

# Programová knihovna pro zápis dat do paměti flash

Bc. Pavel Stodulka

---

Diplomová práce  
2015



Univerzita Tomáše Bati ve Zlíně  
Fakulta aplikované informatiky

---

Univerzita Tomáše Bati ve Zlíně  
Fakulta aplikované informatiky  
akademický rok: 2014/2015

## ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Bc. Pavel Stodulka**  
Osobní číslo: **A13450**  
Studijní program: **N3902 Inženýrská informatika**  
Studijní obor: **Počítačové a komunikační systémy**  
Forma studia: **prezenční**

Téma práce: **Programová knihovna pro zápis dat do paměti flash**  
Téma anglicky: **A Program Library for Writing Data to Flash Memory**

Zásady pro vypracování:

1. Zpracujte literární rešerši na téma způsobu využití paměti typu flash v mikropočítačových systémech se zaměřením na řešení předčasného opotřebení paměti opakovaným přepisem.
2. Navrhněte koncepci a rozhraní programové knihovny umožňující snadné využití paměti typu flash pro ukládání dat v mikropočítačových aplikacích.
3. Realizujte navrženou knihovnu v jazyce C nebo C++ pro zvolený mikropočítač.
4. Ověřte správnou funkci vytvořené knihovny a zhodnoťte její vlastnosti.
5. Vytvořte ukázkovou aplikaci demonstrující využití vytvořené knihovny.

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. BARR, Michael a Anthony J MASSA. Programming embedded systems. 2nd ed. Sebastopol: O'Reilly, 2006, xxi, 301 s. ISBN 978-0-596-00983-0.
2. CATSOULIS, John. Designing embedded hardware. 2nd ed. Sebastopol: O'Reilly, 2005, xvi, 377 s. ISBN 05-960-0755-8.
3. MANN, Burkhard. C pro mikrokontroléry: ANSI-C, kompilátory C, spojovací programy - linkery, práce s ATMEL AVR a MSC-51, příklady programování v jazyce C, nástroje pro programování, tipy a triky. Vyd. 1. Praha: BEN, 2003, 279 s. ISBN 80-730-0077-6.
4. MORTON, Todd D. Embedded microcontrollers. Upper Saddle River, N.J.: Prentice Hall, c2001, x, 694 p. ISBN 01-390-7577-1.
5. PINKER, Jiří. Mikroprocesory a mikropočítače. 1. vyd. Praha: BEN - technická literatura, 2004, 159 s. ISBN 80-730-0110-1.
6. Atmel. Atmel AVR116: Wear Leveling on DataFlash [online]. Atmel Corporation 2012. [cit. 2015-01-15]. Dostupné z: <http://www.adeptotech.com/wp-content/uploads/doc32194.pdf>.
7. PERDUE, Ken. Wear Leveling [online]. 2008 [cit. 2015-01-27]. Dostupné z: [http://www.eettaiwan.com/STATIC/PDF/200808/EETOL\\_2008IIC\\_Spansion\\_AN\\_13.pdf](http://www.eettaiwan.com/STATIC/PDF/200808/EETOL_2008IIC_Spansion_AN_13.pdf).
8. WOODHOUSE, David. JFFS : The Journalling Flash File System [online]. Red Hat, Inc., 2005 [cit. 2015-01-27]. Dostupné z <http://sourceware.org/jffs2/jffs2.pdf>.

Vedoucí diplomové práce:

**Ing. Jan Dolinay, Ph.D.**

Ústav automatizace a řídicí techniky

Datum zadání diplomové práce:

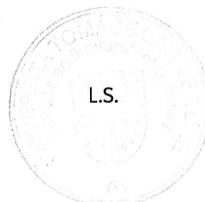
**12. ledna 2015**

Termín odevzdání diplomové práce:

**15. května 2015**

Ve Zlíně dne 6. února 2015

doc. Mgr. Milan Adámek, Ph.D.  
*děkan*



Ing. Miroslav Matýšek, Ph.D.  
*ředitel ústavu*

### **Prohlašuji, že**

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

### **Prohlašuji,**

- že jsem na diplomové/bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně 20.5.2015

  
.....  
podpis diplomanta

## **ABSTRAKT**

Tato práce se zabývá využitím interní paměti flash v mikropočítačích. V teoretické části jsou shrnuty základní informace o mikropočítačích, jeho pamětech a možnostech ukládání dat do flash paměti, a to pomocí ovladače flash paměti nebo speciálních souborových systémů pro flash paměti s rovnoměrným opotřebením paměti. V praktické části je vytvořena knihovna pro zápis do flash paměti, jejímž cílem je eliminovat nerovnoměrné opotřebenění paměti.

Klíčová slova:

Mikropočítač, flash paměť, nerovnoměrné opotřebenění paměti, flash souborové systémy, wear leveling

## **ABSTRACT**

This work deals with utilization of internal flash memory in microcontrollers. In the theoretical part basic information about microcontrollers, its memories and possibilities to write to flash memory are summarized. Methods for writing to flash memory using flash memory driver or specialized flash file system with wear leveling are described. In the practical part is created a library for writing to flash memory, where the goal is to eliminate unequal wear of the memory.

Keywords:

Microcontroller, flash memory, unequal wear memory, flash file system, wear leveling

Tímto bych chtěl poděkovat panu Ing. Janu Dolinayovi, Ph.D. za skvělé rady a připomínky při řešení této práce.

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

# OBSAH

<b>ÚVOD</b> .....	<b>9</b>
<b>I TEORETICKÁ ČÁST</b> .....	<b>10</b>
<b>1 MIKROPOČÍTAČE A JEJICH ARCHITEKTURA</b> .....	<b>11</b>
1.1 STRUKTURA MIKROPOČÍTAČE.....	11
1.2 PROGRAMOVÁNÍ MIKROPOČÍTAČŮ .....	12
1.2.1 Assembler.....	12
1.2.2 Jazyk C .....	12
1.3 DRUHY PROCESORŮ.....	13
1.3.1 CISC.....	13
1.3.2 RISC.....	13
1.4 VÝVOJOVÁ PROSTŘEDÍ .....	13
1.5 ARCHITEKTURA MIKROPOČÍTAČŮ .....	14
1.5.1 Architektura Von Neumann .....	14
1.5.2 Architektura harvardská .....	14
1.5.3 Modifikovaná harvardská architektura .....	15
1.6 PROCESORY S ARCHITEKTUROU ARM .....	15
<b>2 PAMĚTI</b> .....	<b>16</b>
2.1 ENERGETICKY ZÁVISLÉ PAMĚTI.....	16
2.2 ENERGETICKY NEZÁVISLÉ PAMĚTI .....	16
2.2.1 ROM.....	16
2.2.2 PROM .....	16
2.2.3 EPROM .....	17
2.2.4 EEPROM.....	17
2.2.5 Flash .....	17
<b>3 FLASH PAMĚŤ</b> .....	<b>18</b>
3.1 KONSTRUKCE FLASH PAMĚTI.....	18
3.1.1 SLC .....	19
3.1.2 MLC .....	19
3.1.3 NOR architektura .....	19
3.1.4 NAND architektura .....	20
3.1.5 Porovnání NAND a NOR pamětí.....	20
3.2 OPOTŘEBENÍ FLASH PAMĚTI .....	21
3.2.1 Wear Leveling .....	21
3.2.2 Flash Translation Layer.....	21
3.2.3 Způsoby řešení opotřebení flash paměti .....	23
<b>II PRAKTICKÁ ČÁST</b> .....	<b>27</b>
<b>4 POUŽITÝ MIKROPOČÍTAČ</b> .....	<b>28</b>
4.1 PARAMETRY MIKROPOČÍTAČE .....	28
<b>5 PRINCIP KNIHOVNY PRO ZÁPIS DO FLASH PAMĚTI</b> .....	<b>29</b>
5.1 FORMÁT ZÁZNAMU.....	29
5.2 VYHLEDÁVÁNÍ ZÁZNAMŮ V PAMĚTI.....	31
5.2.1 Vyhledávání bez pomocného pole proměnných .....	31

5.2.2	Vyhledávání s pomocným polem proměnných.....	31
5.3	VRSTVY KNIHOVNY .....	33
5.4	PRINCIP ZÁPISU DO FLASH PAMĚTI.....	34
5.5	PRINCIP ČTENÍ Z FLASH PAMĚTI.....	34
5.6	PRINCIP INICIALIZACE KNIHOVNY.....	34
5.7	PRINCIP DEFRAGMENTACE PAMĚTI .....	35
5.8	PODPOROVANÉ DRUHY DAT .....	36
<b>6</b>	<b>STRUKTURA KNIHOVNY .....</b>	<b>37</b>
6.1	NASTAVITELNÉ KONSTANTY A CHYBOVÉ KÓDY .....	39
6.2	VRSTVA OVLADAČE FLASH PAMĚTI .....	40
6.3	VRSTVA OVLADAČE VIRTUÁLNÍ FLASH PAMĚTI.....	41
6.4	VRSTVA HAL.....	42
6.5	VRSTVA LOGIKY KNIHOVNY .....	43
6.5.1	Funkce Mount .....	43
6.5.2	Funkce pro zápis dat do paměti.....	43
6.5.3	Funkce pro čtení dat z paměti .....	47
6.5.4	Funkce pro defragmentaci paměti.....	48
<b>7</b>	<b>VYTVORENÍ NOVÉHO PROJEKTU .....</b>	<b>52</b>
7.1	PROJEKT SE ZÁPISEM DO FLASH PAMĚTI .....	52
7.2	PROJEKT SE ZÁPISEM DO VIRTUÁLNÍ FLASH V RAM.....	53
<b>8</b>	<b>UKÁZKOVÉ PROJEKTY .....</b>	<b>54</b>
8.1	PROJEKT Č. 1: ZÁPIS DO FLASH PAMĚTI.....	54
8.2	PROJEKT Č. 2: ZÁPIS DO VIRTUÁLNÍ FLASH V RAM.....	56
	<b>ZÁVĚR .....</b>	<b>59</b>
	<b>SEZNAM POUŽITÉ LITERATURY .....</b>	<b>60</b>
	<b>SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK.....</b>	<b>64</b>
	<b>SEZNAM OBRÁZKŮ .....</b>	<b>66</b>
	<b>SEZNAM TABULEK.....</b>	<b>67</b>
	<b>SEZNAM ZDROJOVÝCH KÓDŮ .....</b>	<b>68</b>
	<b>SEZNAM PŘÍLOH.....</b>	<b>69</b>

## ÚVOD

Flash paměti jsou v dnešní době velice rozšířeným druhem pamětí. Najdeme je v USB flash discích, paměťových kartách a postupně nahrazují i klasické pevné disky (HDD) pracující na magnetickém principu. Mezi jejich hlavní výhodu patří cena a rychlost.

Oproti svým konkurentům ale mají jeden zásadní problém, a tím je omezená životnost, která plyne z konstrukce flash pamětí. Každá paměťová buňka má omezený počet zápisů, po který funguje korektně. Jakmile její životnost překročíme, již není schopná nést informace.

Výrobci flash pamětí na tento problém zareagovali a upravili princip vnitřního zápisu do flash paměti, který se snaží zapisovat do paměťových buněk tak, aby měly všechny paměťové buňky vždy stejné, nebo alespoň podobné, opotřebení. Paměť se tak při používání opotřebovává rovnoměrně a nehrozí, že jedna část paměti bude za hranicí své životnosti a přitom do druhé části paměti ještě nebyl proveden jediný zápis. Takovéto paměti (USB flash disk nebo paměťové karty) jsou vybaveny řadičem, jehož úkolem je zajistit rovnoměrné opotřebení flash paměti.

Existují ale i takové flash paměti, které žádný elektronický obvod pro rovnoměrné opotřebování nemají. U takových pamětí nezbyvá nic jiného, než problém rovnoměrného opotřebení řešit softwarově. Bylo proto vyvinuto několik souborových systémů, které dokáží řadič flash paměti nahradit.

Právě mikropočítače mají svou vnitřní flash paměť připojenou napřímo, bez elektronického obvodu pro rovnoměrné opotřebení. Uživatel tak vždy zapisuje na určité adresy a tím opět snižuje životnost flash paměti díky nerovnoměrnému opotřebení.

Cílem této práce je proto vytvořit programovou knihovnu pro práci s flash pamětí, která dokáže alespoň z části nahradit plnohodnotný flash souborový systém a tím umožnit rovnoměrné opotřebení paměti u mikropočítačových aplikací, které využívají interní flash paměť pro ukládání dat.

## **I. TEORETICKÁ ČÁST**

# 1 MIKROPOČÍTAČE A JEJICH ARCHITEKTURA

Mikropočítač je elektronický obvod umístěný na jednom čipu, spolu se všemi podpůrnými obvody potřebnými pro jeho správnou funkčnost. Funkce, kterou mikropočítač vykonává, je dána programem.

## 1.1 Struktura mikropočítače

Každý mikropočítač se skládá z aritmeticko-logické jednotky (ALU), řadiče, paměti a vstupně/výstupních rozhraní.

Funkcí řadiče je řídit chod celého mikropočítače, ALU provádí všechny výpočty a mezivýsledky svých výpočtů ukládá do paměti. V paměti je mimo dat uložen i program, podle kterého se řídí řadič. Veškerá komunikace s okolím prochází skrz vstupně/výstupní rozhraní, které je tvořeno piny. Několik pinů se sdružuje do portů.

Stejně jako běžný osobní počítač, i mikropočítač potřebuje pro svou bezvadnou funkčnost několik různých typů pamětí, které můžeme rozdělit do tří skupin – paměť programu, paměť dat a operační paměť. [1]

### - Paměť programu

V paměti programu jsou uloženy jednotlivé strojové instrukce, které postupně provádí procesor. Jedna z výhod speciální paměti pro program je okamžité provádění instrukcí po připojení napájení.

### - Paměť dat

Paměť dat slouží pro dlouhodobé uchovávání informací v mikropočítači.

### - Operační paměť

Slouží pro krátkodobé uchovávání proměnných a mezivýsledků procesoru.

Vstupně/výstupní rozhraní můžeme rozdělit do několika základních skupin [2]:

- digitální vstup/výstup
- analogový vstup/výstup, který je opatřen AD nebo DA převodníkem
- komunikační rozhraní (např. SPI, I2C, UART)

## 1.2 Programování mikropočítačů

Programování je proces, při kterém se počítači předávají příkazy (instrukce) v takovém pořadí a s takovými operandy, aby ve výsledku počítač vykonával námi požadovanou činnost. Programování zahrnuje kromě samotného psaní programu (instrukcí) jeho návrh, ladění a testování. [3]

### 1.2.1 Assembler

Assembler (česky také Jazyk symbolických adres) je nižší, strojově orientovaný programovací jazyk. Program psaný v assembleru je složen z instrukcí, kde každá instrukce obsahuje pokyn, co má procesor udělat. Cílem assembleru je usnadnit psaní programu tím, že uživatel místo strojového kódu používá pojmenované instrukce. Kód se tak díky assembleru zpřehlednil. Nevýhodou tohoto programovacího jazyka je nepřenositelnost kódu, kde daný program jde většinou spustit jen na procesoru, pro který byl vytvořen. [4], [5]

```
mov    eax, 1
mov    ebx, eax
mov    ecx, dword ptr dwValue

mov    ah, al
mov    ch, cl
mov    bx, ax
ret
```

*Zdrojový kód 1: Ukázka programování v assembleru [5]*

### 1.2.2 Jazyk C

Programovací jazyk C vznikl v roce 1972 a patří do skupiny vyšších programovacích jazyků. Hlavními rysy jazyka C jsou snadná přenositelnost, efektivita kódu a možnost tvoření programů i na systémové úrovni. Program napsaný v jazyce C může mít svou rychlost vykonávání podobnou rychlosti stejnému programu napsaného v assembleru. [6]

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("hello world\n");
    return 0;
}
```

*Zdrojový kód 2: Ukázka programu v jazyce C*

### 1.3 Druhy procesorů

Dalším možným dělením mikropočítačů může být podle procesoru a instrukční sady, kterou používá.

#### 1.3.1 CISC

Mikropočítače typu CISC (Complex Instruction Set Computer) obsahují velké množství instrukcí, které dokáží vykonat i složité operace. Zpracování instrukcí probíhá ve více strojových cyklech. [7]

#### 1.3.2 RISC

Mikropočítače RISC (Reduced Instruction Set Computer) mají redukovaný instrukční soubor. Omezení vychází z předpokladu, že ne všechny instrukce (v porovnání s CISC) bude mikropočítač vykonávat stejně často. Mikropočítač má tak k dispozici jen omezený počet základních instrukcí a pomocí skládání základních instrukcí může vytvořit instrukce složitější. [7]

### 1.4 Vývojová prostředí

Ve vývojových prostředích, které může mít každý výrobce jiné, se vytváří program. Vývojové prostředí můžeme rozdělit na proprietární, kde každý výrobce ke svým mikropočítačům dodá software, pomocí kterého ho může uživatel naprogramovat. Zástupci takového prostředí může být Kinetis Design Studio od Freescale nebo Atmel Studio od firmy Atmel.

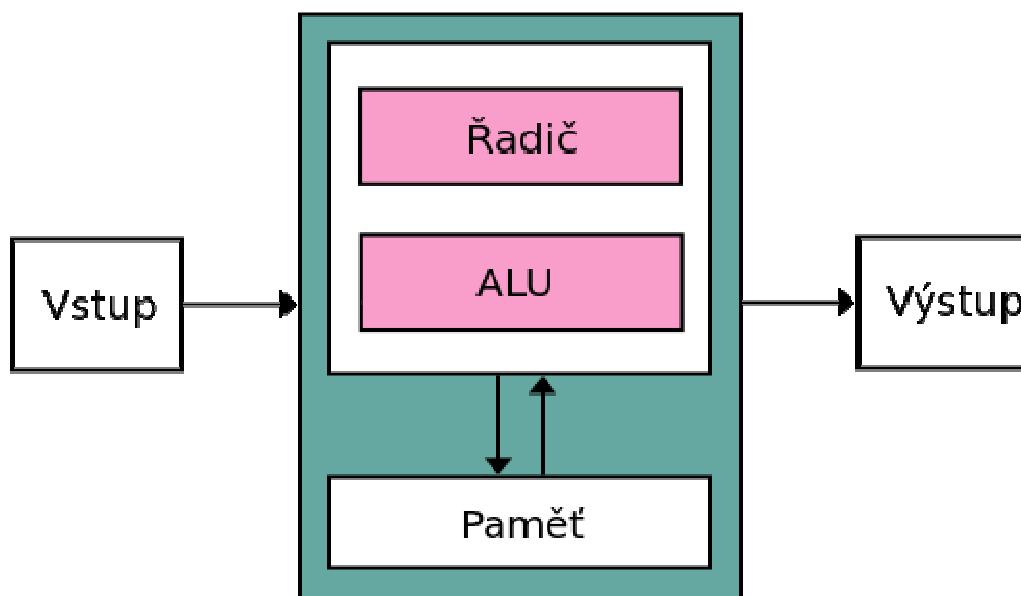
Další skupinou jsou vývojové prostředí s otevřeným kódem, kam můžeme zařadit Eclipse nebo populární Arduino, které podporuje širokou škálu mikropočítačů a obsahuje vlastní nadstavbovou knihovnu se kterou je programování ještě o něco jednodušší.

## 1.5 Architektura mikropočítačů

Jednočipové mikropočítače mohou být postaveny na dvou základních architekturách, a to na architektuře Von Neumann nebo Harvardské architektuře.

### 1.5.1 Architektura Von Neumann

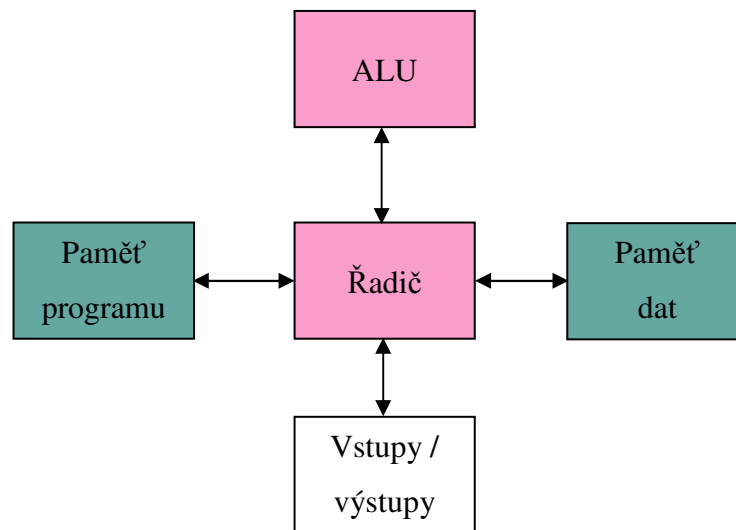
Ve Von Neumannově architektuře jsou program i data uloženy ve stejném paměťovém prostoru. Jedna společná paměť má výhodu stejných instrukcí pro přístup k programu nebo datům. Jedna paměť vede ke zjednodušení vnitřního zapojení mikropočítače, protože nám stačí jen jedna datová sběrnice pro komunikaci s pamětí. Na druhou stranu můžeme v jeden okamžik přenášet po sběrnici buď instrukci programu nebo data. Nikdy oba zároveň. [8]



Obrázek 1: Von Neumannova architektura [9]

### 1.5.2 Architektura harvardská

Harvardská architektura používá oproti architektuře Von Neumann dvě paměti. Jednu paměť pro uložení programu a druhou paměť pro ukládání dat. Nevýhodou je, že potřebujeme vytvořit dvě datové sběrnice, jednu pro data, druhou pro program. Toto řešení komplikuje návrh čipu. Mezi výhody můžeme zařadit, že program nelze přepsat daty, díky dvěma různým pamětem můžeme načítat v jeden okamžik jak instrukce programu, tak i data. Vykonávání instrukcí je tak rychlejší než u architektury Von Neumann. [8]



Obrázek 2: Harvardská architektura [10]

### 1.5.3 Modifikovaná harvardská architektura

Modifikovaná harvardská architektura je podobná klasické harvardské architektuře. Rozdíl je, že modifikovaná harvardská architektura neodděluje striktně paměť programu a paměť dat, i když ke každé paměti si zachovává vlastní datovou sběrnici.

Nejčastější modifikace zahrnuje oddělení datové a instrukční cache paměti, které mají společný adresový prostor. Tato modifikace umožňuje pracovat s instrukcemi jako s daty. Tento typ architektury je rozšířen u moderních procesorů ARM a procesorů X86. [11]

## 1.6 Procesory s architekturou ARM

Procesory s ARM architekturou a RISC instrukční sadou jsou moderní procesory, které dnes najdeme v mnoha aplikacích. Mezi výhody ARM můžeme zařadit nízkou energetickou náročnost a možnost zakoupení licence na jádro procesoru a vyrábění vlastního ARM procesoru, který může navíc obsahovat např. grafickou kartu nebo modul pro komunikaci po síti. Tyto procesory můžeme najít mj. v mobilním telefonu, tabletu, nebo netbooku. [12]

## 2 PAMĚTI

Počítačové paměti jsou elektronické obvody, které jsou schopné uchovávat informace. Používají se jak pro dočasné, tak i trvalé uložení programů nebo dat. Paměti můžeme rozdělit do dvou základních skupin, a to na energeticky závislé a energeticky nezávislé paměti. [13]

### 2.1 Energeticky závislé paměti

U těchto typů paměti, též zvaných volatilní, je důležité udržovat paměti pod napětím. Jedině za této podmínky je schopná si informace udržet. Po odpojení napájení paměť informace ztrácí.

### 2.2 Energeticky nezávislé paměti

Paměti energeticky nezávislé (nevolatilní), nepotřebují být připojeny k napájení, aby udržely informace. Když se do nich informace (data) zapíše, tak jsou schopné si tato data pamatovat i po několik let, bez nutnosti být připojené k napájení. Nevolatilní paměti můžeme dělit do několika skupin, a to podle druhu zápisu a mazání. [14]

#### 2.2.1 ROM

U paměti typu ROM (Read Only Memory) můžeme provádět pouze čtení. Zápis informace je proveden při výrobě a tyto informace již nelze měnit.

#### 2.2.2 PROM

Do paměti typu PROM (Programmable Read Only Memory) můžeme zapsat informace pouze jednou. Je stejná jako paměť ROM s tím rozdílem, že ROM programuje výrobce a PROM uživatel.

Při programování paměti se provede nevratná změna hodnoty paměťové buňky. Prázdná paměť (ještě neprogramovaná) má všechny své buňky nastaveny do hodnoty logické 0. Při programování se v požadované paměťové buňce přepálí programovací spojka a tím se hodnota paměťové buňky změní na logickou 1. Programuje se vždy jen jedna paměťová buňka a to zvýšeným napětím 10 V. Po dobu programování musí být paměť deaktivována. K programování se používá programátor – speciální elektronické zařízení. [15]

### 2.2.3 EPROM

Paměť typu EPROM (Erasable PROM) se již dá speciálním způsobem smazat a znovu přeprogramovat. Paměť se může smazat ultrafialovým zářením.

### 2.2.4 EEPROM

Paměť EEPROM (Electrically Erasable PROM) se dá mazat elektricky, bez nutnosti použít ultrafialové záření. Tuto paměť je možné mazat buňku po buňce a má omezený počet zápisů.

### 2.2.5 Flash

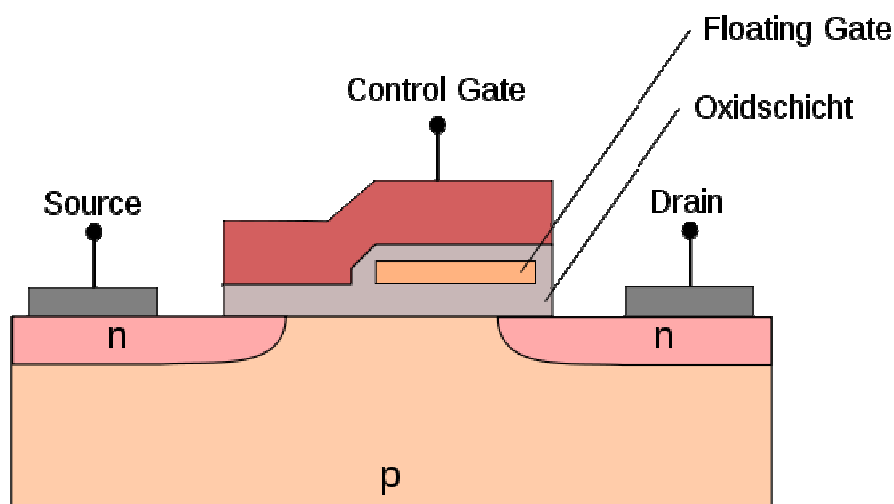
Flash paměť patří do skupiny pamětí EEPROM. Je organizována po blocích a jednotlivé operace jsou také prováděny nad jednotlivými bloky – při přeprogramování není nutné změnit (mazat) celou paměť, ale jen ty bloky, které potřebujeme změnit. Stejně jako EEPROM i flash paměť má omezený počet zápisů.

### 3 FLASH PAMĚŤ

Jak již bylo psáno dříve, paměť flash je elektricky nezávislá a mazatelná paměť organizována po blocích, kde lze každý blok programovat a mazat samostatně. Flash paměť najdeme v paměťových kartách (SD, xD Picture card, a jiné), USB flash discích nebo také v mikropočítačích. [16]

#### 3.1 Konstrukce flash paměti

Samotná paměť je složena z paměťových buněk, které jsou tvořeny unipolárními tranzistory s plovoucím hradlem. V základním stavu je v hradle uložená log. 1. Informace určené k zápisu do paměti se ukládají do plovoucího hradla (Floating Gate), které se nachází mezi řídicím hradlem (Control Gate) a substrátem. Plovoucí hradlo je od okolí izolováno vrstvou oxidu křemičitého ( $\text{SiO}_2$ ). Zápis probíhá pomocí tunelování z řídicího hradla. Elektrony, které se dostanou do plovoucího hradla, jsou v něm díky izolaci uvězněny a za normálních okolností se nemohou nikde přesunout. Při čtení informace se na řídicí hradlo přivede čtecí napětí a čtená informace závisí na napětí mezi Source a Drain. [17]



Obrázek 3: Struktura paměťové buňky [16]

### 3.1.1 SLC

Technologie SLC (Single Level Cell) znamená, že každá paměťová buňka je schopná uchovávat pouze 1 bit informace – můžeme do ní uložit log. 1 nebo log. 0. Plovoucí hradlo tedy „nabíjíme“ elektrony v rozsahu buď nula elektronů nebo maximum elektronů. [18]

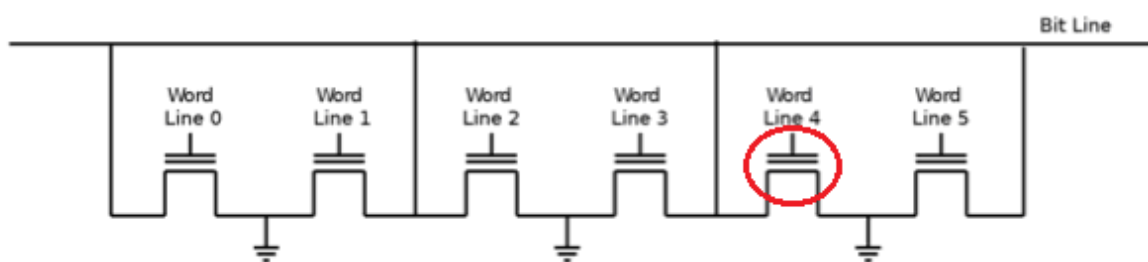
### 3.1.2 MLC

Technologie MLC (Multi Level Cell) znamená, že každá buňka je schopná si uchovávat více než jeden bit informace. Může se ukládat dva bity informace (celkem čtyři stavy - napětí) nebo tři bity informace (celkem osm stavů - napětí). Díky MLC dokážeme získat vyšší informační hustotu. [19]

Z důvodu, že si u této technologie buňka dokáže zapamatovat více bitů informací, tak plovoucí hradlo „nabíjíme“ elektrony po určitých krocích, tedy čtyři kroky u dvou bitů nebo devět kroků při třech bitech informací. MLC technologie tak nabízí větší kapacitu pamětí oproti SLC, která je ale vykoupená větší možnou chybovostí a pomalejším čtením/zápisem. [18]

### 3.1.3 NOR architektura

Paměťové buňky jsou zapojeny do mřížky, je možné každou buňku adresovat (a programovat) samostatně, mazání probíhá po blocích. [19]



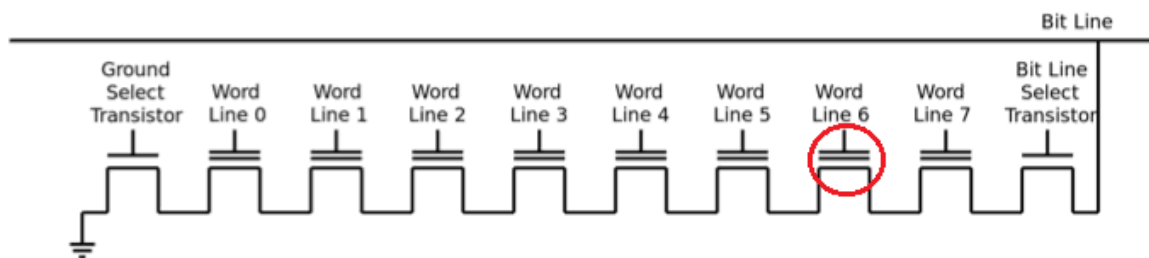
Obrázek 4: NOR flash paměť, v kroužku jedna paměťová buňka [20]

Na obrázku (Obrázek 4) lze vidět strukturu NOR flash paměti. Každou buňku (tranzistor) je možné adresovat zvlášť.

### 3.1.4 NAND architektura

U NAND architektury nelze adresovat jednotlivé paměťové buňky, adresují se jejich seskupení do stránek (page). Každá stránka je tvořena několika v sérii zapojenými paměťovými buňkami. Několik stránek tvoří blok. Informace se tak zapisují do paměti po stránkách a mažou se po blocích. Díky seskupení do stránek se redukuje počet vodičů [18], tím se lépe využije plocha čipu a technologie NAND má tak až o 45 % vyšší informační hustotu než NOR.

Paměť typu NAND je vybavena pomocným registrem, který má stejnou velikost jako stránka. Veškeré operace čtení a zápisu tak probíhá přes pomocný registr. Při čtení se celá stránka zkopíruje do pomocného registru a z něho dále k uživateli. Zápis probíhá podobně – nejprve se zapíše do pomocného registru a z něj dále do paměti.



Obrázek 5: NAND flash paměť, v kroužku jedna paměťová buňka [20]

Na obrázku (Obrázek 5) lze vidět strukturu NAND flash paměti. Na obrázku je zobrazena jedna stránka, která je tvořena v sérii zapojenými paměťovými buňkami.

Paměť typu NAND je většinou vybavena řadičem, který obstarává požadované operace nad paměť. Díky řadiči se tak může provádět realokace vadných stránek nebo bloků. Díky realokaci se zvyšuje životnost paměti a zároveň se snižuje jejich cena, protože tyto paměti může výrobce prodávat i s vadnými buňkami (stránkami). [19]

### 3.1.5 Porovnání NAND a NOR paměti

Díky řadiči se NAND paměť nedá použít jako náhrada EPROM či EEPROM. Pro svoji nízkou cenu a rychlé čtení/zápis se používá u USB klíčenek, paměťových karet nebo pevných disků SSD.

Díky adresaci jednotlivých buněk u pamětí typu NOR se tyto paměti dají použít jako náhrada na EPROM nebo EEPROM. Kvůli chybějícímu řadiči se snadno připojuje k procesoru. [19]

## 3.2 Opotřebení flash paměti

Flash paměť má omezený počet zápisů - je to dáno její strukturou. Při každém zápisu do paměti dojde k poškození izolační vrstvy plovoucího hradla. Postupem času dojde k takovému poškození, že izolační vrstva oxidu křemičitého není schopná dostatečně izolovat plovoucí hradlo.

### 3.2.1 Wear Leveling

Aby se zajistilo, že životnost všech sektorů (bloků) flash paměti skončí ve stejnou dobu, je nutné mazání sektorů provádět rovnoměrně po celé flash paměti. Rovnoměrnému mazání flash paměti se říká „Wear Leveling“ (rovnoměrné opotřebování).

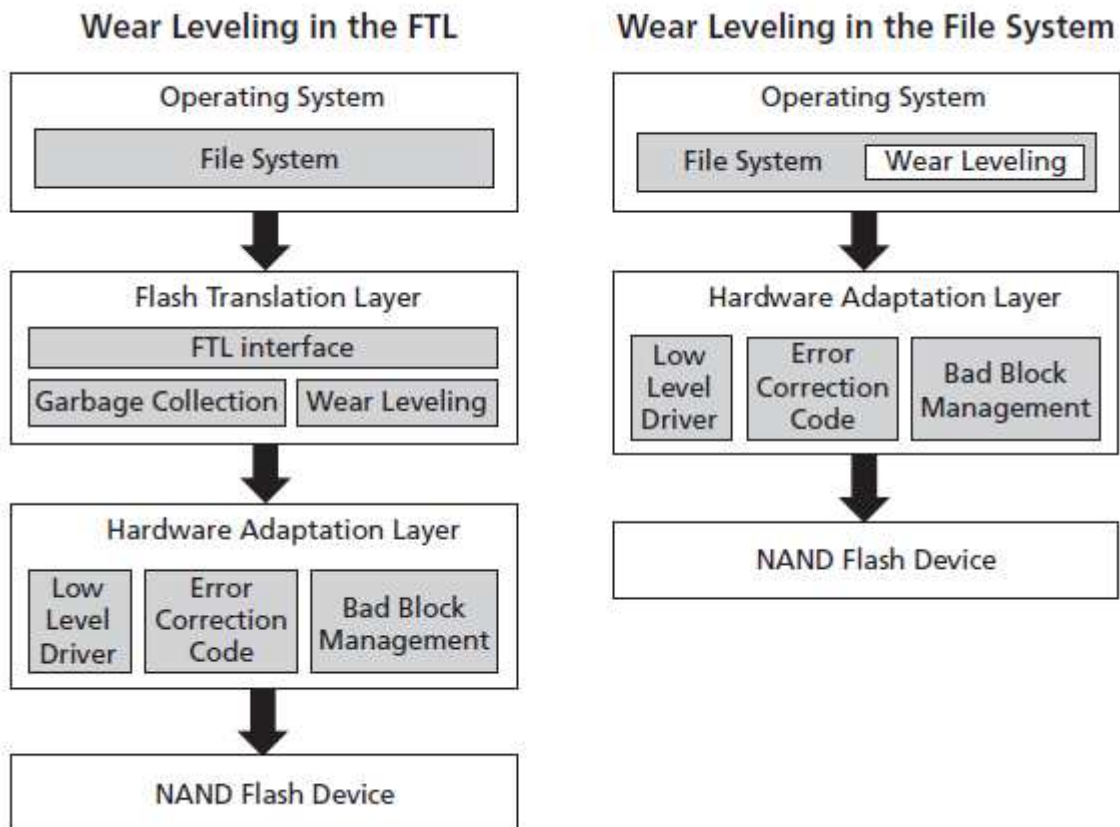
Wear Leveling tak zajišťuje rovnoměrné opotřebení tím, že rozkládá jednotlivé zápisy po celé paměti, aby nedocházelo k ukládání neustále na stejné místo. Kdyby se ukládalo neustále na jedno a to samé místo, paměť by na tomto místě byla brzo za hranicí své životnosti, a to i přesto, že do zbytku paměti nemuselo být ještě ani jednou zapsáno.

Wear Leveling by dnes měly obsahovat všechny NAND flash paměti (např. USB flash disky, paměťové karty). Existují tři druhy Wear Leveling: žádný, statický a dynamický. Dynamický Wear Leveling střídá zápisy pouze mezi volnými sektory paměti, statický Wear Leveling přesouvá všechny data na disk, a to včetně těch, které se tak často nemění. Pokud např. uložíme jeden soubor, který dále needitujeme a uložíme ještě druhý soubor, který budeme neustále přepisovat, u dynamického Wear levelingu bude první soubor uložen pořád na stejné adrese. U statického Wear Levelingu se ale po určitém čase přesune na jiné místo v paměti i první soubor, a to i přesto, že v něm nebyla provedena žádná změna. [21]

### 3.2.2 Flash Translation Layer

Flash Translation Layer (FTL) je přídavná softwarová vrstva mezi systémem souborů (klasickým, např. NTFS, ext3) a NAND flash pamětí. FTL umožňuje operačnímu systému

provádět čtení a zápis do flash paměti naprosto stejně, jako by přistupoval k běžnému, rotačnímu, pevnému disku (HDD). Dále provádí překlad adres z virtuálních na fyzické. Wear leveling je implementován do FTL. [22]



Obrázek 6: Wear Leveling jako součást FTL nebo systému souborů [22]

Na obrázku (Obrázek 6) vidíme dvě možná schémata použití Wear Leveling. V levém schématu lze vidět, že operační systém používá běžný systém souborů, který komunikuje s FTL (FTL obsahuje Wear Leveling) a dále FTL komunikuje se samotným ovladačem flash paměti.

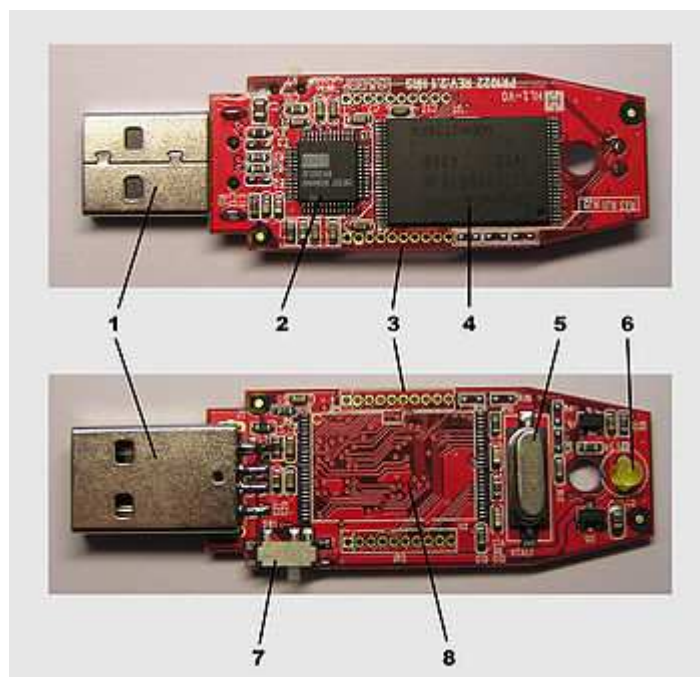
V pravém schématu (Obrázek 6) lze vidět, že operační systém používá speciální systém souborů pro flash paměti, který umí Wear Leveling. Systém souborů dále komunikuje s ovladačem flash paměti napřímo.

### 3.2.3 Způsoby řešení opotřebení flash paměti

Existují dva základní způsoby, jak řešit opotřebení flash paměti. První z nich, hardwarový, používá řadič a druhý způsob, softwarový, používá souborový systém speciálně navržený pro flash paměti.

- **Běžné flash paměti s řadičem**

Flash paměti používané v běžném, každodenním životě jako USB flash disk nebo paměťová karta obsahují kromě samotné flash paměti řadič, jehož úkolem je provádět zápis do paměti tak, aby se paměť opotřebovávala rovnoměrně.



Obrázek 7: USB flash disk [23]

Na obrázku (Obrázek 7) vidíme strukturu flash USB disku. Nejzajímavějším na obrázku jsou pozice č. 2 a 4. Na pozici č. 4 najdeme flash paměťový čip, se kterým komunikuje řadič (pozice č. 2). Veškeré požadavky na práci s pamětí jsou prováděny přes řadič.

- **Flash paměti bez řadiče**

Abychom dosáhli rovnoměrného opotřebení i u paměti bez řadiče, bylo vyvinuto několik souborových systémů, určených přímo pro paměti flash, které dokáží řadič nahradit.

- JFFS

The Journalling Flash File System (JFFS) vychází z Linuxových souborových systémů. Jedná se o záznamově strukturovaný systém souborů (Log-Structured File System - LFS). Záznamy jsou uloženy na flash čipu lineárně, za sebou, skrz veškerý dostupný paměťový prostor. V systému souborů JFFS existuje pouze jeden typ záznamu – uzlu - struktura „struct jffs\_raw\_inode“. Každý takový uzel (záznam) obsahuje hlavičku, ve které najdeme číslo inode a metadata souborového systému. Každý uzel může nést různé množství dat a obsahuje navíc svou verzi, která je číslována takovým způsobem, že každý nově zapsaný uzel (záznam) musí mít verzi vyšší než předchozí záznamy patřícímu stejnému inode.

JFFS zapisuje pomocí své struktury do paměti pokaždé, když dojde ke změně obsahu a je dostatek paměti. Pokud paměť dochází, JFFS musí začít „recyklovat“ staré verze uzlů (záznamů). Můžeme říct, že staré verze uzlů se nacházejí na začátku paměti a nové, aktuální uzly jsou na konci paměti. Tato recyklace (Garbage Collection) se může spouštět systémově (spouští jádro operačního systému, a to i tehdy, když „recyklace“ ještě není potřeba) nebo ji může spustit uživatelský proces, pokud dochází místo a není už kam zapisovat.

Cílem „recyklace“ je uvolnit první sektor (blok) flash paměti. Pokud při průchodu narazí na zastaralý záznam, nic se neděje a jde dál. Pokud narazí na aktuální záznam, tak z aktuálního záznamu udělá zastaralý – vytvoří se nový záznam (uzel), s vyšší verzí na konci paměti. Po takovémto průchodu jedním sektorem flash paměti můžeme sektor smazat, protože obsahuje pouze staré verze záznamů (uzlů).

V paměti musí zůstat vždy dostatek volného místa, aby Garbage Collection měl kam převádět záznamy (uzly), které brání smazání daného sektoru. Při „recyklaci“ je nejlepší situace, kdy daný sektor (blok) paměti obsahuje pouze staré (neaktuální) záznamy. Nejhorší situace ale nastane, když chceme smazat sektor paměti, který obsahuje jen platné záznamy – všechny tyto záznamy se musí zkopírovat na konec paměti. [24]

JFFS provádí perfektní Wear Leveling, kde je každý sektor paměti smazán naprosto stejně krát jako ostatní sektory. To také znamená, že sektory jsou mazány častěji než je potřeba. [25]

### ○ JFFS2

JFFS2 je druhá verze systému JFFS. Hlavní rozdíly oproti JFFS jsou [25]:

- JFFS2 umožňuje zapsat různé druhy záznamů (uzlů), ne jenom jeden typ. Každý typ záznamu má stejnou hlavičku.
- JFFS2 si udržuje seznam prázdných a plných sektorů paměti.
- JFFS2 podporuje kompresi a tvrdé odkazy (hard links)

### ○ YAFFS

YAFFS (Yet Another Flash File System) je dalším systémem souborů určených pro NAND flash paměti. Stejně jako JFFS, i tento systém souborů je záznamově strukturovaný. Existují dvě verze YAFFS, kde druhá verze má oproti první verzi rozšířenou funkcionalitu. V následujících řádcích se budeme věnovat druhé verzi.

Struktura YAFFS je následující. YAFFS ukládá do paměti objekty (což jsou např. samotná data, složky nebo odkazy) a každý objekt má mimo jiné své unikátní identifikační číslo (ID) a informaci o délce dat v Bytech.

Garbage collection („recyklace“) probíhá tak, že se pomocí heuristického algoritmu vybere blok, který se bude „recyklovat“. Garbage collection má dvě možnosti:

- Pokud máme k dispozici spoustu prázdných sektorů, pak se provede pouze „pokus o recyklaci“ těch bloků, které mají málo aktuálních záznamů. Tento styl se nazývá pasivní Garbage collection.
- Pokud máme málo prázdných sektorů, pak se musí „recyklovat“ více sektorů s více aktuálními záznamy. Tento styl „recyklace“ se jmenuje agresivní Garbage Collection.

Pokud provádíme agresivní „recyklaci“, pak je celý blok „recyklován“ v jednom průchodu Garbage collection. Při pasivní „recyklaci“ se počet stejných záznamů redukuje v několika průchodech, což snižuje zátěž systému.

Cílem heuristického algoritmu je pomocí střídání pasivního a agresivního Garbage Collection rozložit „recyklaci“ do delšího časového úseku, aby neprobíhalo vše v jeden okamžik a tím nebyl systém nadměrně zatížen. Ve výsledku se tak zvýší průměrný výkon systému.

YAFFS byl použit i u NOR flash pamětí a dokonce i jako souborový systém paměti RAM. [26]

- **LogFS**

LogFS vznikl jako reakce na zdlouhavé připojování zařízení („mount“) se systémem souborů JFFS2. Doba připojování měla být u 1 GB USB flash disku až 15 minut. Rozložení dat ve flash paměti je uloženo ve formě stromové struktury. Při každé změně na disku je stromová struktura aktualizována a uložena do flash paměti jako soubor (Inode file). Aktualizace začíná od změněného souboru a postupuje směrem nahoru až ke kořenu stromu. Proto se této stromové struktuře říká „wandering tree“ („putující“ strom). Po připojení flash paměti s tímto systémem souborů tak stačí pouze najít aktuální stromovou strukturu. [27]

- **UBIFS**

UBIFS má podobnou strukturu záznamů jako JFFS a navíc používá stromovou strukturu „wandering tree“ k uchování „mapy“ záznamů ve flash paměti. Stromová struktura uchovává ve svých „listech“ odkazy na jednotlivé záznamy uložené v paměti.[28]

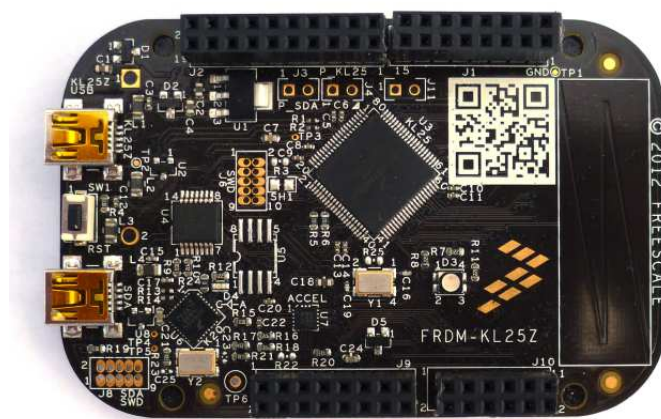
- **Flash paměť mikropočítačů**

Flash paměť mikropočítačů je připojena přímo k procesoru a adresuje se přímo. Proto zde neexistují žádné mechanismy rovnoměrného opotřebení a právě z tohoto důvodu vznikla tato práce.

## **PRAKTICKÁ ČÁST**

## 4 POUŽITÝ MIKROPOČÍTAČ

Cílem této práce je vytvořit univerzální programovou knihovnu, kterou bude možno použít na různých mikropočítačových platformách. V první fázi je ovšem nutno ji implementovat a otestovat alespoň na jedné platformě. K tomu byla vybrána vývojová deska FRDM-KL25Z od společnosti Freescale. Jedná se o dostupnou vývojovou desku, která je osazena mikropočítačem Freescale MKL25Z128VLK4 s moderní architekturou ARM. [2]



Obrázek 8: Freescale FRDM-KL25Z

### 4.1 Parametry mikropočítače

Mikropočítač má následující parametry

- 128 KB Flash
  - 128 sektorů
    - Každý po 1 KB
  - Nejmenší mazatelná jednotka 1 sektor (1 sektor = 1024 B)
  - Nejmenší jednotka pro čtení i zápis jsou 4 B
- 16 KB SRAM
- 66 vstupů/výstupů (I/O)

## 5 PRINCIP KNIHOVNY PRO ZÁPIS DO FLASH PAMĚTI

Celá knihovna je z důvodu snadné přenositelnosti na jiné mikropočítače napsána v jazyce C a pracuje s vnitřní flash pamětí mikropočítače. Na začátku si uživatel definuje část paměti flash, která bude pod správou knihovny. Tato definice se provádí po celých sektorech flash paměti. Jako první se zvolí začátek přidělené paměti a počet sektorů (musí být sudý počet sektorů).

Knihovna zapisuje jednotlivá data do paměti po záznamech, kde každý záznam obsahuje hlavičku a data. Záznam má svůj jedinečný identifikátor, záznamy jsou do paměti ukládány za sebou, každá nová verze záznamu je do paměti zapsána za předchozí záznamy. V paměti tak můžeme najít mnoho verzí jednoho záznamu, kde nejaktuálnější je vždy ten poslední (s nejvyšší adresou). Uživatel si na začátku práce s knihovnou také definuje maximální počet různých záznamů, které půjdou do paměti uložit.

Protože se časem paměť zaplní velkým množstvím stejných záznamů, pouze s jinými daty, je potřeba provést defragmentaci. Jejím cílem je promazat všechny staré (neplatné) záznamy a ponechat pouze aktuální.

Z tohoto důvodu je přidělená paměť rozdělena na dvě poloviny. Zápis záznamů se provádí vždy do jedné poloviny, druhá polovina zůstává prázdná. Zaplnění poloviny paměti pro zápis záznamů je uživateli indikováno a ten provede defragmentaci, při které se všechny nejaktuálnější záznamy zkopírují do druhé poloviny paměti a následně se první polovina smaže. Zápis záznamů po defragmentaci pokračuje vždy za defragmentované záznamy, tzn. polovina paměti musí vždy zůstat prázdná.

### 5.1 Formát záznamu

Do paměti flash se vždy ukládá záznam, který obsahuje hlavičku a data. Hlavička záznamu je tvořena:

- Identifikátorem záznamu
  - o Číslo o velikosti 2 B, jedinečný identifikátor záznamu (jeho „jméno“)
- Velikostí záznamu
  - o Číslo o velikosti 1 B. Udává, kolik místa v paměti (v bytech) záznam zabírá

Data záznamu mají různou velikost, kterou si uživatel omezí v hlavičkovém souboru.

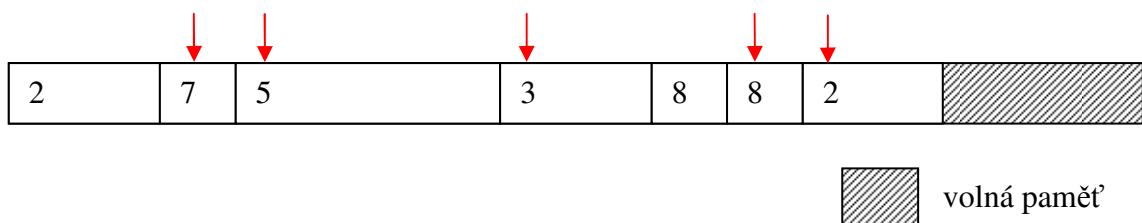
Můžeme tedy psát, že jeden záznam v paměti vypadá následovně:

- **Identifikátor záznamu**, 2B
- **Velikost záznamu**, 1B
- **Data**, 1B až povolené maximum (nastavení v hlavičkovém souboru, výchozí hodnota 17 B)

ID	velikost	data
----	----------	------

Obrázek 9: Formát jednoho záznamu

Záznamy jsou v paměti ukládány za sebou (Obrázek 10), tak jak přijdou požadavky na uložení z hlavního programu. V paměti se tak může vyskytovat několik záznamů se stejným identifikátorem za sebou. Protože jsou všechny záznamy uloženy za sebou, platný je vždy ten nejnovější záznam, tedy ten, který uložíme naposled.



Obrázek 10: Ukázka uložení záznamů v paměti

Na obrázku (Obrázek 10) je pro větší přehlednost jednotlivých záznamů zobrazeno pouze jeho identifikační číslo (ID). Z obrázku lze také vyčíst, že jednotlivé záznamy mohou mít proměnlivou délku, podle množství dat, které chceme do paměti uložit. Záznamy s červenou šipkou značí ty aktuální, záznamy bez červené šipky jsou zastaralé (neplatné).

Maximální počet různých záznamů (podle jejich identifikačního čísla) si uživatel navolí při připojování knihovny do svého projektu. Jediným omezením je velikost identifikátoru (což jsou 2B) a jeho maximální hodnota (0xFFFF), která znamená, že na tomto místě v paměti není nic zapsáno a nesmí se jako identifikátor použít.

## 5.2 Vyhledávání záznamů v paměti

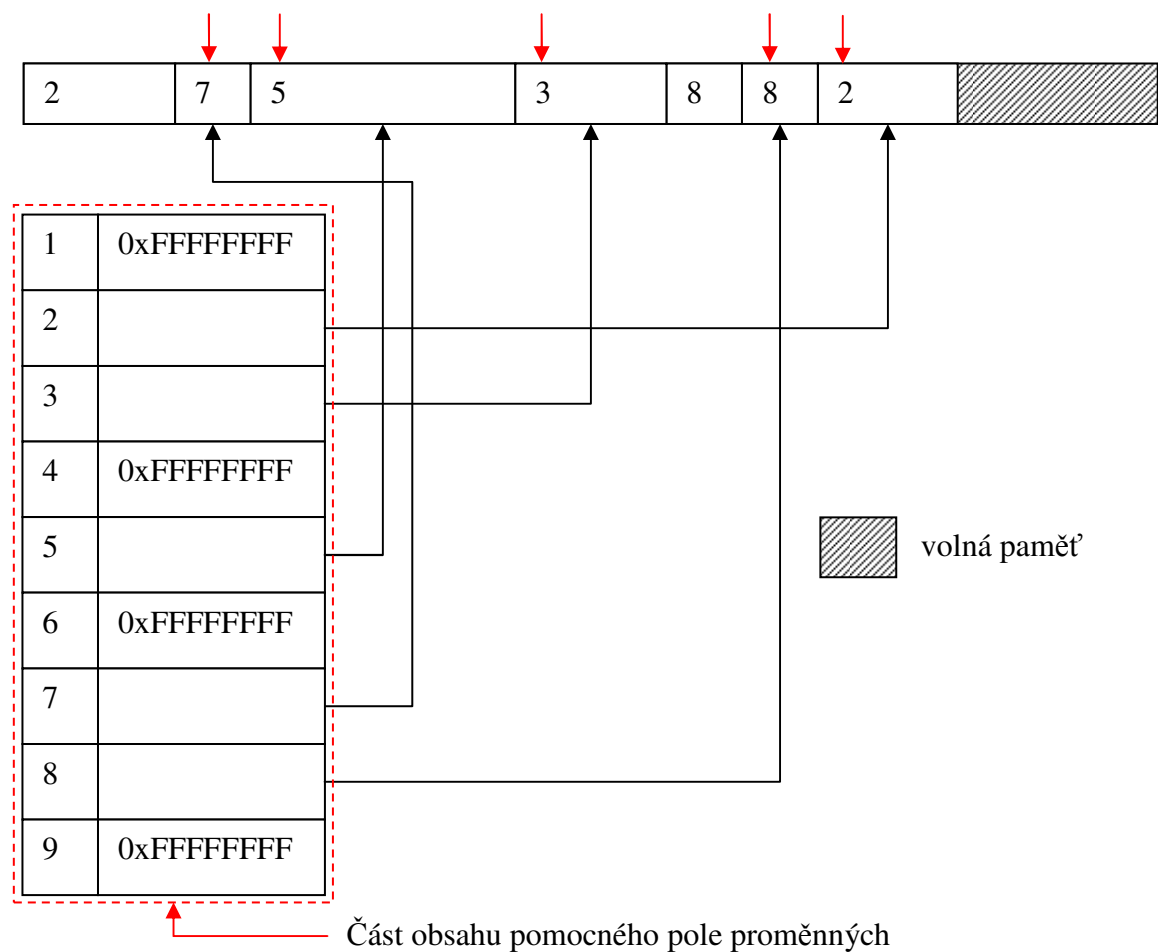
Knihovna je vybavena dvojitým hledáním záznamů v paměti. Záleží na uživateli, který z nich si zvolí.

### 5.2.1 Vyhledávání bez pomocného pole proměnných

Při požadavku na získání hodnoty určitého záznamu z paměti se projde celá přidělená paměť záznam po záznamu a uživateli se vrátí nejaktuálnější hodnota (ta, na kterou se narazilo naposledy)

### 5.2.2 Vyhledávání s pomocným polem proměnných

Při výběru této metody se vytvoří pomocné pole proměnných o velikosti maximálního počtu různých záznamů v paměti. Indexem do pomocného pole proměnných je identifikátor záznamu a výsledkem je adresa nejaktuálnějšího záznamu v paměti. Při zapnutí mikropočítače a následném připojení knihovny do programu uživatele se pouze jednou projde celá přidělená flash paměť a pomocné pole proměnných se naplní nejaktuálnějšími adresami jednotlivých záznamů. Pokud daný identifikátor uživatel nepoužívá, jeho hodnota v pomocném poli proměnných je 0xFFFFFFFF (tato hodnota koresponduje s prázdnou pamětí flash, která má stejnou hodnotu).



Obrázek 11: Funkce pomocného pole proměnných

Na obrázku (Obrázek 11) lze vidět princip pomocného pole proměnných. Pokud uživatel dané ID záznamu používá, na jeho pozici se nachází odkaz do paměti, kde se nachází aktuální záznam. Pokud uživatel dané ID nepoužívá, na jeho pozici je hodnota 0xFFFFFFFF.

Při čtení z paměti se pouze podle identifikátoru záznamu (který je indexem do pomocného pole proměnných) z pomocného pole proměnných získá adresa aktuální verze záznamu a z této adresy se přečte hodnota.

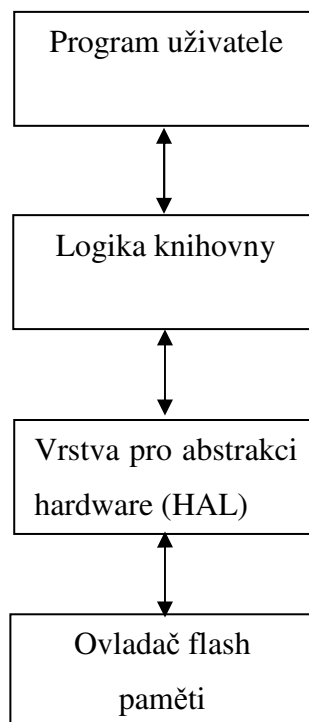
Při zápisu nového záznamu je vždy pomocné pole proměnných aktualizováno o adresu, na kterou se nový záznam zapsal.

Výhodou tohoto řešení je rychlé čtení z paměti a defragmentace, kdy se nemusí oproti metodě vyhledávání bez pomocného pole proměnných procházet celá paměť

při každém příchozím požadavku na čtení. Nevýhodou je náročnost na paměť RAM, ve které je pomocné pole alokováno.

### 5.3 Vrstvy knihovny

Knihovna je z důvodu zjednodušení a možné migrace na jiné mikropočítače rozdělena do několika vrstev, z nichž každá má svůj úkol.



Obrázek 12: Struktura knihovny

**Ovladač flash paměti** je vrstva kódu specifická pro daný mikropočítač. Může být dodán výrobcem, případně vytvořen na základě dokumentace výrobce či ukázkových programů. Musí poskytovat funkce pro zápis a čtení flash paměti a mazání sektorů.

Pomocí vrstvy **HAL** je prováděna komunikace mezi logikou knihovny a ovladačem flash paměti. Tato vrstva je součástí knihovny právě kvůli snadné přenositelnosti knihovny mezi mikropočítači. Při migraci na jiný mikropočítač se do této vrstvy pouze přidají funkce z ovladače flash paměti daného mikropočítače.

**Logika knihovny** obsahuje samotné funkce pro zápis dat do paměti, jejich čtení a defragmentaci. Právě funkce z této vrstvy volá **uživatel ve svém programu**.

#### 5.4 Princip zápisu do flash paměti

Při zápisu si knihovna zjistí, do jaké poloviny paměti se již zapisovalo a následně v této části obsazené paměti najde první volné místo. Pokud je volné místo dostatečně velké, aby se do něj záznam vlezl, provede jeho zápis.

Knihovna si zároveň uchovává v globální proměnné informaci o tom, na jaké adrese se zápisem skončila. Při opětovném zápisu se tak nehledá první volné místo v paměti, ale pokračuje se na adrese, která je uložena v této globální proměnné.

#### 5.5 Princip čtení z flash paměti

Pokud je nastavena verze čtení bez pomocného pole proměnných, pak se prochází celá polovina obsazené paměti s tím, že poslední nalezený záznam je aktuální a tato hodnota se vrátí uživateli.

Pokud použijeme variantu s pomocným polem proměnných, pak se při příchozím požadavku na čtení dat z paměti mikropočítač podívá do pomocného pole proměnných (podle indexu, který zadá uživatel), z tohoto pole získá adresu nejnovějšího záznamu ve flash paměti a hodnotu záznamu (data) vrátí uživateli.

#### 5.6 Princip inicializace knihovny

Inicializace knihovny (např. při zapnutí mikropočítače) je důležitá hlavně pro variantu s pomocným polem proměnných. Cílem inicializace knihovny je projít veškerou přidělenou paměť pro knihovnu zápisu do flash paměti. Při procházení paměti se postupně aktualizuje pomocné pole proměnných o nejnovější záznamy a jejich adresy do paměti flash. Ve výsledku máme veškeré informace o jednotlivých záznamech uloženy v pomocném poli proměnných.

## 5.7 Princip defragmentace paměti

Protože se všechny záznamy ukládají v paměti za sebou, časem se stane, že záznam má kromě své nejnovější verze mnoho verzí starších, neaktuálních. Přidělenou flash paměť nám tak zabírá velké množství duplicitních záznamů, které už ke své práci nepotřebujeme. Řešením je provést defragmentaci, kdy se paměť flash přeorganizuje tak, že se zachovají pouze aktuální verze záznamů a zbytek (neaktuální záznamy, starší verze záznamů) se smaže.

Defragmentace je řešena následujícím způsobem. Uživatel může knihovně pro záznam do flash paměti přidělit určitý počet sektorů. Je nutné, aby tento počet sektorů byl sudý. Přidělená paměť se rozdělí na polovinu (podle sektorů, např. čtyři přidělené sektory se rozdělí na dva a dva sektory). Knihovna své záznamy vždy zapisuje jen do jedné poloviny přidělené paměti. Jakmile se daná polovina přidělené paměti zaplní, je nutné provést defragmentaci.

Pokud je zapnuta varianta s pomocným polem proměnných, pak defragmentace tohoto pole využívá. Projde se celé pomocné pole proměnných a všechny nejaktuálnější záznamy se zkopírují do druhé poloviny přidělené paměti.

Pokud je varianta s pomocným polem proměnných vypnuta, pak se pro každý možný identifikátor záznamu projde fragmentovaná polovina paměti a nejnovější záznam se zkopíruje do druhé, defragmentované půlky paměti.

Po tomto kroku máme obsah paměti následující. Jedna polovina přidělené paměti obsahuje všechny záznamy, tedy i ty, které se v paměti nacházejí vícekrát. Druhá polovina paměti (defragmentovaná) obsahuje pouze nejaktuálnější verze záznamů. Nyní se provede poslední krok a smažeme polovinu přidělené paměti, která obsahuje duplicitní záznamy.

Pokud po defragmentaci přijde požadavek na zápis dalších záznamů do paměti, zápis proběhne do defragmentované poloviny paměti, přímo za poslední záznam. Při zápisu tak vždy zůstává jedna polovina paměti prázdná.

Toto řešení defragmentace s rozdělením paměti na dvě poloviny má následující výhody:

- Máme jistotu, že se defragmentace musí vždy podařit – nemůže přetéct paměť. Co se vejde do jedné poloviny paměti, se musí vejít i do druhé poloviny.
- Přidělená paměť je opotřebovávaná rovnoměrně.

- Přehledný kód (v knihovně pro zápis do flash paměti)

Toto řešení má ale i svou nevýhodu - vždy máme polovinu paměti prázdnou.

## 5.8 Podporované druhy dat

Knihovna podporuje zápis a čtení několika druhů dat:

- Data o velikosti 1 B (uint8\_t, byte)
- Data o velikosti 2 B (uint16\_t, word)
- Data o velikosti 4 B (uint32\_t, double word)
- Data o velikosti 8 B (uint64\_t, double long)
- Data typu String
  - o Maximální velikost si volí uživatel, výchozí hodnota je 17 B (jeden záznam má maximální velikost 20B – 3 B hlavička, 17 B data)

## 6 STRUKTURA KNIHOVNY

Knihovna je složena ze tří hlavních souborů, a to:

- ovladač pro práci s pamětí flash (soubory „kl25\_drv“, záleží na výrobci mikropočítače)
- logika knihovny pro zápis do paměti (soubory „flashMem\_storage“)
- mezivrstva (soubory „flashMem\_hal“), která dělá prostředníka mezi logikou knihovny („flashMem\_storage“) a ovladačem pro práci s pamětí flash („flash\_drv“)

Při tvoření této práce byl pro zjednodušení testování vytvořen nový ovladač flash paměti, jehož cílem je vytvořit virtuální paměť flash v RAM mikropočítače (soubory fir\_drv). Při testování tak nedocházelo ke znehodnocování flash paměti opakovaným zápisem, protože se všechny data zapisovaly do RAM, ve které si uživatel vytvořil dostatečně velké pole. Dalším výhodou byla variabilita virtuální flash, protože si uživatel může nastavit libovolnou velikost sektoru.

Tato práce je složena z následujících souborů:

- kl25\_drv.c
  - o soubor, který se liší podle druhu mikropočítače, obsahuje funkce pro obsluhu paměti flash (driver pro flash zvoleného mikropočítače)
- kl25\_drv.h
  - o hlavičkový soubor k driveru pro flash
- fir\_drv.c
  - o soubor, který slouží pouze pro ověření správné funkce knihovny na vyhrazené paměti v RAM (fir = Flash In RAM). Struktura je naprosto totožná s kl25\_drv.c, zápis je ale do RAM
- fir\_drv.h
  - o hlavičkový soubor, totožný s kl25\_drv.h
- flashMem\_storage.c

- soubor má na starosti logiku ukládání dat do paměti flash
- flashMem\_storage.h
  - hlavičkový soubor k flashMem\_storage.c, obsahuje chybové kódy knihovny
- flashMem\_hal.c
  - mezivrstva mezi flashMem\_storage.c a kl25\_drv.c (fir\_drv.c). Při přechodu na jiný mikropočítač se pouze přidají funkce daného mikropočítače do tohoto souboru
- flashMem\_hal.h
  - hlavičkový soubor k flashMem\_hal.c
- config.h
  - Konfigurační soubor, uživatel si v něm nastavuje:
    - Maximální počet různých záznamů, které jdou do paměti uložit
    - Jestli chce použít pomocné pole proměnných nebo ne
    - Základní parametry flash paměti (velikost sektoru, nejmenší zapisovatelná jednotka)
    - Přidělený paměťový prostor paměti flash
    - Maximální velikost jednoho záznamu

Pokud bychom chtěli migrovat z jednoho mikropočítače na jiný, dodá se pouze ovladač flash paměti daného mikropočítače a provede se změna v mezivrstvě „hal“, kde se přidají volané funkce z nově přidaného ovladače flash paměti. Knihovna je tak snadno přenositelná.

## 6.1 Nastavitelné konstanty a chybové kódy

V následujících dvou tabulkách (*Tabulka 1*, *Tabulka 2*) jsou vypsány všechny konstanty, které může uživatel změnit (v souboru *config.h*) a chybové kódy knihovny.

*Tabulka 1: Tabulka nastavitelných konstant*

Název konstanty	Význam	Poznámka
FLASH_MAXIMUM_VARIABLES	Maximální počet různých záznamů v paměti flash	Povolený číselný rozsah záznamů je 0 až (FLASH_MAXIMUM_VARIABLES - 1)
USE_VARIABLE_LIST	Použití pomocného pole proměnných	0 – zakázáno 1 – povoleno
FLASH_ADDRESS_ALIGNMENT	Zarovnání adresy flash paměti	
FLASH_SECTOR_SIZE	Velikost sektoru flash paměti	
FLASH_WRITE_SIZE	Nejmenší zapisovatelná jednotka	
FLASH_START_SECTOR	První přidělený sektor knihovně	
FLASH_COUNT_SECTOR	Počet přidělených sektorů knihovně	
STORAGE_RECORD_MAXIMUM_SIZE	Maximální velikost jednoho záznamu	Celý záznam, i s hlavičkou. Nesmí být větší než 254.
PLATFORM	Platforma mikropočítače	KL25 - flash paměť FIR - virtuální flash v RAM

Tabulka 2: Tabulka chybových kódů

<b>Chybový kód</b>	<b>Význam</b>
LIBRARY_OK	Vše proběhlo v pořádku, bez chyb
LIBRARY_FULL_MEMORY	Paměť je plná
LIBRARY_DEFRAGMENT_ERROR	Chyba při defragmentaci
LIBRARY_ID_OUT_OF_RANGE	ID záznamu mimo povolený rozsah
LIBRARY_ID_DO_NOT_EXISTS	Záznam s požadovaným ID neexistuje
LIBRARY_MEMORY_READ_ERROR	Chyba při čtení z paměti
LIBRARY_DATA_LENGTH_NOT_ALLOWED	Požadované data nelze zapsat pro jejich velikost
LIBRARY_MOUNT_ERROR	Chyba při připojení (Mount) knihovny

## 6.2 Vrstva ovladače flash paměti

Tato vrstva je implementována v souborech kl25\_drv.h a kl25\_drv.c. Tyto soubory jsou ovladači pro používání flash paměti a závisí na mikropočítači. Zpravidla obsahují tři základní funkce:

- mazání flash paměti
- zápis do flash paměti
- čtení z flash paměti

```
/* Public functions of the driver */
uint32_t flash_init(void);
uint32_t flash_erase(uint32_t startAddr, uint32_t sizeBytes);
uint32_t flash_write(uint32_t startAddr, uint32_t* data, uint32_t
sizeBytes);
/** read sizeBytes bytes from address startAddr to provided buffer.
 * */
uint32_t flash_read(uint32_t startAddr, uint32_t* buffer, uint32_t
sizeBytes);
```

*Zdrojový kód 3: Ukázka funkcí z ovladače paměti flash mikropočítače Freescale MKL25Z128VLK4*

### 6.3 Vrstva ovladače virtuální flash paměti

Tato vrstva je implementována v souborech `fir_drv.h` a `fir_drv.c`. Ovladač pro virtuální paměť flash v RAM vznikl z důvodu testování knihovny. Obsahuje stejné funkce jako standardní ovladač, navíc je ale potřeba provést inicializaci ovladače, kde ovladači předáme informaci o tom, na jaké adrese v paměti RAM začíná virtuální flash (*Zdrojový kód 4*).

```
uint32_t ram_flash_init(uint32_t* ramMemoryAddress)
{
    ramFlashStartAddress = (uint32_t)ramMemoryAddress;

    uint32_t i;

    ram_flash_erase(FLASH_START_ADDRESS, FLASH_REGION_SIZE);

    return FLASH_ERROR_OK;
}
```

*Zdrojový kód 4: Inicializace virtuální paměti flash v RAM*

Součástí inicializace je smazání přidělené paměti pro knihovnu. Při mazání se nastavují jednotlivé buňky na hodnotu `0xFF` tak, aby kopírovaly skutečnou paměť flash.

## 6.4 Vrstva HAL

Vrstva HAL, která je implementována v souborech `flashMem_hal.h` a `flashMem_hal.c`, slouží jako prostředník pro komunikaci mezi logikou knihovny a ovladačem flash paměti. Funkce `FlashErase`, `FlashWrite` a `FlashRead` jsou volány z knihovny a funkce `ram_flash_erase` nebo `flash_erase` jsou volány z jednotlivých ovladačů paměti (reálná paměť flash nebo virtuální paměť flash v RAM).

```
uint32_t FlashErase(uint32_t startSector, uint32_t countSector)
{
    uint32_t startAddr = startSector * FLASH_SECTOR_SIZE;
    uint32_t sizeBytes = countSector * FLASH_SECTOR_SIZE;

    #if PLATFORM == KL25
        return flash_erase(startAddr, sizeBytes); // real flash
    #elif PLATFORM == FIR
        return ram_flash_erase(startAddr, sizeBytes); // simulated flash
        // in RAM
    #endif
}

uint32_t FlashWrite(uint32_t startAddr, uint32_t* data, uint32_t
sizeBytes)
{
    #if PLATFORM == KL25
        return flash_write(startAddr, data, sizeBytes); // real flash
    #elif PLATFORM == FIR
        return ram_flash_write(startAddr, data, sizeBytes); // simulated
        // flash in RAM
    #endif
}

uint32_t FlashRead(uint32_t startAddr, uint32_t* data, uint32_t
sizeBytes)
{
    #if PLATFORM == KL25
        return flash_read(startAddr, data, sizeBytes); // real flash
    #elif PLATFORM == FIR
        return ram_flash_read(startAddr, data, sizeBytes); // simulated
        // flash in RAM
    #endif
}
```

## 6.5 Vrstva logiky knihovny

V této vrstvě, implementované v souborech `flashMem_storage.h` a `flashMem_storage.c`, je soustředěna veškerá logika knihovny. Funkce v těchto souborech jsou rozděleny do dvou skupin. První skupinu tvoří funkce dostupné uživateli, tedy funkce připojení knihovny, zápisu, čtení a defragmentace. Druhou skupinu tvoří funkce interní, podpůrné. Mezi ně můžeme zařadit např. vyhledávání záznamů v paměti.

V těchto souborech najdeme dvě globální proměnné, a to *variableList* reprezentující pomocné pole proměnných a *gFlashWriteAddress*, které si uchovává informaci o adrese, na kterou se zapíše záznam při příchozím požadavku na zápis do paměti.

### 6.5.1 Funkce Mount

Tato funkce musí být volána vždy před začátkem práce s knihovnou. Cílem této funkce je provést kontrolu nastavených údajů (při překladu) a následně za běhu programu, pokud je povoleno pomocné pole proměnných toto pole naplnit aktuálními adresami záznamů v paměti flash. Současně se zapíše do globální proměnné *wasMount* hodnota 1, která značí že *Mount* byl proveden (a tím je naplněno pomocné pole proměnných). Pokud je zvolena virtuální flash v RAM, pak se předá ovladači virtuální flash v RAM informace o tom, kde v paměti začíná přidělená paměť.

### 6.5.2 Funkce pro zápis dat do paměti

Funkce pro zápis dat se jmenují podle toho, jaké data chceme do paměti uložit. Strukturou jsou stejné, liší se pouze v délce dat, které se zapisují.

```
uint8_t WriteWord(uint16_t id, uint16_t data)
{
    // vytvoreni docasneho zaznamu pro data
    uint8_t tmpRecord[STORAGE_RECORD_MAXIMUM_SIZE];

    // vynulovani docasneho zaznamu
    NullTmpRecord(tmpRecord);

    // zkopirovani dat do docasneho zaznamu
    memcpy(tmpRecord + STORAGE_RECORD_HEAD, &data, 2);

    // zapis zaznamu do flash pameti
    return WriteValue(id, 2, tmpRecord);
}
```

#### Zdrojový kód 6: Funkce WriteWord

Ze zdrojového kódu (Zdrojový kód 6) lze vyčíst, že jako první se vytvoří dočasný záznam *tmpRecord* o jeho maximální velikosti. V následujícím kroku se všechny prvky *tmpRecord* nastaví na hodnotu 0.

Za místo o velikosti hlavičky záznamu se zkopírují požadovaná data, která mají být uložena. Ve funkci *memcpy()* na konci vidíme číslo 2, které znamená, že zkopírujeme 2 B informací. Pokud by se jednalo o funkci *WriteByte*, číslo by bylo 1 nebo u funkce *WriteDoubleWord* je číslo rovno 4. Jako poslední se zavolá interní funkce *WriteValue*, jejichž úkolem je doplnit hlavičku záznamu, najít volné místo v paměti flash a záznam do paměti zapsat.

Funkce *WriteValue* (Zdrojový kód 7) funguje následovně:

- Pokud je povoleno pomocné pole proměnných, tak se zkontroluje, zda byl proveden *Mount*. Pokud nebyl proveden, funkce končí s chybou.
- V dalším kroku se zjistí, zda požadované ID, pod kterým chce uživatel záznam uložit, leží v povoleném rozsahu.
- Zaokrouhlí se směrem nahoru potřebné místo v paměti na daný záznam. Toto zaokrouhlení je provedeno z důvodu, že do flash paměti může být povolen zápis jen po určité skupině bytů (např. 4B u vybraného mikropočítače Freescale). Pokud tedy chceme u vybraného mikropočítače zapsat data o velikosti 2B, pak potřebujeme celkem 8B paměti flash (3B hlavička, 2B data a 3B jsou nevyužité – zaokrouhlení na nejmenší zapisovatelnou jednotku, v tomto případě 8B).

- Vytvoření a naplnění struktury hlavičky daty (identifikátorem a velikostí záznamu v paměti)
- Nalezení volného místa v paměti, kam se bude zapisovat (funkce *GetAndCheckWriteAddress()*)
- Pokud je dostatek místa v paměti, provede se zápis záznamu, v opačném případě je uživateli vrácena chyba.
- Pokud je povoleno pomocné pole proměnných, pak se aktualizuje adresa právě uloženého záznamu v tomto poli.
- Aktualizace proměnné pro další zápis záznamu (*gFlashWriteAddress*)

```
static uint8_t WriteValue(uint16_t id, uint8_t size, uint8_t*
tmpRecord)
{
#if (USE_VARIABLE_LIST == 1)
    // kontrola, zda byl proveden Mount
    if(wasMount == 0)
    {
        return LIBRARY_MOUNT_ERROR;
    }
#endif

    // kontrola platneho id
    if(id >= FLASH_MAXIMUM_VARIABLES)
    {
        return LIBRARY_ID_OUT_OF_RANGE;
    }

    // zaokrouhleni velikosti na nejblizsi vyssi zapisovatelny blok
    uint8_t roundedUpSize = RoundUpRecordSize(STORAGE_RECORD_HEAD +
size);

    STORAGE_RECORD* rec = (STORAGE_RECORD*)tmpRecord;
    rec->id = id;
    rec->size = roundedUpSize; // uklada se celkova velikost zaznamu

    uint8_t result = GetAndCheckWriteAddress(roundedUpSize);

    // pokud se nepodarilo najit adresu k zapisu, vrati se chyba
    if(result != LIBRARY_OK)
    {
        return result;
    }

    // zapis zaznamu do flash
    result = FlashWrite( gFlashWriteAddress, (uint32_t*)tmpRecord,
roundedUpSize);

    // kontrola zapisu do flash
    if(result != LIBRARY_OK)
    {
        return result;
    }

#if (USE_VARIABLE_LIST == 1)
    // záznam do variableList
    variablesList[id] = gFlashWriteAddress;
#endif

    // posun adresy pro dalsi zapis
    gFlashWriteAddress = gFlashWriteAddress + roundedUpSize;

    return LIBRARY_OK;
}
```

Zdrojový kód 7: Funkce WriteValue společná pro všechny zapisující funkce

### 6.5.3 Funkce pro čtení dat z paměti

Existuje celkem pět funkcí pro čtení z paměti, každá pro jiný druh dat. V podstatě jsou ale podobně řešené.

```
uint16_t ReadWord(uint16_t id)
{
    // vypocita se delka zaznamu
    uint8_t roundedUpSize = RoundUpRecordSize(STORAGE_RECORD_HEAD +
    2);

    // vytvori se promenna s vychozi hodnotou, která se vrati v
    // pripade chyby cteni
    uint16_t result = 0xFFFF;

    uint32_t recordAddress;

    // hledani adresy zaznamu v pameti
    #if (USE_VARIABLE_LIST == 1)
    // kontrola, zda byl proveden Mount
    if(wasMount == 0)
    {
        return result;
    }
    recordAddress = variablesList[id];
    #else
    recordAddress = FindRecordAddressById(id);
    #endif

    // kontrola nalezeného zaznamu
    if(recordAddress == 0xFFFFFFFF)
    {
        return result;
    }

    // precteni hlavicky z dane adresy
    STORAGE_RECORD record;
    ReadRecordHeader(recordAddress, &record);

    // testovani, zda ulozena velikost zaznamu v pameti odpovida te,
    // jakou by mela mit velikost
    if(record.size == roundedUpSize)
    {
        uint8_t tmp[record.size];

        // precteni zaznamu z pameti
        FlashRead(recordAddress, (uint32_t*)tmp, record.size);

        // ulozeni prectene hodnoty do "result"
        memcpy((void*)&result, tmp + STORAGE_RECORD_HEAD, 2);
    }
    return result;
}
```

*Zdrojový kód 8: Funkce ReadWord pro čtení 2 B dat z flash paměti*

Princip funkce ReadWord, která čte z flash paměti data o velikosti 2 B, je následující:

- Vypočítá se délka záznamu v paměti pro data o velikosti 2 B
- Vytvoří se proměnná *result* s výchozí hodnotou 0xFFFF. Hodnota 0xFFFF se uživateli vrátí v případě, že záznam s požadovaným identifikátorem v paměti neexistuje.
- Proveďte se kontrola, zda byl proveden *Mount* v případě použití pomocného pole proměnných.
- Proveďte se vyhledávání adresy daného záznamu v paměti podle identifikátoru.
  - o Pokud nalezená adresa má hodnotu 0xFFFFFFFF, pak to znamená, že záznam s požadovaným identifikátorem neexistuje a funkce čtení se ukončí.
- Pokud je adresa platná, pak se z ní přečte hlavička záznamu a přečtená velikost záznamu se porovná s předem vypočítanou.
- V případě správné velikosti záznamu se provede čtení hodnoty záznamu z paměti a přečtená hodnota se vrátí uživateli.

#### 6.5.4 Funkce pro defragmentaci paměti

Defragmentace paměti probíhá následovně:

- Zjistí se adresa první a druhé poloviny přidělené paměti (*Zdrojový kód 9*)
- Z těchto dvou adres se přečte hlavička
- Následuje porovnání získaných hlaviček
  - o Pokud je v první polovině paměti uložen nějaký záznam a v druhé polovině paměti není uloženo nic, pak je první polovina označena za fragmentovanou polovinu a do druhé poloviny (prázdné) je prováděna defragmentace.
  - o To samé platí i opačně. Pokud je první polovina paměti prázdná a v druhé polovině je uložen nějaký záznam, pak je do první poloviny paměti prováděna defragmentace a v druhé polovině jsou fragmentované záznamy.
  - o Pokud předchozí podmínky neplatí, pak je vrácena chyba defragmentace.

```
uint32_t Defragment()
{
    // zjisteni adresy, kde zacina pridelená pamet
    uint32_t firstMemoryAddress = FLASH_START_ADDRESS;

    // zjisteni adresy, kde je presne pulka pameti
    uint32_t halfMemoryAddress = FLASH_START_ADDRESS +
    (FLASH_COUNT_SECTOR / 2) * FLASH_SECTOR_SIZE;

    // nacteni hlavicky na zacatku pridelené pameti
    STORAGE_RECORD recordFirst;
    ReadRecordHeader(firstMemoryAddress, &recordFirst);

    // nacteni hlavicky v pulce pridelené pameti
    STORAGE_RECORD recordHalf;
    ReadRecordHeader(halfMemoryAddress, &recordHalf);

    // adresa zacatku zapisu defragmentovanych zaznamu
    uint32_t defragmentStartAddress;

    // adresa zacatku cteni fragmentovanych zaznamu
    uint32_t defragmentReadStartAddress;

    if(recordFirst.id == NO_VALID_ID) // v recordFirs.Id je
        // ulozen neplatny zaznam (je tam prazdna pamet)
    {
        if(recordHalf.id != NO_VALID_ID) // v recordHalf.Id je
            // ulozen platny zaznam - zaplnena pamet
        {
            defragmentStartAddress = firstMemoryAddress;
            defragmentReadStartAddress = halfMemoryAddress;
        }else{
            return LIBRARY_DEFRAGMENT_ERROR;
        }
    }else{
        if(recordHalf.id == NO_VALID_ID) // v recordFirs.Id je
            // ulozen platny zaznam (plna pamet) a v recordHalf.Id je ulozen
            // neplatny zaznam (volna pamet)
        {
            defragmentStartAddress = halfMemoryAddress;
            defragmentReadStartAddress = firstMemoryAddress;
        }else{
            return LIBRARY_DEFRAGMENT_ERROR;
        }
    }
}
```

*Zdrojový kód 9: Funkce defragmentování paměti, část 1/3*

- Nastaví se aktuální pozice při defragmentování na začátek oblasti pro defragmentování (*Zdrojový kód 10*)

- V cyklu se projdou všechny možné záznamy (podle identifikátoru), které se můžou v paměti nacházet.

```
uint32_t defragmentActualAddress = defragmentStartAddress;

uint32_t i;

// projdou se všechny záznamy, od 0 do (FLASH_MAXIMUM_VARIABLES - 1)
for(i = 0; i < FLASH_MAXIMUM_VARIABLES; i = i + 1)
{
    uint32_t recordAddress;
    // hledání adresy záznamu v paměti
    #if (USE_VARIABLE_LIST == 1)
        // kontrola, zda byl proveden Mount
        if(wasMount == 0)
        {
            return LIBRARY_MOUNT_ERROR;
        }
        recordAddress = variablesList[i];
    #else
        // adresa částí paměti, která se defragmentuje
        recordAddress = DefragmentFindRecordAddressById(i,
defragmentReadStartAddress);
    #endif

    // na adrese je něco uloženého
    if(recordAddress != 0xFFFFFFFF){
        // vytvoření prázdného dočasného záznamu
        uint8_t tmpRecord[STORAGE_RECORD_MAXIMUM_SIZE];

        // přečtení hlavičky záznamu
        STORAGE_RECORD record;
        ReadRecordHeader(recordAddress, &record);

        // zkopírování záznamu do dočasného záznamu
        // tmpRecord
        FlashRead(recordAddress, (uint32_t*)tmpRecord,
record.size);

        // samotná defragmentace záznamu
        FlashWrite(defragmentActualAddress,
(uint32_t*)tmpRecord, record.size);

        // pokud je povoleno použít "variableList",
        // pak do něj uložím aktuální adresu výskytu záznamu v paměti
        #if (USE_VARIABLE_LIST == 1)
            variablesList[i] = defragmentActualAddress;
        #endif

        // nastavení ukládání dalšího záznamu ihned za aktuálně uložený
        defragmentActualAddress = defragmentActualAddress +
record.size;
    }
}
```

- Pokud je povoleno pomocné pole proměnných, pak se vezme aktuální adresa záznamu z něj. Pokud povoleno není, adresa záznamu se najde v paměti.
- Pokud je na adrese něco uloženo, pak se záznam zkopíruje z fragmentované poloviny paměti do defragmentované. Při povoleném pomocném poli proměnných se zároveň aktualizuje adresa daného záznamu v tomto poli.
- Adresa pro uložení dalšího defragmentovaného záznamu (*defragmentActualAddress*) se zvýší o velikost právě defragmentovaného záznamu.

```
// po skončení defragmentace se nastaví adresa, na kterou se budou
// zapisovat další uživatelské záznamy
gFlashWriteAddress = defragmentActualAddress;

// smazání defragmentovaných sektorů
if(defragmentStartAddress >= halfMemoryAddress)
{
    FlashErase(FLASH_START_SECTOR, FLASH_COUNT_SECTOR / 2);
} else {
    FlashErase(FLASH_START_SECTOR + (FLASH_COUNT_SECTOR / 2) ,
FLASH_COUNT_SECTOR / 2);
}

return LIBRARY_OK;
}
```

*Zdrojový kód 11: Funkce defragmentování paměti, část 3/3*

- Až cyklus skončí a projdou se všechny možné záznamy, nastaví se globální proměnná *gFlashWriteAddress* na hodnotu adresy, kde se při dalším požadavku na zápis záznamu bude zapisovat.
- V posledním kroku se smaže právě defragmentovaná polovina paměti (kde jsou duplicitní záznamy, *Zdrojový kód 11*)

## 7 VYTVOŘENÍ NOVÉHO PROJEKTU

V této kapitole bude probráno, jak vytvořit nový projekt zahrnující knihovnu pro zápis do paměti flash. Najdeme zde, jak vytvořit projekt se zápisem do skutečné flash paměti a do virtuální flash paměti v RAM. Projekty se budou tvořit pro mikropočítač Freescale MKL25Z128VLK4 ve vývojovém prostředí Kinetis Design Studio.

### 7.1 Projekt se zápisem do flash paměti

- 1) Vytvoříme si nový Kinetis Design Studio projekt, pojmenujeme ho a z nabídky zvolíme správný mikropočítač.
- 2) Do složky „Sources“ zkopírujeme veškeré zdrojové soubory:
  - a. config.h
  - b. kl25\_drv.h, kl25\_drv.c
  - c. fir\_drv.h, fir\_drv.c
  - d. flashMem\_hal.h, flashMem\_hal.c
  - e. flashMem\_storage.h, flashMem\_storage.c
  - f. MKL25Z4\_ftfa.h, MKL25Z4\_mcm.h, fsl\_bitaccess.h - tyto soubory jsou potřeba pro správnou funkci ovladače flash paměti
- 3) Do souboru main.c provedeme vložení těchto souborů:
  - a. #include <stdio.h>
  - b. #include "MKL25Z4.h"
  - c. #include "flashMem\_storage.h"
- 4) V souboru config.h nastavíme parametry
  - a. FLASH\_MAXIMUM\_VARIABLES
  - b. USE\_VARIABLES\_LIST
  - c. Nastavíme parametry flash paměti podle zvoleného mikropočítače
    - i. FLASH\_ADDRESS\_ALIGMENT
    - ii. FLASH\_SECTOR\_SIZE
    - iii. FLASH\_WRITE\_SIZE

- d. Nastavíme přidělenou paměť flash pro knihovnu
    - i. FLASH\_START\_SECTOR
    - ii. FLASH\_COUNT\_SECTOR
  - e. Nastavíme PLATFORM (ovladač flash paměti)
    - i. KL25 pro mikropočítač MKL25Z128VLK4
    - ii. FIR pro virtuální flash paměť v RAM
- 5) V souboru *main.c* ve funkci *main(void)* jako první voláme funkci *Mount()*

## 7.2 Projekt se zápisem do virtuální flash v RAM

Postup je stejný jako při zápisu do flash paměti s následujícími rozdíly:

- Bod 5 se změnil takto
  - o V souboru *main.c* před funkcí *main(void)* vytvoříme dostatečně velké pole, které bude sloužit jako virtuální flash v RAM  
Např. `uint32_t arrayFlashInRam[(10*64) / 4];`  
Vytvoříme si 10 sektorů po 64B. Protože je pole typu `uint32_t` (4B), podělíme i (10\*64) čtyřma.
  - o V souboru *main.c* ve funkci *main(void)* voláme funkci *Mount(arrayFlashInRam);*

## 8 UKÁZKOVÉ PROJEKTY

Byly vytvořeny celkem dva ukázkové projekty, kde jeden zapisuje do flash paměti a druhý zapisuje do virtuální flash v RAM.

### 8.1 Projekt č. 1: Zápis do flash paměti

V ukázkovém projektu č. 1 je prováděn zápis do reálné flash paměti. Byly použity čtyři sektory od sektoru 100.

```
int main(void)
{
    uint8_t status;

    status = Mount();

    status = WriteByte(2,0x55);
    status = WriteWord(7,0x1122);
    status = WriteDoubleWord(3,0x885544AA);
    status = WriteDoubleLong(12, 0x1122334455667788);

    uint8_t string1[] = "Hello world";
    status = WriteString(15, string1,sizeof(string1));

    status = WriteByte(2,0x66);
    status = WriteWord(7,0x7744);
    status = WriteDoubleWord(3,0xAABBCCDD);
    status = WriteDoubleLong(12, 0xAABBCCDD11223344);

    uint8_t string2[] = "Hello world 2015";
    status = WriteString(15, string2,sizeof(string2));

    // #1
    uint8_t var1 = ReadByte(2);           // 0x66
    uint16_t var2 = ReadWord(7);         // 0x7744
    uint32_t var3 = ReadDoubleWord(3);   // 0xAABBCCDD
    uint64_t var4 = ReadDoubleLong(12);  // 0xAABBCCDD11223344

    uint8_t string3[sizeof(string2)];
    status = ReadString(15, string3, sizeof(string3)); // string3 =
                                                    // "Hello world 2015"

    status = Defragment();
    // #2

    status = WriteByte(2, 0xCC);
    // #3

    status = Defragment();

    // #4
}
```

*Zdrojový kód 12: Ukázka z ukázkového projektu č. 1*

Na obrázku (*Obrázek 13*) lze vidět, jak jsou za sebou jednotlivé záznamy uloženy v paměti (#1). Výsledky čtení z paměti za bodem #1 jsou umístěny v komentářích za jednotlivými funkcemi.

```
1024*100
Ox19000 - 1024*100 <Traditional> X
0x00019000 55040002 22080007 00000011 AA080003 00885544 880C000C 44556677 00112233 4810000F 6F6C6C65 726F7720 0000646C 66040002
0x00019034 44080007 00000077 DD080003 00AABBCC 440C000C DD112233 00AABBCC 4814000F 6F6C6C65 726F7720 3220646C 00353130 FFFFFFFF
0x00019068 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0x0001909C FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
```

*Obrázek 13: Obsah paměti v bodě #1 ukázkového projektu č. 1*

Z obrázku (*Obrázek 14*) lze vidět, že po provedení defragmentace se data defragmentují do druhé poloviny přidělené paměti (sektor 102). Záznamy po provedení defragmentace již nejsou duplicitní.

```
1024*102
Ox19800 - 1024*102 <Traditional> X
0x00019800 66040002 DD080003 00AABBCC 44080007 00000077 440C000C DD112233 00AABBCC 4814000F 6F6C6C65 726F7720 3220646C 00353130
0x00019834 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0x00019868 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0x0001989C FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
```

*Obrázek 14: Obsah paměti v bodě #2 ukázkového projektu č. 1*

V dalším kroku (#3) se provedl zápis jednoho bytu do paměti, který můžeme vidět na obrázku (*Obrázek 15*). Je patrné, že zápis se provedl za defragmentovaná data.

```
1024*102
Ox19800 - 1024*102 <Traditional> X
0x00019800 66040002 DD080003 00AABBCC 44080007 00000077 440C000C DD112233 00AABBCC 4814000F 6F6C6C65 726F7720 3220646C 00353130
0x00019834 CC040002 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0x00019868 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
0x0001989C FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
```

*Obrázek 15: Obsah paměti v bodě #3 ukázkového programu č. 1*

V bodě #4, kdy je opět provedena defragmentace, se data přesunula do první poloviny paměti (*Obrázek 16*).

1024*100													
0x19000 - 1024*100 <Traditional> X													
0x00019000	CC040002	DD080003	00AABBCC	44080007	00000077	440C000C	DD112233	00AABBCC	4814000F	6F6C6C65	726F7720	3220646C	00353130
0x00019034	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
0x00019068	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
0x0001909C	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF

Obrázek 16: Obsah paměti v bodě #4 ukázkového projektu č. 1

## 8.2 Projekt č. 2: Zápis do virtuální flash v RAM

Ukázkový projekt č. 2 je stejný jako ukázkový projekt č. 1. Rozdíl je v tom, že projekt č. 2 zapisuje do pole *arrayFlashInRam*. V tomto projektu je nastavena velikost sektoru na hodnotu 32B. Začátek paměti přidělené knihovně začíná na sektoru č. 2 a má velikost 4 sektory. Na začátku je tak vytvořeno pole o velikosti deset sektorů po 32B (*Zdrojový kód 13*). Následně je pomocí funkce *Mount(arrayFlashInRam)* předán parametr, kde v paměti pole *arrayFlashInRam* začíná.

Z důvodu takového nastavení ovladače (čtyři sektory po 32 B) se provede poslední zápis u položky *status = WriteWord(7, 0x7744);*, protože pak už došla paměť a není již kam ukládat.

Na obrázku (*Obrázek 17*) lze vidět obsah paměti v bodě #1. Poslední zapsaný záznam je skutečně hodnota 0x7744 pod indexem 7.

Z obrázku (*Obrázek 18*) lze vidět, jak proběhla defragmentace a zároveň, proč nešlo dříve zapsat do paměti více záznamů – zbývalo místo pouze pro zápis jednoho bytu dat (jeden čtyř bytový záznam).

Další body programu jsou podobné jako u ukázkového programu č. 1.

(x)- arrayFlashInRam[13]	uint32_t	0x0
(x)- arrayFlashInRam[14]	uint32_t	0x0
(x)- arrayFlashInRam[15]	uint32_t	0x0
(x)- arrayFlashInRam[16]	uint32_t	0x55040002
(x)- arrayFlashInRam[17]	uint32_t	0x22080007
(x)- arrayFlashInRam[18]	uint32_t	0x11
(x)- arrayFlashInRam[19]	uint32_t	0xaa080003
(x)- arrayFlashInRam[20]	uint32_t	0x885544
(x)- arrayFlashInRam[21]	uint32_t	0x880c000c
(x)- arrayFlashInRam[22]	uint32_t	0x44556677
(x)- arrayFlashInRam[23]	uint32_t	0x112233
(x)- arrayFlashInRam[24]	uint32_t	0x4810000f
(x)- arrayFlashInRam[25]	uint32_t	0x6f6c6c65
(x)- arrayFlashInRam[26]	uint32_t	0x726f7720
(x)- arrayFlashInRam[27]	uint32_t	0x646c
(x)- arrayFlashInRam[28]	uint32_t	0x66040002
(x)- arrayFlashInRam[29]	uint32_t	0x44080007
(x)- arrayFlashInRam[30]	uint32_t	0x77
(x)- arrayFlashInRam[31]	uint32_t	0xffffffff
(x)- arrayFlashInRam[32]	uint32_t	0xffffffff
(x)- arrayFlashInRam[33]	uint32_t	0xffffffff

Obrázek 17: Obsah paměti v bodě #1  
ukázkového projektu č. 2

(x)- arrayFlashInRam[30]	uint32_t	0xffffffff
(x)- arrayFlashInRam[31]	uint32_t	0xffffffff
(x)- arrayFlashInRam[32]	uint32_t	0x66040002
(x)- arrayFlashInRam[33]	uint32_t	0xaa080003
(x)- arrayFlashInRam[34]	uint32_t	0x885544
(x)- arrayFlashInRam[35]	uint32_t	0x44080007
(x)- arrayFlashInRam[36]	uint32_t	0x77
(x)- arrayFlashInRam[37]	uint32_t	0x880c000c
(x)- arrayFlashInRam[38]	uint32_t	0x44556677
(x)- arrayFlashInRam[39]	uint32_t	0x112233
(x)- arrayFlashInRam[40]	uint32_t	0x4810000f
(x)- arrayFlashInRam[41]	uint32_t	0x6f6c6c65
(x)- arrayFlashInRam[42]	uint32_t	0x726f7720
(x)- arrayFlashInRam[43]	uint32_t	0x646c
(x)- arrayFlashInRam[44]	uint32_t	0xffffffff
(x)- arrayFlashInRam[45]	uint32_t	0xffffffff
(x)- arrayFlashInRam[46]	uint32_t	0xffffffff

Obrázek 18: Obsah paměti v bodě #2 ukázkového projektu

č. 2

```
uint32_t arrayFlashInRam[(10*32) / 4];
int main(void)
{
    uint8_t status;

    status = Mount(arrayFlashInRam);

    status = WriteByte(2,0x55);
    status = WriteWord(7,0x1122);
    status = WriteDoubleWord(3,0x885544AA);
    status = WriteDoubleLong(12, 0x1122334455667788);

    uint8_t string1[] = "Hello world";
    status = WriteString(15, string1,sizeof(string1));

    status = WriteByte(2,0x66);
    status = WriteWord(7,0x7744);
    status = WriteDoubleWord(3,0xAABBCCDD);
    status = WriteDoubleLong(12, 0xAABBCCDD11223344);

    uint8_t string2[] = "Hello world 2015";
    status = WriteString(15, string2,sizeof(string2));

    // #1
    uint8_t var1 = ReadByte(2);           // 0x66
    uint16_t var2 = ReadWord(7);         // 0x7744
    uint32_t var3 = ReadDoubleWord(3);   // 0x885544AA
    uint64_t var4 = ReadDoubleLong(12);  // 0x1122334455667788

    uint8_t string3[sizeof(string2)];
    status = ReadString(15, string3, sizeof(string3)); // string3 =
                                                    // "Hello world"

    status = Defragment();
    // #2

    status = WriteByte(2, 0xCC);
    // #3

    status = Defragment();
    // #4
}
```

*Zdrojový kód 13: Ukázka z ukázkového projektu č. 2*

## ZÁVĚR

Tato práce se zabývá využitím interní flash paměti mikropočítače pro ukládání dat s rovnoměrným opotřebením paměti. Jako první byl v teoretické části proveden rozbor stávajících řešení, především speciálních systému souborů pro flash paměti. Základní princip je u všech systémů souborů stejný – neustále ukládat nové verze záznamů a při docházející paměti provést úklid paměti tak, že se smažou staré verze záznamů a tím vzniknou nové prázdné sektory. Podobně je navrhuta i programová knihovna.

Programová knihovna je vytvořena tak, aby byla co nejvíce variabilní. Dokáže zapsat informace o délce 1 B, 2 B, 4 B i 8 B. Navíc je k dispozici i funkce pro zápis textového řetězce, kde může být délka uložených dat ve výchozím nastavení až 17B. Maximální délka dat v jednom záznamů se dá v případě potřeby zvýšit až na 254 B.

Defragmentace je řešena tak, že si ji uživatel volá sám a není spouštěna automaticky po zaplnění paměti. Tento způsob byl zvolen proto, aby měl uživatel kontrolu nad strojovým časem. Defragmentace může trvat i relativně dlouhou dobu a uživatel si ji tak může spustit např. když ví, že v následujícím okamžiku není potřeba zpracovávat žádné úkoly a je tak dostupný procesorový čas.

V rámci práce byla vytvořena knihovna pro ukládání dat do flash pamětí mikropočítače. Kromě samotného zápisu do flash paměti byl vytvořen i ovladač pro virtuální paměť flash v RAM. Uživatel si tak může vyzkoušet funkci knihovny na libovolných parametrech, které jsou na reálných flash pamětech pevně dané a nelze je měnit. Mezi tyto parametry můžeme zařadit velikost jednoho sektoru nebo minimální velikost dat, která jdou do paměti zapsat.

Tato knihovna může být použita na všech mikropočítačích, ale v současné době je implementována pouze pro mikropočítač MKL25Z128VLK4 z rodiny Kinetis. Je navržena tak, aby byla snadno rozšířitelná na další platformy. Není problém tuto knihovnu do budoucna přemigrovat na stále oblíbenější platformu Arduino a použít ji tak na procesorech Atmel.

**SEZNAM POUŽITÉ LITERATURY**

- [1] PALACKÝ, Petr. MIKROPOČÍTAČOVÉ ŘÍDICÍ SYSTÉMY I [online]. Ostrava, 2007 [cit. 2015-05-19]. Dostupné z: [http://www.elearn.vsb.cz/archivcd/FEI/MRS1/Palacky\\_MRS\\_elerning.pdf](http://www.elearn.vsb.cz/archivcd/FEI/MRS1/Palacky_MRS_elerning.pdf). učební text.
- [2] FREESCALE SEMICONDUCTOR, INC. *Kinetis KL25 Sub-Family: Data Sheet: Technical Data* [online]. 2014 [cit. 2015-05-18]. Dostupné z: [http://cache.freescale.com/files/32bit/doc/data\\_sheet/KL25P80M48SF0.pdf](http://cache.freescale.com/files/32bit/doc/data_sheet/KL25P80M48SF0.pdf)
- [3] Co je programování? *Jak se naučit programovat: Co je to programování?* [online]. [cit. 2015-05-18]. Dostupné z: <http://jaksenaucitprogramovat.py.cz/cztutwhat.html>
- [4] ASSEMBLER. *Programování - Assembler* [online]. [cit. 2015-05-18]. Dostupné z: <http://k-prog.wz.cz/progjaz/asmemb.php>
- [5] Výuka assembleru: Začínáme s assemblerem. ZEZULA, Ladislav. *Výuka assembleru - 1. díl* [online]. 2003 [cit. 2015-05-18]. Dostupné z: [http://www.zezula.net/cz/teach/assembler\\_01.html](http://www.zezula.net/cz/teach/assembler_01.html)
- [6] FALTÝNEK, Lukáš. Jazyk C a C++. *Linux Expres* [online]. 2007 [cit. 2015-05-18]. Dostupné z: <http://www.linuxexpres.cz/praxe/jazyk-c-a-c>
- [7] DŘÍNEK, Milan. RISC versus CISC architektura. *RISC versus CISC* [online]. 2000 [cit. 2015-05-18]. Dostupné z: [http://avr.hw.cz/architektura/risc\\_cisc.html](http://avr.hw.cz/architektura/risc_cisc.html)
- [8] Jednočipový počítač. *Wikipedia* [online]. 2014 [cit. 2015-05-18]. Dostupné z: [http://cs.wikipedia.org/wiki/Jedno%C4%8Dipov%C3%BD\\_po%C4%8D%C3%A4ta%C4%8D](http://cs.wikipedia.org/wiki/Jedno%C4%8Dipov%C3%BD_po%C4%8D%C3%A4ta%C4%8D)
- [9] Von Neumannova architektura. *Wikipedia* [online]. 2014 [cit. 2015-05-18]. Dostupné z: [http://cs.wikipedia.org/wiki/Von\\_Neumannova\\_architektura#/media/File:Von\\_Neumann\\_Architecture\\_CZ.svg](http://cs.wikipedia.org/wiki/Von_Neumannova_architektura#/media/File:Von_Neumann_Architecture_CZ.svg)
- [10] Harvard architecture. *Wikipedia* [online]. 2010 [cit. 2015-05-18]. Dostupné z: [http://en.wikipedia.org/wiki/Harvard\\_architecture#/media/File:Harvard\\_architecture.svg](http://en.wikipedia.org/wiki/Harvard_architecture#/media/File:Harvard_architecture.svg)
- [11] Processor architectures: Harvard, von Neumann and Modified Harvard architectures. *Embedded Matters* [online]. 2010 [cit. 2015-05-18]. Dostupné z:

- <http://embeddedknowledge.blogspot.cz/2010/02/processor-architectures-harvard-von.html>
- [12] TIŠNOVSKÝ, Pavel. Mikroprocesory s architekturou ARM. *Root.cz* [online]. 2012 [cit. 2015-05-18]. Dostupné z: <http://www.root.cz/clanky/mikroprocesory-s-architekturou-arm/>
- [13] Počítačová paměť. Wikipedia [online]. 2015 [cit. 2015-05-18]. Dostupné z: [http://cs.wikipedia.org/wiki/Po%C4%8D%C3%ADta%C4%8Dov%C3%A1\\_pam%C4%9B%C5%A5](http://cs.wikipedia.org/wiki/Po%C4%8D%C3%ADta%C4%8Dov%C3%A1_pam%C4%9B%C5%A5)
- [14] Elektronická paměť. Wikipedia [online]. 2015 [cit. 2015-05-18]. Dostupné z: [http://cs.wikipedia.org/wiki/Elektronick%C3%A1\\_pam%C4%9B%C5%A5](http://cs.wikipedia.org/wiki/Elektronick%C3%A1_pam%C4%9B%C5%A5)
- [15] PROM. Pandatron.cz [online]. 2015 [cit. 2015-05-18]. Dostupné z: <http://pandatron.cz/?193&prom>
- [16] Flash paměť. Wikipedia [online]. 2015 [cit. 2015-05-18]. Dostupné z: [http://cs.wikipedia.org/wiki/Flash\\_pam%C4%9B%C5%A5](http://cs.wikipedia.org/wiki/Flash_pam%C4%9B%C5%A5)
- [17] VONDRÁŠEK, Martin. Princip a vlastnosti USB flash pamět. Princip a vlastnosti USB flash pamět [online]. 2009 [cit. 2015-05-18]. Dostupné z: <http://fallvonder.net/rest/flashtech/vondrm4-y31-eliflash.pdf>
- [18] HÁNA, Jan. Vnější paměti a principy jejich činnosti [online]. Brno, 2014 [cit. 2015-05-18]. Dostupné z: [https://is.muni.cz/th/325076/fi\\_m/Vnejsi\\_pameti\\_a\\_pricipy\\_jejich\\_cinnosti.pdf](https://is.muni.cz/th/325076/fi_m/Vnejsi_pameti_a_pricipy_jejich_cinnosti.pdf).  
Diplomová práce.
- [19] TIŠNOVSKÝ, Pavel. Technologie flash pamětí a způsoby jejich využití. *Root.cz* [online]. 2008 [cit. 2015-05-18]. Dostupné z: <http://www.root.cz/clanky/technologie-flash-pameti-a-zpusoby-jejich-vyuziti/>
- [20] MILLER, Warren. Understanding the Differences Between NAND Flash and NOR Flash Memory and Key Future Trends. AVNET [online]. [cit. 2015-05-18]. Dostupné z: <http://www.em.avnet.com/en-us/design/technical-articles/Pages/Articles/Understanding-the-Differences-Between-NAND-Flash-and-NOR-Flash-Memory-and-Key-Future-Trends.aspx>
- [21] ČERNÝ, Jan. Solidní budoucnost pevných disků – úvod k velkému testu SSD disků. *Pc tuning* [online]. 2010 [cit. 2015-05-18]. Dostupné z:

- <http://pctuning.tyden.cz/hardware/disky-cd-dvd-br/18914-solidni-budoucnost-pevnych-disku-uvod-k-velkemu-testu-ssd-disku>
- [22] Wear Leveling in NAND Flash Memory. Micron [online]. 2011, : 5 [cit. 2015-05-18]. Dostupné z: [https://www.google.cz/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&ved=0C DIQFjAB&url=https%3A%2F%2Fwww.micron.com%2F~%2Fmedia%2Fdocuments%2Fproducts%2Ftechnical-note%2Fnand-flash%2Ftn2961\\_wear\\_leveling\\_in\\_nand.pdf&ei=3rZUVZW2DMXfUYbmgfAD&usg=AFQjCNGM3OObkO6B7iIpIXeB4OkELHNJrQ&bvm=bv.93112503,d.d24&cad=rja](https://www.google.cz/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&ved=0C DIQFjAB&url=https%3A%2F%2Fwww.micron.com%2F~%2Fmedia%2Fdocuments%2Fproducts%2Ftechnical-note%2Fnand-flash%2Ftn2961_wear_leveling_in_nand.pdf&ei=3rZUVZW2DMXfUYbmgfAD&usg=AFQjCNGM3OObkO6B7iIpIXeB4OkELHNJrQ&bvm=bv.93112503,d.d24&cad=rja)
- [23] USB flash drive. Wikipedia [online]. 2015 [cit. 2015-05-18]. Dostupné z: [http://en.wikipedia.org/wiki/USB\\_flash\\_drive](http://en.wikipedia.org/wiki/USB_flash_drive)
- [24] WOODHOUSE, David. JFFS Version 1. JFFS Version 1 [online]. 2001 [cit. 2015-05-18]. Dostupné z: <https://sourceware.org/jffs2/jffs2-html/node2.html>
- [25] WOODHOUSE, David. JFFS : The Journalling Flash File System [online]. Red Hat, Inc., 2005, : 12 [cit. 2015-05-18]. Dostupné z: <https://www.sourceware.org/jffs2/jffs2.pdf>
- [26] How Yaffs works. MANNING, Charles. Yaffs [online]. 2012 [cit. 2015-05-18]. Dostupné z: <http://www.yaffs.net/documents/how-yaffs-works#Scanning>
- [27] ENGEL, Jörn a Robert MERTENS. LogFS - finally a scalable flash file system. LogFS - finally a scalable flash file system [online]. [2005], : 8 [cit. 2015-05-18]. Dostupné z: [http://www2.informatik.uni-osnabrueck.de/papers\\_pdf/2005\\_07.pdf](http://www2.informatik.uni-osnabrueck.de/papers_pdf/2005_07.pdf)
- [28] SHU, Liu, Guan XUETAO, Tong DONG a Cheng XU. Analysis and Comparison of NAND Flash Specific File Systems. Chinese Journal of Electronics [online]. 2010, 19(3): 6 [cit. 2015-05-18]. Dostupné z: <http://mprc.pku.edu.cn/~tongdong/papers/2010sliu.CJE10.pdf>
- [29] BARR, Michael a Anthony J MASSA. Programming embedded systems. 2nd ed. Sebastopol: O'Reilly, 2006, xxi, 301 s. ISBN 978-0-596-00983-0.
- [30] CATSOULIS, John. Designing embedded hardware. 2nd ed. Sebastopol: O'Reilly, 2005, xvi, 377 s. ISBN 05-960-0755-8.
- [31] MANN, Burkhard. C pro mikrokontroléry: ANSI-C, kompilátory C, spojovací programy - linkery, práce s ATMEL AVR a MSC-51, příklady programování v

- jazyce C, nástroje pro programování, tipy a triky. Vyd. 1. Praha: BEN, 2003, 279 s. ISBN 80-730-0077-6.
- [32] MORTON, Todd D. Embedded microcontrollers. Upper Saddle River, N.J.: Prentice Hall, c2001, x, 694 p. ISBN 01-390-7577-1.
- [33] PINKER, Jiří. Mikroprocesory a mikropočítače. 1. vyd. Praha: BEN - technická literatura, 2004, 159 s. ISBN 80-730-0110-1.
- [34] Atmel. Atmel AVR116: Wear Leveling on DataFlash [online]. Atmel Corporation 2012 [cit. 2015-01-15]. Dostupné z: <http://www.adeptotech.com/wp-content/uploads/doc32194.pdf>.
- [35] PERDUE, Ken. Wear Leveling [online]. 2008 [cit. 2015-01-27]. Dostupné z: [http://www.eettaiwan.com/STATIC/PDF/200808/EETOL\\_2008IIC\\_Spansion\\_A N\\_13.pdf](http://www.eettaiwan.com/STATIC/PDF/200808/EETOL_2008IIC_Spansion_A N_13.pdf).

**SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK**

AD	Analog to Digital
ALU	Arithmetic Logic Unit
ARM	Advanced RISC Machine
CISC	Complex Instruction Set Computing
DA	Digital to Analog
EEPROM	Electrically Erasable Programmable Read-Only Memory
EPROM	Erasable Programmable Read-Only Memory
FTL	Flash Translation Layer
HDD	Hard Disk Drive
I/O	Input/Output
I2C	Inter Integrated Circuit
ID	Identification
JFFS	Journalling Flash File System
JFFS2	Journalling Flash File System version 2
LFS	Log-structured File System
MLC	Multi Level Cell
NTFS	New Technology File System
PROM	Programmable Read-Only Memory
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
ROM	Read-Only Memory
SD	Secure Digital
SLC	Single Level Cell
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory

SSD	Solid State Drive
UART	Universal Asynchronous Receiver / Transmitter
UBIFS	Unsorted Block Image File System
USB	Universal Serial Bus
xD	extreme Digital
YAFFS	Yet Another Flash File System

**SEZNAM OBRÁZKŮ**

<i>Obrázek 1: Von Neumannova architektura [9] .....</i>	14
<i>Obrázek 2: Harvardská architektura [10].....</i>	15
<i>Obrázek 3: Struktura paměťové buňky [16] .....</i>	18
<i>Obrázek 4: NOR flash paměť, v kroužku jedna paměťová buňka [20].....</i>	19
<i>Obrázek 5: NAND flash paměť, v kroužku jedna paměťová buňka [20] .....</i>	20
<i>Obrázek 6: Wear Leveling jako součást FTL nebo systému souborů [22].....</i>	22
<i>Obrázek 7: USB flash disk [23] .....</i>	23
<i>Obrázek 8: Freescale FRDM-KL25Z .....</i>	28
<i>Obrázek 9: Formát jednoho záznamu.....</i>	30
<i>Obrázek 10: Ukázka uložení záznamů v paměti .....</i>	30
<i>Obrázek 11: Funkce pomocného pole proměnných.....</i>	32
<i>Obrázek 12: Struktura knihovny .....</i>	33
<i>Obrázek 13: Obsah paměti v bodě #1 ukázkového projektu č. 1 .....</i>	55
<i>Obrázek 14: Obsah paměti v bodě #2 ukázkového projektu č. 1 .....</i>	55
<i>Obrázek 15: Obsah paměti v bodě #3 ukázkového programu č. 1 .....</i>	55
<i>Obrázek 16: Obsah paměti v bodě #4 ukázkového projektu č. 1 .....</i>	56
<i>Obrázek 17: Obsah paměti v bodě #1 ukázkového projektu č. 2.....</i>	57
<i>Obrázek 18: Obsah paměti v bodě #2 ukázkového projektu č. 2.....</i>	57

**SEZNAM TABULEK**

<i>Tabulka 1: Tabulka nastavitelných konstant .....</i>	39
<i>Tabulka 2: Tabulka chybových kódů .....</i>	40

**SEZNAM ZDROJOVÝCH KÓDŮ**

<i>Zdrojový kód 1: Ukázka programování v assembleru [5] .....</i>	12
<i>Zdrojový kód 2: Ukázka programu v jazyce C .....</i>	13
<i>Zdrojový kód 3: Ukázka funkcí z ovladače paměti flash mikropočítače Freescale MKL25Z128VLK4 .....</i>	41
<i>Zdrojový kód 4: Inicializace virtuální paměti flash v RAM .....</i>	41
<i>Zdrojový kód 5: Vrstva hal .....</i>	42
<i>Zdrojový kód 6: Funkce WriteWord .....</i>	44
<i>Zdrojový kód 7: Funkce WriteValue společná pro všechny zapisující funkce .....</i>	46
<i>Zdrojový kód 8: Funkce ReadWord pro čtení 2 B dat z flash paměti .....</i>	47
<i>Zdrojový kód 9: Funkce defragmentování paměti, část 1/3 .....</i>	49
<i>Zdrojový kód 10: Funkce defragmentování paměti, část 2/3 .....</i>	50
<i>Zdrojový kód 11: Funkce defragmentování paměti, část 3/3 .....</i>	51
<i>Zdrojový kód 12: Ukázka z ukázkového projektu č. 1 .....</i>	54
<i>Zdrojový kód 13: Ukázka z ukázkového projektu č. 2 .....</i>	58

## SEZNAM PŘÍLOH

P1 CD se zdrojovými kódy a ukázkovými programy