BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

TOOL FOR MANAGEMENT OF FLASH MEMORY WEAR-LEVELING ON EMBEDDED SYSTEM DEVICE

NÁSTROJ PRO SPRÁVU OPOTŘEBENÍ FLASH PAMĚTI U VESTAVĚNÉHO ZAŘÍZENÍ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR AUTOR PRÁCE	MARTIN HAVLÍK
SUPERVISOR VEDOUCÍ PRÁCE	Ing. VÁCLAV ŠIMEK

BRNO 2023

Bachelor's Thesis Assignment

148632

Institut:	Department of Computer Systems (UPSY)
Student:	**Havlík Martin**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Tool for Management of Flash Memory Wear-Leveling on Embedded System Device**
Category:	Embedded Systems
Academic year:	2022/23

Assignment:

1. Study the fundamental principles of the wear-leveling technique. Survey the existing approaches for various types of modern data storage devices.
2. Based on the evidence collected in point 1) of the assignment, make a comparison to the wear-leveling features available in the latest version of the ESP-IDF framework.
3. Outline the conception of a tool that would provide diagnostic information on all necessary operations and the status of a target Flash-based memory module on the ESP32 platform. The tool should comprise 2 parts: an embedded-side data collector and a PC-side analyzer and visualizer.
4. Implement the embedded part (data-collector code) of the tool. Provide testing data and show the actual functionality.
5. Implement the PC host part (ideally in Python language) of the tool. Test various scenarios in a combination with the outcome from point 4) of the assignment.
6. Add documentation and publish your solution on GitHub under an open-source license (Apache License preferred, see http://www.apache.org/licenses/LICENSE-2.0).
7. Evaluate the functionality of the complete tool. Assess the achieved results and try to propose further directions for the development.

Literature:

- According to the instructions of the supervisor.

Requirements for the semestral defence:

- Fulfillment of points 1 to 3 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Šimek Václav, Ing.**
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	31.10.2022

Abstract

Thesis focuses on wear leveling layer over flash memory as provided by the Espressif IoT Development Framework with the goal of creating a tool for monitoring and managing flash memory wear caused by erase operations. For the purposes of such tool an extended version of wear leveling is implemented, addressing shortcomings of the Espressif's version. The enhancements include per sector erase count tracking and address randomization using a format-preserving cipher based on an unbalanced Feistel network for improved wear evenness. Said address randomization is tested by simulating full memory lifetime in selected erase stressing scenarios, with results showing up to a few percent improvement in wear uniformity over original wear leveling. Finally, a monitoring tool, consisting of an embedded back-end and a PC side graphical front-end, is created on top of the extended version of wear leveling.

Abstrakt

Práce se zabývá mechanismem wear leveling nad pamětí typu flash, zejména implementací poskytovanou společností Espressif v rámci IoT Development Framework, a to s cílem vytvořit nástroj pro sledování opotřebení flash paměti způsobené operacemi mazání. Pro účely tohoto nástroje je vytvořena vylepšená verze mechanismu, řešící některé nedostatky a rozšiřující původní wear leveling poskytovaný společností Espressif. Vylepšení zahrnuje možnost sledovat opotřebení paměti s rozlišením jednotlivých sektorů a zavádí pseudonáhodnou složku do mapovacího algoritmu za pomocí šifry zachovávající formát založené na nevyvážené Feistelově síti, vedoucí k rovnoměrnějšímu rozložení operací mazání po paměti. Toto pseudonáhodné mapování je následně otestováno pomocí simulace životnosti paměti ve vybraných zátěžových scénářích, s výsledky vykazujícími až pár procentní vylepšení v rovnoměrnosti opotřebení oproti původnímu algoritmu. Závěrem je vytvořen nástroj pro sledování opotřebení flash paměti, sestávající z vestavěné back-end části a PC front-end GUI vizualizační části.

Keywords

Monitoring tool, wear leveling, ESP32, ESP-IDF, flash memory, memory wear, SPI NOR flash, unbalanced Feistel Network

Klíčová slova

Nástroj pro sledování opotřebení paměti, wear leveling, ESP32, ESP-IDF, flash paměť, SPI NOR flash, nevyvážená Feistelova síť

Reference

HAVLÍK, Martin. *Tool for Management of Flash Memory Wear-Leveling on Embedded System Device.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Václav Šimek

Rozšířený abstrakt

Práce se zabývá pojmem wear leveling jakožto mechanismem využívaným při přístupu k paměti typu flash, s cílem rovnoměrného rozložení opotřebení paměťových buněk přes celé médium. Tohoto mechanismu je zapotřebí, jelikož každá buňka flash paměti má životnost udávanou omezeným počtem cyklů přepsání. Při dosažení tohoto maxima, či jen přiblížení se k němu, se snižuje spolehlivost buňky pro bezchybný zápis a čtení, až k možnému selhání buňky.

Wear leveling je povětšinou zajišťován řadičem paměti a tvoří transparentní vrstvu mezi logickými adresami při přístupu k paměti a konkrétními fyzickými adresami na médiu; typicky u SSD či flash USB disků, kde použitá paměť typu NAND flash vyžaduje průběžné dynamické změny mapování i kvůli vadným sektorům, které mohou vznikat během běžného využívání paměti tohoto typu.

Oproti tomu je ve vestavěných zařízeních hojně využívána flash paměť typu NOR, která běžně defacto netrpí vznikem vadných sektorů, v tomto ohledu je tedy spolehlivější. Za to, kde běžná kapacita NAND flash je v řádu desítek, stovek či tisíců GB, bývá NOR flash kapacitně o tři až čtyři řády menší. A to jednak z důvodu ceny na bit, která je u NOR flash výrazně vyšší, tak také jelikož vestavěné systémy mnohokrát nevyžadují tak velkých kapacit a spolehlivost NOR flash paměti je převažujícím faktorem. Dalším benefitem NOR flash paměti může být její schopnost přímo spustitelného kódu (eXecute In Place – XIP) bez typického mezikroku načtení instrukcí do operační paměti před jejich vykonáním.

Specifická pozornost je věnována vybrané cílové platformě ESP32. Moduly s čipy této řady obsahují integrovanou flash paměť typu SPI NOR, kde typická životnost paměťové buňky činí zhruba 100 000 přepsání. Sériové rozhraní SPI umožňuje těsnou integraci paměti v rámci modulů či systému na čipu (System on a Chip – SoC). Zároveň firma Espressif poskytuje modulární vývojový framework, ESP-IDF, jenž obsahuje implementaci wear levelingu přímo pro danou platformu.

Tato implementace realizuje tzv. „statický” wear leveling na základě algebraického mapování logických adres sektorů na fyzické sektory paměti. Statický je nazýván, jelikož se i málo přepisovaná, statická, data přesunují při změnách mapování, a tedy veškeré sektory mají časem vyrovnanou šanci na frekventované přepisování. Mapování logických sektorů na fyzické pomocí algebraické funkce je výhodné pro vestavěná zařízení s omezenou pamětí, poněvadž není nutné uchovávat častokrát rozsáhlé dynamické mapovací tabulky všech sektorů, navíc vyhledání a aktualizace informací v takové tabulce bývá časově nákladnější než pouhý matematický výpočet adresy.

Ve zmiňované implementaci wear levelingu v ESP-IDF byly nalezeny kritické nedostatky znemožňující dlouhodobé sledování opotřebení paměti. Z tohoto důvodu byla vytvořena vlastní rozšiřující verze, umožňující vést a uchovávat dlouhodobé záznamy o přepisech sektorů a nadto též vylepšující vlastnosti mapovacího algoritmu pro dosažení lepšího rozložení přepisů po paměti oproti algoritmu původnímu.

Implementovaná rozšiřující verze zajišťuje, byť s určitými odchylkami způsobenými návrhem vyvažujícím přesnost monitoringu vůči využití zdrojů nebo případnými výpadky napájení zařízení, vedení záznamů o počtech přepisů jednotlivých sektorů v dané části paměti, která wear leveling využívá. Tyto záznamy poskytují informaci, zda se některá oblast paměti, s rozlišením na jednotlivé sektorů, neblíží v počtech přepisů její živostnosti, v kterémžto případě je nutno naplánovat výměnu ať již samotné paměti či přímo celého vestavného zařízení, pokud je paměť integrována. Zde je výchozím předpokladem nasazení aplikace s rozšířeným wear levelingem pro novou, neopotřebovanou paměť. V opačném pří-

padě odhad zbývající životnosti paměti musí být proveden externě a počty přepisů uchovávané rozšířenou verzí interpretovány v kontextu předchozího opotřebení paměti.

Dále také zmiňované vylepšení mapovacího algoritmu spočívá v přidání mezivrstvy, která provádí aplikaci šifry zachovávající formát využívající nevyváženou Feistelovu síť na adresu sektoru při každém přístupu do paměti. Takováto šifra poskytuje pseudonáhodné mapování 1:1 a dosahuje lepšího rozložení přepisů po paměti zejména v tom případě, kdy jsou často přepisovány pospolité bloky sektorů, které se nyní díky šifře budou mapovat po jednotlivých sektorech na rozdílná místa v paměti.

Srovnání původního mapovacího algoritmu s výše popsaným proběhlo pomocí simulace životnosti paměti zatížené opakovanými přepisy. Experimentálně bylo v určitých případech zjištěno až několikaprocentní vylepšení oproti původnímu algoritmu znamenající dosažení až 99% „normalizované živostnosti" (perfektní wear leveling – všechny sektory přepsány stejněkrát – by měl 100%). Výpočetní režie přidané mezivrstvy s Feistelovou sítí je díky využití bitových logických operací minimální.

Nad popisovanou rozšířenou verzí wear levelingu byl navržen a implementován nástroj, rozdělen do dvou částí. První z nich – C++ back-end data-collector s názvem `wlmon` – umožňuje rekonstrukci informací o wear leveling vrstvě společně se záznamy o přepisech z flash paměti. Následně ze získaných informací sestaví hlášení a ve formátu JSON jej předá, pomocí spojení sériovou linkou s PC, druhé části nástroje. Tato část – Python front-end s názvem `espwlmon` – pomocí grafického uživatelského rozhraní zobrazí relevantní vyčtené informace a interní struktury a vizualizuje počty přepisů jednotlivých sektorů.

Tool for Management of Flash Memory Wear-Leveling on Embedded System Device

Declaration

I hereby declare that this Bachelor's thesis was created as an original work by the author under the supervision of Ing. Václav Šimek. Close guidance and extensive feedback along the entire journey was provided by Ing. Martin Vychodil from Espressif's Brno office. I have listed all the literary sources, publications, and other sources that were used during the creation of this thesis.

....
Martin Havlík
May 10, 2023

Acknowledgements

Nature does not hurry, yet everything is accomplished.

Lao Tzu

Aside from my supervisor and the man with a cozy house in a beautiful valley – whom I especially thank – I would like to dedicate this section to other precious people and dainty things. Firstly my supportive family, as without them I could not sustain the way I approach things, which at points is surely wickedly easygoing. Secondly Babák's girl, daresay light of my life, who continues to be simply astonishing. Thirdly the few remaining people I cherish, with all the memories back to the pavements that we walked, and walked (and the talking, knightly noelka or that one time my ALT got bad...).

A rapid-fire of associations in no particular order: MdB, Psota, PP & AB, Dune, Davies, Veselá Bartošová, Natka, Adamov, focaccia, the thing and Alina, bnb & para, Bonz Atron, Křižíkova, Κουφονησι, Herr Morgan, malaka, U Kostelíčka, Baru N. et al., Starkid, Osho, Kvothe, čistička Velvet Vary, daydreaming, 22, 418 ... Gielinor.

And now, gentlemen – dog-eyed or not, a short view back to the past. Quoting, at this point half a decade old, myself

> I sometimes feel like breathing the red dust out, just watching the old pond, listening to the warrior of love and dancing to songs about beginner's mind sung to people of Orphalese, thus spoke me – awkward ellipsis for I was not capable of typesetting Japanese here.

And so it happens that things have come full circle once again. Good gracious god. May I find my worth in the waking world.

Contents

List of Figures

Chapter 1

Introduction

Since the 1990s, flash memory has become the go-to non-volatile rewritable random access memory technology used in embedded systems. Power-independent long-term storage of both code and data, together with reliability and the prospect of single package integration with CPU, RAM, and peripherals fits the embedded world nicely. Moreover, the ability to quickly re-program firmware even by Over-The-Air (OTA) updates and to split hardware and software code design into parallel development branches makes flash memory flexible and desirable even from the cost perspective. In addition to other factors, these are what make flash memory dominant in the Micro Controller Unit (MCU) market [10].

But flash memory does have certain disadvantages and concerns that need to be addressed, mainly its durability in terms of repeated write and erase, which affect the lifespan of its memory cells. More on this and flash memory in a more technical sense in Chapter 2.

To minimize the possibility that some memory cells wear out much more than others (and impact the reliability of the memory as a whole), a mechanism for spreading or leveling the wear is needed. Wear leveling is such a solution and has been approached and implemented in many different ways and in various systems over the years. Chapter 3 gives an overview of various approaches to wear leveling, including a look at Espressif's implementation of wear leveling in their IoT Development Framework (ESP-IDF), which is the main concern of this thesis.

Chapter 4 opens with an issue discovered in the ESP-IDF wear leveling implementation, which makes it not possible to monitor memory wear long-term. To mitigate the issue and make the to-be-created tool actually useful, an extended version of wear leveling is proposed and implemented. Together with solving the main issue, an improved mapping algorithm is implemented. It introduces address randomization using a format-preserving cipher based on an unbalanced Feistel network, as proposed by relevant literature.

To evaluate the improved mapping algorithm, a simple memory erase simulator was also implemented. Chapter 5 delves into experiments with simulated memory erase stress over its lifetime in selected scenarios and configurations and analyzes the effects and improved wear leveling efficiency of the introduced address randomization layer.

On top of the extended wear leveling, a tool for monitoring memory wear is designed and implemented. Key features being reconstruction of wear leveling layer status and visualization of memory wear, used for assessing the lifetime of flash memory in a given application. More about the tool, including an overview of testing on an embedded device, with examples of the tool functionality and output, is presented in Chapter 6.

Lastly, Chapter 7 offers a summary of the results achieved, as well as some closing thoughts.

Chapter 2

Need for wear leveling in flash memory

Flash memory consists of a large number of individual memory cells – modified Metal Oxide Semiconductor Field Effect Transistors (MOSFETs) with floating gate (FG) – each storing a single or multiple bits. Thus, we differentiate SLC (Single Level Cell) and MLC (Multi Level Cell) flash memories. Both operate on the same principle of trapping electric charge to manipulate the threshold voltage (V_T) of the transistor. This results in a measurable current at a given reference voltage or a measurable voltage at a given reference current; either way, the measurable value corresponds to the cell programmed state [12].

So, 2^n distinct measurable values of a cell encode an n bit information, up to penta-level cells (PLCs) that store 5 bits. The higher the bit amount per cell, the narrower the gap that separates neighboring levels, making the cell susceptible to read and write disturbances and noise, which can corrupt stored information. Therefore, Error Correction Code (ECC) technique employing parity bits is often used together with repeated read attempts to recover data. In addition, reading the entire data value stored in an MLC cell requires multiple reads at different V_{READ} voltages. The great bit density of MLC memories has the disadvantage of introducing energy, time, and circuit overheads. SLC offers the highest performance, but also at a high cost [12].

To program the cell, an electric charge has to be transferred to and from the floating gate. This is generally achieved by Channel Hot Electron (CHE) injection and Fowler Nordheim Tunneling (FNT) for programming and erasing, respectively [19]. An illustration of charge transfer can be seen in Figure 2.1.

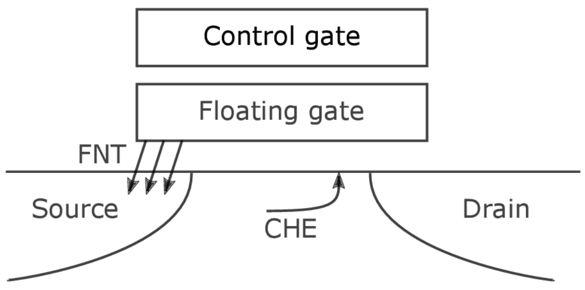


Figure 2.1: Schematic cross section of floating gate MOSFET with visualized Fowler Nordheim Tunneling and Channel Hot Electron charge transfer (based on [19])

Now a practical issue of flash memory is the fact that rewrites cannot be done in place, such as in typical magnetic media of Hard Disk Drives (HDDs). Clean flash memory will have all bits set to logical one. The write operation can manipulate selected bits to logical zero, thus forming the written data in flash. However, turning a 0-bit into a 1-bit cannot be done individually. Instead, a whole block unit has to be erased, which sets all bits in a given block back to one.

On top of that, the core problematic property of flash memory is its limited cell lifetime. Repeated writing (programming) and erasing of a cell, also called program erase (PE) cycling, degrades the current-voltage characteristics of the cell. The threshold voltage (V_T) is shifted, the leakage current increases, and the overall performance of the cell degrades. Degradation is usually more or less uniform in effect, but with few anomalously strongly affected cells that mainly suffer from data retention problems. These single cell failures are what influences overall endurance, as they affect the reliability of the whole memory [3].

The reliability of a flash memory cell is generally comprised of its endurance (ability to store data after program erase cycling) and retention (ability to store data long-term without power) [17]. The focus of this thesis is on the program erase endurance aspect of flash memory.

Then comes the question of how to prolong the lifetime of flash memory when simply repeated rewriting (write–erase–write) wears out the cells. The main technique to deal with this issue is called wear leveling and is usually managed by the memory controller (such as in an SSD). The goal of wear leveling is to distribute erase operations in an even manner across memory cells in order to minimize the potential for an earlier failure of some cells compared to others. In other words, the objective is for all cells to deteriorate at the same rate and reach their end of lifetime at the same time [14], [2]. More on wear leveling will follow in Chapter 3.

2.1 NOR and NAND flash memory

There are two broad flash memory types, NOR and NAND. They differ in their on-die cell array structure and, in effect, also in the way read/write/erase operations are performed and at what scale. More focus will be given to NOR flash going forward, as it is the memory type of main concern of this thesis.

2.1.1 NOR flash memory

NOR flash offers fast random access and can be efficiently reprogrammed since its cells are individually connected and thus addressable, programmable, and erasable [12] – see Figure 2.2 case a); each cell is individually connected to Wordline, Bitline and Sourceline. The listed qualities make NOR flash suitable for code storage, even enabling eXecute In Place (XIP), where the code is executed right from flash, sidestepping the usual load to RAM as is typical on desktop computers. For this, NOR memory has to be designed with read latency low enough to enable efficient execution directly from memory, so as not to bottleneck the system.

At a lower density, when compared to NAND flash, it has a higher cost per bit. However, a capacity in the range of megabytes can suffice for most embedded applications. Larger storage requirements might be satisfied by micro-SD card modules or various memory mediums other than NOR flash.

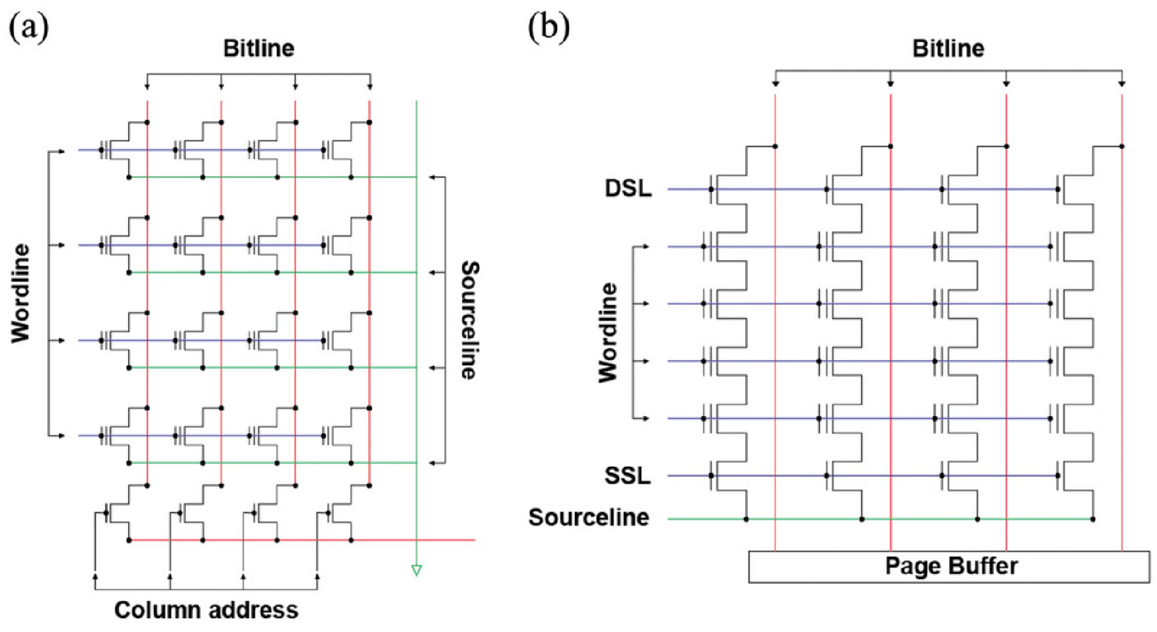


Figure 2.2: Schematic diagrams of flash memory cell array. a) NOR flash memory, b) NAND flash memory (taken from [12])

In the context of NOR flash memory, the size unit for read, write, and erase might be the same for all operations and, as such, is typically referred to as a sector.

2.1.2 NAND flash memory

On the other hand, NAND flash memory cells are typically arranged in a serially connected string-like structure – see Figure 2.2 case b); all cells in a string share Sourceline and Bitline. This allows for a higher density because each cell does not need its own contacts (as is the case for NOR). But it is inefficient in having to charge up the whole array and creates disturbances between neighboring cells [12].

A block in NAND, which is the smallest erase unit, is typically further divided into pages, which are the smallest read and write units. This disparity means that a page cannot be erased by itself; the whole block in which it resides has to be erased for that. Therefore, it is simpler to invalidate the old page, write a new one to an available area, and provide a mapping for the page to its new location. Invariably, by doing this, the available area will run out of places to write and will need to erase a block, ideally full of invalid pages, to make some pages writable again. Few valid pages will almost certainly remain in the block. Thus, it will require copying valid pages elsewhere to allow for block erase, which will free up pages for writing. This process is called Garbage Collection (GC) and is of immense importance for NAND flash to work optimally and efficiently.

This type of flash memory also suffers from occasional failed read and write operations due to disturbances between cells. For this reason, each page (smallest read and write unit) has an additional private capacity reserved for Error Correction Code (ECC) used to mitigate raw bit errors [15].

2.2 Effect of wear leveling on flash memory lifetime

But as aforementioned, both types of flash memory cannot perform rewrites in-place, and every write operation has to be preceded by an erase. This is the reason program erase cycling occurs and why memory wear is the unavoidable consequence with wear leveling only reducing the impact.

To demonstrate the tangible need for wear leveling and, in turn, the improved memory lifetime it can achieve, examples of calculations, heavily based on [11], are presented below.

Assume NOR flash memory where each sector can be rewritten 100 000 times. A single sector (e.g. containing a file) will be continuously rewritten nonstop with 50 updates every hour. Without wear leveling, this will always target a single sector. As can be seen from Equation (2.1), the expected lifetime of flash memory in this scenario is less than 84 days.

$$L_{exp} = \frac{100\,000 \times 1\ sector\ targeted}{1\ sector\ rewritten \times 50\ updates/hour \times 24\ hours/day} < 84\ days \ll 1\ year \quad (2.1)$$

Now assume that the full wear leveled memory is 1 MB in size, with sectors being 4096 B (4 KB) each. This results in 256 sectors in total. In the case of perfect wear leveling, the continuous rewriting of a single sector will over time target every sector in the memory the same number of times. This greatly extends the lifetime of full memory since it is no longer a single sector that bears all the erase-induced wear. According to Equation (2.2) this can result, in the perfect case, in more than 50 years of rewrites, that memory can endure.

$$L_{exp} = \frac{100\,000 \times 256\ sectors\ targeted}{1\ sector\ rewritten \times 50\ updates/hour \times 24\ hours/day} \approx 21\,000\ days \approx 58\ years \quad (2.2)$$

The substantial prolonging of expected memory lifetime in the above example calculations shows that wear leveling plays an important role in establishing reliable and long-lasting embedded solutions. Without it, some write-heavy applications would turn embedded devices into disposables with turnover in months or even less.

2.3 SPI NOR flash memory in ESP32

Moving towards specifics, this section outlines the hardware used during the creation of this thesis, including a closer look at the flash memory used in ESP32 Systems On a Chip (SoCs) and modules – or at least a flash memory chip model assumed to be of similar attributes to the one integrated in ESP32.

2.3.1 Used and relevant hardware

The development and testing of both the extended version of wear leveling and the embedded back-end part of the tool was done on `ESP-WROVER-KIT V4.1` development board, which uses the `ESP32-WROVER-E` generic Wi-Fi + Bluetooth + Bluetooth LE MCU module with integrated Quad SPI flash memory and comes in multiple flash size variants, ranging from 4 MB to 16 MB [6].

Real flash memory erase stress testing was performed on `ESP32-C3-DevKitC-02` which contains the `ESP32-C3-WROOM-02` module, this time in 4 MB or 8 MB variants [5].

All mentioned use cases for the boards (both graciously lent by Espressif) in possession are further discussed in Chapter 4 and Section 6.2 for extended wear leveling and embedded part of the tool, respectively, as well as Section 6.4 for `erase_stress_example` used for testing the full implementation. The selection of specific boards was completely arbitrary, as attention is drawn solely to the flash memory within.

2.3.2 SPI NOR flash memory

Flash memory is, in practice, external to CPU and other on-chip memory and takes the form of an integrated component on the module. Creating, together with other components such as external crystal oscillators, peripherals, and wireless connectivity modules, the complete package. On-module integration of flash memory is achieved by using the Serial Peripheral Interface (SPI). Due to the serial nature of the interface, it does not require many pins for connection and can even outperform 8bit/16bit parallel flash memories in Dual or Quad SPI modes, and is thus widely used in embedded systems as the go-to flash memory interface [21].

Detecting the actual flash memory size in a given board's Espressif module can be done by using `esptool.py flash_id` (the tool is included in ESP-IDF). The flash sizes reported for both boards in possession were 4 MB.

The `flash_id` command also returns the manufacturer and device IDs of the flash memory chip itself. These should be then looked up in a list of IDs maintained by Espressif. But for the development boards at disposal, the reported values could not be matched to any memory model or manufacturer in this way.

Using a list of IDs maintained by the Flashrom project [7] also did not single out a specific chip. However, it narrowed the manufacturer down to presumably XMC, which was later confirmed by Espressif. Yet, the specific device ID still could not be matched.

2.3.3 Exemplary flash memory model

Following the result of the previous Section 2.3.2, an assumption had to be made about the exact flash memory model used. In that case, a flash memory device with an ID numerically close to what `esptool.py flash_id` reported was chosen to represent the SPI NOR flash integrated in the `ESP32-WROVER-E` and `ESP32 C3 WROOM 02` modules. This being the `XM25QH32C` flash chip [21].

A summary of the assumed target memory characteristics is as follows. SPI NOR flash with 32Mbit capacity, from 108 MHz standard SPI clock up to 432 MHz in Quad I/O with specialized instructions, more than 100 000 PE cycles and 20 years of data retention, allows true eXecute In Place (XIP), typical sector erase time 50 ms, page program typically 0.5 ms, uniform 4 KB sectors.

Presumably, the only attribute relevant to this thesis is the proclaimed 100 000 PE cycles cell/sector lifetime. This is a well-known ballpark value and was already used in this thesis; the data sheet then just supports this claim for further application of the value, e.g. for simulation as described in Chapter 5.

Aside from that, the data sheet mentions a page unit (256 B each) being separate from the sector unit (4 KB each), but from the point of view of wear leveling in ESP-IDF, even though it does use two separate variables for page and sector sizes, there is no distinction between them, and all read/write/erase operations are performed at the 4 KB sector granularity.

Chapter 3

Overview of approaches to wear leveling

Wear leveling is commonly implemented as a mapping between logical addresses or sectors and their physical counterparts. This mapping changes over time, resulting in various locations in physical memory being accessed and manipulated when the same logical address is used. For the higher software or firmware layer on top of the flash, this mapping is transparent and provides an abstraction over the memory medium. If no wear leveling layer was present, repeated program erase cycling could target a single memory location, prematurely wearing out a select few cells in that area.

Thus, over time, the said mapping needs to provide as even a spread of erase stress as possible while keeping in mind the constraints of the system such as memory usage, algorithm complexity, and overall resource overhead.

What follows is an overview of typical approaches to wear leveling algorithms, their benefits and shortcomings.

3.1 Dynamic and static wear leveling

In a general sense, there are two categories of wear leveling techniques, namely static wear leveling and dynamic wear leveling. They differ in the extent to which they affect logical to physical memory mapping.

First, dynamic wear leveling is focused solely on the so-called "hot spots" which form due to the fact that data access is not uniform. Memory contents can be classified as "cold data", that is, static and unchanging in the long term, and "hot data", meaning that the data are accessed and rewritten regularly, thus forming "hot spots". This approach brings with itself few inherent issues, mainly the fact that a heavily written area will get wear leveled, while rarely written areas will not participate as much or not at all, resulting in the active memory used for wear leveling being a fraction of full available memory. This reduces the potential memory lifetime. Dynamic wear leveling also requires efficient identification of hot data, which might not be trivial. Despite all that, a potential benefit of this approach is the lack of a requirement for all memory accesses performing logical to physical mapping; static areas can be accessed directly without mapping or locating overhead [11].

For static wear leveling, the objective is to also include static or infrequently updated data in the wear leveling process and move them around to other locations so as to prevent any static data from staying at any location for a long period of time. This approach has the potential to ensure good wear leveling across the whole memory, depending on the specifics of the algorithm used. However, this approach also requires logical to physical mapping for all memory accesses together with additional erases needed for moving static data. Despite these overheads, good or close to ideal static wear leveling has the potential to greatly extend memory lifetime, as previously shown in Section 2.2.

3.2 Embedded flash file systems

File systems with a design specific to flash memory can address wear leveling intrinsically either as a by-product of a certain mechanism used or intentionally by implementing a victim sector selection for garbage collection in a way that provides some form of wear evenness.

Journal Flash File System (JFFS) in version 1 is purely log-based. Nodes containing file data and metadata are written to flash sequentially, maintaining a linear progression of new nodes being written and old marked obsolete. These obsolete nodes create "dirty space" which is garbage collected by copying possible valid nodes from the `head` of the log to its `tail` and erasing full blocks freed by copying. This `head` and `tail`, shrink and grow, of a circular log provides perfect wear leveling, where each block is erased exactly the same number of times [20].

SPI Flash File System (SPIFFS) is specifically designed for SPI NOR flash memory used in embedded devices. Inspired by another flash file system, YAFFS, it distinguishes itself by no heap low RAM requirements, using flash write and erase in clever masking ways and also by ensuring static wear leveling by once again being log structured. On the other hand, it does not support directories, only a flat file structure, is poorly scalable (due to fixed low RAM usage), and it does not detect or handle potential bad blocks [1].

Or in the case of HNFFS [16], a proposed file system with design specific to NOR flash which improves upon both JFFS and SPIFFS, a specific active algorithm in the form of Random Static Wear Leveling (RSWL) is used. As the file system tracks *clean* sectors, containing only valid nodes, and *dirty* sectors, containing invalid nodes, together with sector erase counts, garbage collection can select the victim sector for erasing from the *dirty* sectors and also with a small chance also from the *clean* sectors, thus introducing a randomness element to victim selection and, in turn, wear leveling.

Littlefs is another microcontroller-oriented file system, which combines small-scale logging for atomic metadata updates and Copy On Write (COW) structure, which ensures power loss resilience. Somewhat differently from others, it provides dynamic wear leveling and detects and works around potential bad blocks in flash memory [9].

Despite the options available, Espressif has opted for a more modular and flexible approach in the form of separating the wear leveling layer and an embedded flash file system on top of it. Specifically `fatfs` which is a generic platform-independent FAT/exFAT file system module for embedded systems with very small code footprint and memory usage [4]. When compared to previously summarized embedded file systems and also taking into account that it is based on FAT/exFAT and is completely separated from the disk I/O layer, `fatfs` does suffer in the prospect of using the characteristics of NOR flash memory to its advantage as other flash file systems might do.

Moreover, the fact that wear leveling layer is separate allows for modifications to its logic and algorithms, which is happily used in this thesis by implementing an extended wear leveling version, which of course has to respect the API of initialization and accessing the flash memory. For more about this extended version, see Chapter 4.

3.3 Wear leveling in ESP-IDF

Espressif IoT Development Framework (ESP-IDF) is a framework for developing applications targeting Systems on a Chip (SoCs) made by Espressif. It is supported on Windows, Linux, and MacOS. The framework provides a modular build environment using CMake, multiple Python tools, and a plethora of components, wear leveling being one of them, which can be used in user-created projects. This implies that wear leveling in this implementation is not provided by means of a memory controller unit; instead, it is just an indirection layer of CPU executed code which is slotted between raw flash access and file system routines. Optionally, wear leveling functions, such as partition `mount` and `unmount`, are accessible directly, without an overarching file system. This raw access is, for example, utilized in the `erase_stress_example` described in Section 6.4.

As of writing this thesis – and implementing the monitoring tool – version 5.0 of ESP-IDF is the latest major release and, when not stated otherwise, is the version of concern.

What follows, aside from a quick detour to flash partitions, is a summary of the current wear leveling approach in ESP-IDF, mainly the logic behind the mapping algorithm used.

3.3.1 Flash partitions

The flash memory address space for the ESP32-xx series chips is designed to be split into partitions. Partition being a contiguous block of memory together with a name, its type, and subtype. These descriptive attributes are stored in a "partition table", a binary encoded list of partitions residing at a fixed offset in the flash. Thus, reading and parsing the table at a known offset provides the information about the present partitions and by type/subtype their intended use.

The main two types of partitions are `app`, for an executable binary that the chip runs, and `data`, for the storage that the application can use.

So, given a `data` partition (its size and start address are specified in the partition table), a flash file system can be mounted on top of it, giving access to typical file manipulation functions. At the time of writing this thesis, the main embedded file system used and integrated into ESP-IDF is `fatfs` and is henceforth assumed to be the file system used in any and all examples or mentions, if not stated otherwise.

As described in Section 2.3.1, available hardware offered a 4 MB flash chip. From that capacity a 1 MB `data` partition was typically allocated for use by `fatfs` with the wear leveling layer enabled and present between the file system and flash. This 1 MB wear leveled partition case will be occasionally used in examples and also in Chapter 5 which focuses on memory erase stress simulation.

3.3.2 Wear leveling component in ESP-IDF

The approach and algorithm used are in many factors similar to the Start-Gap Wear Leveling proposed by Qureshi et al. [18]. They differ in terminology and ESP-IDF wear leveling leaves out address space randomization. The original algorithm by Qureshi was designed for Phase Change Memory (PCM), but it is easily applicable to flash as well, also being a memory with limited write (erase) cell lifetime. A summary of the algorithm, with terminology as used in ESP-IDF, follows.

The main idea is an algebraic mapping of `N` usable (e.g. for `fatfs`) sectors to `N+1` physical flash sectors. The additional reserved sector is "dummy sector", it does not contain useful data and serves as a buffer for copying sectors. The position of the dummy sector is determined by a `pos` value, starting at 0. Every erase operation increments an `access_count` counter. Once `access_count` reaches the `updaterate` threshold (Qureshi labels it as ψ) – currently fixed at 16 – it triggers a dummy sector move.

A sector at `pos+1` – next to the dummy sector – is copied to the dummy sector (which is first erased) and `pos` is incremented. Also, `access_count` is reset to zero, which means that it will take another `updaterate` erase operations to trigger another dummy sector move.

For every such dummy sector move and `pos` increment, a new so-called `pos update record` gets written to flash. These are written in a tally mark like fashion one after each other, meaning that the sector which contains them does not have to be erased every time said new record is written. These records enable reconstruction of the up-to-date value of `pos` when mounting the wear leveled partition.

Once the dummy sector travels to the end of address space of a given partition, it loops back to the beginning of said partition and an overall mapping shift is introduced via a `move_count` counter. As it also starts at 0, the first dummy sector partition go-through does not really have shifted mapping, but subsequent loops do. As the dummy sector loops and `move_count` is incremented, all existing `pos update records` are erased, making room for a new round of tally marks.

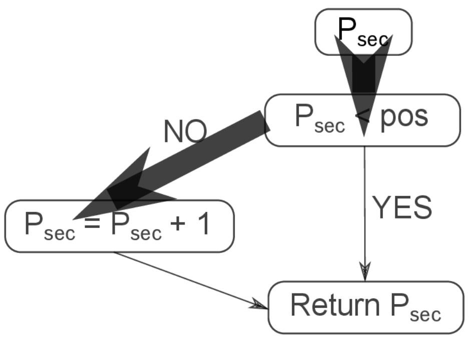
Thus, there is `move_count` which dictates the shift performed when calculating the physical sector address – P_{sec} – from the logical sector address – L_{sec} – as can be seen in Equation (3.1).

$$P_{sec} = (\texttt{sector\_count} - \texttt{move\_count} + L_{sec})\ mod\ \texttt{sector\_count} \tag{3.1}$$

P_{sec} and L_{sec} denote **sector addresses**. For example a 1 MB partition contains 256 sectors each being 4 KB, so possible sector addresses are 0 to 255. This allows us to use `move_count` and `sector_count` (256 for the example values mentioned) in the calculation.

On a side note, sector size of 512 B is also possible with existing alternative wear leveling modes (so-called `performance` and `safe`), but this thesis assumes sectors to be the 4 KB "default" size; this applies to implementation as well.

Before using P_{sec} to access the memory, the position of the dummy sector must also be considered. Figure 3.1 shows the logic of shifting sectors that map to the dummy sector or to the „right" of it by an offset of one.



A simple example of the mapping principle can be seen in Figure 3.2. The numbers inside the squares are the addresses of logical sectors and `D` denotes the dummy sector. Having 4 total sectors with the dummy occupying one, we are left with 3 usable sectors.

In case (a), only the shift described in Figure 3.1 occurs as `move_count` is 0.

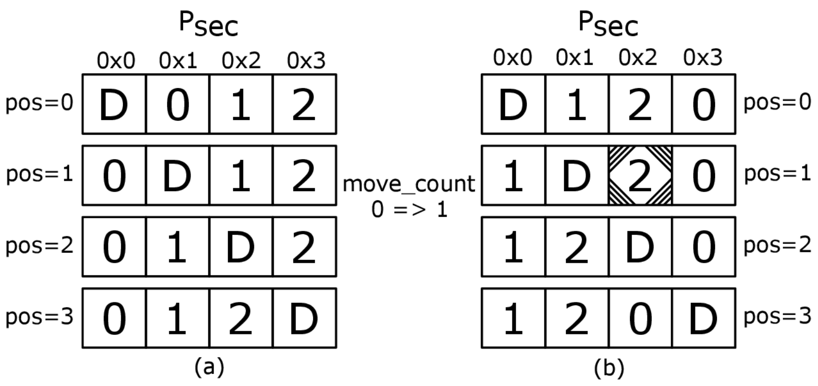


Figure 3.2: Simple logical to physical sector address mapping example with 4 total sectors. Case (a) with `move_count=0`, case (b) with `move_count=1`.

In case (b), `move_count` becomes 1. Now both the calculation in Equation (3.1) and the shift in Figure 3.1 take place. For example, we want to access $L_{sec} = 2$ while the dummy sector is at $pos = 1$. In that case $P_{sec} = (4 - 1 + 2)\ mod\ 4 = 1$ and $1 < 1$ ($P_{sec} < pos$) is *not* true, thus $P_{sec} = P_{sec} + 1 = 1 + 1 = 2$ and that is the final physical sector address (also highlighted by hatching in the figure).

Note that the last row with $pos = 3$ in case (a) and the first row with $pos = 0$ in case (b) differ only in the first and last sectors (physical addresses `0x0` and `0x3`) being swapped. This exact behavior of the algorithm ensures that even with `move_count` incrementing and introducing shifts, the mapping remains correct, consistent, and transparent for any higher layer.

The last important detail about the implementation is the existence of reserved sectors at the end of wear leveled partition. As they are allocated and written with offsets calculated from the end, they are to be read "from right to left" as can be seen in Figure 3.3. These contain a sector reserved only for a `config` structure (memory size, sector size, updaterate, etc.) and `state` structure (move count, erase and pos thresholds, unique device id, etc.) together with `pos update records` in two copies for redundancy. As the maximum number of `pos update records` must accommodate for all possible `pos` values, dictated by the sector count in the partition, it might be necessary to reserve multiple sectors for this (and then again in two copies). The remaining sectors are, bar 1 acting as the dummy sector, usable, e.g. for a file system to be mounted on top of the wear leveling layer.

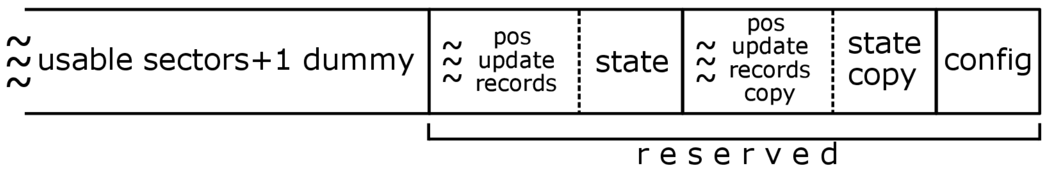


Figure 3.3: Schematic of sectors reserved by wear leveling at the end of a data partition

Chapter 4

Improvements to wear leveling

In the process of studying the current implementation of the wear leveling component in ESP-IDF, a fatal flaw was discovered that makes monitoring and determining long-term flash memory wear impossible.

The mapping is determined by two previously mentioned counters; `pos` and `move_count`. From their values, an estimate could be made for the total erase count of all sectors, which could then be divided by the number of sectors in a partition to get a rough per sector estimate. As the expected lifetime of NOR flash memory cell/sector is around 100 000 erases, the leeway for monitoring precision is generous. For example, a resolution of a few thousands of program erase cycles can still provide useful information on the possibility of the memory reaching its lifetime.

But these counters are periodically[1] reset back to zero and written to flash. This means that reaching the period, which can realistically occur, results in all information about past erases being lost. Therefore, it is not possible to reconstruct long-term memory wear statistics.

To make the to-be-created monitoring tool of any use, long-term erase count tracking must be possible. Thus, an extended and improved version of wear leveling for ESP-IDF has been implemented under the name `WL_Advanced`.

First, it fixes the counter resetting problem while also adding a more precise erase count tracking. Additionally, address randomization using an unbalanced Feistel network is used to improve erase distribution across the wear leveled partition.

This extended wear leveling diverges from the goal of a management tool, as specified by the assignment of this thesis. However, as it is necessary for obtaining any long-term statistics, this detour was prioritized over other – planned, possible or potential – functionality of the tool.

4.1 Extending the current approach

The existing approach of the ESP-IDF wear leveling component, described in Section 3.3.2, offers a simple, good and reliable static wear leveling algorithm. The counter resetting problem can be trivially solved by adding another counter which increments when the original counters are reset. This counter of “full cycles” (`cycle_count`) is never zeroed,

[1] circa 1M erase operations for 1MB partition, 4K sectors, dummy sector moves every 16 erases; for 250 usable sectors in such partition it is 4000 erases/sector which is $\ll$ 100 000

making it possible to calculate all time overall erase count, with capacity way beyond possible memory lifetime.

But even with the counter resetting solved, the implementation suffers in the possibility of monitoring erase operations in a precise enough way. Using redundancy in the design of some structures, an extension was thought of and implemented.

As already present in the implementation, writing to flash was done at minimum in `pos update records` of 16 B, for that is required by flash encryption and its block size and alignment[2]. But this 16 B record was "wasted" only on four Cyclic Redundancy Check (CRC) values where a single one could achieve the same functionality of an "over-engineered flag". The minimum write size has to remain, but the usage is surely to be improved.

The capacity of the record can be used to store information about erase operations, although still with a resolution set by `updaterate` (only every `updaterate`th erase is recorded). This `updaterate` value is currently fixed at 16, so information only about every 16th erase is saved. Every `pos update record` then now contains chiefly *the physical sector number whose erase triggered the writing of the record to flash* (as well as `pos` for future reconstruction as in the original algorithm, and additionally unique `device_id` and a CRC of all previous fields).

The `updaterate` value of 16 is used in ESP-IDF and is also adopted in the extended version implemented. It is believed to achieve a good balance between precise monitoring potential and frequent record keeping, which also requires moving the dummy sector.

Once the `pos update records` are to be erased, as is done once the dummy sector gets to the end of the partition and loops back to the start, and `move_count` is incremented, erase count aggregation takes place.

Quickly aside, additional sectors are reserved at the end of the wear leveled partition to contain per sector erase counts, again in two copies. On mounting the partition, an attempt is made to read existing erase counts to a buffer.

Then the aggregation simply increments `erase_count` in the buffer for the given sector that is found in a `pos update record` while traversing them. After such aggregation, the buffer contains the up-to-date erase counts of all sectors, which inherently requires large enough buffer especially for bigger partitions – each erase count is represented by a `uint16_t` value. Finally, the updated per sector erase counts from the buffer are written to their respective reserved sectors in flash, and `pos update records` are erased, finishing the diversion from the original algorithm flow.

Due to the fixed `updaterate` erases resolution for writing one `pos update record`, the `erase_count` kept and saved in flash **is to be multiplied by said `updaterate` to obtain the real and correct estimation of per sector erase count**. This fact that stored – in buffer and in flash – erase counts are noticeably smaller, by the factor of `updaterate`, than real values allows for a more storage efficient representation. For this, `uint16_t` was deemed sufficient; the maximum representable erase count is $(2^{16} - 1) \cdot$ `updaterate`, and when `updaterate=16` this becomes over a million sector erases. It may be safely assumed that no current NOR flash memory technology comes even close to being able to achieve such cell/sector erase lifetime.

This addition of erase counts records and aggregation vastly improves the monitoring accuracy achievable, as without per sector records only an overall erase count could be calculated from `pos`, `move_count` and `cycle_count` and then divided by the number of sectors in partition to give a rough estimate. With per-sector tracking, proper heat spots

[2] https://docs.espressif.com/projects/esp-idf/en/latest/esp32/security/flash-encryption.html#writing-to-encrypted-flash

can be localized and compared to expected behavior of the monitored long-term running application.

But, as mentioned, only every `N`th (`updaterate`th) erase gets a record written to flash. Until that threshold is reached, `access_count` counter holds the number of erase operations performed. Therefore, if the embedded device is restarted or suffers a power loss without proper unmounting of the wear leveled partition, the information about `access_count` erases that have not reached the threshold is lost.

Lowering `updaterate` would result in `pos update records` being written more often, but at the price of also more frequent dummy sector moves and erases of reserved sectors, which is just adding more wear leveling layer overhead with questionable improvement in leveling efficiency. However, in an environment where the embedded device is expected to restart or lose power before reaching the threshold most of the times, lower `updaterate` would make sense. On the other hand, in a more stable and even performance-oriented applications, a higher threshold would be beneficial, as it would lower the overhead by reducing the frequency of dummy sector moves.

This aspect of manipulating `updaterate` has not been explored and could prove to be a viable improvement to the wear leveling algorithm. If, for example, `updaterate` could dynamically adjust to the erase behavior of a given application (if such thing is even possible with the current way of record keeping logic).

4.2 Address randomization

Also proposed by Qureshi et al. [18], besides the algebraic mapping approach to static wear leveling, is a solution for the fact that erase operations are likely to occur spatially close to each other, which can result in a highly erased sector becoming mapped to another highly erased one, as the mapping based on the dummy sector shifts by one sector at a time.

By adding a layer of randomization, even adjacent sectors can get mapped to vastly different areas of memory. Thus, ensuring that locality of erases does not factor into memory lifetime.

Because flash memory, in the context of embedded systems, is typically about three orders of magnitude smaller in capacity compared to the PCM memory that Qureshi is concerned with (16GB), the issue of erase locality is not critical for flash. Even repeated erases of a single sector will cause the mapping to loop through the entire memory or partition before the sector erase lifetime is reached. However, for “big” data partitions – such as 16 MB of 4096 4 KB sectors – the fact that dummy sector has to move all the way to the end of partition before `move_count` mapping shift is changed could prove problematic. If a single sector is erased in that scenario, it can accumulate tens of thousands of erases before the mapping is shifted. This is far from ideal, as it results in poor wear leveling, and solving it would potentially require deeper changes to the wear leveling algorithm or even the approach itself. However, this thesis focuses on extending the current ESP-IDF approach with available hardware, and thus assumes partitions only in the range of few megabytes at maximum, with a typical example of 1 MB of 256 4 KB sectors.

That aside, experiments of simulating memory erase wear over its lifetime in small partitions have shown that randomization does indeed improve the uniformity of distribution of erases, even if only marginally. For said simulation, see Chapter 5.

Qureshi [18] proposes two approaches to address-space randomization, namely Feistel Network Based Randomization and Random Invertible Binary Matrix. Both perform quite well and are closely similar to each other in the analytical and experimental assessment of normalized memory lifetime for various workloads he presents. Where they primarily differ is in the storage overhead of each method. For a B-bit address space, the RIB matrix presents B^2 bits of overhead, while the Feistel network only $1.5 \cdot B$ bits, as instead of having to store a full matrix, only three short randomly generated keys are required.

Based on this storage overhead quality of Feistel network address randomization, this approach has been selected and implemented, albeit with a minor modification in the form of unbalanced nature of the network, where the randomized address can be split into two parts of differing lengths.

4.2.1 Unbalanced Feistel network

The alluded to address randomization uses an integer format-preserving encryption (FPE) based on a 3-stage unbalanced Feistel network with randomly generated keys, as seen in Figure 4.1. It provides a 1-to-1 mapping between a logical sector address `LA` and an intermediate address `IA`. In each stage, the address is divided into two parts – `L` and `R`, which are `msb` and `lsb` bits long, respectively. If `LA` is B bits long, then $lsb = \lceil \frac{B}{2} \rceil$ and $msb = B - lsb$. So, for an odd bit-long logical sector address, the two parts are of different lengths; thus, the unbalanced nature of this particular design of Feistel network.

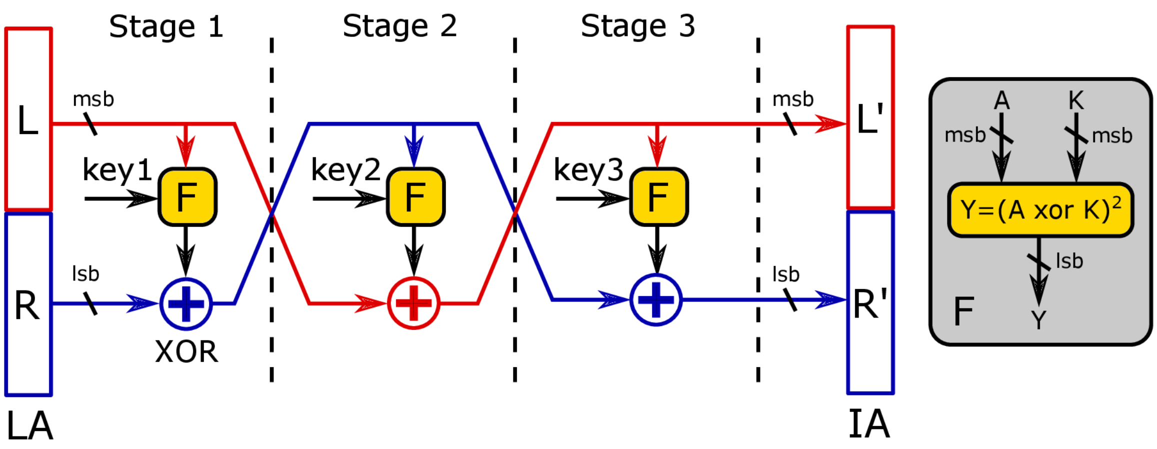


Figure 4.1: Unbalanced 3-stage Feistel network schematic performing translation between Logical Address (LA) and Intermediate Address (IA)

Once the address is split for the stage, a function `F` is performed on `L` and a `key` specific to that stage. All keys are randomly generated at the creation of reserved flash sectors and written to flash together with `state` structures, which means that for the lifetime of the memory or the wear leveling structures in flash, the keys are fixed and provide consistent mapping results. `F` is same for all stages and is simply a squaring of XOR of its inputs, just as used by Qureshi for ease of implementation. On the output of `F` another XOR is performed, this time with the second part of the address, and then the parts are swapped; `L` becomes `R` for the next stage and vice versa. It also goes with ensuring the correct bit lengths to match `lsb` and `msb` for the next stage.

After three stages, which suffice to create a pseudorandom result, the intermediate address `IA` is assembled. This address is then treated as a new logical address to be used in the calculation of the physical sector address based on `move_count` and `pos` as described in Section 3.3.2 and shown in Equation (3.1) and Figure 3.1. From these, the physical sector address is obtained, which is the final result of the mapping.

4.2.2 Cycle walking

One issue with the presented randomization based on Feistel network is the fact that the domains of possible sector addresses using `B` bits (addresses between 0 and $2^B - 1$) and valid sector addresses differ, as some sectors are reserved and thus invalid for access. Feistel network can then output a sector address that targets one of the reserved sectors, practically falling outside the valid domain. This can be resolved by what is called cycle walking [13].

When `IA` is outside the valid sector addresses, another round of randomization is performed. The invalid sector address is supplied to the input of the network as `LA` and new `IA` is then obtained. This must be repeated until the network produces a valid `IA`. Cycle walking does introduce a performance penalty, as it will be performed every time such an address is mapped from logical to physical. No caching or any other form of advantageously using previous translations is implemented. Mainly because these would introduce additional storage overhead in the limited embedded device RAM, which is already used, when compared to original wear leveling, for allocating a buffer for per sector erase counts.

In the case presented in this thesis, only a handful of sectors are reserved in the wear-leveled partition, with the vast majority being usable and thus valid. The probability of the necessity of cycle walking is low enough to not pose a significant overhead. More details will be discussed based on the simulation results in Section 5.3.1.

4.3 Implementation and integration with ESP-IDF

`WL_Advanced` has been implemented as a C++ class of the same name and its file is put into a copy of `components/wear_levelling` directory from ESP-IDF. This full copy of the component's sources will require updating to reflect changes made to the component in ESP-IDF upstream, if supporting future versions of the framework will be desired. `WL_Advanced.cpp` is then included in the wear leveling component's `CMakeLists.txt` to involve it in the build process. Configuration option is added to the `Kconfig` file – allowing turning on `WL_ADVANCED_MODE` in `idf.py menuconfig`. Finally instantiating of `WL_Advanced`, based on configuration, is added to the `wear_leveling.cpp` file.

One implementation caveat introduced is that saving the sector number in `pos update records` meant adding the sector as an argument to the `updateWL()` method, now becoming `updateWL(size_t sector)`. This method implements logic surrounding `pos`, `move_count` and `cycle_count`. And crucially, it is called on every erase, as well as "flushing" of wear leveling – by calling `flush()` which is done on unmounting of the partition. But such an operation is not a sector erase, so we are missing a sector number to pass to `updateWL()`. A solution is presented by passing the current value of `pos` to `updateWL()`, since for every dummy sector move, a direct sector erase is performed, which is not recorded by any means. By creating this "fake" record on unmounting, at least some sectors will obtain records reflecting the untracked erases caused by the overhead of moving the dummy sector.

In any case, all projects or applications that previously used the wear leveling component can now turn on `WL_Advanced` and take advantage of its improved leveling and erase count record keeping abilities. **However**, this comes with a few precautions to take into careful consideration. Additional reserved sectors are needed for record keeping, and address randomization will break any current memory accesses, including `fatfs`. For this reason no migration or "upgrade to `WL_Advanced`" process is offered. Currently the recommended approach is to *erase the flash contents*, for example using `idf.py erase-flash`, prior to utilizing `WL_Advanced`.

Chapter 5

Simulation of erase stressed memory lifetime

For the purpose of testing address randomization described in Section 4.2, a rudimentary memory erase stress simulator `wl-sim` was created. The following sections summarize the simulation logic, the variables at both the input and output of the simulation, a few details about its implementation, and, most crucially, the results comparing the original wear leveling mapping algorithm efficiency to the extended version using randomized mapping as implemented in this thesis.

5.1 Simulation logic and parameters

One simulation run proceeds as follows.

1. Initialize counters and thresholds, and generate keys for Feistel network.
2. Simulate erasing a block of $\texttt{B}$ sectors starting from address $\texttt{A} = \texttt{F}_{\texttt{A}}(\texttt{A}_{\texttt{max}})$.
3. If any sector reaches the threshold of $\texttt{PE}_{\texttt{max}}$ erases, declare that the memory has reached its lifetime and proceed to step 4. Otherwise, go back to step 2.
4. Report the values of mapping counters and statistics on wear leveling efficiency of this run.

Now, $\texttt{A}_{\texttt{max}}$ is a fixed value, derived from simulated wear leveled partition size, which was chosen to be 1 MB. $\texttt{PE}_{\texttt{max}}$ denotes the maximum number of program erase cycles a sector can endure and is set to 100 000, as was previously stated to be the erase lifetime of a NOR flash memory cell. The erase block size $\texttt{B}$ can vary and was arbitrarily selected to be in the range of 1 to 50 sectors; see the following Section 5.3.

Each call to the address function $\texttt{F}_{\texttt{A}}$ outputs an address from the range of valid addresses (up to $\texttt{A}_{\texttt{max}}$). The output address can either be a constant (e.g. fixed at $\texttt{A}_{\texttt{max}}/2$) or randomly selected according to a probability distribution. Based on [22], the discrete Zipf distribution was chosen to represent the memory erase access pattern. For the shape of the distribution, see Figure 5.1 (taken from [8]).

C++ implementation of the Zipf distribution was provided by [8] which falls under the MIT license. The skew factor of the distribution was set to 0.99 to properly show the improvement made by address randomization, since the less skewed the distribution,

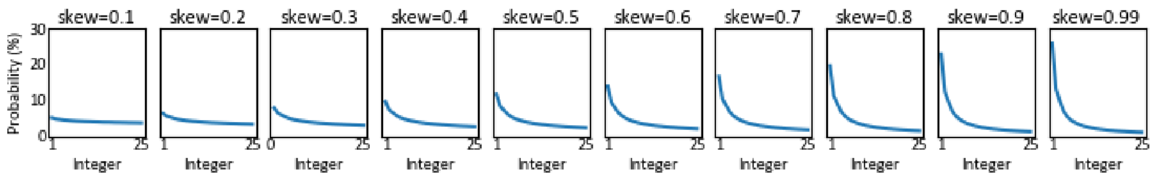


Figure 5.1: Zipfian distribution plots for various skew factors (taken from [8])

the more uniform it becomes, and the uniformity in memory accesses achieves some wear leveling by itself, up to a fully uniform distribution of accesses that does not require wear leveling, as it targets all sectors with the same probability.

Measuring wear leveling efficiency in simulation is done in the same way as in [18], the figure of merit being "Normalized Endurance" (NE), see Equation (5.1).

$$NE = \frac{\texttt{Total Sector Erases Before Lifetime Reached}}{\texttt{PE}_{\texttt{max}} \times \texttt{Num Sectors In Partition}} \times 100\% \tag{5.1}$$

The theoretically perfect case is such that every sector (of which there are `Num Sectors In Partition`) gets erased as much as all the other sectors. This means that a perfectly wear leveled partition can withstand up to $\texttt{PE}_{\texttt{max}} \times$ `Num Sectors In Partition` erases.

Attaining perfect wear leveling is not realistic in practice, as the algorithm itself may not be perfect, or the device could suffer power losses and resets, straying from the ideal run-time behavior. So some sectors will inherently get erased more than others, and the challenge is to keep the difference as small as possible between the least erased and most erased sectors. Once the most erased sector reaches $\texttt{PE}_{\texttt{max}}$, it is declared that the lifetime of the memory is reached. Other sectors will have lower erase counts than $\texttt{PE}_{\texttt{max}}$.

Then the sum of all erase counts of all sectors becomes `Total Sector Erases Before Lifetime Reached`. In perfect wear leveling, the numerator will be the same value as the denominator and the normalized endurance will be 100%. This is the reference value.

In practice, the numerator value will not reach its possible maximum and NE will fall below 100%. Thus, signaling how close to the ideal the algorithm tested and/or run-time configuration is.

5.2 wl-sim implementation

The simulator is implemented as an ESP-IDF project for the ease of compiling using the established build environment with `idf.py`, even though it is to be run only on PC – not on an embedded target – and is split into multiple C++ files (with associated header files of the same names); namely the following.

`WLsim_Flash.cpp` implements mocked behavior of `WL_Advanced` with the same algorithm for mapping between logical and physical sector addresses using unbalanced Feistel network and it also tracks precise erase counts, allowing for analysis of wear leveling efficiency. Most importantly, it contains the `erase_range()` function which simulates erasing the given range of sectors, specified by `start_address` and `size` in bytes as in the real implementation. However, here in simulation both of these values are converted, for the purposes of mapping algorithm and the Feistel network, to sector address representations. Included are also functions for printing the statistics at the end of each simulation run, namely calculated Normalized Endurance and, in the case of employing Feistel network for randomized mapping, the number of cycle walks as well as total calls of the the Feistel mapping function.

`wl_sim_random.cpp` contains the implementations of `constant()` and `zipf()` address functions. `constant()` simply outputs an address equal to half of the maximum address supplied as its argument. The `zipf()` address function uses the previously mentioned Zipf discrete distribution implementation from [8], in the form of a single header file class contained in `dirty_zipfian_int_distribution.h`.

Lastly, `main.cpp` contains argument parsing logic allowing for parameterizing the simulation run (chiefly selecting a mapping algorithm, address function and block size) and the erase stressing logic itself. Some parameters are left unused in the final measurements, these being variable block sizes and a probability for a simulated restart of the device after every erase. Aside from that, a so-called `feistel_test()` is also implemented, which checks that the randomized mapping for all sectors in a given simulated partition is 1:1 and valid. In other words, no two sectors obtained as a result of the unbalanced Feistel network map to the same sector, and, in turn, every sector is the result of said mapping exactly once.

For running the simulation multiple times, up to the 100 averaged runs used for results in the next Section 5.3, a simple Bash script – `run.sh` – was created. It passes the simulation arguments to the simulator binary built by `idf.py` and logs the output of each run to a created log file. Interest is taken in Normalized Endurance and cycle walks out of total erases performed. Upon finishing the final run, it averages the results of all logs recorded by previous runs with the same configuration of simulation parameters. From there, the results were manually copied to `plot.py` which graphs the results (plotting based on Matplotlib documentation example for grouped bar charts[1]).

5.3 Simulation results

Each graph discussed below in this section summarizes a comparison between `base` wear leveling mapping algorithm (the original from ESP-IDF) and `advanced` which uses the address randomization explained in Section 4.2.

Each comparison is parameterized by erase block size in sectors, for which few values have been mostly arbitrarily chosen, and by address function ($\mathtt{F_A}$) which can either be constant or Zipfian – the address of every erase is obtained from discrete Zipf distribution.

Each configuration (block size × address function) has been run 100 times and the reported values are the averages.

As can be seen in Figure 5.2, the `advanced` mapping algorithm employing address randomization does indeed improve wear evenness up to around 3% in this case of erasing blocks of varying sizes at constant address offset. For erase block of a single sector, no improvement is made over the base wear leveling mapping algorithm. Against that, larger block sizes attain greater improvement. Both outcomes are to be expected, as the address randomization targets blocks of sectors which go from physically consecutive mapping – bad leveling – to being randomly spread – good leveling – by the randomized mapping algorithm. For block sizes of 20 sectors and above, `advanced` achieves over 99% of normalized endurance, which theoretically translates to more than 700 000 total additional erase operations the memory could sustain over `base` mapping (the typical total erase lifetime of the simulated memory is just under 25M erases across all sectors and Normalized Endurance improving by up to 3% in good cases).

[1]https://matplotlib.org/stable/gallery/lines_bars_and_markers/barchart.html

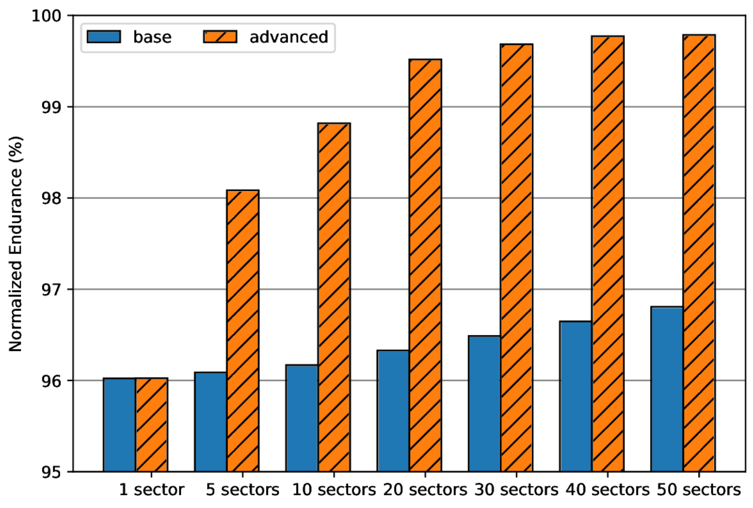


Figure 5.2: Simulation results showing Normalized Endurance comparison between `base` and `advanced` wear leveling algorithms with erase operations at a constant address

Figure 5.3 shows another comparison, this time using Zipfian address function. Being not a constant, the distribution results in better leveling by itself. However, also being heavily skewed, there is an improvement to be made with `advanced` address randomization, that is, up to 1.4%. Larger block sizes once again reach 99% NE and notably see a lesser improvement over the `base` mapping, when compared to previous results in Figure 5.2. This may be attributed to the shape of the distribution, as blocks of sector erasures get spread over the memory in come capacity even without randomized mapping.

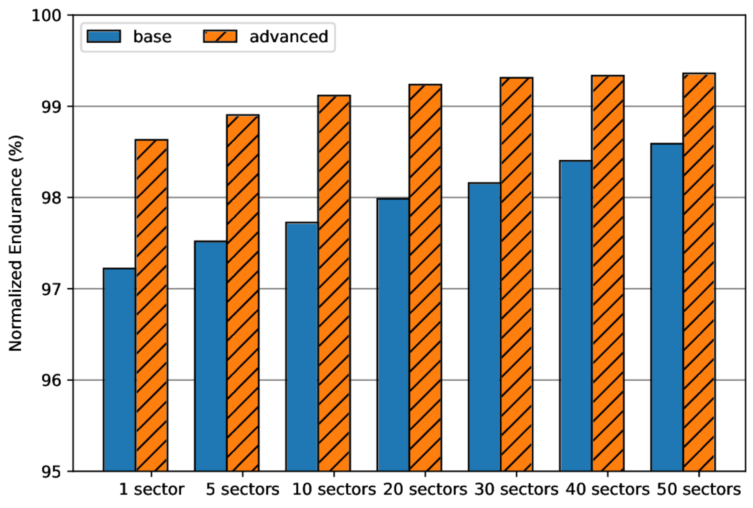


Figure 5.3: Simulation results showing Normalized Endurance comparison between `base` and `advanced` wear leveling algorithms with erase operations at addresses supplied by Zipfian distribution (skew factor 0.99)

Additionally, erase block of a single sector in Figure 5.3 does show an improvement over `base` mapping, where for constant address, in Figure 5.2, no such improvement was measured. Once again, the shape of the Zipfian distribution is heavily skewed towards the start of address space yet with a variance, resulting in accesses in a “fuzzy block” even for a single sector. And from that, `advanced` performs address randomization, which spreads the erases across the full memory.

The presented results clearly show an improvement in wear uniformity caused by the implemented address randomization. Even for a small partition size, where the original mapping already shows good results in the range of 96% NE and above. Curiously, and maybe also due to such a small wear leveled memory size, the `advanced` results surpass those reported by Qureshi. The comparison is not entirely objective, as the workloads are different, yet both implementations utilize a Feistel network in the same way. Where Qureshi averages 97% NE, the results presented in this thesis average almost at 99% NE. Therefore, the addition of address randomization to the mapping algorithm is considered a worthwhile undertaking and a success.

5.3.1 Overheads of randomized mapping

After taking a look at the improvements and benefits of randomized mapping, the drawbacks must also be addressed. Mainly the fact that now every memory access requires an additional translation step on top of the already present calculations based on `pos` and `move_count` as outlined previously. This overhead, in the computational context, consists mainly of logical bitwise operations, so even for three iterations that are necessary for a 3-stage Feistel network, it is deemed negligible.

What may be of greater importance is the necessity of cycle walking, as described in Section 4.2.2. Depending on the number of sectors in wear leveled partition, a varying number of sectors will have to be reserved for storing internal structures and records about erase counts. Say, we have 256 total sectors in a partition with 6 being reserved, leaving us with 250 usable sectors. The chance of cycle walking for any address translation – that is Feistel network “hitting” one of the reserved sectors with its output address – is in such a case $\frac{6}{256}$ or in other words 2.34375%. Now, depending on the memory access pattern and the values of generated keys for Feistel network, it might happen that cycle walking is a rare occurrence, as only few sector addresses result in cycle walks. Or, if unlucky and a frequently accessed location contains a cycle walkable sector, this will present an unavoidable overhead.

For the example of 250 usable sectors out of 256 total sectors, Figure 5.4 shows the – average from 100 simulation runs – probabilities of cycle walking per every erase operation at various sector erase block sizes. Up to blocks of 40 sectors, the chance for cycle walking is less than 1%, at least for this simulation and its chosen parameters. The variance in cycle walking performance based on the generated Feistel network keys is included in the results, as each of the 100 averaged simulated runs created a new set of keys. Based on the results, it can be stated that cycle walking does not, on average, present significant overhead either.

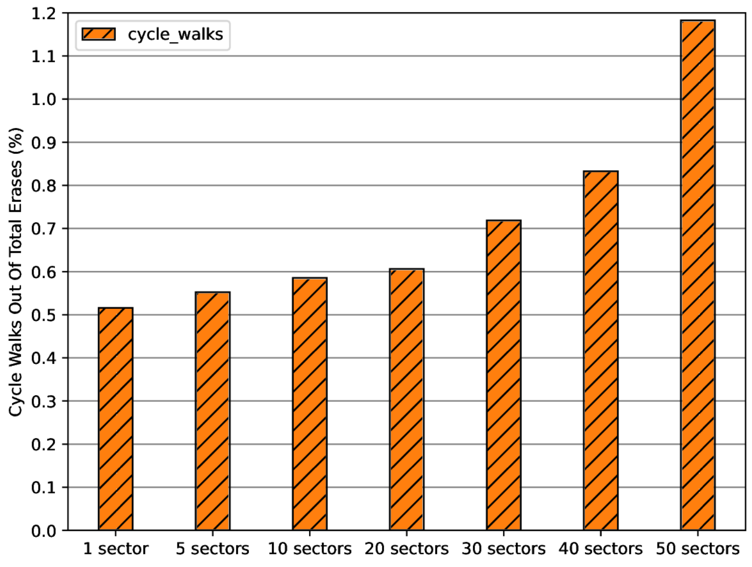


Figure 5.4: Simulation results showing the chance of cycle walking during randomized mapping of every erase operation at address supplied by Zipfian distribution (skew factor 0.99). Values presented are for 250 usable sectors out of 256 total sectors in a simulated partition.

Chapter 6

Design, implementation and testing of monitoring tool

As required by the thesis assignment, the tool has been developed and published on GitHub under the Apache-2.0 license and is available at https://github.com/omnitex/espwlmon. As the conception of the tool is based on Espressif's internal assignment, the entire process of design and implementation was primarily discussed with Ing. Martin Vychodil from Espressif's Brno office, who also did code review on the final form and functionality of the tool. This ensured at least some compatibility between the ideas of what the tool should be capable of and its potential internal use by Espressif.

The following sections outline the tool design, implementation of both its parts, and testing performed on a real embedded device.

6.1 Tool design and functionality

Due to unforeseen circumstances of developing `WL_Advanced`, see Chapter 4, and in turn testing and simulating the address randomization, see Chapter 5, the goal for the tool somewhat degraded from a "flash memory management" standpoint to simply "flash memory monitoring".

The key focus of the tool is then on assessing flash memory lifetime in any given application and on accessing internal structures and values for debug purposes.

The final tool consists of, as specified by the thesis assignment, two parts – an embedded C++ so-called `wlmon` data collector back-end and a Python PC side front-end for presentation and visualization of the obtained wear leveling status, named `espwlmon`.

For passing data between the two parts of the tool, the JSON format has been selected for its universality, readability, and ease of use — namely parsing in Python.

The usage flow of the tool is as follows; an application that is to be monitored is recompiled with `WL_Advanced` enabled and uploaded ("flashed") to an ESP32 embedded device. Now the device and application is used and runs as normal, all the while `WL_Advanced` creates records about the performed erase operations in flash. After some time or when an update on the memory wear status is desired, a special application `wlmon` is flashed to the embedded device replacing only the existing application. Then Python `espwlmon` is to be launched; it attaches to `wlmon` running on the device via a serial connection and displays a Graphical User Interface (GUI) with a listing of structures used by wear leveling and a heatmap visualizing per sector erase counts in the partition.

6.2 wlmon embedded back-end

The embedded wlmon is implemented as a C++ ESP-IDF project and its sources are located in data-collector/wear_levelling/wlmon in the published GitHub repository (or its copy on the included storage media). Similarly to WL_Advanced, it is put into a copy of the wear leveling component directory from ESP-IDF. This allows access to private header files, namely Partition.h, which is used for instantiating a class of the same name, acting as a flash driver and providing access to read, write, and erase functions. A short summary of the logic and workings of wlmon is as follows.

WLmon_Flash.cpp contains a class of the same name, inheriting from WL_Advanced, which implements multiple methods centered around reconstructing internal wear leveling structures from flash as well as the per sector erase counts recorded by WL_Advanced. It includes a twofold detection of original vs. extended wear leveling. First by recovered pos value – as the contents of pos update records are utilized differently, see Section 4.1, each recoverPos() implementation will return zero for the "other" version it cannot read. And second by checking the presence of a saved set of keys for the Feistel network in flash, as these are missing in the original algorithm and the reserved – but not used – space in the state structure is initialized with zeros.

wlmon.cpp serves as a wrapper around WLmon_Flash.cpp. Via a single convenience function entry point of wlmon_get_status(), an instance of WLmon_Flash is created and used for attaching to wear leveling structures in flash and further reconstruction of wear leveling status together with per sector erase counts. All mentioned information is then formatted as JSON and written to a buffer. This buffer is dynamically reallocated to fit the JSON based on the number of sectors in the wear leveled partition, as each can potentially have an erase count to be reported, and the partition attributes are unknown beforehand.

Finally, main.cpp obtains the reconstructed so-called wear leveling status and prints it repeatedly to ensure reliable transmission over serial connection to PC and, in turn, to espwlmon. In the event that an error occurs throughout the wear leveling status reconstruction or JSON assembly process, an error message stating the ESP-IDF specific error code is printed – also in JSON format – instead of the report.

An example of assembled JSON with values from internal use structures and reconstructed per sector erase counts, with an ellipsis for brevity, can be seen in Listing 6.1.

```
{
"wl_mode":"advanced",
"config": {
    "start_addr":"0x0", "full_mem_size":"0x100000", "page_size":"0x1000",
    "sector_size":"0x1000", "updaterate":"0x10", "wr_size":"0x10",
    "version":"0x2", "temp_buff_size":"0x20", "crc":"0x4fb562e0"},
"state":{
    "pos":"0x8","max_pos":"0xf9","move_count":"0x11","access_count":"0x0",
    "max_count":"0x10","block_size":"0x1000","version":"0x2",
    "device_id":"0xf6d5487f","cycle_count":"0x0",
    "feistel_keys":["0xa2","0x35","0x77"],"crc":"0x2c3c5c49"},
"erase_counts":{"0":"5","1":"3","2":"9","3":"5", ...},
}
```

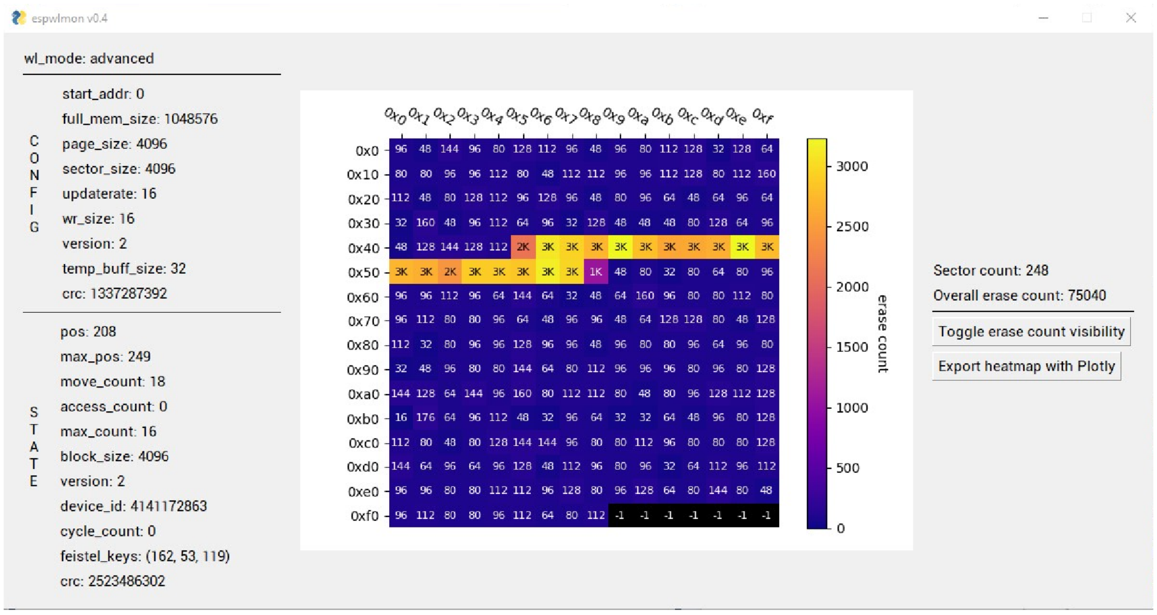
Listing 6.1: An example of a JSON formatted listing of internal wear leveling structures and per sector erase counts – the output of wlmon

6.3 espwlmon PC side front-end

Before launching the GUI, the single file `espwlmon.py` takes an argument specifying the serial port of the embedded device – with `wlmon` running on it – connected to the PC. Upon establishing serial connection, done trivially by using the `serial` Python module, the script waits for two consecutive received JSONs to match each other. This is done as a way of validating the reception, after which the JSON is parsed – once again trivially by the `json` Python module – and with that the `gui()` function gets passed a Python dictionary of the obtained wear leveling status, ready for visualization.

PySimpleGUI[1] was chosen as the Python graphical user interface wrapper for underlying Tkinter, mainly for its simplicity and ease of use, allowing rapid prototyping. As the GUI is not the main focus of this thesis, some corners were cut, making it possibly not up to par with proper graphical front ends. The final look of the GUI can be seen in Figure 6.1.

A static heatmap of per sector erase counts for display in the launched GUI was generated using Matplotlib, using helper functions from its documentation[2]. Such static heatmap is meant for on glance memory wear status report. The presented GUI window in Figure 6.1 shows a heatmap for 1 MB partition of 4 KB sectors. Scaling this static heatmap to properly show larger partitions (2 MB and above) proved to be an implementation challenge and, at the time of submission, is not fully resolved. For such partitions, it is advised to toggle off the erase count numbers shown in the heatmap using the button on the right side of the window or take advantage of the exporting described below.



wl_mode: advanced

start_addr: 0
full_mem_size: 1048576
page_size: 4096
sector_size: 4096
updaterate: 16
wr_size: 16
version: 2
temp_buff_size: 32
crc: 1337287392

pos: 208
max_pos: 249
move_count: 18
access_count: 0
max_count: 16
block_size: 4096
version: 2
device_id: 4141172863
cycle_count: 0
feistel_keys: (162, 53, 119)
crc: 2523486302

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x0	96	48	144	96	80	128	112	96	48	96	80	112	128	32	128	64
0x10	80	80	96	96	112	80	48	112	112	96	96	112	128	80	112	160
0x20	112	48	80	128	112	96	128	96	48	80	96	64	48	64	96	64
0x30	32	160	48	96	112	64	96	32	128	48	48	48	80	128	64	96
0x40	48	128	144	128	112	2K	3K	3K	3K	3K	3K	3K	3K	3K	3K	3K
0x50	3K	3K	2K	3K	3K	3K	3K	3K	1K	48	80	32	80	64	80	96
0x60	96	96	112	96	64	144	64	32	48	64	160	96	80	80	112	80
0x70	96	112	80	80	96	64	48	96	96	48	64	128	128	80	48	128
0x80	112	32	80	96	96	128	96	96	48	96	80	80	96	64	96	80
0x90	32	48	96	80	80	144	64	80	112	96	96	96	80	96	80	128
0xa0	144	128	64	144	96	160	80	112	112	80	48	80	96	128	112	128
0xb0	16	176	64	96	112	48	32	96	64	32	32	64	48	96	80	128
0xc0	112	80	48	80	128	144	144	96	80	80	112	96	80	80	80	128
0xd0	144	64	96	64	96	128	48	112	96	80	96	32	64	112	96	112
0xe0	96	96	80	80	112	112	96	128	80	96	128	64	80	144	80	48
0xf0	96	112	80	80	96	112	64	80	112	-1	-1	-1	-1	-1	-1	-1

Sector count: 248
Overall erase count: 75040

Figure 6.1: `espwlmon` GUI window. On the left: detected mode (base v. advanced) and internal structures listing. In the center: static heatmap of per sector erase counts. On the right: additional information together with buttons for 1) toggling the visibility of erase count numbers in the heatmap, and 2) exporting the heatmap using Plotly

[1]https://www.pysimplegui.org/en/latest/
[2]https://matplotlib.org/stable/gallery/images_contours_and_fields/image_annotated_heatmap.html

For a more informative and exportable report on erase counts, Plotly Express[3] was used for on-command generation of a heatmap in `html` format with embedded JavaScript, allowing scalable and hoverable visualization. An example of such heatmap can be seen in Figure 6.2. This example also follows the 1 MB of 4 KB sectors configuration in a real embedded device, meaning that some sectors are reserved for internal structures and records. For both heatmaps in Figures 6.1 and 6.2, note the `-1` values in the lower right-hand corner; these mark sectors serving as padding (some of them being reserved), making the heatmap either a full square or a rectangle. Otherwise, the numbers denote the erase counts of each sector as reconstructed from records kept by `WL_Advanced`.

The right column of the GUI window in Figure 6.1 also lists an "Overall erase count". If a precise number is shown, the overall erase count has been successfully reconstructed from first, the `pos`, `move_count` and `cycle_count` counters and second, from per sector erase counts kept by `WL_Advanced` and both values are equal; confirming the reconstruction to be valid. Otherwise, if either calculated overall erase count mismatches the other one, an error is shown to draw attention to this fact.

6.4 Real embedded device testing

For the purpose of testing the per sector erase count tracking of `WL_Advanced` in a reasonable time, an example ESP-IDF project for performing memory erase stressing on a real embedded device, under the name of `erase_stress_example`, was implemented (based on an existing wear leveling example[4]).

All erase stressing logic is contained in the `erase_stress_example.c` file. The prescribed definition of partition table, which is to be created in the flash memory of the embedded device, is left untouched from the original example in the `partitions_example.csv` file. For one more time it is mentioned that the wear leveled `data` partition, labeled as `storage`, is set to be 1 MB in size. Lastly, `sdkconfig.defaults` file sets the `WL_ADVANCED_MODE` to be enabled, as the file name suggests, by default when building the project with `idf.py`.

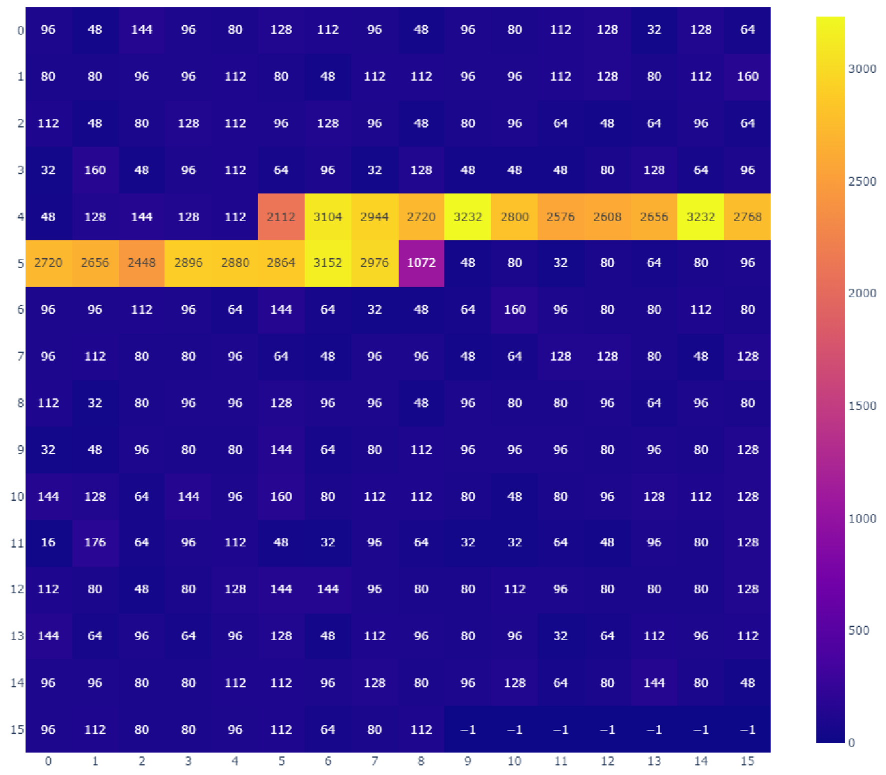
When run on an embedded device, `erase_stress_example` simply repeatedly erases a sector in the middle of the wear leveled `data` partition address space. The erase stressing is organized into *runs*, each consisting of *erase iterations*. At various iteration milestones, a random chance to directly `unmount` and `mount` the wear leveled partition or skip the rest of the current run is introduced to mildly "shake up" the access pattern. In between these erase stress runs, a `fatfs` file read and write test is performed to check the validity of the randomized mapping in terms of file system consistency.

An example of an erase pattern created by a few tens of runs of `erase_stress_example` on `ESP32-C3-DevKitC-02` can be seen in the previously mentioned heatmaps in Figure 6.1 or preferably Figure 6.2. Note that only a single sector is repeatedly erased, yet the erase counts are more spread out, showing the workings of wear leveling. However, the memory access pattern of `erase_stress_example` is created artificially to achieve dynamically looking output in a relatively short time. Any real application over `WL_Advanced` will access the memory in numerous different patterns over time, and erase counts in the heatmap might look less leveled. Yet from the viewpoint of full memory lifetime, at least according to the simulation results of Section 5.3, good wear leveling will be achieved in due time.

[3]https://plotly.com/python/heatmaps/

[4]https://github.com/espressif/esp-idf/tree/master/examples/storage/wear_levelling

The purpose of the implemented example project is then twofold; a showcase application that hastens creating records about erase operations in flash, allowing for visualization, and a testing ground for checking the correctness of randomized mapping. Both showed good and expected behavior, and thus the functionality, of all `WL_Advanced`, `wlmon` and `espwlmon`, is proclaimed compliant and tested.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	96	48	144	96	80	128	112	96	48	96	80	112	128	32	128	64
1	80	80	96	96	112	80	48	112	112	96	96	112	128	80	112	160
2	112	48	80	128	112	96	128	96	48	80	96	64	48	64	96	64
3	32	160	48	96	112	64	96	32	128	48	48	48	80	128	64	96
4	48	128	144	128	112	2112	3104	2944	2720	3232	2800	2576	2608	2656	3232	2768
5	2720	2656	2448	2896	2880	2864	3152	2976	1072	48	80	32	80	64	80	96
6	96	96	112	96	64	144	64	32	48	64	160	96	80	80	112	80
7	96	112	80	80	96	64	48	96	96	48	64	128	128	80	48	128
8	112	32	80	96	96	128	96	96	48	96	80	80	96	64	96	80
9	32	48	96	80	80	144	64	80	112	96	96	96	80	96	80	128
10	144	128	64	144	96	160	80	112	112	80	48	80	96	128	112	128
11	16	176	64	96	112	48	32	96	64	32	32	64	48	96	80	128
12	112	80	48	80	128	144	144	96	80	80	112	96	80	80	80	128
13	144	64	96	64	96	128	48	112	96	80	96	32	64	112	96	112
14	96	96	80	80	112	112	96	128	80	96	128	64	80	144	80	48
15	96	112	80	80	96	112	64	80	112	−1	−1	−1	−1	−1	−1	−1

Figure 6.2: An example of Plotly Express heatmap of per sector erase counts exported from `espwlmon` GUI part of the created monitoring tool (image edited due to dimensions)

Chapter 7

Conclusion

The aim of this Bachelor's thesis was to design and implement a software tool for managing and monitoring flash memory wear leveling in an embedded device, namely the ESP32.

The need for wear leveling, stemming from technological details of flash memory, as well as various approaches that satisfy the need, in varying degrees, have been surveyed. The specific implementation of wear leveling and an embedded flash file system, as found in the Espressif IoT Development Framework (ESP-IDF), has been studied in great depth and throughout compared with other relevant existing solutions and literature. Considerable issues not permitting long-term memory wear monitoring were discovered in said implementation of wear leveling.

Therefore, an extended version of wear leveling was proposed and implemented. It allows long-term tracking of per sector erase counts. On top of that, it improves the mapping algorithm by introducing address randomization based on format-preserving cipher using an unbalanced Feistel network. Said address randomization was tested, and its efficiency was measured using simulated lifetime of memory stressed with erase operations. The simulation results show an improvement of up to 3% in wear uniformity over the original algorithm.

The monitoring tool, consisting of C++ embedded back-end and Python PC side GUI front-end, was designed, implemented, and tested. This includes an example project for erase stressing the memory in a real embedded device and testing the consistency of randomized mapping via an embedded flash file system. Combined with the above-described extended wear leveling, the created tool offers a view at internal structures and a visualization of memory erase wear.

The tool, as well as the extended wear leveling version, has been published on GitHub under the Apache-2.0 license and is available at `https://github.com/omnitex/espwlmon` (the `IBT` branch is archiving the submitted version).

The presented work could be expanded in terms of usage flexibility of the tool. To name a few proposals; Some form of snapshot creation, allowing for building a browsable and visual history of memory wear in a given application, potentially with remaining lifetime prediction. Ability to analyze a memory dump, not just a live embedded device. Investigating the possibility of hooking into a running application and performing real-time monitoring. Regarding the extended wear leveling, it lacks integration with non 4 KB sector modes. All in all, the room for expansion based on desired or required functionality of the tool in a given use case is plentiful.

Bibliography

[1] Andersson, P. *SPI Flash File System (SPIFFS)* [online]. 2017 [cit. 2023-4-12]. Available at: https://github.com/pellepl/spiffs.

[2] Cai, Y., Haratsch, E. F., Mutlu, O. and Mai, K. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. In: *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2012, p. 521–526. DOI: 10.1109/DATE.2012.6176524.

[3] Cappelletti, P., Bez, R., Modelli, A. and Visconti, A. What we have learned on flash memory reliability in the last ten years. In: *IEDM Technical Digest. IEEE International Electron Devices Meeting, 2004*. 2004, p. 489–492. DOI: 10.1109/IEDM.2004.1419196.

[4] Elm chan. *FatFs - Generic FAT Filesystem Module* [online]. 2022 [cit. 2023-04-21]. Available at: http://elm-chan.org/fsw/ff/00index_e.html.

[5] Espressif Systems. *ESP32-C3-WROOM-02 & ESP32-C3-WROOM-02U Datasheet* [online]. 2023 [cit. 2023-04-26]. Available at: https://www.espressif.com/sites/default/files/documentation/esp32-c3-wroom-02_datasheet_en.pdf.

[6] Espressif Systems. *ESP32-WROVER-E & ESP32-WROVER-IE Datasheet* [online]. 2023 [cit. 2023-04-01]. Available at: https://www.espressif.com/sites/default/files/documentation/esp32-wrover-e_esp32-wrover-ie_datasheet_en.pdf.

[7] Flashrom project (open source). *Flash chip ID list* [online]. 2023 [cit. 2023-04-16]. Available at: https://github.com/flashrom/flashrom/blob/master/include/flashchips.h.

[8] Garrison, E. *Zipf and selfsimilar C++ Random Distributions using a fast approximate pow function* [online]. 2022 [cit. 2023-04-21]. Available at: https://github.com/ekg/dirtyzipf.

[9] Haster, C. *Littlefs* [online]. 2022 [cit. 2023-4-12]. Available at: https://github.com/littlefs-project/littlefs.

[10] Hidaka, H. Applications and Technology Trend in Embedded Flash Memory. In: Hidaka, H., ed. *Embedded Flash Memory for Embedded Systems: Technology, Design for Sub-systems, and Innovations*. Cham: Springer International Publishing, 2018, p. 7–27. DOI: 10.1007/978-3-319-55306-1_2. ISBN 978-3-319-55306-1.

[11] Infineon Technologies AG. *AN98521 - Wear Leveling* [online]. 2021 [cit. 2023-04-29]. Available at:

https://www.infineon.com/dgdl/Infineon-AN98521_Wear_Leveling-ApplicationNotes-v06_00-EN.pdf?fileId=8ac78c8c7cdc391c017d07429b0b65b0.

[12] Kim, S. S., Yong, S. K., Kim, W., Kang, S., Park, H. W. et al. Review of Semiconductor Flash Memory Devices for Material and Process Issues. *Advanced Materials*. 2022. DOI: 10.1002/adma.202200659. ISSN 0935-9648.

[13] Liang, S., Zhang, Y., Guo, J., Dong, C., Liu, Z. et al. Efficient Format-Preserving Encryption Mode for Integer. In: *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*. 2017, vol. 2, p. 96–102. DOI: 10.1109/CSE-EUC.2017.203.

[14] Meza, J., Wu, Q., Kumar, S. and Mutlu, O. A Large-Scale Study of Flash Memory Failures in the Field. *SIGMETRICS Perform. Eval. Rev.* New York, NY, USA: Association for Computing Machinery. 2015, vol. 43, no. 1, p. 177–190. DOI: 10.1145/2796314.2745848. ISSN 0163-5999.

[15] Mielke, N. R., Frickey, R. E., Kalastirsky, I., Quan, M., Ustinov, D. et al. Reliability of Solid-State Drives Based on NAND Flash Memory. *Proceedings of the IEEE*. 2017, vol. 105, no. 9, p. 1725–1750. DOI: 10.1109/JPROC.2017.2725738. ISSN 0018-9219.

[16] Pan, Y., Hu, Z., Zhang, N., Hu, H., Xia, W. et al. HNFFS: Revisiting the NOR Flash File System. In: *2022 IEEE 11th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. 2022, p. 14–19. DOI: 10.1109/NVMSA56066.2022.00012.

[17] Pavan, P., Bez, R., Olivo, P. and Zanoni, E. Flash memory cells-an overview. *Proceedings of the IEEE*. 1997, vol. 85, no. 8, p. 1248–1271. DOI: 10.1109/5.622505. ISSN 00189219.

[18] Qureshi, M. K., Karidis, J., Franceschini, M., Srinivasan, V., Lastras, L. et al. Enhancing Lifetime and Security of PCM-Based Main Memory with Start-Gap Wear Leveling. In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. New York, NY, USA: Association for Computing Machinery, 2009, p. 14–23. MICRO 42. DOI: 10.1145/1669112.1669117. ISBN 9781605587981.

[19] Suhail, M., Harp, T., Bridwell, J. and Kuhn, P. Effects of Fowler Nordheim tunneling stress vs. Channel Hot Electron stress on data retention characteristics of floating gate non-volatile memories. In: *2002 IEEE International Reliability Physics Symposium. Proceedings. 40th Annual (Cat. No.02CH37320)*. 2002, p. 439–440. DOI: 10.1109/RELPHY.2002.996685. ISBN 0-7803-7352-9.

[20] Woodhouse, D. JFFS : The Journalling Flash File System. In: *Ottawa Linux Symposium* [online]. 2001 [cit. 2023-04-29]. Available at: https://www.kernel.org/doc/mirror/ols2001/jffs2.pdf.

[21] Wuhan Xinxin Semiconductor Manufacturing Co., Ltd. (XMC®). *3V 32M-BIT SERIAL FLASH MEMORY WITH DUAL/QUAD SPI & QPI Datasheet*

[online]. 2022 [cit. 2023-04-16]. Available at:
https://www.xmcwh.com/uploads/210/XM25QH32C_Ver1.9.pdf.

[22] YANG, Y. and ZHU, J. Write Skew and Zipf Distribution: Evidence and Implications. *ACM Trans. Storage*. New York, NY, USA: Association for Computing Machinery. 2016, vol. 12, no. 4. DOI: 10.1145/2908557. ISSN 1553-3077.

Appendix A

Contents of the included storage media

```
/
├── espwlmon - copy of the GitHub repository at the time of submission
├── esp-idf-v5.0.zip - archive of used ESP-IDF v5.0
└── thesis
    ├── thesis.pdf - thesis version for electronic viewing
    ├── thesis_print.pdf - thesis version for print
    └── src - sources of the thesis as exported from Overleaf
```