Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Design and implementation of a data storage abstraction layer for the Internet of Things

Lucas Andreas Jenß

Masterarbeit

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Lucas Andreas Jenß

Design and implementation of a data storage abstraction layer for the Internet of Things

Masterarbeit eingereicht im Rahmen der Masterprüfung

im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas Schmidt
Zweitgutachter: Prof. Dr. Franz Korf

Eingereicht am: 29.03.2016

Lucas Andreas Jenß

Title of the paper
Design and implementation of
a data storage abstraction layer
for the Internet of Things

Keywords
Internet of Things, IoT, File systems, Object storage, Flash memory

Abstract
The usage of persistent flash storage to overcome the memory limitations of Wireless Sensor Network nodes emerged in the early 2000s. But research on usable and reusable storage systems for embedded Operating Systems has stagnated, while the emergence of the Internet of Things (IoT) has yielded new storage requirements, which existing literature does not take into account. In addition, the usage of NAND flash is scarcely discussed, even though it is significantly cheaper and offers lower energy consumption than NOR flash. This thesis proposes a design for a flash-based storage system for constrained IoT nodes which supports NAND flash memory, and is evaluated based on a proof-of-concept implementation for the RIOT Operating System. The design is based on an extensive literature review which was used to gather a list of requirements that a storage system for the IoT should meet. The resulting system can be used to provide both low-level storage with very low resource requirements as well as a more sophisticated storage API with moderate resource requirements. The evaluation of the implementation suggests that it is significantly more resource efficient than most previous systems providing a similar feature set, and that it can be more easily adapted to different use cases, improving its reusability.

Lucas Andreas Jenß

Thema der Arbeit
Design und Implementation einer
Datenspeicherabstraktionsschicht
für das Internet der Dinge

Stichworte
Internet der Dinge, IoT, Dateisysteme, Flash Speicher

Kurzzusammenfassung
Persistenter Flash Speicher wird seit den frühen 2000ern eingesetzt, um die Speicherlimitierung von Wireless Sensor Network Knoten zu überwinden. Während durch das Aufkommen des Internets der Dinge (IoT) neue Datenspeicherungsanforderungen entstanden sind, ist die Forschung im Bereich der Datenspeichersysteme jedoch stagniert. Des Weiteren geht bestehende Literatur kaum auf die Nutzung von NAND Flash Speicher ein, welcher jedoch signifikant günstiger und energieeffizienter als NOR Flash ist. In dieser Arbeit wird deshalb ein flashbasiertes Datenspeichersystem für IoT Knoten mit eingeschränkten Resourcen vorgestellt, welches NAND Flash Speicher unterstützt. Das Design dieses Systems baut auf einer ausführlichen Auswertung der bestehenden Literatur auf, welche die Basis für eine Anforderungsanlyse von IoT Datenspeichersysteme bietet. Die Evaluation der Implementierung des Systems suggeriert, dass es erheblich ressourceneffizienter ist als die meisten vergleichbaren Systeme. Darüber hinaus ist das Sytstem besser an neue Anwendungsfälle anpassbar, wodurch die Wiederverwendbarkeit verbessert wird.

Contents

1 Introduction

Virtually all applications running on a computer need to store and retrieve data. In some cases, the memory available in the address space of the process itself is adequate. For other cases, where computations are performed over large data sets, data needs to be persistently stored, or data needs to be accessed by many users simultaneously, separate, persistent mass-storage approaches are required.

Flash memory, as a means of mass storage, has replaced Hard Disk Drives (HDDs) in many areas, since it exhibits higher durability due to the lack of moving parts, high heat and shock resistance as well as a low noise profile. Furthermore, it outperforms HDDs in terms of random Input/Output Operations per Second (IOPS) by about two orders of magnitude [16]. But Flash has another important trait: its energy footprint is considerably lower compared to HDDs [52]. These properties facilitate mass-storage for mobile devices, allowing millions of users to store their personal data on their phones. Enabling ubiquitous mass storage for the Internet of Things (IoT) might be the next breakthrough based on flash memory, due to its low cost and high utility value.

Wireless Sensor Networks (WSNs) consist of a large number of small, cheap, and resource-constrained nodes with ~10 KiB of RAM and ~100 KiB of ROM, which are physically spread across an area. They communicate wirelessly with one another, often self-organizing themselves into wireless ad hoc networks [55]. WSNs are commonly used for gathering data which is evaluated outside of the sensor network itself. The Internet of Things (IoT) is an evolution of WSNs, where sensors are connected to the Internet and can act autonomously based on their environment, possibly interacting with other nodes of the IoT or devices from the Internet, such as servers or smartphones. Use cases for both WSN and the IoT are plentiful [33]:

- In so-called “precision agriculture”, WSN can be used to control selective irrigation and fertilization as well as detection of weeds, pests and soil conditions [47].
- In civil and environmental engineering, IoT nodes can be employed to monitor the condition of objects created by humans as well as the environment, or even individuals themselves.
- In home automation, i.e. the “Smart Home”, a multitude of different systems can be remotely and automatically managed: lighting, heating, security systems such as locks or garage doors, water supply, air conditioning, home entertainment etc.

Many of these use cases can be implemented without mass storage support in the context of a WSN, since the main functionality of its nodes is to record and relay data. In the IoT, however, nodes are expected to operate autonomously. This implies that the evaluation of the sensed data and the resulting decision making process must take place on the node itself. By equipping IoT nodes with flash memory, it becomes feasible to implement these evaluation and decision making strategies without requiring the usage of less resource constrained, and thus more expensive, hardware.

One area where flash memory could substantially improve the capabilities of IoT applications is in Information Centric Networking (ICN) [8], which aims to save energy and radio resources as well as increase the availability of data through in-network caching and hop-wise replication. ICNs are inherently limited by the amount of memory available for caching, especially considering that not only the ICN layer, but also the actual application using ICN for communication must be fit into the RAM of the IoT nodes. The size of such an ICN cache could be drastically improved by providing a flash based storage solution.

Even though mass storage has great potential to extend the range of capabilities of IoT nodes, the research area has been stagnant. Embedded Operating System (OS) for WSN and the IoT do not currently offer storage systems with feature sets that match the requirements of WSN applications [46], nor do they examine the usability and extensibility of their approaches in detail. Furthermore, there are no publications discussing storage systems in context of the IoT.

This thesis presents the design of a storage system tailored to both WSN and IoT use cases, with special focus on the reusability and usability of the system. To this end, it provides an extensive survey of existing flash-based storage solutions employed in the WSN context in Chapter 3, laying the foundation of the requirements analysis for a storage system tailored to the IoT (Chapter 4). The design itself is presented in Chapter 5 as well as Chapter 6 and the implementation process is outlined in Chapter 7. Finally, the resulting design and the implementation are evaluated (Chapter 8) and the thesis is concluded in Chapter 9.

2 Background

2.1 The Internet of Things

Advances in wireless communication and electronics of the early 2000s allowed WSNs to emerge. Such networks are formed of many so-called "constrained devices" with limited resources in terms of CPU, memory and power. Many use cases for such networks exist, including monitoring of disaster areas, patient health, product quality or buildings. By adapting Internet protocols to operate on the devices that form WSN, they become part of the global Internet, forming the IoT.

2.2 Internet-connected embedded devices

Embedded devices which may be connected to the Internet have been grouped into three different classes depending on their hardware capabilities in RFC7228 [13] (see Figure 2.1). Class 0 devices are the most constrained and can be compared to sensor nodes of WSN. They are not expected to have enough resources to connect to the Internet themselves. Instead, they rely on less constrained proxies to achieve Internet connectivity. Class 1 devices are still very constrained, but can be expected to communicate with Internet nodes using protocols designed for constrained nodes (see Section 2.3). Class 2 devices are less constrained and can support most of the protocol stacks employed in the global Internet. Embedded devices beyond the capabilities of Class 2 do not pose significant limitations on the development of Internet-connected applications and are thus not further classified.

Name	data size (e.g., RAM)	code size (e.g, Flash)
Class 0, C0	$\ll$ 10 KiB	$\ll$ 100 KiB
Class 1, C1	10 KiB	100 KiB
Class 2, C2	50 KiB	250 KiB

Figure 2.1: Classes of constrained devices (KiB = 1024 bytes), as seen in RFC7228 [13].

2.3 Protocols in the Internet of Things

The rapid growth of the IoT has been facilitated not only by the decreasing cost of hardware in Class 1 devices, but also by the development of two significant low-power network access layer protocols: IEEE 802.15.4 [1] and Bluetooth Low-Energy [12]. On top of these, a suite of standard protocols has been created by the Internet Engineering Task Force (IETF) to provide a common language that enables IoT nodes to converse among themselves and become part of the global Internet. Part of this standard suite is 6LoWPAN [32], an adaption layer which translates IPv6 packets into a minimal format which fits the small payload sizes of low-power radio protocols such as IEEE 802.15.4. RPL [65] was designed specifically as a routing protocol for Low-power Lossy Networks (LLNs), and the HTTP-like Constrained Application Protocol (CoAP) [59] was developed to achieve standardized communication between Application Programming Interface (API) endpoints on the Internet and IoT nodes.

The existence of these protocols is one of the factors which make the IoT possible. Without them, Class 1 devices would not be able to exchange data over the Internet in a meaningful way, since they do not provide the necessary resources for a full TCP/IPv6 and HTTP(S) stack. A more complete survey of the technologies enabling the IoT can be found in [4].

2.4 Embedded operating systems for the IoT

Desktop and server OSs are designed for fundamentally different system requirements than what we encounter in WSN or IoT applications on constrained nodes. As a result, a number of OSs specifically designed for constrained nodes have emerged over the past decade, the prominent ones being TinyOS [53], Contiki [25] and, more recently, RIOT [7].

TinyOS is an OS initially developed at the University of Berkeley. It was originally intended to be used for nodes in WSNs (1 KiB memory), but has since transitioned to targeting a multitude of different use cases, such as personal area networks, smart buildings and smart meters (mostly Class 1 devices). For most of its existence, TinyOS has been an OS aimed at the research community and experts in the field of WSN, causing it to be less approachable for other people seeking to participate [39]. It uses a custom C dialect for event-driven programming called nesC. In the event-driven model, processes only run when an event is triggered, after which they manually return control to the kernel. The impact of such decisions has only recently become evident: with constrained nodes becoming affordable for personal or small-scale projects, potential users are alienated by the complexity of the system.

Contiki was also targeted at WSN in the beginning. Unlike TinyOS, its applications are written in C and it provides facilities for dynamically loading, unloading and updating modules at runtime, allowing the behavior of an application to be changed while deployed. Contiki's concurrency model uses preemptable threads as opposed to the event-based system employed by TinyOS. Contiki is currently advertising itself as "the Open Source OS for the IoT".

RIOT is a recent development in the space of IoT OSs, targeting Class 1 devices. Unlike TinyOS, which was primarily aimed at the research community, RIOT aims to simplify development of IoT applications. It is written in C and allows applications to be developed in C or C++. It uses a multithreaded programming model and a POSIX-like API, allowing many libraries to be ported from other POSIX OSs with little effort.

All the previously mentioned systems currently support the relevant IoT protocols which have been developed to connect embedded devices to the global Internet (see Section 2.3).

2.5 Flash Memory

Flash memory is a type of Electrically Erasable Programmable Read-Only Memory (EEPROM). In contrast to HDDs, where data is stored on one or more rotating magnetic disks, flash memory does not have any moving parts. It stores data in memory cells composed of floating-gate transistors which store an electrical charge applied to them. The fabrication of flash memory is among the fastest scaling semiconductor technologies, with 16 nm fabrication processes being common since 2014 and at least one semiconductor vendor preparing mass-production of 14 nm flash in 2016.

Figure 2.2 shows the components of NAND flash memory: the die, planes, blocks and pages. The die is the memory chip, which contains one or more planes. Each plane is outfitted with a number of blocks (typically 2,048–16,384). Each block, in turn, is composed of pages (typically 32–256), which are the smallest units than can be programmed (written to) [68]. Each page has a fixed number of bytes for data storage and a dedicated region for page metadata, for example 256 bytes of data + 8 bytes of metadata. The rapid evolution of flash semiconductors allows for larger memory capacities on smaller dies, but also also has implications for performance: for a fixed page size, write throughput decreases as the fabrication process shrinks. As a result, vendors have been continually increasing page sizes to compensate this effect [2]. The family of flash memory chips on the Mica platform (2004), had a page size of 264 bytes [6]. Current chips tend to have a page size of two or four kilobytes.

Flash memory has some properties which set it apart from traditional HDDs:

Bulk-erase An erase operation can only be performed on an entire block, erasing all resident pages. An erase operation sets all bits to `1`. This is a time consuming operation and should therefore be performed infrequently [68].

Write-once Programming a page consists of selectively setting bits of a previously erased page to `0`, such that the desired bit sequence is achieved. While multiple program operations setting bits to `0` can be issued (though limited in number, depending on the flash chip) the reverse operation requires the erasure of the entire resident block [34]. As a result, fine grained modifications of already written data is more complex than on traditional HDDs. For NOR

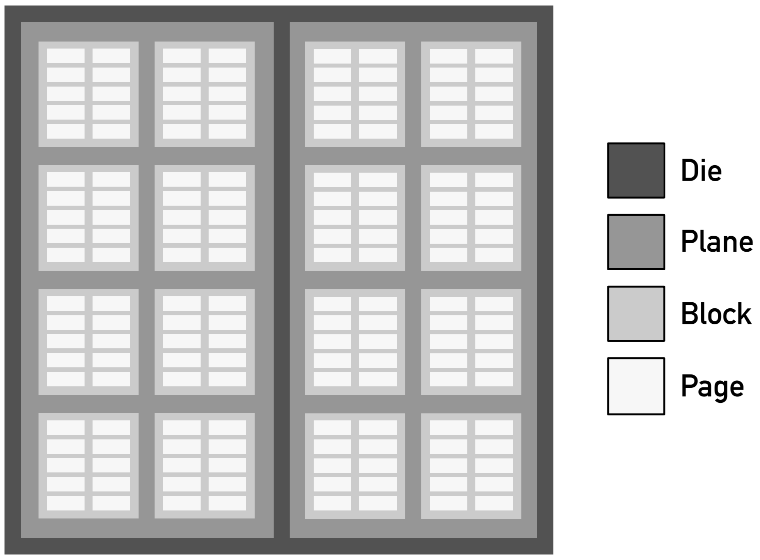


Figure 2.2: Components of a flash die. Note that a block commonly has many more than the 10 pages which were used for illustration (typically 32-256).

flash, the same limitations apply except that, because pages do not exist, each byte of a block can be independently programmed.

Block deterioration: Each block will fail after a number of program/erase cycles, depending on the type of flash memory being used.

Depending on the type of flash memory, the above properties are subjected to small variations. There are two primary types of flash memory, named according to the way their memory cells are built. One of them is NOR flash whose memory cells resemble NOR gates. NOR flash is the most reliable and expensive type of flash, but also the slowest. Instead of having to program entire pages at once, most NOR flash allows to program arbitrary bytes of a page (write-once still applies). NAND flash is again divided into two categories depending on how many bits are stored in each of the memory cells. If only a single bit can be stored it is called Single-Level Cell (SLC) flash, if more than one bit can be stored it is called Multi-level Cell (MLC) flash. NAND flash generally provides higher density storage at lower cost per bit. However, using NAND flash comes with a number of additional limitations [18]:

- Partial page programming is not possible (MLC) or only possible a limited number of times (SLC, up to 4 times)

	NOR	NAND	
		SLC	MLC
Size range	256 KB to 2 GB	128 MB to 512 GB	16 GB to 2 TB
Page size range	256 B to 2 KB	2 KB to 4 KB	4 KB+
Program/Erase cycles	100,000+	100,000	3,000 to 10,000
Price factor	40x	10x	1x
Read speed	> 100 MB/s	> 20 MB/s	> 15 MB/s
Write speed	< 1 MB/s	> 8 MB/s	> 2 MB/s
Block erase time	900 ms	2 ms	2 ms

Figure 2.3: Comparison between different types of flash memory.

- Bit errors are expected, so Error-Correcting Codes (ECCs) must be used. Flash memory vendors provide ECC requirements for each chip, meaning that they vary between types of flash, size of pages, etc.
  - Reading a page many times may cause read disturbance, causing bits of the same page or adjacent pages to flip. For MLC this happens around 100,000 reads and around 1,000,000 reads for SLC.
  - Writing a page may cause write disturbance, causing bits of the same page, which were not written, or bits of adjacent pages to flip.
- Durability of MLC flash erase blocks much lower (see Figure 2.3).
- Pages of an erase block must be programmed sequentially to reduce write disturbance. The effect is particularly severe on MLC flash, but must also be taken into account for SLC.

As a result, NAND flash is also less reliable and its usage implies significant additional implementation complexity [52, 68]. An overview of the different types of flash memory is given in Figure 2.3.

The properties of flash memory have implications for systems which aim to store data on them. Wear-levelling is a generic term for techniques to distribute program/erase operations evenly across all blocks, thus prolonging the lifetime of the device. Updating a page is problematic because it requires the prior erasure of the resident block. As a result, out-of-place updates are performed: pages (or entire files) to be modified are re-written with their modifications at a different location. Out-of-place updates come with a side effect, however. The pages where the updated data previously lived become obsolete and must be erased at some point in time. Garbage collection techniques are employed to determine blocks which have a high percentage of obsolete pages. Remaining live data is then copied to another block and the page is erased, making it available for writing again.

The implications of the continually growing page sizes of flash memory primarily impact constrained devices. Since it is not possible to write to a page many times without erasing the resident block first, data to be written must be buffered in RAM. Normally, this buffer would need to have at least the size of a single page. Assuming a page size of e.g. 2 KiB, this is already 20% of total RAM provided by a Class 1 device.

File Systems (FSs) not explicitly designed for usage with flash memory are difficult to adapt to its special properties. For desktop computing, vendors of NAND flash Solid-State Drives (SSDs) are employing complex translations layers which convert operations intended for HDDs to operations on flash memory. With wider availability of SSDs, operating system vendors are also including functionality to optimize their existing FSs for usage with flash memory.

2.6 Secure Digital (SD) cards

A special type of flash memory currently available is the SD card. It is notable because its storage is managed by a dedicated microcontroller. This controller omits some of the complexities of its underlying flash memory, such as ECC computation. Unfortunately, this controller has negative consequences for the usage of SD cards in low-power embedded scenarios. Flash memory is well-suited for embedded use because of its very low power requirements, which an additional controller undermines by drawing power for every operation. While the SD card controller can be powered down/up on demand to reduce its sleep current, this operation is very costly in terms of energy (equivalent to thousands of read operations [44]). As a result, SD cards should only be used for low-power embedded scenarios if no other option is available.

2.7 Efficiency of wireless communication compared to flash storage

Wireless communication is commonly the most expensive operation an IoT node can perform, and the energy footprint has not improved significantly in the last decade (see Figure 2.4). In comparison to the 11 year old Texas Instruments CC2420 transceiver, the best current generation counterpart (the Atmel AT86RF233) performs 38% better for receiving data and 21% better for sending data. Integrating the transceiver into a Microcontroller Unit (MCU) (e.g., the CC2630) achieves further improvements. In comparison, a recent survey puts the energy consumption when reading from a 8 GB SLC NAND flash chip at $0.001 \mu J$ per Byte, and at $0.025 \mu J$ when writing a Byte [49].

The estimated energy consumed when taking into account CPU operation as well as the cost of transferring data is about 5–7x higher than the cost of accessing a flash device, depending on the efficiency of the CPU [45]. However, this overhead is also incurred when sending data wirelessly. In summary, it can be said that storing a byte on flash is more than an order of magnitude more energy efficient than transmitting it wirelessly, and reading a byte from flash is more than two orders of magnitude more efficient.

	8 GB Flash	AT86RF233	CC2420	CC2520	C2630
MCU	✗	✗	✗	✗	✓
RX (mA)		11.8	18.8	18.5	6.1
TX (mA)		13.8	17.4	33.6	9.1
RX at 3V (μJ/byte)	0.001	1.13	1.80	1.78	0.59
TX at 3V (μJ/byte)	0.025	1.32	1.67	3.23	0.87

Figure 2.4: Overview of energy consumption of current IEEE 802.15.4 transceivers for the 2.4 GHz band compared to an 8 GB flash chip. Energy per byte calculated using maximum 802.15.4 compliant data rate of 250 kbps. RX (receive) and TX (transmit) current are based on manufacturer datasheets. The MCU row indicates whether or not the transceiver comes integrated into a microcontroller unit.

3 Literature Review

This section provides a broad overview of data storage paradigms which have surfaced in the context of WSNs and the IoT (Section 3.1), followed by an examination of the prevalent themes of embedded storage regarding physical storage structures (Section 3.2) and logical storage structures (Section 3.3).

3.1 Overview

In 2014, the International Telecommunication Union (ITU) declared WSNs one of the most rapidly developing information technologies, providing an elaborate survey of possible use cases ranging from agricultural, civil and environmental monitoring to smart home applications and emergency management [33]. As these WSN devices are connected to the global Internet, they become part of the IoT.

The engineering constraints for the implementation of these use cases have not changed when compared to early literature establishing the WSN research field [54, 3]. Sensor nodes are often required to be autonomous of a wired energy source, requiring careful management of available resources in order to prolong their lifetime. Their autonomy can only be useful, however, if nodes are also capable of reliable self-organization without human interaction throughout their lifetime [41]. These constraints must be met while at the same time keeping cost of production low, to make WSN economically viable products, and sizes small, to allow unintrusive deployment in many situations [33]. The latter can only be achieved by developing flexible hardware and software solutions not tied to a single use case, thus increasing possible production volume and, as a result, reducing development cost per unit.

One way to reduce energy consumption and broadening the applicability of WSN and IoT nodes is the utilization of mass storage made possible by advances in the area of flash memory [46]. Early approaches to data storage in WSN mostly focus on providing a convenient way to access named byte streams inspired by common FS. The nature of temporal sensor data, being a series of sequential records which does not change once written, was identified as a primary design requirement early on [29, 21, 70].

Matchbox [29, 30] is, to the author's best knowledge, the earliest storage system intended for WSN. It stored byte streams on flash memory using a log-structured approach, providing only appending writes and sequential reads. The authors of Efficient Log-Structured Flash File System (ELF) [21] developed a very similar system based on similar constraints, but acknowledge that, e.g. for storage of configuration files and binary images used in Over-the-air (OTA) updates, the ability to modify already written data is necessary. The Transactional Flash Filesystem (TFFS) [28] supports reading and writing files grouped into transactions, with the aim of preventing inconsistent state on the storage medium if the device should suddenly fail during any operation. TFFS supports modification of already written data. The Coffee file system [62] builds upon previous log-structured approaches. The authors goal was to eliminate the high in-RAM metadata overhead which comes with systems such as ELF by having a constant per-file memory footprint while at the same time reducing the complexity of the storage system in order to reduce general RAM and ROM footprint. A later extension to the Coffee FS enables support for encryption of stores data [9].

MicroHash [70] is the first approach designed based on the assumption that just providing a byte stream is not sufficient in a WSN environment. Instead an indexed sequence of timestamped records is employed, allowing more efficient access to written records. The authors of Capsule [45, 46] take this idea further by proposing a data storage system which can be adapted to many different use cases while still maintaining energy and memory efficiency, based on a variety storage objects adapted to different use cases. They present a system composed of a variety of storage objects matching the different storage requirements encountered in WSN applications. Along with the previously identified need for sequential storage of sensor records, the authors acknowledge network-related data such as packet buffers or routing tables as a potential source of data which can be stored on flash memory to reduce RAM requirements. In addition, data-rich sensing applications such as acoustic or seismic sensing need to perform operations on large data sets which do not fit into memory at once, but can be computed when data is loaded partially into RAM from flash memory. The storage objects the authors propose are streams, indices, queues and stacks as well as the compound object stream-index.

The authors of Squirrel [50] put special focus on the in-network processing capabilities of WSN data storage, providing a stream-oriented programming model for sensor applications, aiming to decouple data processing from data storage and handling the latter transparently for the application developer. When developing an application based on Squirrel, a directed graph of predefined stream operators is formed, each of which implements different storage policies. Depending on the operators as well as the size and volume of data entering the graph, data is either stored on flash memory or in RAM to improve energy efficiency. This data flow oriented approach is inspired by previous, similar applications in the WSN context without additional external storage [42, 10]. In contrast to the previous approaches, the authors of Squirrel do not mention capabilities for long-term storage of data on flash memory, but focus on processing large amounts of data before sending it. Unfortunately, the details regarding how and when data is stored on flash memory are entirely omitted from the paper presenting the approach, making further analysis in context of storage systems futile. As such, it is not further discussed in the remainder of this thesis.

Similar to Capsule [46], the authors of Antelope [63] reason that functionality for managing and querying data in WSN should be merged into a dedicated system to avoid reimplementation for every application. For this purpose they propose a Database Management System (DBMS) named Antelope. At its core, the database kernel coordinates database logic and query execution. The interaction with the database kernel happens, as is common in DBMS, through a dedicated query language, in this case called Antelope Query Language (AQL). AQL can be used to execute queries locally or remotely. The authors also stress the importance of being able to select different indexing algorithms depending on the use-case. As a result, Antelope's indexing subsystem allows for the selection of several provided algorithms as well as the addition of new ones. At first glance, systems such as Cougar [69] and TinyDB [41] seem comparable to Antelope in the sense that they also provide a database-like query interface to nodes of a WSN. The latter, however, are only used to program the way in which sensor data is sent towards the network sink, and not how the data is stored on the node itself.

One property which can be observed across the approaches mentioned so far is their lack of adaptability to current NAND flash memory designs, primarily large storage sizes and large page sizes. Since platform flexibility, that is the ease of adaption to different use cases, is one of the most important aspects of WSN [33], this thesis proposes an adaptable data storage abstraction which facilitates both long-term storage and in-network processing. An additional issue with previous approaches is that most of them are implemented for the TinyOS operating system, which has not seen a release in over three years. The system proposed by this thesis is implemented on the emerging embedded OS RIOT [7].

3.2 Physical storage structures

The design of flash memory entails a range of physical limitations which complicate its management (see Section 2.5). This section details how previous approaches have structured data in flash memory to overcome these issues, and highlights their individual shortcomings.

All of the previously mentioned approaches to data storage on constrained nodes employ some variation of log-structured data storage. Log-structured file systems were initially intended to improve performance on traditional HDDs, especially for write operations. Their main idea is that all information is written to disk sequentially, structured as a log of write operations [58]. This makes write operations extremely fast, since they are always sequential, but lowers read performance since files have to be reconstructed from the log. On a constrained node, write/read performance is not commonly a bottleneck due to the limited CPU and networking speeds of the involved hardware. However, the log structure was found to be a good fit for flash storage due to the implicit wear levelling that comes with a file system that has a sequential log structure [66, 43]: when all write operations just append to the existing log on the flash storage, the device is linearly filled up until no more space is available. Only then it becomes necessary to delete old data, so that new one can be written. And since existing log entries are not modified in a log-structured approach, updates to existing data are also appended to the log, i.e., data is automatically updated out-of-place.

Existing log-structured FSs such as JFFS [66] and YAFFS2 [5] have proven useful on flash-based SSDs in desktop computing. Unfortunately, they have been designed with the assumption that main memory is abundant [45], often using several megabytes of RAM per gigabyte of storage [31]. Given that WSN and IoT node are often Class 1 devices, such approaches are difficult to impossible to employ in constrained scenarios [45]. As a result, only Flash Translation Layers (FTLs) which have been specifically designed for constrained nodes are examined.

Matchbox [29], ELF [21] and Capsule [44, 46] were all developed for the Mica hardware platform with 512 KB of NOR flash memory and a page size of 264 bytes. Coffee [62] was evaluated on a Tmote Sky with a 1 MB NOR flash module (256 B pages). The authors of TFFS only mention evaluation on simulated flash chips.

Matchbox is a storage system for constrained nodes that only supports appending writes to existing data, i.e., already written data cannot be modified, only deleted. The physical storage layout of Matchbox is shown in Figure 3.1. Each page has a data portion (256 Byte) and a metadata portion (8 Byte). The metadata contains a pointer to the next page that belongs to the same logical storage structure, a Cyclic Redundancy Check (CRC) sequence, a page write counter for wear-levelling and a magic number identifying the type of page. In addition, Matchbox also stores the length of the data in the current page, in case it is only partially in use. An index of all files on the FS is stored in a “root metadata page” which is located somewhere on the flash storage. Since metadata is stored on a single root page, the number of file metadata which can be accommodated is limited by its size. On startup, Matchbox scans the entire flash memory for the newest root page, identified by its version number. In addition, free space is tracked using a bitmap. This approach was deemed reasonable for flash sizes as small as 512 KB, where scanning the entire flash is fast and a bitmap tracking free pages is only 256 bytes in size.

ELF stores its data in so-called “nodes”, each of which occupies one or more pages. The first page of each node contains information about the node, such as its type and its length in pages. At the end of each page, ELF stores metadata regarding the page. It contains a pointer to the next page belonging to the node, as well as a CRC and a write counter. Unlike Matchbox, it also stores a flag whether the page

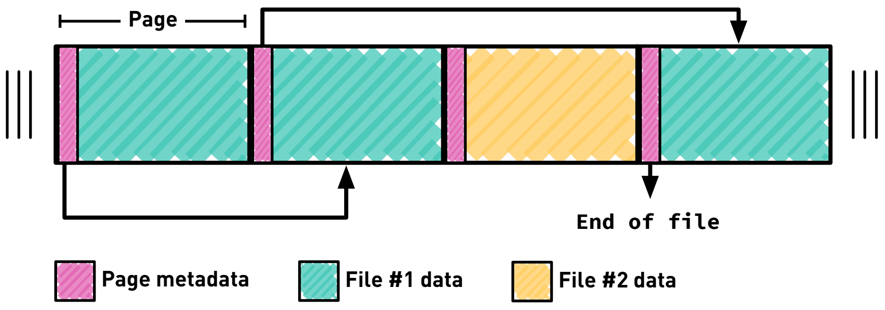


Figure 3.1: Page organization on flash memory for Matchbox.

is obsolete. Matchbox does not need such functionality, since deletion is solely reflected in Matchbox's "root metadata page". Information about the nodes which make up a file is stored in a separate log in RAM (see Figure 3.2). When a file is updated, for example, a new node will be created with the updated data. After the node is written, the information about the node is updated in RAM. ELF expects the target platform to have a separate EEPROM where it can periodically save the in-RAM state of the filesystem, so that it is not lost on failure/reboot. This avoids scanning the entirety of the flash storage for file system metadata, but requires a separate storage device on the platform. ELF also tracks free space as a bitmap. Given that storage has been scarce on constrained devices due to its cost in the past, it is unlikely that future constrained devices will come equipped with two forms of storage at the same time, especially considering that the type of EEPROM described for ELF must at least have the capabilities of the rather expensive NOR flash.

The Transactional Flash Filesystem (TFFS) [28] is a log-structured system that is organized in erase blocks instead of pages. This method of storage is only possible on NOR flash, since all other types of flash have a hard limit on how often pages can be partially programmed. Each erase block has two areas. The first are is the descriptor area, which starts at the beginning of the erase block and grows upwards. It contains information about what data is stored in the erase block and where, as well as the status of the data (live/obsolete). The second area is the data area and starts at the end of the erase block and grows downwards (see Figure 3.3). An erase block at the beginning of the flash device is reserved for the main file system log which stores file metadata and where data for a file is stored. A significant difference to all other systems discussed is that data in the main file system log is

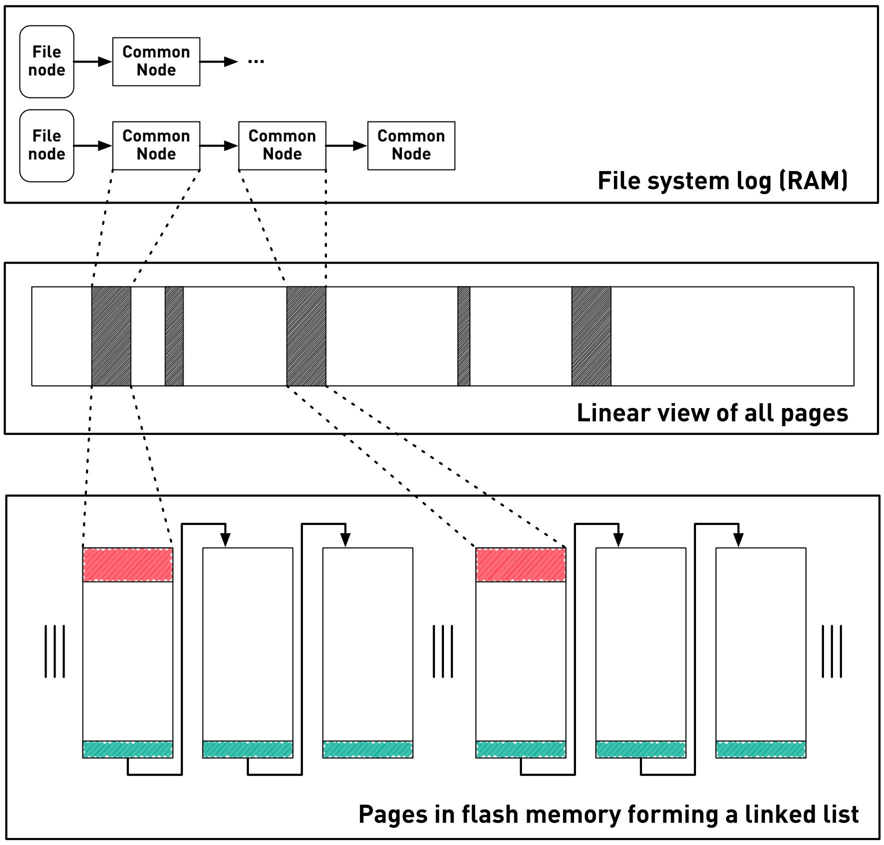


Figure 3.2: Page organization on flash memory for ELF [21]

addressed via logical erase units. These are mapped to physical erase blocks using the “logical-to-physical erase unit table”, which is stored in RAM. This mechanism is used so that, when performing garbage collection during which live data has to be copied to a different block, the metadata does not have to be updated. Instead, the entry in the logical-to-physical table is modified. The drawback of this approach is that said table grows with the amount of erase units, which makes RAM usage proportional to flash memory size.

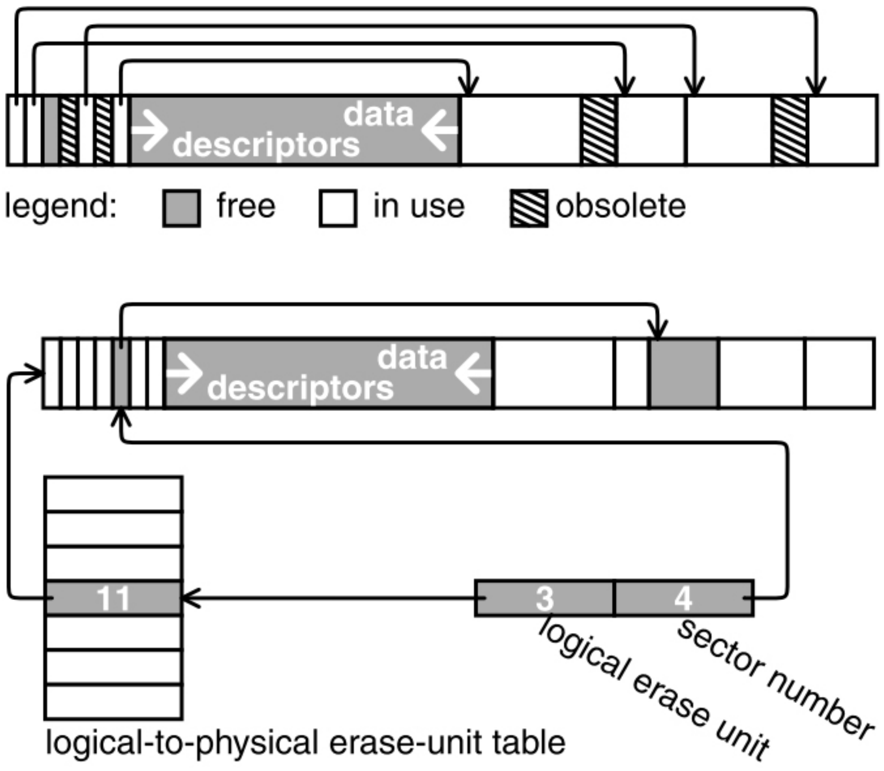


Figure 3.3: Physical layout of TFFS [28]

Coffee assumes that pages of a block can be written in random order and pre-allocates a default or given number of pages for each created file, to which its data is then written. If, at some point, the file size exceeds the pre-allocated space, It creates a new, larger file and copies the contents from the old one. Coffee stores information about files in the file header, which contains the size of the file, a number of file flags and information about the file’s “micro log”. The micro log data structure is used to log changes to files without having to update the original page, thus overcoming the write-once property of flash memory. Coffee's physical layout is shown in Figure 3.4. Matchbox and ELF store most of the metadata related to a file in RAM at all times. In order to reduce memory usage, the authors of Coffee decided to store that information on flash memory at the beginning of each file. Since Coffee, unlike Matchbox and ELF, has no central point where it stores file system metadata, it needs to be recreated from the state of flash memory. When Coffee is first requested to open or create a file, it will scan flash memory to find the file, or enough free space to allocate a new file, respectively. An in-RAM metadata cache is filled with information from prior search operations, though the paper [62] does not make clear if all files found during this operation are cached, or only the matching one. As for Matchbox, this approach works well for small flash devices, but is problematic for larger ones, as scanning the entirety of the device becomes very costly energy wise. In addition, Coffee does not track free space on flash memory. Instead it scans for free space whenever a file is created, allocating the file at the first suitable position (first-fit allocation). This approach has the advantage of reduced RAM requirements, but finding free space for a new file becomes dependent on the size of the storage device. In addition, as storage usage increases , so does the time required to find a fitting storage location.

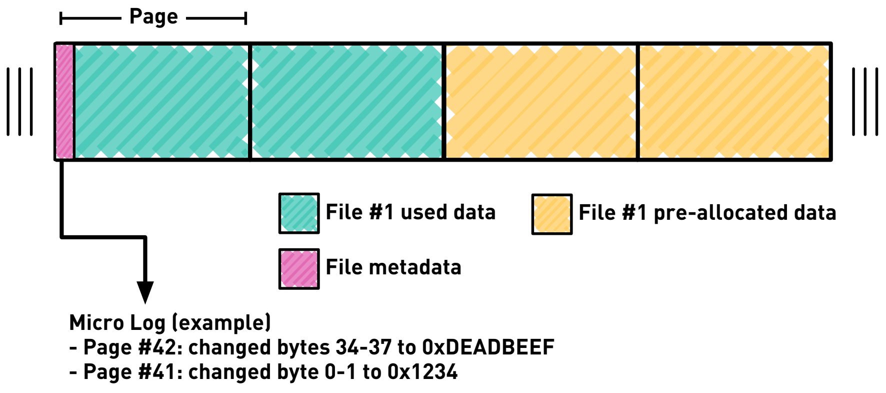

Page
File #1 used data
File #1 pre-allocated data
File metadata
Micro Log (example)
- Page #42: changed bytes 34-37 to 0xDEADBEEF
- Page #41: changed byte 0-1 to 0x1234

Figure 3.4: Page organization on flash memory for Coffee

MicroHash [70] is an early system which aims to provide storage and indexed retrieval of fixed-size records, without resorting to the already well-known files/directories abstraction. MicroHash is log-structured by page and uses a backward-pointing approach, meaning that each page includes a pointer to its predecessor, thus forming a reverse linked-list of pages (see Figure 3.5). This has the advantage of not requiring updates of already written pages to point to the next data page (compare Matchbox and ELF). However, it entails performance problems when iterating through a file from the beginning, since the successor of any given page can only be retrieved by iterating through the log from its end. When looking at Figure 3.5, to determine the successor of page one, it is necessary to read all pages beginning at page four until reaching page two, if this structure is not cached in RAM. A small part ("a few erase blocks") of the flash storage is separately managed as the "root pages". MicroHash regularly writes a serialized version of the in-RAM metadata to these blocks, such that its state can be restored if the node fails or is restarted. This is a similar approach to that of ELF, since it avoids scanning the entirety of the flash memory on startup, but it is an improvement over ELF for not requiring a separate EEPROM. Since erase blocks have a limited amount of erase cycles, the root directory must shrink with time, ultimately not being able to store any more data. The authors of MicroHash do not explain how block deterioration is handled, but go into great detail about their record index structure optimized for flash memory.

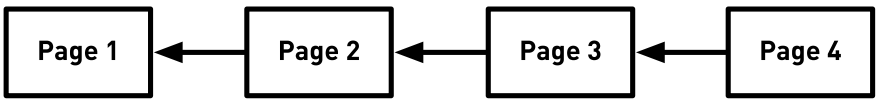


Figure 3.5: Page-based backward-pointing log structure as employed by MicroHash [70]

Capsule uses a similar approach to the that of MicroHash, but where MicroHash uses a backward-pointing log of pages, Capsule uses the same technique for its records. That is, each record includes a pointer to its predecessor, thus forming a reverse linked-list ending at the first record written for a data structure. Each page can contain multiple – possibly interleaved – records belonging to multiple data structures, such as a stack or a stream (see Figure 3.6). In contrast to the previous concepts, Capsule does not store mutable metadata for its records, thus eliminating the need to update a page after it has been written. The metadata for the data structures, which are composed of records, are stored in RAM. For most cases, this is simply a pointer to the most recently written record, from which the entirety of the data can be restored by following the linked list. To preserve the metadata in case the device loses power or is turned off, Capsule employs the same approach as MicroHash. It designates a number of blocks at the beginning of the flash medium for metadata, called the "root directory", to which metadata is written periodically. As with MicroHash, the authors do not explain how block deterioration of erase blocks is handled.

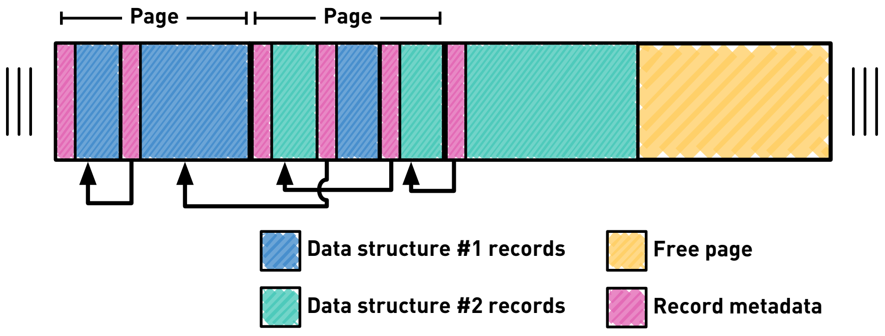


Figure 3.6: Page organization on flash memory for Capsule

Matchbox and Coffee share the approach that file/page metadata contains a pointer to the next page, i.e., their log is forward-pointing. Coffee uses this approach as well, but only for file updates. This method implies that part of a page (the pointer) has to be updated after it was initially written, because at that time the next (or micro log) page is not yet known. Updating a page is, however, not possible for MLC flash and only possible a limited number of times for SLC flash without causing severe write disturbance. The same problem applies to ELF's and Coffee's metadata flags. As a result, Matchbox, ELF and Coffee fail to support all MLC NAND flash and are only applicable in a limited fashion to SLC flash. The Capsule system, in contrast, supports both NOR and NAND flash due to its reverse linked-list approach. Capsule, however, only supports updating written data in a very limited fashion, which can be considered a step back from previous approaches.

	Matchbox	ELF	TFFS	MicroHash	Coffee	Capsule
Log structure	yes	yes	yes	yes	(yes)	yes
No forward pointing log	no	no	yes	yes	no	yes
Supported flash types	NOR	NOR	NOR	Unknown	NOR, MMC, (NAND)	NOR, MMC, NAND

Figure 3.7: Properties of the physical storage structure of the examined storage systems. Entries in parentheses indicate partial support.

3.3 Logical storage structures

The logical structure is the representation of data that the storage system exposes to the application developer, regardless of how it is stored on flash memory. For most of the discussed approaches, namely Matchbox, ELF, TFFS and Coffee, this representation is that of a named byte stream, better known as a file. In case of Matchbox, these files are append-only, meaning that data cannot be modified once written, whereas ELF and Coffee support file modification. ELF allows the organization of files into a hierarchy of directories, whereas Matchbox and Coffee are flat FS. The author of Coffee argues that, since the number of files on constrained devices is typical small, this is not a limitation in the context of WSN. The API provided by the three systems is similar. They all provide means of opening, closing and deleting a file, as well as listing all files (in a directory, in case of ELF). Matchbox only provides a sequential read operation, whereas ELF and Coffee allow read operations anywhere in a file.

Authors of systems such as MicroHash [70], Capsule and Squirrel [50] argue that files are not an ideal level of abstraction for WSN use cases. According to these authors, there are some basic requirements found in most WSN applications which a storage system for constrained nodes should take into account: support for in-network querying, processing, filtering and aggregation of sensor data in the form of a sequence of sensor readings (records) of fixed size [70, 46, 40]. Ignoring them causes application developers to fill in the gap, constantly reimplementing basic storage functionality specifically for each application, and thus limiting reusability and degrading maintainability due to the amount of code and complexity involved. In a study of three different WSN applications, more than 40% of the entire codebase, in one case 60%, were dedicated to data storage, excluding any functionality provided by the OS [50].

A major limitation is that MicroHash only supports append-only streams of fixed size values which cannot be deleted once written. If the storage device becomes full, the system will simply overwrite data at the beginning of the device, thus acting like a ring buffer. From the logical storage perspective, MicroHash provides indexed streams of records, which can be queried by time or value. The publication goes into detail about the structure of the employed index as well as search algorithms, but does not address the exposed API or how the system is used.

The authors of Capsule go further by incorporating other data structures into their storage abstraction layer. Their concept allows the creation of storage objects of different types (stream, queue, stack, file, index), each of which intended for a different storage requirement (see Figure 3.8). They argue that, for in-network archival of sensor data, immutable streams are an appropriate data structure. When combined with a supplementary index, such a stream also allows to query values with good performance. For applications which need to perform computations requiring a lot of memory (e.g., Fast Fourier Transform (FFT) or wavelet transforms), large arrays are commonly used as storage backend. Finally, queues and stacks are common data structures which can be employed to reduce the memory usage of many applications, including OS components such as packet buffers. An arbitrary number of Capsule objects can be created as well as deleted, making it adaptable to different application requirements. While Capsule has high ROM requirements (25 kB), it only needs about 1.6 kB of RAM.

Application	Data type	Storage object
Archival storage	Raw sensor data	Stream
Archival storage and querying	Raw sensor data	Stream-Index
Signal processing and aggregation	Temporary array	Index
Network routing	Packet buffer	Queue/Stack
Debugging logs	Time-series logs	Stream(-Index)
Calibration	Tables	File

Figure 3.8: Taxonomy of applications and storage objects for Capsule [45].

A different concept is followed by the authors of Antelope [63], which is built on top of Coffee. Antelope is a relational DBMS [17] which provides a Structured Query Language (SQL)-like interface to the flash storage on a constrained node. Data is stored in the form of relations, which can be created, filled and deleted at runtime. A relation consists of a name, a number of attributes (columns) and data tuples (rows). In addition, each attribute my be outfitted with an index for faster retrieval. Antelope provides a sophisticated data storage mechanism on constrained nodes, which effectively hides all of the underlying complexity of flash storage. However, using a domain-specific query language requires facilities for parsing and evaluating it, which comes at a cost as far as memory and runtime are concerned. Antelope alone uses between 3.4 and 3.7 kB of RAM and 17 kB of

ROM. Being built on the Coffee FS, its memory footprint must also be considered. In total, using Antelope requires between 3.7 and 4 kB of RAM, or 40% of a typical Class 1 device. Furthermore, Antelope is hindered by the same limitations as Coffee, discussed in the previous section.

3.4 Indexing algorithms

An index is an auxiliary data structure with the primary goal of limiting the set of records that have to be processed when filtering a range of values [63]. Creating and maintaining indexes is a well-researched problem in the field of databases, but the resource limitations of WSN applications [70] and the properties of NAND flash memory [36] result in different challenges.

The B^+-Tree is one of the more popular indexing data structures for file systems and DBMS, maintaining a relationship between indexed values and thus allowing querying of value ranges as well as successor or predecessor information. But when applied to embedded systems with flash memory it quickly becomes impractical, because updating a leaf node of the tree requires updates of all its parent nodes. Due to the write-once property of flash, this requires re-writing all data along the path from the updated node to the root node of the tree. Approaches like µ-Tree [36] and FlashDB [51] built upon B^+-Trees, adapting their concepts to the properties of flash memory. µ-Tree tries to improve the locality of the B^+-Tree updates by grouping all elements along the path from a node to the tree's root into the same physical flash page. The cost of updating a node – in terms of write operations – is thus reduced from $\mathcal{O}(n)$ to $\mathcal{O}(1)$. FlashDB [51] introduces a log-structured concept into the B^+-Tree. Updates are first written into a log, which is optimized for writing, and at some point converted to disk mode, which is optimized for reading. A more recent publication – TL-Tree [40] – presents an unbalanced tree structure for time-based indexing, instead of building upon B^+-Tree. At the same time, they present significant performance improvements to µ-Tree [36], to which TL-Tree is then compared.

In summary, indexing algorithms designed for non-flash storage, even when built for embedded devices, are not easily usable on flash memory because of the limitations it imposes on updating data on a page. The previously examined approaches show that this problem can be addressed by workarounds as in FlashDB and µ-Tree, or by designing a flash-adapted tree structure such as as TL-Tree. A detailed evaluation of indexing algorithms – beyond what general APIs are necessary to incorporate them into a storage system – is, however, out of scope for this thesis.

3.5 Application Programming Interface

While different definitions exist, the term API has come to mean "any well-defined interface that describes the service that one component, module, or application provides to others software elements" [23]. In this section, the APIs exposed by previous storage systems will be compared in terms of exposed functionality. Since MicroHash [70] does not provide information about the exposed API, it is omitted in this section. Note that while the systems developed for TinyOS all share the event-based nature of its nesC programming language, as exemplified in Listing 1, they are discussed here as if the had a more common C-like API.

The file based storage systems – Matchbox, ELF and Coffee – all expose a similar API, as can be seen in Figure 3.9. Their similarity stems from their similar storage structure and functionality. A concrete API example from the Coffee FS is shown in Listing 2. The only file based system which provides a storage abstraction beyond a simple byte stream is TFFS. In addition to reading and writing binary content, it allows storage of fixed size records. Furthermore, since it is transaction based, every operation must be given a transaction in which it should be carried out. All operations are accumulated and executed in bulk once the transaction is commited (see Listing 3).

Listing 1: Example of event-based API usage in TinyOS

```
// Call the command
call File.create("file name");

// Subscribe to the event which is invoked once the command has completed.
event void File.createDone(result_t res)
{
    call Console.string("Created file\n");
}
```

Listing 2: Example of reading and writing data using the Coffee FS

```
// Writing
fd_write = cfs_open(filename, CFS_WRITE);
cfs_write(fd_write, message, sizeof(message));
cfs_close(fd_write);

// Reading
fd_read = cfs_open(filename, CFS_READ);
cfs_read(fd_read, buf, sizeof(message));
cfs_close(fd_read);
```

Listing 3: Example of TFFS' API supporting transactions of multiple FS operations.

```
tid transaction_id = BeginTransaction();
AddRecord(file, buffer, length, transaction_id);
AddRecord(file, other_buffer, other_length, transaction_id);
CommitTransaction(transaction_id);
```

Since Capsule and Antelope implement a fundamentally different storage paradigm, it is natural that their APIs differ from the previously examined ones. The object-based storage of Capsule provides a different set of API calls for every provided object. This is comparable to the standard library of many programming languages, where each data structure provides its own interface. An example of Capsule's implementation of this concept is shown in Listing 4, in which L1–18 show the initialization and append operation of a file object and L20–33 show the initialization and push operation of a stack object. Interestingly, Capsule implements these in two different ways. While the file object is addressed by name (L4), the stack object is addressed by a numeric stack ID. Capsule's maximum number of stacks is defined at compile. Note the API was not taken from the publications on Capsule [44, 45, 46], but from the TinyOS 1.x source code [20].

Finally, Antelope exposes a SQL-like interface called the Antelope Query Language (AQL). Note that how the connection between the query language and the C code of the application is made is not made clear in the publication [63] and was thus taken from an Antelope usage tutorial [38]. Listing 5 shows the AQL being used for the purpose of creating a sensor data relation, inserting data, and subsequently performing a query. In the example, post processing is directly applied during the query, such that only the mean and maximum humidity values are returned.

	Matchbox	ELF	Coffee	TFFS
Open/Create	✓	✓	✓	✓
Read (sequential)	✓	✓	✓	✓
Read (random)	✗	✓	✓	✓
Append	✓	✓	✓	✓
Modify	✗	✓	✓	✓
Delete	✓	✓	✓	✓
Rename	✓	✓	✓	✓
Flush	✓	✗	✗	✗
Reserve space	✓	✗	✓	✗
Transactions	✗	✗	✗	✓
Records	✗	✗	✗	✓

Figure 3.9: API functionality provided by previous flash file systems for embedded devices.

3.6 Discussion

Summarizing the literature review presented in this section, the following systems were examined, categorized by their abstraction:

- File based: Matchbox, ELF, Coffee, TFFS
- Object based: MicroHash, Capsule
- Stream based: Squirrel
- DBMS based: Antelope

It is important to note that all these types of storage systems are designed for the same purpose, but with very different requirements. For example, Antelope provides an easy to use, SQL-like API that can be used to query data locally and remotely, and allows various forms of data processing without writing any code. All this functionality comes at a significant complexity and memory cost. On the other hand, systems like Coffee and ELF have a much smaller range of capabilities, but also require less resources to function. To choose an appropriate level of abstraction, it is necessary to compile a list of the requirements a storage system

Listing 4: Capsule API example using a File object and a Stack object

```
// File Object API
// Entry point
command result_t StdControl.init() {
    call File.create("my filename");
}

event void File.createDone(result_t res) {
    if (SUCCESS != call File.append(buff, LEN)) {
        call Console.string("File appending failed\n");
    }
}

event void File.appendDone(result_t res) {
    if (SUCCESS != res) {
        call Console.string("File appending failed\n");
    }
    call File.close();
}

// Stack Object API
// Entry point
command result_t StdControl.init() {
    call Stack.init(FALSE);
    if (SUCCESS != call Stack.push(stack_id, LEN, &buff)) {
         call Console.string("Error pushing to stack\n")
    }
}

event void Stack.pushDone(result_t res)) {
    if (res == FAIL) {
        [...]
    }
}
```

for WSN and IoT applications should fulfill. This task is carried out in the following section.

Listing 5: Antelope's AQL usage example for creating a relation, inserting data, and performing a query

```
db_init();

// Create relation
db_query(&handle, "CREATE RELATION samples;");
db_query(&handle, "CREATE ATTRIBUTE time DOMAIN INT IN samples;");
db_query(&handle, "CREATE ATTRIBUTE humidity DOMAIN INT IN samples;");

// Insert data
db_query(&handle, "INSERT (%u, %u) INTO samples;", 1, 2);

// Query data
db_query(&handle, "SELECT MEAN(humidity), MAX(humidity) FROM samples;");
db_print_tuple(&handle);
```

4 Requirements analysis

Sensors connected to IoT and WSN devices generate data which must be processed, filtered and possibly archived. But the main reason to collect sensor data is to interpret it, in order to extract useful information for its users [24]. In this section, typical use cases that require short or long term storage of data which exceeds the RAM capacity of sensor nodes will be examined. This is done with the goal of compiling a list of requirements that components of an adaptable data storage abstraction should meet. Based on this, the case will be made for an object-based storage approach, showing that it best fits the requirements imposed by WSN and IoT use cases.

4.1 Use cases

#1 – Facilitating caching in Information Centric Networks

The idea behind ICN is that communication is no longer host-centric, i.e., data is no longer retrieved from a known host. Instead, ICN-based techniques take advantage of in-network caching and hop-by-hop replication of named data in order to spread it throughout the network. IoT applications can benefit from ICN, since it can reduce wireless communication and increase availability of data in case of node failures [8, 71]. One major limitation is the small amount of data that can be stored (cached) in the memory of a constrained node. Consequently, employing an additional, flash-based storage back-end could vastly enhance the storage capacities of IoT ICN applications.

#2 – Reducing operating system and application memory footprint

It is common for operating systems outside the contrained device domain to virtually extend the amount of available RAM by “swapping out” data to persistent storage. Unfortunately, doing so automatically on constrained devices is difficult due to the management overhead as well as the missing Memory Management Unit (MMU) on the nodes. It is possible, however, to manually decide on data structures which must not necessarily live in RAM, and implementing them such that they are stored in flash memory instead. This may be done by the application or by the operating system. For example, a kernel developer could decide to implement a routing protocol such that the routing table can optionally be stored on flash to preserve RAM. Reducing the amount of memory the OS and applications occupy can broaden the range of applications that can be implemented or allow additional features to be added. Storing OS network layer components (e.g. packet buffers and routing tables) on external flash memory instead of RAM can alleviate memory usage of such components and may even “lead to increased performance” [62].

#3 – Improving management mechanisms for IoT nodes

Observing the behavior of a WSN from edges of the network only provides limited insight into what is going on inside it. While it is possible to include additional information in the packets delivered to the observing entity – as is done in network management protocols such as SNMP [15] and NETCONF [26] – this comes with an energy overhead due to high energy per byte cost when transmitted wirelessly. As such, it is preferable to store such information locally and evaluate the information in bulk when retrieved from network nodes. For long-term deployments for example, gathered network performance data could be used to fine-tune network topology for future deployments [63]. Such data could also be employed to improve simulation systems by comparing actual performance data with simulated data. For failing nodes, error logs could help reproduce issues that occurred on deployed nodes.

#4 – Improving capabilities of intermittently connected entities

A WSN may be composed of roaming entities or entities deployed in physically remote areas where connectivity is not constantly available, that is the number of neighbors of each network node is typically less than one. Connectivity becomes available sporadically, or constantly but with very low connection capacity [51]. Nodes with sufficient local storage can store data from their sensors until connectivity is available and then transmit the entirety of the data or some form of preprocessed summary [19]. In Delay Tolerant Netork (DTN) terminology, this type of packet delivery is called “store-carry-and-forward” [60]. Alternatively, a node may not be wirelessly connected at all, requiring that its data is physically retrieved by replacing the storage medium periodically. Such deployments would benefit from local storage since it removes the need for any wireless communication, ultimately increasing battery life.

#5 – Processing data which does not fit into main memory

Processing data on a constrained node can be desirable if the entirety of the data is not of interest. For example, when monitoring a forest environment through a WSN, biologists are commonly interested in the long-term behavior [70]. Only transmitting summaries of captured data would result in less wireless communication, thus extending the node lifetime. Post-processing might even be necessary if roaming entities (see above) are expected to have connectivity only for short instances of time, which would not allow to transmit the sensed data in its entirety. For other applications, high frequency collection of sensor data is necessary. This includes cases such as vibration measurement to evaluate structural integrity of buildings, but also any form of audio capture and analysis.

Processing such data on the node is cumbersome for the application developer, since it is beyond the size that can be accommodated in RAM. It requires manually managing items that are currently held in RAM and identify which items need to be loaded/unloaded. Listing 6 shows a simplified version of such an application. Providing this functionality on the storage system level allows application developers to process large sets of data without implementation overhead for data management.

Listing 6: Simplified example of processing a number of items which are stored on an external storage medium and do not fit into memory in their entirety. The developer would also have to implement all used functions regarding counting, loading/unloading and accessing items.

```c
int num_items = get_number_of_items();
for(int i=0; i<num_items; i++) {
    if(!item_in_memory(i)) {
        item_load(i);
    }

    int item = item_get(i);
    // Perform calculation with item

    item_unload(i);
}
```

#6 – Enhancing support for over-the-air programming

WSN and IoT devices are often expected to operate autonomously for long periods of time, and their requirements may change over the course of their lifetime depending on their deployment scenario. To account for such changes, it should be possible to reprogram these constrained nodes wirelessly. A wide variety of protocols exist for this task, most of which store the received program code on external storage prior to updating the actual application section of the node's memory [64] (for a recent survey, see [14]). A simpler form is OTA reconfiguration, where operational parameters stored on external memory are changed instead of reprogramming the application code.

4.2 Functional Requirements

What functionality the storage system must provide, i.e., the system's functional requirements, are listed below.

- The system should provide storage abstractions for common WSN and IoT use cases, as identified in the previous section. Most authors of previous storage systems agree that an – optionally indexed – stream of sensor data is a very common requirement. In addition – based on the authors of Capsule [46] –

other common storage abstractions are arrays (e.g., for use case #5), stacks and queues (e.g., for use case #2).

- The system must support different types of indexes, due to the diverse nature of use cases. For example, information in ICNs is named and possible hierarchical [67], suggesting trie-based indexing. For temporal data, such as that provided by sensors, a B^+-Tree or TL-Tree [40] is preferable.
- The system must be built to support the properties of flash memory, such that it is applicable to as many varieties as possible.
  - It must support NOR and SLC NAND flash with page sizes up to 1024 bytes. It should support SLC NAND flash with page sizes greater than 1024 and may support MLC NAND.
  - It must provide wear levelling facilities, such that lifetime of flash memory is maximized.
  - It must support identification and bit errors when writing to and reading from flash memory.
  - It must implement garbage collection to free obsolete blocks
  - It must support a wide variety of flash memory sizes. The support must range from small 512 kB NOR chips – for short-term storage of configuration data – to large 1 GB+ SLC NAND chips for long-term storage of historical data or in-network caching application in ICNs.
- The storage system must be able to provide a dedicated area on the flash memory reserved for OTA updates. It must be possible to write data to this area without storage system headers added to them, as these would corrupt the application binary (use case #6).
- The system must provide the means for locally processing data which does not fit into the constrained node's RAM.
- The system must reduce the complexity of locally processing data which does not fit into the constrained node's RAM (use case #5). This must be achieved such that the application developer is not confronted with the complexity shown in Listing 6 every time they wish to iterate through an object.

- The system should support the secure (i.e., encrypted) storage of confidential data.

4.3 Non-functional requirements

The identified, non-functional requirements define how the storage system should behave, and are examined below.

- Reliability
  - The system must be able to handle sudden energy loss, such that data written to the storage medium is not corrupted or lost.
  - The system must only fail to write data if the storage medium is full and it is not possible to create free space by reclaiming blocks with obsolete data.
  - The system must be able to operate on a multi-threaded OS, that is, it must be thread-safe. This implies that performing operations concurrently from different threads must not result in data corruption or other non-recoverable errors.
  - There is **no need** for the system to support parallel reading/writing. Threads trying to perform such operations while another one is in progress may block or the operations may return an error.
- Performance
  - The system should occupy no more than 1–2 KB of RAM and 10–20 KB of ROM, so that it can extend existing applications running on Class 1 devices.
  - The system must provide sufficient write speed such that audio and other data sensed at high frequencies can be buffered to the storage medium prior to post-processing. The write performance of prior systems, ranging between 10 KB/s [21] and 40 KB/s [62], should serve as a reference for evaluating the performance of the developed system.
- Usability

  - The system must come with clearly defined interfaces between the different layers of abstractions, such that a lower layer may always be used in isolation.
  - The API must have consistent naming of functions, as well as consistent naming and ordering of parameters.
  - All non-internal functions of the system must be documented.
  - The system must come with examples explaining how common functionality is used. These examples must include initialization of the system and storing/accessing/erasing data.

- Extensibility
  - The storage system's extensibility encompasses its ability to be adapted to circumstances which were not originally thought of, and the ability to improve or replace components without having to change the overall structure of the system. Most importantly, the system must be extendable with new ECC algorithms to support specific flash memory requirements, and also extendable with index data structures to account for different application scenarios.

- Reusability
  - The system must not be built specifically for a single hardware platform, and, if possible, it should be built such that it is portable to other OS.

4.4 Discussion

One of the most important research priorities in WSN and the IoT is the reduction of development and deployment cost [33]. Since hardware cost is incurred per node, lowering resource usage of constrained applications is an important factor, given that it allows cheaper devices to be used. But even though a large portion of deployment cost depends on the hardware, the software development cost must not be underestimated. As a result, reusability and maintainability are important, non-functional requirements of the storage system to be developed.

File-based storage, when examined in isolation, is the least expensive in terms of resources and implementation complexity. When put in context of an application, however, it becomes apparent that using file-based storage simply shifts complexity and resources cost from the operating system to the application, where abstractions such as indices must then be implemented. The other extreme for storage on constrained nodes is a fully fledged DBMS such as Antelope [63]. Such a system makes the complexities involved with data storage completely transparent to the developer and provides common post-processing mechanisms. Common programming errors (e.g., null pointer, off by one, overflow, etc.) when querying and inserting data are avoided due to the SQL-like interface. This usability comes at the cost of significant resource usage and complexity. In addition, it hinders extensibility of the system, since new features must be implemented at least in the query language parser and the query evaluation mechanism. If concurrent storage access was a requirement, a DBMS would make its implementation easier due to the possibility to schedule incoming queries, but concurrent access was not deemed useful on constrained devices.

In terms of complexity and resource cost, an object-based storage system such as Capsule [44] can be found somewhere between file systems and DBMS. Commonly needed features and data structures can be provided, so that an application developer must not implement them. This greatly increases reusability and – since storage access is less abstract compared to a DBMS – the cost in terms of implementation complexity and resource usage is not as severe.

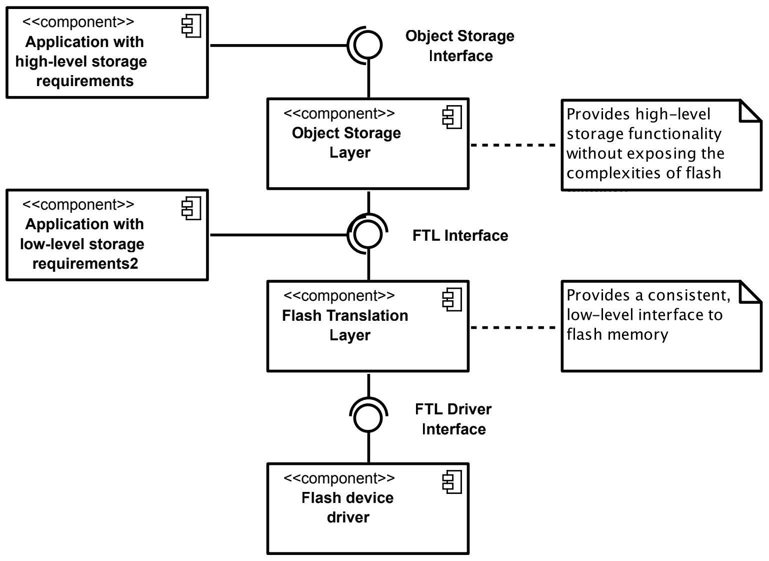
The design of the the storage system based on the requirements gathered in this section is detailed in the following section.

5 Design of the Flash Translation Layer

In the upcoming two sections, the design of the storage system for the IoT is presented. Based on the literature review performed in Chapter 3 and the requirements analysis of Chapter 4, the file-based approach was deemed unreasonable to provide a reusable storage system for the IoT, given that storage abstractions must be implemented in the application itself, which are then hard to reuse. On the other hand, a DBMS is powerful in terms of functionality, but its usage has a range of disadvantageous implications, primarily the necessity to parse and evaluate a query language. This makes a DBMS resource intensive. As a result, the object-based storage paradigm – as implemented by Capsule [44] – was chosen. An object-based storage system provides the ability to implement common storage abstractions in a resuable way at little resource cost, making it a good fit for use on constrained devices.

The system is structured as two main components which communicate through well defined interfaces (see Figure 5.1). The component which handles the direct interaction with the flash storage device is the Flash Translation Layer (FTL), presented in Chapter 5. On the basis of the FTL, the Object Storage Layer (OSL) provides a set of reusable storage abstraction – or objects – to be used by the application developer, without exposing the complexities of the employed flash storage. The Object Storage Layer (OSL) is presented in Chapter 6.

The design of the FTL presented in this chapter is primarily shaped by the constraints imposed by flash memory (see Section 2.5). Its main goal is to provide a convenient, low-level interface for data storage on flash memory. The FTL is designed such that it can be used without any further storage-related components except for the flash memory driver. The FTL also serves as the foundation for the higher-level storage system to be presented in Chapter 6.



<<component>> Application with high-level storage requirements
Object Storage Interface
<<component>> Object Storage Layer
Provides high-level storage functionality without exposing the complexities of flash
<<component>> Application with low-level storage requirements2
FTL Interface
<<component>> Flash Translation Layer
Provides a consistent, low-level interface to flash memory
FTL Driver Interface
<<component>> Flash device driver

Figure 5.1: Overview of the storage system components as well as the types of applications they are intended for

This chapter is structured as follows. The first two sections describe the basic physical storage layout of the FTL, which divides pages into subpages (Section 5.1) and the flash device into partitions (Section 5.2). How data is read and written is explained in Section 5.3, followed by a short note on wear levelling (Section 5.4) and an explanation of the free space tracking and garbage collection mechanics provided be the FTL (Section 5.5). How the state of the storage system is persisted to flash memory is detailed in Section 5.6. The chapter is concluded by presenting an overview of the FTLs API (Section 5.7) and a discussion of the chapter in Section 5.8.

5.1 Subpages

One of the most important requirements for the storage system is the support for common types of flash memory, so that it can be used without modification in a wide range of scenarios. This implies that only the most restrictive properties of the various types of flash storage may be depended upon, which are exhibited by NAND flash. As detailed in Section 2.5, NAND flash memory only supports a limited number of write operations per page until the parent block has to be erased. MLC

NAND exhibits the extreme case, where only a single write operation is possible. As a result, writing small fractions of a page is not easily possible.

To address this problem, data must be collected in a page-sized buffer in RAM. Once this buffer is full, the collected data can then be written to the flash page all at once. Given that current NAND flash storage commonly has page sizes of 2 KiB or more, such buffers are a problem for constrained nodes, since these have very little RAM. In order to reduce the required buffer size, the FTL divides each flash memory page into *subpages* of $p\,/\,k$ Bytes, where p is the page size and k is the maximum number of write operations which can be performed per page before it has to be erased [46]. The resulting storage buffers must only have the size of a single subpage, and each subpage must be fully written in one write operation. For example, a device with $p = 2048$ Bytes and $k = 4$ would result in a subpage size of 512 Bytes (see Figure 5.2a), reducing RAM usage by 75%.

The FTL also allows the application developer to sacrifice storage space on the flash device in order to reduce buffer size. This is done by setting subpage size to $p\,/\,x$ Bytes, where $x > k$, and instructing the FTL to only write to the first k subpages. This makes $(x-k)\,/\,x$ percent of memory per page unreachable by the storage system. For example, if x was set to 8 in the example above, then each subpage would have a size of 256 Bytes. Since the flash device only supports four writes per page, however, only the first 4 subpages can be written to. As a result, $(8-4)\,/\,8 = 50\%$ of memory would become unreachable, as shown in Figure 5.2b.

In summary, division of pages into subpages is performed to significantly reduce required buffer sizes. Since subpage size is proportional to the maximum number of write operations per page (k), NOR flash allows for the smallest sizes, followed by SLC NAND. Finally, using MLC NAND comes with the highest subpage sizes since its $k = 1$.

5.2 Partitioning

Partitions are a well-known concept in file system design, allowing the definition of – usually named – regions designated to hold a certain kind of data. Such regions can be useful for a number of reasons, such as having a dedicated area for OTA update storage or storing metadata (see Section 5.6). While previously developed

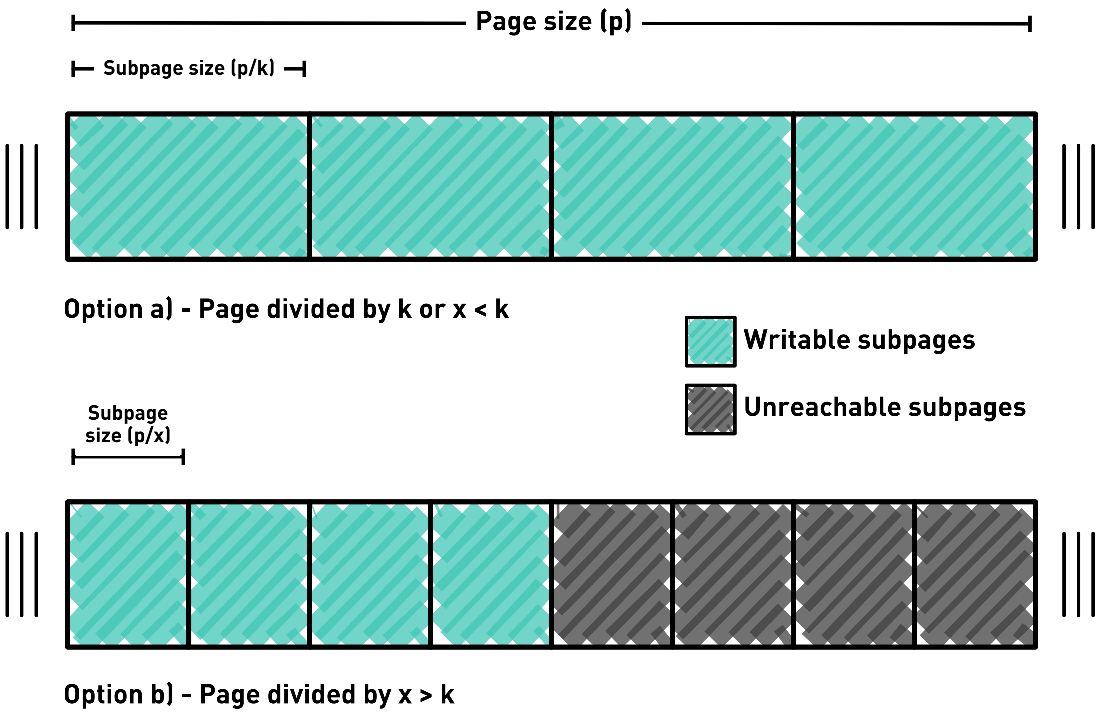


Figure 5.2: Subdivision of pages into subpages with the aim of reducing buffer size. Option b) shows the special case where the amount of subpages is greater than the maximum number of write operations per page, resulting in unreachable storage space.

storage systems sometimes featured dedicated regions for specific kinds of data, it was never possible for an application developer to create custom partitions.

Partitions in the designed FTL provide this flexibility by making the amount of partitions, their size and their offset configurable at compile time. A partition is a named area on the flash device – identified by its start offset and its size – which does not overlap with any other partition. Whenever data should be written using the FTL, the developer must actively choose the target partition.

To simplify partition management, the offset and size is specified in erase blocks. If partition boundaries were specified in pages instead, the situation depicted in Figure 5.3 could occur. In that case, erasing the highlighted block is difficult, since the deletion routine must be aware of both the end of partition #1 and the beginning of partition #2. Partitions must also be larger than one block, since erasing a block is only possible if its live data can be copied to another block first.

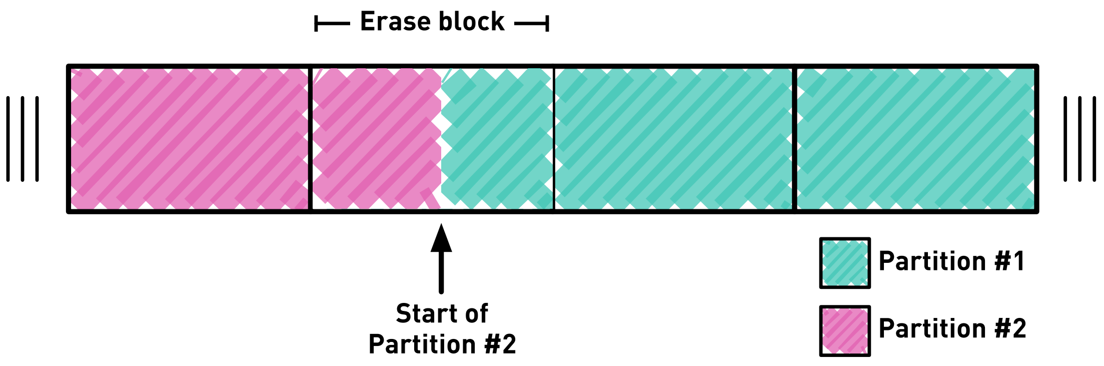


Figure 5.3: Partition boundaries inside an erase block are problematic, because it makes the erasure of the highlighted block dependant on data in both partitions.

Apart from simplifying the issue of metadata storage (see Section 5.6) and providing a way to store OTA update data, configurable named partitions can be seen as very simple fixed-size files. This has the advantage of allowing the FTL to be used as a (very limited) file system, where the available files (partitions) and their maximum size are defined at compile time.

5.3 Reading and writing data

Since the storage system should be compatible with all common flash types, the FTL must ensure that all subpages of an erase block are written sequentially from the beginning to avoid write disturbances and resulting bit errors. To achieve this, the FTL keeps track of the next subpage that should be written for every partition. After the last subpage of a partition has been written, the FTL will simply wrap around and start at the beginning, if free space is available there (see Section 5.5).

Reading and writing data occurs on a per-subpage basis and is performed either in raw or managed mode. When writing in raw mode, the passed buffer will be written to the storage device as-is, without any metadata. This mode is useful, for example, when writing firmware images for OTA updates which need to be flashed to ROM exactly as received. When writing data in managed mode, each subpage is preceded by a metadata header (shown in Listing 7). The metadata header stores the amount of data on that subpage as well as an optional ECC. Disabling the ECC is desirable, for example, when the used flash storage has an integrated hardware ECC. Reading data in raw mode simply returns the contents of the given subpage, whereas managed mode is aware of the metadata. This allows the storage system to verify the integrity and possibly repair a subpage based on its ECC.

Listing 7: The two byte subpage header which is prepended to each page written by the FTL.

```c
typedef struct __attribute__((__packed__)) {
    unsigned int data_length:15;
    unsigned int ecc_enabled:1;
} subpageheader_s;
```

Listing 7 shows that the ECC is not part of the FTL's subpage header structure. This is because ECC requirements vary between different types of flash memory, and different ECC algorithms may also result in different ECC sizes. To provide support for a wide range of devices, the FTL instead allows per-device configuration of a fixed ECC size (e), as well as a custom ECC generation and checking function, such that developers can provide their own implementations if necessary.

If the `ecc_enabled` flag is set, the next e bytes following the subpage header will be assumed to contain the ECC.

5.4 Wear levelling

As discussed in the previous section, the FTL is built so that it writes to all pages of a partition sequentially and wraps around at the end. This provides automatic, per-partition wear levelling, since all pages must have been written once before being written a second time. Even with perfect wear levelling, bad blocks are expected to appear throughout the lifetime of a flash storage device. When bad blocks are encountered, the storage system will simply skip them, finding the first subsequent good block that it can write to.

5.5 Free space management

Since small amounts of data cannot be easily deleted on flash memory, a different technique has to be employed to reclaim blocks which have been written to, but no longer contain any live data. To be able to reason about when to reclaim storage, and which areas to reclaim, it is important to keep track of areas of flash storage containing data and areas that can be reclaimed by the garbage collection. The FTL does this by keeping track of three different parameters for each partition:

1. The index of the block which is currently in use, i.e., the block where data will be written to next (I_{used}).
2. The index of the last known free block, i.e., the block which does not contain any live data but has not been erased yet (I_{free}).
3. The index of the last block which is free and has already been erased (I_{erased}).

Four different example states of the storage system are shown in Figure 5.4. When the storage system is first initialized (Figure 5.4a), no pages of the partition are yet in use and all of them are free, so I_{used} points to the first and I_{free} points to the last block of the flash medium. Since erasing a block is a time-consuming operation (compare Section 2.5), some, but not all blocks are erased on storage system initialization so that this does not have to be done on-demand. The exact amount of pages that are erased can be configured ($I_{erased} = 5$ for this example).

After some pages have been written, the storage situation could look like that of Figure 5.4b, where some pages are now in use ($I_{used} > 0$) and additional pages have been erased ($I_{erased} = I_{used} + 5$), but I_{free} remains the same. As data is being written to the storage system, I_{used} will reach a pre-defined threshold (50% in the example – Figure 5.4c) at which garbage collection is initiated.

Garbage collection is a process that the FTL cannot complete on its own, since it does not know by itself what data is stored on the subpages and whether or not this data is still needed. As a result, the FTL provides a mechanism to register a garbage collection handler function. This handler is given a block index as well as the amount of blocks to be cleared, and is expected to remove all live data from the requested blocks. After the handler is executed successfully, the cleared blocks will be marked as free by the FTL (Figure 5.4d), that is, the I_{free} index will be incremented by the amount of cleared blocks.

This technique of tracking free data is convenient for the FTL, since it only requires three indices to be stored, as opposed to, e.g., a bitmap of free blocks. In addition, one of these indices is already being stored in the form of the next free subpage to be written (see Section 5.3), since the resident block of that subpage is the last block currently in use. As a result, the FTL only uses an additional eight bytes (four bytes per block index) to track free space across a partition of arbitrary size. While the involved copying operation of live data to new blocks could also be seen as the cost of this space management scheme, one can argue that copying live is always required on flash memory, since per-page deletion of obsolete data is not possible.

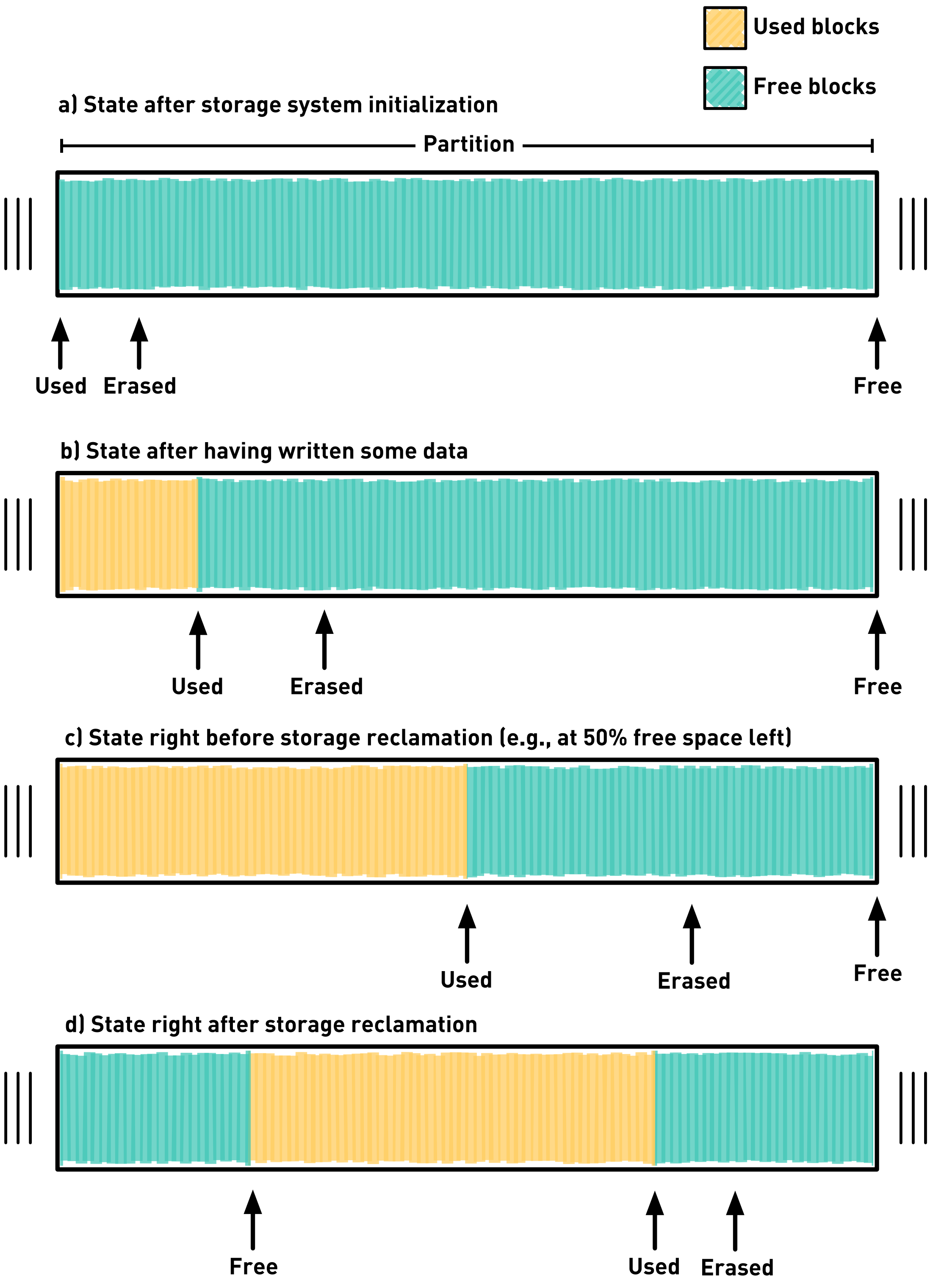

Used blocks
Free blocks
a) State after storage system initialization
Partition
Used
Erased
Free
b) State after having written some data
Used
Erased
Free
c) State right before storage reclamation (e.g., at 50% free space left)
Used
Erased
Free
d) State right after storage reclamation
Free
Used
Erased

Figure 5.4: Overview of the different states encountered when managing free space of a partition.

5.6 Persistent storage of metadata

The state of the FTLs – its metadata – is held in Random-Access Memory (RAM), and thus needs to be persisted by the FTL such that it can be restored if the device crashes or reboots. The metadata changes frequently because it needs to be updated whenever data is written to the flash device. This is a problem when metadata is persisted, since updating already written data is costly on flash memory. Previous approaches to data storage on constrained nodes have employed one of the following techniques for metadata storage:

1. Do not store metadata. Instead, restore it from the flash device itself when the system is initialized [29, 62] – This approach is feasible for small flash capacities, but is inconvenient for larger flash devices. In the worst case, it requires every page of the flash device to be read when the storage system is initialized.
2. Store metadata on a separate storage device which is less restricted in terms of updating already written data[21] – This is a simple solution in terms of storage system design, but requires every embedded device to provide at least two forms of persistent storage.
3. Designate a dedicated area on the flash device for metadata storage [45]

The FTL adopts the last technique, since it is not dependant on the size of the flash device and does not require an additional storage medium to be present. The persistence of metadata in the FTL is provided by the checkpointing mechanism, which writes all the relevant metadata state to a dedicated partition, either periodically or on demand.

Designating a dedicated partition where only metadata is stored efficiently solves the problem of finding the newest metadata entries. If the partition is dedicated only to a certain kind of data (e.g., storage system metadata), and said data is versioned (i.e., it has a monotonically increasing version number), it is possible to apply a variation of the binary search algorithm to find the latest datum in $\mathcal{O}(log(n))$, where n is the amount of pages in the partition. The algorithm in question is shown in Listing 8. It begins by fetching the version number from the center of the partition and comparing it to the beginning. If the version number is greater in the center, it will go on to examine the second half of the partition, otherwise the first. It continues splitting the remaining area in half until the newest version is found.

Listing 8: Adaption of the binary search algorithm used to find the newest index partition entry using a variant of binary search.

```python
# Returns the version number of page
def version_number(page): # -> int

def latest_index(num_pages): # -> int
    pivot = 0
    version = version_number(pivot)
    jump = ceil(num_pages / 2)

    while True:
        next_pivot = pivot + jump
        next_version = version_number(next_pivot)
        if next_version > version:
            pivot = next_pivot
        elif jump > 1:
            jump = ceil(jump/2)
        else:
            return pivot # this is the highest version number!
```

The stored metadata state is composed of its version number, the information which subpage should be written to next (see Section 5.3) as well as the free space management pointers (see Section 5.5). In addition to the FTLs metadata, any storage system built on top of the FTL would also have to store metadata, for example file or object names.Persisting both sets of metadata through different mechanisms (one for the FTL and one for the the system built on top) could lead to diverging information, where both sets of metadata represent a different storage system state. To prevent this, the FTL allows to store arbitrary “foreign metadata” alongside its own.

After metadata has been stored, it also needs to be restored, for example when the device reboots. If no data has been written to the flash device since the last checkpoint, the restoration process is simple, since only the latest checkpoint must be loaded. If this is not the case, that is, if data new data has been written since the last checkpoint, there are two options to handle this discrepancy.

1. Update the metadata state by scanning each record which has been written since the latest checkpoint.

2. Ignore the data which has been written since the last checkpoint and continue based on the latest metadata state.

Option one is costly in terms of energy, but preserves all written data, whereas option two is cheap in terms of energy but loses all data written since the latest checkpoint. Which approach is applicable ultimately depends on the application requirements.

5.7 Application Programming Interface

The usability of the APIs a software system provides can have a significant impact on its adoption and sustained use. As such, the design of usable APIs can be seen as critical for any software project, but especially for open-source projects which aim at widespread adoption. Designing an API involves many stakeholders, such as API producers, consumers and testers, all of which need to write software based on API prototypes, evaluating whether the API meets their needs [23]. As a result, the APIs presented in this thesis are prototypes based on feedback from the RIOT community. For final inclusion in the RIOT kernel, the API is expected to change based on further feedback from kernel developers and the community. In case of the FTL, the amount of functionality that needs to be exposed is relatively small:

- System initialization
- Writing, reading and erasing data
- Storing metadata
- Configuring the ECC algorithm

System initialization is performed by populating a struct with the required configuration parameters and invoking the FTL initialization function. An example is shown in Listing 9. The configuration parameters consist of basic information about the flash device (L6–11), function pointers for the flash device driver interface (L13–16) as well as the necessary buffers (L18–19). In addition, all partitions must be defined before the FTL is initialized (L22–43) and a list of partitions must be made available in the configuration (L46). Finally, "ftl_init" initializes the storage system, such that data can be read or written once it returns successfully.

When examining the flash driver interface, one might notice that there are two different functions for the erase operation, “erase” and “erase_bulk". Since erasing blocks in bulk is not supported by all flash storage devices, the FTL will detect whether or not this feature is supported by checking if the function pointer to ”erase_bulk" is set. If not, block erasure will be performed with the “erase” function.

Writing data is performed using either “ftl_write_raw", ”ftl_write" or "ftl_write_ecc" (see Listing 10). These functions correspond to the three options for writing data described in Section 5.3. While writing in raw mode requires the selection of a target subpage, writing in managed mode will automatically write to the next free subpage and return its number. When specifying pages (or blocks), these are always relative to the beginning of their respective partition, making it impossible to accidentally modify data outside of the selected partition.

Listing 10: Example usage of the API for writing data using the FTL

```
// Signature
int ftl_write_raw(const ftl_partition_s *partition, unsigned char *buffer,
                  uint32_t subpage);
// Example
int ret = ftl_write_raw(&firmware_partition, &buffer, 42);

// Signature
int ftl_write(const ftl_partition_s *partition, const unsigned char *buffer,
              uint16_t data_length);
// Example
int subpage = ftl_write(&data_partition, &buffer, 100);

// Signature
int ftl_write_ecc(const ftl_partition_s *partition, const unsigned char *buffer,
                  uint16_t data_length);
// Example
int subpage = ftl_write_ecc(&data_partition, &buffer, 100);
```

Reading data is also possible in either raw or managed mode using “ftl_read_raw" or ”ftl_read" (see Listing 11). The only relevant difference is that, when reading in managed mode, a header struct must be passed to the "ftl_read" function so that the header information can be returned. The managed read call will automatically recognize if an ECC is present and verify the integrity of the subpage. Erasing data can be performed per erase block (used in garbage collection) or per partition (see Listing 12).

Listing 11: Example usage of the API for reading data using the FTL

```c
// Signature
int ftl_read_raw(const ftl_partition_s *partition, unsigned char *buffer,
                 uint32_t subpage);
// Example
int ret = ftl_read_raw(&firmware_partition, &buffer, 42);

// Signature
int ftl_read(const ftl_partition_s *partition, unsigned char *buffer,
             subpageheader_s *header, uint32_t subpage);
// Example
subpageheader_s header;
int ret = ftl_read(&data_partition, &buffer, &header, 42);
```

Listing 12: Example usage of the API for erasing data using the FTL

```c
// Signature
int ftl_erase(const ftl_partition_s *partition, uint32_t block);
// Example
int ret = ftl_erase(&data_partition, 123);

// Signature
int ftl_format(const ftl_partition_s *partition);
// Example
int ret = ftl_format(&firmware_partition);
```

For metadata storage (see Section 5.6) the functions shown in Listing 13 were designed. “ftl_write_metadata" stores the FTLs metadata in a dedicated partition. Through the ”foreign_metadata" parameter, it allows to store additional data not related to the FTL. This mechanism is used, for example, to store the state of the OSL. Storing both states using the same mechanism ensures that both states represent the same point in time. After storing it, the FTL can either load the latest metadata or a certain metadata version (“ftl_load_latest_metadata" and ”ftl_load_metadata" respectively). The "set_ftl_state" parameter dictates whether or not the state stored in the retrieved metadata should supersede the current state of the FTL.

Listing 13: Example usage of the API for storing metadata using the FTL

```c
// Signature
int ftl_write_metadata(ftl_device_s *device, const void *foreign_metadata,
                       uint16_t length);
// Example which stores only the FTL's metadata
int ret = ftl_write_metadata(&device, NULL, 0);

// Signature
int ftl_load_latest_metadata(ftl_device_s *device, void *buffer,
                                  ftl_metadata_header_s *header, bool set_ftl_state);
// Example which will update the FTL's state to the latest metadata
ftl_metadata_header_s header;
int ret = ftl_load_latest_metadata(device, foreign_metadata_buffer, &header, true);

// Signature
int ftl_load_metadata(ftl_device_s *device, void *buffer, ftl_metadata_header_s *header,
                      uint32_t version, bool set_ftl_state);
// Example which retrieves a specific metadata version but
// does not update the FTL's state
ftl_metadata_header_s header;
int ret = ftl_load_metadata(&device, foreign_metadata_buffer, &header, 123, false);
```

Finally, the FTL allows the specification of a custom ECC computing and verification function as well as the size of the resulting ECC (see Listing 14). Both are passed a data buffer of given size, for which the ECC is computed or verified. Note that it is only possible to change the ECC algorithm before any data has been written. If the ECC was changed after data has been written to the storage system, the data would be assumed to be corrupt when next checked because the ECC written using the previous algorithm would be verified using the new algorithm.

Listing 14: Example usage of the API for changing the ECC algorithm of the FTL

```c
typedef int (*ecc_compute)(const unsigned char *data, uint32_t size,
                           unsigned char *code);
typedef int (*ecc_verify) (unsigned char *data, uint32_t size,
                           const unsigned char *code);
// Signature
int ftl_set_ecc(ftl_device_s *device, uint16_t size,
                ecc_compute compute_fn, ecc_verify verify_fn);
```

5.8 Discussion

In summary, the functionality provided by the FTL is intentionally kept simple, as it is primarily meant to provide a robust foundation for the Object Storage Layer (OSL) to be presented in the following chapter. Unlike Capsule's Flash Abstraction Layer, however, it is also designed to be used on its own, or to provide a basis for other possible storage systems to be implemented in the future.

The FTL supports a wide variety of flash devices, including NOR flash as well as SLC and MLC NAND flash. A major limitation for the usage of NAND flash memory on Class 1 embedded devices is the memory impact of the required buffers. By dividing pages into subpages the FTL tries to alleviate this problem. Because MLC NAND flash does not support this subdivision – and commonly features page sizes of 4 KiB or more – it is conceivable that this type of flash will not find widespread adoption for Class 1 IoT appliances.

Listing 9: FTL API example configuration of an 8 MiB flash device with a page size of 512 Bytes and $k = 4$, such that $\frac{512}{4} = 128$ Bytes $=$ subpage size. The configuration has an index partition as well as a "sensordata" partition. The interface to the flash driver (`flash_driver_*` functions) is omitted in this example.

```c
#import "storage/ftl.h"
unsigned char subpage_buffer[128];
unsigned char ecc_buffer[6];

ftl_device_s device = {
    .total_pages = 16384,
    .page_size = 512,
    .subpage_size = 128,
    .pages_per_block = 1024,
    .ecc_size = 6,
    .partition_count = 2,

    ._write = flash_driver_write,
    ._read = flash_driver_read,
    ._erase = flash_driver_erase,
    ._bulk_erase = flash_driver_bulk_erase,

    ._subpage_buffer = subpage_buffer,
    ._ecc_buffer = ecc_buffer
};

ftl_partition_s index_partition = {
    .device = &device,
    .base_offset = 0,
    .size = 3,
    .next_page = 0,
    .erased_until = 0,
    .free_until = 0
};

ftl_partition_s sensordata_partition = {
    .device = &device,
    .base_offset = 3,
    .size = 2,
    .next_page = 0,
    .erased_until = 0,
    .free_until = 0
};

ftl_partition_s *partitions[] = {
    &index_partition,
    &sensordata_partition
};

int main(void) {
    device.partitions = partitions;
    int ret = ftl_init(&device);
    assert(ret == 0);

}
```

6 Design of the Object Storage Layer

While the design of the FTL is mainly shaped by the constraints of the underlying flash storage, the design of the Object Storage Layer (OSL) is primarily meant to provide a usable and feature-rich storage system for common WSN and IoT use cases. The OSL resembles the object-based storage paradigm as used by Capsule [44], storing data in the form of so-called "storage objects", each providing a basic data structure and exposing an API tailored to the object.

In this chapter, the physical structure of stored data (Section 6.1) and the way data is buffered before being written to flash (Section 6.2) are examined first. Subsequently, the implications of the designed storage structure and the supplementary mechanism necessary for its proper operation are discussed in Section 6.3 to Section 6.5. The following Section 6.6 details the core concept of the OSL, its storage objects and their features, followed by how their state is held in RAM and how it is persisted to flash memory (Section 6.7). Section 6.8 examines the problem of knowing whether or not an object with a certain name exists, which is relevant trying to retrieve metadata of a certain object from flash, i.e., opening it (Section 6.9). The subsequent overview of the garbage collection (Section 6.10) and thread safety (Section 6.11) mechanisms is followed by the presentation of the OSLs API (Section 6.12). The chapter is concluded by a discussion of the chapter in Section 6.13.

6.1 Storage structure

All storage objects provided by the OSL share the same basic structure when stored on flash memory. A storage object is composed of one or more records which form a backward-pointing log, i.e., the n-th record always points to the $(n-1)$-th record. Whenever data is added to an object, a new record reflecting the change is created.

An example showing a possible on-flash composition of two storage objects and a total of five records is shown in Figure 6.1. Each record has a header which stores the predecessor and the record length (compare Listing 15). The pointer to the predecessor is stored as a subpage index and the offset inside the subpage where the record begins (see Figure 6.2). A backward-pointing log-structured approach was chosen because it meets all the requirements for usage on both NOR and NAND flash storage. In addition, many of the previously storage systems, such as Capsule [44] and ELF [21], have shown that log structured storage is a good fit for flash devices.

The disadvantages of the backward-pointing log-structured approach are twofold. First, adding an eight byte header to every record, whose data might be as small as one byte, comes with a large storage overhead. This problem is discussed in Section 6.3. Second, this approach implies with poor random-read and front-to-back iteration performance. This isssue is discussed in Section 6.4.

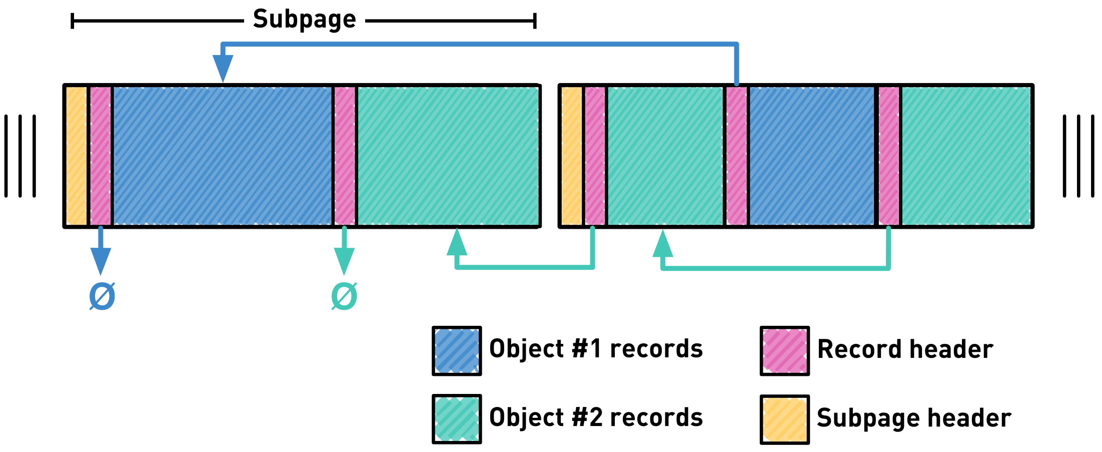


Figure 6.1: Physical OSL storage structure example using two objects spread across two subpages. The subpage header (yellow) is added by the FTL.

6.2 Buffering records

Modifying an object of the OSL does not immediately store requested changes on flash memory, since the modification may be much smaller than the size of a subpage. Instead, a buffer in RAM is used to accumulate object records until they

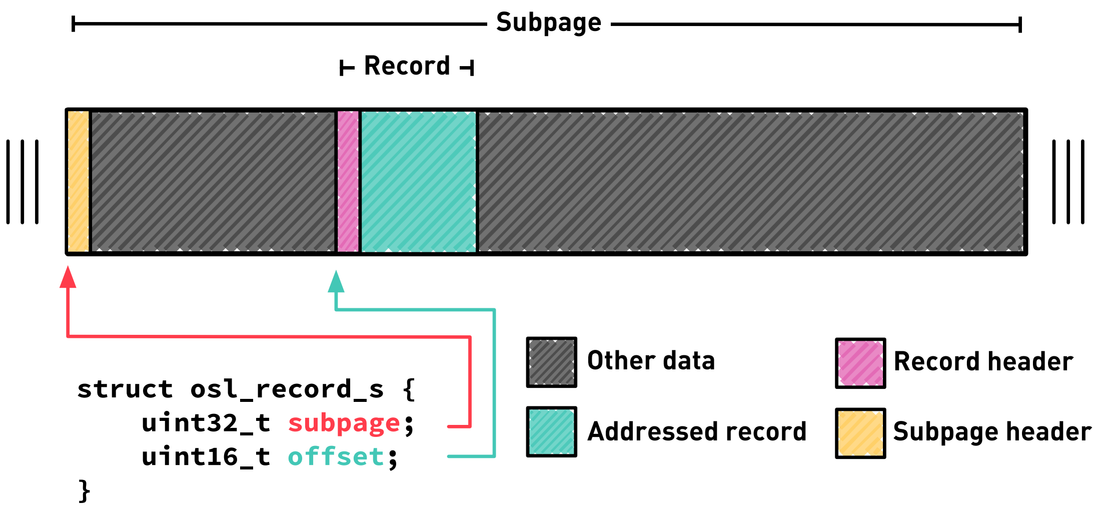


Figure 6.2: Visualization of the way in which records are addressed by the OSL.

Listing 15: OSL Record header in C packed structure notation.

```c
typedef struct osl_record_s {
    uint32_t subpage;
    int16_t offset;
} osl_record_s;

typedef struct __attribute__((__packed__)) {
    osl_record_s predecessor;
    unsigned int length:15;     //!< Length of the data contained in this record
    unsigned int is_first:1;    //!< Is this the first record of the log?
} osl_record_header_s;
```

exceed the size of a single subpage and then “flushes” the changes to flash memory. The buffer is automatically flushed whenever a checkpoint is created (see Section 6.7).

6.3 Combined records

An eight byte record header per data element is an efficiency problem, because the element stored in the record may be as small as one byte, resulting in a storage overhead of 800%. Even a 64 bit integer per record would still result in an overhead of 100%. In other words, storing one Gigabyte of 64 bit sensor data would require another Gigabyte of record headers.

To improve efficiency, the OSL combines consecutively written data elements into the same record if they belong to the same object. As a result, a new header will only be created if a different object is written or if the subpage is full. Compared to the naive approach of writing one record per data element – which results in a constant userdata to metadata ratio - the combined approach performs depending on the number of interleaved objects on the subpage, with a best-case user data percentage of 96%.

This is visualized in Figure 6.3 for a subpage size of 256 bytes. When not combining records, the user data percentage is constant and proportional to the size of the data element. While an eight byte data element achieves 50% user data, a one byte element only achieves 12.5%. When combining records, the user data percentage depends on how many data elements of the same object are written consecutively, because, as soon as a data element from another object is written (interleaved), a new record header must be created as well.

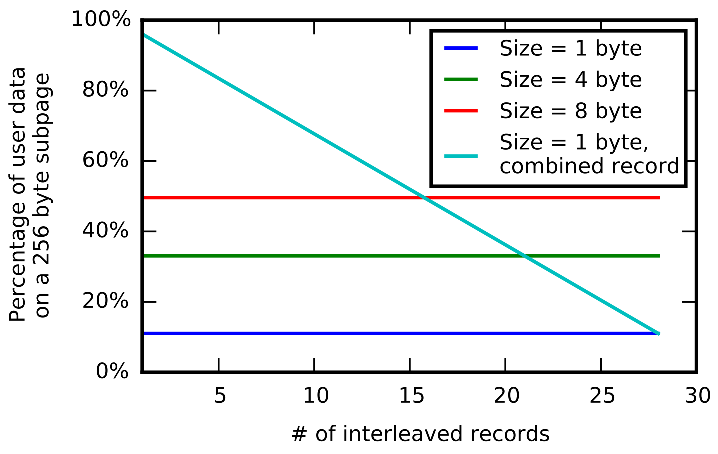


Figure 6.3: Comparison between naive and combined record storage with element sizes of one, four, and eight bytes, assuming a page size of 256 bytes.

6.4 Record caching

The backward-pointing approach has a severe negative side-effect in terms of read performance. Given an object with n log records, the complexity of sequentially traversing the log structure is $\mathcal{O}(n^2)$, since $r(n)$ records need to be traversed (Equation 6.1) in order to iterate through such an object from the beginning to the end. This is because, given any record in the backward-pointing log, there is no simple way to determine its successor. As a result, one must always start start at the end of the log and traverse it until the desired log record is found. Note that this is not an issue when sequentially traversing the log from the end towards the beginning, in which case iteration complexity is $\mathcal{O}(n)$.

In order to improve performance when iterating through an object, a read cache of configurable size k is introduced. This read cache stores the page numbers of k log records, so that, when accessing a record in the object, log traversal can start from the nearest cache entry instead of the end of the log. This is exemplified in Figure 6.4. The first cache entry always points to the subpage that was last read, since it is probable that records near it are going to be read as well (principle of locality). The remaining cache entries are evenly distributed along the log records of the object. If an object has 500 log entries and the cache size is five, for example, the cached records would be 100, 200, 300 and 400, plus the last read subpage.

$$n + (n-1) + (n-2) + ... + 2 + 1 = \frac{n(n+1)}{2} = r(n) \tag{6.1}$$

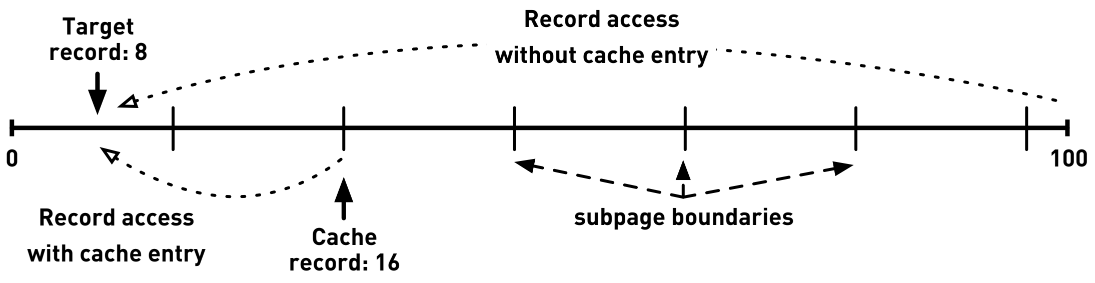

Target record: 8
Record access without cache entry
0
100
Record access with cache entry
Cache record: 16
subpage boundaries

Figure 6.4: Example of accessing a record of an object – which has a total of 100 records – with and without cache entry

6.5 Object defragmentation

The caching technique described in Section 6.4 improves the sequential read performance by reducing n, i.e., the number of records which need to be traversed by a factor proportional to k. As an the number of records in an object grows, however, the performance will still degrade exponentially, since the size of the record cache stays the same.

To provide good read performance even for storage objects containing many records, an additional – compacted - storage mode similar to the “Disk Mode” of FlashDB [51] is introduced. When a storage object grows beyond a certain size, it is transitioned from the normal log-structured mode into the compacted form. In this process, the records of the object are rewritten to sequential pages on flash memory (see Figure 6.5). Since the compacted form guarantess that the logical successor of a record is also its physical successor on flash memory, sequentially reading in compacted mode has a complexity of $\mathcal{O}(n)$. After defragmentation, new records are again written in log-structured mode until the defragmentation threshold is reached.

6.6 Storage objects

After having examined the physical structure of storage objects, this section presents the taxonomy of the OSLs storage objects and their applications (see Figure 3.8). The storage objects provided by the OSL are mostly the same as the ones suggested by the authors of Capsule [45], given that they cover most of the typical data storage requirements that can be expected in IoT scenarios. Unlike Capsule, however, the taxonomy includes a Cache Table object, which is primarily intended to accommodate ICN cache data. An overview of the provided objects as well as their supported operations is shown in Figure 6.7.

The Stream is a core component of the OSL, allowing to store records from different types of sensor sources for archival, querying and processing. It is an append-only data structure, meaning that it is not possible to modify elements once they have been added to the stream. Appending an element to a stream creates a new log entry and lets the metadata head field point to it. The stream object also allows to

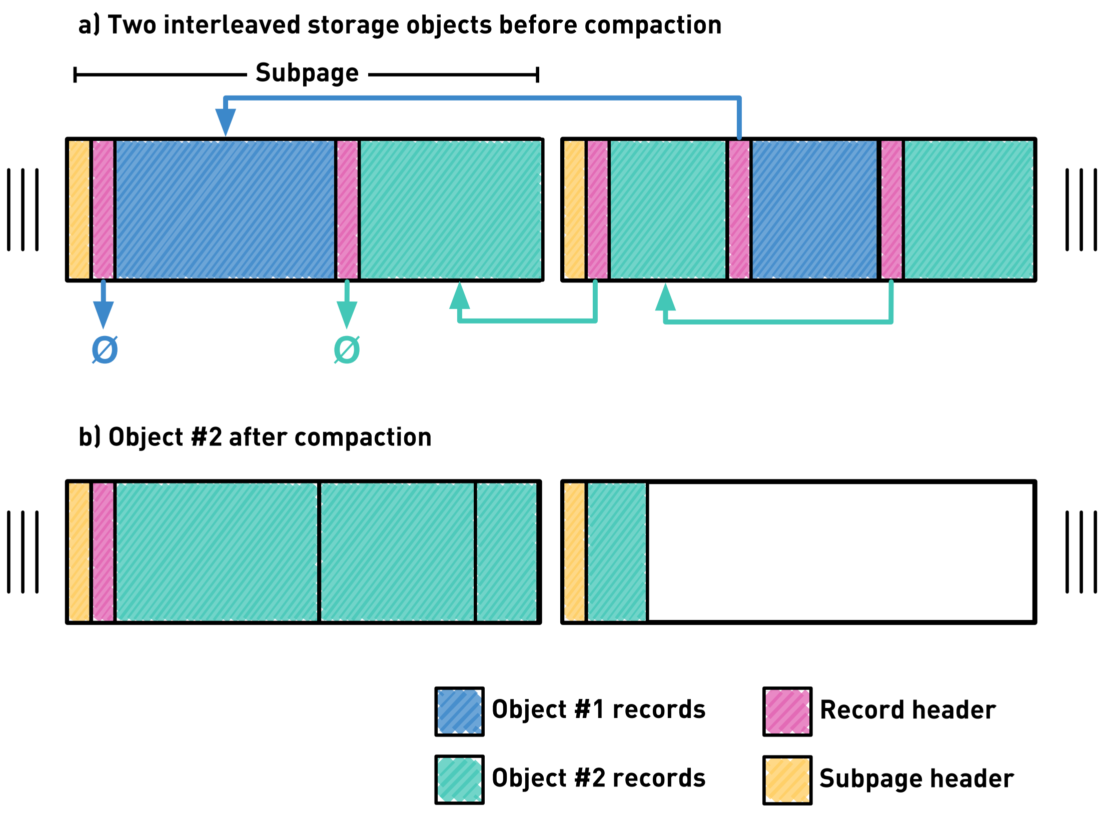

a) Two interleaved storage objects before compaction
Subpage
Ø
Ø
b) Object #2 after compaction
Object #1 records
Record header
Object #2 records
Subpage header

Figure 6.5: Example of the OSLs storage object defragmentation. The two subpages in b) are not the same subpages as in a)

Application	Data type	Storage object
Archival storage (and querying)	Sensor data	Stream
Data processing	Array	Index
Network routing	Packet buffer	Queue/Stack
Debugging logs	Time-series logs	Stream
ICN Cache	Named binary data	Cache Table

Figure 6.6: Taxonomy of the OSLs storage objects and their applications.

Storage object	Operations
Stream	append(value), get(index)
Index	add(key, value), find(key), remove(key), query(min, max)
Queue	add(value), remove(), peek()
Stack	push(value), pop(), peek()
Cache Table	set(key, value), get(key)

Figure 6.7: Overview of the OSLs storage objects as well as the operations they support.

discard the n oldest elements of the stream, such that the object can be trimmed if it grows beyond a certain size. This operation advances the tail metadata field by the desired amount of elements to be dropped. The Stream operations are visualized in part one of Figure 6.8.

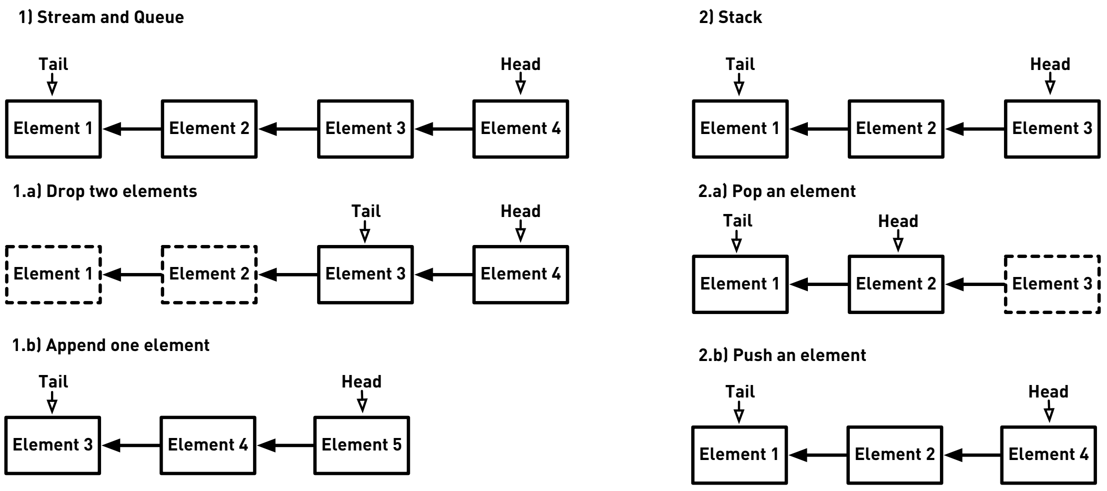


Figure 6.8: Operations on the OSL objects Stream, Queue and Stack visualized

Queues and Stacks are data structures encountered in many applications, and as such they are provided as a general measure to allow RAM usage of embedded applications to be reduced by storing such data on flash memory (e.g., the OS packet buffers). The addition of the "drop" operation to the Stream – which Capsule did not implement on Streams – not only allows them to be trimmed, but also enables the Queue object to be implemented as a special case of a Stream. The "add(value)" operation of a Queue is equivalent to the "append(value)" operation of the Stream, the “remove()” operation of the Queue is the same as “drop(1)” in the Stream, and the Queue’s “peek()” is equal to Stream’s “get(0)”. The Stack operations are shown in part two of Figure 6.8. Pushing an element to a Stack is equivalent to appending an element to a Stream, and popping an element off a Stack sets the head pointer to the predecessor of the head element being popped.

The OSL also provides a data structure not previously seen on storage systems for flash memory. It is tailored to the needs of ICNs, allowing arbitrary, named binary data to be stored in a flash-optimized “Cache Table”. This data structure is based on the commonly known hash table, which provides named access to data divided into buckets depending on their name’s hash value. Similarly, the “Cache Table” manages a dedicated FTL partition and caches named ICN data among the blocks of the partition based on the data’s name hash.

An example of a three-block cache table is shown in Figure 6.9. When an element should be inserted – using “set(key, value)” – the OSL calculates the hash of its key modulo the amount of blocks, and thus decides into which block the value should be stored. The data is then written into the first unused subpage in that block. Retrieving an element from the Cache Table is done by determining in which block the element lives based on its name, and then iterating through the pages from the newest to the oldest subpage, until the element is found. This has the side-effect that overwriting an existing key does not involve any extra operations, since it will always be found before the older version. When any block becomes full, it is simply erased. This is possible because there is no guarantee on the lifetime of cached data in ICNs. An alternative approach where cache data is not discarded would require two blocks per Cache Table bucket, such that, when a block is full, the remaining live data in the full block could be copied to the empty one. However, this would also necessitate tracking whether data is obsolete, and would thus further complicate Cache Table management.

The Index object is intended to be used as a means of storing associated key-value pairs and quickly retrieving either concrete values based on a key or ranges of values based on a query. In contrast to the previously presented storage objects, there are many variants of indexing data structures which are useful under different circumstances and in different applications. As a detailed evaluation and selection of suitable data structures is out of scope of this thesis (compare Section 3.4), no distinct Index object designs are presented here. Instead of providing a concrete

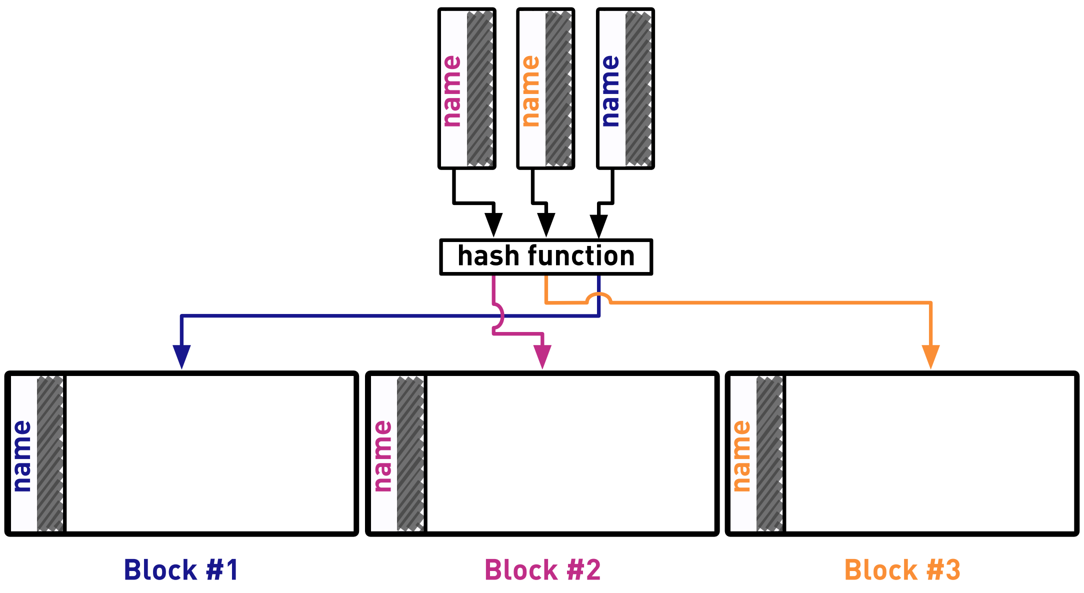


Figure 6.9: Distribution of named data based on name hash in a OSL Cache Table with three blocks

Index object, the OSL facilitates the integration of custom indexing data structures (see Section 6.12).

6.7 Metadata

The OSL keeps information about a limited number of objects in RAM. These are the objects which are considered “open”. How many objects can be open simultaeneously is defined at compile time. Every storage object held in RAM consists of the fields shown in Listing 16. The “head” and “tail” fields designate the last and first record which belongs to the storage object log. The location of the first record is necessary for garbage collection purposes (see Section 6.10).

The next field is the “name hash”. While, in the OSLs API, objects are addressed by human readable string names, these are internally converted to a 64-bit hash to allow object names of arbitrary length and to save memory. The only requirement for the hash function is that it guarantees a uniform distribution of hashed values. Such a hash – of any length – can be truncated to 64 bits, with the uniform distribution criterium being maintained [37]. While the chance for collision exists, its

Listing 16: OSL structure for storing object information

```c
typedef struct osl_object {
    osl_record_s head;
    osl_record_s tail;
    uint64_t name_hash;
    uint32_t num_records;
    uint16_t record_size;
    uint8_t type;
} osl_object_s;
```

likeliness is minimal, especially given that an embedded IoT application is unlikely to create millions of files. The approximate collision probability is given by the Poisson approximation of the Birthday Paradox (compare Equation 6.2), where n is the space of possible hash values and k is the number of hashed values. For $k = 200,000$ object names and $n = 2^{64}$, for example, the chance of collision is 1.08×10^{-9}, i.e., one in a billion.

$$P(n,k) \approx 1 - e^{\frac{-k^2}{2n}} \tag{6.2}$$

The remaining three fields are for tracking the amount of records in the log, the record size (which is fixed per object) and the type of the object (e.g., stream or queue). In addition to the object data, the OSL also keeps a dedicated subpage buffer (see Section 6.2) as well as the record cache (see Section 6.4) in RAM.

Storing metadata primarily on flash would be hard to update due to its write-once property. As a result, it is stored primarily in RAM. Because of this, the OSL needs a mechanism to achieve persistence for when the device loses power or crashes. As a solution, the array of open storage objects is periodically, or on demand, stored on flash memory using the FTLs metadata persistence API (see Listing 13). As in Capsule [46], the OSL calls this process “checkpointing”. Creating a checkpoint also causes the subpage buffer (Section 6.2) to be flushed.

6.8 Checking object existence

Retrieving an object's persisted metadata from flash memory is costly since it potentially involves reading several subpages (see Section 6.9). To prevent searching on flash for objects that do not exist, the OSL persists a Bloom filter populated with the name hashes from all objects which it currently manages.

A Bloom filter [11] is a space-efficient, probalistic data structure which allows to test whether an element is member of a set. It is probabilistic, since it has a chance of returning a false positive. This chance increases with the number of members in the set. The Bloom filter is used to efficiently determine whether or not an object with a given name exists. To achieve this, the Bloom filter is populated with the names of all existing objects.

As mentioned above, a Bloom filter has a non-zero chance of a false positive. The probability of such an event depends on three parameters:

- The size of the Bloom filter m (in bits)
- The number of elements in the set (n)
- The number of *independent* hash functions k. The optimal value for k is a function of m and n (see Equation 6.3).

To choose the optimal size of the Bloom filter, it is thus necessary to estimate how many objects are expected to exist at the same time in common IoT applications. Unfortunately, due to lack of data, a reliable estimate could not be made. As a result, a compromise between Bloom filter size and false positive rate based on Figure 6.10 was made. The Bloom filters of 128 and 256 bits have a high false positive rate even with under 50 elements, whereas 1024 bits (128 bytes) is excessive when considering that subpage sizes of 256 bytes are realistic. An experimental size of 512 bits was thus chosen for the Bloom filter.

$$k = \frac{m}{n} \ln 2 \tag{6.3}$$

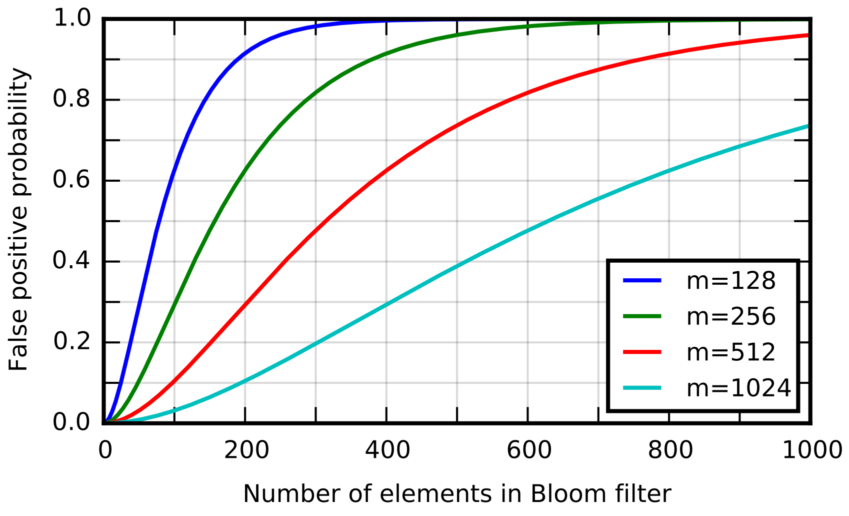


Figure 6.10: False positive probability for a Bloom filter of different sizes (in bits)

6.9 Opening and closing objects

The process of opening an object involves checking whether it already exists, and if so, finding its metadata in RAM or on flash memory. The algorithm is shown as Python sample code in Listing 17. The first phase simply scans the objects in RAM for an object with the requested name, and returns it. If non is found, the latest metadata is loaded and the Bloom filter is consulted to know whether or not an object of the requested name exists. If that is the case, or in case of a false positive, the OSLs metadata is scanned from newest to oldest metadata version, until either the requested object is found or the oldest version is reached.

Closing an object does not immediately delete it from RAM, since it is likely that a previously opened file is re-opened at by the application at a later time. Instead, the object is marked as “closed” and will only be evicted from RAM once another object which is not currently held in memory is requested for opening. Before evicting an object from flash memory, its metadata is written to flash such that its current state is persisted.

Listing 17: Algorithm employed by the OSL to find the metadata of an object requested to be openend (in Python-based sample code)

```python
def find_object_metadata(name):
    # Search in RAM
    for obj in objects_in_ram:
        if obj.name == name:
            return obj

    # Consult Bloom filter
    version = latest_metadata_version()
    metadata = load_metadata(version)

    # Check if the object exists on flash memory
    if not metadata.bloom_filter.contains(name):
        return False

    # Search on flash memory
    while version >= 0:
        for obj in metadata.objects:
            if obj.name == name:
                return obj

        version -= 1
        metadata = load_metadata(version)

    return False
```

6.10 Garbage collection

On initialization, the OSL registers itself with the FTLs as to handle garbage collection (compare Section 5.5). Whenever the FTL needs to free additional space, it will invoke said handler and instruct the OSL to copy all remaining live data on one or more blocks to the end of the log structure. To identify which objects need to be copied, the OSL has to iterate through all of them. For each object, the system checks whether or not the object's head record (see Section 6.7) is located on one of the pages to be erased. If that is the case, the object is added to the list of objects that must be copied. After all objects have been examined, the ones from the list are then rewritten to flash and their metadata is updated. After this process is completed, control is returned to the FTL, informing it which blocks have been cleared. Note that, in the process of rewriting objects to flash, they are also automatically compacted into “Disk Mode” (see Section 6.5), which improves subsequent read performance and reduces metadata overhead for the objects in question.

6.11 Thread safety

In multi-threaded desktop OS it is common that many processes write or read data simultaenously, and that their performance depends on that the host OS is able to accommodate these operations in a timely fashion. Being able to provide this functionality, however, comes with significant complexity and resource usage (e.g., for buffers) at the OS-level.

To avoid this complexity for the storage system at hand, the OSL does not include the ability for concurrent write or read operations. In a multi-threaded OS, the OSL provides minimalistic safe-guarding against errors caused by concurrent operations by using a simple locking strategy composed of two locks.

1. The flash device lock safeguards against concurrent write or read operations on the FTL and thus the flash device. Since, on the FTL level, both use a shared buffer, it is necessary that only a single write or read operation is active at any given time.
2. The write buffer lock safeguards against concurrent use of the OSL record buffer. This buffer can be filled while another FTL operation is in use, but as soon as it is full – and thus needs to be flushed – the flash device lock must be acquired.

6.12 Application Programming Interface

The OSL exposes an API which completely abstracts the special properties of flash memory in order to provide a storage system for IoT devices which is as simple to use as possible. As indicated in Section 5.7, this API is only a prototype and is expected to change before inclusion into the RIOT kernel. The OSL design – in its current state – features the following functionality:

- System initialization

- Creation, retrieval and modification of the object types listed in Section 6.6
- Iteration through an object
- Defragmentation of existing objects
- Creation of a metadata checkpoint

Initializing the OSL requires that the FTL has already been successfully initialized, but is otherwise very simple, as can be seen in Listing 18. The listing also shows how a checkpoint is created.

Listing 18: API for initialization and checkpointing of the OSL

```c
// Signatures
int osl_init(osl_s *osl, ftl_device_s *device, ftl_partition_s* data_partition);
int osl_create_checkpoint(osl_s* osl);
// Example
osl_s osl;
int main() {
    // The FTL must already be initialized in "device".
    int ret = osl_init(&osl, &device, &data_partition);
    [...]
    osl_create_checkpoint(&osl);
}
```

In the OSL, each storage object comes with its own function for creation, modification and access, but their usage is very similar. Opening a storage object always requires a reference to the OSL configuration as well as an Object Descriptor (OD). The latter contains all information necessary to perform operations on an object, and is used is passed to all functions operating on an object after it has been created. Listing 19 shows the API of the Stream object. In the example, the Stream is set up to store `uint32_t` records and is given the name "some stream" (L7). In line 9 and 10, an element is then appended to and retrieved from the Stream, respectively. The very similar Queue and Stack APIs are shown in Listing 20.

As identified in Chapter 4, post-processing of stored data is an important task for IoT and WSN applications. The OSL currently supports simple iteration over Streams, Queues and Stacks using the API shown in Listing 21. To iterate over an object, the application developer creates an `osl_iter` struct – which holds the

Listing 19: API of the Stream object provided by the OSL

```c
// Signatures
int osl_stream(osl_s* osl, osl_od* od, char* name, size_t object_size);
int osl_stream_get(osl_od* od, void* object_buffer, uint32_t index);
int osl_stream_append(osl_od* od, void* object);
// Example
osl_od some_stream;
osl_stream(&osl, &some_stream, "some stream", sizeof(uint32_t));
uint32_t x = 42;
osl_stream_append(&some_stream, &x); // Append the value of x
osl_stream_get(&some_stream, 0);     // Get the first element of the stream
```

state of the iteration – and initializes it with the object over which they wish to iterate (L8–9). The "osl_iterator_next" function is then called – here in a while loop – until it returns `false`, indicating the the iteration has finished. If desired, the index of the current element can be extracted from the `osl_iter` structure as shown in line 11.

The API for the Cache Table differs from the previously examined storage objects, because it operates on an entire FTL partition. Creation of a Cache Table thus requires a reference to the partition on which it should be created. It does not need a name, since the partition already unique identifies it, i.e., there can not be multiple Cache Tables per partition. The API for storing and retrieving data (L14–15) is also somewhat more complicated compared to the previous objects, since the Cache Table elements must not be of fixed size. As a result, both setting and retrieving an element requires the length of the key and the value to be given, since they are not inferable from the void pointers once passed to the OSL. In the case of retrieving data from the Cache Table, the "value_size" parameter is set by the OSL such that the application developer knows how much data was written to the buffer. Finally, it is possible to wipe all data from a Cache Table using the "format" function (L17).

Since there are many different kinds of index data structures, which vary in their intended applications, the Index object API is built to provide the ability to be extended with different index types depending on the requirements of the application. Listing 23 shows the API of the Index object, which can either be created by specifying a type of Index to be used (L2–3), or by creating an Index object with the default type (L4–5). It is then possible to add, retrieve, and remove key-value pairs

Listing 20: API of the Queue and Stack objects provided by the OSL

```c
// Signatures
int osl_queue(osl_s* osl, osl_od* od, char* name, size_t object_size);
int osl_queue_add(osl_od* od, void* item);
int osl_queue_peek(osl_od* od, void* item);
int osl_queue_remove(osl_od* od, void* item);
// Example
osl_od some_queue;
osl_queue(&osl, &some_queue, "example queue", sizeof(uint8_t));
uint8_t x = 123;
osl_queue_add(&some_queue, &x);   // Add the value of x
osl_queue_peek(&some_queue &x);   // Set x to the first element of the queue
osl_queue_remove(&some_queue &x); // Set x to the first element and remove
                                  // the value from the queue

// Signatures
int osl_stack(osl_s* osl, osl_od* od, char* name, size_t object_size);
int osl_stack_push(osl_od* od, void* item);
int osl_stack_peek(osl_od* od, void* item);
int osl_stack_pop(osl_od* od, void* item);
// Example
osl_od some_stack;
osl_stack(&osl, &some_stack, "example stack", sizeof(uint64_t));
uint64_t x = 123;
osl_stack_push(&some_stack, &x);  // Push the value of x
osl_stack_peek(&some_stack &x);   // Set x to the top element of the stack
osl_stack_pop(&some_stack &x);    // Set x to the top element and pop the
                                  // value off the stack
```

from the index (L15–17). In addition, the API allows to query ranges of keys (L24). This API call is notable because it is passed a Stream object as a parameter, into which the resulting values of the range query are written. This is done because it is not possible to know how many elements the requested range contains. Once the query has completed, the results can then be processed as previously shown in the Stream object API.

The API for adding a new index type requires a number of functions to be implemented and to be registered with the OSL, using the “osl_index_register_type" function, as shown in Listing 24. If successful, it returns a type number which can then be used when creating an Index object. The operations that must be implemented are essentially the same as the ones provided by the Index object (see Listing 23). While all index type must support the ”add“ and ”find“ operations, the function pointers for ”remove“ and ”query" may be set to NULL if the index does not

Listing 21: API of the object iteration functionality provided by the OSL

```c
// Signatures
int osl_iterator(osl_od* od, osl_iter *iterator);
bool osl_iterator_next(osl_iter *iter, void *buffer);
// Example
osl_od some_stream; // Assuming a stream one or more uint16_t elements
                    // has been created and is referenced by this descriptor
uint16_t x;
osl_iter iterator;
osl_iterator(&some_stream, &iterator);
while(osl_iterator_next(&iterator, &x)) {
    printf("Index: %"PRIu32" Value: %"PRIu16"\n", iterator.index, x);
}
```

support them. Finally, it is possible to modify the default index type by calling the function shown in L9 with one of the previously registered index types.

6.13 Discussion

Internally, the record-based, backward-pointing log-structure is the OSLs most prominent design feature, as this chapter has shown. While this storage organization supports virtually all types of flash memory in existence today, it is not without downsides. It requires several different techniques – such as record caching, combined records, and object defragmentation – to provide acceptable, i.e., non quadratic, performance when iterating from the beginning of the log towards the end. Given that iterating from the end to the beginning of the log exhibits linear performance, this is only an issue when the iteration order is relevant.

It should be noted that, while Capsule provides an “Index + Stream”, due to the time constraints of this thesis this object has not yet been designed for the OSL. Furthermore, the usage of the Bloom filter for tracking object existence was a late addition, and was thus not included in the implementation and the evaluation. Whether the usage of the Bloom filter benefits the application may depenend on its storage behavior, e.g., how many objects it uses and how often it opens and closes them.

Listing 22: API of the Cache Table object provided by the OSL

```c
// Signatures
int osl_cache_table(osl_s* osl, osl_od* od, ftl_partition_s* partition);
int osl_cache_table_set(osl_od* od, void* key, void* value, size_t key_size,
                        size_t value_size);
int osl_cache_table_get(osl_od* od, void* key, void* value, size_t key_size,
                        size_t *value_size);
int osl_cache_table_format(osl_od* od);
// Example
osl_od ct;
size_t size;
unsigned char buffer[128];
char *name = "test element";
osl_cache_table(&osl, &ct, cache_partition, 128);           // Sizes chosen arbitrarily
osl_cache_table_set(&ct, name, buffer, strlen(name), 42);   // Insert an element of 42 bytes
osl_cache_table_get(&ct name, buffer, strlen(name), &size); // Retrieve an element.
                                                            // Size is set to 42
osl_cache_table_format(&ct); // Delete all data from the cache table
```

Listing 23: API of the Index object provided by the OSL

```c
// Signatures
int osl_index(osl_s* osl, osl_od* od, char* name, uint8_t type, size_t key_size,
              size_t value_size);
int osl_index_default(osl_s* osl, osl_od* od, char* name, size_t key_size,
                      size_t value_size);
int osl_index_add(osl_od* od, void* key, void* value);
int osl_index_find(osl_od* od, void* key, void* value);
int osl_index_query(osl_od* od, osl_od* stream_od, void* key_min, void* key_max);
int osl_index_remove(osl_od* od, void* key);
// Example
osl_od some_index;
osl_index_default(&osl, &some_index, "example index", sizeof(uint32_t), sizeof(uint8_t));
uint32_t key = 123;
uint8_t value = 456;
osl_index_add(&some_index, &key, &value); // Add value 456 pointed to by key 123
osl_index_find(&some_index &key, &value); // Retrieve value pointed to by key
osl_index_remove(&some_index &key);       // Remove value pointed to by key

uint32_t key_min = 100;
uint32_t key_max = 200;
osl_od target_stream; // This has been initialized with a stream object
// Retrieves all values between (and including) min and max and stores appends
// them to the provided stream.
osl_index_query(&some_index &target_stream, &key_min, &key_max);
```

Listing 24: API for registering new index types using the OSL

```
// Signatures
int16_t osl_index_register_type(
    osl_s* osl,
    int (*add_fn)(osl_od* od, void* key, void *value, size_t key_size, size_t value_size),
    int (*find_fn)(osl_od* od, void* key, void *value, size_t key_size, size_t value_size),
    int (*remove_fn)(osl_od* od, void* key, size_t key_size),
    int (*query_fn)(osl_od* od, osl_od* stream_od, void* key_min, void *key_ma, size_t key_size),
);

int16_t osl_index_set_default(osl_s* osl, uint8_t type);
```

7 Implementation

A prototype of the designed storage system was implemented in C for the RIOT operating system. Due to the time constraints of the project at hand, it was not possible to implement the entirety of the system as designed in Chapter 5 and Chapter 6. The goal was to build a proof-of-concept implementation to determine whether or not the conceived design is viable. This required all of the FTLs functionality as well as the core functionality of the OSL to be implemented. Said core functionality is comprised of:

- Creation of record log structure
- Stream and queue objects
- Iteration through streams
- Record caching
- Storage and restoration of FTL and OSL metadata

The storage system was implemented for RIOT, an OS targeted at Class 1 devices (see Section 2.4). In contrast to other embedded operating systems for the IoT, it provides a multi-threaded programming environment comparable to that of desktop OS such as Linux. RIOT and its applications are fully implemented in the C programming language (C99 compliant).

The implementation was split into two RIOT modules, the FTL module and the OSL module. A RIOT module commonly consists of a Makefile containing the module name and dependencies, a header file defining the exposed API, one or more implementation files as well as optional tests. A module is structured as shown in Listing 25.

Even though RIOT provides a programming environment similar to that of Linux, its focus on Class 1 embedded devices implies a number of tradeoffs, the most

Listing 25: Directory and file structure that makes up the RIOT FTL module.

```
o
|-- sys/
|   |-- include/
|   |   |-- storage/
|   |   |   `-- ftl.h
|   |   `-- [..]
|   |-- storage/
|   |   |-- ftl/
|   |   |   `-- ftl.c
|   |   `-- [..]
|   `-- [..]
`-- tests/
    |-- storage_ftl/
    |   |-- Makefile
    |   `-- main.c
    `-- [..]
```

prominent of which is the discouragement of dynamic memory allocation at runtime using malloc. For kernel components, the use of dynamic memory allocation is forbidden in RIOT. While this is a feature which is used in virtually every application for desktop operating systems, it is discouraged – yet possible – in RIOT[1], since it breaks real-time guarantees, increases code complexity and makes it more likely that an application fails at runtime due to memory exhaustion. Avoiding dynamic memory allocations makes it possible to reason more confidently about memory usage of an embedded application, and completely prevents the problem of memory leaks.

The remainder of this chapter will examine the development methodology in Section 7.1 and highlight the challenges encountered during the implementation in Section 7.2.

7.1 Methodology

Test-driven Development (TDD) was chosen as the approach for the implementation of the prototype. The idea behind TDD is to write executable test cases for a

[1]https://github.com/RIOT-OS/RIOT/wiki/Coding-conventions

piece of functionality before implementing it, thus formalizing its API and evaluating whether it supports the intended use cases. Once the tests are written, the functionality is implemented and iterated upon until the tests pass. Due to the formalization phase before writing the actual functionality, this approach "encourages decomposition [and] improves understanding of the underlying requirements", ultimately improving productivity and code quality [27]. In addition, it reduces the effort it takes to change functionality at a later time, since running the existing tests automatically verifies that previously made assumptions about the code are still correct (assuming that the tests are implemented correctly).

RIOT uses embUnit [56] as a testing framework, thus it was also used for writing the tests for the implemented storage system. An example test case from the FTL test suite is shown in Listing 26. It tests the different variants of the FTLs ability to write and read data to a given partition, such as writing data exceeding the page size (L4–5) and writing data smaller than the subpage size (L11–13). Note that, countrary to what the name of the framework indiciates, the storage system is not tested in the form of unit tests. Instead, the FTL and OSL are treated as black boxes and the exposed APIs functionality is tested.

When following the TDD approach, it is useful to have a fast feedback loop, allowing the developer to quickly see whether the implementation of a feature or a recent change breaks any tests. Unfortunately, the feedback loop is relatively long when having to flash the code onto the embedded device. As an example, running the FTL test suite on the MSBA2 platform takes 37.23 seconds total, 16.91 seconds to flash the test suite and 20.32 seconds to run it.

To deal with the problem of long feedback loops in embedded software development, RIOT features a "native mode", where the application is built as an x86 exectuable which can be run directly on the development machine. Using native mode, the application does not need to be flashed, and due to the much faster hardware of common desktop computers, the tests can be executed with negligible delay. It allows to run the FTL test suite in about 0.5 to 0.75 seconds, or roughly 50 times faster than on the embedded device. Taking advantage of native mode, most implementation problems were taken care of before having to flash the application to the embedded device at all when following the process of Figure 7.1.

Listing 26: An embUnit-based test case checking the functionality of writing and reading data using the FTL

```
static void test_write_read(void) {
    [variable initialization]

    ret = ftl_write(&data_partition, page_buffer, 512);
    TEST_ASSERT_EQUAL_INT(-EFBIG, ret);

    ret = ftl_write(&data_partition, page_buffer, data_length);
    TEST_ASSERT_EQUAL_INT(0, ret);
    TEST_ASSERT_EQUAL_INT(0, data_partition.last_written_subpage);

    ret = ftl_write(&data_partition, page_buffer, data_length/2);
    TEST_ASSERT_EQUAL_INT(0, ret);
    TEST_ASSERT_EQUAL_INT(1, data_partition.last_written_subpage);

    subpageheader_s header;
    ret = ftl_read(&data_partition, page_buffer, &header, subpage);
    TEST_ASSERT_EQUAL_INT(0, ret);
    TEST_ASSERT_EQUAL_INT(data_length, header.data_length);
    memset(expect_buffer, 0xAB, data_length);
    TEST_ASSERT_EQUAL_INT(0, USTRNCMP(page_buffer, expect_buffer, data_length));

    ret = ftl_read(&data_partition, page_buffer, &header, subpage+1);
    TEST_ASSERT_EQUAL_INT(0, ret);
    TEST_ASSERT_EQUAL_INT(data_length/2, header.data_length);
    memset(expect_buffer, 0xAB, data_length/2);
    TEST_ASSERT_EQUAL_INT(0, USTRNCMP(page_buffer, expect_buffer, data_length/2));
}
```

7.2 Challenges

To test the implemented storage system using RIOT's native mode, a layer to emulate the flash memory's properties had to be implemented (see subsection 7.2.1). Furthermore, because of the complexity involved when configuring the FTL, a script was written to generate the necessary configuration structures based on the flash memory parameters (see subsection 7.2.2). Finally, since RIOT did not previously provide an ECC, an implementation of the Hamming algorithm (as described in [48]) was ported and has since been merged into the RIOT kernel [35].

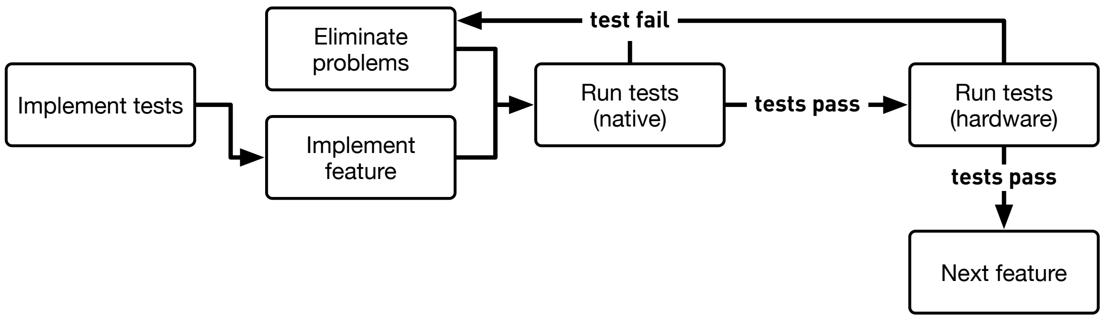


Figure 7.1: Development process taking advantage of the shortened feedback loop by using native mode, depicted as a flowchart.

7.2.1 Flash storage emulation

Using native mode to shorten the development feedback loop has one downside: since all storage interface which are conveniently available in common desktop operating systems provide storage only through several layers of abstraction, it is difficult to directly access a storage medium that presents the properties of flash memory. Since it is important that the implemented storage system is tested on such a medium, its properties need to be emulated in native mode.

To this end, the “flash_sim" RIOT module was developed, whose API is shown in Listing 27. Before usage, the module is initialized with the desired flash parameters to be emulated (L1–8), namely the page size, the block size and the total storage size. Upon initialization, the module creates a file which serves as storage location for the emulated flash medium. One can then read, write and erase data by page, as is typical for flash memory (see Section 2.5). The module only allows writing pages which have been previously erased and returns an error if a page inside a block is written non-sequentially, as exhibited by NAND flash. The ”flash_sim" module is not intended to perfectly replicate the behavior of flash memory. Instead, it aims to provide a best-effort mechanism to check if the typical properties of flash memory are being respected when executing tests or example applications in RIOT native mode.

Listing 27: API of the "flash_sim" module used to emulate flash storage characteristics in RIOT native mode

```
typedef struct flash_sim {
    uint32_t page_size;         //!< Page size to be emulated
    uint32_t block_size;        //!< Block size
    uint64_t storage_size;      //!< Total amount of storage to be made available

    uint32_t pages;             //!< Number of pages
    FILE *_fp;                  //!< File descriptor for the file simulating the flash device
} flash_sim;

int flash_sim_init(flash_sim *fs);
int flash_sim_read(const flash_sim *fs, void *buffer, uint32_t page);
int flash_sim_write(const flash_sim *fs, const void *buffer, uint32_t page);

int flash_sim_read_partial(const flash_sim *fs, void *buffer,
                           uint32_t page, uint32_t offset, uint32_t length);

int flash_sim_write_partial(const flash_sim *fs, const void *buffer,
                            uint32_t page, uint32_t offset, uint32_t length);

int flash_sim_format(flash_sim *fs);
int flash_sim_erase(const flash_sim *fs, uint32_t block);
```

7.2.2 Flash Translation Layer configuration

The fact that RIOT discourages dynamic memory allocation at runtime has led to a relatively complex setup procedure for the FTL, since the device and partition configuration structures as well as buffers must be manually created (as shown in Listing 9). This has proven to be a task prone to errors. To improve the usability of the system, a script has been developed which – given the parameters of the target flash medium – generates a basic RIOT application skeleton with the appropriate configuration.

While this makes set up of the FTL easier for a concrete flash device, it still demands fundamental knowledge of the properties of the targeted flash memory. Fortunately, the FTL configuration can be avoided completely – from the application developers perspective – if the flash device to be used is a fixed part of the targeted hardware platform. In such a case, the main FTL configuration (compare Listing 9) is stored with the code to support the specific targeted hardware, such as drivers, interrupt configuration, etc. In that case, the application developer needs to simply

Listing 28: Example usage of the FTL configuration script. It configures three partitions on the flash device: index, OTA and OSL. The script's output is a C file containing the configuration seen in Listing 9 as well as a function initializing the FTL.

```
$ ./ftl_config.py
usage: ftl_config.py [-h] --page-size PAGE_SIZE --pages-per-block
                     PAGES_PER_BLOCK --total-blocks TOTAL_BLOCKS --ecc-size
                     ECC_SIZE --number-of-subpages NUMBER_OF_SUBPAGES
                     [--partition-name PARTITION_NAME]
                     [--partition-size PARTITION_SIZE]
ftl_config.py: error: the following arguments are required: --page-size, --pages-per-block,
    --total-blocks, --ecc-size, --number-of-subpages

$ ./ftl_config.py --page-size 1024 --pages-per-block 2048 --total-blocks 512 \
                  --ecc-size 12 --number-of-subpages 4 \
                  --partition-name index --partition-size 4 \
                  --partition-name ota --partition-size 4 \
                  --partition-name osl --partition-size "remaining"
```

include the provided configuration header and can then proceed to configure the FTL partitions, as shown in Listing 29.

Listing 29: Simplification of FTL configuration process for hardware platforms which can be pre-configured

```c
#import "storage/ftl.h"

// The platform-specific FTL configuration. Exports the "msba2_ftl_device"
// variable which contains the FTL config, as well as a standard partition
// layout ("msba2_ftl_partitions").
#import "msba2/ftl-config.h"

int main(void) {
    device.partitions = msba2_ftl_partitions;
    int ret = ftl_init(&msba2_ftl_device);
    assert(ret == 0);
}
```

8 Evaluation

This chapter presents an evaluation of the design and the implementation of the storage system based on the requirements formulated in Chapter 3. First, a performance evaluation in terms of resource usage and throughput of both FTL and OSL is carried out in Section 8.1, followed by an evaluation of the presented design. Finally, the results are discussed in Section 8.3.

8.1 Performance

Unfortunately, no common standards for testing storage system performance currently exists. In a publication on the Linux B-Tree FS, the authors state that “The only realistic way to check which file system is the best match for a particular use case, is to try several file systems, and see which one works best” [57]. A plethora of benchmarking approaches exist for different types of systems such as desktop FS, Network File Systems (NFSs) and databases which are all not easily applicable to embedded systems [61].

A detailed comparative study of different benchmarks, applications and file systems was not possible in the context of this thesis. As a result, the non-functional performance requirements established in Section 4.3 were examined: ROM usage, RAM usage and write throughput.

8.1.1 Platform

The performance evaluation of the developed storage system was performed on MSBA2 hardware platform developed by the FU Berlin (shown in Figure 8.1). The MSBA2 features a LPC2387 ARM7 processor with 98 KB of RAM, 512 KB of ROM and a microSD slot. As SD card, an unbranded SD-C02G (2 GB) and a Transcend

TS1GUSD (1 GB) were used. If not otherwise specified, the measurements were performed on the 2 GB card.

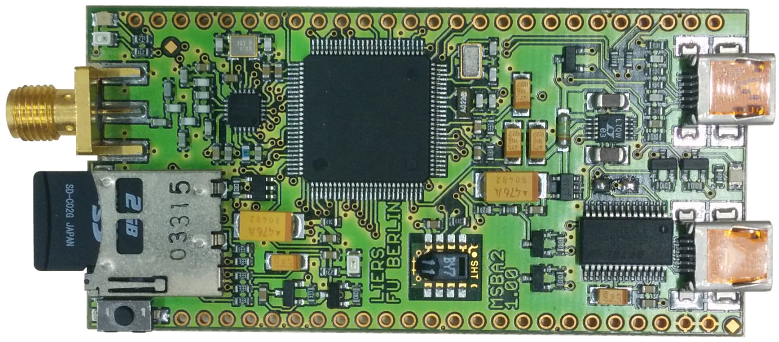


Figure 8.1: Top view of the MSBA2 platform with a 2 GB microSD card

8.1.2 Reproducibility

The reproducibility of the performance evaluation presented in this chapter is severely limited by a number of factors. To achieve high reproducibility, the different systems would have to be tested on the same physical platform, running code generated by the same compiler and being benchmarked on the same workload. Unfortunately, none of these goals could be met. This is primarily because many of the previous developments in the area of WSN storage are comparatively old, and unfortunately unmaintained. To the author's best knowledge, this applies to Matchbox, ELF, TFFS and Capsule – all implemented for TinyOS 1.x – which has not been under development since 2005. But even if all these systems were under active development, a shared hardware platform which runs TinyOS, Contiki and RIOT would still be needed, but does not currently exist. In addition, the performance evaluation was carried out using a microSD card due to the lack of a platform with RIOT support that featured native flash memory. This is not optimal since, as explained in Section 2.6, SD cards feature a dedicated microcontroller that performs many of the tasks carried out by the FTL (e.g., wear levelling). As a result, any measurements of throughput, latency, and energy consumption made using an SD card are worse compared to using raw flash memory when a separate FTL is already implemented. In summary, the comparisons to other storage systems in this chapter must be understood under the premise that they are mere comparisons of benchmarks done on different systems, partially in different decades, and that the performance of the system is expected to be higher on raw flash memory than it is using an SD card.

8.1.3 ROM usage

Read-Only Memory (ROM) size reflects the quantity of code which was written for the storage system, each function occupying a fixed number of bytes. The maximum amount of ROM used by the FTL (2.41 KiB) and the OSL (2.55 KiB) is shown in Figure 8.2, grouped into functionality for reading, writing, and other purposes, such as initialization and utility functions. In comparison, an average Class 1 device comes with about 100 KiB of ROM. The maximum is examined here because the actual ROM usage depends on which functionality is used by the application. Unused functionality will be stripped by the linker and thus does not end up in the application binary. ROM measurements were performed by inspecting an application's ELF file using the `nm` utility, and summing up the sizes of the relevant symbols.

Figure 8.2 shows that, in the current state of the implementation, the FTL and the OSL occupy a roughly equal amount of ROM. This is deceptive, however, since the FTLs functionality is fully implemented, whereas that of the OSL is not. The two object types which have already been implemented (stream and queue) account for 19% of the OSLs ROM usage, or 488 Bytes. Given that four storage objects remain to be implemented, some more complex than the existing ones, a final maximum ROM usage of roughly 5–8 KiB can be projected. Further comparing both systems shows that the FTL requires about the same portion for reading and writing data, whereas the OSL uses more than double the code size for reading. The reason for this disparity is the complexity involved when stepping through log-structured objects – including record caching (see Section 6.4) – which is entirely implemented in the OSL.

A comparison of the ROM of different WSN storage systems is shown in Figure 8.3. Given that the OSL is not fully implemented yet, it is not suprising that it currently uses less ROM than all other approaches. However, even the projected ROM is still 6 KiB below that of Capsule. All approaches except Coffee and the developed storage system exhibit a ROM usage of 16 KiB or more.

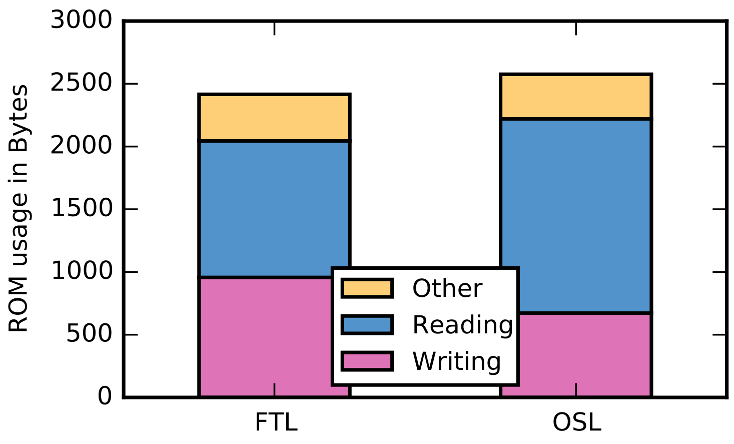


Figure 8.2: Maximum ROM usage for the implemented storage systems. Grouped by subsystem and categorized as functionality relating to reading and writing data as well as other functionality, such as initialization and utility functions

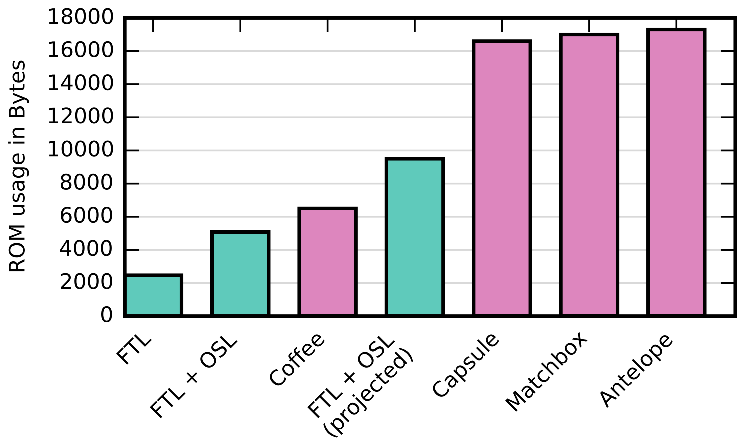


Figure 8.3: ROM usage of different storage systems for embedded WSN devices. The system developed for this thesis is highlighted in green, including the projected ROM usage of the final implementation.
Sources: Coffee [62], Capsule and Matchbox [46], Antelope [63]

8.1.4 RAM usage

RAM is the scarcer resource, of which Class 1 embedded devices commonly feature about 10 KiB. In contrast to ROM, whose maximum can be easily determined, RAM usage of the storage system depends on various hardware and application-specific parameters:

- Properties of the flash storage device (FTL)
  - Page size and subpage size
  - ECC requirements
- Number of configured partitions (FTL)
- Number of maximum open objects (OSL)
- Size of the record cache (OSL)

For the FTL, the necessary amount of static RAM usage can be calculcated as shown in Equation 8.1. No dynamic memory allocation was used. The sizes of the in-RAM elements which do not depened on the flash memory parameters are shown in Figure 8.4. The usage values were obtained by using C's `sizeof` operator on the data structures of the storage system.

An example setup employing two partitions, a subpage size of 256 Bytes, and a 6 Byte ECC would result in a RAM usage of 374 Bytes. This example is realistic when using, e.g., an SLC NAND flash chip with a page size of 1024.

$$\begin{aligned} &\text{FTL RAM usage (Bytes)} = \\ &\quad \text{FTLConfigurationBaseSize} + \text{SubpageSize} \ + \text{ECCSize} + \\ &\quad \text{ConfiguredPartitions} * \text{FTLPartitionStructSize} \end{aligned} \tag{8.1}$$

While the OSL is mostly shielded from the properties of the underlying flash memory by the FTL, the RAM usage of OSL is still dependant on the subpage size because of the needed record buffer. Based on the above scenario with a 256 Byte subpage size, as well as the standard maximum open objects of four and record cache size of six, the RAM usage of the OSL is 492 Bytes (based on Equation 8.2).

Description	Size (bytes)
FTL Configuration Base Size	56
FTL Partition Struct Size	28
OSL Configuration Base Size	36
OSL Open Object Size	32
OSL Cache Record Size	12

Figure 8.4: Portions of in-RAM which are static, i.e., they do not depend on the properties of the employed flash memory

This results in a total RAM usage of 866 Bytes for the example scenario, when both FTL and OSL are employed. Note that, while ROM usage will significantly increase as all OSL functionality is implemented, the same is not the case for RAM, since the in-memory state of the current implementation is already capable of supporting all designed objects.

$$\begin{aligned} &\text{OSL RAM usage (bytes)} = \\ &\quad \text{OSLConfigurationBaseSize} + \text{SubpageSize} + \\ &\quad \text{MaxOpenObjects} * \text{OSLOpenObjectSize} + \\ &\quad \text{RecordCacheSize} * \text{OSLCacheRecordSize} \end{aligned} \tag{8.2}$$

In Figure 8.5 the RAM usage of the FTL and OSL – in two different configurations – is compared to the systems examined in Chapter 3. Since Matchbox, TFFS, Coffee and Antelope were all built for and evaluated on NOR flash, the first configuration (#1) uses a 64 Byte subpage size which is easily achieved on NOR flash. The second configuration (#2) uses a 256 Byte subpage size, which is realistic when using the larger SLC NAND flash.

As Figure 8.5 shows, the developed storage system is about 40% more RAM efficient than Capsule while providing a comparable feature set. Antelope uses almost four times the memory of the OSL #2, but also provides very different features, which makes a direct comparison less significant. When compared to early systems, the developed storage system (#1) outperforms Matchbox by about 45% but performs 37% worse than TFFS. In both cases, however, it provides a more sophisticated feature set. Finally, Coffee appears to be the most memory efficient storage system for the functionality it provides. FTL #1 is on par in terms of memory requirements, provides a less sophisticated abstraction. Note that ELF and TFFS are missing from this comparison because a general figure of their memory usage is impossible to determine. This is because RAM usage in ELF depends on the amount of files in use as well as their size, and RAM usage in TFFS depends on the size of the addressed flash memory.

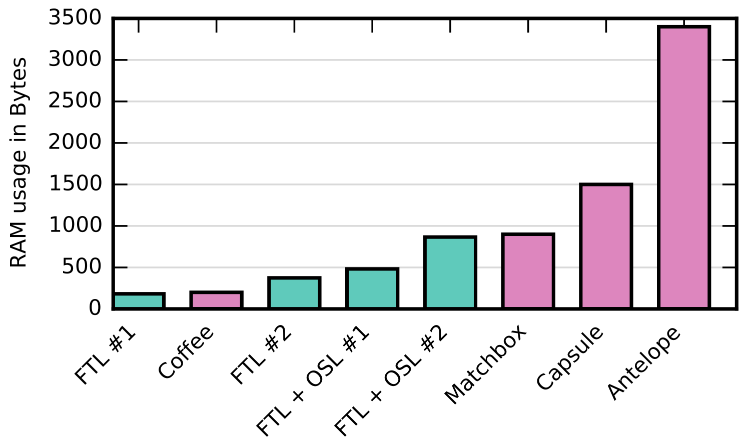


Figure 8.5: RAM usage of different storage systems for embedded WSN devices. The system developed for this thesis is highlighted in green. It is examined in two configurations, #1 using a 64 Byte page size and #2 using a 256 Byte page size
Sources: Coffee [62], Capsule and Matchbox [46], Antelope [63]

In summary, given that the FTL can be used without the OSL, its small memory footprint allows it to be employed even on the lower-end spectrum of Class 1 devices. As far as the OSL is concerned, Coffee is the only system developed thus far which outperforms it by a relevant margin. When considering that Coffee is limited in its applicability, both in terms of flash type and size, the additional memory the OSL requires appears to be an acceptable trade-off.

8.1.5 FTL throughput

Figure 8.6 shows the measured throughput achieved by the FTL implementation on two different microSD cards. The “bare” measurement shows maximum throughput which is achievable using the existing driver, which was not developed as part of this thesis. For every benchmark, the storage medium was first formatted and then the measurements were performed as shown in Listing 30. Every benchmark was repeated ten times (outer loop), and every operation was repeated 10,000 times (inner loop). For every repetition, the time taken was measured. The amount of written or read data was derived from the number of operations and the subpage size, that is, 10,000 subpage writes with a subpage size of 512 byte equal five MiB of data written. This amount was then divided by the time taken to obtain the throughput figure. For all measurements, the standard deviation was always below 0.1% and is thus not shown in the graphs.

Listing 30: Benchmark code used to evaluate the throughput of the FTL and OSL

```c
#define REPETITIONS 10
#define ITERATIONS 10000
timex_t then; // Timex is RIOT’s timer subsystem
timex_t now;
timex_t elapsed;
for(int i=0; i<REPETITIONS; i++) {
    xtimer_now_timex(&then);
    for(int p=0; p<ITERATIONS; p++) {
        ret = [operation to be tested];
        myassert(ret == 0);
        page++;
    }
    xtimer_now_timex(&now);
    elapsed = timex_sub(now, then);
    timex_to_str(elapsed, sprint_buffer);
    printf("%s, \n", sprint_buffer);
}
```

When observing the measured write speeds, it seems like the different modes of operation (e.g., ECC calculation) do not impact throughput at all. This is not possible, however, since header management and ECC calculation do take time to complete. As a result, the assumption is that write throughput on the available SD cards is too slow, such that additional FTL operations simply fill up cycles the CPU would otherwise idle waiting for the previous write operation to complete.

The same does not apply to read throughput. Since the bare read performance is much better compared to the write performance, the overhead of evaluating storage headers and ECC calculation can be clearly seen for both SD cards. When examining the higher-performing SD card, “w/o ECC” mode incurs a performance penalty of about 30% and adding ECC calculation results in a total of 57% performance reduction in comparison to raw mode. Furthermore, when comparing the read throughput of both SD cards, it is notable that, why bare and raw speeds differ between both cards, “w/o ECC” and ECC mode achieve very similar results. This indicates that about 1 MiB/s for “w/o ECC” mode and about 650 KiB/s is the upper throughput bound on the evaluated platform.

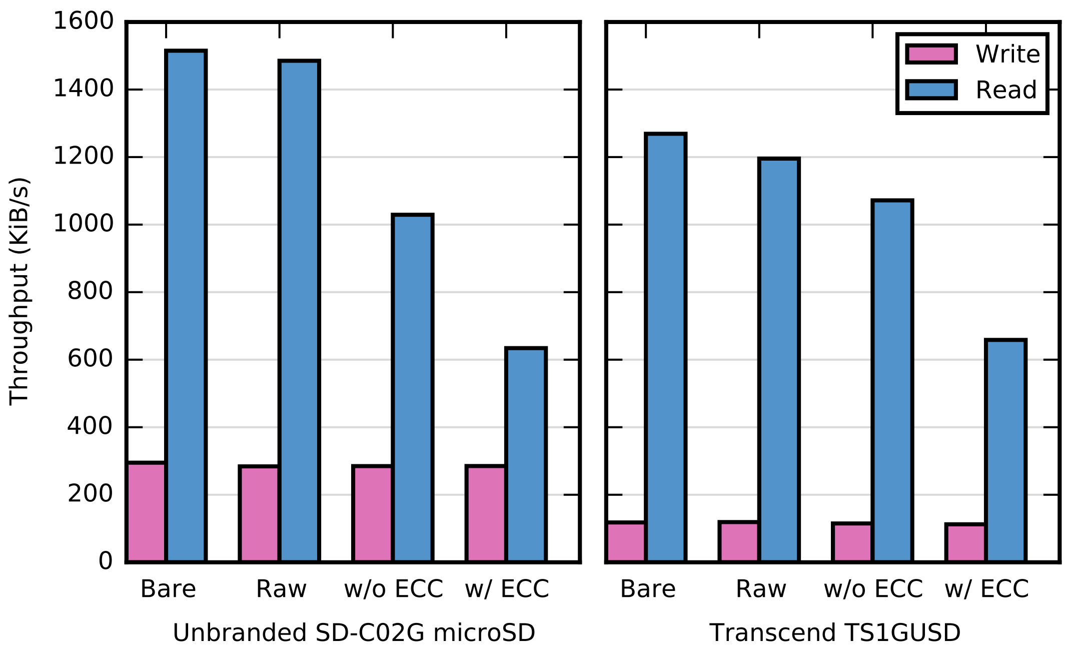


Figure 8.6: Throughput measurements of the FTL, using a subpage size of 512 Bytes. The "Bare" columns indicate the maximum achievable throughput given the current driver. The other columns correspond to the modes of reading and writing data provided by the FTL (see Chapter 5)

8.1.6 OSL throughput

The read speed measurements for front-to-back iteration using the OSLs are shown in Figure 8.7. They compare throughput of streams with different numbers of elements, both with cache disabled and enabled. The throughput measurement is made based on the element number, because when iterating through a stream from the front to the back it exhibits $\mathcal{O}(n^2)$ read complexity due to the backward pointing log structure (compare Section 6.4). The benchmark was performed the same way as for the FTL.

In the results, it is clearly visible that the OSL does not provide acceptable read performance when caching is disabled, where it already falls below the 2 KiB/s mark with a stream of just 500 elements. While enabling caching (as described in Section 6.4) alleviates the problem, a quadratic decrease in front-to-back read throughput is still discernible, with a 4000 element stream reaching just above 20 KiB/s.

Write throughput does not suffer from the same problem, since appending an element to the log is a $\mathcal{O}(1)$ operation. As a result, the mean write speed of the OSL is about 220 KiB/s regardless of the number of elements in the object. This is circa 22% slower than the ECC write speed achieved by the FTL.

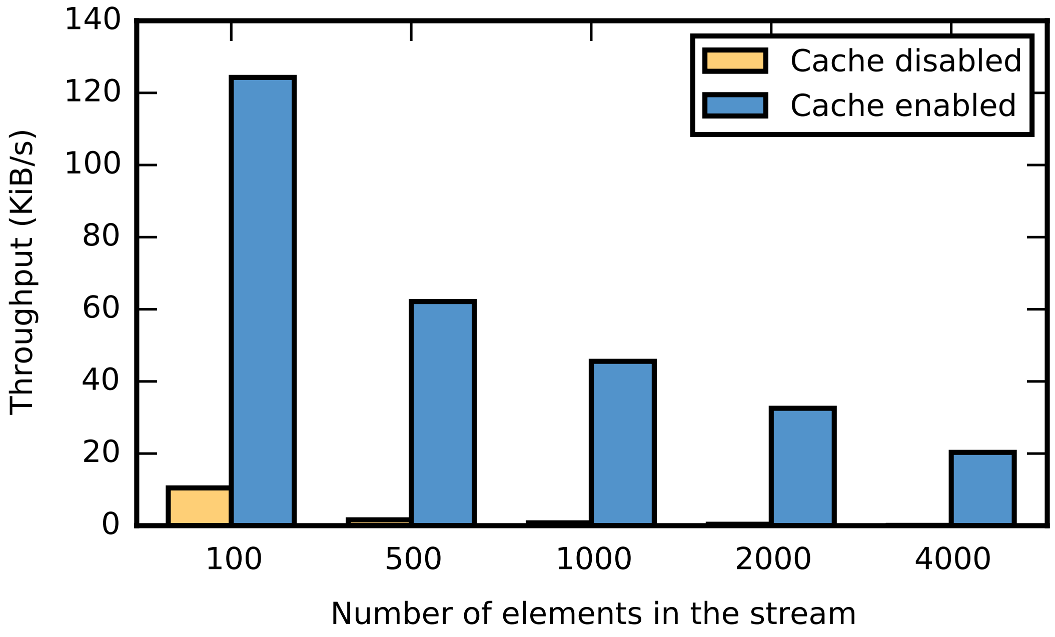


Figure 8.7: Measured OSL read throughput using a subpage size of 512 Bytes and a record cache size of six elements

8.2 Design

In contrast to the performance, the design and its characteristics are less tangible and therefore harder to evaluate. As such, this section attempts to contrast the capabilities of the designed system with the requirements gathered in Section 4.2 and Section 4.3.

When considering the functional requirements that were established at the beginning of this thesis (see Section 4.2), the designed systems meets all of these except for the support of secure storage, which was omitted due to time constraints. The design and implementation provide a storage system built to take into account the properties of all common kinds of flash storage. Partitions can accommodate OTA update binaries, and local processing of arbitrarily large data is provided by the iteration API of the OSL. And while the OSL does not currently contain concrete indexing algorithms, it provides an API for registering custom implementations.

In terms of the non-functional requirements (see Section 4.3), the performance has already been extensively evaluated, and the remaining requirement categories are examined in the following sections.

8.2.1 Extensibility and Reusability

The FTL and OSL were designed to be applicable in the widest possible range of scenarios in the context of IoT and WSN applications. To what degree this was achieved is difficult to evaluate without an extensive study of relevant applications. It is possible, however, to examine the features of the designed storage system which aim to further its applicability.

In contrast to most previous approaches to storage on embedded systems (compare Figure 3.7), the FTL and OSL support both NOR and NAND flash memory, which is both cheaper and more energy efficient than NOR flash (compare Section 2.5). Furthermore, the system supports platforms with multiple flash media, including NOR and NAND flash at the same time. This makes the designed system applicable to a wider range of platforms than previous systems.

For scenarios where RAM is particularly scarce, the FTL can be used on its own to provide resource efficient storage support, although limited in its functionality. This is not only useful for OTA updates, where a dedicated partition can be used to store updated firmware. It can also be used used to provide a simple and resource efficient means of storage, e.g., for configuration data (as shown in Listing 31).

Note that it is also possible to use the FTL in the way shown in the listing while also using the OSL to store sensor data at the same time.

Listing 31: Example of using the FTL without the OSL to store configuration data

```c
// Setup
struct config {
    [...]
};
struct config example;

// Storing configuration
int ret = ftl_write_ecc(data_partition, &example, sizeof(struct config));

// Retrieving latest written configuration
int ret = ftl_read(data_partition, &example, NULL, data_partition.next_page - 1);
```

Another option to reduce RAM usage is to sacrifice part of the available flash storage to decrease buffer sizes, as explained in Section 5.1. The OSL was built to be easily extensible using different ECC algorithms and index datastructures, such that requirements that are not covered by its base featureset may be added without the need to modify its core functionality (see Section 6.12).

To further improve the reusability of the system, the Cache Table object designed for ICN data should be made more generally applicable. Currently it is based on the assumption that the data stored in it is cache data, and as such does not preserve them when a block becomes full and must be deleted.

Finally, the system's SD card support should be improved. Their internal controller already handles many of the special properties of flash storage, such as its ECC requirements or bad block detection. In order to prevent resources from being wasted, the FTL should provide an SD card mode where such functionality is disabled.

8.2.2 Robustness and Reliability

In terms of robustness, the primary drawback of the designed system is its current inability to automatically create checkpoints of the metadata stored in RAM. To resolve this, one or more strategies for automatic metadata storage should be devised. The default strategy should be a cautious one, possibly sacrificing storage space in order to persist checkpoints more frequently. To prevent checkpoints from being written even when no changes have ocurred, the FTL and OSL should separately track whether their metadata has to be persisted.

Beyond the aforementioned issue, the prototype implementation functions reliably under load tests such as the ones in the performance evaluation, the written test suites and demo applications. Another aspect relevant to robustness is how the system reacts to a crash or power loss when an operation is in progress.

- When a write operation of user data gets aborted, the target subpage will be in an inconsistent state. If the subpage was written with ECC enabled – which is the default when using the OSL – then the inconsistency will be detected by the read operation and an error is returned. If an ECC is not used, then garbage may be returned if the subpage header was written correctly.
- If no checkpoint has been written since the last user data, the RAM metadata is going to be in a more current state than the metadata on flash. If a crash occurs in this situations, the data written since the last checkpoint is lost, but could be restored by analysing every subpage written since the last checkpoint. The same can be done if a writing metadata fails, which should always be detectable since ECC is enabled by default.

8.2.3 Usability

“Usability is a multi-faceted set of trade-offs as opposed to a singular, objective and attainable end goal” [22]. The biggest such trade-off in the design of the storage system’s API is the way the FTL is initially configured. For this task, a rather complex configuration structure must be created, currently containing twelve fields. The partition configuration is created in a similar fashion. The developer must also make sure that, e.g., the provided subpage buffer has the same size as the configured subpage size.

The need for this complex initialization procedure is caused by the best practice on embedded systems to not allocate memory dynamically at runtime, which prevents the system from automatically creating buffers of necessary size. While this approach makes reasoning about memory usage of the resulting application much easier and prevents memory leaks in the storage system, it also complicates initial setup significantly. The configuration tool presented in subsection 7.2.2 tries to alleviate this problem. Given that this configuration is hardware specific, it is fortunately possible to omit this complexity for common hardware supported by the RIOT OS (see subsection 7.2.2). For SD cards, it is additionally possible to determine their size due to their dedicated MCU. In that case, the FTL could automatically adapt to the size of the SD card employed.

When assuming that the FTL configuration is provided as part of the RIOT hardware configuration, a simple application storing data retrieved from sensor can be implemented using the OSL as seen in Listing 32, providing high-level data storage support with low coginitive overhead required.

Listing 32: Example of a small application using the OSL to store sensor data in a stream object.

```c
#include "storage/ftl.h"
#include "storage/osl.h"

// Hardware specific FTL configuration structure
// Provides the "device" variable as well as data_partition.
#include "ftl-configuration.h"

osl_s osl;

int main(void)
{
    // This depends on the hardware and is not
    // provided by the storage system
    initialize_storage_driver();

    ftl_init(&device);
    osl_init(&osl, &device, &data_partition);

    osl_od stream;
    osl_stream(&osl, &stream, "sensor_data", sizeof(uint32_t));

    while(1) {
        uint32_t datum = get_sensor_value();
        osl_stream_append(&stream, &datum);
        // Wait until next sensor datum should be retrieved.
    }
}
```

The API is designed to be uniform in terms of naming, and provide only a single option for achieving a certain task. Its consistent use of returned error codes improves the ability to pinpoint problems quickly. One big issue regarding naming is the chosen designator “index partition” for the partition which stores the system’s metadata, while its name suggests that it stores data somehow related to the index object. Since the confusion this may cause is foreseeable, the naming of this partition will be changed to “metadata partition” instead.

8.3 Discussion

Based on the proof of concept implementation, the evaluation suggests that the designed storage system meets the requirements formulated in Chapter 4. In terms of resource usage, it currently outperforms all previous storage systems except for Coffee. In terms of throughput, the front-to-back read performance is exhibiting $\mathcal{O}(n^2)$ complexity, where n are the number of records, and should thus be improved. Back-to-front iteration performance is $\mathcal{O}(n)$ instead, but a throughput measurement could not be performed due to time constraints.

The robustness and usability of the system must be further evaluated. For the former, existing methods for testing the recoverability of storage systems should be researched. The latter requires peer review as well as audits of applications which use the storage systems, such that it can be improved on their basis.

9 Conclusion

The field of storage systems for constrained WSN devices has not seen many new developments in recent years. In addition, many of the early storage systems were built for TinyOS 1.x, which is no longer under development. Given the steady increase in popularity of the IoT and the resulting new challenges and requirements – such as in Information Centric Networking – a re-evaluation of previous concepts and their adaption to the current environment was necessary. This thesis provides such an adaption.

9.1 Key Achievements

Literature Review A thorough review of existing embedded storage systems provided the foundation for the requirements analysis and the resulting design decisions. In summary, most previous systems were designed for WSN as well as outdated platforms and operating systems, but still provided valuable insights into the main hurdles identified by previous researchers.

Requirements Analysis An examination of common IoT and WSN use cases provided the basis for a list of requirements for the storage system. The identified key requirement was the native storage of common data structures such as streams of records and ICN cache data, as opposed to unstructured storage of byte streams as provided by traditional file systems. Additionally, the support for all common types of flash was identified as a strong requirement.

Design Based on the foregoing literature review and requirements analysis, a storage system composed of a Flash Translation Layer (FTL) and an Object Storage Layer (OSL) was designed, where the former acts as a lower-level managed interface to flash memory and the latter provides a storage interface for structured data that fully abstracts from the properties of flash memory. The

FTL was explicitly designed to be usable on its own, and thus on systems that are especially resource constrained.

Implementation A proof of concept implementation for the RIOT operating system was created to evaluate the feasiblity of the designed system.

Evaluation The implementation was evaluated based on the requirements analysis. The results suggest that the implemented system is more resource efficient than systems with comparable functionality. However, sequential read throughput – when iterating from start to end – degrades quadratically as the number of records increases.

9.2 Future work

While this thesis suggests that providing a reusable, yet resource efficient storage system for the IoT is feasible, it still leaves plenty of room for future work.

The presented evaluation had several shortcomings. Comparability was hindered by the lack of a common platform able to host the different storage systems being compared. One of the main causes for this was that previous storage systems are mostly outdated. Future work should thus include porting the designed storage system to platforms that host comparable systems like Coffee, ELF or Capsule and employ native flash memory as opposed to SD cards. In addition, further evaluation of the performance characteristics of the system must be performed, such as back-to-front iteration of objects, which was omitted due to time constraints. The same applies to measurements of the systems energy consumption. Also, existing storage system benchmarks should be examined to determine whether they can be applied to the system at hand.

Furthermore, several of the designed OSL features remain to be implemented. These are the Stack, Index, and ICN Cache data structures as well as the garbage collection mechanics. Since additional performance optimizations were designed, but have not yet been implemented, future work should include their implementation as well as performance evaluation. More specifically, these optimizations are object defragmentation and using bloom filters to track existing objects.

The integration of the storage system into the RIOT kernel, along with the associated extensive peer review, will likely result in both usability and technical improvements to the system. In the long run, the integration will also provide insights into how people are using the system and which problems arise in its usage.

The availability of mass storage on IoT and WSN platforms is still very limited. Research in this area has stagnated at the point where feasible flash memories for embedded systems had sizes ranging from hundreds of Kilobytes to several Megabytes. This threshold has since increased by three to four orders of magnitude. By advancing research in this area, the author hopes that interest in flash storage for IoT devices can be raised.

Glossary

API Application Programming Interface. 5, 23, 26–29, 37, 40, 50–56, 65, 66, 70–77, 79, 81, 82, 94, 96, 98, 104, 107, 108

AQL Antelope Query Language. 14, 27, 30, 107

CoAP Constrained Application Protocol. 5

CRC Cyclic Redundancy Check. 16

DBMS Database Management System. 14, 24, 25, 38, 39

DTN Delay Tolerant Netork. 33

ECC Error-Correcting Code. 9, 10, 37, 44, 45, 50, 52–54, 80, 88, 91, 93, 95, 96, 107

EEPROM Electrically Erasable Programmable Read-Only Memory. 7, 17, 21

ELF Efficient Log-Structured Flash File System. 13

FFT Fast Fourier Transform. 24

FS File System. 10, 13, 16, 23, 25–27, 84, 107

FTL Flash Translation Layer. 16, 39–41, 43–46, 48–57, 64, 66, 69–72, 77–80, 82–86, 88–97, 99, 100, 105–108

HDD Hard Disk Drive. 1, 7, 10, 15

ICN Information Centric Networking. 2, 31, 35, 61, 64, 95, 99

IETF Internet Engineering Task Force. 5

IOPS Input/Output Operations per Second. 1

IoT Internet of Things. 1–6, 11, 12, 16, 29, 31, 34, 37, 39, 54, 56, 61, 66, 67, 70, 71, 77, 94, 99–101

ITU International Telecommunication Union. 12

LLN Low-power Lossy Network. 5

MCU Microcontroller Unit. 11, 97

MLC Multi-level Cell. 8, 9, 22, 40

MMU Memory Management Unit. 32

NFS Network File System. 84

OD Object Descriptor. 71

OS Operating System. 2, 5, 6, 15, 23, 24, 32, 36, 37, 63, 70, 77, 97

OSL Object Storage Layer. 39, 52, 54, 56–59, 61–77, 79, 83, 84, 86, 88–97, 99, 100, 105–108

OTA Over-the-air. 13, 34, 35, 41, 43, 44, 83, 94, 108

RAM Random-Access Memory. 48, 56, 57, 65, 66, 68, 88–90, 94–96, 106

ROM Read-Only Memory. 86–89, 106

SLC Single-Level Cell. 8, 9, 22

SQL Structured Query Language. 24, 27, 28, 38

SSD Solid-State Drive. 10, 16

TDD Test-driven Development. 78, 79

TFFS Transactional Flash Filesystem. 13, 16, 17, 26

WSN Wireless Sensor Network. 1–6, 12–16, 23, 25, 29, 31–34, 37, 56, 71, 85–87, 90, 94, 99, 101, 106

List of Figures

List of Listings

Bibliography

[1] "IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)," *IEEE Std 802.15.4-2011 (Revision of IEEE Std 802.15.4-2006)*, pp. 1–314, sep 2011. doi: 10.1109/IEEESTD.2011.6012487

[2] M. Abraham and Micron Technology Inc., "NAND Flash Architecture and Specification Trends," in *Proceedings of the 2012 Flash Memory Summit*, Santa Clara, CA, 2012. URL: http://bit.ly/1T6dibz

[3] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "A survey on sensor networks," *IEEE Communications Magazine*, vol. 40, no. 8, pp. 102–105, 2002. doi: 10.1109/MCOM.2002.1024422

[4] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2347–2376, jan 2015. doi: 10.1109/COMST.2015.2444095. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7123563

[5] Aleph One Ltd., "Yaffs2 specification and development nodes," 2005. URL: http://www.yaffs.net/yaffs-2-specification. Accessed on: 2016-03-01

[6] Atmel Inc., "Specification of the AT45DB041D serial-interface Flash memory," 2010.

[7] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *Proc. of the 32nd IEEE INFOCOM. Poster*. Piscataway, NJ, USA: IEEE Press, 2013.

[8] E. Baccelli, C. Mehlis, O. Hahm, T. C. Schmidt, and M. Wählisch, "Information Centric Networking in the IoT: Experiments with NDN in the Wild," in *Proc.*

of 1st ACM Conf. on Information-Centric Networking (ICN-2014). New York: ACM, 2014, pp. 77–86. doi: 10.1145/2660129.2660144

[9] I. Bagci, M. Pourmirza, S. Raza, U. Roedig, and T. Voigt, “Codo: confidential data storage for wireless sensor networks,” in *Proceedings of the 2012 IEEE 9th International Conference on Mobile Ad-Hoc and Sensor Systems (MASS 2012) - Supplement*. Las Vegas, NV: IEEE, 2012, pp. 1–6. doi: 10.1109/MASS.2012.6708508

[10] A. Bakshi, V. K. Prasanna, J. Reich, and D. Larner, “The Abstract Task Graph: A Methodology for Architecture-independent Programming of Networked Sensor Systems,” in *Proceedings of the 2005 Workshop on End-to-end, Sense-and-respond Systems, Applications and Services*, ser. EESR ’05. Berkeley, CA, USA: USENIX Association, 2005, pp. 19–24.

[11] B. Bloom, “Space/Time Trade-Offs in Hash Coding with Allowable Errors,” *ACM Communications*, vol. 13, no. 7, 1970. doi: 10.1145/1061453.1061454

[12] Bluetooth SIG Inc., “Bluetooth Core Specification 4.2,” Tech. Rep., 2015. URL: https://www.bluetooth.com/specifications/adopted-specifications

[13] C. Bormann, M. Ersue, and A. Keranen, “Terminology for Constrained-Node Networks,” RFC 7228 (Informational), may 2014. URL: http://www.ietf.org/rfc/rfc7228.txt

[14] S. Brown and C. Sreenan, *Software Updating in Wireless Sensor Networks: A Survey and Lacunae*, 2013, vol. 2, no. 4. doi: 10.3390/jsan2040717

[15] J. Case, R. Mundy, D. Partain, and B. Stewart, “Introduction and Applicability Statements for Internet-Standard Management Framework,” RFC 3410 (Informational), dec 2002. URL: http://www.ietf.org/rfc/rfc3410.txt

[16] S. Cho, C. Park, Y. Won, S. Kang, J. Cha, S. Yoon, and J. Choi, “Design Tradeoffs of SSDs: From Energy Consumption&Rsquo;s Perspective,” *Trans. Storage*, vol. 11, no. 2, pp. 8:1—-8:24, mar 2015. doi: 10.1145/2644818

[17] E. F. Codd, “A relational model of data for large shared data banks,” *Commun. ACM*, vol. 13, no. 6, 1970. doi: 10.1145/362384.362685

[18] J. Cooke, “The Inconvenient Truths of NAND Flash Memory,” in *Proceedings from the 2007 Flash Memory Summit*, no. August, Santa Clara, CA, 2007, pp. 1–32.

[19] D. Culler, J. Hill, M. Horton, K. Pister, R. Szewczyk, and A. Woo, “Mica: The commercialization of microsensor motes,” *Sensors Magazine*, vol. 19, no. 4, pp. 40–48, 2002.

[20] D. Culler, D. Ganesan, O. Gnawali, and D. Gay, “TinyOS on SourceForge,” 2005. URL: https://sourceforge.net/projects/tinyos. Accessed on: 2016-03-14

[21] H. Dai, M. Neufeld, and R. Han, “ELF: An Efficient Log-structured Flash File System for Micro Sensor Nodes,” in *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, ser. SenSys ’04. New York, NY, USA: ACM, 2004, pp. 176–187. doi: 10.1145/1031495.1031516

[22] J. Daughtry, U. Farooq, B. Myers, and J. Stylos, “API usability,” *ACM SIGSOFT Software Engineering Notes*, vol. 34, no. 4, p. 27, 2009. doi: 10.1145/1543405.1543429

[23] C. de Souza, D. Redmiles, L.-t. Cheng, D. Millen, and J. Patterson, “Sometimes You Need to See Through Walls — A Field Study of Application Programming Interfaces,” *Proceedings of the ACM conference on Computer supported cooperative work (CSCW)*, pp. 63–71, 2004. doi: 10.1145/1031607.1031620

[24] Y. Diao, D. Ganesan, G. Mathur, and P. Shenoy, “Rethinking Data Management for Storage-centric Sensor Networks,” in *Proceedings of the 3rd biennial conference on Innovative Data Systems Research*. CIDR, 2007. URL: http://www-db.cs.wisc.edu/cidr/cidr2007/index.html

[25] A. Dunkels, B. Gronvall, and T. Voigt, “Contiki - a lightweight and flexible operating system for tiny networked sensors,” in *29th Annual IEEE International Conference on Local Computer Networks*. IEEE (Comput. Soc.), 2004, pp. 455–462. doi: 10.1109/LCN.2004.38

[26] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, “Network Configuration Protocol (NETCONF),” RFC 6241 (Proposed Standard), jun 2011. URL: http://www.ietf.org/rfc/rfc6241.txt

[27] H. Erdogmus, M. Morisio, and M. Torchiano, “On the effectiveness of the test-first approach to programming,” *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 226–237, 2005. doi: 10.1109/TSE.2005.37

[28] E. Gal and S. Toledo, “A Transactional Flash File System for Microcontrollers,” *(USENIX 2005): Proceedings of the annual conference on USENIX Annual Technical Conference*, p. 7, 2005.

[29] D. Gay, “Matchbox: A simple filing system for motes,” pp. 1–13, 2003. URL: http://webs.cs.berkeley.edu/tos/dist-1.1.0/snapshot-1.1.3Dec2003cvs/doc/matchbox.pdf

[30] D. Gay, “Design of Matchbox , the simple filing system for motes,” pp. 1–4, 2003. URL: http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/matchbox-design.pdf

[31] G. Goodson and R. Iyer, “Design Tradeoffs in a Flash Translation Layer,” *Proceedings of Workshop on the Use of Emerging Storage and Memory Technologies HPCA WEST 2010*, 2010. URL: http://drona.csa.iisc.ernet.in/ gopi/west10/goodson.pdf

[32] J. Hui and P. Thubert, “Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks,” RFC 6282 (Proposed Standard), sep 2011. URL: http://www.ietf.org/rfc/rfc6282.txt

[33] International Telecommunication Union, “Applications of Wireless Sensor Networks in Next Generation Networks,” Telecommunication Standardization Sector of ITU, Tech. Rep. February, 2014.

[34] A. Jagmohan, M. Franceschini, and L. Lastras, “Write amplification reduction in NAND Flash through multi-write coding,” in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, may 2010, pp. 1–6. doi: 10.1109/MSST.2010.5496985

[35] L. Jenß, “Add 256 byte block hamming code implementation,” 2015. URL: https://github.com/RIOT-OS/RIOT/pull/4229. Accessed on: 2016-03-03

[36] D. Kang, D. Jung, J.-U. Kang, and J.-S. Kim, “μ-Tree : An Ordered Index Structure for NAND Flash Memory,” *7th ACM & IEEE Conference on*

Embedded Software (EMSOFT '07), pp. 144–153, 2007. doi: 10.1145/1289927.1289953

[37] J. Kelsey, "SHA-160: A Truncation Mode for SHA256 (and most other hashes)," 2005. URL: http://csrc.nist.gov/groups/ST/hash/documents/Kelsey_Truncation.pdf. Accessed on: 2016-03-16

[38] G. Krishna, "Antelope (Database Management System) - Contiki," 2014. URL: https://bit.ly/1Lnhaml. Accessed on: 2016-03-14

[39] P. Levis, "Experiences from a Decade of TinyOS Development," *10th USENIX conference on Operating Systems Design and Implementation*, pp. 207 – 220, 2012.

[40] H. Li, D. Liang, L. Xie, G. Zhang, and K. Ramamritham, "Flash-Optimized Temporal Indexing for Time-Series Data Storage on Sensor Platforms," *ACM Transactions on Sensor Networks*, vol. 10, no. 4, pp. 1–30, jun 2014. doi: 10.1145/2526687

[41] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: An Acquisitional Query Processing System for Sensor Networks," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, mar 2005. doi: 10.1145/1061318.1061322

[42] G. Mainland, G. Morrisett, and M. Welsh, "Flask: Staged Functional Programming for Sensor Networks," in *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '08. New York, NY, USA: ACM, 2008, pp. 335–346. doi: 10.1145/1411204.1411251

[43] C. Manning, "YAFFS (yet another Flash File System)," 2001. URL: https://bit.ly/1SObK5o

[44] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy, "Ultra-low power data storage for sensor networks," in *Proceedings of the fifth international conference on Information processing in sensor networks - IPSN '06*. New York, New York, USA: ACM Press, apr 2006, p. 374. doi: 10.1145/1127777.1127833

[45] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy, "Capsule: an energy-optimized object storage system for memory-constrained sensor devices," *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, p. 195, 2006. doi: 10.1145/1182807.1182827

[46] G. Mathur, P. Desnoyers, P. Chukiu, D. Ganesan, and P. Shenoy, "Ultra-low power data storage for sensor networks," *ACM Transactions on Sensor Networks*, vol. 5, no. 4, pp. 1–34, nov 2009. doi: 10.1145/1614379.1614385

[47] J. McCulloch, P. McCarthy, S. M. Guru, W. Peng, D. Hugo, and A. Terhorst, "Wireless Sensor Network Deployment for Water Use Efficiency in Irrigation," in *Proceedings of the Workshop on Real-world Wireless Sensor Networks*, ser. REALWSN '08. New York, NY, USA: ACM, 2008, pp. 46–50. doi: 10.1145/1435473.1435487

[48] Micron Technology Inc., "TN-29-08 : Hamming Codes for NAND Flash Memory Devices Technical Note Hamming Codes for NAND Flash Memory Devices," *Micron*, pp. 1–7, 2005.

[49] V. Mohan, T. Bunker, L. M. Grupp, S. Gurumurthi, M. R. Stan, and S. Swanson, "Modeling Power Consumption of NAND Flash Memories Using FlashPower," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 7, pp. 1031–1044, 2013. doi: 10.1109/TCAD.2013.2249557

[50] L. Mottola, "Programming Storage-centric Sensor Networks with Squirrel," *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks - IPSN '10*, p. 1, 2010. doi: 10.1145/1791212.1791214

[51] S. Nath and A. Kansal, "FlashDB: Dynamic Self-tuning Database for NAND Flash," in *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, ser. IPSN '07. New York, NY, USA: ACM, 2007, pp. 410–419. doi: 10.1145/1236360.1236412

[52] S. Park, Y. Kim, B. Urgaonkar, J. Lee, and E. Seo, "A comprehensive study of energy efficiency and performance of flash-based SSD," *Journal of Systems*

Architecture, vol. 57, no. 4, pp. 354–365, 2011. doi: 10.1016/j.sysarc.2011.01.005

[53] D. C. Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, "TinyOS: An operating system for sensor networks," *Ambient Intelligence*, 2004. doi: 10.1007/3-540-27139-2_7

[54] G. J. Pottie and W. J. Kaiser, "Wireless integrated network sensors," *Communications of the ACM*, vol. 43, no. 5, pp. 51–58, may 2000. doi: 10.1145/332833.332838

[55] K. S. Prabh and T. F. Abdelzaher, "Energy-conserving Data Cache Placement in Sensor Networks," *ACM Trans. Sen. Netw.*, vol. 1, no. 2, pp. 178–203, nov 2005. doi: 10.1145/1105688.1105690

[56] T. Punkka, "embUnit - Embedded Unit Testing Framework," 2006. URL: http://embunit.sourceforge.net/embunit/. Accessed on: 2016-03-15

[57] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-Tree Filesystem," *ACM Transactions on Storage*, vol. 9, no. 3, pp. 1–32, 2013. doi: 10.1145/2501620.2501623

[58] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-structured File System," in *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '91. New York, NY, USA: ACM, 1991, pp. 1–15. doi: 10.1145/121132.121137

[59] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," RFC 7252 (Proposed Standard), jun 2014. URL: http://www.ietf.org/rfc/rfc7252.txt

[60] T. Small and Z. J. Haas, "Resource and performance tradeoffs in delay-tolerant wireless networks," *Proceeding of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking - WDTN '05*, pp. 260–267, 2005. doi: 10.1145/1080139.1080144

[61] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright, "A nine year study of file system and storage benchmarking," *ACM Transactions on Storage (TOS)*, vol. 4, no. 2, pp. 1–56, 2008. doi: 10.1145/1367829.1367831

[62] N. Tsiftes, A. Dunkels, Z. H. Z. He, and T. Voigt, "Enabling large-scale storage in sensor networks with the Coffee file system," *2009 International Conference on Information Processing in Sensor Networks*, no. 1, 2009. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5211918

[63] N. Tsiftes and A. Dunkels, "A Database in Every Sensor," *Technology*, p. 316, 2011. doi: 10.1145/2070942.2070974

[64] Q. W. Q. Wang, Y. Z. Y. Zhu, and L. C. L. Cheng, "Reprogramming wireless sensor networks: challenges and approaches," *IEEE Network*, vol. 20, no. June, pp. 48–55, 2006. doi: 10.1109/MNET.2006.1637932

[65] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. P. Vasseur, and R. Alexander, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks," RFC 6550 (Proposed Standard), mar 2012. URL: http://www.ietf.org/rfc/rfc6550.txt

[66] D. Woodhouse and Red Hat Inc., "JFFS : The Journalling Flash File System," 2001. URL: http://www.sourceware.org/jffs2/jffs2-html/

[67] G. Xylomenos, C. N. Ververidis, V. A. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K. V. Katsaros, and G. C. Polyzos, "A Survey of information-centric networking research," *IEEE Communications Surveys and Tutorials*, vol. 16, no. 2, pp. 1024–1049, 2014. doi: 10.1109/SURV.2013.070813.00063

[68] M.-C. Yang, Y.-M. Chang, C.-W. Tsao, P.-C. Huang, Y.-H. Chang, and T.-W. Kuo, "Garbage collection and wear leveling for flash memory: Past and future," in *Smart Computing (SMARTCOMP), 2014 International Conference on*, nov 2014, pp. 66–73. doi: 10.1109/SMARTCOMP.2014.7043841

[69] Y. Yao and J. Gehrke, "The cougar approach to in-network query processing in sensor networks," *ACM SIGMOD Record*, vol. 31, no. 3, p. 9, 2002. doi: 10.1145/601858.601861

[70] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar, "Microhash: An Efficient Index Structure for Fash-based Sensor Devices," in *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, ser. FAST'05. Berkeley, CA, USA: USENIX Association, 2005, p. 3.

[71] Y. Zhang, L. Grieco, E. Baccelli, J. Burke, R. Ravindran, and G. Wang, “ICN based Architecture for IoT - Requirements and Challenges,” 2015. URL: https://tools.ietf.org/html/draft-zhang-iot-icn-challenges-02

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 29.03.2016 Lucas Andreas Jenß