Instead of overwriting data, they treat storage like a log new changes are simply added to the end. This makes power-loss recovery easy: just ignore incomplete logs. It also spreads writes across the entire storage, helping with wear. However, reading files requires scanning the log, which can be slow.
- **Journaling filesystems**, like ext4 and NTFS, combine the two ideas. They write changes to a journal before applying them to the main filesystem. This allows them to recover from crashes, while keeping the performance of block filesystems. But they still tie data to fixed locations, so they don't solve wear leveling, and the added journal increases complexity and code size.
- **Copy-on-Write (COW) filesystems**, like Btrfs and ZFS. These write updates to new blocks and then update pointers, all the way up to the root. This gives atomic updates and better wear leveling, since data moves around. However, COW systems can suffer from "write amplification," where a small change causes many updates, and the root blocks may wear out faster than the rest.

Each design has its own trade-offs. LittleFS borrows ideas from several of these, but adapts them to work with very limited resources which is what makes it so interesting.

3.2. Merging Logging and Copy-on-Write: The LittleFS Approach

LittleFS combines ideas from logging filesystems and copy-on-write (COW) designs to achieve atomicity, performance, and flash wear control. On their own, both logging and COW structures have serious drawbacks — but LittleFS merges them in a way that avoids their worst limitations.

- **Logging LittleFS takes a different approach.** Instead of one big global log, it uses many small local logs, one per file or directory called *metadata pairs*. Each of these logs uses only two blocks, and contains just the recent history of that file or directory.

  - This makes updates atomic and power-loss safe.
  - Reading is fast, since only two blocks need to be scanned.
  - RAM usage is small and fixed, no matter how many files are stored.

  By bounding the size of each log, LittleFS avoids the performance problems of traditional logging filesystems, while keeping the benefits of safety and wear distribution. This design is a key reason why LittleFS works well on low-RAM microcontrollers.

- **Copy-on-write** LittleFS solves the limitation mentioned in the previous subsection with a strategy called Copy-on-Bounded-Writes (CObW):instead of updating parent blocks immediately after every change, LittleFS delays these updates. It only creates a new parent block after a fixed number of modifications (n). This bounds how often writes move up the block tree, which spreads wear more evenly and reduces the total number of writes.

  - This still provides atomic updates and consistency.
  - It reduces wear on frequently modified structures like the root.

  By combining CObW with its small metadata logs, LittleFS avoids both the recursive update cost of traditional COW and the memory bloat of full logging systems,making it efficient and durable on small flash storage.

3.3. Metadata pairs

Metadata pairs are the fundamental structure LittleFS uses to store filesystem metadata like directory contents, file descriptors, and the root structure.Each pair consists of two flash blocks that work together as a small append-only log.

3.3.1. Why Two Blocks?

Flash memory allows bits to be programmed from 1 to 0, but reverting them back to 1 requires erasing the entire block. Because of this limitation, in-place updates are not possible.

To ensure safe and atomic metadata updates, LittleFS uses a two-block system for each metadata pair. One block contains the last valid state, while the other is used for new writes. If the update completes successfully, it becomes the new active block; if interrupted (e.g., by power loss), the system can fall back to the previous one.

This design provides reliable power-loss resilience and simplifies recovery without requiring complex metadata tracking.

```
metadata pair pointer: {block 0, block 1}
                         |        '-------------------.
                         '-.                           |
disk                       v                           v
.--------.--------.--------.--------.--------.--------.--------.
|        |        |metadata|        |metadata|        |        |
|        |        |block 0 |        |block 1 |        |        |
|        |        |        |        |        |        |        |
'--------'--------'--------'--------'--------'--------'--------'
                       '-.            .----'
                          v          v
        metadata pair .---------------.---------------.
                      |  revision 11  |  revision 12  |
        block 1 is    |---------------|---------------|
        most recent   |      A        |      A''      |
                      |---------------|---------------|
                      |   checksum    |   checksum    |
                      |---------------|---------------|
                      |      B        |      A'''     | <- most recent A
                      |---------------|---------------|
                      |     A''       |   checksum    |
                      |---------------|---------------|
                      |   checksum    |      |        |
                      |---------------|      v        |
                      '---------------'---------------'
```
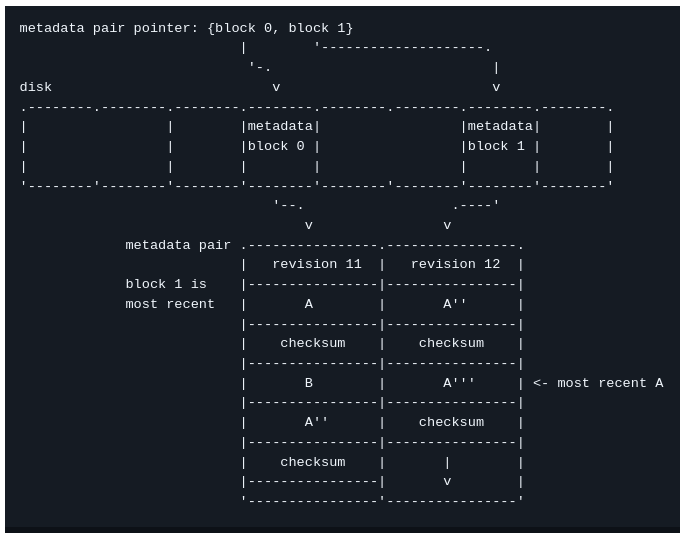
Figure 20. Structure of a metadata pair in LittleFS. Each pair consists of two blocks, where entries are appended and committed using a checksum. The most recent valid revision is determined by comparing revision numbers, allowing safe fallback in case of power loss. In this example, block 1 holds the latest revision (12), and A''' is the most recent version of entry A.

3.3.2. How Updates Work?

- Entries (e.g., filenames, directory entries) are appended to the current active block.
- Once a group of updates is ready, it is committed with a CRC32 checksum for validation.
- If power is lost mid-update, the older block remains valid and the checksum allows detecting incomplete updates.

3.4. Compaction in LittleFS(Garbage collector)

When a metadata block gets filled with entries, not all of them are still needed. For example, a file might be renamed or deleted — so earlier entries for that file are now outdated. These are considered garbage.

3.4.1. how it works?

- **Scan the full block:** LittleFS looks at all entries and checks if each one has a newer version later in the log.
- **Keep only the most recent entries:** Outdated ones are discarded. The latest versions are copied to the second block of the metadata pair.
- **Commit to the new block:**Once the valid entries are in place, a new checksum is written. The block becomes the new active one, and the old one is erased.

```
metadata pair pointer: {block 0, block 1}
                         |        '-------------------.
                         '-.                           |
disk                       v                           v
.--------.--------.--------.--------.--------.--------.--------.
|        |        |metadata|        |metadata|        |        |
|        |        |block 0 |        |block 1 |        |        |
|        |        |        |        |        |        |        |
'--------'--------'--------'--------'--------'--------'--------'
                       '-.            .----'
                          v          v
        metadata pair .---------------.---------------.
                      |  revision 11  |  revision 12  |
        block 1 is    |---------------|---------------|
        most recent   |      A        |      A''      |
                      |---------------|---------------|
                      |   checksum    |   checksum    |
                      |---------------|---------------|
                      |      B        |      A'''     | <- most recent A
                      |---------------|---------------|
                      |     A''       |   checksum    |
                      |---------------|---------------|
                      |   checksum    |      |        |
                      |---------------|      v        |
                      '---------------'---------------'
```

Figure 21. Compaction(GC).

3.4.2. Why It's Safe

Because the new block isn't "live" until the checksum is written, if power is lost during compaction: The old block is still intact and the system will ignore the partially compacted one This ensures atomicity and power-loss resilience even during GC.
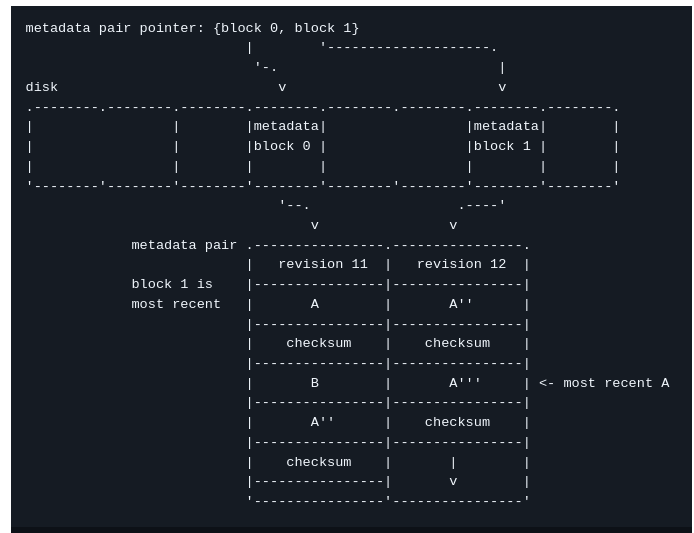
3.4.3. Splitting Metadata Pairs

- If the block is full and no more compaction is possible (i.e., all entries are still valid), LittleFS splits the metadata pair:
    - It creates a new metadata pair and moves some entries there.
    - The two pairs are linked using a tail pointer.
- This forms a lightweight, scalable linked list of small logs instead of a single large log.

3.5. CTZ skip-lists

Metadata pairs are ideal for safe and atomic updates, but they are space-inefficient. Each pair uses two blocks, and due to splitting and compaction overhead, metadata entries can consume up to four times their actual size. For this reason, LittleFS stores file contents using a custom copy-on-write (COW) structure called a backward CTZ skip list.

3.5.1. How It Works?

- The data blocks are linked backwards: the newest block points to the previous one.
- Some blocks contain extra pointers to skip further back ,this creates a layered 'skip list'.

3.5.2. CTZ Rule:

- For block number n, it stores ctz(n) + 1 pointers.
- Example: block 8 (1000b) → 3 trailing zeros → 4 pointers.

This creates fast traversal paths:Skipping through blocks is efficient and Worst-case read time is O(n log n), but append remains O(1).

```
A backwards CTZ skip-list
.--------. .--------. .--------. .--------. .--------. .--------.
| data 0 |<-| data 1 |<-| data 2 |<-| data 3 |<-| data 4 |<-| data 5 |
|        |<-|        |--|        |<-|        |--|        | |        |
|        |<-|        |--|        |--|        |--|        | |        |
'--------' '--------' '--------' '--------' '--------' '--------'
```
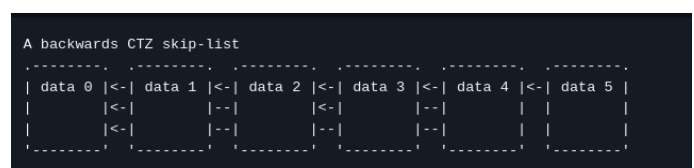
Figure 22. A backward CTZ skip-list in LittleFS.

To keep storage overhead predictable and avoid excessive complexity, the design assumes an average of two pointers per block, which has been shown to provide good runtime behavior in practice without exceeding typical flash block size limits.

3.6. Cooperation Between Metadata Pairs and CTZ Skip-Lists

CTZ skip-lists give us a COW data structure that is easily traversable in O(n), can be appended in O(1), and can be read in O(n log n). All of these operations work in a bounded amount of RAM and require only two words of storage overhead per block. In combination with metadata pairs, CTZ skip-lists provide power resilience and compact storage of data.

3.6.1. Step 1: Original File Structure

Figure 23. AOriginal File Structure.

- The file is stored as a backwards CTZ skip-list.
- Each block (e.g., data 3) points backward to the previous one (data 2, data 1, etc.).
- The metadata pair holds a pointer to the current head of the file (data 3) and the file size
- No updates have occurred yet.

3.6.2. Step 2: Copy-on-Write Append (New Data Write)

Figure 24. New Data Write.

To append data 4 and data 5, LittleFS:

- Does not modify existing blocks.
- Instead, creates new blocks via copy-on-write:
    - Copies data 2 and data 3 → new data 2, new data 3
    - Appends data 4, data 5 to form a new tail:
- At this stage, the metadata still points to the old file, meaning the update is not yet committed.

3.6.3. Step 3: Commit via Metadata Pair Update

Figure 25. Metadata Pair Update.

Finally, once all data is safely written:

- The metadata pair is updated to point to the new head of the CTZ skip-list (data 5).
- This update is protected by a CRC checksum, ensuring atomicity.
- If a power failure occurs before this point, the metadata still points to the old file structure, so nothing is lost.

3.7. The block allocator

So we now have the framework for an atomic, wear leveling filesystem. Small two block metadata pairs provide atomic updates, while CTZ skip-lists provide compact storage of data in COW blocks.Where do all these blocks come from?

In a copy-on-write filesystem like LittleFS, nearly every write requires a fresh block. Ensuring power resilience and wear leveling makes block allocation a central component of the design.

3.7.1. No Free List – Just Scan

Rather than maintaining a fragile on-disk free list or bitmap (which could be corrupted during a crash), LittleFS infers free blocks by scanning the current state of the filesystem. Any block not reachable from the root is considered free. "Drop it on the floor" strategy: once metadata no longer references a block, it's forgotten and can be reused.

3.7.2. Lookahead Buffer – Efficient Scanning

To avoid scanning the entire disk on every allocation, LittleFS uses a small lookahead buffer (bitmap) in RAM:

- Tracks a moving window of blocks.
- Searches it first when allocating.
- If no free block is found, it triggers a scan to refill the buffer.
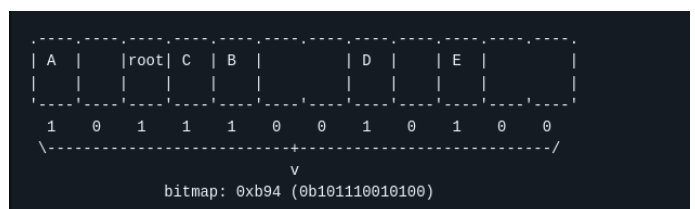- This balances performance with LittleFS's bounded RAM guarantee.

Figure 26. Lookahead Buffer

3.7.3. Runtime Behavior

Here's an example log of block allocation:

- Shows how the allocator rotates across storage.

- One or two scans are usually enough.
- Complexity is O(n²) worst-case, but fast in practice.

```
boot...        lookahead:
               fs blocks: fffff9fffffffffeffffffffffff0000
scanning...    lookahead: fffff9ff
               fs blocks: fffff9fffffffffeffffffffffff0000
alloc = 21     lookahead: fffffdff
               fs blocks: fffffdfffffffffeffffffffffff0000
alloc = 22     lookahead: ffffffff
               fs blocks: fffffffffffffffeffffffffffff0000
```

Figure 27. LittleFS log output showing live allocation and scanning cycles.

3.8. Wear leveling

3.8.1. Bad Block Detection and Recovery

- LittleFS performs immediate verification after each write.
- If the written data does not match what's in RAM, the block is marked as bad.
- The system evicts the bad block and retries the write on a new block using copy-on-write (COW).
- If all blocks are bad, LittleFS returns an "out of space" error, a safe and expected failure mode.

3.8.2. Read Errors and ECC(error-correction-codes) Support

ECC is an extension to the idea of a checksum. Where a checksum such as CRC can detect that an error has occurred in the data, ECC can detect and actually correct some amount of errors.

- Unlike writes, read errors can't be recovered without redundancy.
- LittleFS does not implement ECC itself, due to RAM and block size constraints.
- However, it fully supports external ECC mechanisms, such as those in NOR/NAND flash chips or driver layers.

3.8.3. Dynamic Wear Leveling

- LittleFS only supports dynamic wear leveling, which distributes writes across unused (dynamic) blocks.
- It does not move static data (static wear leveling), keeping the implementation lightweight.

3.8.4. Statistical Approach to Wear Distribution

LittleFS doesn't use counters or history to track how often blocks are written. Instead, it tries to simulate even wear over time by controlling where it starts writing.

During normal use:

- It writes to blocks in order (block $0 \rightarrow 1 \rightarrow 2$ ...).
- Once it reaches the end, it wraps around to the beginning.
- This ensures that all blocks get used, not just a few early ones.

After a reboot / power cycle:

- If it always restarted from block 0, early blocks would wear out faster.
- To avoid that, LittleFS starts allocation from a different block each time it's mounted.

Randomized Starting Point

- LittleFS calculates a pseudo-random offset based on checksums of metadata blocks (which are already being read during mount).
- This becomes the new starting block for allocation.
- This way, writes start from a new position every time, reducing repeated wear on the same blocks.
- it's not a real RNG because Because the random number only changes when the filesystem changes: If you don't modify the filesystem, the checksums don't change $\rightarrow$ the offset stays the same.No writes = no need to randomize.

To summarize, LittleFS starts writing from a new place every time it's mounted, based on metadata to avoid wearing out the beginning of storage. But it only gets a "new place" when the filesystem changes.

3.9. Files

3.9.1. Default file storage

In its basic form, LittleFS stores each file using a dedicated metadata pair and CTZ skip-list.
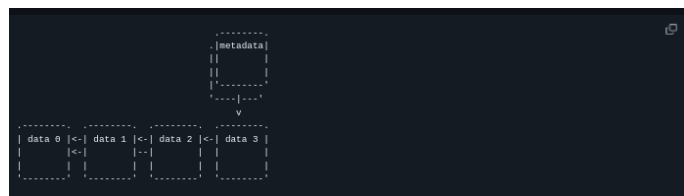
Figure 28. Default file storage in LittleFS using a metadata pair and a CTZ skip-list.

- Metadata Pair (Inode):Each file is associated with a metadata pair (two blocks), which contains:
    - A revision number for tracking updates.
    - The file's name.
    - A pointer to the CTZ skip-list, which describes where the file's data is stored.
    - A checksum to detect corruption and ensure atomicity.

- CTZ Skip-List:The CTZ skip-list stores the actual data blocks of the file

3.9.2. File Storage Optimizations

Shared Metadata Pairs

Initially, each file could be stored using its own metadata pair, which includes a revision counter, skip-list pointer, and checksum. However, this approach incurs high overhead: even a 4-byte file might occupy up to 12 KiB of storage (3 blocks of 4 KiB). To address this, LittleFS allows multiple files to share a single metadata pair, typically by associating it with a directory. This significantly reduces the collective storage cost.

Figure 29. Shared Metadata Pairs.

Inline Files

For very small files, even the shared metadata pair model may be too expensive. LittleFS introduces inline files, where both the file's metadata and its actual data are stored directly inside the directory's metadata pair. This method is ideal for files smaller than 1/4 of the

block size. For example, a 4-byte file may take only 16 bytes of storage when inlined, offering excellent space efficiency..
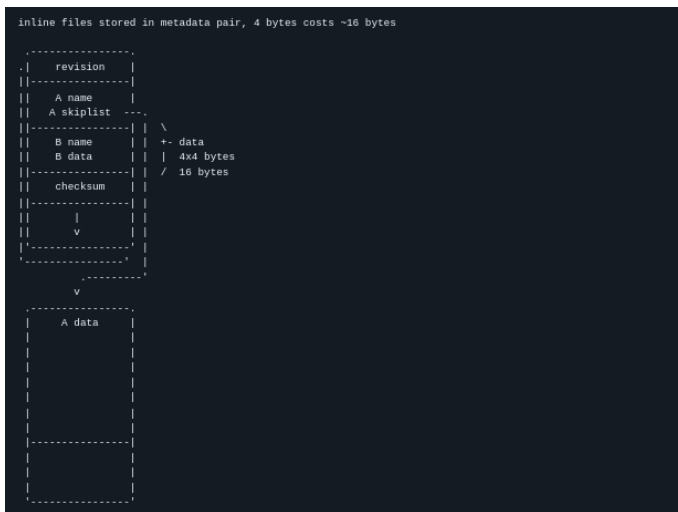
Figure 30. Inline Files.

3.10. Directories

In LittleFS, directories are stored as linked lists of metadata pairs, where each pair can hold multiple file or subdirectory entries. This design allows directories to grow without predefined limits and supports atomic updates.

3.10.1. Directory Structure and Traversal

To make traversal possible with bounded RAM, LittleFS maintains a threaded linked list across the entire directory tree. Each directory entry is linked not only hierarchically (as in a typical tree) but also sequentially, forming a flat linked list of all directories.

Figure 31. Directory Structure.

3.10.2. Orphans

An orphan occurs when a directory exists in the threaded list but hasn't yet been added to its parent.

- dir B is allocated → temporarily orphaned.
- It's inserted into the threaded list → visible to traversal.
- It's added to the parent (root) → orphan resolved.
- If power is lost during these steps, an orphan may remain.
- On next boot, LittleFS scans for orphans and cleans them.

3.10.3. Block Replacement Handling

If a block (e.g., dir B) goes bad during an update:

- A new block is allocated with the same data.
- The new block is inserted into the threaded list.
- The reference in the parent directory is fixed to point to the new block. This process may also create a half-orphan, which is cleaned on boot.

3.10.4. Deletion of Directories

To remove a directory:

- First, it is removed from the parent directory, making it an orphan.
- Then, it's removed from the threaded list.
- Finally, its blocks are returned to the free list.

4. littlefs technical specification (important for the tool)

4.1. Disk Layout Basics

- **Block-based structure:**littlefs is a block-based filesystem. The disk is divided into an array of evenly sized blocks that are used as the logical unit of storage.
- **Block addressing:**Block pointers are stored in 32 bits, with the special value 0xffffffff representing a null block address.
- **Alignment constraints:**In addition to the logical block size (which usually matches the erase block size), littlefs also uses a program block size and read block size. These determine the alignment of block device operations, but don't need to be consistent for portability.
- by default, all values in littlefs are stored in little-endian byte order.

4.2. Directories / Metadata pairs

4.2.1. littlefs technical specification

Each block in a metadata pair is structured as a small append-only log containing metadata commits. These commits track file and directory changes while ensuring power-loss resilience.
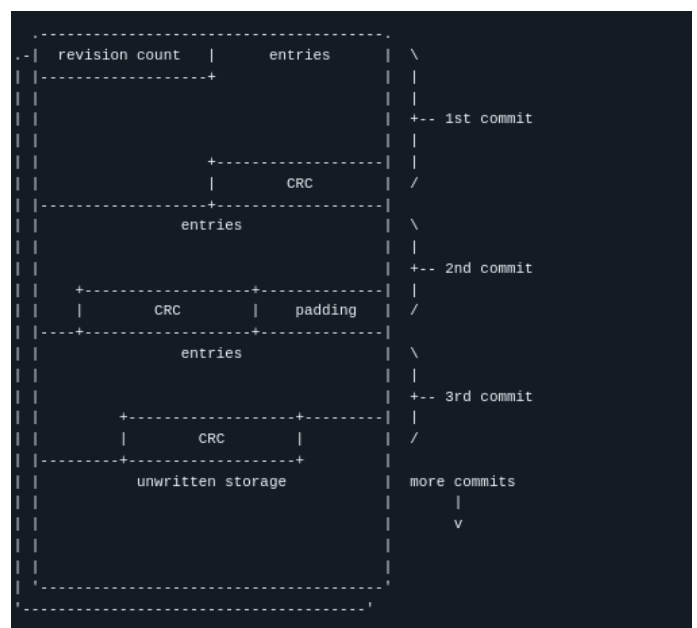
Figure 32. Metadata block.

A metadata block contains the following:

- **Revision Count (32-bit):**Increments every time the block is erased. It's used to determine which block (out of the pair) is

the most recent. Since revision counts can overflow, sequence comparison (modulo arithmetic) is used to identify the newer block.

- **Commits:** Each commit contains:
    - A sequence of metadata entries
    - A 32-bit CRC (using polynomial 0x04C11DB7, init 0xFFFFFFFF). This validates the integrity of the entire commit.
- **Padding (if needed):** Since flash writes must align to the device's program block size, commits may include padding bytes after their entries to align the CRC write.

Commit Format

- Multiple commits can be stored sequentially.
- Commits are validated by their CRC.
- Only the latest valid commit is used at runtime.
- Remaining space in the block is left unwritten to allow future appends without needing an erase.

4.2.2. *Forward/Backward Traversal XOR Tags*

To allow both forward and backward traversal of entries in a metadata block without doubling the space, littlefs encodes tags using XOR chaining.
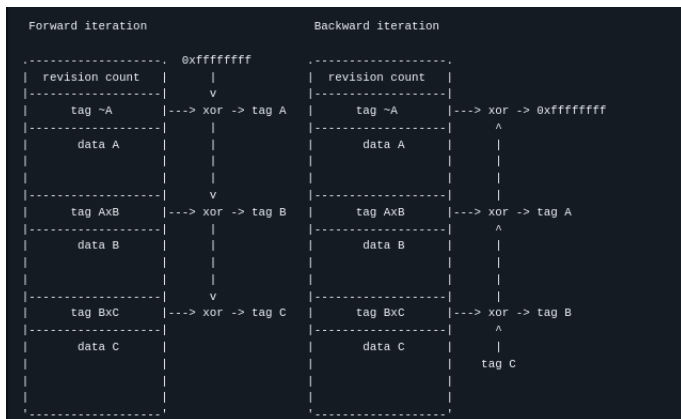
Figure 33. Forward/Backward Traversal XOR Tags.

- Every tag (which describes a metadata entry) is XORed with its previous tag before being stored.
- The first tag is XORed with 0xffffffff.
- To decode during iteration:
    - Forward traversal: Apply XOR cumulatively with the previous raw tag.
    - Backward traversal: Start from the end and reverse XOR step-by-step to recover the original tags.

4.2.3. *Valid Bit and Commit Finalization*
What's a "valid bit"?
Every metadata tag in littlefs contains one special bit called the valid bit.

- If this bit is 0 → the tag is valid
- If this bit is 1 → the tag is invalid (maybe due to a crash during writing)
- This bit helps littlefs detect corrupted or incomplete entries, especially after a sudden power loss.

What's happening at the end of a commit? After a group of entries (a "commit") is written, littlefs wants to invalidate any next entries that weren't fully written.
At the end of a commit, littlefs writes a special CRC tag.

- This CRC tag includes:
    - The checksum of the commit (to detect errors)
    - A type field
    - littlefs XORs the previous tag's valid bit with the lowest bit of the CRC tag's type.
    - This final step forces the next tag to have an invalid valid bit...

Why isn't this enough?
Because of how flash memory works:

- You can set bits from $1 \rightarrow 0$, but not back from $0 \rightarrow 1$ without erasing the block.
- If power cuts during writing, it may partially write the next tag but the valid bit might accidentally still be 0.
- That's why the valid bit is just the first check, littlefs also verifies the CRC of each commit. If that fails, the commit is considered invalid.

4.3. Metadata tags

Every metadata entry in LittleFS — including file records, directory references, and internal control fields is identified by a compact 32-bit tag.
NOTE: on endianness: Tags are stored in big-endian, unlike most of LittleFS which uses little-endian. This quirk ensures the valid bit is positioned at the start of a commit for detection during scanning.
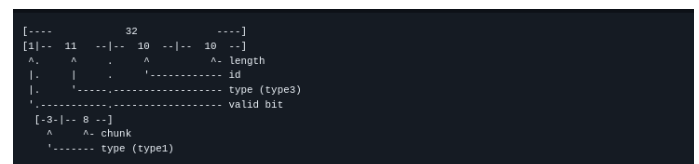
Figure 34. A tag structure.

Here's what the tag encodes:

- **Valid bit (1 bit):** Used to detect whether the entry was successfully committed. If unset, the entry is ignored by LittleFS but can be valuable during recovery.
- **Type (11 bits)** broken into:
    - **Type1 (3 bits):** A high-level category (e.g., file, directory, global).
    - **Chunk (8 bits):** additional tag-specific info used for matching with ID.
- **ID (10 bits):** Unique identifier for a file within a metadata block. The special ID 0x3FF marks entries not linked to a specific file (e.g., directory metadata).
- **Length (10 bits):** Specifies the size of the associated data. The value 0x3FF means deleted entry crucial for file recovery.

4.4. Metadata types

n LittleFS, each file is internally represented using multiple metadata entries, all grouped by a shared file id. These entries are stored as tags, each encoding a specific operation or attribute.

4.4.1. `0x401` *LFS_TYPE_CREATE*

Marks the creation of a new file (with a unique ID). While not mandatory for all files, this tag helps maintain filename ordering in the directory. It effectively "inserts" a file into the directory's internal structure.

4.4.2. `0x4FF` *LFS_TYPE_DELETE*

Marks the deletion of a file associated with a given ID. When present, this tag indicates that a file has been logically removed. However, the associated data and metadata may still reside on disk and can potentially be recovered.

4.4.3. `0x0xx` *LFS_TYPE_NAME*

This tag assigns a human-readable name and a file type to a specific file ID. It is the foundational tag for any file and must always be written first.

- **Data:**ASCII-encoded filename and an 8-bit file type (e.g., regular file, directory).

4.4.4. `0x001` *LFS_TYPE_REG*

- This marks the file as a regular file.
- The file's data blocks are organized using a CTZ skip-list, and the pointer to that structure is found in a separate struct tag.

4.4.5. `0x002` *LFS_TYPE_DIR*

- This marks the entry as a directory.
- The directory is implemented as a linked list of metadata pairs, with each pair containing a group of files, ordered alphabetically.
- The starting point of the directory is referenced in the corresponding struct tag.

4.4.6. `0x0ff` *LFS_TYPE_SUPERBLOCK*

In most filesystems, the superblock is a special structure that stores critical information about the filesystem's configuration and version. In littlefs, the same idea is preserved, but with a twist: the superblock is not stored in a dedicated block. Instead, it's implemented as a regular file entry (id = 0) that is replicated across the first few metadata pairs (starting at blocks 0 and 1).

This design improves resilience, as every time the root directory is compacted, the superblock is extended to another pair, creating a chain that exponentially extends the lifetime of the flash. It is composed of two tags:

- **name tag** with the type LFS_TYPE_SUPERBLOCK (0x0ff) that stores the ASCII string "littlefs", identifying the filesystem.
- **inline-struct tag)** LFS_TYPE_INLINESTRUCT (0x201that stores:

    - **Version:**of littlefs (e.g., 0x00020000 for version 2.0)
    - Block size (e.g., 4096 bytes)
    - **Block count** (total number of blocks in storage)
    - **Name max:** maximum filename length
    - **File max:** maximum file size supported
    - **Attr max:** max size of file attributes

The superblock is always:

- Stored at id = 0
- The first entry in the metadata block
- Starts exactly at offset 8, so the string "littlefs" is easy to locate and verify

To recover or scan a littlefs image:

- Look for the magic string "littlefs" at offset 0x08 in early blocks (0 and 1)
- Validate using the inline struct and version
- From here, you can extract the block size and count, which define the layout for reading the rest of the filesystem

4.4.7. `0x200` *LFS_TYPE_DIRSTRUCT*

Is used to associate a file or directory (via its ID) with a specific on-disk data structure, such as a CTZ skip-list or an inline-struct. This tag is essential in determining how the file's content is stored and how it should be read or traversed:

- The chunk field in the tag determines the type of structure used (e.g., CTZ skip-list or inline).

Importantly, only one struct can be active per file ID. Appending a new struct tag for the same ID overrides the previous structure.

4.4.8. `0x200` *LFS_TYPE_DIRSTRUCT*

Points to the first metadata pair in a directory. The rest of the directory is represented by a tail-linked list of metadata pairs. This allows iterating through directory entries without needing all data in RAM.

4.4.9. `0x201` *LFS_TYPE_INLINESTRUCT*

tells littlefs that the file's content is stored directly inside the metadata pair, no external data blocks needed.

LFS_TYPE_INLINESTRUCT Tag Layout

Valid Bit	Type (11 bits)	ID (10 bits)	Size (10 bits)
32-bit Tag			

$$\Downarrow$$

Inline Data (variable length, as per size)

- type = 0x201 tells littlefs this is an inline file.
- size gives the number of bytes stored.
- The data follows immediately, stored in-place.

4.4.10. `0x202` *LFS_TYPE_CTZSTRUCT*

This metadata type associates an id with a file stored as a CTZ skip-list. CTZ (Count Trailing Zeros) skip-lists are used to efficiently manage large files that cannot be inlined into a metadata pair.

A CTZ-struct tag contains:

- **File head (32 bits):**Block pointer to the head of the skip-list (i.e., the newest block).
- **File size (32 bits):** Total size of the file in bytes.

5. Forensics Tool

5.1. Objective of the tool

5.2. Step 1: Creating a Sample LittleFS Image

To begin testing and validating the forensic tool, I generated a LittleFS image using the littlefs-python library. The goal of this step is to simulate a realistic embedded filesystem scenario containing various files and directories, including one file that will later be deleted to enable recovery testing.

The filesystem image was configured with:

- **Block size:** 512 bytes
- **Block count:** 256
- This results in a total image size of 128 KiB.

```
(venv) (base) emna@emna-ASUS-EXPERTBOOK-B1500CEAEY-B1500CEAE:~/littlefs-forensics-tool/samples$ ls -lh
total 132K
-rwxrwxr-x 1 emna emna 1,3K mai   12 18:35 create_image.py
-rw-rw-r-- 1 emna emna 128K mai   12 18:36 FlashMemory.bin
```

Figure 35. Contents of the samples/ directory after generating the image..

After formatting and mounting the filesystem, I populated it with a variety of files structured as follows:

```
├── first-file.txt
├── config/
│   ├── system.conf
│   └── network.conf
├── logs/
│   └── boot.log
└── temp/
    └── to-be-deleted.txt   ← will be deleted later
```

Figure 36. Directory structure of the sample filesystem image.

5.3. Step2: Listing Files and Inspecting Contents

o perform a forensic inspection of the LittleFS image, I developed a Python script called forensic_analysis_littlefs.py. This tool mounts the raw image in memory, traverses the directory structure recursively, and prints a tree representation of all files and directories. Additionally, an optional flag allows it to dump the contents of all present files.

The tool accepts the following arguments:

- A path to the LittleFS image
- An optional -c (–contents) flag to print the content of each file.
- n optional -b flag to set the block size (default is 512 bytes).

- Example usage:

```
1 python3 forensic_analysis_littlefs.py
      image_path
2 python3 forensic_analysis_littlefs.py
      image_path -c
```

- Output (with -c flag):

```
(venv) (base) emna@emna-ASUS-EXPERTBOOK-B1500CEAEY-B1500CEAE:~/littlefs-forensics-tool/tool$ python3
 forensic_analysis_littlefs.py /home/emna/littlefs-forensics-tool/samples/FlashMemory.bin -c
Mounted '/home/emna/littlefs-forensics-tool/samples/FlashMemory.bin' (128 KiB, 256 blocks)
/
├── config/
│   ├── network.conf
│   └── system.conf
├── first-file.txt
├── logs/
│   └── boot.log
└── temp/

--- /first-file.txt  (22 bytes) ---
This is the root file

--- /config/network.conf  (34 bytes) ---
ip=192.168.1.1
mask=255.255.255.0

--- /config/system.conf  (24 bytes) ---
system=true
version=2.0

--- /logs/boot.log  (27 bytes) ---
Boot successful at 12:34PM
```

Figure 37. Output (with -c flag).

This confirms the correct file structure and content preservation within the LittleFS image. The temp/ directory appears empty as the file inside it was deleted.

5.4. Step3: The core data structures within the LittleFS image.

After successfully listing the visible contents of the filesystem and inspecting file data, I began investigating the core data structures within the LittleFS image.

5.4.1. SuperBlock Inspection

In filesystem design, the superblock is a critical structure that stores global metadata about the filesystem :its layout, geometry, and configuration. In LittleFS, the superblock serves a similar purpose but is implemented using the same flexible, copy-on-write and power-loss-resilient structures as everything else in the filesystem.

Where Is the Superblock Stored?

- LittleFS stores two copies of the superblock for safety ,typically in block 0 and block 1.
- At mount time, the filesystem chooses the more recent copy by comparing their revision numbers
- This design ensures the system can recover safely even if power is lost while writing the superblock.

What's Inside the Superblock? In the super-block there are two distinct directory-entry tags one after the other :

```
| CRC | TAG 0x0ff (name-tag) | "littlefs" |
| TAG 0x201 (inline-struct) |
| version | block_size | block_count | name_max | file_max | attr_max |
```

Figure 38. Superblock structure.

- **Name-tag (type 0x0ff)** : contains only the 8-byte magic string.
- **Inline-struct tag (type 0x201)** : starts with its own 32-bit tag header, followed by the 24 bytes that make up the real superblock fields.

Using a custom script (superblock.py), I printed the super-block fields. The script accepts the following arguments:

- A path to the LittleFS image

- Example usage:

```
1 python3 superblock.py image_path
```

```
(venv) (base) emna@emna-ASUS-EXPERTBOOK-B1500CEAEY-B1500CEAE:~/littlefs-forensic
s-tool/tool$ python3 superblock.py ../samples/FlashMemory.bin
[SUPERBLOCK @ block 0]
  Magic       : littlefs
  Version     : 2.1 (0x00020001)
  Block Size  : 512
  Block Count : 256
  Name Max    : 255
```

Figure 39. Superblock config fields.

This confirms that the superblock was located at block 0, and that the image adheres to the LittleFS version 2.1 specification. The geometry parameters (block size and block count) match the ones used during image creation.

5.4.2. Root directory in littlefs v2.1

In the on-disk v2.x format we're looking at (version = 2.1), the super-block is already the first (and only) metadata-pair of the root directory. LittleFS no longer stores a separate "root-dir pointer"; instead, the super-block's two tags (0x0ff name + 0x201 inline-struct) reside at the beginning of the root directory's metadata-pair. After them, the filesystem continues with any files or directories stored at the root (/), encoded as regular metadata entries (e.g., LFS_TYPE_NAME, LFS_TYPE_STRUCT). So, when you finish parsing the inline-struct tag:

```
pair 0

┌──────────────┬──────────────────────────────────────────┐
| offset 0     | revision counter                         |
| offset 4     | XOR-pad (0xFFFFFFFF)                      |
| offset 8     | TAG 0x0ff00008  ← "littlefs" name-tag     |
| offset 0x14  | TAG 0x20100018  ← inline-struct tag       |
| offset 0x18  |              24-byte payload              |
| offset 0x2c  | TAG …           ← first *real* root entry |
| …            | …                                        |
```

Figure 40. root directory.

- If the image was just formatted, the next thing you'll encounter is the commit's CRC and you're done.
- If the root already contains files, you'll see the usual LFS_TYPE_NAME/LFS_TYPE_STRUCT pairs (ids 1) before the CRC.
- If the directory has spilled into more pairs, you'll also see a hard-tail tag (0x601) pointing to the next pair.

As a first step, I wanted to extract the first actual root directory entry to confirm that I was using the correct offset and parsing logic. As a re-minder, every tag in a LittleFS metadata stream is XOR-chained with the tag that came just before it, we must feed the decoder the value of the previous decoded tag every time we step to a new one,starting with '0xffffffff'. For the very first tag we want to analyse,, the "previous" tag is that inline-struct tag.

```
offset        bytes                 meaning
0x08-0x0F     "littlefs"            ← NAME tag  (id 0, chunk 0xFF)
0x14-0x17     0x20 10 00 18         ← inline-struct TAG WORD  ↔ we need this
0x18-0x2B     24-byte payload       (version, block_size, …)
0x2C-…        first real root entry
```

Figure 41. xoring tags.

Tag bit-field layout (after de-XORing):

```
31         30 …28  27 …20   19 …10     9 …0
┌invalid┐ ┌ type1 ┐ ┌ chunk ┐ ┌  id  ┐ ┌ length ┐
    0        010    00000001  0000000000  0000011000
```

Figure 42. second tag of the superblock(inline struct).

Field	Value	Explanation
invalid	0	Tag is valid (0 = valid, 1 = invalid)
type1	2	Type 2 = STRUCT tag
chunk	0x01	Chunk 1 inline struct
id	0x000	ID 0: tag belongs to the superblock
length	0x018 (24)	Payload is 6 words = 24 bytes

Table 2. Breakdown of a LittleFS inline-struct tag for the superblock

Now let's write those bits back into one 32-bit word (big-endian):

```
type1 (3 bits)  = 010     → 0x2
chunk (8 bits)  = 0x01    → 0x01
id    (10 bits) = 0x000   → 0x000
length(10 bits) = 0x018   → 0x018
-----------------------------------
hex word                  0x20100018
```

Figure 43. The decoded value of the inline-struct tag is 0x20100018.

we start reading at 0x2C, why ?

```
offset    contains
0x00-03   revision counter   (ignored for a static analysis)
0x04-07   XOR-pad word       (all 0xFF if this is the first tag)
0x08-17   NAME tag   "littlefs" (id 0, chunk 0xFF)
0x18-1B   STRUCT tag 0x20100018 (id 0, chunk 1, len 24)
0x1C-2B   24-byte payload (version, block_size, …)
0x2C-…    ► first tag of the real root directory ◄
```

Figure 44. FIRST_TAG = 0x2C.

Decoding the first post-super-block tag

```
# inline-struct payload starts at 0x14+4 = 0x18 and is 24 bytes long
FIRST_TAG_OFF = 0x2c                   # 0x14 + 4 + 24
raw       = be32(buf, FIRST_TAG_OFF)
prev_dec  = 0x20100018                  # decoded inline-struct tag
tag_word  = raw ^ prev_dec
info      = decode(tag_word)
```

Figure 45. Decoding the first post-super-block tag.

Result:

```
(venv) (base) emna@emna-ASUS-EXPERTBOOK-B1500CEAEY-B1500CEAE:~/litt
python3 helper2.py ../samples/FlashMemory.bin
First NAME tag after the super-block
  invalid: 0
  type1  : 0
  chunk  : 1
  id     : 2
  length : 14

Raw filename bytes (hex): 66697273742d66696c652e747874
ASCII filename          : first-file.txt
```

Figure 46. Result.

- **type1 = 0**: it's a name tag
- **chunk = 1**: a regular file.
- **id = 2**:this file's unique object-ID.
- **length = 14**: this file's unique object-ID.

As a next step, I generated a hex dump of the different blocks, I wanted to inspect raw bytes and locate known structures such as the magic string "littlefs" and surrounding metadata tags. I wanted to understand more about the block layout. For example, the figure below shows the raw dump of block 0:

```
00000000  06 00 00 00 f0 0f ff f7  6c 69 74 74 6c 65 66 73  |.......littlefs|
00000010  2f e0 00 10 01 00 02 00  00 02 00 00 00 01 00 00  |/..............|
00000020  ff 00 00 00 ff ff ff 7f  fe 03 00 00 20 00 08 16  |............ ...|
00000030  66 69 72 73 74 2d 66 69  6c 65 2e 74 78 74 20 00  |first-file.txt .|
00000040  00 18 54 68 69 73 20 69  73 20 74 68 65 20 72 6f  |..This is the ro|
00000050  6f 74 20 66 69 6c 65 0a  20 30 0c 10 63 6f 6e 66  |ot file. 0..conf|
00000060  69 67 20 20 00 0e c6 00  00 00 c7 00 00 00 20 20  |ig ..........  |
00000070  08 0c 6c 6f 67 73 20 20  00 0c c8 00 00 00 c9 00  |..logs ........|
00000080  00 00 20 20 1c 0c 74 65  6d 70 20 20 00 0c ca 00  |..  ..temp ....|
00000090  00 00 cb 00 00 00 40 0f  ec 00 ca 00 00 00 cb 00  |......@........|
000000a0  00 00 30 00 01 52 13 ef  cd 48 ff ff ff ff ff ff  |..0..R...H......|
000000b0  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
```

Figure 47. Raw dump of block 0.

- **Revision word (offset 0x0000) 06 00 00 00** : :Little-endian u32 revision = 6.
- **Tag 0 (offset 0x0004):**
  - **stored word** = 0xF00FFFF7
  - **previous-decoded** = 0xFFFFFFFF
  - **decoded=stored xor prev** = 0x0FF00008
    * type1 = 0 → NAME
    * chunk = 0xFF → super-block magic
    * id= 0
    * len = 8
    * next tag starts at offset 0x0004 + 12(4 the tag + 8 it's data) = 0x0010
  - **prev-decoded** = 0x0FF00008
- **Tag 1 (offset 0x0010)**
  - **stored** = 0x2FE00010
  - **prev-decoded** = 0x0FF00008
  - **decoded = stored xor prev-decoded** = 0x20100018
    * type1= 2 → STRUCT
    * chunk = 0x01 → inline-struct
    * id = 0
    * len = 24
    * next tag at 0x0010 + 28(4 the tag + 24 it's data) = 0x002C
- **Tag 2 (offset 0x002C):**
  - **stored** = 0x20000816
  - **decoded= stored xor prev** = 0x0010080E
    * type1 = 0 → NAME
    * chunk = 0x01 → regular file
    * id = 2
    * len = 0x0E (14)
    * Payload = "first-file.txt"
    * next tag at 0x002C + 20(4 the tag+14 it's data +2 padding for block allignement) = 0x0040
  - **prev-decoded**=0x0010080E
- **Tag 3 (offset 0x0040)**
  - **stored** = 0x00185468
  - **decoded = stored xor prev** = 0x00085C66
    * type1 = 0 → NAME
    * chunk = 0x00 → **invalid / reserved**
    * id = 535
    * len = 0x66 (102)
  - That's weird.. I was expecting an inline struct that contains the payload of the first file.txt. I can see it as "This is the root file," but it's marked as a name tag! It looks like garbage; I didn't find a CRC tag in this block, so it looks like an incomplete commit.

```
*
00019400  02 00 00 00 ff ef ff ee  74 6f 2d 62 65 2d 64 65  |........to-be-de|
00019410  6c 65 74 65 64 2e 74 78  74 20 00 00 0b 54 68 69  |leted.txt ...Thi|
00019420  73 20 66 69 6c 65 20 77  69 6c 6c 20 62 65 20 64  |s file will be d|
00019430  65 6c 65 74 65 64 0a 40  1f fc 12 c8 00 00 00 c9  |eleted.@........|
00019440  00 00 00 30 00 01 b1 de  8f 9b bd ff ff ff ff ff  |...0............|
00019450  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
*
```

Figure 48. Hex dump.

I can see the file that I deleted while creating the image, along with it's payload.I will try to recover it later.

After inspecting the raw hex dump of the filesystem, I wanted to extract meaningful metadata entries in a more structured way. To do this, I wrote a custom Python script scan.py that iterates over each block and parses the metadata tags encoded in LittleFS.I dumped the output of the script in **data_struct_dump.txt** file under the tool directory

```
=== BLOCK 0 (super-block) ===
[blk   0] +0004: NAME   id=  0 len=  8 chk=0xFF
          └ littlefs
[blk   0] +0010: STRUCT id=  0 len= 24 chk=0x01
          └ inline head=0x00020001 size=512
[blk   0] +002C: NAME   id=  2 len= 14 chk=0x01
          └ first-file.txt
[blk   0] +0040: NAME   id=535 len=102 chk=0x00
          └ is is the root file
0
 config  ◆◆
        logs
            ◆◆
              temp
                 ◆◆@◆◆◆0R◆◆H
```

Figure 49. example of output.

- The output produced by my custom script closely matched what I had previously observed in the hex dump.
- **Block 0**
  - Super-block NAME + STRUCT (the "littlefs" magic and inline geometry).
  - A good first-file.txt entry.
  - the half written name (id 535, chunk 0x00) and padding.
  - No CRC

```
                  system.com
[blk 199] +0004: NAME   id=  1 len= 11 chk=0x01
          └ system.conf
[blk 200] +0004: NAME   id=  0 len=  8 chk=0x01
          └ boot.log
[blk 200] +0010: STRUCT id=  0 len= 27 chk=0x01
          └ inline head=0x746F6F42 size=1668641568
[blk 201] +0004: NAME   id=  0 len=  8 chk=0x01
          └ boot.log
[blk 201] +0010: STRUCT id=  0 len=  0 chk=0x01
[blk 201] +0014: TAIL   id=1023 len=  8 chk=0x00
[blk 201] +0020: CRC    id=1023 len=476 chk=0x00
```

Figure 50. example of output.

- **Blocks 200 – 201**
  - Normal NAME/STRUCT pairs for system.conf, boot.log.
  - A TAIL tag (TYPE 6, chunk 0x00, len 8).
  - Immediately after that, a CRC tag (TYPE 5), which terminates the commit.

```
[blk 202] +0004: NAME   id=  0 len= 17 chk=0x01
          └ to-be-deleted.txt
[blk 202] +001C: NAME   id=282 len=120 chk=0xB4
          └ s file will be deleted
@◆◆◆0◆c◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
```

Figure 51. example of output.

- **Blocks 202 – 203**
  - the orphan NAME for to-be-deleted.txt and it's payload.

After dumping and analyzing the core metadata structures from the image, I noticed that some parts matched expectations, while others raised questions...

- **Missing Directory Name Tags:** I expected to find NAME tags with type1 = 0 and chk values identifying directories (typically chunk = 0x02 for LFS_TYPE_DIR), but they were absent. I found no explicit name tag entries corresponding to the /config, /logs, or /temp directories. This was suspicious.
- **Directory Names in Raw Data:** Although I did not find structured metadata entries for the directories, I did spot their names as raw strings in the dump (in block 0), which suggests they were written at some point.possibly an offset misalignment in my parsing logic or the way I created the image?

5.5. Step3: recovering the deleted file

Despite some ambiguities, I decided to try and recover the deleted file. For that, I developed a custom Python script (recover.py) to recover the contents of a deleted inline file, such as /temp/to-be-deleted.txt. The script takes the target file name as a command-line argument and scans each block of the image for a NAME tag whose payload matches the specified filename. Once found, it calculates the start of the inline payload (aligned to a 4-byte boundary) and extracts the data up to the first padding byte (0xFF). This approach allows recovery even when the commit lacks a valid CRC and would normally be ignored by the filesystem.

Figure 52. to-be-deleted.txt recovery.

Figure 53. recovered file content.

Observations Limitations

- the recovered file contains the actual payload , but also contains some trailing noise likely leftover or misaligned bytes which appear as unreadable characters in the dump.
- , the script only supports recovering inline files. It does not handle the recovery of files stored using the CTZ skip-list format.

■ References

- **LittleFS official github**

  - **README.md**
  - **LittleFS Header (lfs.h):** Public C API and internal tag definitions for LittleFS.

  - **LittleFS DESIGN.md:** Internal design notes for LittleFS
  - **LittleFS SPEC.md:** technical specification of the FS.

  https://github.com/littlefs-project/littlefs
- **Project Repository – LittleFS Forensics Tool**
  project_github_link