# MAX3267X LittleFS demo

Generated by Doxygen 1.8.11

# Contents

# Chapter 1

# Description

Demonstrates LittleFS usage on MAX32670 MCU.

MCU inernal flash is partitioned as follows:

- Application code area: 64kb (Flash memory pages 0 - 7)
- Flash storage area: 64kb (Flash memory pages 8 - 15)

The application code area should be defined in the linker script file ∗"max32670.ld"∗:

```
MEMORY {
    FLASH (rx) : ORIGIN = 0x10000000, LENGTH = 64K /* 64kB "FLASH" */
    SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 160K /* 160kB SRAM */
}
```

The internal storage flash memory block count is specified by *FLASH_STORAGE_PAGE_CNT* macro.

```
#define FLASH_STORAGE_PAGE_CNT 8
```

that corresponds to 64kb (8 of 8kb blocks)

### Required Connections

- Connect a USB cable between the PC and the CN1 (USB/PWR) connector.
- Select RX0 and TX0 on Headers JP1 and JP3 (UART 0).
- Open an terminal application on the PC and connect to the EV kit's console UART at 115200, 8-N-1.

### Expected Output

The Console UART of the device will output these messages:

```
***** Flash Control Example *****
Filesystem is mounted
boot_count: 12

Example Succeeded
```

# Chapter 2

# The design of littlefs

A little fail-safe filesystem designed for microcontrollers.

```
   | | |     .---._____
  .-----.   |          |
--|o    |---| littlefs |
--|     |---|          |
  '-----'   '----------'
   | | |
```

littlefs was originally built as an experiment to learn about filesystem design in the context of microcontrollers. The question was: How would you build a filesystem that is resilient to power-loss and flash wear without using unbounded memory?

This document covers the high-level design of littlefs, how it is different than other filesystems, and the design decisions that got us here. For the low-level details covering every bit on disk, check out SPEC.md.

## The problem

The embedded systems littlefs targets are usually 32-bit microcontrollers with around 32 KiB of RAM and 512 KiB of ROM. These are often paired with SPI NOR flash chips with about 4 MiB of flash storage. These devices are too small for Linux and most existing filesystems, requiring code written specifically with size in mind.

Flash itself is an interesting piece of technology with its own quirks and nuance. Unlike other forms of storage, writing to flash requires two operations: erasing and programming. Programming (setting bits to 0) is relatively cheap and can be very granular. Erasing however (setting bits to 1), requires an expensive and destructive operation which gives flash its name. `Wikipedia` has more information on how exactly flash works.

To make the situation more annoying, it's very common for these embedded systems to lose power at any time. Usually, microcontroller code is simple and reactive, with no concept of a shutdown routine. This presents a big challenge for persistent storage, where an unlucky power loss can corrupt the storage and leave a device unrecoverable.

This leaves us with three major requirements for an embedded filesystem.

1. **Power-loss resilience** - On these systems, power can be lost at any time. If a power loss corrupts any persistent data structures, this can cause the device to become unrecoverable. An embedded filesystem must be designed to recover from a power loss during any write operation.

1. **Wear leveling** - Writing to flash is destructive. If a filesystem repeatedly writes to the same block, eventually that block will wear out. Filesystems that don't take wear into account can easily burn through blocks used to store frequently updated metadata and cause a device's early death.

1. **Bounded RAM/ROM** - If the above requirements weren't enough, these systems also have very limited amounts of memory. This prevents many existing filesystem designs, which can lean on relatively large amounts of RAM to temporarily store filesystem metadata.

   For ROM, this means we need to keep our design simple and reuse code paths were possible. For RAM we have a stronger requirement, all RAM usage is bounded. This means RAM usage does not grow as the filesystem changes in size or number of files. This creates a unique challenge as even presumably simple operations, such as traversing the filesystem, become surprisingly difficult.

## Existing designs?

So, what's already out there? There are, of course, many different filesystems, however they often share and borrow feature from each other. If we look at power-loss resilience and wear leveling, we can narrow these down to a handful of designs.

1. First we have the non-resilient, block based filesystems, such as `FAT` and `ext2`. These are the earliest filesystem designs and often the most simple. Here storage is divided into blocks, with each file being stored in a collection of blocks. Without modifications, these filesystems are not power-loss resilient, so updating a file is a simple as rewriting the blocks in place.

```
            .--------.
            |  root  |
            |        |
            |        |
            '--------'
            .-'    '-.
           v          v
      .--------.  .--------.
      |    A   |  |    B   |
      |        |  |        |
      |        |  |        |
      '--------'  '--------'
      .-'         .-'    '-.
     v           v          v
.--------.  .--------.  .--------.
|    C   |  |    D   |  |    E   |
|        |  |        |  |        |
|        |  |        |  |        |
'--------'  '--------'  '--------'
```

Because of their simplicity, these filesystems are usually both the fastest and smallest. However the lack of power resilience is not great, and the binding relationship of storage location and data removes the filesystem's ability to manage wear.

1. In a completely different direction, we have logging filesystems, such as `JFFS`, `YAFFS`, and `SPIFFS`, storage location is not bound to a piece of data, instead the entire storage is used for a circular log which is appended with every change made to the filesystem. Writing appends new changes, while reading requires traversing the log to reconstruct a file. Some logging filesystems cache files to avoid the read cost, but this comes at a tradeoff of RAM.

```
                                                          v
.--------.--------.--------.--------.--------.--------.--------.--------.
|        C        | new B  | new A  |                 |   A    |   B    |
|                 |        |        |->               |        |        |
|                 |        |        |                 |        |        |
'--------'--------'--------'--------'--------'--------'--------'--------'
```

Logging filesystem are beautifully elegant. With a checksum, we can easily detect power-loss and fall back to the previous state by ignoring failed appends. And if that wasn't good enough, their cyclic nature means that logging filesystems distribute wear across storage perfectly.

The main downside is performance. If we look at garbage collection, the process of cleaning up outdated data from the end of the log, I've yet to see a pure logging filesystem that does not have one of these two costs:

1. *O(n²)* runtime
2. *O(n)* RAM

SPIFFS is a very interesting case here, as it uses the fact that repeated programs to NOR flash is both atomic and masking. This is a very neat solution, however it limits the type of storage you can support.

1. Perhaps the most common type of filesystem, a journaling filesystem is the offspring that happens when you mate a block based filesystem with a logging filesystem. `ext4` and `NTFS` are good examples. Here, we take a normal block based filesystem and add a bounded log where we note every change before it occurs.

```
                          journal
                         .--------.--------.
            .--------.   | C'| D'|     | E'|
            |  root  |-->|   |   |->   |   |
            |        |   |   |   |     |   |
            |        |   '--------'--------'
            '--------'
            .-'    '-.
           v          v
      .--------.  .--------.
      |   A    |  |   B    |
      |        |  |        |
      |        |  |        |
      '--------'  '--------'
      .-'         .-'    '-.
     v           v          v
.--------.  .--------.  .--------.
|   C    |  |   D    |  |   E    |
|        |  |        |  |        |
|        |  |        |  |        |
'--------'  '--------'  '--------'
```

This sort of filesystem takes the best from both worlds. Performance can be as fast as a block based filesystem (though updating the journal does have a small cost), and atomic updates to the journal allow the filesystem to recover in the event of a power loss.

Unfortunately, journaling filesystems have a couple of problems. They are fairly complex, since there are effectively two filesystems running in parallel, which comes with a code size cost. They also offer no protection against wear because of the strong relationship between storage location and data.

1. Last but not least we have copy-on-write (COW) filesystems, such as `btrfs` and `ZFS`. These are very similar to other block based filesystems, but instead of updating block inplace, all updates are performed by creating a copy with the changes and replacing any references to the old block with our new block. This recursively pushes all of our problems upwards until we reach the root of our filesystem, which is often stored in a very small log.

```
            .--------.                  .--------.
            |  root  |       write      |new root|
            |        |        ==>       |        |
            |        |                  |        |
            '--------'                  '--------'
            .-'    '-.                    |    '-.
           |  .-------|------------------'       v
           v v        v                      .--------.
      .--------.  .--------.                 | new B  |
      |   A    |  |   B    |                  |        |
```

```
      |        |  |        |                  |        |
      |        |  |        |                  '--------'
      '--------'  '--------'                  .-'    |
      .-'         .-'    '-.    .------------|------'
     |           |          |  |             v
     v           v          v  v        .--------.
.--------.  .--------.  .--------.      | new D  |
|   C    |  |   D    |  |   E    |      |        |
|        |  |        |  |        |      |        |
|        |  |        |  |        |      '--------'
'--------'  '--------'  '--------'
```

COW filesystems are interesting. They offer very similar performance to block based filesystems while managing to pull off atomic updates without storing data changes directly in a log. They even disassociate the storage location of data, which creates an opportunity for wear leveling.

Well, almost. The unbounded upwards movement of updates causes some problems. Because updates to a COW filesystem don't stop until they've reached the root, an update can cascade into a larger set of writes than would be needed for the original data. On top of this, the upward motion focuses these writes into the block, which can wear out much earlier than the rest of the filesystem.

## littlefs

So what does littlefs do?

If we look at existing filesystems, there are two interesting design patterns that stand out, but each have their own set of problems. Logging, which provides independent atomicity, has poor runtime performance. And COW data structures, which perform well, push the atomicity problem upwards.

Can we work around these limitations?

Consider logging. It has either a *$O(n^2)$* runtime or *$O(n)$* RAM cost. We can't avoid these costs, *but* if we put an upper bound on the size we can at least prevent the theoretical cost from becoming problem. This relies on the super secret computer science hack where you can pretend any algorithmic complexity is *$O(1)$* by bounding the input.

In the case of COW data structures, we can try twisting the definition a bit. Let's say that our COW structure doesn't copy after a single write, but instead copies after *$n$* writes. This doesn't change most COW properties (assuming you can write atomically!), but what it does do is prevent the upward motion of wear. This sort of copy-on-bounded-writes (CObW) still focuses wear, but at each level we divide the propagation of wear by *$n$*. With a sufficiently large *$n$* ($>$ branching factor) wear propagation is no longer a problem.

See where this is going? Separate, logging and COW are imperfect solutions and have weaknesses that limit their usefulness. But if we merge the two they can mutually solve each other's limitations.

This is the idea behind littlefs. At the sub-block level, littlefs is built out of small, two block logs that provide atomic updates to metadata anywhere on the filesystem. At the super-block level, littlefs is a CObW tree of blocks that can be evicted on demand.

```
                   root
                  .--------.--------.
                  | A'| B'|         |
                  |   |   |->       |
                  |   |   |         |
                  '--------'--------'
               .----'   '--------------.
      A       v                 B       v
     .--------.--------.       .--------.--------.
      | C'| D'|         |       | E'|new|         |
      |   |   |->       |       |   | E'|->       |
      |   |   |         |       |   |   |         |
      '--------'--------'       '--------'--------'
      .-'   '--.                  |   '------------------.
     v          v              .-'                        v
.--------.  .--------.        v                       .--------.
```

```
|    C    | |    D    |   .--------.        write        | new E  |
|         | |         |   |   E    |         ==>         |        |
|         | |         |   |        |                     |        |
'---------' '---------'   |        |                     '--------'
                          '--------'                     .-'    |
                          .-'    '-.    .-------------|------'
                         v          v  v                 v
                     .--------.  .--------.        .--------.
                     |   F    |  |   G    |        | new F  |
                     |        |  |        |        |        |
                     |        |  |        |        |        |
                     '--------'  '--------'        '--------'
```

There are still some minor issues. Small logs can be expensive in terms of storage, in the worst case a small log costs 4x the size of the original data. CObW structures require an efficient block allocator since allocation occurs every *n* writes. And there is still the challenge of keeping the RAM usage constant.

## Metadata pairs

Metadata pairs are the backbone of littlefs. These are small, two block logs that allow atomic updates anywhere in the filesystem.

Why two blocks? Well, logs work by appending entries to a circular buffer stored on disk. But remember that flash has limited write granularity. We can incrementally program new data onto erased blocks, but we need to erase a full block at a time. This means that in order for our circular buffer to work, we need more than one block.

We could make our logs larger than two blocks, but the next challenge is how do we store references to these logs? Because the blocks themselves are erased during writes, using a data structure to track these blocks is complicated. The simple solution here is to store a two block addresses for every metadata pair. This has the added advantage that we can change out blocks in the metadata pair independently, and we don't reduce our block granularity for other operations.

In order to determine which metadata block is the most recent, we store a revision count that we compare using `sequence arithmetic` (very handy for avoiding problems with integer overflow). Conveniently, this revision count also gives us a rough idea of how many erases have occurred on the block.

```
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        |
                            '----------------'----------------'
```

So how do we atomically update our metadata pairs? Atomicity (a type of power-loss resilience) requires two parts: redundancy and error detection. Error detection can be provided with a checksum, and in littlefs's case we use a 32-bit `CRC`. Maintaining redundancy, on the other hand, requires multiple stages.

1. If our block is not full and the program size is small enough to let us append more entries, we can simply append the entries to the log. Because we don't overwrite the original entries (remember rewriting flash requires an erase), we still have the original entries if we lose power during the append.

```
                                 commit A
.----------------.----------------.    .----------------.----------------.
|   revision 1   |   revision 0   | => |   revision 1   |   revision 0   |
|----------------|----------------|    |----------------|----------------|
|       |        |                |    |       A        |                |
|       v        |                |    |----------------|                |
|                |                |    |    checksum    |                |
|                |                |    |----------------|                |
|                |                |    |       |        |                |
|                |                |    |       v        |                |
|                |                |    |                |                |
|                |                |    |                |                |
|                |                |    |                |                |
|                |                |    |                |                |
'----------------'----------------'    '----------------'----------------'
```

Note that littlefs doesn't maintain a checksum for each entry. Many logging filesystems do this, but it limits what you can update in a single atomic operation. What we can do instead is group multiple entries into a commit that shares a single checksum. This lets us update multiple unrelated pieces of metadata as long as they reside on the same metadata pair.

```
                              commit B and A'
.----------------.----------------.    .----------------.----------------.
|   revision 1   |   revision 0   | => |   revision 1   |   revision 0   |
|----------------|----------------|    |----------------|----------------|
|       A        |                |    |       A        |                |
|----------------|                |    |----------------|                |
|    checksum    |                |    |    checksum    |                |
|----------------|                |    |----------------|                |
|       |        |                |    |       B        |                |
|       v        |                |    |----------------|                |
|                |                |    |       A'       |                |
|                |                |    |----------------|                |
|                |                |    |    checksum    |                |
|                |                |    |----------------|                |
'----------------'----------------'    '----------------'----------------'
```

1. If our block *is* full of entries, we need to somehow remove outdated entries to make space for new ones. This process is called garbage collection, but because littlefs has multiple garbage collectors, we also call this specific case compaction.

   Compared to other filesystems, littlefs's garbage collector is relatively simple. We want to avoid RAM consumption, so we use a sort of brute force solution where for each entry we check to see if a newer entry has been written. If the entry is the most recent we append it to our new block. This is where having two blocks becomes important, if we lose power we still have everything in our original block.

   During this compaction step we also erase the metadata block and increment the revision count. Because we can commit multiple entries at once, we can write all of these changes to the second block without worrying about power loss. It's only when the commit's checksum is written that the compacted entries and revision count become committed and readable.

```
                        commit B', need to compact
.----------------.----------------.    .----------------.----------------.
|   revision 1   |   revision 0   | => |   revision 1   |   revision 2   |
|----------------|----------------|    |----------------|----------------|
|       A        |                |    |       A        |       A'       |
|----------------|                |    |----------------|----------------|
|    checksum    |                |    |    checksum    |       B'       |
|----------------|                |    |----------------|----------------|
|       B        |                |    |       B        |    checksum    |
|----------------|                |    |----------------|----------------|
|       A'       |                |    |       A'       |       |        |
|----------------|                |    |----------------|       v        |
|    checksum    |                |    |    checksum    |                |
|----------------|                |    |----------------|                |
'----------------'----------------'    '----------------'----------------'
```

1. If our block is full of entries *and* we can't find any garbage, then what? At this point, most logging filesystems would return an error indicating no more space is available, but because we have small logs, overflowing a log isn't really an error condition.

   Instead, we split our original metadata pair into two metadata pairs, each containing half of the entries, connected by a tail pointer. Instead of increasing the size of the log and dealing with the scalability issues associated with larger logs, we form a linked list of small bounded logs. This is a tradeoff as this approach does use more storage space, but at the benefit of improved scalability.

   Despite writing to two metadata pairs, we can still maintain power resilience during this split step by first preparing the new metadata pair, and then inserting the tail pointer during the commit to the original metadata pair.

```
                    commit C and D, need to split
.----------------.----------------.    .----------------.----------------.
|   revision 1   |   revision 2   | => |   revision 3   |   revision 2   |
|----------------|----------------|    |----------------|----------------|
|       A        |       A'       |    |       A'       |       A'       |
|----------------|----------------|    |----------------|----------------|
|    checksum    |       B'       |    |       B'       |       B'       |
|----------------|----------------|    |----------------|----------------|
|       B        |    checksum    |    |      tail    ---------------------.
|----------------|----------------|    |----------------|----------------| |
|       A'       |        |       |    |    checksum    |                | |
|----------------|        v       |    |----------------|                | |
|    checksum    |                |    |        |       |                | |
|----------------|                |    |        v       |                | |
'----------------'----------------'    '----------------'----------------' |
                                                .----------------.---------'
                                               v                v
                                       .----------------.----------------.
                                       |   revision 1   |   revision 0   |
                                       |----------------|----------------|
                                       |       C        |                |
                                       |----------------|                |
                                       |       D        |                |
                                       |----------------|                |
                                       |    checksum    |                |
                                       |----------------|                |
                                       |        |       |                |
                                       |        v       |                |
                                       |                |                |
                                       |                |                |
                                       '----------------'----------------'
```

There is another complexity the crops up when dealing with small logs. The amortized runtime cost of garbage collection is not only dependent on its one time cost (*$O(n^2)$* for littlefs), but also depends on how often garbage collection occurs.

Consider two extremes:

1. Log is empty, garbage collection occurs once every *n* updates
2. Log is full, garbage collection occurs **every** update

Clearly we need to be more aggressive than waiting for our metadata pair to be full. As the metadata pair approaches fullness the frequency of compactions grows very rapidly.

Looking at the problem generically, consider a log with bytes for each entry, dynamic entries (entries that are outdated during garbage collection), and static entries (entries that need to be copied during garbage collection). If we look at the amortized runtime complexity of updating this log we get this formula:

If we let be the ratio of static space to the size of our log in bytes, we find an alternative representation of the number of static and dynamic entries:

Substituting these in for and gives us a nice formula for the cost of updating an entry given how full the log is:

Assuming 100 byte entries in a 4 KiB log, we can graph this using the entry size to find a multiplicative cost:

So at 50% usage, we're seeing an average of 2x cost per update, and at 75% usage, we're already at an average of 4x cost per update.

To avoid this exponential growth, instead of waiting for our metadata pair to be full, we split the metadata pair once we exceed 50% capacity. We do this lazily, waiting until we need to compact before checking if we fit in our 50% limit. This limits the overhead of garbage collection to 2x the runtime cost, giving us an amortized runtime complexity of $O(1)$.

If we look at metadata pairs and linked-lists of metadata pairs at a high level, they have fairly nice runtime costs. Assuming $n$ metadata pairs, each containing $m$ metadata entries, the *lookup* cost for a specific entry has a worst case runtime complexity of $O(nm)$. For *updating* a specific entry, the worst case complexity is $O(nm^2)$, with an amortized complexity of only $O(nm)$.

However, splitting at 50% capacity does mean that in the best case our metadata pairs will only be 1/2 full. If we include the overhead of the second block in our metadata pair, each metadata entry has an effective storage cost of 4x the original size. I imagine users would not be happy if they found that they can only use a quarter of their original storage. Metadata pairs provide a mechanism for performing atomic updates, but we need a separate mechanism for storing the bulk of our data.

## CTZ skip-lists

Metadata pairs provide efficient atomic updates but unfortunately have a large storage cost. But we can work around this storage cost by only using the metadata pairs to store references to more dense, copy-on-write (COW) data structures.

Copy-on-write data structures, also called purely functional data structures, are a category of data structures where the underlying elements are immutable. Making changes to the data requires creating new elements containing a copy of the updated data and replacing any references with references to the new elements. Generally, the performance of a COW data structure depends on how many old elements can be reused after replacing parts of the data.

littlefs has several requirements of its COW structures. They need to be efficient to read and write, but most frustrating, they need to be traversable with a constant amount of RAM. Notably this rules out B-trees, which can not be traversed with constant RAM, and B+-trees, which are not possible to update with COW operations.

So, what can we do? First let's consider storing files in a simple COW linked-list. Appending a block, which is the basis for writing files, means we have to update the last block to point to our new block. This requires a COW operation, which means we need to update the second-to-last block, and then the third-to-last, and so on until we've copied out the entire file.

```
A linked-list
.--------.  .--------.  .--------.  .--------.  .--------.  .--------.
| data 0 |->| data 1 |->| data 2 |->| data 4 |->| data 5 |->| data 6 |
|        |  |        |  |        |  |        |  |        |  |        |
|        |  |        |  |        |  |        |  |        |  |        |
'--------'  '--------'  '--------'  '--------'  '--------'  '--------'
```

To avoid a full copy during appends, we can store the data backwards. Appending blocks just requires adding the new block and no other blocks need to be updated. If we update a block in the middle, we still need to copy the following blocks, but can reuse any blocks before it. Since most file writes are linear, this design gambles that appends are the most common type of data update.

```
A backwards linked-list
.--------.  .--------.  .--------.  .--------.  .--------.  .--------.
| data 0 |<-| data 1 |<-| data 2 |<-| data 4 |<-| data 5 |<-| data 6 |
|        |  |        |  |        |  |        |  |        |  |        |
|        |  |        |  |        |  |        |  |        |  |        |
'--------'  '--------'  '--------'  '--------'  '--------'  '--------'
```

However, a backwards linked-list does have a rather glaring problem. Iterating over a file *in order* has a runtime cost of *O(n²)*. A quadratic runtime just to read a file! That's awful.

Fortunately we can do better. Instead of a singly linked list, littlefs uses a multilayered linked-list often called a skip-list. However, unlike the most common type of skip-list, littlefs's skip-lists are strictly deterministic built around some interesting properties of the count-trailing-zeros (CTZ) instruction.

The rules CTZ skip-lists follow are that for every *n*th block where *n* is divisible by 2_ˣ_, that block contains a pointer to block *n*-2_ˣ_. This means that each block contains anywhere from 1 to log₂*n* pointers that skip to different preceding elements of the skip-list.

The name comes from heavy use of the CTZ instruction, which lets us calculate the power-of-two factors efficiently. For a give block *n*, that block contains ctz(*n*)+1 pointers.

```
A backwards CTZ skip-list
.--------.  .--------.  .--------.  .--------.  .--------.  .--------.
| data 0 |<-| data 1 |<-| data 2 |<-| data 3 |<-| data 4 |<-| data 5 |
|        |<-|        |--|        |<-|        |--|        |  |        |
|        |<-|        |--|        |--|        |--|        |  |        |
'--------'  '--------'  '--------'  '--------'  '--------'  '--------'
```

The additional pointers let us navigate the data-structure on disk much more efficiently than in a singly linked list.

Consider a path from data block 5 to data block 1. You can see how data block 3 was completely skipped:

```
.--------.  .--------.  .--------.  .--------.  .--------.  .--------.
| data 0 |  | data 1 |<-| data 2 |  | data 3 |  | data 4 |<-| data 5 |
|        |  |        |  |        |<-|        |--|        |  |        |
|        |  |        |  |        |  |        |  |        |  |        |
'--------'  '--------'  '--------'  '--------'  '--------'  '--------'
```

The path to data block 0 is even faster, requiring only two jumps:

```
.--------.  .--------.  .--------.  .--------.  .--------.  .--------.
| data 0 |  | data 1 |  | data 2 |  | data 3 |  | data 4 |<-| data 5 |
|        |  |        |  |        |  |        |  |        |  |        |
|        |<-|        |--|        |--|        |--|        |  |        |
'--------'  '--------'  '--------'  '--------'  '--------'  '--------'
```

We can find the runtime complexity by looking at the path to any block from the block containing the most pointers. Every step along the path divides the search space for the block in half, giving us a runtime of *O(log n)*. To get *to* the block with the most pointers, we can perform the same steps backwards, which puts the runtime at *O(2 log n)* = *O(log n)*. An interesting note is that this optimal path occurs naturally if we greedily choose the pointer that covers the most distance without passing our target.

So now we have a COW data structure that is cheap to append with a runtime of *O(1)*, and can be read with a worst case runtime of *O(n log n)*. Given that this runtime is also divided by the amount of data we can store in a block, this cost is fairly reasonable.

This is a new data structure, so we still have several questions. What is the storage overhead? Can the number of pointers exceed the size of a block? How do we store a CTZ skip-list in our metadata pairs?

To find the storage overhead, we can look at the data structure as multiple linked-lists. Each linked-list skips twice as many blocks as the previous, or from another perspective, each linked-list uses half as much storage as the previous. As we approach infinity, the storage overhead forms a geometric series. Solving this tells us that on average our storage overhead is only 2 pointers per block.

Because our file size is limited the word width we use to store sizes, we can also solve for the maximum number of pointers we would ever need to store in a block. If we set the overhead of pointers equal to the block size, we get the following equation. Note that both a smaller block size () and larger word width () result in more storage overhead.

Solving the equation for gives us the minimum block size for some common word widths:

1. 32-bit CTZ skip-list => minimum block size of 104 bytes
2. 64-bit CTZ skip-list => minimum block size of 448 bytes

littlefs uses a 32-bit word width, so our blocks can only overflow with pointers if they are smaller than 104 bytes. This is an easy requirement, as in practice, most block sizes start at 512 bytes. As long as our block size is larger than 104 bytes, we can avoid the extra logic needed to handle pointer overflow.

This last question is how do we store CTZ skip-lists? We need a pointer to the head block, the size of the skip-list, the index of the head block, and our offset in the head block. But it's worth noting that each size maps to a unique index + offset pair. So in theory we can store only a single pointer and size.

However, calculating the index + offset pair from the size is a bit complicated. We can start with a summation that loops through all of the blocks up until our given size. Let be the block size in bytes, be the word width in bits, be the index of the block in the skip-list, and be the file size in bytes:

This works quite well, but requires *O(n)* to compute, which brings the full runtime of reading a file up to *O(n² log n)*. Fortunately, that summation doesn't need to touch the disk, so the practical impact is minimal.

However, despite the integration of a bitwise operation, we can actually reduce this equation to a *O(1)* form. While browsing the amazing resource that is the `On-Line Encyclopedia of Integer Sequences (OEIS)`, I managed to find `A001511`, which matches the iteration of the CTZ instruction, and `A005187`, which matches its partial summation. Much to my surprise, these both result from simple equations, leading us to a rather unintuitive property that ties together two seemingly unrelated bitwise instructions:

where:

1. ctz() = the number of trailing bits that are 0 in
2. popcount() = the number of bits that are 1 in

Initial tests of this surprising property seem to hold. As approaches infinity, we end up with an average overhead of 2 pointers, which matches our assumption from earlier. During iteration, the popcount function seems to handle deviations from this average. Of course, just to make sure I wrote a quick script that verified this property for all 32-bit integers.

Now we can substitute into our original equation to find a more efficient equation for file size:

Unfortunately, the popcount function is non-injective, so we can't solve this equation for our index. But what we can do is solve for an index that is greater than with error bounded by the range of the popcount function. We can repeatedly substitute into the original equation until the error is smaller than our integer resolution. As it turns out, we only need to perform this substitution once, which gives us this formula for our index:

Now that we have our index , we can just plug it back into the above equation to find the offset. We run into a bit of a problem with integer overflow, but we can avoid this by rearranging the equation a bit:

Our solution requires quite a bit of math, but computers are very good at math. Now we can find both our block index and offset from a size in *O(1)*, letting us store CTZ skip-lists with only a pointer and size.

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.

```
                                    .--------.
                                   .|metadata|
                                   ||        |
                                   ||        |
                                   |'--------'
                                   '----|---'
                                        v
.--------.  .--------.  .--------.  .--------.
| data 0 |<-| data 1 |<-| data 2 |<-| data 3 |
|        |<-|        |--|        |  |        |
|        |  |        |  |        |  |        |
'--------'  '--------'  '--------'  '--------'

write data to disk, create copies
=>
                                    .--------.
                                   .|metadata|
                                   ||        |
                                   ||        |
                                   |'--------'
                                   '----|---'
                                        v
.--------.  .--------.  .--------.  .--------.
| data 0 |<-| data 1 |<-| data 2 |<-| data 3 |
|        |<-|        |--|        |  |        |
|        |  |        |  |        |  |        |
'--------'  '--------'  '--------'  '--------'
     ^ ^           ^
     | |           |    .--------.  .--------.  .--------.  .--------.
     | |           '----| new    |<-| new    |<-| new    |<-| new    |
     | '----------------| data 2 |<-| data 3 |--| data 4 |  | data 5 |
     '------------------|        |--|        |--|        |  |        |
                        '--------'  '--------'  '--------'  '--------'

commit to metadata pair
=>
                                                            .--------.
                                                           .|new     |
                                                           ||metadata|
                                                           ||        |
                                                           |'--------'
                                                           '----|---'
                                                                |
.--------.  .--------.  .--------.  .--------.                  |
| data 0 |<-| data 1 |<-| data 2 |<-| data 3 |                  |
|        |<-|        |--|        |  |        |                  |
|        |  |        |  |        |  |        |                  |
'--------'  '--------'  '--------'  '--------'                  |
     ^ ^           ^                                            v
     | |           |    .--------.  .--------.  .--------.  .--------.
     | |           '----| new    |<-| new    |<-| new    |<-| new    |
     | '----------------| data 2 |<-| data 3 |--| data 4 |  | data 5 |
     '------------------|        |--|        |--|        |  |        |
                        '--------'  '--------'  '--------'  '--------'
```

### 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.

But now we need to look at the `elephant` in the room. Where do all these blocks come from?

Deciding which block to use next is the responsibility of the block allocator. In filesystem design, block allocation is often a second-class citizen, but in a COW filesystem its role becomes much more important as it is needed for nearly every write to the filesystem.

Normally, block allocation involves some sort of free list or bitmap stored on the filesystem that is updated with free blocks. However, with power resilience, keeping these structures consistent becomes difficult. It doesn't help that any mistake in updating these structures can result in lost blocks that are impossible to recover.

littlefs takes a cautious approach. Instead of trusting a free list on disk, littlefs relies on the fact that the filesystem on disk is a mirror image of the free blocks on the disk. The block allocator operates much like a garbage collector in a scripting language, scanning for unused blocks on demand.

```
            .----.
            |root|
            |    |
            '----'
   v-------'  '-------v
.----.    .    .    .----.
| A  |    .    .    | B  |
|    |    .    .    |    |
'----'    .    .    '----'
.    .    .    .  v--'  '------------v---------v
.    .    .    .----.    .         .----.    .----.
.    .    .    | C  |    .         | D  |    | E  |
.    .    .    |    |    .         |    |    |    |
.    .    .    '----'    .         '----'    '----'
.    .    .    .    .    .         .    .    .    .
.----.----.----.----.----.----.----.----.----.----.----.----.
| A  |    |root| C  | B  |         | D  |    | E  |         |
|    |    |    |    |    |         |    |    |    |         |
'----'----'----'----'----'----'----'----'----'----'----'----'
        ^                   ^    ^                   ^    ^
         '-------------------'----'-------------------'----'-- free blocks
```

While this approach may sound complicated, the decision to not maintain a free list greatly simplifies the overall design of littlefs. Unlike programming languages, there are only a handful of data structures we need to traverse. And block deallocation, which occurs nearly as often as block allocation, is simply a noop. This "drop it on the floor" strategy greatly reduces the complexity of managing on disk data structures, especially when handling high-risk error conditions.

Our block allocator needs to find free blocks efficiently. You could traverse through every block on storage and check each one against our filesystem tree; however, the runtime would be abhorrent. We need to somehow collect multiple blocks per traversal.

Looking at existing designs, some larger filesystems that use a similar "drop it on the floor" strategy store a bitmap of the entire storage in `RAM`. This works well because bitmaps are surprisingly compact. We can't use the same strategy here, as it violates our constant RAM requirement, but we may be able to modify the idea into a workable solution.

```
.----.----.----.----.----.----.----.----.----.----.----.----.
| A  |    |root| C  | B  |         | D  |    | E  |         |
|    |    |    |    |    |         |    |    |    |         |
'----'----'----'----'----'----'----'----'----'----'----'----'
  1    0    1    1    1    0    0    1    0    1    0    0
 \---------------------------+----------------------------/
                             v
               bitmap: 0xb94 (0b101110010100)
```

The block allocator in littlefs is a compromise between a disk-sized bitmap and a brute force traversal. Instead of a bitmap the size of storage, we keep track of a small, fixed-size bitmap called the lookahead buffer. During block allocation, we take blocks from the lookahead buffer. If the lookahead buffer is empty, we scan the filesystem for more free blocks, populating our lookahead buffer. In each scan we use an increasing offset, circling the storage as blocks are allocated.

Here's what it might look like to allocate 4 blocks on a decently busy filesystem with a 32 bit lookahead and a total of 128 blocks (512 KiB of storage if blocks are 4 KiB):

```
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
scanning...     lookahead:         fffffffe
                fs blocks: fffffffffffffffeffffffffffff0000
alloc = 63      lookahead:         ffffffff
                fs blocks: ffffffffffffffffffffffffffff0000
scanning...     lookahead:         ffffffff
                fs blocks: ffffffffffffffffffffffffffff0000
scanning...     lookahead:                 ffffffff
                fs blocks: ffffffffffffffffffffffffffff0000
scanning...     lookahead:                         ffff0000
                fs blocks: ffffffffffffffffffffffffffff0000
alloc = 112     lookahead:                         ffff8000
                fs blocks: ffffffffffffffffffffffffffff8000
```

This lookahead approach has a runtime complexity of *O(n²)* to completely scan storage; however, bitmaps are surprisingly compact, and in practice only one or two passes are usually needed to find free blocks. Additionally, the performance of the allocator can be optimized by adjusting the block size or size of the lookahead buffer, trading either write granularity or RAM for allocator performance.

### Wear leveling

The block allocator has a secondary role: wear leveling.

Wear leveling is the process of distributing wear across all blocks in the storage to prevent the filesystem from experiencing an early death due to wear on a single block in the storage.

littlefs has two methods of protecting against wear:

1. Detection and recovery from bad blocks
2. Evenly distributing wear across dynamic blocks

Recovery from bad blocks doesn't actually have anything to do with the block allocator itself. Instead, it relies on the ability of the filesystem to detect and evict bad blocks when they occur.

In littlefs, it is fairly straightforward to detect bad blocks at write time. All writes must be sourced by some form of data in RAM, so immediately after we write to a block, we can read the data back and verify that it was written correctly. If we find that the data on disk does not match the copy we have in RAM, a write error has occurred and we most likely have a bad block.

Once we detect a bad block, we need to recover from it. In the case of write errors, we have a copy of the corrupted data in RAM, so all we need to do is evict the bad block, allocate a new, hopefully good block, and repeat the write that previously failed.

The actual act of evicting the bad block and replacing it with a new block is left up to the filesystem's copy-on-bounded-writes (CObW) data structures. One property of CObW data structures is that any block can be replaced during a COW operation. The bounded-writes part is normally triggered by a counter, but nothing prevents us from triggering a COW operation as soon as we find a bad block.

```
     .----.
     |root|
     |    |
     '----'
   v--'  '----------------------v
.----.                        .----.
| A  |                        | B  |
|    |                        |    |
'----'                        '----'
.    .                      v---'  .
.    .                   .----.    .
.    .                   | C  |    .
.    .                   |    |    .
.    .                   '----'    .
.    .                   .    .    .
.----.----.----.----.----.----.----.----.----.----.
| A  |root|              | C  | B  |              |
|    |    |              |    |    |              |
'----'----'----'----'----'----'----'----'----'----'

update C
=>
     .----.
     |root|
     |    |
     '----'
   v--'  '----------------------v
.----.                        .----.
| A  |                        | B  |
|    |                        |    |
```

```
'----'                        '----'
.    .                      v---'  .
.    .                   .----.    .
.    .                   |bad |    .
.    .                   |blck|    .
.    .                   '----'    .
.    .                   .    .    .
.----.----.----.----.----.----.----.----.----.----.
| A  |root|              |bad | B  |              |
|    |    |              |blck|    |              |
'----'----'----'----'----'----'----'----'----'----'

oh no! bad block! relocate C
=>
     .----.
     |root|
     |    |
     '----'
   v--'  '----------------------v
.----.                        .----.
| A  |                        | B  |
|    |                        |    |
'----'                        '----'
.    .                      v---'  .
.    .                   .----.    .
.    .                   |bad |    .
.    .                   |blck|    .
.    .                   '----'    .
.    .                   .    .    .
.----.----.----.----.----.----.----.----.----.----.
| A  |root|              |bad | B  |bad |         |
|    |    |              |blck|    |blck|         |
'----'----'----'----'----'----'----'----'----'----'
                            --------->
oh no! bad block! relocate C
=>
     .----.
     |root|
     |    |
     '----'
   v--'  '----------------------v
.----.                        .----.
| A  |                        | B  |
|    |                        |    |
'----'                        '----'
.    .                      v---'  .
.    .                   .----.    .    .----.
.    .                   |bad |    .    | C' |
.    .                   |blck|    .    |    |
.    .                   '----'    .    '----'
.    .                   .    .    .    .    .
.----.----.----.----.----.----.----.----.----.----.
| A  |root|              |bad | B  |bad | C' |    |
|    |    |              |blck|    |blck|    |    |
'----'----'----'----'----'----'----'----'----'----'
                            -------------->
successfully relocated C, update B
=>
     .----.
     |root|
     |    |
     '----'
   v--'  '----------------------v
.----.                        .----.
| A  |                        |bad |
|    |                        |blck|
'----'                        '----'
.    .                      v---'  .
.    .                   .----.    .    .----.
.    .                   |bad |    .    | C' |
.    .                   |blck|    .    |    |
.    .                   '----'    .    '----'
.    .                   .    .    .    .    .
.----.----.----.----.----.----.----.----.----.----.
| A  |root|              |bad |bad |bad | C' |    |
|    |    |              |blck|blck|blck|    |    |
'----'----'----'----'----'----'----'----'----'----'

oh no! bad block! relocate B
=>
     .----.
     |root|
     |    |
     '----'
   v--'  '----------------------v
.----.                        .----.         .----.
| A  |                        |bad |         |bad |
```

```
|    |                        |blck|          |blck|
'----'                        '----'          '----'
.    .                      v---'  .         .    .
.    .                   .----.    .    .----.    .
.    .                   |bad |    .    | C' |    .
.    .                   |blck|    .    |    |    .
.    .                   '----'    .    '----'    .
.    .                   .    .    .    .    .    .
.----.----.----.----.----.----.----.----.----.----.
| A  |root|              |bad |bad |bad | C' |bad |
|    |    |              |blck|blck|blck|    |blck|
'----'----'----'----'----'----'----'----'----'----'
                                 -------------->
oh no! bad block! relocate B
=>
     .----.
     |root|
     |    |
     '----'
   v--'  '----------------------v
.----.    .----.              .----.
| A  |    | B' |              |bad |
|    |    |    |              |blck|
'----'    '----'              '----'
.    .    .  | .            .---'  .
.    .    .  '--------------v-------------v
.    .    .    .         .----.    .    .----.
.    .    .    .         |bad |    .    | C' |
.    .    .    .         |blck|    .    |    |
.    .    .    .         '----'    .    '----'
.    .    .    .         .    .    .    .    .
.----.----.----.----.----.----.----.----.----.----.
| A  |root| B' |         |bad |bad |bad | C' |bad |
|    |    |    |         |blck|blck|blck|    |blck|
'----'----'----'----'----'----'----'----'----'----'
------------>                    ------------------
successfully relocated B, update root
=>
     .----.
     |root|
     |    |
     '----'
   v--'  '--v
.----.    .----.
| A  |    | B' |
|    |    |    |
'----'    '----'
.    .    .   '---------------------------v
.    .    .    .                        .----.
.    .    .    .                        | C' |
.    .    .    .                        |    |
.    .    .    .                        '----'
.    .    .    .                        .    .
.----.----.----.----.----.----.----.----.----.----.
| A  |root| B' |         |bad |bad |bad | C' |bad |
|    |    |    |         |blck|blck|blck|    |blck|
'----'----'----'----'----'----'----'----'----'----'
```

We may find that the new block is also bad, but hopefully after repeating this cycle we'll eventually find a new block where a write succeeds. If we don't, that means that all blocks in our storage are bad, and we've reached the end of our device's usable life. At this point, littlefs will return an "out of space" error. This is technically true, as there are no more good blocks, but as an added benefit it also matches the error condition expected by users of dynamically sized data.

Read errors, on the other hand, are quite a bit more complicated. We don't have a copy of the data lingering around in RAM, so we need a way to reconstruct the original data even after it has been corrupted. One such mechanism for this is `error-correction-codes (ECC)`.

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. However, there is a limit to how many errors ECC can detect: the `Hamming bound`. As the number of errors approaches the Hamming bound, we may still be able to detect errors, but can no longer fix the data. If we've reached this point the block is unrecoverable.

littlefs by itself does **not** provide ECC. The block nature and relatively large footprint of ECC does not work well with the dynamically sized data of filesystems, correcting errors without RAM is complicated, and ECC fits better with the geometry of block devices. In fact, several NOR flash chips have extra storage intended for ECC, and many NAND chips can even calculate ECC on the chip itself.

In littlefs, ECC is entirely optional. Read errors can instead be prevented proactively by wear leveling. But it's important to note that ECC can be used at the block device level to modestly extend the life of a device. littlefs respects any errors reported by the block device, allowing a block device to provide additional aggressive error detection.

To avoid read errors, we need to be proactive, as opposed to reactive as we were with write errors.

One way to do this is to detect when the number of errors in a block exceeds some threshold, but is still recoverable. With ECC we can do this at write time, and treat the error as a write error, evicting the block before fatal read errors have a chance to develop.

A different, more generic strategy, is to proactively distribute wear across all blocks in the storage, with the hope that no single block fails before the rest of storage is approaching the end of its usable life. This is called wear leveling.

Generally, wear leveling algorithms fall into one of two categories:

1. `Dynamic wear leveling`, where we distribute wear over "dynamic" blocks. The can be accomplished by only considering unused blocks.
2. `Static wear leveling`, where we distribute wear over both "dynamic" and "static" blocks. To make this work, we need to consider all blocks, including blocks that already contain data.

As a tradeoff for code size and complexity, littlefs (currently) only provides dynamic wear leveling. This is a best effort solution. Wear is not distributed perfectly, but it is distributed among the free blocks and greatly extends the life of a device.

On top of this, littlefs uses a statistical wear leveling algorithm. What this means is that we don’t actively track wear, instead we rely on a uniform distribution of wear across storage to approximate a dynamic wear leveling algorithm. Despite the long name, this is actually a simplification of dynamic wear leveling.

The uniform distribution of wear is left up to the block allocator, which creates a uniform distribution in two parts. The easy part is when the device is powered, in which case we allocate the blocks linearly, circling the device. The harder part is what to do when the device loses power. We can't just restart the allocator at the beginning of storage, as this would bias the wear. Instead, we start the allocator as a random offset every time we mount the filesystem. As long as this random offset is uniform, the combined allocation pattern is also a uniform distribution.

Initially, this approach to wear leveling looks like it creates a difficult dependency on a power-independent random number generator, which must return different random numbers on each boot. However, the filesystem is in a relatively unique situation in that it is sitting on top of a large of amount of entropy that persists across power loss.

We can actually use the data on disk to directly drive our random number generator. In practice, this is implemented by xoring the checksums of each metadata pair, which is already calculated to fetch and mount the filesystem.

```
           .--------. \                         probably random
          .|metadata| |                                ^
          ||        | +-> crc ----------------------> xor
          ||        | |                                ^
          |'--------' /                                |
          '---|--|-'                                   |
           .-'    '-------------------------.          |
          |                                  |         |
          |        .--------------> xor ------------> xor
           |        |                 ^       |         ^
           v       crc               crc      v        crc
      .--------. \  ^   .--------. \  ^   .--------. \  ^
     .|metadata|-|--|-->|metadata| |  |  .|metadata| |  |
     ||        | +--'  ||        | +--'  ||        | +--'
     ||        | |     ||        | |     ||        | |
     |'--------' /     |'--------' /     |'--------' /
     '---|--|-'        '----|---'        '---|--|-'
      .-'    '-.            |             .-'    '-.
     v          v           v            v          v
.--------.  .--------.  .--------.  .--------.  .--------.
|  data  |  |  data  |  |  data  |  |  data  |  |  data  |
|        |  |        |  |        |  |        |  |        |
|        |  |        |  |        |  |        |  |        |
'--------'  '--------'  '--------'  '--------'  '--------'
```

Note that this random number generator is not perfect. It only returns unique random numbers when the filesystem is modified. This is exactly what we want for distributing wear in the allocator, but means this random number generator is not useful for general use.

Together, bad block detection and dynamic wear leveling provide a best effort solution for avoiding the early death of a filesystem due to wear. Importantly, littlefs's wear leveling algorithm provides a key feature: You can increase the life of a device simply by increasing the size of storage. And if more aggressive wear leveling is desired, you can always combine littlefs with a `flash translation layer (FTL)` to get a small power resilient filesystem with static wear leveling.

## Files

Now that we have our building blocks out of the way, we can start looking at our filesystem as a whole.

The first step: How do we actually store our files?

We've determined that CTZ skip-lists are pretty good at storing data compactly, so following the precedent found in other filesystems we could give each file a skip-list stored in a metadata pair that acts as an inode for the file.

```
                                .--------.
                               .|metadata|
                               ||        |
                               ||        |
                               |'--------'
                               '----|---'
                                    v
.--------.  .--------.  .--------.  .--------.
| data 0 |<-| data 1 |<-| data 2 |<-| data 3 |
|        |<-|        |--|        |  |        |
|        |  |        |  |        |  |        |
'--------'  '--------'  '--------'  '--------'
```

However, this doesn't work well when files are small, which is common for embedded systems. Compared to PCs, *all* data in an embedded system is small.

Consider a small 4-byte file. With a two block metadata-pair and one block for the CTZ skip-list, we find ourselves using a full 3 blocks. On most NOR flash with 4 KiB blocks, this is 12 KiB of overhead. A ridiculous 3072x increase.

```
file stored as inode, 4 bytes costs ~12 KiB

 .----------------.                  \
.|    revision    |                  |
||----------------|    \             |
||    skiplist   ---.  +- metadata   |
||----------------| |  /  4x8 bytes  |
||    checksum    | |     32 bytes   |
||----------------| |                |
||       |        | |                +- metadata pair
||       v        | |                |  2x4 KiB
||                | |                |  8 KiB
||                | |                |
||                | |                |
||                | |                |
|'----------------' |                |
'----------------'  |                /
          .--------'
         v
 .----------------.    \             \
 |      data      |    +- data       |
 |----------------|    /  4 bytes    |
 |                |                  |
 |                |                  |
 |                |                  |
 |                |                  +- data block
 |                |                  |  4 KiB
 |                |                  |
 |                |                  |
 |                |                  |
 |                |                  |
 |                |                  |
 '----------------'                  /
```

We can make several improvements. First, instead of giving each file its own metadata pair, we can store multiple files in a single metadata pair. One way to do this is to directly associate a directory with a metadata pair (or a linked list of metadata pairs). This makes it easy for multiple files to share the directory's metadata pair for logging and reduces the collective storage overhead.

The strict binding of metadata pairs and directories also gives users direct control over storage utilization depending on how they organize their directories.

```
multiple files stored in metadata pair, 4 bytes costs ~4 KiB

       .----------------.
      .|    revision    |
      ||----------------|
      ||    A name      |
      ||   A skiplist  -----.
      ||----------------|   |  \
      ||    B name      |   |  +- metadata
      ||   B skiplist  ---. |  |  4x8 bytes
      ||----------------| | |  /  32 bytes
      ||    checksum    | | |
      ||----------------| | |
      ||       |        | | |
      ||       v        | | |
      |'----------------' | |
      '----------------'  | |
         .----------------' |
        v                   v
.----------------.  .----------------.  \           \
|     A data     |  |     B data     |  +- data     |
|                |  |----------------|  /  4 bytes  |
|                |  |                |              |
|                |  |                |              |
|                |  |                |              |
|                |  |                |              + data block
|                |  |                |              | 4 KiB
|                |  |                |              |
|----------------|  |                |              |
|                |  |                |              |
|                |  |                |              |
|                |  |                |              |
'----------------'  '----------------'              /
```

The second improvement we can make is noticing that for very small files, our attempts to use CTZ skip-lists for compact storage backfires. Metadata pairs have a $\sim$4x storage cost, so if our file is smaller than 1/4 the block size, there's actually no benefit in storing our file outside of our metadata pair.

In this case, we can store the file directly in our directory's metadata pair. We call this an inline file, and it allows a directory to store many small files quite efficiently. Our previous 4 byte file now only takes up a theoretical 16 bytes on disk.

```
inline files stored in metadata pair, 4 bytes costs ~16 bytes

 .----------------.
.|    revision    |
||----------------|
||    A name      |
||   A skiplist  ---.
||----------------| |  \
||    B name      | |  +- data
||    B data      | |  |  4x4 bytes
||----------------| |  /  16 bytes
||    checksum    | |
||----------------| |
||       |        | |
||       v        | |
|'----------------' |
'----------------'  |
          .---------'
         v
 .----------------.
 |     A data     |
 |                |
 |                |
 |                |
 |                |
 |                |
```

```
 |                |
 |                |
 |----------------|
 |                |
 |                |
 |                |
 '----------------'
```

Once the file exceeds 1/4 the block size, we switch to a CTZ skip-list. This means that our files never use more than 4x storage overhead, decreasing as the file grows in size.

### Directories

Now we just need directories to store our files. As mentioned above we want a strict binding of directories and metadata pairs, but there are a few complications we need to sort out.

On their own, each directory is a linked-list of metadata pairs. This lets us store an unlimited number of files in each directory, and we don't need to worry about the runtime complexity of unbounded logs. We can store other directory pointers in our metadata pairs, which gives us a directory tree, much like what you find on other filesystems.

```
            .---------.
           .| root    |
           ||         |
           ||         |
           |'---------'
           '---|--|-'
            .-'     '---------------------------.
           v                                     v
     .---------.         .---------.         .---------.
    .| dir A   |-------->| dir A   |        .| dir B   |
    ||         |        ||         |        ||         |
    ||         |        ||         |        ||         |
    |'---------'        |'---------'        |'---------'
    '---|--|-'          '----|---'          '---|--|-'
      .-'     '-.            |                .-'     '-.
     v           v           v              v            v
.---------.  .---------.  .---------.  .---------.  .---------.
| file C  |  | file D  |  | file E  |  | file F  |  | file G  |
|         |  |         |  |         |  |         |  |         |
|         |  |         |  |         |  |         |  |         |
'---------'  '---------'  '---------'  '---------'  '---------'
```

The main complication is, once again, traversal with a constant amount of RAM. The directory tree is a tree, and the unfortunate fact is you can't traverse a tree with constant RAM.

Fortunately, the elements of our tree are metadata pairs, so unlike CTZ skip-lists, we're not limited to strict COW operations. One thing we can do is thread a linked-list through our tree, explicitly enabling cheap traversal over the entire filesystem.

```
            .---------.
           .| root    |-.
           ||         | |
  .--------||         |-'
  |        |'---------'
  |        '---|--|-'
  |         .-'     '---------------------------.
  |        v                                     v
  |  .---------.         .---------.         .---------.
   '->| dir A   |-------->| dir A   |-------->| dir B   |
    ||         |        ||         |        ||         |
    ||         |        ||         |        ||         |
    |'---------'        |'---------'        |'---------'
    '---|--|-'          '----|---'          '---|--|-'
      .-'     '-.            |                .-'     '-.
     v           v           v              v            v
.---------.  .---------.  .---------.  .---------.  .---------.
| file C  |  | file D  |  | file E  |  | file F  |  | file G  |
|         |  |         |  |         |  |         |  |         |
|         |  |         |  |         |  |         |  |         |
'---------'  '---------'  '---------'  '---------'  '---------'
```

Unfortunately, not sticking to pure COW operations creates some problems. Now, whenever we want to manipulate the directory tree, multiple pointers need to be updated. If you're familiar with designing atomic data structures this should set off a bunch of red flags.

To work around this, our threaded linked-list has a bit of leeway. Instead of only containing metadata pairs found in our filesystem, it is allowed to contain metadata pairs that have no parent because of a power loss. These are called orphaned metadata pairs.

With the possibility of orphans, we can build power loss resilient operations that maintain a filesystem tree threaded with a linked-list for traversal.

Adding a directory to our tree:

```
         .--------.
        .| root   |-.
        ||        | |
.-------||        |-'
|       |'--------'
|       '---|--|-'
|        .-'    '-.
|       v          v
|  .--------.  .--------.
'->| dir A  |->| dir C  |
  ||        | ||        |
  ||        | ||        |
  |'--------' |'--------'
  '--------'  '--------'

allocate dir B
=>
         .--------.
        .| root   |-.
        ||        | |
.-------||        |-'
|       |'--------'
|       '---|--|-'
|        .-'    '-.
|       v          v
|  .--------.    .--------.
'->| dir A  |--->| dir C  |
  ||        | .->|        |
  ||        | |  ||       |
  |'--------' | |'--------'
  '--------'  | '--------'
              |
   .--------. |
  .| dir B  |-'
  ||        |
  ||        |
  |'--------'
  '--------'

insert dir B into threaded linked-list, creating an orphan
=>
         .--------.
        .| root   |-.
        ||        | |
.-------||        |-'
|       |'--------'
|       '---|--|-'
|        .-'    '-------------.
|       v                      v
|  .--------.  .--------.  .--------.
'->| dir A  |->| dir B  |->| dir C  |
  ||        | || orphan!| ||        |
  ||        | ||        | ||        |
  |'--------' |'--------' |'--------'
  '--------'  '--------'  '--------'

add dir B to parent directory
=>
               .--------.
              .| root   |-.
              ||        | |
.-------------||        |-'
|             |'--------'
|             '--|-|-|-'
|        .------'  |  '-------.
|       v          v           v
|  .--------.  .--------.  .--------.
```

```
'->| dir A  |->| dir B  |->| dir C  |
   ||       | ||       | ||       |
   ||       | ||       | ||       |
   |'--------' |'--------' |'--------'
   '--------'  '--------'  '--------'
```

Removing a directory:

```
               .--------.
              .| root   |-.
              ||        | |
.-------------||        |-'
|             |'--------'
|             '--|-|-|-'
|        .------'  |  '-------.
|       v          v           v
|  .--------.  .--------.  .--------.
'->| dir A  |->| dir B  |->| dir C  |
   ||       | ||       | ||       |
   ||       | ||       | ||       |
   |'--------' |'--------' |'--------'
   '--------'  '--------'  '--------'

remove dir B from parent directory, creating an orphan
=>
         .--------.
        .| root   |-.
        ||        | |
.-------||        |-'
|       |'--------'
|       '---|--|-'
|        .-'    '-------------.
|       v                      v
|  .--------.  .--------.  .--------.
'->| dir A  |->| dir B  |->| dir C  |
   ||       | || orphan!| ||       |
   ||       | ||       | ||       |
   |'--------' |'--------' |'--------'
   '--------'  '--------'  '--------'

remove dir B from threaded linked-list, returning dir B to free blocks
=>
         .--------.
        .| root   |-.
        ||        | |
.-------||        |-'
|       |'--------'
|       '---|--|-'
|        .-'    '-.
|       v          v
|  .--------.  .--------.
'->| dir A  |->| dir C  |
   ||       | ||       |
   ||       | ||       |
   |'--------' |'--------'
   '--------'  '--------'
```

In addition to normal directory tree operations, we can use orphans to evict blocks in a metadata pair when the block goes bad or exceeds its allocated erases. If we lose power while evicting a metadata block we may end up with a situation where the filesystem references the replacement block while the threaded linked-list still contains the evicted block. We call this a half-orphan.

```
               .--------.
              .| root   |-.
              ||        | |
.-------------||        |-'
|             |'--------'
|             '--|-|-|-'
|        .------'  |  '-------.
|       v          v           v
|  .--------.  .--------.  .--------.
'->| dir A  |->| dir B  |->| dir C  |
   ||       | ||       | ||       |
   ||       | ||       | ||       |
   |'--------' |'--------' |'--------'
   '--------'  '--------'  '--------'

try to write to dir B
```

```
=>
                  .--------.
                 .| root   |-.
                 ||        | |
.----------------||        |-'
|                |'--------'
|                '-|-||-|-'
|        .--------'  ||  '-----.
|       v            |v         v
|  .--------.     .--------.  .--------.
'->| dir A  |---->| dir B  |->| dir C  |
  ||        |-.   |        | ||        |
  ||        | |   |        | ||        |
  |'--------' |   '--------' |'--------'
  '--------'  |      v       '--------'
              |  .--------.
              '->| dir B  |
                 | bad    |
                 | block! |
                 '--------'

oh no! bad block detected, allocate replacement
=>
                  .--------.
                 .| root   |-.
                 ||        | |
.----------------||        |-'
|                |'--------'
|                '-|-||-|-'
|        .--------'  ||  '-------.
|       v            |v           v
|  .--------.     .--------.    .--------.
'->| dir A  |---->| dir B  |--->| dir C  |
  ||        |-.   |        | .->|        |
  ||        | |   |        | | ||        |
  |'--------' |   '--------' | |'--------'
  '--------'  |      v       | '--------'
              |  .--------.  |
              '->| dir B  |  |
                 | bad    |  |
                 | block! |  |
                 '--------'  |
                             |
                 .--------.  |
                 | dir B  |--'
                 |        |
                 |        |
                 '--------'

insert replacement in threaded linked-list, creating a half-orphan
=>
                  .--------.
                 .| root   |-.
                 ||        | |
.----------------||        |-'
|                |'--------'
|                '-|-||-|-'
|        .--------'  ||  '-------.
|       v            |v           v
|  .--------.     .--------.    .--------.
'->| dir A  |---->| dir B  |--->| dir C  |
  ||        |-.   |        | .->|        |
  ||        | |   |        | | ||        |
  |'--------' |   '--------' | |'--------'
  '--------'  |      v       | '--------'
              |  .--------.  |
              |  | dir B  |  |
              |  | bad    |  |
              |  | block! |  |
              |  '--------'  |
              |              |
              |  .--------.  |
              '->| dir B  |--'
                 | half   |
                 | orphan!|
                 '--------'

fix reference in parent directory
=>
               .--------.
              .| root   |-.
              ||        | |
.-------------||        |-'
|             |'--------'
|             '--|-|-|-'
|        .------'  |  '-------.
|       v          v           v
```

```
|  .--------.  .--------.  .--------.
'->| dir A  |->| dir B  |->| dir C  |
  ||        | ||        | ||        |
  ||        | ||        | ||        |
  |'--------' |'--------' |'--------'
  '--------'  '--------'  '--------'
```

Finding orphans and half-orphans is expensive, requiring a *O(n²)* comparison of every metadata pair with every directory entry. But the tradeoff is a power resilient filesystem that works with only a bounded amount of RAM. Fortunately, we only need to check for orphans on the first allocation after boot, and a read-only littlefs can ignore the threaded linked-list entirely.

If we only had some sort of global state, then we could also store a flag and avoid searching for orphans unless we knew we were specifically interrupted while manipulating the directory tree (foreshadowing!).

## The move problem

We have one last challenge: the move problem. Phrasing the problem is simple:

How do you atomically move a file between two directories?

In littlefs we can atomically commit to directories, but we can't create an atomic commit that spans multiple directories. The filesystem must go through a minimum of two distinct states to complete a move.

To make matters worse, file moves are a common form of synchronization for filesystems. As a filesystem designed for power-loss, it's important we get atomic moves right.

So what can we do?

- We definitely can't just let power-loss result in duplicated or lost files. This could easily break users' code and would only reveal itself in extreme cases. We were only able to be lazy about the threaded linked-list because it isn't user facing and we can handle the corner cases internally.
- Some filesystems propagate COW operations up the tree until a common parent is found. Unfortunately this interacts poorly with our threaded tree and brings back the issue of upward propagation of wear.
- In a previous version of littlefs we tried to solve this problem by going back and forth between the source and destination, marking and unmarking the file as moving in order to make the move atomic from the user perspective. This worked, but not well. Finding failed moves was expensive and required a unique identifier for each file.

In the end, solving the move problem required creating a new mechanism for sharing knowledge between multiple metadata pairs. In littlefs this led to the introduction of a mechanism called "global state".

Global state is a small set of state that can be updated from *any* metadata pair. Combining global state with metadata pairs' ability to update multiple entries in one commit gives us a powerful tool for crafting complex atomic operations.

How does global state work?

Global state exists as a set of deltas that are distributed across the metadata pairs in the filesystem. The actual global state can be built out of these deltas by xoring together all of the deltas in the filesystem.

```
 .--------.  .--------.  .--------.  .--------.  .--------.
.|        |->| gdelta |->|        |->| gdelta |->| gdelta |
||        | || 0x23   | ||        | || 0xff   | || 0xce   |
||        | ||        | ||        | ||        | ||        |
|'--------' |'--------' |'--------' |'--------' |'--------'
'--------'  '----|---'  '--------'  '----|---'  '----|---'
                 v                       v           v
       0x00 --> xor ------------------> xor ------> xor --> gstate 0x12
```

To update the global state from a metadata pair, we take the global state we know and xor it with both our changes and any existing delta in the metadata pair. Committing this new delta to the metadata pair commits the changes to the filesystem's global state.

```
 .--------.  .--------.  .--------.  .--------.  .--------.
.|        |->| gdelta |->|        |->| gdelta |->| gdelta |
||        | || 0x23   | ||        | || 0xff   | || 0xce   |
||        | ||        | ||        | ||        | ||        |
|'--------' |'--------' |'--------' |'--------' |'--------'
'--------'  '----|---'  '--------'  '--|---|-'  '----|---'
                 v                     v   |         v
       0x00 --> xor ----------------> xor -|------> xor --> gstate = 0x12
                                           |                          |
                                           |                          |
change gstate to 0xab --> xor <------------|--------------------------'
=>                         |               v
                           '------------> xor
                                           |
                                           v
 .--------.  .--------.  .--------.  .--------.  .--------.
.|        |->| gdelta |->|        |->| gdelta |->| gdelta |
||        | || 0x23   | ||        | || 0x46   | || 0xce   |
||        | ||        | ||        | ||        | ||        |
|'--------' |'--------' |'--------' |'--------' |'--------'
'--------'  '----|---'  '--------'  '----|---'  '----|---'
                 v                       v           v
       0x00 --> xor ------------------> xor ------> xor --> gstate = 0xab
```

To make this efficient, we always keep a copy of the global state in RAM. We only need to iterate over our metadata pairs and build the global state when the filesystem is mounted.

You may have noticed that global state is very expensive. We keep a copy in RAM and a delta in an unbounded number of metadata pairs. Even if we reset the global state to its initial value, we can't easily clean up the deltas on disk. For this reason, it's very important that we keep the size of global state bounded and extremely small. But, even with a strict budget, global state is incredibly valuable.

Now we can solve the move problem. We can create global state describing our move atomically with the creation of the new file, and we can clear this move state atomically with the removal of the old file.

```
               .--------.    gstate = no move
              .| root   |-.
              ||        | |
.-------------||        |-'
|             |'--------'
|             '--|-|-|-'
|        .------'  |  '-------.
|       v          v           v
|  .--------.  .--------.  .--------.
'->| dir A  |->| dir B  |->| dir C  |
  ||        | ||        | ||        |
  ||        | ||        | ||        |
  |'--------' |'--------' |'--------'
  '----|---'  '--------'  '--------'
       v
   .--------.
   | file D |
   |        |
   |        |
   '--------'

begin move, add reference in dir C, change gstate to have move
=>
               .--------.    gstate = moving file D in dir A (m1)
              .| root   |-.
              ||        | |
.-------------||        |-'
|             |'--------'
|             '--|-|-|-'
|        .------'  |  '-------.
|       v          v           v
|  .--------.  .--------.  .--------.
'->| dir A  |->| dir B  |->| dir C  |
  ||        | ||        | || gdelta |
  ||        | ||        | || =m1    |
  |'--------' |'--------' |'--------'
  '----|---'  '--------'  '----|---'
       |     .----------------'
```

```
       v    v
     .--------.
     | file D |
     |        |
     |        |
     '--------'

complete move, remove reference in dir A, change gstate to no move
=>
               .--------.    gstate = no move (m1^~m1)
              .| root   |-.
              ||        | |
.-------------||        |-'
|             |'--------'
|             '--|-|-|-'
|        .------'  |  '-------.
|       v          v           v
|  .--------.  .--------.  .--------.
'->| dir A  |->| dir B  |->| dir C  |
  || gdelta | ||        | || gdelta |
  || =~m1   | ||        | || =m1    |
  |'--------' |'--------' |'--------'
  '--------'  '--------'  '----|---'
                               v
                           .--------.
                           | file D |
                           |        |
                           |        |
                           '--------'
```

If, after building our global state during mount, we find information describing an ongoing move, we know we lost power during a move and the file is duplicated in both the source and destination directories. If this happens, we can resolve the move using the information in the global state to remove one of the files.

```
                 .--------.    gstate = moving file D in dir A (m1)
                .| root   |-.             ^
                ||        |------------> xor
.---------------||        |-'             ^
|               |'--------'               |
|               '--|-|-|-'                |
|        .--------'  |  '---------.       |
|       |            |             |      |
|       |     .----------> xor --------> xor
|       v     |      v      ^      v      ^
|  .--------. |  .--------. |  .--------. |
'->| dir A  |-|->| dir B  |-|->| dir C  | |
  ||        |-' ||        |-' || gdelta |-'
  ||        |   ||        |   || =m1    |
  |'--------'   |'--------'   |'--------'
  '----|---'    '--------'    '----|---'
       |     .---------------------'
       v    v
     .--------.
     | file D |
     |        |
     |        |
     '--------'
```

We can also move directories the same way we move files. There is the threaded linked-list to consider, but leaving the threaded linked-list unchanged works fine as the order doesn't really matter.

```
               .--------.    gstate = no move (m1^~m1)
              .| root   |-.
              ||        | |
.-------------||        |-'
|             |'--------'
|             '--|-|-|-'
|        .------'  |  '-------.
|       v          v           v
|  .--------.  .--------.  .--------.
'->| dir A  |->| dir B  |->| dir C  |
  || gdelta | ||        | || gdelta |
  || =~m1   | ||        | || =m1    |
  |'--------' |'--------' |'--------'
  '--------'  '--------'  '----|---'
                               v
                           .--------.
                           | file D |
```

```
                           |        |
                           |        |
                           '--------'

begin move, add reference in dir C, change gstate to have move
=>
                .--------.    gstate = moving dir B in root (m1^~m1^m2)
               .| root   |-.
               ||        | |
.--------------||        |-'
|              |'--------'
|              '--|-|-|-'
|        .-------'  |  '----------.
|       v           |              v
|  .--------.       |          .--------.
'->| dir A  |-.     |       .->| dir C  |
  || gdelta | |     |       | || gdelta |
  || =~m1   | |     |       | || =m1^m2 |
  |'--------' |     |       | |'--------'
  '--------'  |     |       | '---|--|-'
              |     |    .-------'   |
              |     v   v   |        v
              |  .--------. |    .--------.
              '->| dir B  |-'    | file D |
                ||        |      |        |
                ||        |      |        |
                |'--------'      '--------'
                '--------'

complete move, remove reference in root, change gstate to no move
=>
             .--------.    gstate = no move (m1^~m1^m2^~m2)
            .| root   |-.
            || gdelta | |
.-----------|| =~m2   |-'
|           |'--------'
|           '---|--|-'
|        .-----'    '-----.
|       v                  v
|  .--------.          .--------.
'->| dir A  |-.     .->| dir C  |
  || gdelta | |     | || gdelta |
  || =~m1   | |     '-|| =m1^m2 |-------.
  |'--------' |       |'--------'       |
  '--------'  |       '---|--|-'        |
              |        .-'    '-.       |
              |       v          v      |
              |  .--------.  .--------. |
              '->| dir B  |--| file D |-'
                ||        |  |        |
                ||        |  |        |
                |'--------'  '--------'
                '--------'
```

Global state gives us a powerful tool we can use to solve the move problem. And the result is surprisingly performant, only needing the minimum number of states and using the same number of commits as a naive move. Additionally, global state gives us a bit of persistent state we can use for some other small improvements.

## Conclusion

And that's littlefs, thanks for reading!

## Chapter 3

# LICENSE

# Chapter 4

# littlefs

A little fail-safe filesystem designed for microcontrollers.

```
   | | |     .---._____
  .-----.   |          |
--|o    |---| littlefs |
--|     |---|          |
  '-----'   '----------'
   | | |
```

**Power-loss resilience** - littlefs is designed to handle random power failures. All file operations have strong copy-on-write guarantees and if power is lost the filesystem will fall back to the last known good state.

**Dynamic wear leveling** - littlefs is designed with flash in mind, and provides wear leveling over dynamic blocks. Additionally, littlefs can detect bad blocks and work around them.

**Bounded RAM/ROM** - littlefs is designed to work with a small amount of memory. RAM usage is strictly bounded, which means RAM consumption does not change as the filesystem grows. The filesystem contains no unbounded recursion and dynamic memory is limited to configurable buffers that can be provided statically.

## Example

Here's a simple example that updates a file named `boot_count` every time main runs. The program can be interrupted at any time without losing track of how many times it has been booted and without corrupting the filesystem:

```
#include "lfs.h"

// variables used by the filesystem
lfs_t lfs;
lfs_file_t file;

// configuration of the filesystem is provided by this struct
const struct lfs_config cfg = {
    // block device operations
     .read  = user_provided_block_device_read,
     .prog  = user_provided_block_device_prog,
     .erase = user_provided_block_device_erase,
     .sync  = user_provided_block_device_sync,

     // block device configuration
     .read_size = 16,
     .prog_size = 16,
     .block_size = 4096,
     .block_count = 128,
     .cache_size = 16,
     .lookahead_size = 16,
```

```
    .block_cycles = 500,
};

// entry point
int main(void) {
    // mount the filesystem
    int err = lfs_mount(&lfs, &cfg);

    // reformat if we can't mount the filesystem
    // this should only happen on the first boot
    if (err) {
        lfs_format(&lfs, &cfg);
        lfs_mount(&lfs, &cfg);
    }

    // read current count
    uint32_t boot_count = 0;
    lfs_file_open(&lfs, &file, "boot_count", LFS_O_RDWR | LFS_O_CREAT);
    lfs_file_read(&lfs, &file, &boot_count, sizeof(boot_count));

    // update boot count
    boot_count += 1;
    lfs_file_rewind(&lfs, &file);
    lfs_file_write(&lfs, &file, &boot_count, sizeof(boot_count));

    // remember the storage is not updated until the file is closed successfully
    lfs_file_close(&lfs, &file);

    // release any resources we were using
    lfs_unmount(&lfs);

    // print the boot count
    printf("boot_count: %d\n", boot_count);
}
```

## Usage

Detailed documentation (or at least as much detail as is currently available) can be found in the comments in `lfs.h`.

littlefs takes in a configuration structure that defines how the filesystem operates. The configuration struct provides the filesystem with the block device operations and dimensions, tweakable parameters that tradeoff memory usage for performance, and optional static buffers if the user wants to avoid dynamic memory.

The state of the littlefs is stored in the `lfs_t` type which is left up to the user to allocate, allowing multiple filesystems to be in use simultaneously. With the `lfs_t` and configuration struct, a user can format a block device or mount the filesystem.

Once mounted, the littlefs provides a full set of POSIX-like file and directory functions, with the deviation that the allocation of filesystem structures must be provided by the user.

All POSIX operations, such as remove and rename, are atomic, even in event of power-loss. Additionally, file updates are not actually committed to the filesystem until sync or close is called on the file.

## Other notes

Littlefs is written in C, and specifically should compile with any compiler that conforms to the `C99` standard.

All littlefs calls have the potential to return a negative error code. The errors can be either one of those found in the `enum lfs_error` in `lfs.h`, or an error returned by the user's block device operations.

In the configuration struct, the `prog` and `erase` function provided by the user may return a `LFS_ERR_CORRUPT` error if the implementation already can detect corrupt blocks. However, the wear leveling does not depend on the return code of these functions, instead all data is read back and checked for integrity.

If your storage caches writes, make sure that the provided `sync` function flushes all the data to memory and ensures that the next read fetches the data from memory, otherwise data integrity can not be guaranteed. If the `write` function does not perform caching, and therefore each `read` or `write` call hits the memory, the `sync` function can simply return 0.

## Design

At a high level, littlefs is a block based filesystem that uses small logs to store metadata and larger copy-on-write (COW) structures to store file data.

In littlefs, these ingredients form a sort of two-layered cake, with the small logs (called metadata pairs) providing fast updates to metadata anywhere on storage, while the COW structures store file data compactly and without any wear amplification cost.

Both of these data structures are built out of blocks, which are fed by a common block allocator. By limiting the number of erases allowed on a block per allocation, the allocator provides dynamic wear leveling over the entire filesystem.

```
                   root
                  .--------.--------.
                  | A'| B'|         |
                  |   |   |->       |
                  |   |   |         |
                  '--------'--------'
               .----'   '--------------.
      A       v                 B       v
     .--------.--------.       .--------.--------.
      | C'| D'|         |       | E'|new|         |
      |   |   |->       |       |   | E'|->       |
      |   |   |         |       |   |   |         |
      '--------'--------'       '--------'--------'
      .-'   '--.                  |   '------------------.
     v          v              .-'                        v
.--------.  .--------.        v                       .--------.
|   C    |  |   D    |   .--------.       write       | new E  |
|        |  |        |   |   E    |        ==>        |        |
|        |  |        |   |        |                   |        |
'--------'  '--------'   |        |                   '--------'
                         '--------'                   .-'    |
                         .-'    '-.    .-------------|------'
                        v          v  v              v
                   .--------.  .--------.       .--------.
                   |   F    |  |   G    |       | new F  |
                   |        |  |        |       |        |
                   |        |  |        |       |        |
                   '--------'  '--------'       '--------'
```

More details on how littlefs works can be found in DESIGN.md and SPEC.md.

- DESIGN.md - A fully detailed dive into how littlefs works. I would suggest reading it as the tradeoffs at work are quite interesting.
- SPEC.md - The on-disk specification of littlefs with all the nitty-gritty details. May be useful for tooling development.

## Testing

The littlefs comes with a test suite designed to run on a PC using the `emulated block device` found in the emubd directory. The tests assume a Linux environment and can be started with make:

```
make test
```

## License

The littlefs is provided under the `BSD-3-Clause` license. See LICENSE.md for more information. Contributions to this project are accepted under the same license.

Individual files contain the following tag instead of the full license text.

`SPDX-License-Identifier:    BSD-3-Clause`

This enables machine processing of license information based on the SPDX License Identifiers that are here available: `http://spdx.org/licenses/`

## Related projects

- `littlefs-fuse` - A `FUSE` wrapper for littlefs. The project allows you to mount littlefs directly on a Linux machine. Can be useful for debugging littlefs if you have an SD card handy.
- `littlefs-js` - A javascript wrapper for littlefs. I'm not sure why you would want this, but it is handy for demos. You can see it in action `here`.
- `littlefs-python` - A Python wrapper for littlefs. The project allows you to create images of the filesystem on your PC. Check if littlefs will fit your needs, create images for a later download to the target memory or inspect the content of a binary image of the target memory.
- `mklfs` - A command line tool built by the `Lua RTOS` guys for making littlefs images from a host PC. Supports Windows, Mac OS, and Linux.
- `Mbed OS` - The easiest way to get started with littlefs is to jump into Mbed which already has block device drivers for most forms of embedded storage. littlefs is available in Mbed OS as the `LittleFileSystem` class.
- `SPIFFS` - Another excellent embedded filesystem for NOR flash. As a more traditional logging filesystem with full static wear-leveling, SPIFFS will likely outperform littlefs on small memories such as the internal flash on microcontrollers.
- `Dhara` - An interesting NAND flash translation layer designed for small MCUs. It offers static wear-leveling and power-resilience with only a fixed $O(|address|)$ pointer structure stored on each block and in RAM.

# Chapter 5

# littlefs technical specification

This is the technical specification of the little filesystem. This document covers the technical details of how the littlefs is stored on disk for introspection and tooling. This document assumes you are familiar with the design of the littlefs, for more info on how littlefs works check out DESIGN.md.

```
   | | |     .---._____
  .-----.   |          |
--|o    |---| littlefs |
--|     |---|          |
  '-----'   '----------'
   | | |
```

### Some quick notes

- 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 pointers are stored in 32 bits, with the special value `0xffffffff` representing a null block address.
- 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.

### Directories / Metadata pairs

Metadata pairs form the backbone of littlefs and provide a system for distributed atomic updates. Even the superblock is stored in a metadata pair.

As their name suggests, a metadata pair is stored in two blocks, with one block providing a backup during erase cycles in case power is lost. These two blocks are not necessarily sequential and may be anywhere on disk, so a "pointer" to a metadata pair is stored as two block pointers.

On top of this, each metadata block behaves as an appendable log, containing a variable number of commits. Commits can be appended to the metadata log in order to update the metadata without requiring an erase cycles. Note that successive commits may supersede the metadata in previous commits. Only the most recent metadata should be considered valid.

The high-level layout of a metadata block is fairly simple:

```
  .---------------------------------------.
.-|  revision count   |      entries      |  \
| |-------------------+                   |  |
| |                                       |  |
| |                                       |  +-- 1st commit
| |                                       |  |
| |                   +-------------------|  |
| |                   |        CRC        |  /
| |-------------------+-------------------|
| |                entries                |  \
| |                                       |  |
| |                                       |  +-- 2nd commit
| |    +-------------------+--------------|  |
| |    |        CRC        |    padding   |  /
| |----+-------------------+--------------|
| |                entries                |  \
| |                                       |  |
| |                                       |  +-- 3rd commit
| |         +-------------------+---------|  |
| |         |        CRC        |         |  /
| |---------+-------------------+         |
| |           unwritten storage           |  more commits
| |                                       |       |
| |                                       |       v
| |                                       |
| |                                       |
| '---------------------------------------'
'---------------------------------------'
```

Each metadata block contains a 32-bit revision count followed by a number of commits. Each commit contains a variable number of metadata entries followed by a 32-bit CRC.

Note also that entries aren't necessarily word-aligned. This allows us to store metadata more compactly, however we can only write to addresses that are aligned to our program block size. This means each commit may have padding for alignment.

Metadata block fields:

1. **Revision count (32-bits)** - Incremented every erase cycle. If both blocks contain valid commits, only the block with the most recent revision count should be used. Sequence comparison must be used to avoid issues with integer overflow.
2. **CRC (32-bits)** - Detects corruption from power-loss or other write issues. Uses a CRC-32 with a polynomial of `0x04c11db7` initialized with `0xffffffff`.

Entries themselves are stored as a 32-bit tag followed by a variable length blob of data. But exactly how these tags are stored is a little bit tricky.

Metadata blocks support both forward and backward iteration. In order to do this without duplicating the space for each tag, neighboring entries have their tags XORed together, starting with `0xffffffff`.

```
Forward iteration                        Backward iteration

.-------------------.  0xffffffff        .-------------------.
|  revision count   |      |             |  revision count   |
|-------------------|      v             |-------------------|
|      tag ~A       |---> xor -> tag A   |      tag ~A       |---> xor -> 0xffffffff
|-------------------|      |             |-------------------|      ^
|      data A       |      |             |      data A       |      |
|                   |      |             |                   |      |
|                   |      |             |                   |      |
|-------------------|      v             |-------------------|      |
|      tag AxB      |---> xor -> tag B   |      tag AxB      |---> xor -> tag A
|-------------------|      |             |-------------------|      ^
|      data B       |      |             |      data B       |      |
|                   |      |             |                   |      |
|                   |      |             |                   |      |
|-------------------|      v             |-------------------|      |
|      tag BxC      |---> xor -> tag C   |      tag BxC      |---> xor -> tag B
|-------------------|                    |-------------------|      ^
|      data C       |                    |      data C       |      |
|                   |                    |                   |    tag C
|                   |                    |                   |
|                   |                    |                   |
'-------------------'                    '-------------------'
```

One last thing to note before we get into the details around tag encoding. Each tag contains a valid bit used to indicate if the tag and containing commit is valid. This valid bit is the first bit found in the tag and the commit and can be used to tell if we've attempted to write to the remaining space in the block.

Here's a more complete example of metadata block containing 4 entries:

```
  .----------------------------------------.
.-|  revision count   |      tag ~A        |        \
| |-------------------+--------------------|        |
| |                 data A                 |        |
| |                                        |        |
| |-------------------+--------------------|        |
| |      tag AxB      |       data B       | <--.   |
| |-------------------+                    |    |   |
| |                                        |    |   +-- 1st commit
| |         +-------------------+----------|    |   |
| |         |      tag BxC      |          | <-.|   |
| |---------+-------------------+          |   ||   |
| |                 data C                 |   ||   |
| |                                        |   ||   |
| |-------------------+--------------------|   ||   |
| |     tag CxCRC     |        CRC         |   ||   /
| |-------------------+--------------------|   ||
| |     tag CRCxA'    |      data A'       |   ||   \
| |-------------------+                    |   ||   |
| |                                        |   ||   |
| |              +-------------------+-----|   ||   +-- 2nd commit
| |              |     tag CRCxA'    |     |   ||   |
| |--------------+-------------------+-----|   ||   |
| | CRC          |        padding          |   ||   /
| |--------------+----+--------------------|   ||
| |     tag CRCxA''   |      data A''      | <---.  \
| |-------------------+                    |   |||  |
| |                                        |   |||  |
| |         +-------------------+----------|   |||  |
| |         |     tag A''xD     |          | < |||  |
| |---------+-------------------+          |  ||||  +-- 3rd commit
| |                data D                  |  ||||  |
| |                             +----------|  ||||  |
| |                             |   tag Dx |  ||||  |
| |---------+-------------------+----------|  ||||  |
| |CRC      |        CRC        |          |  ||||  /
| |---------+-------------------+          |  ||||
| |           unwritten storage            |  ||||  more commits
| |                                        |  ||||       |
| |                                        |  ||||       v
| |                                        |  ||||
| |                                        |  ||||
| '----------------------------------------'  ||||
'---------------------------------------'     |||'- most recent A
                                              ||'-- most recent B
                                              |'--- most recent C
                                              '---- most recent D
```

## Metadata tags

So in littlefs, 32-bit tags describe every type of metadata. And this means *every* type of metadata, including file entries, directory fields, and global state. Even the CRCs used to mark the end of commits get their own tag.

Because of this, the tag format contains some densely packed information. Note that there are multiple levels of types which break down into more info:

```
[----            32             ----]
[1|--  11   --|--  10  --|--  10  --]
 ^.       ^     .     ^          ^- length
 |.       |     .     '------------ id
 |.       '-----.------------------ type (type3)
 '.-----------.------------------ valid bit
  [-3-|-- 8 --]
    ^     ^- chunk
    '------- type (type1)
```

Before we go further, there's one important thing to note. These tags are **not** stored in little-endian. Tags stored in commits are actually stored in big-endian (and is the only thing in littlefs stored in big-endian). This little bit of craziness comes from the fact that the valid bit must be the first bit in a commit, and when converted to little-endian, the valid bit finds itself in byte 4. We could restructure the tag to store the valid bit lower, but, because none of the fields are byte-aligned, this would be more complicated than just storing the tag in big-endian.

Another thing to note is that both the tags `0x00000000` and `0xffffffff` are invalid and can be used for null values.

Metadata tag fields:

1. **Valid bit (1-bit)** - Indicates if the tag is valid.
2. **Type3 (11-bits)** - Type of the tag. This field is broken down further into a 3-bit abstract type and an 8-bit chunk field. Note that the value `0x000` is invalid and not assigned a type.
3. **Type1 (3-bits)** - Abstract type of the tag. Groups the tags into 8 categories that facilitate bitmasked lookups.
4. **Chunk (8-bits)** - Chunk field used for various purposes by the different abstract types. type1+chunk+id form a unique identifier for each tag in the metadata block.
5. **Id (10-bits)** - File id associated with the tag. Each file in a metadata block gets a unique id which is used to associate tags with that file. The special value `0x3ff` is used for any tags that are not associated with a file, such as directory and global metadata.
6. **Length (10-bits)** - Length of the data in bytes. The special value `0x3ff` indicates that this tag has been deleted.

### Metadata types

What follows is an exhaustive list of metadata in littlefs.

#### `0x401` LFS_TYPE_CREATE

Creates a new file with this id. Note that files in a metadata block don't necessarily need a create tag. All a create does is move over any files using this id. In this sense a create is similar to insertion into an imaginary array of files.

The create and delete tags allow littlefs to keep files in a directory ordered alphabetically by filename.

#### `0x4ff` LFS_TYPE_DELETE

Deletes the file with this id. An inverse to create, this tag moves over any files neighboring this id similar to a deletion from an imaginary array of files.

0x0xx **LFS_TYPE_NAME**

Associates the id with a file name and file type.

The data contains the file name stored as an ASCII string (may be expanded to UTF8 in the future).

The chunk field in this tag indicates an 8-bit file type which can be one of the following.

Currently, the name tag must precede any other tags associated with the id and can not be reassigned without deleting the file.

Layout of the name tag:

```
        tag                          data
[--      32      --][---        variable length        ---]
[1| 3| 8 | 10 | 10 ][---          (size * 8)           ---]
 ^  ^  ^    ^    ^- size                   ^- file name
 |  |  |    '------ id
 |  |  '----------- file type
 |  '-------------- type1 (0x0)
 '----------------- valid bit
```

Name fields:

1. **file type (8-bits)** - Type of the file.
2. **file name** - File name stored as an ASCII string.

   0x001 **LFS_TYPE_REG**

Initializes the id + name as a regular file.

How each file is stored depends on its struct tag, which is described below.

0x002 **LFS_TYPE_DIR**

Initializes the id + name as a directory.

Directories in littlefs are stored on disk as a linked-list of metadata pairs, each pair containing any number of files in alphabetical order. A pointer to the directory is stored in the struct tag, which is described below.

0x0ff **LFS_TYPE_SUPERBLOCK**

Initializes the id as a superblock entry.

The superblock entry is a special entry used to store format-time configuration and identify the filesystem.

The name is a bit of a misnomer. While the superblock entry serves the same purpose as a superblock found in other filesystems, in littlefs the superblock does not get a dedicated block. Instead, the superblock entry is duplicated across a linked-list of metadata pairs rooted on the blocks 0 and 1. The last metadata pair doubles as the root directory of the filesystem.

```
 .--------.  .--------.  .--------.  .--------.  .--------.
.| super  |->| super  |->| super  |->| super  |->| file B |
|| block  | || block  | || block  | || block  | || file C |
||        | ||        | ||        | || file A | || file D |
|'--------' |'--------' |'--------' |'--------' |'--------'
'--------'  '--------'  '--------'  '--------'  '--------'

\----------------+----------------/ \----------+----------/
          superblock pairs               root directory
```

The filesystem starts with only the root directory. The superblock metadata pairs grow every time the root pair is compacted in order to prolong the life of the device exponentially.

The contents of the superblock entry are stored in a name tag with the superblock type and an inline-struct tag. The name tag contains the magic string "littlefs", while the inline-struct tag contains version and configuration information.

Layout of the superblock name tag and inline-struct tag:

```
        tag                          data
[--      32      --][--      32      --|--      32      --]
[1|- 11 -| 10 | 10 ][---              64               ---]
 ^    ^     ^    ^- size (8)           ^- magic string ("littlefs")
 |    |     '------ id (0)
 |    '------------ type (0x0ff)
 '----------------- valid bit

        tag                          data
[--      32      --][--      32      --|--      32      --|--      32      --]
[1|- 11 -| 10 | 10 ][--      32      --|--      32      --|--      32      --]
 ^    ^     ^    ^            ^- version         ^- block size      ^- block count
 |    |     |    |  [--      32      --|--      32      --|--      32      --]
 |    |     |    |  [--      32      --|--      32      --|--      32      --]
 |    |     |    |            ^- name max        ^- file max        ^- attr max
 |    |     |    '- size (24)
 |    |     '------ id (0)
 |    '------------ type (0x201)
 '----------------- valid bit
```

Superblock fields:

1. **Magic string (8-bytes)** - Magic string indicating the presence of littlefs on the device. Must be the string "littlefs".
2. **Version (32-bits)** - The version of littlefs at format time. The version is encoded in a 32-bit value with the upper 16-bits containing the major version, and the lower 16-bits containing the minor version.

   This specification describes version 2.0 (`0x00020000`).
3. **Block size (32-bits)** - Size of the logical block size used by the filesystem in bytes.
4. **Block count (32-bits)** - Number of blocks in the filesystem.
5. **Name max (32-bits)** - Maximum size of file names in bytes.
6. **File max (32-bits)** - Maximum size of files in bytes.
7. **Attr max (32-bits)** - Maximum size of file attributes in bytes.

The superblock must always be the first entry (id 0) in a metadata pair as well as be the first entry written to the block. This means that the superblock entry can be read from a device using offsets alone.

`0x2xx` **LFS_TYPE_STRUCT**

Associates the id with an on-disk data structure.

The exact layout of the data depends on the data structure type stored in the chunk field and can be one of the following.

Any type of struct supersedes all other structs associated with the id. For example, appending a ctz-struct replaces an inline-struct on the same file.

`0x200` **LFS_TYPE_DIRSTRUCT**

Gives the id a directory data structure.

Directories in littlefs are stored on disk as a linked-list of metadata pairs, each pair containing any number of files in alphabetical order.

```
     |
     v
 .--------.  .--------.  .--------.  .--------.  .--------.  .--------.
.| file A |->| file D |->| file G |->| file I |->| file J |->| file M |
|| file B | || file E | || file H | ||        | || file K | || file N |
|| file C | || file F | ||        | ||        | || file L | ||        |
|’--------’ |’--------’ |’--------’ |’--------’ |’--------’ |’--------’
’--------’  ’--------’  ’--------’  ’--------’  ’--------’  ’--------’
```

The dir-struct tag contains only the pointer to the first metadata-pair in the directory. The directory size is not known without traversing the directory.

The pointer to the next metadata-pair in the directory is stored in a tail tag, which is described below.

Layout of the dir-struct tag:

```
        tag                          data
[--      32      --][--      32      --|--      32      --]
[1|- 11 -| 10 | 10 ][---              64               ---]
 ^    ^     ^    ^- size (8)           ^- metadata pair
 |    |     ’------ id
 |    ’------------ type (0x200)
 ’----------------- valid bit
```

Dir-struct fields:

1. **Metadata pair (8-bytes)** - Pointer to the first metadata-pair in the directory.

   `0x201` **LFS_TYPE_INLINESTRUCT**

Gives the id an inline data structure.

Inline structs store small files that can fit in the metadata pair. In this case, the file data is stored directly in the tag's data area.

Layout of the inline-struct tag:

```
        tag                          data
[--      32      --][---        variable length        ---]
[1|- 11 -| 10 | 10 ][---           (size * 8)          ---]
 ^    ^     ^    ^- size                    ^- inline data
 |    |     ’------ id
 |    ’------------ type (0x201)
 ’----------------- valid bit
```

Inline-struct fields:

1. **Inline data** - File data stored directly in the metadata-pair.

#### 0x202 LFS_TYPE_CTZSTRUCT

Gives the id a CTZ skip-list data structure.

CTZ skip-lists store files that can not fit in the metadata pair. These files are stored in a skip-list in reverse, with a pointer to the head of the skip-list. Note that the head of the skip-list and the file size is enough information to read the file.

How exactly CTZ skip-lists work is a bit complicated. A full explanation can be found in the `DESIGN.md`.

A quick summary: For every *n*th block where *n* is divisible by 2_ˣ_, that block contains a pointer to block *n*-2_ˣ_. These pointers are stored in increasing order of *x* in each block of the file before the actual data.

```
                                                               |
                                                               v
.--------.  .--------.  .--------.  .--------.  .--------.  .--------.
| A      |<-| D      |<-| G      |<-| J      |<-| M      |<-| P      |
| B      |<-| E      |--| H      |<-| K      |--| N      |  | Q      |
| C      |<-| F      |--| I      |--| L      |--| O      |  |        |
'--------'  '--------'  '--------'  '--------'  '--------'  '--------'
  block 0     block 1     block 2     block 3     block 4     block 5
              1 skip      2 skips     1 skip      3 skips     1 skip
```

Note that the maximum number of pointers in a block is bounded by the maximum file size divided by the block size. With 32 bits for file size, this results in a minimum block size of 104 bytes.

Layout of the CTZ-struct tag:

```
        tag                           data
[--      32      --][--      32      --|--      32      --]
[1|- 11 -| 10 | 10 ][--      32      --|--      32      --]
 ^    ^     ^    ^            ^                  ^- file size
 |    |     |    |            '-------------------- file head
 |    |     |    '- size (8)
 |    |     '------ id
 |    '------------ type (0x202)
 '----------------- valid bit
```

CTZ-struct fields:

1. **File head (32-bits)** - Pointer to the block that is the head of the file's CTZ skip-list.
2. **File size (32-bits)** - Size of the file in bytes.

#### 0x3xx LFS_TYPE_USERATTR

Attaches a user attribute to an id.

littlefs has a concept of "user attributes". These are small user-provided attributes that can be used to store things like timestamps, hashes, permissions, etc.

Each user attribute is uniquely identified by an 8-bit type which is stored in the chunk field, and the user attribute itself can be found in the tag's data.

There are currently no standard user attributes and a portable littlefs implementation should work with any user attributes missing.

Layout of the user-attr tag:

```
        tag                          data
[--      32      --][---        variable length        ---]
[1| 3| 8 | 10 | 10 ][---           (size * 8)          ---]
 ^  ^  ^    ^    ^- size                   ^- attr data
 |  |  |    '------ id
 |  |  '----------- attr type
 |  '-------------- type1 (0x3)
 '----------------- valid bit
```

User-attr fields:

1. **Attr type (8-bits)** - Type of the user attributes.
2. **Attr data** - The data associated with the user attribute.

`0x6xx` **LFS_TYPE_TAIL**

Provides the tail pointer for the metadata pair itself.

The metadata pair's tail pointer is used in littlefs for a linked-list containing all metadata pairs. The chunk field contains the type of the tail, which indicates if the following metadata pair is a part of the directory (hard-tail) or only used to traverse the filesystem (soft-tail).

```
         .--------.
        .| dir A  |-.
        ||softtail| |
.--------|        |-'
|       |'--------'
|       '---|--|-'
|        .-'    '-------------.
|       v                      v
|  .--------.  .--------.  .--------.
'->| dir B  |->| dir B  |->| dir C  |
   ||hardtail| ||softtail| ||        |
   ||        | ||        | ||        |
   |'--------' |'--------' |'--------'
   '--------'  '--------'  '--------'
```

Currently any type supersedes any other preceding tails in the metadata pair, but this may change if additional metadata pair state is added.

A note about the metadata pair linked-list: Normally, this linked-list contains every metadata pair in the filesystem. However, there are some operations that can cause this linked-list to become out of sync if a power-loss were to occur. When this happens, littlefs sets the "sync" flag in the global state. How exactly this flag is stored is described below.

When the sync flag is set:

1. The linked-list may contain an orphaned directory that has been removed in the filesystem.
2. The linked-list may contain a metadata pair with a bad block that has been replaced in the filesystem.

If the sync flag is set, the threaded linked-list must be checked for these errors before it can be used reliably. Note that the threaded linked-list can be ignored if littlefs is mounted read-only.

Layout of the tail tag:

```
        tag                          data
[--      32      --][--      32      --|--      32      --]
[1| 3| 8 | 10 | 10 ][---            64            ---]
 ^  ^  ^    ^    ^- size (8)           ^- metadata pair
 |  |  |    '------ id
 |  |  '----------- tail type
 |  '------------- type1 (0x6)
 '---------------- valid bit
```

Tail fields:

1. **Tail type (8-bits)** - Type of the tail pointer.
2. **Metadata pair (8-bytes)** - Pointer to the next metadata-pair.

`0x600` **LFS_TYPE_SOFTTAIL**

Provides a tail pointer that points to the next metadata pair in the filesystem.

In this case, the next metadata pair is not a part of our current directory and should only be followed when traversing the entire filesystem.

`0x601` **LFS_TYPE_HARDTAIL**

Provides a tail pointer that points to the next metadata pair in the directory.

In this case, the next metadata pair belongs to the current directory. Note that because directories in littlefs are sorted alphabetically, the next metadata pair should only contain filenames greater than any filename in the current pair.

`0x7xx` **LFS_TYPE_GSTATE**

Provides delta bits for global state entries.

littlefs has a concept of "global state". This is a small set of state that can be updated by a commit to *any* metadata pair in the filesystem.

The way this works is that the global state is stored as a set of deltas distributed across the filesystem such that the global state can be found by the xor-sum of these deltas.

```
 .--------.  .--------.  .--------.  .--------.  .--------.
.|        |->| gdelta |->|        |->| gdelta |->| gdelta |
||        | || 0x23   | ||        | || 0xff   | || 0xce   |
||        | ||        | ||        | ||        | ||        |
|'--------' |'--------' |'--------' |'--------' |'--------'
'--------'  '----|---'  '--------'  '----|---'  '----|---'
                 v                       v           v
       0x00 --> xor ------------------> xor ------> xor --> gstate = 0x12
```

Note that storing globals this way is very expensive in terms of storage usage, so any global state should be kept very small.

The size and format of each piece of global state depends on the type, which is stored in the chunk field. Currently, the only global state is move state, which is outlined below.

`0x7ff` **LFS_TYPE_MOVESTATE**

Provides delta bits for the global move state.

The move state in littlefs is used to store info about operations that could cause to filesystem to go out of sync if the power is lost. The operations where this could occur is moves of files between metadata pairs and any operation that changes metadata pairs on the threaded linked-list.

In the case of moves, the move state contains a tag + metadata pair describing the source of the ongoing move. If this tag is non-zero, that means that power was lost during a move, and the file exists in two different locations. If this happens, the source of the move should be considered deleted, and the move should be completed (the source should be deleted) before any other write operations to the filesystem.

In the case of operations to the threaded linked-list, a single "sync" bit is used to indicate that a modification is ongoing. If this sync flag is set, the threaded linked-list will need to be checked for errors before it can be used reliably. The exact cases to check for are described above in the tail tag.

Layout of the move state:

```
        tag                                    data
[--      32      --][--      32      --|--      32      --|--      32      --]
[1|- 11 -| 10 | 10 ][1|- 11 -| 10 | 10 |---              64              ---]
 ^    ^     ^    ^   ^    ^     ^    ^- padding (0)       ^- metadata pair
 |    |     |    |   |    |     '------ move id
 |    |     |    |   |    '------------ move type
 |    |     |    |   '----------------- sync bit
 |    |     |    |
 |    |     |    '- size (12)
 |    |     '------ id (0x3ff)
 |    '------------ type (0x7ff)
 '----------------- valid bit
```

Move state fields:

1. **Sync bit (1-bit)** - Indicates if the metadata pair threaded linked-list is in-sync. If set, the threaded linked-list should be checked for errors.
2. **Move type (11-bits)** - Type of move being performed. Must be either `0x000`, indicating no move, or `0x4ff` indicating the source file should be deleted.
3. **Move id (10-bits)** - The file id being moved.
4. **Metadata pair (8-bytes)** - Pointer to the metadata-pair containing the move.

   `0x5xx` **LFS_TYPE_CRC**

Last but not least, the CRC tag marks the end of a commit and provides a checksum for any commits to the metadata block.

The first 32-bits of the data contain a CRC-32 with a polynomial of `0x04c11db7` initialized with `0xffffffff`. This CRC provides a checksum for all metadata since the previous CRC tag, including the CRC tag itself. For the first commit, this includes the revision count for the metadata block.

However, the size of the data is not limited to 32-bits. The data field may larger to pad the commit to the next program-aligned boundary.

In addition, the CRC tag's chunk field contains a set of flags which can change the behaviour of commits. Currently the only flag in use is the lowest bit, which determines the expected state of the valid bit for any following tags. This is used to guarantee that unwritten storage in a metadata block will be detected as invalid.

Layout of the CRC tag:

```
        tag                                    data
[--      32      --][--      32      --|---        variable length        ---]
[1| 3| 8 | 10 | 10 ][--      32      --|---        (size * 8 - 32)        ---]
 ^  ^  ^    ^    ^            ^- crc                             ^- padding
 |  |  |    |    '- size
 |  |  |    '------ id (0x3ff)
 |  |  '----------- valid state
 |  '-------------- type1 (0x5)
 '----------------- valid bit
```

CRC fields:

1. **Valid state (1-bit)** - Indicates the expected value of the valid bit for any tags in the next commit.
2. **CRC (32-bits)** - CRC-32 with a polynomial of `0x04c11db7` initialized with `0xffffffff`.
3. **Padding** - Padding to the next program-aligned boundary. No guarantees are made about the contents.

# Chapter 6

# Data Structure Index

## 6.1 Data Structures

Here are the data structures with brief descriptions:

# Chapter 7

# File Index

## 7.1 File List

Here is a list of all files with brief descriptions:

# Chapter 8

# Data Structure Documentation

## 8.1 lfs Struct Reference

```
#include <lfs.h>
```

### Data Structures

- struct lfs_free
- struct lfs_mlist

### Data Fields

- lfs_cache_t rcache
- lfs_cache_t pcache
- lfs_block_t root [2]
- struct lfs::lfs_mlist * mlist
- uint32_t seed
- lfs_gstate_t gstate
- lfs_gstate_t gdisk
- lfs_gstate_t gdelta
- struct lfs::lfs_free free
- const struct lfs_config * cfg
- lfs_size_t name_max
- lfs_size_t file_max
- lfs_size_t attr_max

### 8.1.1 Field Documentation

#### 8.1.1.1 lfs_size_t lfs::attr_max

#### 8.1.1.2 const struct lfs_config* lfs::cfg

#### 8.1.1.3 lfs_size_t lfs::file_max

#### 8.1.1.4 struct lfs::lfs_free lfs::free

#### 8.1.1.5 lfs_gstate_t lfs::gdelta

#### 8.1.1.6 lfs_gstate_t lfs::gdisk

#### 8.1.1.7 lfs_gstate_t lfs::gstate

#### 8.1.1.8 struct lfs::lfs_mlist * lfs::mlist

#### 8.1.1.9 lfs_size_t lfs::name_max

#### 8.1.1.10 lfs_cache_t lfs::pcache

#### 8.1.1.11 lfs_cache_t lfs::rcache

#### 8.1.1.12 lfs_block_t lfs::root[2]

#### 8.1.1.13 uint32_t lfs::seed

The documentation for this struct was generated from the following file:

- littlefs/lfs.h

## 8.2 lfs_attr Struct Reference

```
#include <lfs.h>
```

### Data Fields

- uint8_t type
- void * buffer
- lfs_size_t size

### 8.2.1 Field Documentation

#### 8.2.1.1 void* lfs_attr::buffer

#### 8.2.1.2 lfs_size_t lfs_attr::size

#### 8.2.1.3 uint8_t lfs_attr::type

The documentation for this struct was generated from the following file:

- littlefs/lfs.h

## 8.3 lfs_cache Struct Reference

internal littlefs data structures ///

`#include <lfs.h>`

### Data Fields

- lfs_block_t block
- lfs_off_t off
- lfs_size_t size
- uint8_t * buffer

### 8.3.1 Detailed Description

internal littlefs data structures ///

### 8.3.2 Field Documentation

#### 8.3.2.1 lfs_block_t lfs_cache::block

#### 8.3.2.2 uint8_t* lfs_cache::buffer

#### 8.3.2.3 lfs_off_t lfs_cache::off

#### 8.3.2.4 lfs_size_t lfs_cache::size

The documentation for this struct was generated from the following file:

- littlefs/lfs.h

## 8.4 lfs_commit Struct Reference

### Data Fields

- lfs_block_t block
- lfs_off_t off
- lfs_tag_t ptag
- uint32_t crc
- lfs_off_t begin
- lfs_off_t end

### 8.4.1 Field Documentation

#### 8.4.1.1 lfs_off_t lfs_commit::begin

#### 8.4.1.2 lfs_block_t lfs_commit::block

#### 8.4.1.3 uint32_t lfs_commit::crc

#### 8.4.1.4 lfs_off_t lfs_commit::end

#### 8.4.1.5 lfs_off_t lfs_commit::off

#### 8.4.1.6 lfs_tag_t lfs_commit::ptag

The documentation for this struct was generated from the following file:

- littlefs/lfs.c

## 8.5 lfs_config Struct Reference

```
#include <lfs.h>
```

### Data Fields

- void ∗ context
- int(∗ read )(const struct lfs_config ∗c, lfs_block_t block, lfs_off_t off, void ∗buffer, lfs_size_t size)
- int(∗ prog )(const struct lfs_config ∗c, lfs_block_t block, lfs_off_t off, const void ∗buffer, lfs_size_t size)
- int(∗ erase )(const struct lfs_config ∗c, lfs_block_t block)
- int(∗ sync )(const struct lfs_config ∗c)
- lfs_size_t read_size
- lfs_size_t prog_size
- lfs_size_t block_size
- lfs_size_t block_count
- int32_t block_cycles
- lfs_size_t cache_size
- lfs_size_t lookahead_size
- void ∗ read_buffer
- void ∗ prog_buffer
- void ∗ lookahead_buffer
- lfs_size_t name_max
- lfs_size_t file_max
- lfs_size_t attr_max
- lfs_size_t metadata_max

### 8.5.1 Field Documentation

#### 8.5.1.1 lfs_size_t lfs_config::attr_max

#### 8.5.1.2 lfs_size_t lfs_config::block_count

#### 8.5.1.3 int32_t lfs_config::block_cycles

#### 8.5.1.4 lfs_size_t lfs_config::block_size

#### 8.5.1.5 lfs_size_t lfs_config::cache_size

#### 8.5.1.6 void∗ lfs_config::context

#### 8.5.1.7 int(∗ lfs_config::erase) (const struct lfs_config ∗c, lfs_block_t block)

#### 8.5.1.8 lfs_size_t lfs_config::file_max

#### 8.5.1.9 void∗ lfs_config::lookahead_buffer

#### 8.5.1.10 lfs_size_t lfs_config::lookahead_size

#### 8.5.1.11 lfs_size_t lfs_config::metadata_max

#### 8.5.1.12 lfs_size_t lfs_config::name_max

#### 8.5.1.13 int(∗ lfs_config::prog) (const struct lfs_config ∗c, lfs_block_t block, lfs_off_t off, const void ∗buffer, lfs_size_t size)

#### 8.5.1.14 void∗ lfs_config::prog_buffer

#### 8.5.1.15 lfs_size_t lfs_config::prog_size

#### 8.5.1.16 int(∗ lfs_config::read) (const struct lfs_config ∗c, lfs_block_t block, lfs_off_t off, void ∗buffer, lfs_size_t size)

#### 8.5.1.17 void∗ lfs_config::read_buffer

#### 8.5.1.18 lfs_size_t lfs_config::read_size

#### 8.5.1.19 int(∗ lfs_config::sync) (const struct lfs_config ∗c)

The documentation for this struct was generated from the following file:

- littlefs/lfs.h

## 8.6 lfs_file::lfs_ctz Struct Reference

```
#include <lfs.h>
```

### Data Fields

- lfs_block_t head
- lfs_size_t size

### 8.6.1 Field Documentation

#### 8.6.1.1 lfs_block_t lfs_file::lfs_ctz::head

#### 8.6.1.2 lfs_size_t lfs_file::lfs_ctz::size

The documentation for this struct was generated from the following file:

- littlefs/lfs.h

## 8.7 lfs_dir Struct Reference

```
#include <lfs.h>
```

### Data Fields

- struct lfs_dir * next
- uint16_t id
- uint8_t type
- lfs_mdir_t m
- lfs_off_t pos
- lfs_block_t head [2]

### 8.7.1 Field Documentation

#### 8.7.1.1 lfs_block_t lfs_dir::head[2]

#### 8.7.1.2 uint16_t lfs_dir::id

#### 8.7.1.3 lfs_mdir_t lfs_dir::m

#### 8.7.1.4 struct lfs_dir* lfs_dir::next

#### 8.7.1.5 lfs_off_t lfs_dir::pos

#### 8.7.1.6 uint8_t lfs_dir::type

The documentation for this struct was generated from the following file:

- littlefs/lfs.h

## 8.8 lfs_dir_commit_commit Struct Reference

### Data Fields

- lfs_t * lfs
- struct lfs_commit * commit

### 8.8.1 Field Documentation

#### 8.8.1.1 struct lfs_commit* lfs_dir_commit_commit::commit

#### 8.8.1.2 lfs_t* lfs_dir_commit_commit::lfs

The documentation for this struct was generated from the following file:

- littlefs/lfs.c

## 8.9 lfs_dir_find_match Struct Reference

### Data Fields

- lfs_t * lfs
- const void * name
- lfs_size_t size

### 8.9.1 Field Documentation

#### 8.9.1.1 lfs_t* lfs_dir_find_match::lfs

#### 8.9.1.2 const void* lfs_dir_find_match::name

#### 8.9.1.3 lfs_size_t lfs_dir_find_match::size

The documentation for this struct was generated from the following file:

- littlefs/lfs.c

## 8.10 lfs_diskoff Struct Reference

### Data Fields

- lfs_block_t block
- lfs_off_t off

### 8.10.1 Field Documentation

#### 8.10.1.1 lfs_block_t lfs_diskoff::block

#### 8.10.1.2 lfs_off_t lfs_diskoff::off

The documentation for this struct was generated from the following file:

- littlefs/lfs.c

## 8.11 lfs_file Struct Reference

```
#include <lfs.h>
```

### Data Structures

- struct lfs_ctz

### Data Fields

- struct lfs_file ∗ next
- uint16_t id
- uint8_t type
- lfs_mdir_t m
- struct lfs_file::lfs_ctz ctz
- uint32_t flags
- lfs_off_t pos
- lfs_block_t block
- lfs_off_t off
- lfs_cache_t cache
- const struct lfs_file_config ∗ cfg

### 8.11.1 Field Documentation

#### 8.11.1.1 lfs_block_t lfs_file::block

#### 8.11.1.2 lfs_cache_t lfs_file::cache

#### 8.11.1.3 const struct lfs_file_config∗ lfs_file::cfg

#### 8.11.1.4 struct lfs_file::lfs_ctz lfs_file::ctz

#### 8.11.1.5 uint32_t lfs_file::flags

#### 8.11.1.6 uint16_t lfs_file::id

#### 8.11.1.7 lfs_mdir_t lfs_file::m

#### 8.11.1.8 struct lfs_file* lfs_file::next

#### 8.11.1.9 lfs_off_t lfs_file::off

#### 8.11.1.10 lfs_off_t lfs_file::pos

#### 8.11.1.11 uint8_t lfs_file::type

The documentation for this struct was generated from the following file:

- littlefs/lfs.h

## 8.12 lfs_file_config Struct Reference

```
#include <lfs.h>
```

### Data Fields

- void * buffer
- struct lfs_attr * attrs
- lfs_size_t attr_count

### 8.12.1 Field Documentation

#### 8.12.1.1 lfs_size_t lfs_file_config::attr_count

#### 8.12.1.2 struct lfs_attr* lfs_file_config::attrs

#### 8.12.1.3 void* lfs_file_config::buffer

The documentation for this struct was generated from the following file:

- littlefs/lfs.h

## 8.13 lfs::lfs_free Struct Reference

```
#include <lfs.h>
```

### Data Fields

- lfs_block_t off
- lfs_block_t size
- lfs_block_t i
- lfs_block_t ack
- uint32_t * buffer

### 8.13.1 Field Documentation

#### 8.13.1.1 lfs_block_t lfs::lfs_free::ack

#### 8.13.1.2 uint32_t* lfs::lfs_free::buffer

#### 8.13.1.3 lfs_block_t lfs::lfs_free::i

#### 8.13.1.4 lfs_block_t lfs::lfs_free::off

#### 8.13.1.5 lfs_block_t lfs::lfs_free::size

The documentation for this struct was generated from the following file:

- littlefs/lfs.h

## 8.14 lfs_fs_parent_match Struct Reference

### Data Fields

- lfs_t * lfs
- const lfs_block_t pair [2]

### 8.14.1 Field Documentation

#### 8.14.1.1 lfs_t* lfs_fs_parent_match::lfs

#### 8.14.1.2 const lfs_block_t lfs_fs_parent_match::pair[2]

The documentation for this struct was generated from the following file:

- littlefs/lfs.c

## 8.15 lfs_gstate Struct Reference

```
#include <lfs.h>
```

### Data Fields

- uint32_t tag
- lfs_block_t pair [2]

### 8.15.1 Field Documentation

#### 8.15.1.1 lfs_block_t lfs_gstate::pair[2]

#### 8.15.1.2 uint32_t lfs_gstate::tag

The documentation for this struct was generated from the following file:

- littlefs/lfs.h

## 8.16 lfs_info Struct Reference

```
#include <lfs.h>
```

### Data Fields

- uint8_t type
- lfs_size_t size
- char name [LFS_NAME_MAX+1]

### 8.16.1 Field Documentation

#### 8.16.1.1 char lfs_info::name[LFS_NAME_MAX+1]

#### 8.16.1.2 lfs_size_t lfs_info::size

#### 8.16.1.3 uint8_t lfs_info::type

The documentation for this struct was generated from the following file:

- littlefs/lfs.h

## 8.17 lfs_mattr Struct Reference

### Data Fields

- lfs_tag_t tag
- const void ∗ buffer

### 8.17.1 Field Documentation

#### 8.17.1.1 const void* lfs_mattr::buffer

#### 8.17.1.2 lfs_tag_t lfs_mattr::tag

The documentation for this struct was generated from the following file:

- littlefs/lfs.c

## 8.18 lfs_mdir Struct Reference

```
#include <lfs.h>
```

### Data Fields

- lfs_block_t pair [2]
- uint32_t rev
- lfs_off_t off
- uint32_t etag
- uint16_t count
- bool erased
- bool split
- lfs_block_t tail [2]

### 8.18.1 Field Documentation

#### 8.18.1.1 uint16_t lfs_mdir::count

#### 8.18.1.2 bool lfs_mdir::erased

#### 8.18.1.3 uint32_t lfs_mdir::etag

#### 8.18.1.4 lfs_off_t lfs_mdir::off

#### 8.18.1.5 lfs_block_t lfs_mdir::pair[2]

#### 8.18.1.6 uint32_t lfs_mdir::rev

#### 8.18.1.7 bool lfs_mdir::split

#### 8.18.1.8 lfs_block_t lfs_mdir::tail[2]

The documentation for this struct was generated from the following file:

- littlefs/lfs.h

## 8.19 lfs::lfs_mlist Struct Reference

```
#include <lfs.h>
```

### Data Fields

- struct lfs_mlist * next
- uint16_t id
- uint8_t type
- lfs_mdir_t m

### 8.19.1 Field Documentation

#### 8.19.1.1 uint16_t lfs::lfs_mlist::id

#### 8.19.1.2 lfs_mdir_t lfs::lfs_mlist::m

#### 8.19.1.3 struct lfs_mlist* lfs::lfs_mlist::next

#### 8.19.1.4 uint8_t lfs::lfs_mlist::type

The documentation for this struct was generated from the following file:

- littlefs/lfs.h

## 8.20 lfs_superblock Struct Reference

```
#include <lfs.h>
```

### Data Fields

- uint32_t version
- lfs_size_t block_size
- lfs_size_t block_count
- lfs_size_t name_max
- lfs_size_t file_max
- lfs_size_t attr_max

### 8.20.1 Field Documentation

#### 8.20.1.1 lfs_size_t lfs_superblock::attr_max

#### 8.20.1.2 lfs_size_t lfs_superblock::block_count

#### 8.20.1.3 lfs_size_t lfs_superblock::block_size

#### 8.20.1.4 lfs_size_t lfs_superblock::file_max

#### 8.20.1.5 lfs_size_t lfs_superblock::name_max

#### 8.20.1.6 uint32_t lfs_superblock::version

The documentation for this struct was generated from the following file:

- littlefs/lfs.h

# Chapter 9

# File Documentation

## 9.1 flash.c File Reference

Flash read/write/erase functions implementation.

```
#include "flash.h"
#include <stdio.h>
#include "icc.h"
#include "flc.h"
#include "flc_regs.h"
#include "gcr_regs.h"
```

### Functions

- int flash_read (const struct lfs_config *c, lfs_block_t block, lfs_off_t off, void *buffer, lfs_size_t size)

  *Reads flash memory.*
- int flash_write (const struct lfs_config *c, lfs_block_t block, lfs_off_t off, const void *buffer, lfs_size_t size)

  *Writes flash memory.*
- int flash_erase (const struct lfs_config *c, lfs_block_t block)

  *Erases flash memory block.*
- int flash_sync (const struct lfs_config *c)

  *Performs pending flash operations.*
- int flash_verify (uint32_t address, uint32_t length, uint8_t *data)

  *Verifies data in flash.*
- int check_mem (uint32_t startaddr, uint32_t length, uint32_t data)

  *Compares data in flash with value specified.*
- int check_erased (uint32_t startaddr, uint32_t length)

  *Checks whether flash memory is erased.*
- int flash_write4 (uint32_t startaddr, uint32_t length, uint32_t *data, bool verify)

  *Writes 32bit data words to flash.*

### 9.1.1 Detailed Description

Flash read/write/erase functions implementation.

### 9.1.2 Function Documentation

#### 9.1.2.1 int check_erased ( uint32_t *startaddr,* uint32_t *length* )

Checks whether flash memory is erased.

**Parameters**

| | |
|---|---|
| *startaddr* | Flash memory address |
| *length* | Memory block size |

**Returns**

Error code

#### 9.1.2.2 int check_mem ( uint32_t *startaddr*, uint32_t *length*, uint32_t *data* )

Compares data in flash with value specified.

**Parameters**

| | |
|---|---|
| *startaddr* | Flash memory address |
| *length* | Data size |
| *data* | The value to compare to |

**Returns**

Error code

#### 9.1.2.3 int flash_erase ( const struct lfs_config * *c*, lfs_block_t *block* )

Erases flash memory block.

**Note**

LittleFS callback method

**Parameters**

| | |
|---|---|
| *c* | LittleFS config |
| *block* | Flash memory block number |

**Returns**

Error code

#### 9.1.2.4 int flash_read ( const struct lfs_config * *c*, lfs_block_t *block*, lfs_off_t *off*, void * *buffer*, lfs_size_t *size* )

Reads flash memory.

**Note**

LittleFS callback method

**Parameters**

| | |
|---|---|
| *c* | LittleFS config |
| *block* | Flash memory block number |
| *off* | Data offset in the block |
| *buffer* | Data buffer |
| *size* | Data size |

**Returns**

Error code

#### 9.1.2.5 int flash_sync ( const struct lfs_config * *c* )

Performs pending flash operations.

**Note**

LittleFS callback method. Not supported by Maxim SDK

**Parameters**

| | |
|---|---|
| *c* | LittleFS config |

**Returns**

Error code

#### 9.1.2.6 int flash_verify ( uint32_t *address,* uint32_t *length,* uint8_t * *data* )

Verifies data in flash.

**Parameters**

| | |
|---|---|
| *address* | Flash memory address |
| *length* | Data size |
| *data* | Data buffer |

**Returns**

Error code

#### 9.1.2.7 int flash_write ( const struct lfs_config * *c,* lfs_block_t *block,* lfs_off_t *off,* const void * *buffer,* lfs_size_t *size* )

Writes flash memory.

**Note**

LittleFS callback method

**Parameters**

| | |
|---|---|
| *c* | LittleFS config |
| *block* | Flash memory block number |
| *off* | Data offset in the block |
| *buffer* | Data buffer |
| *size* | Data size |

**Returns**

Error code

#### 9.1.2.8 int flash_write4 ( uint32_t *startaddr*, uint32_t *length*, uint32_t * *data*, bool *verify* )

Writes 32bit data words to flash.

**Parameters**

| | |
|---|---|
| *startaddr* | Flash memory address |
| *length* | Data size |
| *data* | Data buffer |
| *verify* | Whether to verify written data |

**Returns**

Error code

## 9.2 flash.h File Reference

Flash read/write/erase functions declaration.

```
#include <stdint.h>
#include <stdbool.h>
#include "littlefs/lfs.h"
```

### Macros

- #define LOGF(...)

## Functions

- int flash_read (const struct lfs_config ∗c, lfs_block_t block, lfs_off_t off, void ∗buffer, lfs_size_t size)
  *Reads flash memory.*
- int flash_write (const struct lfs_config ∗c, lfs_block_t block, lfs_off_t off, const void ∗buffer, lfs_size_t size)
  *Writes flash memory.*
- int flash_erase (const struct lfs_config ∗c, lfs_block_t block)
  *Erases flash memory block.*
- int flash_sync (const struct lfs_config ∗c)
  *Performs pending flash operations.*
- int flash_verify (uint32_t address, uint32_t length, uint8_t ∗data)
  *Verifies data in flash.*
- int check_mem (uint32_t startaddr, uint32_t length, uint32_t data)
  *Compares data in flash with value specified.*
- int check_erased (uint32_t startaddr, uint32_t length)
  *Checks whether flash memory is erased.*
- int flash_write4 (uint32_t startaddr, uint32_t length, uint32_t ∗data, bool verify)
  *Writes 32bit data words to flash.*

### 9.2.1 Detailed Description

Flash read/write/erase functions declaration.

### 9.2.2 Macro Definition Documentation

#### 9.2.2.1 #define LOGF( *...* )

### 9.2.3 Function Documentation

#### 9.2.3.1 int check_erased ( uint32_t *startaddr,* uint32_t *length* )

Checks whether flash memory is erased.

**Parameters**

| | |
|---|---|
| *startaddr* | Flash memory address |
| *length* | Memory block size |

**Returns**

Error code

#### 9.2.3.2 int check_mem ( uint32_t *startaddr,* uint32_t *length,* uint32_t *data* )

Compares data in flash with value specified.

**Parameters**

| | |
|---|---|
| *startaddr* | Flash memory address |
| *length* | Data size |
| *data* | The value to compare to |

**Returns**

Error code

#### 9.2.3.3 int flash_erase ( const struct lfs_config * *c*, lfs_block_t *block* )

Erases flash memory block.

**Note**

LittleFS callback method

**Parameters**

| | |
|---|---|
| *c* | LittleFS config |
| *block* | Flash memory block number |

**Returns**

Error code

#### 9.2.3.4 int flash_read ( const struct lfs_config * *c*, lfs_block_t *block*, lfs_off_t *off*, void * *buffer*, lfs_size_t *size* )

Reads flash memory.

**Note**

LittleFS callback method

**Parameters**

| | |
|---|---|
| *c* | LittleFS config |
| *block* | Flash memory block number |
| *off* | Data offset in the block |
| *buffer* | Data buffer |
| *size* | Data size |

**Returns**

Error code

#### 9.2.3.5 int flash_sync ( const struct lfs_config * *c* )

Performs pending flash operations.

**Note**

LittleFS callback method. Not supported by Maxim SDK

**Parameters**

| | |
|---|---|
| *c* | LittleFS config |

**Returns**

Error code

#### 9.2.3.6 int flash_verify ( uint32_t *address*, uint32_t *length*, uint8_t * *data* )

Verifies data in flash.

**Parameters**

| | |
|---|---|
| *address* | Flash memory address |
| *length* | Data size |
| *data* | Data buffer |

**Returns**

Error code

#### 9.2.3.7 int flash_write ( const struct lfs_config * *c*, lfs_block_t *block*, lfs_off_t *off*, const void * *buffer*, lfs_size_t *size* )

Writes flash memory.

**Note**

LittleFS callback method

**Parameters**

| | |
|---|---|
| *c* | LittleFS config |
| *block* | Flash memory block number |
| *off* | Data offset in the block |
| *buffer* | Data buffer |
| *size* | Data size |

**Returns**

Error code

#### 9.2.3.8 int flash_write4 ( uint32_t *startaddr,* uint32_t *length,* uint32_t ∗ *data,* bool *verify* )

Writes 32bit data words to flash.

**Parameters**

| | |
|---|---|
| *startaddr* | Flash memory address |
| *length* | Data size |
| *data* | Data buffer |
| *verify* | Whether to verify written data |

**Returns**

Error code

## 9.3 littlefs/DESIGN.md File Reference

## 9.4 littlefs/lfs.c File Reference

```
#include "lfs.h"
#include "lfs_util.h"
```

### Data Structures

- struct lfs_mattr
- struct lfs_diskoff
- struct lfs_dir_find_match
- struct lfs_commit
- struct lfs_dir_commit_commit
- struct lfs_fs_parent_match

### Macros

- #define LFS_BLOCK_NULL ((lfs_block_t)-1)
- #define LFS_BLOCK_INLINE ((lfs_block_t)-2)
- #define LFS_MKTAG(type, id, size) (((lfs_tag_t)(type) << 20) | ((lfs_tag_t)(id) << 10) | (lfs_tag_t)(size))
- #define LFS_MKTAG_IF(cond, type, id, size) ((cond) ? LFS_MKTAG(type, id, size) : LFS_MKTAG(LFS_F↩ ROM_NOOP, 0, 0))
- #define LFS_MKTAG_IF_ELSE(cond, type1, id1, size1, type2, id2, size2) ((cond) ? LFS_MKTAG(type1, id1, size1) : LFS_MKTAG(type2, id2, size2))
- #define LFS_MKATTRS(...)
- #define LFS_LOCK(cfg) ((void)cfg, 0)

  *Public API wrappers ///.*
- #define LFS_UNLOCK(cfg) ((void)cfg)

## Typedefs

- typedef uint32_t lfs_tag_t
- typedef int32_t lfs_stag_t

## Enumerations

- enum { LFS_CMP_EQ = 0, LFS_CMP_LT = 1, LFS_CMP_GT = 2 }

## Functions

- static void lfs_cache_drop (lfs_t *lfs, lfs_cache_t *rcache)

  *Caching block device operations ///.*
- static void lfs_cache_zero (lfs_t *lfs, lfs_cache_t *pcache)
- static int lfs_bd_read (lfs_t *lfs, const lfs_cache_t *pcache, lfs_cache_t *rcache, lfs_size_t hint, lfs_block_t block, lfs_off_t off, void *buffer, lfs_size_t size)
- static int lfs_bd_cmp (lfs_t *lfs, const lfs_cache_t *pcache, lfs_cache_t *rcache, lfs_size_t hint, lfs_block_t block, lfs_off_t off, const void *buffer, lfs_size_t size)
- static int lfs_bd_flush (lfs_t *lfs, lfs_cache_t *pcache, lfs_cache_t *rcache, bool validate)
- static int lfs_bd_sync (lfs_t *lfs, lfs_cache_t *pcache, lfs_cache_t *rcache, bool validate)
- static int lfs_bd_prog (lfs_t *lfs, lfs_cache_t *pcache, lfs_cache_t *rcache, bool validate, lfs_block_t block, lfs_off_t off, const void *buffer, lfs_size_t size)
- static int lfs_bd_erase (lfs_t *lfs, lfs_block_t block)
- static void lfs_pair_swap (lfs_block_t pair[2])

  *Small type-level utilities ///.*
- static bool lfs_pair_isnull (const lfs_block_t pair[2])
- static int lfs_pair_cmp (const lfs_block_t paira[2], const lfs_block_t pairb[2])
- static bool lfs_pair_sync (const lfs_block_t paira[2], const lfs_block_t pairb[2])
- static void lfs_pair_fromle32 (lfs_block_t pair[2])
- static void lfs_pair_tole32 (lfs_block_t pair[2])
- static bool lfs_tag_isvalid (lfs_tag_t tag)
- static bool lfs_tag_isdelete (lfs_tag_t tag)
- static uint16_t lfs_tag_type1 (lfs_tag_t tag)
- static uint16_t lfs_tag_type3 (lfs_tag_t tag)
- static uint8_t lfs_tag_chunk (lfs_tag_t tag)
- static int8_t lfs_tag_splice (lfs_tag_t tag)
- static uint16_t lfs_tag_id (lfs_tag_t tag)
- static lfs_size_t lfs_tag_size (lfs_tag_t tag)
- static lfs_size_t lfs_tag_dsize (lfs_tag_t tag)
- static void lfs_gstate_xor (lfs_gstate_t *a, const lfs_gstate_t *b)
- static bool lfs_gstate_iszero (const lfs_gstate_t *a)
- static bool lfs_gstate_hasorphans (const lfs_gstate_t *a)
- static uint8_t lfs_gstate_getorphans (const lfs_gstate_t *a)
- static bool lfs_gstate_hasmove (const lfs_gstate_t *a)
- static bool lfs_gstate_hasmovehere (const lfs_gstate_t *a, const lfs_block_t *pair)
- static void lfs_gstate_fromle32 (lfs_gstate_t *a)
- static void lfs_gstate_tole32 (lfs_gstate_t *a)
- static void lfs_ctz_fromle32 (struct lfs_ctz *ctz)
- static void lfs_ctz_tole32 (struct lfs_ctz *ctz)
- static void lfs_superblock_fromle32 (lfs_superblock_t *superblock)
- static void lfs_superblock_tole32 (lfs_superblock_t *superblock)
- static bool lfs_mlist_isopen (struct lfs_mlist *head, struct lfs_mlist *node)
- static void lfs_mlist_remove (lfs_t *lfs, struct lfs_mlist *mlist)

- static void lfs_mlist_append (lfs_t *lfs, struct lfs_mlist *mlist)
- static int lfs_dir_commit (lfs_t *lfs, lfs_mdir_t *dir, const struct lfs_mattr *attrs, int attrcount)

  *Internal operations predeclared here ///.*
- static int lfs_dir_compact (lfs_t *lfs, lfs_mdir_t *dir, const struct lfs_mattr *attrs, int attrcount, lfs_mdir_←
t *source, uint16_t begin, uint16_t end)
- static lfs_ssize_t lfs_file_rawwrite (lfs_t *lfs, lfs_file_t *file, const void *buffer, lfs_size_t size)
- static int lfs_file_rawsync (lfs_t *lfs, lfs_file_t *file)
- static int lfs_file_outline (lfs_t *lfs, lfs_file_t *file)
- static int lfs_file_flush (lfs_t *lfs, lfs_file_t *file)
- static int lfs_fs_preporphans (lfs_t *lfs, int8_t orphans)
- static void lfs_fs_prepmove (lfs_t *lfs, uint16_t id, const lfs_block_t pair[2])
- static int lfs_fs_pred (lfs_t *lfs, const lfs_block_t dir[2], lfs_mdir_t *pdir)
- static lfs_stag_t lfs_fs_parent (lfs_t *lfs, const lfs_block_t dir[2], lfs_mdir_t *parent)
- static int lfs_fs_relocate (lfs_t *lfs, const lfs_block_t oldpair[2], lfs_block_t newpair[2])
- static int lfs_fs_forceconsistency (lfs_t *lfs)
- static int lfs_dir_rawrewind (lfs_t *lfs, lfs_dir_t *dir)
- static lfs_ssize_t lfs_file_rawread (lfs_t *lfs, lfs_file_t *file, void *buffer, lfs_size_t size)
- static int lfs_file_rawclose (lfs_t *lfs, lfs_file_t *file)
- static lfs_soff_t lfs_file_rawsize (lfs_t *lfs, lfs_file_t *file)
- static lfs_ssize_t lfs_fs_rawsize (lfs_t *lfs)
- static int lfs_fs_rawtraverse (lfs_t *lfs, int(*cb)(void *data, lfs_block_t block), void *data, bool includeorphans)

  *Filesystem filesystem operations ///.*
- static int lfs_deinit (lfs_t *lfs)
- static int lfs_rawunmount (lfs_t *lfs)
- static int lfs_alloc_lookahead (void *p, lfs_block_t block)

  *Block allocator ///.*
- static void lfs_alloc_ack (lfs_t *lfs)
- static void lfs_alloc_drop (lfs_t *lfs)
- static int lfs_alloc (lfs_t *lfs, lfs_block_t *block)
- static lfs_stag_t lfs_dir_getslice (lfs_t *lfs, const lfs_mdir_t *dir, lfs_tag_t gmask, lfs_tag_t gtag, lfs_off_t goff, void *gbuffer, lfs_size_t gsize)

  *Metadata pair and directory operations ///.*
- static lfs_stag_t lfs_dir_get (lfs_t *lfs, const lfs_mdir_t *dir, lfs_tag_t gmask, lfs_tag_t gtag, void *buffer)
- static int lfs_dir_getread (lfs_t *lfs, const lfs_mdir_t *dir, const lfs_cache_t *pcache, lfs_cache_t *rcache, lfs_size_t hint, lfs_tag_t gmask, lfs_tag_t gtag, lfs_off_t off, void *buffer, lfs_size_t size)
- static int lfs_dir_traverse_filter (void *p, lfs_tag_t tag, const void *buffer)
- static int lfs_dir_traverse (lfs_t *lfs, const lfs_mdir_t *dir, lfs_off_t off, lfs_tag_t ptag, const struct lfs_mattr *attrs, int attrcount, lfs_tag_t tmask, lfs_tag_t ttag, uint16_t begin, uint16_t end, int16_t diff, int(*cb)(void *data, lfs_tag_t tag, const void *buffer), void *data)
- static lfs_stag_t lfs_dir_fetchmatch (lfs_t *lfs, lfs_mdir_t *dir, const lfs_block_t pair[2], lfs_tag_t fmask, lfs_←
tag_t ftag, uint16_t *id, int(*cb)(void *data, lfs_tag_t tag, const void *buffer), void *data)
- static int lfs_dir_fetch (lfs_t *lfs, lfs_mdir_t *dir, const lfs_block_t pair[2])
- static int lfs_dir_getgstate (lfs_t *lfs, const lfs_mdir_t *dir, lfs_gstate_t *gstate)
- static int lfs_dir_getinfo (lfs_t *lfs, lfs_mdir_t *dir, uint16_t id, struct lfs_info *info)
- static int lfs_dir_find_match (void *data, lfs_tag_t tag, const void *buffer)
- static lfs_stag_t lfs_dir_find (lfs_t *lfs, lfs_mdir_t *dir, const char **path, uint16_t *id)
- static int lfs_dir_commitprog (lfs_t *lfs, struct lfs_commit *commit, const void *buffer, lfs_size_t size)
- static int lfs_dir_commitattr (lfs_t *lfs, struct lfs_commit *commit, lfs_tag_t tag, const void *buffer)
- static int lfs_dir_commitcrc (lfs_t *lfs, struct lfs_commit *commit)
- static int lfs_dir_alloc (lfs_t *lfs, lfs_mdir_t *dir)
- static int lfs_dir_drop (lfs_t *lfs, lfs_mdir_t *dir, lfs_mdir_t *tail)
- static int lfs_dir_split (lfs_t *lfs, lfs_mdir_t *dir, const struct lfs_mattr *attrs, int attrcount, lfs_mdir_t *source, uint16_t split, uint16_t end)
- static int lfs_dir_commit_size (void *p, lfs_tag_t tag, const void *buffer)

- static int lfs_dir_commit_commit (void ∗p, lfs_tag_t tag, const void ∗buffer)
- static int lfs_rawmkdir (lfs_t ∗lfs, const char ∗path)

  *Top level directory operations ///.*
- static int lfs_dir_rawopen (lfs_t ∗lfs, lfs_dir_t ∗dir, const char ∗path)
- static int lfs_dir_rawclose (lfs_t ∗lfs, lfs_dir_t ∗dir)
- static int lfs_dir_rawread (lfs_t ∗lfs, lfs_dir_t ∗dir, struct lfs_info ∗info)
- static int lfs_dir_rawseek (lfs_t ∗lfs, lfs_dir_t ∗dir, lfs_off_t off)
- static lfs_soff_t lfs_dir_rawtell (lfs_t ∗lfs, lfs_dir_t ∗dir)
- static int lfs_ctz_index (lfs_t ∗lfs, lfs_off_t ∗off)

  *File index list operations ///.*
- static int lfs_ctz_find (lfs_t ∗lfs, const lfs_cache_t ∗pcache, lfs_cache_t ∗rcache, lfs_block_t head, lfs_size_t size, lfs_size_t pos, lfs_block_t ∗block, lfs_off_t ∗off)
- static int lfs_ctz_extend (lfs_t ∗lfs, lfs_cache_t ∗pcache, lfs_cache_t ∗rcache, lfs_block_t head, lfs_size_t size, lfs_block_t ∗block, lfs_off_t ∗off)
- static int lfs_ctz_traverse (lfs_t ∗lfs, const lfs_cache_t ∗pcache, lfs_cache_t ∗rcache, lfs_block_t head, lfs_size_t size, int(∗cb)(void ∗, lfs_block_t), void ∗data)
- static int lfs_file_rawopencfg (lfs_t ∗lfs, lfs_file_t ∗file, const char ∗path, int flags, const struct lfs_file_config ∗cfg)

  *Top level file operations ///.*
- static int lfs_file_rawopen (lfs_t ∗lfs, lfs_file_t ∗file, const char ∗path, int flags)
- static int lfs_file_relocate (lfs_t ∗lfs, lfs_file_t ∗file)
- static lfs_soff_t lfs_file_rawseek (lfs_t ∗lfs, lfs_file_t ∗file, lfs_soff_t off, int whence)
- static int lfs_file_rawtruncate (lfs_t ∗lfs, lfs_file_t ∗file, lfs_off_t size)
- static lfs_soff_t lfs_file_rawtell (lfs_t ∗lfs, lfs_file_t ∗file)
- static int lfs_file_rawrewind (lfs_t ∗lfs, lfs_file_t ∗file)
- static int lfs_rawstat (lfs_t ∗lfs, const char ∗path, struct lfs_info ∗info)

  *General fs operations ///.*
- static int lfs_rawremove (lfs_t ∗lfs, const char ∗path)
- static int lfs_rawrename (lfs_t ∗lfs, const char ∗oldpath, const char ∗newpath)
- static lfs_ssize_t lfs_rawgetattr (lfs_t ∗lfs, const char ∗path, uint8_t type, void ∗buffer, lfs_size_t size)
- static int lfs_commitattr (lfs_t ∗lfs, const char ∗path, uint8_t type, const void ∗buffer, lfs_size_t size)
- static int lfs_rawsetattr (lfs_t ∗lfs, const char ∗path, uint8_t type, const void ∗buffer, lfs_size_t size)
- static int lfs_rawremoveattr (lfs_t ∗lfs, const char ∗path, uint8_t type)
- static int lfs_init (lfs_t ∗lfs, const struct lfs_config ∗cfg)

  *Filesystem operations ///.*
- static int lfs_rawformat (lfs_t ∗lfs, const struct lfs_config ∗cfg)
- static int lfs_rawmount (lfs_t ∗lfs, const struct lfs_config ∗cfg)
- static int lfs_fs_parent_match (void ∗data, lfs_tag_t tag, const void ∗buffer)
- static int lfs_fs_demove (lfs_t ∗lfs)
- static int lfs_fs_deorphan (lfs_t ∗lfs)
- static int lfs_fs_size_count (void ∗p, lfs_block_t block)
- int lfs_format (lfs_t ∗lfs, const struct lfs_config ∗cfg)

  *Filesystem functions ///.*
- int lfs_mount (lfs_t ∗lfs, const struct lfs_config ∗cfg)
- int lfs_unmount (lfs_t ∗lfs)
- int lfs_remove (lfs_t ∗lfs, const char ∗path)

  *General operations ///.*
- int lfs_rename (lfs_t ∗lfs, const char ∗oldpath, const char ∗newpath)
- int lfs_stat (lfs_t ∗lfs, const char ∗path, struct lfs_info ∗info)
- lfs_ssize_t lfs_getattr (lfs_t ∗lfs, const char ∗path, uint8_t type, void ∗buffer, lfs_size_t size)
- int lfs_setattr (lfs_t ∗lfs, const char ∗path, uint8_t type, const void ∗buffer, lfs_size_t size)
- int lfs_removeattr (lfs_t ∗lfs, const char ∗path, uint8_t type)
- int lfs_file_open (lfs_t ∗lfs, lfs_file_t ∗file, const char ∗path, int flags)

  *File operations ///.*

- int lfs_file_opencfg (lfs_t ∗lfs, lfs_file_t ∗file, const char ∗path, int flags, const struct lfs_file_config ∗cfg)
- int lfs_file_close (lfs_t ∗lfs, lfs_file_t ∗file)
- int lfs_file_sync (lfs_t ∗lfs, lfs_file_t ∗file)
- lfs_ssize_t lfs_file_read (lfs_t ∗lfs, lfs_file_t ∗file, void ∗buffer, lfs_size_t size)
- lfs_ssize_t lfs_file_write (lfs_t ∗lfs, lfs_file_t ∗file, const void ∗buffer, lfs_size_t size)
- lfs_soff_t lfs_file_seek (lfs_t ∗lfs, lfs_file_t ∗file, lfs_soff_t off, int whence)
- int lfs_file_truncate (lfs_t ∗lfs, lfs_file_t ∗file, lfs_off_t size)
- lfs_soff_t lfs_file_tell (lfs_t ∗lfs, lfs_file_t ∗file)
- int lfs_file_rewind (lfs_t ∗lfs, lfs_file_t ∗file)
- lfs_soff_t lfs_file_size (lfs_t ∗lfs, lfs_file_t ∗file)
- int lfs_mkdir (lfs_t ∗lfs, const char ∗path)

  *Directory operations ///.*
- int lfs_dir_open (lfs_t ∗lfs, lfs_dir_t ∗dir, const char ∗path)
- int lfs_dir_close (lfs_t ∗lfs, lfs_dir_t ∗dir)
- int lfs_dir_read (lfs_t ∗lfs, lfs_dir_t ∗dir, struct lfs_info ∗info)
- int lfs_dir_seek (lfs_t ∗lfs, lfs_dir_t ∗dir, lfs_off_t off)
- lfs_soff_t lfs_dir_tell (lfs_t ∗lfs, lfs_dir_t ∗dir)
- int lfs_dir_rewind (lfs_t ∗lfs, lfs_dir_t ∗dir)
- lfs_ssize_t lfs_fs_size (lfs_t ∗lfs)

  *Filesystem-level filesystem operations.*
- int lfs_fs_traverse (lfs_t ∗lfs, int(∗cb)(void ∗, lfs_block_t), void ∗data)

### 9.4.1 Macro Definition Documentation

#### 9.4.1.1 #define LFS_BLOCK_INLINE ((lfs_block_t)-2)

#### 9.4.1.2 #define LFS_BLOCK_NULL ((lfs_block_t)-1)

#### 9.4.1.3 #define LFS_LOCK( cfg ) ((void)cfg, 0)

Public API wrappers ///.

#### 9.4.1.4 #define LFS_MKATTRS( ... )

**Value:**

```
(struct lfs_mattr[]){__VA_ARGS__}, \
    sizeof((struct lfs_mattr[]){__VA_ARGS__}) / sizeof(struct lfs_mattr)
```

#### 9.4.1.5 #define LFS_MKTAG( *type,* *id,* *size* ) (((lfs_tag_t)(type) << 20) | ((lfs_tag_t)(id) << 10) | (lfs_tag_t)(size))

#### 9.4.1.6 #define LFS_MKTAG_IF( *cond,* *type,* *id,* *size* ) ((cond) ? LFS_MKTAG(type, id, size) : LFS_MKTAG(LFS_FROM_NOOP, 0, 0))

#### 9.4.1.7 #define LFS_MKTAG_IF_ELSE( *cond,* *type1,* *id1,* *size1,* *type2,* *id2,* *size2* ) ((cond) ? LFS_MKTAG(type1, id1, size1) : LFS_MKTAG(type2, id2, size2))

#### 9.4.1.8 #define LFS_UNLOCK( cfg ) ((void)cfg)

### 9.4.2 Typedef Documentation

#### 9.4.2.1 typedef int32_t lfs_stag_t

#### 9.4.2.2 typedef uint32_t lfs_tag_t

### 9.4.3 Enumeration Type Documentation

#### 9.4.3.1 anonymous enum

**Enumerator**

***LFS_CMP_EQ***
***LFS_CMP_LT***
***LFS_CMP_GT***

### 9.4.4 Function Documentation

#### 9.4.4.1 static int lfs_alloc ( lfs_t * *lfs,* lfs_block_t * *block* ) [static]

#### 9.4.4.2 static void lfs_alloc_ack ( lfs_t * *lfs* ) [static]

#### 9.4.4.3 static void lfs_alloc_drop ( lfs_t * *lfs* ) [static]

#### 9.4.4.4 static int lfs_alloc_lookahead ( void * *p,* lfs_block_t *block* ) [static]

Block allocator ///.

#### 9.4.4.5 static int lfs_bd_cmp ( lfs_t * *lfs,* const lfs_cache_t * *pcache,* lfs_cache_t * *rcache,* lfs_size_t *hint,* lfs_block_t *block,* lfs_off_t *off,* const void * *buffer,* lfs_size_t *size* ) [static]

#### 9.4.4.6 static int lfs_bd_erase ( lfs_t * *lfs,* lfs_block_t *block* ) [static]

#### 9.4.4.7 static int lfs_bd_flush ( lfs_t * *lfs,* lfs_cache_t * *pcache,* lfs_cache_t * *rcache,* bool *validate* ) [static]

#### 9.4.4.8 static int lfs_bd_prog ( lfs_t * *lfs,* lfs_cache_t * *pcache,* lfs_cache_t * *rcache,* bool *validate,* lfs_block_t *block,* lfs_off_t *off,* const void * *buffer,* lfs_size_t *size* ) [static]

#### 9.4.4.9 static int lfs_bd_read ( lfs_t * *lfs,* const lfs_cache_t * *pcache,* lfs_cache_t * *rcache,* lfs_size_t *hint,* lfs_block_t *block,* lfs_off_t *off,* void * *buffer,* lfs_size_t *size* ) [static]

#### 9.4.4.10 static int lfs_bd_sync ( lfs_t * *lfs,* lfs_cache_t * *pcache,* lfs_cache_t * *rcache,* bool *validate* ) [static]

#### 9.4.4.11 static void lfs_cache_drop ( lfs_t * *lfs,* lfs_cache_t * *rcache* ) [inline], [static]

Caching block device operations ///.

#### 9.4.4.12 static void lfs_cache_zero ( lfs_t * lfs, lfs_cache_t * pcache ) [inline], [static]

#### 9.4.4.13 static int lfs_commitattr ( lfs_t * lfs, const char * path, uint8_t type, const void * buffer, lfs_size_t size ) [static]

#### 9.4.4.14 static int lfs_ctz_extend ( lfs_t * lfs, lfs_cache_t * pcache, lfs_cache_t * rcache, lfs_block_t head, lfs_size_t size, lfs_block_t * block, lfs_off_t * off ) [static]

#### 9.4.4.15 static int lfs_ctz_find ( lfs_t * lfs, const lfs_cache_t * pcache, lfs_cache_t * rcache, lfs_block_t head, lfs_size_t size, lfs_size_t pos, lfs_block_t * block, lfs_off_t * off ) [static]

#### 9.4.4.16 static void lfs_ctz_fromle32 ( struct lfs_ctz * ctz ) [static]

#### 9.4.4.17 static int lfs_ctz_index ( lfs_t * lfs, lfs_off_t * off ) [static]

File index list operations ///.

#### 9.4.4.18 static void lfs_ctz_tole32 ( struct lfs_ctz * ctz ) [static]

#### 9.4.4.19 static int lfs_ctz_traverse ( lfs_t * lfs, const lfs_cache_t * pcache, lfs_cache_t * rcache, lfs_block_t head, lfs_size_t size, int(*)(void *, lfs_block_t) cb, void * data ) [static]

#### 9.4.4.20 static int lfs_deinit ( lfs_t * lfs ) [static]

#### 9.4.4.21 static int lfs_dir_alloc ( lfs_t * lfs, lfs_mdir_t * dir ) [static]

#### 9.4.4.22 int lfs_dir_close ( lfs_t * lfs, lfs_dir_t * dir )

#### 9.4.4.23 static int lfs_dir_commit ( lfs_t * lfs, lfs_mdir_t * dir, const struct lfs_mattr * attrs, int attrcount ) [static]

Internal operations predeclared here ///.

#### 9.4.4.24 static int lfs_dir_commit_commit ( void * p, lfs_tag_t tag, const void * buffer ) [static]

#### 9.4.4.25 static int lfs_dir_commit_size ( void * p, lfs_tag_t tag, const void * buffer ) [static]

#### 9.4.4.26 static int lfs_dir_commitattr ( lfs_t * lfs, struct lfs_commit * commit, lfs_tag_t tag, const void * buffer ) [static]

#### 9.4.4.27 static int lfs_dir_commitcrc ( lfs_t * lfs, struct lfs_commit * commit ) [static]

#### 9.4.4.28 static int lfs_dir_commitprog ( lfs_t * lfs, struct lfs_commit * commit, const void * buffer, lfs_size_t size ) [static]

#### 9.4.4.29 static int lfs_dir_compact ( lfs_t * lfs, lfs_mdir_t * dir, const struct lfs_mattr * attrs, int attrcount, lfs_mdir_t * source, uint16_t begin, uint16_t end ) [static]

#### 9.4.4.30 static int lfs_dir_drop ( lfs_t * *lfs*, lfs_mdir_t * *dir*, lfs_mdir_t * *tail* ) [static]

#### 9.4.4.31 static int lfs_dir_fetch ( lfs_t * *lfs*, lfs_mdir_t * *dir*, const lfs_block_t *pair[2]* ) [static]

#### 9.4.4.32 static lfs_stag_t lfs_dir_fetchmatch ( lfs_t * *lfs*, lfs_mdir_t * *dir*, const lfs_block_t *pair[2]*, lfs_tag_t *fmask*, lfs_tag_t *ftag*, uint16_t * *id*, int(*)(void *data, lfs_tag_t tag, const void *buffer) *cb*, void * *data* ) [static]

#### 9.4.4.33 static lfs_stag_t lfs_dir_find ( lfs_t * *lfs*, lfs_mdir_t * *dir*, const char ** *path*, uint16_t * *id* ) [static]

#### 9.4.4.34 static int lfs_dir_find_match ( void * *data*, lfs_tag_t *tag*, const void * *buffer* ) [static]

#### 9.4.4.35 static lfs_stag_t lfs_dir_get ( lfs_t * *lfs*, const lfs_mdir_t * *dir*, lfs_tag_t *gmask*, lfs_tag_t *gtag*, void * *buffer* ) [static]

#### 9.4.4.36 static int lfs_dir_getgstate ( lfs_t * *lfs*, const lfs_mdir_t * *dir*, lfs_gstate_t * *gstate* ) [static]

#### 9.4.4.37 static int lfs_dir_getinfo ( lfs_t * *lfs*, lfs_mdir_t * *dir*, uint16_t *id*, struct lfs_info * *info* ) [static]

#### 9.4.4.38 static int lfs_dir_getread ( lfs_t * *lfs*, const lfs_mdir_t * *dir*, const lfs_cache_t * *pcache*, lfs_cache_t * *rcache*, lfs_size_t *hint*, lfs_tag_t *gmask*, lfs_tag_t *gtag*, lfs_off_t *off*, void * *buffer*, lfs_size_t *size* ) [static]

#### 9.4.4.39 static lfs_stag_t lfs_dir_getslice ( lfs_t * *lfs*, const lfs_mdir_t * *dir*, lfs_tag_t *gmask*, lfs_tag_t *gtag*, lfs_off_t *goff*, void * *gbuffer*, lfs_size_t *gsize* ) [static]

Metadata pair and directory operations ///.

#### 9.4.4.40 int lfs_dir_open ( lfs_t * *lfs*, lfs_dir_t * *dir*, const char * *path* )

#### 9.4.4.41 static int lfs_dir_rawclose ( lfs_t * *lfs*, lfs_dir_t * *dir* ) [static]

#### 9.4.4.42 static int lfs_dir_rawopen ( lfs_t * *lfs*, lfs_dir_t * *dir*, const char * *path* ) [static]

#### 9.4.4.43 static int lfs_dir_rawread ( lfs_t * *lfs*, lfs_dir_t * *dir*, struct lfs_info * *info* ) [static]

#### 9.4.4.44 static int lfs_dir_rawrewind ( lfs_t * *lfs*, lfs_dir_t * *dir* ) [static]

#### 9.4.4.45 static int lfs_dir_rawseek ( lfs_t * *lfs*, lfs_dir_t * *dir*, lfs_off_t *off* ) [static]

#### 9.4.4.46 static lfs_soff_t lfs_dir_rawtell ( lfs_t * *lfs*, lfs_dir_t * *dir* ) [static]

#### 9.4.4.47 int lfs_dir_read ( lfs_t * *lfs*, lfs_dir_t * *dir*, struct lfs_info * *info* )

#### 9.4.4.48 int lfs_dir_rewind ( lfs_t * *lfs*, lfs_dir_t * *dir* )

#### 9.4.4.49 int lfs_dir_seek ( lfs_t * *lfs*, lfs_dir_t * *dir*, lfs_off_t *off* )

#### 9.4.4.50 static int lfs_dir_split ( lfs_t * lfs, lfs_mdir_t * dir, const struct lfs_mattr * attrs, int attrcount, lfs_mdir_t * source, uint16_t split, uint16_t end ) [static]

#### 9.4.4.51 lfs_soff_t lfs_dir_tell ( lfs_t * lfs, lfs_dir_t * dir )

#### 9.4.4.52 static int lfs_dir_traverse ( lfs_t * lfs, const lfs_mdir_t * dir, lfs_off_t off, lfs_tag_t ptag, const struct lfs_mattr * attrs, int attrcount, lfs_tag_t tmask, lfs_tag_t ttag, uint16_t begin, uint16_t end, int16_t diff, int(*)(void *data, lfs_tag_t tag, const void *buffer) cb, void * data ) [static]

#### 9.4.4.53 static int lfs_dir_traverse_filter ( void * p, lfs_tag_t tag, const void * buffer ) [static]

#### 9.4.4.54 int lfs_file_close ( lfs_t * lfs, lfs_file_t * file )

#### 9.4.4.55 static int lfs_file_flush ( lfs_t * lfs, lfs_file_t * file ) [static]

#### 9.4.4.56 int lfs_file_open ( lfs_t * lfs, lfs_file_t * file, const char * path, int flags )

File operations ///.

#### 9.4.4.57 int lfs_file_opencfg ( lfs_t * lfs, lfs_file_t * file, const char * path, int flags, const struct lfs_file_config * cfg )

#### 9.4.4.58 static int lfs_file_outline ( lfs_t * lfs, lfs_file_t * file ) [static]

#### 9.4.4.59 static int lfs_file_rawclose ( lfs_t * lfs, lfs_file_t * file ) [static]

#### 9.4.4.60 static int lfs_file_rawopen ( lfs_t * lfs, lfs_file_t * file, const char * path, int flags ) [static]

#### 9.4.4.61 static int lfs_file_rawopencfg ( lfs_t * lfs, lfs_file_t * file, const char * path, int flags, const struct lfs_file_config * cfg ) [static]

Top level file operations ///.

#### 9.4.4.62 static lfs_ssize_t lfs_file_rawread ( lfs_t * lfs, lfs_file_t * file, void * buffer, lfs_size_t size ) [static]

#### 9.4.4.63 static int lfs_file_rawrewind ( lfs_t * lfs, lfs_file_t * file ) [static]

#### 9.4.4.64 static lfs_soff_t lfs_file_rawseek ( lfs_t * lfs, lfs_file_t * file, lfs_soff_t off, int whence ) [static]

#### 9.4.4.65 static lfs_soff_t lfs_file_rawsize ( lfs_t * lfs, lfs_file_t * file ) [static]

#### 9.4.4.66 static int lfs_file_rawsync ( lfs_t * lfs, lfs_file_t * file ) [static]

#### 9.4.4.67 static lfs_soff_t lfs_file_rawtell ( lfs_t * lfs, lfs_file_t * file ) [static]

#### 9.4.4.68 static int lfs_file_rawtruncate ( lfs_t * lfs, lfs_file_t * file, lfs_off_t size ) [static]

#### 9.4.4.69 static lfs_ssize_t lfs_file_rawwrite ( lfs_t * *lfs*, lfs_file_t * *file*, const void * *buffer*, lfs_size_t *size* ) [static]

#### 9.4.4.70 lfs_ssize_t lfs_file_read ( lfs_t * *lfs*, lfs_file_t * *file*, void * *buffer*, lfs_size_t *size* )

#### 9.4.4.71 static int lfs_file_relocate ( lfs_t * *lfs*, lfs_file_t * *file* ) [static]

#### 9.4.4.72 int lfs_file_rewind ( lfs_t * *lfs*, lfs_file_t * *file* )

#### 9.4.4.73 lfs_soff_t lfs_file_seek ( lfs_t * *lfs*, lfs_file_t * *file*, lfs_soff_t *off*, int *whence* )

#### 9.4.4.74 lfs_soff_t lfs_file_size ( lfs_t * *lfs*, lfs_file_t * *file* )

#### 9.4.4.75 int lfs_file_sync ( lfs_t * *lfs*, lfs_file_t * *file* )

#### 9.4.4.76 lfs_soff_t lfs_file_tell ( lfs_t * *lfs*, lfs_file_t * *file* )

#### 9.4.4.77 int lfs_file_truncate ( lfs_t * *lfs*, lfs_file_t * *file*, lfs_off_t *size* )

#### 9.4.4.78 lfs_ssize_t lfs_file_write ( lfs_t * *lfs*, lfs_file_t * *file*, const void * *buffer*, lfs_size_t *size* )

#### 9.4.4.79 int lfs_format ( lfs_t * *lfs*, const struct lfs_config * *cfg* )

Filesystem functions ///.

#### 9.4.4.80 static int lfs_fs_demove ( lfs_t * *lfs* ) [static]

#### 9.4.4.81 static int lfs_fs_deorphan ( lfs_t * *lfs* ) [static]

#### 9.4.4.82 static int lfs_fs_forceconsistency ( lfs_t * *lfs* ) [static]

#### 9.4.4.83 static lfs_stag_t lfs_fs_parent ( lfs_t * *lfs*, const lfs_block_t *dir[2]*, lfs_mdir_t * *parent* ) [static]

#### 9.4.4.84 static int lfs_fs_parent_match ( void * *data*, lfs_tag_t *tag*, const void * *buffer* ) [static]

#### 9.4.4.85 static int lfs_fs_pred ( lfs_t * *lfs*, const lfs_block_t *dir[2]*, lfs_mdir_t * *pdir* ) [static]

#### 9.4.4.86 static void lfs_fs_prepmove ( lfs_t * *lfs*, uint16_t *id*, const lfs_block_t *pair[2]* ) [static]

#### 9.4.4.87 static int lfs_fs_preporphans ( lfs_t * *lfs*, int8_t *orphans* ) [static]

#### 9.4.4.88 static lfs_ssize_t lfs_fs_rawsize ( lfs_t * *lfs* ) [static]

#### 9.4.4.89 int lfs_fs_rawtraverse ( lfs_t * *lfs*, int(*)(void *data, lfs_block_t block) *cb*, void * *data*, bool *includeorphans* ) [static]

Filesystem filesystem operations ///.

#### 9.4.4.90 static int lfs_fs_relocate ( lfs_t * lfs, const lfs_block_t oldpair[2], lfs_block_t newpair[2] ) [static]

#### 9.4.4.91 lfs_ssize_t lfs_fs_size ( lfs_t * lfs )

Filesystem-level filesystem operations.

#### 9.4.4.92 static int lfs_fs_size_count ( void * p, lfs_block_t block ) [static]

#### 9.4.4.93 int lfs_fs_traverse ( lfs_t * lfs, int(*)(void *, lfs_block_t) cb, void * data )

#### 9.4.4.94 lfs_ssize_t lfs_getattr ( lfs_t * lfs, const char * path, uint8_t type, void * buffer, lfs_size_t size )

#### 9.4.4.95 static void lfs_gstate_fromle32 ( lfs_gstate_t * a ) [inline], [static]

#### 9.4.4.96 static uint8_t lfs_gstate_getorphans ( const lfs_gstate_t * a ) [inline], [static]

#### 9.4.4.97 static bool lfs_gstate_hasmove ( const lfs_gstate_t * a ) [inline], [static]

#### 9.4.4.98 static bool lfs_gstate_hasmovehere ( const lfs_gstate_t * a, const lfs_block_t * pair ) [inline], [static]

#### 9.4.4.99 static bool lfs_gstate_hasorphans ( const lfs_gstate_t * a ) [inline], [static]

#### 9.4.4.100 static bool lfs_gstate_iszero ( const lfs_gstate_t * a ) [inline], [static]

#### 9.4.4.101 static void lfs_gstate_tole32 ( lfs_gstate_t * a ) [inline], [static]

#### 9.4.4.102 static void lfs_gstate_xor ( lfs_gstate_t * a, const lfs_gstate_t * b ) [inline], [static]

#### 9.4.4.103 static int lfs_init ( lfs_t * lfs, const struct lfs_config * cfg ) [static]

Filesystem operations ///.

#### 9.4.4.104 int lfs_mkdir ( lfs_t * lfs, const char * path )

Directory operations ///.

#### 9.4.4.105 static void lfs_mlist_append ( lfs_t * lfs, struct lfs_mlist * mlist ) [static]

#### 9.4.4.106 static bool lfs_mlist_isopen ( struct lfs_mlist * head, struct lfs_mlist * node ) [static]

#### 9.4.4.107 static void lfs_mlist_remove ( lfs_t * lfs, struct lfs_mlist * mlist ) [static]

#### 9.4.4.108 int lfs_mount ( lfs_t * lfs, const struct lfs_config * cfg )

#### 9.4.4.109 static int lfs_pair_cmp ( const lfs_block_t paira[2], const lfs_block_t pairb[2] ) [inline], [static]

#### 9.4.4.110 static void lfs_pair_fromle32 ( lfs_block_t pair[2] ) [inline], [static]

#### 9.4.4.111 static bool lfs_pair_isnull ( const lfs_block_t pair[2] ) [inline], [static]

#### 9.4.4.112 static void lfs_pair_swap ( lfs_block_t pair[2] ) [inline], [static]

Small type-level utilities ///.

#### 9.4.4.113 static bool lfs_pair_sync ( const lfs_block_t paira[2], const lfs_block_t pairb[2] ) [inline], [static]

#### 9.4.4.114 static void lfs_pair_tole32 ( lfs_block_t pair[2] ) [inline], [static]

#### 9.4.4.115 static int lfs_rawformat ( lfs_t * lfs, const struct lfs_config * cfg ) [static]

#### 9.4.4.116 static lfs_ssize_t lfs_rawgetattr ( lfs_t * lfs, const char * path, uint8_t type, void * buffer, lfs_size_t size ) [static]

#### 9.4.4.117 static int lfs_rawmkdir ( lfs_t * lfs, const char * path ) [static]

Top level directory operations ///.

#### 9.4.4.118 static int lfs_rawmount ( lfs_t * lfs, const struct lfs_config * cfg ) [static]

#### 9.4.4.119 static int lfs_rawremove ( lfs_t * lfs, const char * path ) [static]

#### 9.4.4.120 static int lfs_rawremoveattr ( lfs_t * lfs, const char * path, uint8_t type ) [static]

#### 9.4.4.121 static int lfs_rawrename ( lfs_t * lfs, const char * oldpath, const char * newpath ) [static]

#### 9.4.4.122 static int lfs_rawsetattr ( lfs_t * lfs, const char * path, uint8_t type, const void * buffer, lfs_size_t size ) [static]

#### 9.4.4.123 static int lfs_rawstat ( lfs_t * lfs, const char * path, struct lfs_info * info ) [static]

General fs operations ///.

#### 9.4.4.124 static int lfs_rawunmount ( lfs_t * lfs ) [static]

#### 9.4.4.125 int lfs_remove ( lfs_t * lfs, const char * path )

General operations ///.

#### 9.4.4.126 int lfs_removeattr ( lfs_t * lfs, const char * path, uint8_t type )

#### 9.4.4.127 int lfs_rename ( lfs_t * lfs, const char * oldpath, const char * newpath )

#### 9.4.4.128 int lfs_setattr ( lfs_t * lfs, const char * path, uint8_t type, const void * buffer, lfs_size_t size )

#### 9.4.4.129 int lfs_stat ( lfs_t * lfs, const char * path, struct lfs_info * info )

#### 9.4.4.130 static void lfs_superblock_fromle32 ( lfs_superblock_t * superblock ) [inline], [static]

#### 9.4.4.131 static void lfs_superblock_tole32 ( lfs_superblock_t * superblock ) [inline], [static]

#### 9.4.4.132 static uint8_t lfs_tag_chunk ( lfs_tag_t tag ) [inline], [static]

#### 9.4.4.133 static lfs_size_t lfs_tag_dsize ( lfs_tag_t tag ) [inline], [static]

#### 9.4.4.134 static uint16_t lfs_tag_id ( lfs_tag_t tag ) [inline], [static]

#### 9.4.4.135 static bool lfs_tag_isdelete ( lfs_tag_t tag ) [inline], [static]

#### 9.4.4.136 static bool lfs_tag_isvalid ( lfs_tag_t tag ) [inline], [static]

#### 9.4.4.137 static lfs_size_t lfs_tag_size ( lfs_tag_t tag ) [inline], [static]

#### 9.4.4.138 static int8_t lfs_tag_splice ( lfs_tag_t tag ) [inline], [static]

#### 9.4.4.139 static uint16_t lfs_tag_type1 ( lfs_tag_t tag ) [inline], [static]

#### 9.4.4.140 static uint16_t lfs_tag_type3 ( lfs_tag_t tag ) [inline], [static]

#### 9.4.4.141 int lfs_unmount ( lfs_t * lfs )

## 9.5 littlefs/lfs.h File Reference

```
#include <stdint.h>
#include <stdbool.h>
#include "lfs_util.h"
```

## Data Structures

- struct lfs_config
- struct lfs_info
- struct lfs_attr
- struct lfs_file_config
- struct lfs_cache

  *internal littlefs data structures ///*
- struct lfs_mdir
- struct lfs_dir
- struct lfs_file
- struct lfs_file::lfs_ctz
- struct lfs_superblock
- struct lfs_gstate
- struct lfs
- struct lfs::lfs_mlist
- struct lfs::lfs_free

## Macros

- #define LFS_VERSION 0x00020004

  *Version info ///.*
- #define LFS_VERSION_MAJOR (0xffff & (LFS_VERSION >> 16))
- #define LFS_VERSION_MINOR (0xffff & (LFS_VERSION >> 0))
- #define LFS_DISK_VERSION 0x00020000
- #define LFS_DISK_VERSION_MAJOR (0xffff & (LFS_DISK_VERSION >> 16))
- #define LFS_DISK_VERSION_MINOR (0xffff & (LFS_DISK_VERSION >> 0))
- #define LFS_NAME_MAX 255
- #define LFS_FILE_MAX 2147483647
- #define LFS_ATTR_MAX 1022

## Typedefs

- typedef uint32_t lfs_size_t

  *Definitions ///.*
- typedef uint32_t lfs_off_t
- typedef int32_t lfs_ssize_t
- typedef int32_t lfs_soff_t
- typedef uint32_t lfs_block_t
- typedef struct lfs_cache lfs_cache_t

  *internal littlefs data structures ///*
- typedef struct lfs_mdir lfs_mdir_t
- typedef struct lfs_dir lfs_dir_t
- typedef struct lfs_file lfs_file_t
- typedef struct lfs_superblock lfs_superblock_t
- typedef struct lfs_gstate lfs_gstate_t
- typedef struct lfs lfs_t

### Enumerations

- enum lfs_error {
  LFS_ERR_OK = 0, LFS_ERR_IO = -5, LFS_ERR_CORRUPT = -84, LFS_ERR_NOENT = -2,
  LFS_ERR_EXIST = -17, LFS_ERR_NOTDIR = -20, LFS_ERR_ISDIR = -21, LFS_ERR_NOTEMPTY = -39,
  LFS_ERR_BADF = -9, LFS_ERR_FBIG = -27, LFS_ERR_INVAL = -22, LFS_ERR_NOSPC = -28,
  LFS_ERR_NOMEM = -12, LFS_ERR_NOATTR = -61, LFS_ERR_NAMETOOLONG = -36 }
- enum lfs_type {
  LFS_TYPE_REG = 0x001, LFS_TYPE_DIR = 0x002, LFS_TYPE_SPLICE = 0x400, LFS_TYPE_NAME = 0x000,
  LFS_TYPE_STRUCT = 0x200, LFS_TYPE_USERATTR = 0x300, LFS_TYPE_FROM = 0x100, LFS_TYP↩E_TAIL = 0x600,
  LFS_TYPE_GLOBALS = 0x700, LFS_TYPE_CRC = 0x500, LFS_TYPE_CREATE = 0x401, LFS_TYPE_↩DELETE = 0x4ff,
  LFS_TYPE_SUPERBLOCK = 0x0ff, LFS_TYPE_DIRSTRUCT = 0x200, LFS_TYPE_CTZSTRUCT = 0x202,
  LFS_TYPE_INLINESTRUCT = 0x201,
  LFS_TYPE_SOFTTAIL = 0x600, LFS_TYPE_HARDTAIL = 0x601, LFS_TYPE_MOVESTATE = 0x7ff, LF↩S_FROM_NOOP = 0x000,
  LFS_FROM_MOVE = 0x101, LFS_FROM_USERATTRS = 0x102 }
- enum lfs_open_flags {
  LFS_O_RDONLY = 1, LFS_O_WRONLY = 2, LFS_O_RDWR = 3, LFS_O_CREAT = 0x0100,
  LFS_O_EXCL = 0x0200, LFS_O_TRUNC = 0x0400, LFS_O_APPEND = 0x0800, LFS_F_DIRTY = 0x010000,
  LFS_F_WRITING = 0x020000, LFS_F_READING = 0x040000, LFS_F_ERRED = 0x080000, LFS_F_INLINE = 0x100000 }
- enum lfs_whence_flags { LFS_SEEK_SET = 0, LFS_SEEK_CUR = 1, LFS_SEEK_END = 2 }

### Functions

- int lfs_format (lfs_t ∗lfs, const struct lfs_config ∗config)

  *Filesystem functions ///.*
- int lfs_mount (lfs_t ∗lfs, const struct lfs_config ∗config)
- int lfs_unmount (lfs_t ∗lfs)
- int lfs_remove (lfs_t ∗lfs, const char ∗path)

  *General operations ///.*
- int lfs_rename (lfs_t ∗lfs, const char ∗oldpath, const char ∗newpath)
- int lfs_stat (lfs_t ∗lfs, const char ∗path, struct lfs_info ∗info)
- lfs_ssize_t lfs_getattr (lfs_t ∗lfs, const char ∗path, uint8_t type, void ∗buffer, lfs_size_t size)
- int lfs_setattr (lfs_t ∗lfs, const char ∗path, uint8_t type, const void ∗buffer, lfs_size_t size)
- int lfs_removeattr (lfs_t ∗lfs, const char ∗path, uint8_t type)
- int lfs_file_open (lfs_t ∗lfs, lfs_file_t ∗file, const char ∗path, int flags)

  *File operations ///.*
- int lfs_file_opencfg (lfs_t ∗lfs, lfs_file_t ∗file, const char ∗path, int flags, const struct lfs_file_config ∗config)
- int lfs_file_close (lfs_t ∗lfs, lfs_file_t ∗file)
- int lfs_file_sync (lfs_t ∗lfs, lfs_file_t ∗file)
- lfs_ssize_t lfs_file_read (lfs_t ∗lfs, lfs_file_t ∗file, void ∗buffer, lfs_size_t size)
- lfs_ssize_t lfs_file_write (lfs_t ∗lfs, lfs_file_t ∗file, const void ∗buffer, lfs_size_t size)
- lfs_soff_t lfs_file_seek (lfs_t ∗lfs, lfs_file_t ∗file, lfs_soff_t off, int whence)
- int lfs_file_truncate (lfs_t ∗lfs, lfs_file_t ∗file, lfs_off_t size)
- lfs_soff_t lfs_file_tell (lfs_t ∗lfs, lfs_file_t ∗file)
- int lfs_file_rewind (lfs_t ∗lfs, lfs_file_t ∗file)
- lfs_soff_t lfs_file_size (lfs_t ∗lfs, lfs_file_t ∗file)
- int lfs_mkdir (lfs_t ∗lfs, const char ∗path)

  *Directory operations ///.*

- int lfs_dir_open (lfs_t *lfs, lfs_dir_t *dir, const char *path)
- int lfs_dir_close (lfs_t *lfs, lfs_dir_t *dir)
- int lfs_dir_read (lfs_t *lfs, lfs_dir_t *dir, struct lfs_info *info)
- int lfs_dir_seek (lfs_t *lfs, lfs_dir_t *dir, lfs_off_t off)
- lfs_soff_t lfs_dir_tell (lfs_t *lfs, lfs_dir_t *dir)
- int lfs_dir_rewind (lfs_t *lfs, lfs_dir_t *dir)
- lfs_ssize_t lfs_fs_size (lfs_t *lfs)

    *Filesystem-level filesystem operations.*
- int lfs_fs_traverse (lfs_t *lfs, int(*cb)(void *, lfs_block_t), void *data)

## 9.5.1 Macro Definition Documentation

### 9.5.1.1 #define LFS_ATTR_MAX 1022

### 9.5.1.2 #define LFS_DISK_VERSION 0x00020000

### 9.5.1.3 #define LFS_DISK_VERSION_MAJOR (0xffff & (LFS_DISK_VERSION >> 16))

### 9.5.1.4 #define LFS_DISK_VERSION_MINOR (0xffff & (LFS_DISK_VERSION >> 0))

### 9.5.1.5 #define LFS_FILE_MAX 2147483647

### 9.5.1.6 #define LFS_NAME_MAX 255

### 9.5.1.7 #define LFS_VERSION 0x00020004

Version info ///.

### 9.5.1.8 #define LFS_VERSION_MAJOR (0xffff & (LFS_VERSION >> 16))

### 9.5.1.9 #define LFS_VERSION_MINOR (0xffff & (LFS_VERSION >> 0))

## 9.5.2 Typedef Documentation

### 9.5.2.1 typedef uint32_t lfs_block_t

### 9.5.2.2 typedef struct lfs_cache lfs_cache_t

internal littlefs data structures ///

#### 9.5.2.3 typedef struct lfs_dir lfs_dir_t

#### 9.5.2.4 typedef struct lfs_file lfs_file_t

#### 9.5.2.5 typedef struct lfs_gstate lfs_gstate_t

#### 9.5.2.6 typedef struct lfs_mdir lfs_mdir_t

#### 9.5.2.7 typedef uint32_t lfs_off_t

#### 9.5.2.8 typedef uint32_t lfs_size_t

Definitions ///.

#### 9.5.2.9 typedef int32_t lfs_soff_t

#### 9.5.2.10 typedef int32_t lfs_ssize_t

#### 9.5.2.11 typedef struct lfs_superblock lfs_superblock_t

#### 9.5.2.12 typedef struct lfs lfs_t

### 9.5.3 Enumeration Type Documentation

#### 9.5.3.1 enum lfs_error

**Enumerator**

***LFS_ERR_OK***

***LFS_ERR_IO***

***LFS_ERR_CORRUPT***

***LFS_ERR_NOENT***

***LFS_ERR_EXIST***

***LFS_ERR_NOTDIR***

***LFS_ERR_ISDIR***

***LFS_ERR_NOTEMPTY***

***LFS_ERR_BADF***

***LFS_ERR_FBIG***

***LFS_ERR_INVAL***

***LFS_ERR_NOSPC***

***LFS_ERR_NOMEM***

***LFS_ERR_NOATTR***

***LFS_ERR_NAMETOOLONG***

#### 9.5.3.2 enum lfs_open_flags

**Enumerator**

- ***LFS_O_RDONLY***
- ***LFS_O_WRONLY***
- ***LFS_O_RDWR***
- ***LFS_O_CREAT***
- ***LFS_O_EXCL***
- ***LFS_O_TRUNC***
- ***LFS_O_APPEND***
- ***LFS_F_DIRTY***
- ***LFS_F_WRITING***
- ***LFS_F_READING***
- ***LFS_F_ERRED***
- ***LFS_F_INLINE***

#### 9.5.3.3 enum lfs_type

**Enumerator**

- ***LFS_TYPE_REG***
- ***LFS_TYPE_DIR***
- ***LFS_TYPE_SPLICE***
- ***LFS_TYPE_NAME***
- ***LFS_TYPE_STRUCT***
- ***LFS_TYPE_USERATTR***
- ***LFS_TYPE_FROM***
- ***LFS_TYPE_TAIL***
- ***LFS_TYPE_GLOBALS***
- ***LFS_TYPE_CRC***
- ***LFS_TYPE_CREATE***
- ***LFS_TYPE_DELETE***
- ***LFS_TYPE_SUPERBLOCK***
- ***LFS_TYPE_DIRSTRUCT***
- ***LFS_TYPE_CTZSTRUCT***
- ***LFS_TYPE_INLINESTRUCT***
- ***LFS_TYPE_SOFTTAIL***
- ***LFS_TYPE_HARDTAIL***
- ***LFS_TYPE_MOVESTATE***
- ***LFS_FROM_NOOP***
- ***LFS_FROM_MOVE***
- ***LFS_FROM_USERATTRS***

#### 9.5.3.4 enum lfs_whence_flags

**Enumerator**

***LFS_SEEK_SET***
***LFS_SEEK_CUR***
***LFS_SEEK_END***

### 9.5.4 Function Documentation

#### 9.5.4.1 int lfs_dir_close ( lfs_t * *lfs*, lfs_dir_t * *dir* )

#### 9.5.4.2 int lfs_dir_open ( lfs_t * *lfs*, lfs_dir_t * *dir*, const char * *path* )

#### 9.5.4.3 int lfs_dir_read ( lfs_t * *lfs*, lfs_dir_t * *dir*, struct lfs_info * *info* )

#### 9.5.4.4 int lfs_dir_rewind ( lfs_t * *lfs*, lfs_dir_t * *dir* )

#### 9.5.4.5 int lfs_dir_seek ( lfs_t * *lfs*, lfs_dir_t * *dir*, lfs_off_t *off* )

#### 9.5.4.6 lfs_soff_t lfs_dir_tell ( lfs_t * *lfs*, lfs_dir_t * *dir* )

#### 9.5.4.7 int lfs_file_close ( lfs_t * *lfs*, lfs_file_t * *file* )

#### 9.5.4.8 int lfs_file_open ( lfs_t * *lfs*, lfs_file_t * *file*, const char * *path*, int *flags* )

File operations ///.

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

#### 9.5.4.10 lfs_ssize_t lfs_file_read ( lfs_t * *lfs*, lfs_file_t * *file*, void * *buffer*, lfs_size_t *size* )

#### 9.5.4.11 int lfs_file_rewind ( lfs_t * *lfs*, lfs_file_t * *file* )

#### 9.5.4.12 lfs_soff_t lfs_file_seek ( lfs_t * *lfs*, lfs_file_t * *file*, lfs_soff_t *off*, int *whence* )

#### 9.5.4.13 lfs_soff_t lfs_file_size ( lfs_t * *lfs*, lfs_file_t * *file* )

#### 9.5.4.14 int lfs_file_sync ( lfs_t * *lfs*, lfs_file_t * *file* )

#### 9.5.4.15 lfs_soff_t lfs_file_tell ( lfs_t * *lfs*, lfs_file_t * *file* )

#### 9.5.4.16 int lfs_file_truncate ( lfs_t * *lfs*, lfs_file_t * *file*, lfs_off_t *size* )

#### 9.5.4.17 lfs_ssize_t lfs_file_write ( lfs_t * *lfs*, lfs_file_t * *file*, const void * *buffer*, lfs_size_t *size* )

#### 9.5.4.18 int lfs_format ( lfs_t * *lfs*, const struct lfs_config * *config* )

Filesystem functions ///.

#### 9.5.4.19 lfs_ssize_t lfs_fs_size ( lfs_t * lfs )

Filesystem-level filesystem operations.

#### 9.5.4.20 int lfs_fs_traverse ( lfs_t * lfs, int(*)(void *, lfs_block_t) cb, void * data )

#### 9.5.4.21 lfs_ssize_t lfs_getattr ( lfs_t * lfs, const char * path, uint8_t type, void * buffer, lfs_size_t size )

#### 9.5.4.22 int lfs_mkdir ( lfs_t * lfs, const char * path )

Directory operations ///.

#### 9.5.4.23 int lfs_mount ( lfs_t * lfs, const struct lfs_config * config )

#### 9.5.4.24 int lfs_remove ( lfs_t * lfs, const char * path )

General operations ///.

#### 9.5.4.25 int lfs_removeattr ( lfs_t * lfs, const char * path, uint8_t type )

#### 9.5.4.26 int lfs_rename ( lfs_t * lfs, const char * oldpath, const char * newpath )

#### 9.5.4.27 int lfs_setattr ( lfs_t * lfs, const char * path, uint8_t type, const void * buffer, lfs_size_t size )

#### 9.5.4.28 int lfs_stat ( lfs_t * lfs, const char * path, struct lfs_info * info )

#### 9.5.4.29 int lfs_unmount ( lfs_t * lfs )

## 9.6 littlefs/lfs_util.c File Reference

```
#include "lfs_util.h"
```

### Functions

- uint32_t lfs_crc (uint32_t crc, const void *buffer, size_t size)

### 9.6.1 Function Documentation

#### 9.6.1.1 uint32_t lfs_crc ( uint32_t *crc,* const void ∗ *buffer,* size_t *size* )

## 9.7 littlefs/lfs_util.h File Reference

```
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <inttypes.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
```

### Macros

- #define LFS_TRACE(...)
- #define LFS_DEBUG_(fmt, ...) printf("%s:%d:debug: " fmt "%s\n", __FILE__, __LINE__, __VA_ARGS__)
- #define LFS_DEBUG(...) LFS_DEBUG_(__VA_ARGS__, "")
- #define LFS_WARN_(fmt, ...) printf("%s:%d:warn: " fmt "%s\n", __FILE__, __LINE__, __VA_ARGS__)
- #define LFS_WARN(...) LFS_WARN_(__VA_ARGS__, "")
- #define LFS_ERROR_(fmt, ...) printf("%s:%d:error: " fmt "%s\n", __FILE__, __LINE__, __VA_ARGS__)
- #define LFS_ERROR(...) LFS_ERROR_(__VA_ARGS__, "")
- #define LFS_ASSERT(test) assert(test)

### Functions

- static uint32_t lfs_max (uint32_t a, uint32_t b)
- static uint32_t lfs_min (uint32_t a, uint32_t b)
- static uint32_t lfs_aligndown (uint32_t a, uint32_t alignment)
- static uint32_t lfs_alignup (uint32_t a, uint32_t alignment)
- static uint32_t lfs_npw2 (uint32_t a)
- static uint32_t lfs_ctz (uint32_t a)
- static uint32_t lfs_popc (uint32_t a)
- static int lfs_scmp (uint32_t a, uint32_t b)
- static uint32_t lfs_fromle32 (uint32_t a)
- static uint32_t lfs_tole32 (uint32_t a)
- static uint32_t lfs_frombe32 (uint32_t a)
- static uint32_t lfs_tobe32 (uint32_t a)
- uint32_t lfs_crc (uint32_t crc, const void ∗buffer, size_t size)
- static void ∗ lfs_malloc (size_t size)
- static void lfs_free (void ∗p)

### 9.7.1 Macro Definition Documentation

#### 9.7.1.1 #define LFS_ASSERT( *test* ) assert(test)

#### 9.7.1.2 #define LFS_DEBUG( *...* ) LFS_DEBUG_(__VA_ARGS__, "")

#### 9.7.1.3 #define LFS_DEBUG_( *fmt,* *...* ) printf("%s:%d:debug: " fmt "%s\n", __FILE__, __LINE__, __VA_ARGS__)

#### 9.7.1.4 #define LFS_ERROR( *...* ) LFS_ERROR_(__VA_ARGS__, "")

#### 9.7.1.5 #define LFS_ERROR_( *fmt,* *...* ) printf("%s:%d:error: " fmt "%s\n", __FILE__, __LINE__, __VA_ARGS__)

#### 9.7.1.6 #define LFS_TRACE( *...* )

#### 9.7.1.7 #define LFS_WARN( *...* ) LFS_WARN_(__VA_ARGS__, "")

#### 9.7.1.8 #define LFS_WARN_( *fmt,* *...* ) printf("%s:%d:warn: " fmt "%s\n", __FILE__, __LINE__, __VA_ARGS__)

### 9.7.2 Function Documentation

#### 9.7.2.1 static uint32_t lfs_aligndown ( uint32_t *a,* uint32_t *alignment* ) [inline], [static]

#### 9.7.2.2 static uint32_t lfs_alignup ( uint32_t *a,* uint32_t *alignment* ) [inline], [static]

#### 9.7.2.3 uint32_t lfs_crc ( uint32_t *crc,* const void * *buffer,* size_t *size* )

#### 9.7.2.4 static uint32_t lfs_ctz ( uint32_t *a* ) [inline], [static]

#### 9.7.2.5 static void lfs_free ( void * *p* ) [inline], [static]

#### 9.7.2.6 static uint32_t lfs_frombe32 ( uint32_t *a* ) [inline], [static]

#### 9.7.2.7 static uint32_t lfs_fromle32 ( uint32_t *a* ) [inline], [static]

#### 9.7.2.8 static void* lfs_malloc ( size_t *size* ) [inline], [static]

#### 9.7.2.9 static uint32_t lfs_max ( uint32_t *a,* uint32_t *b* ) [inline], [static]

#### 9.7.2.10 static uint32_t lfs_min ( uint32_t *a,* uint32_t *b* ) [inline], [static]

#### 9.7.2.11 static uint32_t lfs_npw2 ( uint32_t *a* ) [inline], [static]

#### 9.7.2.12 static uint32_t lfs_popc ( uint32_t *a* ) [inline], [static]

#### 9.7.2.13 static int lfs_scmp ( uint32_t *a,* uint32_t *b* ) [inline], [static]

#### 9.7.2.14 static uint32_t lfs_tobe32 ( uint32_t *a* ) [inline], [static]

#### 9.7.2.15 static uint32_t lfs_tole32 ( uint32_t *a* ) [inline], [static]

## 9.8 littlefs/LICENSE.md File Reference

## 9.9 littlefs/SPEC.md File Reference

## 9.10 main.c File Reference

Flash Control Mass Erase & Write 32-bit enabled mode Example.

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include "mxc_assert.h"
#include "mxc_device.h"
#include "flc.h"
#include "flash.h"
#include "littlefs/lfs.h"
```

### Macros

- #define APP_PAGE_CNT 8

  *Flash memory blocks reserved for the app code.*
- #define APP_SIZE (MXC_FLASH_PAGE_SIZE∗APP_PAGE_CNT)

  *The app code flash memory area size.*
- #define TESTSIZE (MXC_FLASH_PAGE_SIZE∗8/4)

  *8 pages of 32 bit samples*
- #define TOTAL_FLASH_PAGES (MXC_FLASH_MEM_SIZE / MXC_FLASH_PAGE_SIZE)

  *Flash memory blocks reserved for internal storage.*
- #define FLASH_STORAGE_START_PAGE 8

  *Internal storage first flash memory block.*
- #define FLASH_STORAGE_PAGE_CNT 8

  *Flash memory blocks reserved for the internal storage.*
- #define FLASH_STORAGE_START_ADDR MXC_FLASH_PAGE_ADDR(FLASH_STORAGE_START_P↩ AGE)

  *Internal storage start address.*
- #define FLASH_STORAGE_SIZE FLASH_STORAGE_PAGE_CNT ∗ MXC_FLASH_PAGE_SIZE

  *Internal storage size.*
- #define FULL_WRITE_TEST 0
- #define FULL_READ_TEST 0

### Functions

- int main (void)

  *Application entry point.*

## Variables

- uint32_t testdata [TESTSIZE]
  *Test data buffer.*
- lfs_t lfs
  *File system instance.*
- uint32_t start_block = FLASH_STORAGE_START_PAGE
  *Internal memory start block to be passed to flash functions by littlefs.*
- const struct lfs_config cfg

### 9.10.1 Detailed Description

Flash Control Mass Erase & Write 32-bit enabled mode Example.

This example shows how to mass erase the flash using the library and also how to Write and Verify 4 Words to the flash.

### 9.10.2 Macro Definition Documentation

#### 9.10.2.1 #define APP_PAGE_CNT 8

Flash memory blocks reserved for the app code.

#### 9.10.2.2 #define APP_SIZE (MXC_FLASH_PAGE_SIZE*APP_PAGE_CNT)

The app code flash memory area size.

#### 9.10.2.3 #define FLASH_STORAGE_PAGE_CNT 8

Flash memory blocks reserved for the internal storage.

#### 9.10.2.4 #define FLASH_STORAGE_SIZE FLASH_STORAGE_PAGE_CNT * MXC_FLASH_PAGE_SIZE

Internal storage size.

#### 9.10.2.5 #define FLASH_STORAGE_START_ADDR MXC_FLASH_PAGE_ADDR(FLASH_STORAGE_START_PAGE)

Internal storage start address.

#### 9.10.2.6 #define FLASH_STORAGE_START_PAGE 8

Internal storage first flash memory block.

#### 9.10.2.7 #define FULL_READ_TEST 0

#### 9.10.2.8 #define FULL_WRITE_TEST 0

#### 9.10.2.9 #define TESTSIZE (MXC_FLASH_PAGE_SIZE*8/4)

8 pages of 32 bit samples

#### 9.10.2.10 #define TOTAL_FLASH_PAGES (MXC_FLASH_MEM_SIZE / MXC_FLASH_PAGE_SIZE)

Flash memory blocks reserved for internal storage.

### 9.10.3 Function Documentation

#### 9.10.3.1 int main ( void )

Application entry point.

**Returns**

Exit code

### 9.10.4 Variable Documentation

#### 9.10.4.1 const struct lfs_config cfg

**Initial value:**

```
= {
        .context = &start_block,

        .read = flash_read,
        .prog = flash_write,
        .erase = flash_erase,
        .sync = flash_sync,


        .read_size = 1,
        .prog_size = 4,
        .block_size = MXC_FLASH_PAGE_SIZE,
        .block_count = FLASH_STORAGE_PAGE_CNT,
        .cache_size = 16,
        .lookahead_size = 16,
        .block_cycles = 500,
}
```

#### 9.10.4.2 lfs_t lfs

File system instance.

#### 9.10.4.3 uint32_t start_block = FLASH_STORAGE_START_PAGE

Internal memory start block to be passed to flash functions by littlefs.

#### 9.10.4.4 uint32_t testdata[TESTSIZE]

Test data buffer.

## 9.11 README.md File Reference

## 9.12 littlefs/README.md File Reference

# Index