JBLopen

Embedded Software Insight

TREEspan File System User Manual

UM0002

February 12, 2020

Contents

Chapter

1

Introduction

1.1 About the User Manual

Welcome to the TREEspanTM File System (TSFS) User Manual. This user manual covers every aspects of TSFS, including theoretical background, usage examples, performance analysis and API descriptions. A separate API Reference Manual is also available for more convenience.

1.1.1 Audience

This user manual has been written with application designers in mind. Some background in embedded programming is assumed but experience with file systems and storage technologies is not required. Much effort as been spent in making this user manual both complete and easy to read, providing the reader with background information where needed, while limiting the scope and the level of details to include only those aspects relevant to an application designer.

1.1.2 Roadmap

To facilitate the navigation of this user manual, a short description of each chapter follows.

Chapter 2 introduces basic notions regarding file systems and storage technologies, with the emphasis on flash and flash-based devices. It does so in a broad and generic language, independent of TSFS peculiarities and terminology. Readers with an extensive knowledge of file systems and storage technologies may only want to skim through this chapter.

Chapter 3 summarizes the most important characteristics and features of TSFS. It is meant as a quick reference to evaluate and compare TSFS with other available solutions.

Chapter 4 is a comprehensive presentation of TSFS performances, including theoretical concepts and in-depth analysis of key performance metrics. Above all, this chapter provides application designers with rational performance expectations for various storage devices and configurations.

Chapter 5 introduces the TSFS API through a series of short tutorials. Each tutorial covers a group of logically-related interfaces through a hands-on approach, giving complete examples and descriptions of function parameters, returned values and side-effects.

Chapter 6 describes optional TSFS configurations that can be used to tailor overall performances according to specific application requirements. These configurations should typically be left untouched — at least in the early stages of the application design — as the default values are suitable for a wide range of applications.

Finally, Chapter 7 contains complete descriptions of all TSFS public functions, macros and data types, including function parameters, returned values, attributes and side-effects.

1.2 Notation and Conventions

1.2.1 Storage Devices and Media

A storage device and a storage media are essentially the same thing: some kind of non-volative memory where data can be stored. However, both expressions are used in a subtly different ways throughout this manual. *Device* is used whenever the emphasis is put on physical aspects like the I/O interface, the dimensions or the memory technology involved. *Media*, on the other hand, is used when more abstract or higher-level characteristics are referred to, like the write throughput or the read granularity.

1.2.2 Size and Speed Units

Sizes and speeds are given using either binary or decimal units, whichever is best to describe the situation at hand. The most common size units, along with their corresponding values, are given in Table 1.

Bit unit symbol	Value in bits	Byte unit symbol	Value in bytes
kbit	1000 bits	kB	1000 bytes
Kibit	1024 bits	KiB	1024 bytes
Mbit	1000 kbit	MB	1000 kB
Mibit	1024 Kibit	MiB	1024 KiB
Gbit	1000 Mbit	GB	1000 MB
Gibit	1024 Mibit	GiB	1024 MiB
Tbit	1000 Gbit	TB	1000 GB
Tibit	1024 Gibit	TiB	1024 GiB

Table 1 – Most common size units and their corresponding values.

1.2.3 Text Formatting

`Monospace` is used throughout this user manual to indicate an element of code like a function name or a data type (e.g. `tsfs_file_write()`, `RTNC_SUCCESS`, `tsfs_file_hndl_t`).

Italic is used to put the emphasis on a word or expression, most notably when it appears for the first time, usually along with a definition.

1.2.4 Abbreviations and Acronyms

Abbreviations and acronyms are used throughout this manual for the sake of conciseness. When a word or expression appears for the first time, it is written in full, along with the abbreviated form between parentheses. Subsequent occurrences of the same word or expression can then appear solely in the abbreviated form. A complete list of abbreviations and acronyms is given in Table 2.

Abbreviation	Meaning
API	Application Programming Interface
COW	Copy-On-Write
CPU	Central Processing Unit
ECC	Error Correction Code
eMMC	Embedded Multi Media Card
FTL	Flash Translation Layer
HDD	Hard Disk Drive
I/O	Input/Output
KOPS	Kilo Operations Per Second
MLC	Multi-Level Cell
PDF	Probability Density Function
RAM	Random Access Memory
RTOS	Real-Time Operating System
SD Card	Secure Digital Card
SDD	Solid-State Drive
SLC	Single-Level Cell
TSFS	TreeSpan File System
UFS	Universal Flash Storage

Table 2 – Abbreviations and Acronyms used in this manual.

Chapter

2 Context and Concepts

This chapter introduces basic notions regarding file systems and storage technologies, with the emphasis on flash and flash-based devices. It does so in a broad and generic language, independent of TSFS peculiarities and terminology. Readers with an extensive knowledge of file systems and storage technologies may only want to skim through this chapter.

2.1 File System Basics

A file system provides a uniform data storage abstraction made of separate logical data containers called *files*. Each file is identified by a name (or sometimes a numeric ID). The content of a file is exposed as a contiguous array of bytes, where each byte can be fetched or updated independently. A file can shrink or grow dynamically.

The file system is responsible for dynamically allocating the needed disk space as files are updated. It must also keep track of the allocated portions of disk space for each file, along with their respective logical positions inside the file. All this bookkeeping is usually supported by various on-disk data structures, often referred to as *metadata*.

2.1.1 The File Abstraction

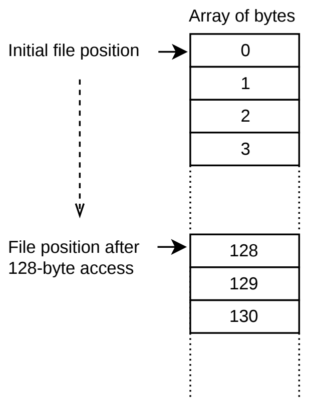
Through the file system interface, the application can:

1. create a new file;
2. delete an existing file;
3. read bytes from a file;
4. write bytes to a file.

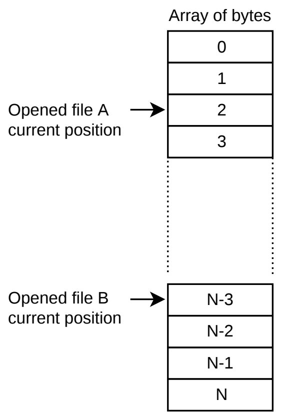
Before a file can be accessed (read from or written to), it must first be opened. When a file is opened, a corresponding in-RAM structure is created whose primary function is to keep track of the current file position. We refer here to this structure as the *file descriptor*, but the exact naming varies across file systems.

When a file is opened, the current position is 0. Read or write accesses are performed at the current file position. After each access, the current position is incremented by the size of the access. This is shown in Figure 1(a). The position can also be modified through a dedicated file system interface.

A file can usually be opened multiple times, resulting in as many file descriptors and access positions. This is illustrated in Figure 1(b).



(a) The position is automatically incremented by the size of the access. In this case from position 0 to position 128.


Array of bytes
0
1
Opened file A current position
2
3
Opened file B current position
N-3
N-2
N-1
N

(b) The same file opened twice. Two independent in-RAM file descriptors, each with its own current position.

Figure 1 – The file abstraction and the relation between in-RAM file descriptors and on-disk byte arrays.

2.1.2 The Directory Abstraction

Most file systems — including TSFS — have their files organized into a tree hierarchy supported by some kind of directory abstraction. A directory is a file container that can be used for grouping related files.

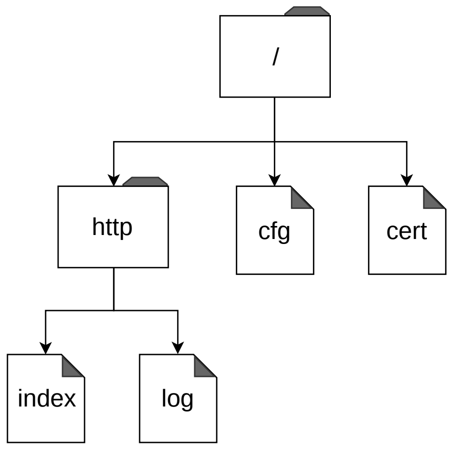
Like files, directories have names. A directory can contain multiple files. It can also contain other directories. However, files and directories have a single parent directory, hence the tree structure. A simple example of a directory tree is given in Figure 2.

The location of a file or directory within the tree hierarchy is specified using a path. The path is formed by concatenating the names of the nodes on the path between the root directory (/) up to the file or directory whose path is being formed. Directory names are separated using a dedicated character, usually '/'. Figure 2 shows the path for each file or directory and its parent directory.

On systems equipped with a human interface, directories are used to logically organize files and facilitate the navigation. This logical grouping can also lead to improved performances on some systems, assuming that files contained in the same directory tend to be accessed together. This is not the case for TSFS though, since file indexing structures are separate from the directory tree structure.

2.1.3 File System Integration

The file system does not usually interacts directly with the underlying physical storage media. Instead, the file system interacts with an intermediate media driver as depicted in Figure 3. All media drivers



File or Directory	Path	Parent Path
/	/	
cfg	/cfg	/
cert	/cert	/
http	/http	/
index	/http/index	/http
log	/http/log	/http

Figure 2 – A simple example of directory hierarchy

present a common media driver interface to the overlying file system, effectively hiding media specificities.

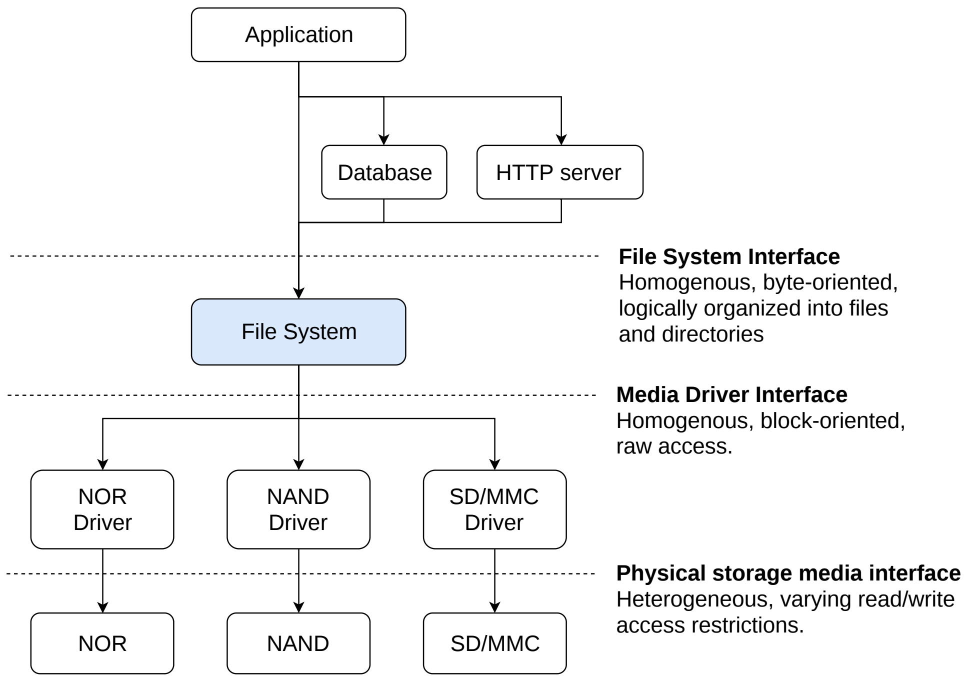

Application
Database
HTTP server
File System Interface
Homogenous, byte-oriented, logically organized into files and directories
File System
Media Driver Interface
Homogenous, block-oriented, raw access.
NOR Driver
NAND Driver
SD/MMC Driver
Physical storage media interface
Heterogeneous, varying read/write access restrictions.
NOR
NAND
SD/MMC

Figure 3 – Storage Layers

In many common scenarios, the application interacts directly with the file system. This is not, however, always the case. Databases and HTTP servers for instance, need a file system for internal storage purposes. These software components generally include some kind of connection layer for the file system. Once the integration with the file system is done, file operations are mostly driven by the overlying modules, keeping only minimal setup operations (e.g. format, mount, etc.) under the application's responsibility.

2.2 Flash Memory

Flash memories are solid-state storage devices. Unlike electromechanical hard drives, flash memories do not have moving parts. They can thus withstand a fair amount of shocks and vibrations, which is a great asset for many embedded applications. Other qualities of flash memories include their low cost, high density (more about that later on) and efficient random accesses.

2.2.1 Flash vs Block Devices

Most file systems are designed to run on top of block devices — although this is not the case for TSFS, as we will see later on. A *block device* is a storage device made of blocks of data that can be individually fetched or updated. Accesses smaller than a block are not allowed. Partial block updates can however be achieved through a read-modify-write procedure, such as depicted in Figure 4.

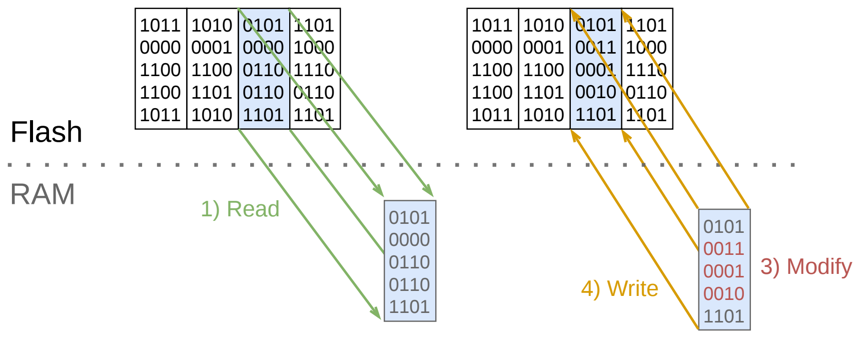


Figure 4 – A block is partially updated through a read-modify-write procedure.

Although flash memories are also made of blocks, they do not behave like block devices. Flash memories are made of two kinds of blocks: *pages* and *erase blocks*. The page is the elementary write unit (much like blocks for block devices). The erase block is the elementary erase unit. An erase block contains several pages (64 pages per block is not unusual).

A page program operation can flip bits from 1 to 0, but not the other way around. The only way to transition from 0 back to 1, is to erase the containing block, effectively setting all the bits to 1. Page program operations can then be used to bring selected bits back to 0.

The update procedure described in Figure 4 does not work on flash devices due to the lack of support for in-place updates. A slightly different read-write-modify procedure, compatible with flash access restrictions, is depicted in Figure 5. It goes as follows:

1. load the block containing the page (or portion of a page) to be modified in RAM;
2. erase the block (setting all bits to 1);
3. modify the loaded block in RAM;
4. write the whole modified block back to its original location.

Although this update procedure superficially works, it suffers a number of problems.

First, there is a performance issue. In the suggested update scheme, every single update requires a full block to be read, erased and written. This is costly because erase blocks are usually quite larger than pages.

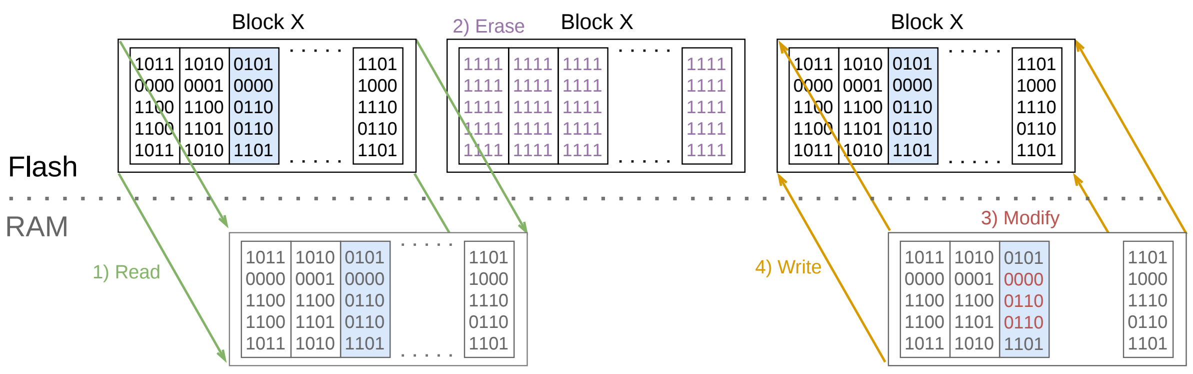


Figure 5 – The third page of block X is updated in-place using a straightforward read-erase-modify-write approach.

Second, there is a wear issue. Indeed, flash blocks can only be erased a limited number of times before wearing out (wear and wear mitigation techniques are covered in Section 2.2.3). Some flash technologies have longer life cycles than others, but erasing a whole block for each update is generally not acceptable.

Finally, and perhaps most importantly, there is a fail-safety issue. More precisely, consider what could happen if the procedure was interrupted (say because of a power outage) between the erase and write steps. In such a scenario, the content of the whole block would be lost, which is obviously unacceptable.

A better way of dealing with flash access restrictions is through the use of copy-on-write (COW), as discussed in Section 2.2.2.

2.2.2 Copy-on-Write and Flash Translation Layers

As previously discussed in Section 2.2.1, flash memories do not allow in-place updates. Because of that limitation, they cannot natively provide the block device abstraction needed by most file systems. An additional software adapter layer is therefore needed.

This software adapter is often referred to as a Flash Translation Layer (FTL). Its main purpose is to emulate a block device for the benefits of the overlying file system. This is depicted in Figure 6.

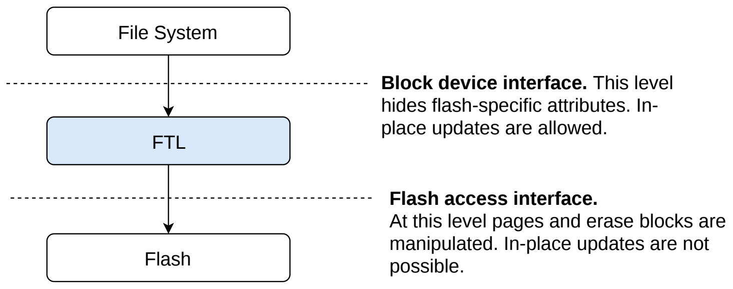

File System
FTL
Flash
Block device interface. This level hides flash-specific attributes. In-place updates are allowed.
Flash access interface. At this level pages and erase blocks are manipulated. In-place updates are not possible.

Figure 6 – A flash translation layer (FTL) emulates a block device on top of a flash memory.

FTLs generally use out-of-place update strategies, often referred to as copy-on-write (COW). In a COW update scheme, modified pages are not written back to their original location, but rather copied to other (pre-erased) pages. This approach, shown in Figure 7, has two important benefits:

1. the original page remains intact so it can be recovered if the update fails midway through;
2. a single page update requires no more than a single page write, as opposed to a whole block erase/write in the case of the in-place update strategy discussed in Section 2.2.1.

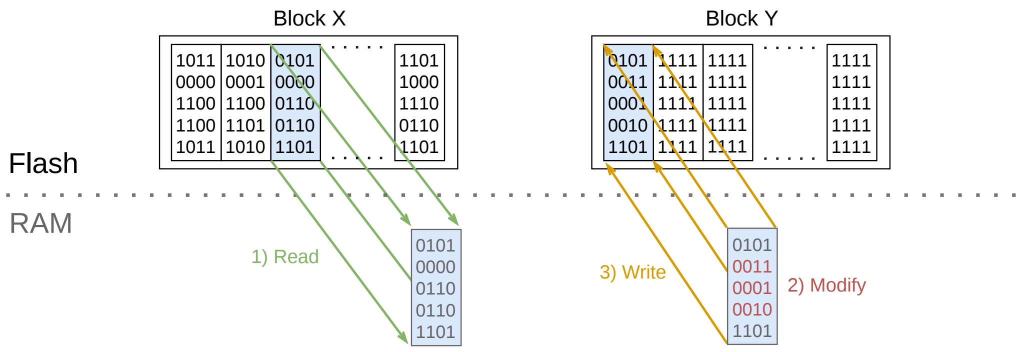


Figure 7 – A copy-on-write (COW) update as performed by an FTL. In this case, the 3^{rd} page of block X is updated into the 1^{st} page of block Y.

Out-of-place updates play nice on top of flash devices, but they do not provide the expected block device abstraction. To reconcile the COW approach with the required block device behavior, FTLs introduce an additional level of indirection through the concept of *logical blocks*. Logical blocks have fixed *logical addresses*, which are exposed to and manipulated by the file system. However, logical blocks can be moved around, stored in different physical pages and thus, at different *physical addresses*.

The logical-to-physical address mapping is stored on the flash device along with the file system. When the overlying file system loads a logical block, the FTL performs the logical-to-physical address translation and loads the content of the corresponding flash page. When a logical block is updated, the FTL moves the updated content to a new page and adjust the logical-to-physical mapping accordingly.

As a flash file system (more on this in Section 2.2.7), TSFS uses COW update strategies, but these are built into the file system core, such that no extra translation layer (FTL) is required.

2.2.3 Wear-leveling

Flash blocks can only be erased a limited amount of times before they become unreliable. Some flash devices can stand up to 100k erase cycles. Others, only 1000. Given this limited life span, erase operations must be distributed across the available blocks as evenly as possible. Failing that, some blocks can become completely unusable while others remain almost unaltered, effectively shortening the entire flash lifetime.

Enforcing uniform block wear is often handled at the FTL level, and is commonly referred to as *wear-leveling*. We distinguish between two types of wear-leveling: *static* and *dynamic*. The difference between static and dynamic wear leveling resides in the block selection algorithm responsible for picking the next block to be freed, erased and written to.

Dynamic wear-leveling operates in an opportunistic way, taking advantage of data being moved around as part of COW updates (hence dynamic). Dynamic leveling never forces valid data out of a long-standing block. Rather, the next destination block is selected among those already made available by previous COW updates. Dynamic leveling performances thus heavily depend on the access patterns of the overlying application.

Static wear-leveling is different in that it proactively moves stale data. Leveling is thus independent of the application's access patterns and guaranteed across the whole flash. Although static leveling may come at the expense of slightly degraded write performances, it is usually required in order to achieve a high level of reliability.

TSFS supports both static and dynamic wear-leveling. Being log-structured (more on this in Section 2.2.7), TSFS architecture naturally lends itself to wear-leveling, such that it comes at a very low cost in terms of performance and code complexity.

2.2.4 Bad block management

Despite efficient wear-leveling, flash memories eventually develop unreliable blocks over their estimated lifetime. In fact, unreliable blocks may already exist as flash devices leave the factory. Unreliable blocks are most often referred to as *bad blocks*.

The initial bad block identification method may vary from one flash manufacturer to another. Usually, some kind of special bad block mark is written in a specific location within the block before the flash comes out of the factory. The flash management software is then responsible for locating these marks.

Sometimes, a block erase or page write operation returns an error. When such an error occurs, valid data should be moved out of the defective block and copied to a healthy block. The bogus block should then be considered bad and should not be further accessed.

The details of bad block management algorithms go beyond the scope of this introductory text. The one thing to remember, though, is that the amount of available blocks may decrease as the flash memory approaches its end-of-life. Flash management software is therefore responsible for keeping spare blocks, enough to compensate for expected bad block occurrences.

2.2.5 Bit Errors and Error Correction

Voltage thresholds within flash cells are purposely modified as part of normal flash operation. But they can also be subject to undesirable side effects. While these side effects can be mitigated by proper flash management, they can sometimes be significant enough to cause bit errors.

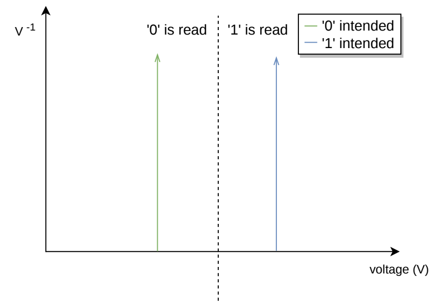
The unwanted variations in voltage thresholds are often modeled as Gaussian noise, where each logical symbol — '0' and '1' for SLC — is associated with a normal probability density function (PDF) representing the corresponding voltage threshold fluctuations. Various scenarios are illustrated in Figure 8. These graphical representations are not meant to be realistic — let alone accurate. They are solely intended to support the reader's intuition.

Figure 8(a) shows the ideal — but impossible — case where nominal voltage thresholds occur with a probability of 1. In this case, the probability of a bit error is null (i.e. the probability that '0' is read while '1' is intended, or reciprocally).

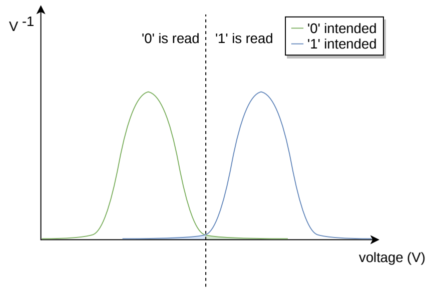
Figure 8(b) and Figure 8(c) illustrate the impact of wear and aging on bit errors. The bit error probability is graphically represented by the colored surfaces. The greater the surface, the higher the probability of a bit error. As wear progresses, noise becomes more pronounced which translates into a higher variance, more overlap between PDFs and, therefore, a higher bit error probability.

Figure 8(d) shows the PDFs for an unused MLC flash. Comparing with the SLC case (see Figure 8(b)), voltage levels are closer from each other, PDFs overlap more and, therefore, the bit error probability is higher.

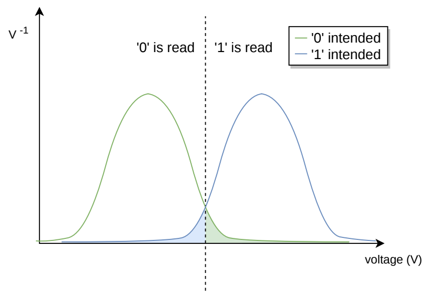
To palliate these bit errors, *error correction codes* (ECC) must be used. It is crucial that the ECC be selected based on the situation at hand. From the previous discussion, it is clear that the selected algorithm must be strong enough to sustain the degradation of reliability resulting from the flash aging process. It must also be adapted to the underlying flash technology, MLC requiring much stronger ECC than SLC.



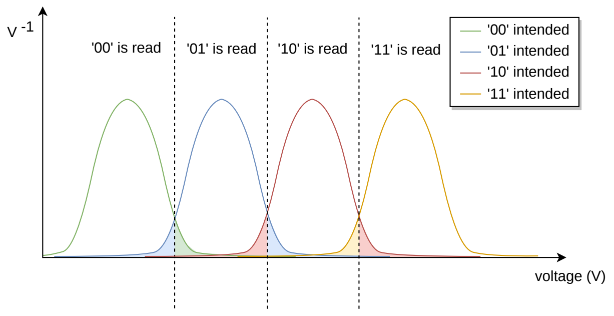
(a) Ideal SLC flash. Nominal voltage levels occur with a probability of 1. Other levels never occur. The probability of a bit error is null.



(b) Unused SLC flash. The variance is low, density functions overlap slightly. The probability of a bit error is small.



(c) Used SLC flash. The variance is high, density functions overlap considerably. The probability of a bit error is significant.



(d) MLC flash. Voltage levels are closer than SLC levels, density functions overlap considerably. The probability of a bit error is significant.

Figure 8 – Voltage probability density functions for various scenario.

2.2.6 Managed and Unmanaged flash

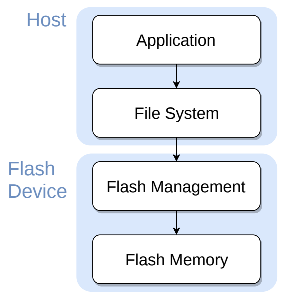
To summarize what has been previously discussed, flash management includes the following tasks:

- logical to physical address translation;
- wear-leveling;
- bad-block management;
- error correction (ECC).

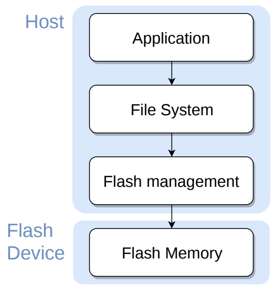
Although these tasks must invariably be performed to achieve the expected level of performance and reliability, the total workload can be distributed between the host (e.g. the microcontroller unit) and the storage device in various ways. This flexibility roughly translates into three categories of flash devices (illustrated in Figure 9):

- *managed*, where the flash management is entirely performed inside the storage device;

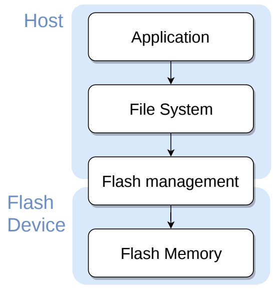
- *unmanged*, where the flash management is entirely performed inside the host;
- *semi-managed*, where the flash management is shared between the host and the storage device.



(a) Managed flash. Flash management is handled by the flash device itself.



(b) Unmanaged flash. Flash management is handled by the host.



(c) Semi-managed flash. Flash management is shared between the host and the flash device.

Figure 9 – Various distributions of the flash management workload between the host and the flash device.

A managed flash device essentially emulates a block device. It can be used as a drop-in replacement for other storage technologies exposing a block device interface. Perhaps the most telling example of such interchangeability is the solid state drive (SSD), which has made its way as the predominant non-volatile PC storage technology, in lieu of hard disk drives (HDD). On the embedded side, SD cards, eMMC and UFS modules are popular alternatives.

Given a block device interface, file systems can be deployed without the need for an extra adaptation layer (FTL). This is assuredly one of the most important benefit of managed flash. However, managed flash devices are more expensive, take up more physical space and consume more energy than equal-sized bare flash counterparts.

The unmanaged flash category includes all bare flash chips. In this case, the host is responsible for performing all the required flash management tasks. Bare flash chips are popular in embedded design, where cost, size and energy consumption must be kept as low as possible.

The semi-managed flash category includes all sorts of hybrid solutions. Perhaps the most notorious members of this category are flash devices with built-in ECC. In this case, the flash device returns a read status reporting whether bit errors have occurred, the number of erroneous bits, and whether the errors have been corrected. The ECC calculations are taken care of by the flash device but the host must still respond with an appropriate action when errors occur. That includes refreshing (moving to another block) data when the number of bit errors is close to the ECC limits.

TSFS is a log-structured flash file system. As such, it supports unmanaged flash devices without the need for an extra adaptation layer (FTL). Unlike other flash file systems, TSFS also supports managed flash. On managed flash devices, TSFS log structure allows for a high-speed random write accesses. This is discussed at length in Section 4.2.

2.2.7 Flash File Systems

As discussed in Section 2.2.1, file systems usually rely on the presence of a block device abstraction. On flash devices, this abstraction is provided by an extra adaptation layer, called the flash translation layer (see Section 2.2.2).

Flash file systems are different. Rather than deferring flash management to a dedicated translation layer, they directly deal with flash-specific access restrictions and other flash peculiarities. The end result is typically more efficient than the combination of a separate file system and FTL : smaller RAM usage, lower write amplification, simpler integration and configuration.

A special but important category of flash file systems — which TSFS belongs to — are *log-structured file systems*. In a log-structured file system, all updates are performed in a circular stream of contiguous write operations. This approach has long been used to work around poor random write performances of electromechanical hard drives, but it makes even more sense on flash-based storage technologies, especially raw flash memories where in-place updates are simply not allowed.

2.2.8 NOR and NAND flash

All flash memories share some basic qualities — inexpensive, robust, physically small. But all flash memories are not equal and choosing the right technology for the job at hand can be a daunting task. Without providing all the answers, we give here some basic guidelines.

Flash memories can be categorized by transistor configuration. There are two widely used configurations: NOR and NAND. NAND flash devices are further categorized based on the number of bits stored per physical cell. Table 3 lists currently available NAND technologies along with their corresponding number of bits per cell and typical endurance specifications.

	SLC	MLC	TLC	QLC
#bits per cell	1	2	3	4
Endurance	100000	10000	3000	1000

Table 3 – Available NAND flash technologies along with typical endurance specifications.

SLC and MLC technologies are widely used in embedded devices. On the other hand, TLC and QLC are more rarely seen. While higher densities are attractive, squeezing more bits into a single physical cell comes at the expense of an increased bit error rate and decreased endurance. This might be good enough for some consumer applications, but not for most industrial, military or medical applications.

Table 4 is a side-by-side comparison of the NOR, NAND SLC and NAND MLC technologies based on various performance metrics. The given values are provided as guidance and may significantly vary across actual devices. Figure 10 roughly depicts the available densities and costs.

Looking at Table 4 and Figure 10, important observations can be made:

- NOR has the highest cost per byte but the lowest absolute cost. In other words, for applications requiring little data space, NOR may be the cheapest solution. However, price goes up rapidly as size increases.
- NOR write energy consumption can be as much as 100 times greater than that of NAND flash. Battery-powered devices are probably better served by NAND flash, unless very few writes are performed.
- The write throughput is higher on NAND, but NOR exhibits finer write access granularity. For workloads dominated by small write accesses, the gap between NOR and NAND net write throughputs can close rapidly.
- The read granularity is much finer on NOR. Given its high read throughput, NOR is usually preferred over NAND for workloads dominated by small random read accesses.

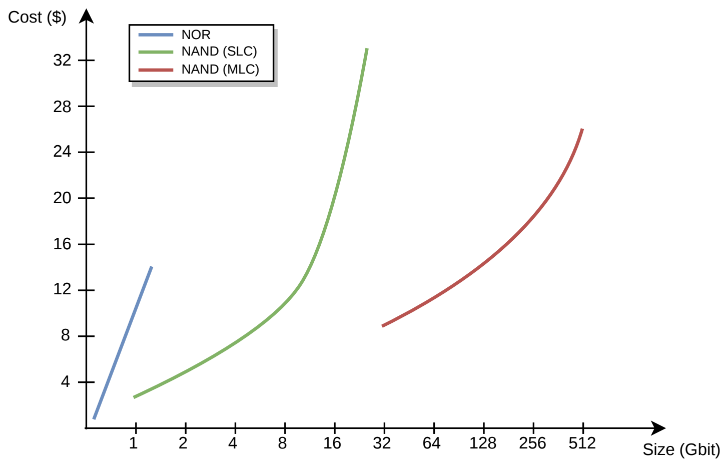


Figure 10 – Flash memories cost versus size

	NOR	NAND (SLC)	NAND (MLC)
Write Throughput	1MB/s	10MB/s	5MB/s
Read Throughput	40MB/s	30MB/s	20MB/s
Erase Throughput	0.1MB/s	300MB/s	600MB/s
Erase-Write Throughput	100kB/s	10MB/s	10MB/s
Read Granularity	Byte	Page (4KiB typical)	Page (8KiB typical)
Write Granularity	Byte (typical)	Page (4KiB typical)	Page (8KiB typical)
Erase Granularity	Block (64KiB typical)	Block (256KiB typical)	Block (2MiB typical)
Write Energy Consumption	1uJ/byte	10nJ/byte	20nJ/byte
Read Energy Consumption	1nJ/byte	1nJ/byte	2nJ/byte
Erase Energy Consumption	600nJ/byte	0.2nJ/byte	0.1nJ/byte

Table 4 – Typical NOR and NAND flash characteristics

Beyond these general observations, application designers should evaluate combined hardware/software performances along with application requirements in the early stages of the design. Chapter 4 offers a comprehensive analysis of TSFS performances for various storage devices and configurations.

2.3 Fail-Safety

2.3.1 Failures and Corruption

Unexpected failures can lead to both data and metadata corruption. When data corruption occurs, the file system can still operate normally. Only those modules whose files have been corrupted are jeopardized. The rest of the application can still run normally. This is unlike metadata corruption which can trigger unrecoverable errors with disastrous consequences for the entire application.

Whether metadata or data corruption occurs depends (among other things) on the timing. This is illustrated in Figure 11, where some file is updated through a succession of `file_write()` calls. This situation is typical of large updates where the data to be written does not fit into a single application buffer.

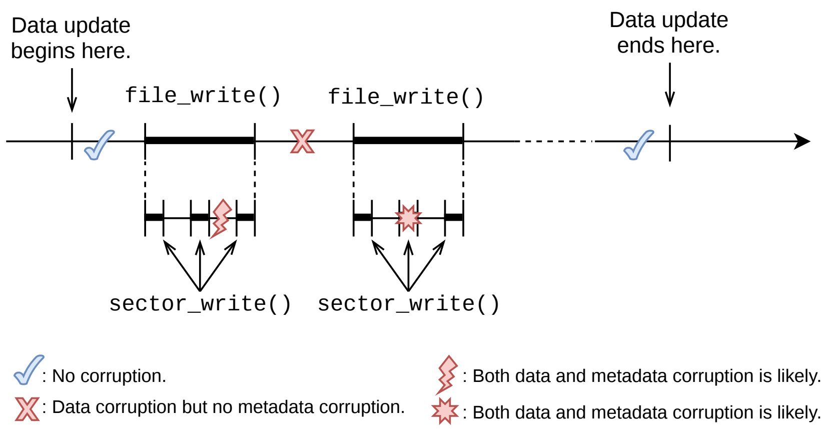

Data update begins here.
Data update ends here.
file_write()
file_write()
sector_write()
sector_write()
: No corruption.
: Data corruption but no metadata corruption.
: Both data and metadata corruption is likely.
: Both data and metadata corruption is likely.

Figure 11 – Timing determines whether corruption occurs or not.

First, consider the two failures represented by blue check marks on the timeline of Figure 11. As both failures happen respectively before and after the update procedure, both the data and metadata are either untouched or fully updated. Therefore, nothing is corrupted.

Next, consider the failure represented by a red 'X'. Since the failure happens midway through the update procedure, the file is only partially updated. As for the metadata, when the first `file_write()` completes, the file system is returned to a consistent state. Therefore, the metadata is safe. The overall outcome is nonetheless disastrous for the application as the updated file is incomplete and the old version is lost.

Now, consider the failure represented by a red lightning bolt. In this case, data corruption is still probable. Furthermore, since the failure happens during a call to `file_write()`, metadata updates may be incomplete. In this case, the file system could be irremediably corrupted, leading to the loss of the entire content.

Finally, consider the failure represented by a red star. This scenario is similar to the previous one in that both data corruption and metadata corruption are possible. But in this case, since the physical sector update itself is interrupted, the actual outcome depends on the underlying media. Some media feature atomic updates, in which case a sector is guaranteed to be either untouched or fully updated (but never in-between). Others, guarantee in-order updates, where a contiguous portion of the sector is updated, the rest being left untouched. Still others, offer no guarantee at all, and the updated sector can end up in pretty much any random state.

2.3.2 Journaled File Systems

Several file systems use a mechanism known as *journaling* to better cope with unexpected failures. In the embedded world, a popular solution is the FAT file system with an added journal. But regardless of the implementation, the purpose of journaling remains the same: protecting against metadata corruption by either rolling back or completing partial updates upon recovery.

With a few (performance-hogging) exceptions, journaled file systems offer no additional protection for application data. Besides, most journal implementations rely on atomic sector updates, which, as we have seen, is not guaranteed for all media.

Finally, journaling is generally not compatible with write-back caching or, at the very least, hampers write-caching performances as it requires strict sector write ordering.

2.3.3 Transactional File Systems

Transactional file systems are different than journaled file systems in that they protect both data and metadata. Data protection is achieved through write transactions.

A *write transaction* is a sequence of write operations that either succeeds or fails as a whole. Partial operations (or sequences of operations) can never happen, and thus, only one of two possible outcomes can be observed:

1. the transaction completes without interruption and the file system state reflects the result of the entire sequence of write operations;
2. a failure occurs during the transaction and the file system is returned to its initial state (i.e. the one it was in when the transaction started).

This all-or-nothing behavior is illustrated in Figure 12.

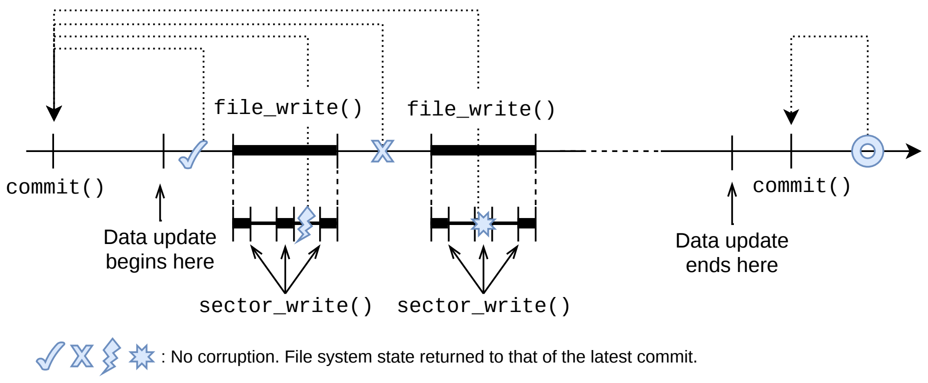


Figure 12 – In a transactional file system, data and metadata are protected against corruption, regardless of the failure timing.

A transaction is ended by the application using a dedicated `commit()` interface. Depending on the actual implementation, a new transaction is either explicitly started using a dedicated interface or automatically started each time a transaction is ended. For the purposes of the following discussion, we assume the latter.

Now, consider the various failure scenarios shown in Figure 12 — respectively represented by a check mark, a lightning bolt, an 'X', a star and a circle. In Section 2.3.1, we have seen how these failures could lead to data and/or metadata corruption. Using transactions, this is no longer the case.

Since all the file write operations are now part of a single transaction, they either succeed or fail as a whole, leaving no room for half-complete updates. No matter how and when it fails, the file system is always returned to the state it was in at the time of the latest commit. This is represented by the dotted arrows in Figure 12.

TSFS is a transactional file system, which means that it protects both data and metadata against abrupt failures.

Chapter

3 TSFS Key Features

3.1 Overview

The TREEspan™ File System (TSFS) is an embedded transactional file system, supporting a wide range of storage technologies, including native flash support with both dynamic and static wear-levelling. Through its support for snapshots and write transactions, TSFS provides the application with flexible, robust and fail-safe data storage. Being RTOS and platform agnostic, with a minimum RAM requirement of less than 4KiB, TSFS can be deployed on almost any platform.

3.2 Log Structure

TSFS is a log-structured file system, meaning that all updates are performed out-of-place, in a circular stream of contiguous write operations. The log structure naturally lends itself to write transactions and snapshots, which come at no additional cost in terms of performance.

3.3 Built-in Flash Support

TSFS is a flash file system (see Section 2.2.7). Its integrated design improves overall performances by factoring various file management aspects into flash management algorithms — and reciprocally. Where, for instance, two separate data structures are typically used to support garbage collection and data block indexing (the former by the FTL, the latter by the file system), TSFS uses only one unified structure. This means less RAM usage, less pressure on the read cache and less write amplification.

TSFS flash support goes down to specific characteristics of each supported flash technology. On NOR flash, for instance, TSFS leverages the fast and granular read operations to minimize the write amplification and avoid costly write and erase operations as much as possible. In contrast, NAND flash erase and write operations are much faster, while read operations are slower. Also, read/write granularity is coarser (i.e. page-oriented). In this case, TSFS metadata is updated such as to minimize random read operations at the expense of a slightly higher write amplification.

3.4 Fail-Safety and Write Transactions

TSFS is designed to handle untimely interruptions in a centralized and strictly defined manner, effectively protecting the application against data corruption and taking the burden of handling such failures off the application designer.

Upon recovering from an unforeseen interruption, TSFS always returns to the latest consistent state, as defined by the application through the dedicated write transaction interface. The application can also choose to drop uncommitted modifications, which is useful for handling non fatal errors such as network errors. In this case, instead of trying to resume the update procedure, partial updates can simply be canceled and started over.

3.5 Snapshot Support

Snapshot support is arguably one of the most unique TSFS feature. Snapshots provide a space-efficient way of saving an entire file system state that can be later accessed or reverted to. Frozen versions of files and directories, isolated from potential concurrent updates, can thus be read from, protecting the application against data corruption induced by race conditions.

3.6 RAM Usage and Scaling

One of the defining characteristics of embedded systems — as opposed to general purpose computers — is the small amount of available resources, most notably the small amount of available RAM.

Unlike other log-structured file systems, TSFS has a very low minimum RAM requirement (under 4KiB). Minimizing RAM usage, while maintaining the best possible performances, is achieved through a combination of flexible read caching and strategic on-disk layout. Incidentally, this approach makes for very low mount times, as very few data structures must be loaded up front.

Even with the most frugal setup, the on-disk tree-based indexing allows for impressive performances, while more capable platforms can reach the highest possible performances by increasing the amount of memory dedicated to the built-in read cache. In both cases, the RAM usage is independent of the size of the storage media and the number/size of stored files.

3.7 High-speed Random Write Accesses

TSFS log-structured approach is most natural on raw flash memories, where in-place updates are simply not allowed (see Section 2.2.1). Still, managed flash devices also benefit from the log structure because sequential write speed is typically much higher than random write speed on these devices. The overall random write performance gain, in this case, can be as much as one order of magnitude. TSFS performances are separately discussed in Chapter 4.

Chapter

4

Performances and Space Management

File system performance is an immensely complex topic. To be fair and accurate, a performance assessment must take into account an array of factors, such as the characteristics of the underlying storage device, the application's access patterns, the amount of available disk space and available system resources. Besides, to be useful, performances should be examined in relation to the actual application requirements.

An exhaustive analysis of TSFS performances goes beyond the scope of this user manual. We therefore focus on those aspects that are the most crucial from an application design standpoint. Above all, the purpose of this chapter is to provide application designers with a performance baseline for various storage devices and configurations.

4.1 Preliminary Notions

This section covers fundamental concepts related to file system performance, including basic definitions and descriptions of key metrics. Actual performances are discussed in Section 4.2.

4.1.1 Net versus Raw Troughput

Net throughput is measured at the application level. The net write throughput is the amount of application-provided data written per unit of time. Conversely, the net read throughput is the amount of application-requested data read per unit of time.

The net throughput can widely differ from the raw throughput which is measured at the storage device interface. Such disparity can be observed, for instance, when comparing NOR and NAND flash performances under small random write workloads. Although NOR exhibits a far lower raw write throughput than NAND, the net write throughput for NOR under this type of workload can approach, even exceed that of NAND.

The gap between raw and net throughputs is determined by numerous factors. Some obvious, some more subtle. Some under the control of the application designer, some not. Generally speaking, a sound application design is not possible without factoring in at least the most determining factors. These are discussed in Section 4.2 and Section 4.4.

4.1.2 Sustained versus Burst Throughput

Sustained throughput is measured when read/write accesses are performed for an extended period of time without interruption. With no pause available for background tasks to progress, the sustained throughput captures the impact of internal file system tasks (such as garbage collection and block erasing) on overall performances.

On the contrary, burst throughput is measured when pauses between successive accesses (or short streaks of accesses) allow for the progression of background tasks. In this case, internal file system tasks do not affect — or only marginally — the measured throughput. Note that proper synchronization between background tasks execution and application requests is not always possible. As a result, the potential gains associated with burst accesses are not always significant in practice.

Real-life applications usually generate some mix of sustained and burst accesses. Still, designing based on sustained performances often makes more sense in terms of consistently meeting application requirements. Therefore, unless specified, sustained performances are assumed throughout this chapter.

4.1.3 Random versus Sequential Throughput

Random read/write throughput is measured when accesses are performed at random positions. Although large random accesses are possible, most random performance tests focus on small accesses (4KiB is common). Also, unless specified, the probability distribution is assumed to be uniform.

Random throughput can be measured at the storage device interface or at the application level. The former is referred to as the raw random throughput while the latter is known as the net random throughput.

While real-life situations are generally not made of purely random operations, random throughput is still of critical importance for many applications. Besides, it represents a hallmark of file system performances since high random throughput is usually much trickier to obtain than high sequential throughput.

Sequential read/write throughput is measured when accesses are performed at contiguous positions. Although small sequential accesses are possible, most sequential performance tests focus on large accesses. A high sequential throughput is often required for audio/video streaming applications, to mention but a few.

TSFS is designed in such a way that the random or sequential nature of read/write accesses have little impact on performance. Consequently, we do not cover sequential performances as such, although we do discuss the impact of the access size on random performances.

4.1.4 CPU Bound versus I/O Bound Perfomances

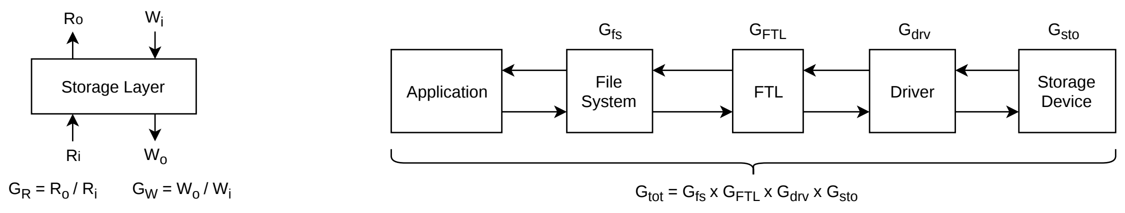
Performances that are primarily limited by the finite amount of CPU power — or sometimes other system resources — are said to be CPU bound. Conversely, performances that are primarily limited by the finite speed of I/O operations are said to be I/O bound.

Real-life performances are rarely purely CPU bound or I/O bound. More often than not, performances are limited by a complex mix of factors including CPU-related and I/O-related factors. CPU-related factors include limited CPU power, memory latency and limited interconnect bandwidth, to name but a few. I/O-related factors include limited bus bandwidth, internal flash load/program latency and managed flash FTL-induced read/write amplification.

4.1.5 Read/Write Amplification

When data is written to or read from a file, it typically goes through a series of layers, most notably, the file system itself, possibly an FTL, and the storage device driver. Taken together these layers are commonly referred to as a storage stack. Each layer of the stack may pass more data to the next layer than it received from the previous one. On the read path this phenomena is called *read amplification*, whereas, on the write path, it is known as *write amplification*.

The amplification of a given layer is the ratio of data exiting to data entering that layer. This is depicted in Figure 13(a). The end-to-end write amplification is the product of each layer's contribution. This is shown Figure 13(b).



(a) The amplification of a given layer is the ratio of data exiting to data entering that layer.

(b) The total write amplification is the product of all the intermediate layers' partial write amplifications.

Figure 13 – Amplification can be observed at various levels within the storage stack.

Write amplification can arise at various levels and for various reasons within a storage stack:

- at the file system level: file system metadata overhead, write granularity and alignment restrictions;
- at the FTL level: garbage collection and logical-to-physical mapping update overhead;
- at the physical memory level: write granularity and alignment restrictions.

Read amplification is also caused by different factors:

- at the file system level: file system lookup overhead, read granularity and alignment restrictions;
- at the FTL level: logical-to-physical address translation;
- at the physical memory level: read granularity and alignment restrictions.

4.1.6 Read and Write Cache

There are many different kinds of caching (or buffering) mechanisms, all serving various purposes. Caches can be roughly characterized by:

- their size: the amount of dedicated RAM;
- their location: at the driver level, the block device abstraction level, the file system core level, etc.;
- their access type: read-only, write-through, write-back;
- their content: objects, raw blocks, metadata, data, etc.;
- their eviction policy: FIFO, LRU, LFU, etc.

These parameters can all greatly affect performance measures. Therefore, a thorough performance assessment cannot be done without including a detailed description of the involved caching/buffering mechanisms.

Because of its log structure, TSFS does not need a complex write cache. Instead, it uses a simple write buffer which size is automatically adjusted based on the underlying storage device technology and the available RAM (the application designer can also manually adjust the size using advanced configurations, as discussed in Chapter 6). The impact of TSFS write buffering on performances is discussed in Section 4.2.6.

TSFS also features an object-level metadata read cache to improve overall performances, especially on storage technologies with page-based read operations. The impact of the read cache on write and read performances are discussed in Section 4.2.5 and Section 4.4.3 respectively.

4.2 Average Net Write Throughput

Average net write performances can drastically vary across storage devices, platforms and configurations. It is mostly determined by:

- the raw write throughput;
- the update size and alignment;
- the transaction size;
- the amount of available disk space;
- read caching;
- write buffering;
- other system-level factors like CPU power.

4.2.1 Raw Throughput

The most obvious factor influencing the net write throughput is the write speed of the underlying storage device, which we refer to as the *raw write throughput*. The raw write throughput represents an absolute upper bound on the net write throughput. No matter how efficient the file system is, and no matter how much care has been put into the application design, the raw write throughput can never be exceeded — that is, of course, disregarding the possible effect of read/write caches.

Since we are primarily interested in sustained performances, unless stated otherwise, the raw write throughput for flash devices takes into account the block erase time. Table 5 shows typical combined erase/write throughputs for bare flash devices, namely NOR, NAND SLC and NAND MLC. These performances come from a combination of the limited bus speed and flash program/erase latency.

NOR	NAND (SLC)	NAND (MLC)
200KB/s	10MB/s	5MB/s

Table 5 – Typical sustained erase/write throughputs for bare flash devices.

Managed flash devices, on the other hand, are more complex and unpredictable. This is due to the embedded FTL, which design is largely kept secret by manufacturers. Figure 14 shows throughputs measured on a class 10 SD card. We can see that the write throughput varies widely with the size of the write operations.

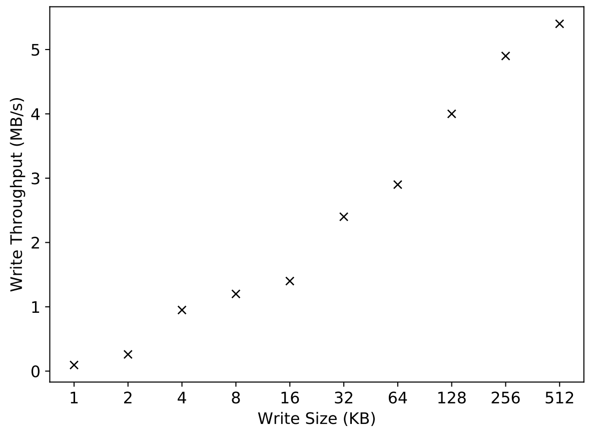


Figure 14 – Write speeds sampled from a class 10 SD card for various write sizes.

Being log-structured, TSFS performs mostly large sequential write accesses. Consequently, it shows little sensitivity to the poor raw random performances of managed flash devices. The most notable exception to this rule occurs when long streaks of short transactions are performed. This particular topic is covered in Section 4.2.3.

4.2.2 Update Size and Alignment

Larger updates usually yield higher average write throughputs. One of the reason for this, is the way that files are subdivided into elementary data blocks called *extents* — other file systems call them *sectors* or simply *blocks*. An extent can only be updated as a whole, partial updates being only possible through a read-modify-write procedure. Partial updates can occur either as the result of a small or an unaligned write access. This is illustrated by Figure 15(a) and Figure 15(b) respectively.

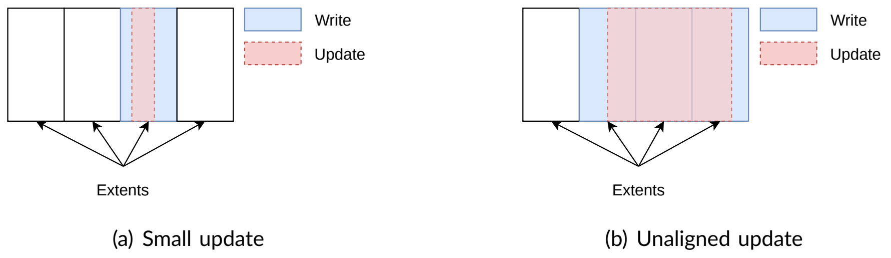


Figure 15 – Partial updates are the result of small and/or unaligned write accesses.

The read-modify-write procedure affects write performances in two different ways:

1. more data is written than updated, which implies some additional write amplification;
2. the updated extent must first be read, which further adds to the update time.

For optimal write performances, application designers should aim for large updates (at least as large as an extent). TSFS default extent size is 512 bytes, but can be adjusted on a per-file basis using `tsfs_file_extent_min_sz_set()`. Updates should also be aligned on extent boundaries as much as possible, although the importance of proper alignment diminishes as the update size grows.

Larger updates are associated with improved write performances for another reason. As updates are performed, internal file system metadata must be updated as well. Because TSFS maintains a near-constant metadata update overhead as the update size increases, larger updates lead to a lower write amplification (shown in Figure 16) and therefore, a higher write throughput.

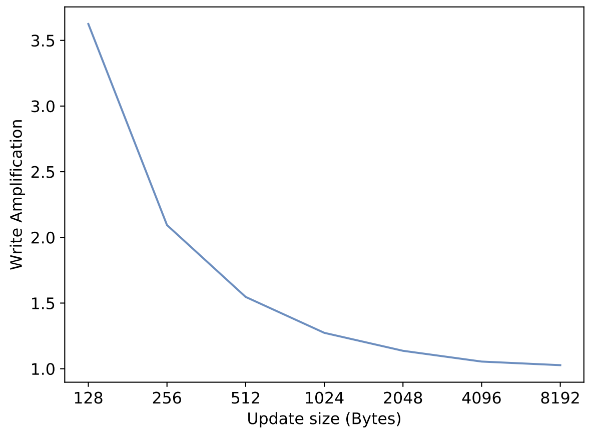


Figure 16 – Write amplification caused by metadata update for various update sizes.

4.2.3 Transaction Size

As previously mentioned in Section 2.3.3, a write transaction is a sequence of write operations that either succeeds or fails as a whole. The number of write operations included in a single transaction can vary from 1 to infinity, such that the amount of data written can also vary quite a lot.

The amount of data written per transaction can have a significant impact on write performances. Each time a transaction ends, the internal write buffer must be written to the storage device, even though it may not be full. on a device characterized by a coarse write granularity (such as NAND and NAND-based devices), a workload dominated by small transactions can cause significant write amplification and severely hamper net write performances.

On managed flash devices, write performances can suffer even for transactions larger than a page, due to the very low raw random write throughput (see Section 4.2.1). On these devices, the application designer should strive for large transactions to obtain the best possible performances.

On the contrary, NOR flash devices are not very sensitive to the transaction size, due to fine-grained program operations. When compared to NAND or NAND-based devices, the average net write throughput can even be higher on NOR for very small transactions, despite the raw write throughput being more than an order of magnitude lower.

4.2.4 Available Disk Space

In log-structured file systems (or FTLs), updates are performed out-of-place (using COW update schemes), leaving invalid pieces of data — old versions of updated data — behind. Eventually, the space occupied by these invalid chunks of data must be reclaimed by the file system. This process is called *garbage collection* and is illustrated in Figure 17.

On flash memories, pieces of data that are still valid must be moved somewhere else as space is reclaimed. This is needed so that erase blocks can be freed, erased and reused. This data relocation

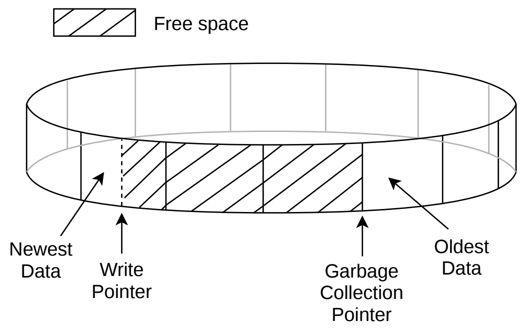


Figure 17 – Garbage collection in a log-structured file system.

process generates additional write amplification. The less available disk space there is, the more often garbage collection must be performed and the higher the write amplification becomes.

To get a sense of how available disk space is tied to write amplification — and ultimately, net write throughput —, we conduct a simple performance analysis assuming uniformly distributed random write accesses on a purely log-structured file system.

To further ease the discussion, we purposely neglect any metadata overhead and consider only equal-sized, aligned updates. The update size, u, is a multiple of the underlying page size — such that there are no partial page updates.

The probability $P(U)$ that a chunk of size u is updated is

$$P(U) = \frac{u}{r \times m} \tag{4.1}$$

where r is the media fill ratio (i.e. the ratio of the data set size to the media size) and m is the media size. The probability $P(V)$ that the chunk is still valid after n updates is

$$P(V) = \Big(1 - P(U)\Big)^n = \left(1 - \frac{u}{r \times m}\right)^n \quad . \tag{4.2}$$

The number of updates that can be performed in one revolution (that is, before the append position of the log returns to its current position) depends on the media size m and on the amount of collected data. In the case of uniformly distributed updates, the amount of data collected over one revolution, f is

$$c = mP(V) \quad . \tag{4.3}$$

The number of updates during one revolution is

$$n = \frac{m - c}{u} = \frac{m(1 - P(V))}{u} \tag{4.4}$$

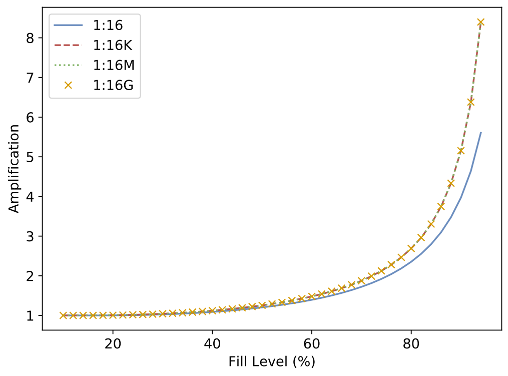
and thus, the probability for an extent to be valid upon collection must satisfy

$$P(V) = \left(1 - \frac{u}{r \times m}\right)^{\frac{m(1-P(V))}{u}} . \tag{4.5}$$

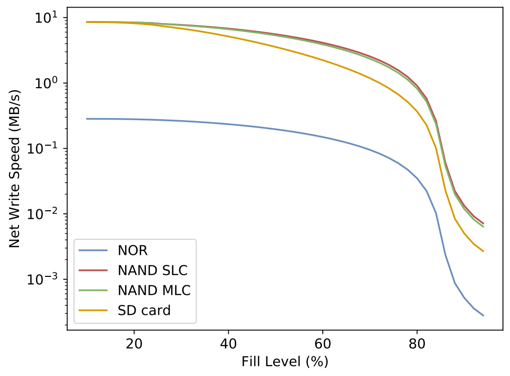
Besides, the write amplification introduced by garbage collection is

$$g_f = \frac{1}{1 - P(V)} . \tag{4.6}$$

Combining Equation 4.5 and Equation 4.6 and solving for various fill ratios yields the curves shown in Figure 18(a). The four curves correspond to four different update to media size ratio (u/m): 1:16, 1:16K, 1:16M, 1:16G. Figure 18(b) shows the corresponding impact on average write throughput for typical storage devices.



(a) Write amplification versus media fill ratio.



(b) Net write throughput versus media fill ratio.

Figure 18 – Impact of the fill ratio on garbage collection-induced write amplification and average net write throughput for 4KiB uniformly distributed random accesses.

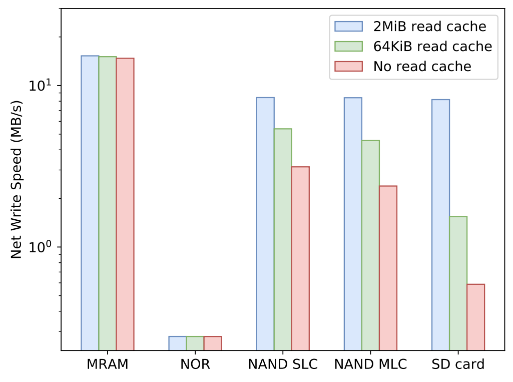
From the preceding analysis, we can make a number of important observations:

- For usual update to media size ratio, the write amplification depends solely on the fill ratio. For the remaining discussion, we can safely ignore both the update and media size.
- Write throughput can be severely hampered by garbage collection when there is too little free space left. The exact amount of free space needed to maintain a given level of performance depends on the application access patterns.
- Write performances of managed flash devices are also affected by garbage collection. However managed flash devices reserve hidden disk space for garbage collection, effectively reducing the maximum write amplification that can be measured from an external standpoint.
- Under the uniform distribution assumption, the cyclic garbage collection algorithm minimizes the amount of valid data to be relocated. Therefore, the curves shown in Figure 18(a) are absolute lower bounds on the write amplification for uniformly distributed write accesses. This holds true for any flash file system or FTL.

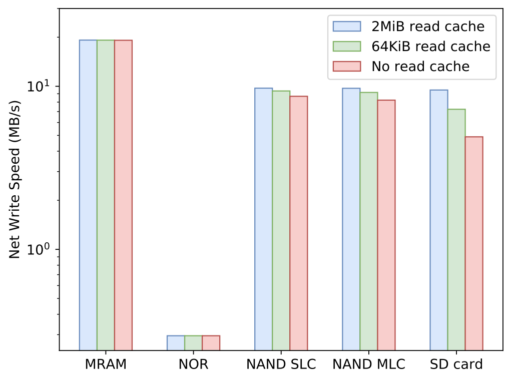
4.2.5 Read Caching

Writing to a file first involves looking for data blocks to be updated. As such, any improvement to read performances can also lead to improved write performances. Various factors contributing to the average net read throughput are covered in Section 4.4. For now, we specifically focus on read caching, as it has the most tangible impact on write performances. It is also the one factor that application designers have the most control over.

Figure 19 illustrates how the read cache size can affect the average net write throughput on typical storage devices. Performance gains are particularly obvious for small write accesses on devices exhibiting a fair amount of read amplification such as NAND-based devices (either raw or managed). Unsurprisingly, the read cache is of little help on NOR flash, where byte-level read accesses are allowed.



(a) 4KiB updates



(b) 64KiB updates

Figure 19 – Average net random write throughput for various storage devices and read cache configurations for a 256MiB data set.

4.2.6 Write Buffering

TSFS's internal write buffer must be at least as big as the elementary write block (i.e. a NAND flash page or SD card sector). On raw flash devices, increasing the write buffer beyond the page size is of little help. This is due to write size having only a marginal effect on the raw write throughput.

As mentioned in Section 4.2.1, the situation is different on managed flash devices, where the write size has a huge impact on the raw write throughput. On these devices, a large write buffer can widely improve average net write performances.

TSFS automatically adjusts the size of the write buffer to obtain the best possible performances, given the underlying storage device and available RAM. The application designer can also manually adjust the size of the write buffer through advanced configurations. This is covered in Chapter 6.

4.2.7 Other Factors

Many system-level factors — like CPU power, external RAM latency, controller design, interrupt-latency, kernel design — can have an impact on write performances. For medium and large read/write payloads, performances tend to be primarily I/O bound.

On NOR flash devices, the combined write/erase throughput is usually not high enough to expose the influence of system-level bottlenecks.

On NAND flash and NAND-based flash devices, performances are dictated by different factors depending on the payload size. For small payloads, performances are mostly limited by the write amplification resulting from coarse write granularity. For large payloads, performances are mostly limited by the raw throughput of the device. In either case, the effect of system-level bottlenecks is not significant.

On RAM-like technologies, the situation is different. The combination of fine-grained write operations and high raw throughputs can make system-level limitations more significant. Figure 20 shows the impact of limited CPU power on a RAM-like storage device with a 20MB/s symmetric read/write throughput. CPU-limited curves are computed assuming 16 and 128 KOPS (kilo read/write operations per second) limits, which are fairly typical figures for a modest low-power processor and a more capable application processor respectively.

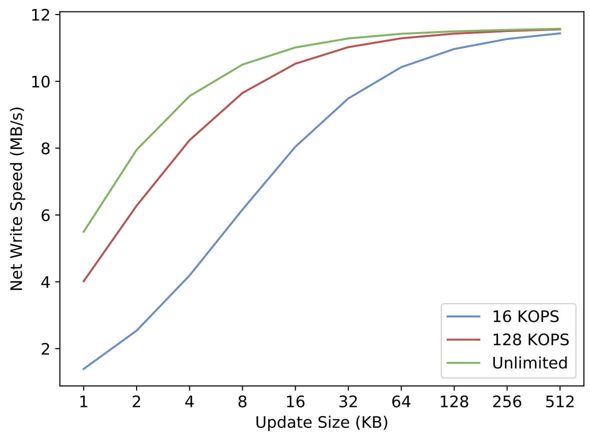


Figure 20 – On RAM-like devices, performances can be limited by system-level bottlenecks.

4.3 Worst Case Write Latency

Write latency is another important aspect of file system performances. It is the amount of time needed to complete a write operation. The write latency may vary across storage devices, file systems and file system configurations. Most importantly, it may vary from one write operation to another. This is why we are interested in the worst case latency, that is, the maximum amount of time needed for a write operation to complete.

On flash-based storage devices, the worst case write latency is essentially determined by:

- the block erase time;
- the latency-throughput trade-off.

4.3.1 Block Erase Time

Before a flash block can be written to, it must be erased. Table 6 shows plausible performances for a 64KiB block erase and a 4KiB page program operation on NOR, NAND SLC and NAND MLC.

Considering these numbers, we can see that the longest write operation can last much longer than the shortest one: 67 times longer for NOR, 25 times for NAND SLC and 10 times for NAND MLC.

	NOR	NAND (SLC)	NAND (MLC)
4KiB page program	15ms	200us	500us
64KiB block erase	1s	5ms	5ms

Table 6 – Typical write and erase times for NOR, NAND SLC and NAND MLC.

Because it is inherent to the physics of flash cells, not much can be done to reduce the impact of the block erase time on the worst case write latency. Still, application designers should be aware of this particular aspect when settling on high-level performance requirements.

4.3.2 Latency vs Throughput Trade-off

As mentioned in Section 4.2.4, the average write amplification introduced by garbage collection is a function of the fill ratio. However, the instantaneous amplification can be much higher than the average, which is significant in terms of worst case write latency.

TSFS limits the maximum amount of garbage collection-induced write amplification by spreading the garbage collection across multiple write requests. In its default configuration, TSFS guarantees a maximum write amplification of 10, which is comparable to the worst case latency associated with flash block erase times (see Section 4.3.1). However, it is possible to adjust the maximum allowable amplification through advanced configurations. This is covered in Chapter 6.

There is a fundamental trade-off between the worst case latency and the average throughput. This relation can be observed in many systems and TSFS makes no exception. Figure 21 shows the relation between the net write throughput and the fill ratio for different maximum write amplifications.

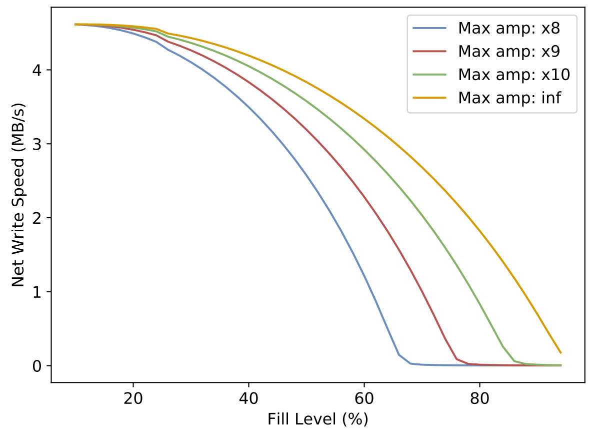


Figure 21 – The evolution of the average net write throughput as the fill ratio varies, is affected by the maximum allowed write amplification. Throughput values are computed for a hypothetical 256MiB NAND SLC device with a write throughput of 10MB/s, a read throughput of 30MB/s and a 64KiB read cache.

4.4 Average Net Read Throughput

The average net read throughput can vary a lot across different storage devices, platforms and configurations. It is mostly determined by:

- the raw read throughput;
- the read size and alignment;
- read caching;
- other system-level factors.

4.4.1 Raw Read Throughput

Table 7 shows typical sustained read throughputs for bare flash devices, namely NOR, NAND SLC and NAND MLC. These performances come from a combination of the limited bus speed and the internal page load latency.

NOR	NAND (SLC)	NAND (MLC)
30MB/s	30MB/s	20MB/s

Table 7 – Typical sustained read throughputs for bare flash devices.

As with write performances, managed flash devices exhibit raw throughputs that are highly dependent on the access size. Again, this is due to the embedded FTL and, more precisely, to the logical-to-physical address resolution process. Figure 22 shows throughputs measured on a class 10 SD card.

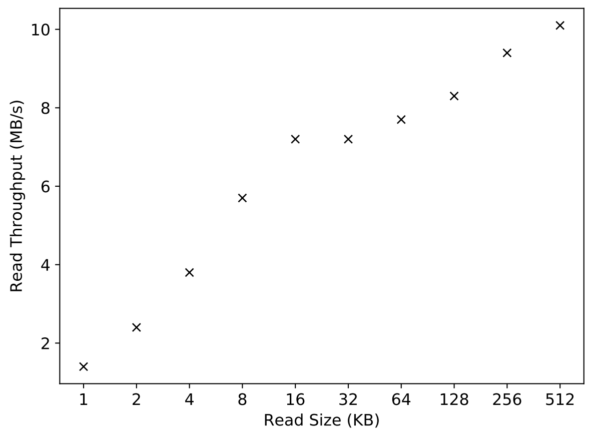


Figure 22 – Read speeds sampled from a class 10 SD card for various read sizes.

4.4.2 Read Size and Alignment

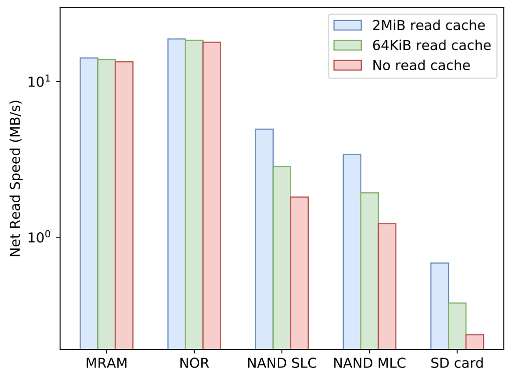
Before data requested by the application can be read, it must first be located. The lookup process typically generates small scattered read operations, such that the amount of read amplification also depends on the read granularity of the underlying storage device.

On NOR flash and RAM-like devices, fined-grained read accesses provide the smallest possible lookup overhead. On the contrary, the coarse read granularity of NAND and NAND-based flash devices cause a much higher read amplification. The worst case is typically observed on managed flash devices which tend to have very low raw random read throughputs (see Figure 22). In this case, the read cache can be used to mitigate the lookup overhead (see Section 4.4.3).

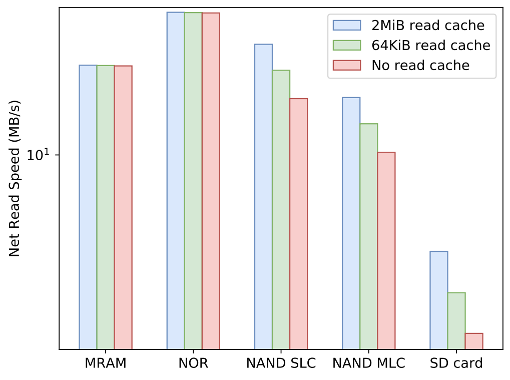
Reading data from a NAND memory triggers one or more internal page load operations. The typical NAND page size is 4KiB for SLC and 8KiB for MLC. Reading less than a page therefore results in read amplification and lower net read throughput. For this reason, SD card transfers should be at least as large as a typical NAND page and aligned on page boundaries, even though smaller transfers are allowed.

4.4.3 Read Caching

Read caching improves random read performances by limiting the amount of read operations actually reaching the storage device during extent lookups. Figure 23 shows the impact of the read cache size on typical storage devices.



(a) 4kB updates



(b) 64kB updates

Figure 23 – Average net random read throughput for various storage devices and read cache configurations for a 256MB data set.

On storage devices with fast and fine-grained read operations, like NOR flash and RAM-like memories, the read cache is not needed. However, on NAND and NAND-based devices, the read cache is often essential to achieve the required read performances.

4.4.4 Other Factors

System-level bottlenecks like limited CPU power can greatly affect read performances, even more so than write performances since the average read throughput is usually higher than the average write throughput.

On NOR flash and RAM-like technologies, the combination of byte-level read accesses and high raw throughputs can lead to CPU bound performances. Figure 24 shows the impact of limited CPU power on a RAM-like storage device with a 20MB/s symmetric read/write throughput. CPU-limited curves are computed assuming 16 and 128 KOPS limits.

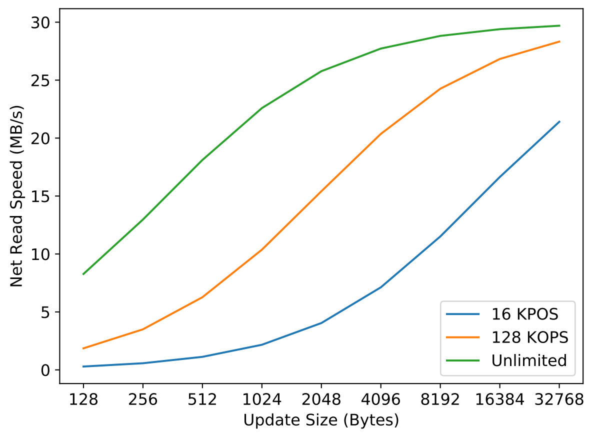


Figure 24 – CPU bound performances on a RAM-like storage device with a 20/MB/s symmetric read/write throughput.

Chapter

5

Usage

5.1 Initializing and Terminating a File Sytem Instance

TSFS initialization and termination is a straightforward process. A typical example of a complete TSFS instance life cycle is given in Listing 1.

tsfs_create() allocates a small part of the memory needed by the new instance using the default memory allocator. The memory allocation is completed by tsfs_mount(), this time, using the memory segment provided by the application — i.e. the p_seg member of the tsfs_cfg_t configuration structure.

Once tsfs_unmount() has been called, the provided memory segment can safely be used for other purposes, until tsfs_mount() is called again. Besides, the memory allocated by tsfs_create() using the default allocator can be freed using tsfs_destroy(), provided that the default allocator supports memory freeing.

Both tsfs_create() and tsfs_mount() can fail because of a lack of available memory. In this case, both functions return RTNC_NO_RESOURCE. Without extended configurations — i.e. with the p_ext_cfg member of the tsfs_cfg_t structure set to NULL —, internal TSFS parameters are automatically adjusted to fit the given amount of memory. The minimum amount of memory that TSFS must be given depends on the underlying storage technology.

```
tsfs_cfg_t fs_cfg;
bp_media_hndl_t media_hndl;
uint8_t t_seg[8 * 1024];
int rtn = RTNC_SUCCESS;


// Instantiate a media instance here to obtain a valid media handle.
// See the BASEplatform Reference Manual for details.
// rtn = bp_ramdisk_create(...
// Create the TSFS instance.
fs_cfg.media_hndl = media_hndl; // Media instance to be tied to the TSFS instance.
fs_cfg.p_seg = (void *)t_seg;   // Segment of memory dedicated to the file system.
fs_cfg.seg_sz = sizeof(t_seg);  // Size of the given memory segment.
fs_cfg.max_entry_cnt = 10u;     // Maximum number of opened files/directories.
fs_cfg.p_tdata = NULL;          // Always set to NULL.
fs_cfg.p_ext_cfg = NULL;        // Set to NULL for default.
rtn = tsfs_create("fs0", &fs_cfg);
if (rtn != RTNC_SUCCESS) { /* Error management */ }


// Format the TSFS instance (set last argument to NULL for default).
rtn = tsfs_format("fs0", NULL);
if (rtn != RTNC_SUCCESS) { /* Error management */ }


// Mount the TSFS instance.
rtn = tsfs_mount("fs0");
if (rtn != RTNC_SUCCESS) { /* Error management */ }

// ...
// Here the file system is available for directory, file and snapshot accesses.
// ...

// Unmount the TSFS instance.
rtn = tsfs_unmount("fs0");
if (rtn != RTNC_SUCCESS) { /* Error management */ }

// Destroy the TSFS instance.
rtn = tsfs_destroy("fs0");
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 1 – TSFS initialization example.

5.2 Creating a File or Directory

Listing 2 illustrates how files and directories can be created.

A new file can be created using `tsfs_file_create()`. If a file or directory already exists at the same location, `RTNC_ALREADY_EXIST` is returned and the original file or directory is left untouched. The existing file or directory can also be removed using `tsfs_file_delete()` or `tsfs_dir_delete()` if needed.

If the parent directory does not exist, the function returns `RTNC_NOT_FOUND`. It is up to the application to create the missing parent directory.

A new directory can be created using `tsfs_dir_create()`. The behaviour for the directory creation interface is identical to that of the file counterpart.

```
int rtn;

//
// TSFS instance setup goes here.
//

// An attempt to create a new file in a directory that
//  does not exist fails with RTNC_NOT_FOUND.
rtn = tsfs_file_create("fs0/d0/f0.txt");
if (rtn != RTNC_SUCCESS) { /* Error management */ }

// Create the missing directory.
rtn = tsfs_dir_create("fs0/d0");
if (rtn != RTNC_SUCCESS) { /* Error management */ }

// Create a new file inside the previously created directory.
rtn = tsfs_file_create("fs0/d0/f0.txt");
if (rtn != RTNC_SUCCESS) { /* Error management */ }

// An attempt to create the same file again fails with RTNC_ALREADY_EXIST.
rtn = tsfs_file_create("fs0/d0/f0.txt");
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 2 – File and directory creation example

5.3 Opening and Closing a File or Directory

Once a file or directory has been created, it can be opened using `tsfs_file_open()` or `tsfs_dir_open()`, with the same path parameter as the one used for the creation.

The file/directory opening functions return an opened file/directory handle, which can be used to perform further read and write operations. Each opened file/directory has an associated internal position that is automatically incremented after each read or write operation. The internal current position is set to 0 upon opening a file or directory. The internal file position can be adjusted by the application using `tsfs_file_seek()`. Directories can only be read from the first entry to the last one.

An opened file or directory handle is valid as long as it is not closed using `tsfs_file_close()` or `tsfs_dir_close()`. Using a closed handle yields undefined behaviour. The maximum number of files and directories that can be simultaneously opened is determined by the `max_entry_cnt` member of the `tsfs_cfg_t` configuration structure.

5.4 Writing to a File

Once opened, a file can be written to using `tsfs_file_write()`. The actual number of bytes written to the file is returned through the last parameter. If enough media space is available to complete the requested write operation, the returned write size is equal to the requested size. If the media becomes full before the write operation completes, `RTNC_FULL` is returned and the returned write size indicates the number of bytes written before the media full condition occurred.

A file can also be written to using `tsfs_file_append()`. Unlike `tsfs_file_write()`, `tsfs_file_append()` always writes at the end of the selected file (hence append), independent of the current file position.

```
int rtn;
tsfs_file_hndl_t fhndl;

//
// TSFS instance setup goes here.
//

// Create a new file.
rtn = tsfs_file_create("fs0/f0.txt");
if (rtn != RTNC_SUCCESS) { /* Error management */ }

// Open the newly created file.
rtn = tsfs_file_open("fs0/f0.txt", &fhndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }

// Write something at the begining of the file.
rtn = tsfs_file_write(fhndl, "hello world!", sizeof("hello world!")+1, &sz);
if (rtn != RTNC_SUCCESS) { /* Error management */ }

// Set the current position to the beginning of the file.
rtn = tsfs_file_seek(fhndl, 0u, TSFS_FILE_SEEK_SET)
if (rtn != RTNC_SUCCESS) { /* Error management */ }

// Write a new message.
rtn = tsfs_file_write(fhndl, "hello you!", sizeof("hello you!")+1, &sz);
if (rtn != RTNC_SUCCESS) { /* Error management */ }

// Set the current position to the beginning of the file.
rtn = tsfs_file_seek(fhndl, 0u, TSFS_FILE_SEEK_SET)
if (rtn != RTNC_SUCCESS) { /* Error management */ }

// Write a second message after the first one.
rtn = tsfs_file_append(fhndl, "hello there!", sizeof("hello there!")+1, &sz);
if (rtn != RTNC_SUCCESS) { /* Error management */ }

rtn = tsfs_file_close(fhndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 3 – File writing example

5.5 Reading From a File

A file can be read from using `tsfs_file_read()` as illustrated in Listing 4. Notice how the end of the file is detected using the last function parameter, which returns the actual amount of bytes read from the file. As long as data is left in the file, the actual read size remains equal to the requested read size. If, and only if, the end of the file is reached, the returned read size is smaller than the requested size. If an error occurs the returned read size value is undefined and should be ignored.

```
uint8_t t_buf[100];
tsfs_file_hndl_t fhndl;
int rtn;

//
// TSFS instance setup and "fs0/f0.txt" file creation/writing goes here.
//

// Open the file to be read (must have been previously created).
rtn = tsfs_file_open("fs0/f0.txt", &fhndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }

// Read from opened file, 100 bytes at a time, until the end is reached.
do {
   rtn = tsfs_file_read(fhndl, &t_buf[0], sizeof(t_buf), &sz);
   if (rtn != RTNC_SUCCESS) { /* Error management */ }

   // Do something with the read data here.

} while (sz < sizeof(t_buf));

// Close the file.
rtn = tsfs_file_close(fhndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 4 – Example file read

5.6 Reading a Directory

The names of all the files and directories contained in a directory can be retrieved using `tsfs_dir_read()`. This is illustrated in Listing 5.

The application is responsible for providing a buffer long enough to accommodate the longest file or directory name. In Listing 5, the buffer used can contain up to `TSFS_MAX_PATH_LEN` + 1 characters, which is guaranteed to be enough for any possible file or directory name.

```
char t_buf[TSFS_MAX_PATH_LEN + 1];
tsfs_dir_hndl_t dhndl;
int rtn;

//
// TSFS instance setup and "fs0/d0" directory creation goes here.
//

// Open the file to be read (must have been previously created).
rtn = tsfs_dir_open("fs0/d0", &dhndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }

// Read the opened directory, one child file/directory at a time.
do {
    rtn = tsfs_dir_read(dhndl, &t_buf[0], sizeof(t_buf));
    if (rtn != RTNC_SUCCESS) { /* Error management */ }

    // Print the child name.
    printf("%s\n", t_buf);

} while (t_buf[0] != '\0');

// Close the file.
rtn = tsfs_dir_close(dhndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 5 – Example directory read

5.7 Deleting a File or Directory

A file can be deleted using `tsfs_file_delete()`. It is safe to delete an opened file. Further access to this file will simply return an error, indicating that the file was not found. This is illustrated in Listing 6.

A directory can be deleted using `tsfs_dir_delete()`. An opened directory can also be safely deleted. However, it is not possible to delete a non-empty directory. If such an attempt is made, `RTNC_INVALID_OP` is returned and the directory is left untouched.

```
uint8_t byte;
tsfs_file_hndl_t fhndl;
tsfs_file_size_t sz;
int rtn;

//
// TSFS instance setup and "fs0/f0.txt" file creation goes here.
//

// Open the file to be read (must have been previously created).
rtn = tsfs_file_open("fs0/f0.txt", &fhndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }

// Delete "fs0/f0.txt" while it is opened.
rtn = tsfs_file_delete("fs0/f0.txt");
if (rtn != RTNC_SUCCESS) { /* Error management */ }

rtn = tsfs_file_read(fhndl, &byte, 1u, &sz);
if (rtn == RTNC_NOT_FOUND) {

    rtn = tsfs_file_close(fhndl);
    if (rtn != RTNC_SUCCESS);

} else {
    /* Error management */
}
```

Listing 6 – Example of opened file deletion.

5.8 Truncating a File

A file can be extended or shrunk using `tsfs_file_truncate()`. If the requested file size is lower than the current size the extraneous data is discarded. If the requested file size is higher than the current size, zeros are appended at the end of the file to fill the gap.

An opened file can safely be extended or shrunk. If the current position of the opened file is beyond the size of the truncated file, the next call will report an actual read size of 0 byte. This is illustrated in Listing 7.

```
uint8_t t_buf[100];
tsfs_file_hndl_t fhndl;
tsfs_file_size_t sz;
int rtn;

//
// TSFS instance setup and "fs0/f0.txt" file creation goes here.
//

// Open a file.
rtn = tsfs_file_open("fs0/f0.txt", &fhndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }

// Write a hundred bytes.
rtn = tsfs_file_write(fnhdl, t_buf, sizeof(t_buf), &sz);
if (rtn != RTNC_SUCCESS) { /* Error management */ }

// Set the file position 10 bytes behind.
rtn = tsfs_file_seek(fhndl, -10, TSFS_FILE_SEEK_CUR);
if (rtn != RTNC_SUCCESS) { /* Error management */ }

// Reduce the file size to 50 bytes.
rtn = tsfs_file_truncate("fs0/f0.txt", sizeof(t_buf)/2);
if (rtn != RTNC_SUCCESS) { /* Error management */ }

// An attempt to read beyond the first half the original file
// will report 0 byte read as the second half of the original
// file has been previously discarded.
rtn = tsfs_file_read(fhndl, &t_buf[0], 1u, &sz);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 7 – Example of opened file truncation

5.9 Starting and Ending a Write Transaction

This section is specifically about TSFS write transactions. For a broader discussion on write transactions, please refer to Section 2.3.3.

Any combination of the following operations are guaranteed to be atomically executed if they are part of the same write transaction:

- `tsfs_file_create()`

- `tsfs_file_delete()`
- `tsfs_file_write()`
- `tsfs_file_append()`
- `tsfs_file_truncate()`
- `tsfs_dir_create()`
- `tsfs_sshot_create()`
- `tsfs_sshot_delete()`

A write transaction is ended by the application using `tsfs_commit()`. The end of a write transaction coincides with the beginning of the next one. Therefore, no explicit function call is needed to start a new write transaction. The very first write transaction is automatically started after `tsfs_mount()` successfully completes.

Only one write transaction can exist at a time. The current state of the file system, up to the most recent operation of the current write transaction, is called the working state. The working state is the one accessed by default, using a path of the form

```
<file system name>/<file or directory path>.
```

The state of the file system as of the latest commit can also be accessed. The latest committed state is specified using the `.c` and `.latest` built-in path components:

```
<file system name>/.c/.latest/<file or directory path>.
```

Alternatively, the working state (i.e. the latest uncommitted state) can be accessed using the `.uc` built-in path component:

```
<file system name>/.uc/.latest/<file or directory path>.
```

5.10 Recovering from Unexpected Interruptions

Listing 8 shows how write transactions can be used to keep two files in sync at all time, independent of possible interruptions. This example contains three write transactions. The first one begins right after the call to `tsfs_mount()`. The first transaction ends where the second one begins, that is, at the first `tsfs_commit()`. Finally, the second transaction ends where the third transaction begins, that is, at the second `tsfs_commit()`.

If the first commit fails (or any function call before), the file system returns to its original (empty) state after the next mount cycle (as if the two files were never created). Otherwise, both files are guaranteed to exist after the next mount cycle. Under no circumstances, may only one of the two files exist.

If the second commit fails (or any function call before), both files remain empty. Otherwise, both files contain "hello world!". Under no circumstances, may only one of the two files be empty.

```
tsfs_file_hndl_t fhndl;
tsfs_file_sz_t wr_sz;
int rtn;

// Mount the file system (assuming that the 'fs0' file system instance
// has been previously created). A new transaction is implicitly started.
rtn = tsfs_mount("fs0");
if (rtn != RTNC_SUCCESS) { /* Error handling */ }

// Create a new file.
rtn = tsfs_file_create("fs0/f0.txt");
if (rtn != RTNC_SUCCESS) { /* Error handling */ }

rtn = tsfs_file_create("fs0/f1.txt");
if (rtn != RTNC_SUCCESS) { /* Error handling */ }

 // End the current write transaction and start the next one.
rtn = tsfs_commit("fs0");
if (rtn != RTNC_SUCCESS) { /* Error handling */ }


// Write something to the first file.
rtn = tsfs_file_open("fs0/f0.txt", &fhndl);
if (rtn != RTNC_SUCCESS) { /* Error handling */ }

rtn = tsfs_file_write(fhndl, "", sizeof("hello world!"), &wr_sz);
if (rtn != RTNC_SUCCESS) { /* Error handling */ }

rtn = tsfs_file_close(fhndl);
if (rtn != RTNC_SUCCESS) { /* Error handling */ }


// Write something to the second file.
rtn = tsfs_file_open("fs0/f1", &fhndl);
if (rtn != RTNC_SUCCESS) { /* Error handling */ }

rtn = tsfs_file_write(fhndl, "white", sizeof("hello world!"), &wr_sz);
if (rtn != RTNC_SUCCESS) { /* Error handling */ }

rtn = tsfs_file_close(fhndl);
if (rtn != RTNC_SUCCESS) { /* Error handling */ }


// Ends the current write transaction and start the next one.
rtn = tsfs_commit("fs0");
if (rtn != RTNC_SUCCESS) { /* Error handling */ }
```

Listing 8 – Beginning and ending a transaction.

5.11 Creating and Deleting Snapshots

A snapshot can be created using `tsfs_sshot_create()`. Once a snapshot of the working state has been created, the saved state is guaranteed to remain available for further read-only operations, as long as it is not explicitly deleted using `tsfs_sshot_delete()`.

A snapshot cannot be modified, it can only be deleted. Any attempt to perform a write operation on a snapshot will return `RTNC_INVALID_OP`.

The number of snapshots that can simultaneously exist is only limited by the size of the underlying media.

Snapshots can be accessed through the path argument of read-only functions. Committed and uncommitted snapshots can be accessed separately. Committed snapshots can be accessed through a path of the form

`<file system name>/.c/.ss/<snapshot name>/<file or directory path>.`

Likewise, uncommitted snapshots can be accessed through a path of the form

`<file system name>/.uc/.ss/<snapshot name>/<file or directory path>.`

Listing 9 shows how snapshots and write transactions can be used to read from a file while it is being updated.

```
tsfs_file_hndl_t fhndl;
tsfs_file_sz_t sz;
char t_buf[50];
int rtn;

// Mount the file system (assuming that the 'fs0' file system instance
// has been previously created). A new transaction is implicitly started.
rtn = tsfs_mount("fs0");
if (rtn != RTNC_SUCCESS) { /* Error handling */ }

// Create a new file.
rtn = tsfs_file_create("fs0/f0.txt");
if (rtn != RTNC_SUCCESS) { /* Error handling */ }

// Write something to the file.
rtn = tsfs_file_open("fs0/f0.txt", &fhndl);
if (rtn != RTNC_SUCCESS) { /* Error handling */ }

rtn = tsfs_file_write(fhndl, "hello world!", sizeof("hello world!")+1u, &sz);
if (rtn != RTNC_SUCCESS) { /* Error handling */ }


// Keep a snapshot of the current state
// (or, equivalently, start a new read transaction).
rtn = tsfs_sshot_create("fs0", "ss0");
if (rtn != RTNC_SUCCESS) { /* Error handling */ }


// Overwrite the original message and close the file.
rtn = tsfs_file_seek(fhndl, 0u, TSFS_FILE_SEEK_SET);
if (rtn != RTNC_SUCCESS) { /* Error handling */ }
```

```
rtn = tsfs_file_write(fhndl, "hello you!", sizeof("hello you!")+1u, &sz);
if (rtn != RTNC_SUCCESS) { /* Error handling */ }

rtn = tsfs_file_close(fhndl);
if (rtn != RTNC_SUCCESS) { /* Error handling */ }


// Read the text from the previously created snapshot.
rtn = tsfs_file_open("fs0/.uc/.ss/ss0/f0.txt", &fhndl);
if (rtn != RTNC_SUCCESS) { /* Error handling */ }

rtn = tsfs_file_read(fhndl, t_buf, sizeof(t_buf), &sz)
if (rtn != RTNC_SUCCESS) { /* Error handling */ }


// Make sure the content is "hello world!".
if (strcmp(t_buf, "hello world!") != 0) {
     printf("Something is wrong!");
     while(1);
}
```

Listing 9 – Managing concurrent read/write operations using snpashots.

Chapter

6 Advanced Configuration

TSFS is controlled by several internal parameters. These parameters are not exposed through TSFS's API for the following reasons:

- Adjusting internal parameters is somewhat complex. A proper tuning requires some understanding of low-level elements of the TSFS design and, more specifically, how they relate to various performance aspects.
- More often than not, the net improvement obtained by tweaking internal parameters is marginal. Internal parameters can thus (and probably should) be ignored for all but the very end of the design flow.
- Because they are not officially part of the public interface, TSFS internal parameters are more prone to change in future versions.

That being said, tuning internal parameters can sometimes be a needed step towards reaching TSFS full potential.

6.1 Extended Configuration

Some internal parameters can be set upon creating a TSFS instance. These parameters are not stored on the media and can be changed without reformatting.

Listing 10 illustrates how the `tsfs_bknd_log_ext_cfg_t` configuration structure can be used to adjust internal TSFS parameters. Individual structure members are described hereafter.

```
#include <souce/tsfs_bknd_log_i.h> // <-- Needed to access advanced config.

media_hndl_t media_hndl;
tsfs_bknd_ext_cfg_t ext_cfg;
tsfs_cfg_t cfg;
uint8_t g_tsfs_mem[8192];
int rtn;

```

```
// Create a media and initialize 'media_hndl' here...

ext_cfg.obj_cache_entry_cnt_log2 = 15;  // 32Ki cache entries.
ext_cfg.obj_cache_assoc_log2 = 4u;      // Associativity of 16.
ext_cfg.rd_cache_blk_cnt_log2 = 1u;     // 2 prefetch cache blocks.
ext_cfg.rd_cache_blk_sz_log2 = 9u;      // 512B prefetch cache blocks.
ext_cfg.wr_buf_sz_log2 = 16u;           // 64KiB write buffer.
ext_cfg.max_gc_wr_amp = 10u;            // Max GC-induced write amp of 10.

cfg.media_hndl = media_hndl;            // Media handle.
cfg.p_seg = g_tsfs_mem;                 // Memory segment dedicated to TSFS.
cfg.seg_sz = sizeof(g_tsfs_mem);        // Size of the memory segment.
cfg.max_entry_cnt = 10u;                // Maximum number of opened files/dirs.
cfg.p_tdata = NULL;                     // Trace data pointer (set to NULL).
cfg.p_ext_cfg = (void *)&ext_cfg;       // Use extended config (NULL for default).

rtn = tsfs_create("fs0", &cfg);
if (rtn != RTNC_SUCCESS) { /* Error handling. */}
```

Listing 10 – TSFS extended configuration example.

6.1.1 `obj_cache_entry_cnt_log2`

`obj_cache_entry_cnt_log2` determines the maximum number of metadata objects in the cache. The default value depends on the amount of available RAM and the underlying storage technology. Higher values tend to increase read/write throughput at the expense of a higher RAM usage. Values between 0 and 32 are accepted. A value of `(uint8_t)-1` disables the object cache.

6.1.2 `obj_cache_assoc_log2`

`obj_cache_assoc_log2` changes the associativity for the metadata object cache. The default value is 3, which corresponds to an associativity of 8. Higher values tend to increase the read/write throughput at the expense of a higher RAM usage. Values between 0 and 5 are accepted.

6.1.3 `rd_cache_blk_cnt_log2`

`rd_cache_blk_cnt_log2` determines the number of blocks in the prefetch cache. The default value is 1, which means 2 blocks. Higher values tend to increase the read/write throughput at the expense of a higher RAM usage. Values between 0 and 32 are accepted.

6.1.4 `rd_cache_blk_sz_log2`

`rd_cache_blk_sz_log2` determines the size of the prefetch cache blocks. The default value is the underlying media's read block size. Changing this value usually yields poorer performances. Values between 0 and 16 are accepted, but the prefetch cache block size must be greater or equal to the media's read block size.

6.1.5 `wr_buf_sz_log2`

`wr_buf_sz_log2` determines the size of the write buffer. The default value depends on the amount of available RAM and the underlying storage technology. Higher values may increase write throughput at the expense of a higher RAM usage. Values between 0 and 32 are accepted, but the size of the write buffer must be greater or equal to the media's write block size.

6.1.6 `max_gc_wr_amp`

`max_gc_wr_amp` determines the maximum amount of write amplification introduced by garbage collection. The default value is 10. Higher values tend to increase the average write throughput at the expense of a higher worst case write latency. Values between 2 and 128 are accepted.

6.2 Stored Parameters

Some internal parameters are stored on the media along with the rest of the file system. These are called *stored parameters*. Stored parameters are determined upon formatting and loaded upon mounting. Once the media is formatted, they cannot be modified.

Stored parameters are automatically determined by TSFS based on the underlying storage technology. However, the application designer can customize these parameters through the `tsfs_bknd_log_sto_params_t` configuration structure. Individual structure members are described hereafter.

```
#include <souce/tsfs_bknd_log_i.h> // <-- Needed to access advanced config.

tsfs_bknd_log_sto_params_t sto_params;
int rtn;


sto_params.blk_sz_log2 = 16u;   // 64KiB GC blocks.
sto_params.ext_sz_log2 = 12u;   // 4KiB default minimum extent size.
sto_params.bf_log2 = 1u;        // 2 children per node.

rtn = tsfs_format("fs0", &sto_params);
if (rtn != RTNC_SUCCESS) { /* Error handling. */}
```

6.2.1 `blk_sz_log2`

`blk_sz_log2` determines the size of the blocks reclaimed by the garbage collection. The default value is the underlying media's erase block size. Changing this value usually has little impact on overall performances. Values between 0 and 32 are accepted, but the block size must be greater or equal to the media's erase block size. The minimum number of blocks is 32.

6.2.2 `ext_sz_log2`

`ext_sz_log2` determines the default minimum extent size, that is the minimum extent size used if not overridden through `tsfs_file_extent_min_sz_set()`. The default value is 9 (512-byte extents). The extent size should usually be set to match the average update size. A value that is too low yields additional write amplification due to an increased metadata overhead. A value that is too high also yields additional write amplification due to overly large updates. Values between 6 (64-byte extents) and `blk_sz_log2 - 1` are accepted.

6.2.3 `bf_log2`

`bf_log2` determines the branching factor for indexing trees. The default value is 4 (16 children per node). Lower values tend to improve the average net write throughput on media with fine-grained read accesses such as NOR flash devices. This improvement comes at the expense of a higher RAM usage. Values between 1 and 4 are accepted.

Chapter

7 API Reference

Function

`tsfs_commit()`

<tsfs.h>

Commits all the updates performed on the given file system instance since the last commit, including file updates, snapshot creations and deletions.

In the event of an unexpected interruption (e.g. power loss) the file system is returned to the state it was in after the last successful call to `tsfs_commit()`.

Prototype

```
int  tsfs_commit  ( const char * p_fs_name );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_fs_name`	Name of the file system instance to be committed.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

`tsfs_create()`

<tsfs.h>

Creates a new file system instance.

Prototype

```
int  tsfs_create  ( const char *        p_fs_name,
                    const tsfs_cfg_t *  p_cfg );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_fs_name`	Name of the created file system instance.
`p_cfg`	TSFS configuration structure.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_destroy()

<tsfs.h>

Frees all the memory tied to the given file system instance.

Prototype

```
int  tsfs_destroy  ( const char *  p_fs_name );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_fs_name`	Name of the created file system instance.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_dir_close()

<tsfs_dir.h>

Closes the given directory. The handle becomes invalid after the directory is closed. Using a directory handle after closing it yields undefined behavior.

Prototype

```
int  tsfs_dir_close  ( tsfs_dir_hndl_t  hndl );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters	`hndl`	Handle to the directory to be closed.

Returned Errors
RTNC_SUCCESS
RTNC_FATAL

Function

tsfs_dir_create()

<tsfs_dir.h>

Creates a directory at the given path.

The given path must lead to a location within the working state.

If a file or directory already exists at this location, RTNC_ALREADY_EXIST is returned and the original file or directory is left untouched.

If the parent directory does not exist, RTNC_NOT_FOUND is returned. If the path is outside the working state, RTNC_INVALID_OP is returned.

Prototype

```
int tsfs_dir_create ( const char * p_path );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters	`p_path`	Path of the directory to be created.

Returned Errors
RTNC_SUCCESS
RTNC_NOT_FOUND
RTNC_INVALID_OP
RTNC_ALREADY_EXIST
RTNC_IO_ERR
RTNC_FATAL

Function

tsfs_dir_delete()

<tsfs_dir.h>

Deletes the directory located at the given path.

The path must lead to an existing directory of the working state. If the directory does not exist, RTNC_NOT_FOUND is returned. If the directory is not in the working state or the given path leads to a file, RTNC_INVALID_OP is returned. Also, if the directory is not empty, RTNC_INVALID_OP is returned.

An opened directory may safely be deleted. In this case, any further read access to this directory will return RTNC_NOT_FOUND.

Prototype

```
int  tsfs_dir_delete  ( const char *  p_path );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_path`	Path of the directory to be deleted.

Returned Errors

RTNC_SUCCESS
RTNC_NOT_FOUND
RTNC_INVALID_OP
RTNC_IO_ERR
RTNC_FATAL

Function

tsfs_dir_exists()

<tsfs_dir.h>

Verifies whether the directory located at the given path exists. The function returns `true` (through the `p_exist` parameter) if the directory exists and `false` otherwise (including when the given path leads to a file).

Prototype

```
int  tsfs_dir_exists  ( const char *  p_path,
                        bool *        p_exist );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_path`	Path to the directory which existence is to be tested.
`p_exist`	Whether the directory located at the given path exists.

Returned Errors

RTNC_SUCCESS
RTNC_IO_ERR
RTNC_FATAL

Function

tsfs_dir_open()

<tsfs_dir.h>

Opens the directory located at the given path. If the directory does not exist, `RTNC_NOT_FOUND` is returned. If the given path leads to a file, `RTNC_INVALID_OP` is returned.

Prototype

```
int  tsfs_dir_open  ( const char *        p_path,
                      tsfs_dir_hndl_t *   p_hndl );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_path`	Path to the directory to be opened.
`p_hndl`	Handle to the opened directory instance.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_INVALID_OP`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

`tsfs_dir_read()`

<tsfs_dir.h>

Reads the content of the given directory, one directory entry at a time. The name of the current directory entry is copied in the given name buffer. Calling `tsfs_dir_read()` after the last directory entry has been reached will return an empty string.

Prototype

```
int  tsfs_dir_read  ( tsfs_dir_hndl_t  hndl,
                      char *           p_name,
                      size_t           name_sz );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`hndl`	Handle to the directory to read from.
`p_name`	Buffer to receive the current entry name.
`name_sz`	Size of the given name buffer in bytes.

Returned Errors

`RTNC_SUCCESS`
`RTNC_IO_ERR`
`RTNC_OVERFLOW`
`RTNC_FATAL`

Function

`tsfs_drop()`

<tsfs.h>

Reverts the file system's state to that of the latest commit. All modifications performed since the latest commit are discarded, including file updates, snapshot creations and deletions.

Prototype

```
int  tsfs_drop  ( const char *  p_fs_name );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_fs_name`	Name of the file system instance.

Returned Errors

RTNC_SUCCESS
RTNC_NOT_FOUND
RTNC_IO_ERR
RTNC_FATAL

Function

`tsfs_file_append()`

<tsfs_file.h>

Writes the supplied buffer at the end of the given file.

The number of bytes written is returned through `p_append_sz`. The returned value may be smaller than `append_sz` if, and only if, the file system is full. In this case `RTNC_FULL` is returned and the value pointed to by `p_append_sz` indicates the number of bytes written before the file system becomes full.

If an error occurs, other than `RTNC_FULL`, the value pointed to by `p_append_sz` is unspecified. Otherwise, if the function completes successfully, the requested number of bytes is guaranteed to have been written. In this case, the value pointed to by `p_append_sz` is always equal to `append_sz`.

Prototype

```
int  tsfs_file_append  ( tsfs_file_hndl_t    hndl,
                         const void *        p_buf,
                         tsfs_file_size_t    append_sz,
                         tsfs_file_size_t *  p_append_sz );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`hndl`	Handle to the file to append data.
`p_buf`	Buffer to be written.
`append_sz`	Size of the buffer to be written in bytes.
`p_append_sz`	Size of the written data in bytes.

Returned Errors

RTNC_SUCCESS
RTNC_FULL
RTNC_IO_ERR
RTNC_FATAL

Function

`tsfs_file_close()`

<tsfs_file.h>

Closes the given file. The handle becomes invalid after the file is closed. Using a file handle after closing it yields undefined behaviour.

Prototype

```
int tsfs_file_close ( tsfs_file_hndl_t hndl );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`hndl`	Handle to the file to be closed.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

`tsfs_file_create()`

<tsfs_file.h>

Creates a file at the given path.

The given path must lead to a location within the working state.

If a file or directory already exists at the same location, `RTNC_ALREADY_EXIST` is returned and the original file or directory is left untouched.

If the parent directory does not exist, `RTNC_NOT_FOUND` is returned. If the path is outside the working state, `RTNC_INVALID_OP` is returned.

Prototype

```
int tsfs_file_create ( const char * p_path );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_path`	Path of the file to be created.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_INVALID_OP`
`RTNC_ALREADY_EXIST`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

`tsfs_file_delete()`

<tsfs_file.h>

Deletes the file located at the given path.

The path must lead to an existing file of the working state. If the file does not exist, `RTNC_NOT_FOUND` is returned. If the file is not in the working state or the given path leads to a directory, `RTNC_INVALID_OP` is returned.

A opened file may safely be deleted. In this case, any further read/write access to this file will return `RTNC_NOT_FOUND`.

Prototype

```
int  tsfs_file_delete  ( const char *  p_path );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_path`	Path to the file to be deleted.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_INVALID_OP`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

`tsfs_file_exists()`

<tsfs_file.h>

Verifies whether the file located at the given path exists. The function returns `true` (through the `p_exist` parameter) if the file exists and `false` otherwise (including when the given path leads to a directory).

Prototype

```
int  tsfs_file_exists  ( const char *  p_path,
                         bool *        p_exist );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_path`	Path of the file which existence is to be tested.
`p_exist`	Whether the file located at the given path exists.

Returned Errors

`RTNC_SUCCESS`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_file_extent_min_sz_set()

<tsfs_file.h>

Sets the minimum extent size for the given file. The file must be empty, otherwise `RTNC_INVALID_OP` is returned.

Prototype

```
void tsfs_file_extent_min_sz_set ( const char * p_path,
                                   uint8_t      min_sz_log2 );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_path`	File path.
`min_sz_log2`	Minimum extent size (base-2 logarithm).

Function

tsfs_file_mode_reset()

<tsfs_file.h>

Resets an opened file access mode.

Prototype

```
void tsfs_file_mode_reset ( tsfs_file_hndl_t hndl,
                            int              mode );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`hndl`	Handle to the file which mode is to be altered.
`mode`	Access mode to be reset (only `TSFS_FILE_MODE_RD_ONLY` is currently supported).

Function

tsfs_file_mode_set()

<tsfs_file.h>

Sets an opened file access mode.

Prototype

```
void tsfs_file_mode_set ( tsfs_file_hndl_t hndl,
                          int              mode );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters	`hndl`	Handle to the file which mode is to be altered.
	`mode`	Access mode to be set (only `TSFS_FILE_MODE_RD_ONLY` is currently supported).

Function

`tsfs_file_open()`

<tsfs_file.h>

Opens the file located at the given path. If the file does not exist `RTNC_NOT_FOUND` is returned. If the given path leads to a directory, `RTNC_INVALID_OP` is returned.

Prototype

```
int  tsfs_file_open  ( const char *          p_path,
                       tsfs_file_hndl_t *    p_hndl );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters	`p_path`	Path of the file to be opened.
	`p_hndl`	Opened file handle.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_INVALID_OP`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

`tsfs_file_read()`

<tsfs_file.h>

Reads the requested amount of bytes from the given file.

The number of bytes read is returned through `p_rd_sz`. The returned value may be smaller than `rd_sz` if the end of the file is reached. If an error occurs, the value pointed to by `p_rd_sz` is unspecified.

Prototype

```
int  tsfs_file_read  ( tsfs_file_hndl_t      hndl,
                       void *                p_buf,
                       tsfs_file_size_t      rd_sz,
                       tsfs_file_size_t *    p_rd_sz );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters	`hndl`	Handle to the file to read from.
	`p_buf`	Buffer to read into.
	`rd_sz`	Number of bytes to be read from the file.
	`p_rd_sz`	Actual number of bytes read from the file.

Returned Errors
`RTNC_SUCCESS`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

`tsfs_file_seek()`

<tsfs_file.h>

Sets the current read/write position of the given file to the specified position.

Prototype

```
int  tsfs_file_seek  ( tsfs_file_hndl_t        hndl,
                       tsfs_file_pos_offset_t  offset,
                       int                     whence );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters	`hndl`	Handle to the file to seek into.
	`offset`	New position relative to the supplied reference position (the `whence` parameter).
	`whence`	Reference position to which `offset` is to be added.

Returned Errors
`RTNC_SUCCESS`
`RTNC_FATAL`

Function

`tsfs_file_size_get()`

<tsfs_file.h>

Gets the size of the file located at the given path.

If the file does not exist, `RTNC_NOT_FOUND` is returned. If the given path leads to a directory, `RTNC_INVALID_OP` is returned.

Prototype

```
int  tsfs_file_size_get  ( const char *        p_path,
                           tsfs_file_size_t *  p_sz );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters	`p_path`	Path of the file to get the size of.
	`p_sz`	Size of the file in bytes.

Returned Errors
`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_INVALID_OP`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_file_truncate()

<tsfs_file.h>

Shrinks or extends the file located at the given path.

The path must lead to an existing file of the working state. If the file does not exist, `RTNC_NOT_FOUND` is returned. If the file is not in the working state or the given path leads to a directory, `RTNC_INVALID_OP` is returned.

A file may safely be truncated while it is opened.

Prototype

```
int  tsfs_file_truncate  ( const char *        p_path,
                           tsfs_file_size_t    new_sz );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters	`p_path`	Path to the file to be truncated.
	`new_sz`	Size of the file after truncation in bytes.

Returned Errors
`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_INVALID_OP`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_file_write()

<tsfs_file.h>

Writes the supplied buffer to the given file.

The number of bytes written is returned through `p_wr_sz`. The returned value may be smaller than `wr_sz` if, and only if, the file system is full. In this case `RTNC_FULL` is returned and the value pointed to by `p_wr_sz` indicates the number of bytes written before the file system becomes full.

Prototype

```
int  tsfs_file_write  ( tsfs_file_hndl_t    hndl,
                        const void *        p_buf,
                        tsfs_file_size_t    wr_sz,
                        tsfs_file_size_t *  p_wr_sz );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`hndl`	File handle.
`p_buf`	Buffer to be written.
`wr_sz`	Size of the buffer to be written in bytes.
`p_wr_sz`	Size of the written data in bytes.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FULL`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

`tsfs_format()`

<tsfs.h>

Formats the given file system instance.

Prototype

```
int  tsfs_format  ( const char *  p_fs_name,
                    const void *  p_params );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_fs_name`	Name of the file system instance to be formatted.
`p_params`	Optional format parameters (set to NULL for default).

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

`tsfs_media_get()`

<tsfs.h>

Gets the media used by the given file system instance.

Prototype
```
int tsfs_media_get ( const char *      p_fs_name,
                     bp_media_hndl_t * p_media_hndl );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_fs_name`	Name of the file system instance.
`p_media_hndl`	Retrieved media handle.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_FATAL`

Function

tsfs_mount()

<tsfs.h>

Mounts the file system residing on the given media. If the media has not been previously formatted using `tsfs_format()` or the on-disk format is invalid, `RTNC_INVALID_FMT` is returned.

Prototype
```
int tsfs_mount ( const char * p_fs_name );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_fs_name`	Name of the file system instance to be mounted.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_INVALID_FMT`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_revert()

<tsfs.h>

Returns the file system to the state it was in at the time of the given snapshot. If the given snapshot does not exist, `RTNC_NOT_FOUND` is returned.

Snapshots created after the revert snapshot are deleted. The revert operation is ended by an implicit commit.

Prototype
```
int tsfs_revert ( const char * p_path );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_path`	Path to the snapshot to revert to.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

`tsfs_sshot_create()`

<tsfs.h>

Takes a snapshot of the current file system state. If the snapshot already exists, `RTNC_ALREADY_EXIST` is returned.

Prototype

```
int  tsfs_sshot_create  ( const char *  p_fs_name,
                          const char *  p_sshot_name );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

`p_fs_name`	Name of the file system instance.
`p_sshot_name`	Name of the newly created snapshot.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_ALREADY_EXIST`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

`tsfs_sshot_delete()`

<tsfs.h>

Discards the given snapshot.

Prototype

```
int  tsfs_sshot_delete  ( const char *  p_fs_name,
                          const char *  p_sshot_name );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters	`p_fs_name`	Name of the file system instance.
	`p_sshot_name`	Name of the snapshot to be deleted.

Returned Errors
`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_sshot_exists()

<tsfs.h>

Verifies whether the snapshot located at the given path exists. The function returns `true` (through the `p_exist` parameter) if the file exists and `false` otherwise.

Prototype

```
int  tsfs_sshot_exists  ( const char *  p_path,
                          bool *        p_exist );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters	`p_path`	Path of the snapshot which existence is to be tested.
	`p_exist`	Whether the snapshot located at the given path exists.

Returned Errors
`RTNC_SUCCESS`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

tsfs_trace_data_get()

<tsfs.h>

Gets the trace data used by the given file system instance.

Prototype

```
int  tsfs_trace_data_get  ( const char *  p_fs_name,
                            void **       pp_tdata );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters	`p_fs_name`	Name of the file system instance.
	`pp_tdata`	Retrieved trace data.

Returned Errors	RTNC_SUCCESS RTNC_NOT_FOUND RTNC_FATAL

Function

tsfs_unmount()

<tsfs.h>

Unmounts the given file system instance.

Prototype

```
int tsfs_unmount ( const char * p_fs_name );
```

Attributes

Blocking	ISR-safe	Critical safe	Thread-safe
✓	✗	✗	✓

Parameters

p_fs_name	Name of the file system instance to be unmounted.

Returned Errors	RTNC_SUCCESS RTNC_NOT_FOUND RTNC_IO_ERR RTNC_FATAL

Data Type

tsfs_file_pos_offset_t

<tsfs_file.h>

TSFS file position offset. This type is used to represent file position offsets. It may contain positive or negative position offsets.

Data Type

tsfs_file_size_t

<tsfs_file.h>

TSFS file size. This type is used to represent file sizes and positions.

Data Type

tsfs_cfg_t

<tsfs.h>

File system configuration structure.

Members

media_hndl	bp_media_hndl_t	Media handle to be bound to the created file system instance.

`max_entry_cnt`	`size_t`	Maximum number of simultaneously opened files or directories.
`p_seg`	`void *`	Memory segment for allocating internal file system structures.
`seg_sz`	`size_t`	Size of the memory segment provided to the file system.
`p_tdata`	`void *`	Trace data.
`p_ext_cfg`	`const void *`	Optional extended configuration. Set to `NULL` for default.

Data Type

`tsfs_dir_hndl_t`

<tsfs_dir.h>

TSFS directory handle. A directory handle is obtained through `tsfs_dir_open()`.

Data Type

`tsfs_file_hndl_t`

<tsfs_file.h>

TSFS file handle. A file handle is obtained through `tsfs_file_open()`. The file handle is internally tied to a file descriptor that contains the current read/write position.

Many file handles can be obtained for the same file, each handle being tied to a different file descriptor and thus a different and independent file position.

Macro

`TSFS_FILE_MODE_RD_ONLY`

<tsfs_file.h>

File access mode flags. Access mode flags only affect opened file instances. They do not alter on-disk file attributes.Allows for write protection on a per-file basis.

Macro

`TSFS_MAX_INSTANCE_NAME_LEN`

<tsfs.h>

Maximum number of characters in an instance name excluding the terminating null character.

Macro

`TSFS_MAX_PATH_LEN`

<tsfs.h>

Maximum number of characters in a file path excluding the terminating null character.

Macro

RTNC_*

<util/rtnc.h>

Return codes.

`RTNC_SUCCESS`	Function completed successfully.
`RTNC_FATAL`	Fatal error occurred.
`RTNC_NO_RESOURCE`	Resource allocation failure.
`RTNC_IO_ERR`	Transfer or peripheral operation failed.
`RTNC_TIMEOUT`	Function timed out.
`RTNC_NOT_SUPPORTED`	API, feature or configuration is not supported.
`RTNC_NOT_FOUND`	Requested object not found.
`RTNC_ALREADY_EXIST`	Object already created or allocated.
`RTNC_ABORT`	Operation aborted by software.
`RTNC_INVALID_OP`	Invalid operation.
`RTNC_WANT_READ`	Read operation requested.
`RTNC_WANT_WRITE`	Write operation requested.
`RTNC_INVALID_FMT`	Invalid format.
`RTNC_INVALID_PATH`	Invalid path.
`RTNC_CORRUPT`	Data corrupted.
`RTNC_FULL`	Container full.
`RTNC_OVERFLOW`	Overflow

Macro

TSFS_FILE_SEEK_*

<tsfs_file.h>

File seek flags. Indicate where the file offset should be applied from.

`TSFS_FILE_SEEK_SET`	Seek from the beginning.
`TSFS_FILE_SEEK_CUR`	Seek from the current position.
`TSFS_FILE_SEEK_END`	Seek from the end of the file.

Chapter

8

Document Revision History

The revision history of the TSFS user manual and reference manuals can be found within the TSFS source package.