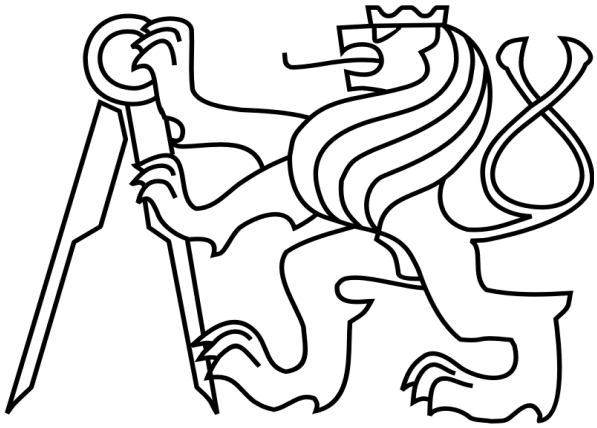
České vysoké učení technické v Praze
Fakulta Elektrotechnická



Diplomová práce

Návrh programového vybavení grafického docházkového terminálu

Bc. Martin Velek

Vedoucí práce: doc. Ing. Jan Fischer, CSc.

Studijní program: Otevřená informatika

Obor: Počítačové inženýrství

Leden 2016

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra řídicí techniky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Martin Velek**

Studijní program: Otevřená informatika
Obor: Počítačové inženýrství

Název tématu: **Návrh programového vybavení grafického docházkového terminálu**

Pokyny pro vypracování:

1. Navrhněte koncepci a stanovte požadavky na funkce grafického docházkového terminálu s mikrořadičem s jádrem ARM.
2. Vytvořte programové vybavení terminálu s využitím operačního systému RTOS zahrnující též podporu grafiky, spolupráci s kartou SD, dalšími periferiemi i komunikaci s rozhraním Ethernet.
3. Při tvorbě programu se orientujte na nástroje typu "Open source".
4. Ověřte bezchybnou funkci programového vybavení a spolupráci jednotlivých programových komponent i v součinnosti s operačním systémem a HW.

Seznam odborné literatury:

[1] YIU, Joseph. The definitive guide to the ARM Cortex-M3. Newnes, 2010,ISBN 78-1-85617-963-8.
[2] SLOSS, Andrew N, Dominic SYMES a Chris WRIGHT. ARM system developer's guide: designing and optimizing system software. Boston: Elsevier/ Morgan Kaufman, 2004,
[3] OSHANA R., KRAELING M. Software engineering for embedded systems: methods, practical techniques, and applications. ISBN 0124159176.

Vedoucí: doc.Ing. Jan Fischer, CSc.

Platnost zadání: do konce letního semestru 2016/2017

L.S.

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 15. 10. 2015

Poděkování

Děkuji panu docentu Fischerovi za věcné připomínky a návrhy při vedení práce, odborné rady a pomoc. Dále pak kolegům v zaměstnání, kteří se podíleli na realizaci docházkového terminálu, v neposlední řadě patří veliké díky manželce a synovi Alexandrovi za jejich podporu během studia a psaní diplomové práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, dne 4.1.2016 .....

Abstrakt

Tato práce se zabývá návrhem nízkoúrovňového programového vybavení grafického docházkového terminálu vhodného pro mikrořadiče založené na architektuře ARM za pomoci využití open-source vývojových nástrojů a softwarových knihoven. Cílem je navrhnout a implementovat systém umožňující rozvrstvení přístupu k jednotlivých hardwarovým částem terminálu, který dovolí aplikačnímu programátorovi nezabývat se technickými aspekty zařízení. V řešení bylo použito metody abstrakce vrstvení ovladačů Periferní API – Ovladače (Device Drivers) – Hardware Abstraction Layer (HAL) Ovladače a definování unifikovaného rozhraní mezi vrstvami ovladačů a Real-time operačním systémem, vrstva OSAL. Vytvořené řešení poskytuje možnost nahradit mikrokontrolér nebo operační systém změnou pouze vrstev HAL a OSAL. Tato vlastnost umožňuje ověřit bezchybnou funkci programového vybavení a spolupráci jednotlivých programových komponent v součinnosti s operačním systémem na osobním počítači s využitím existujících softwarových nástrojů, které nelze spustit na mikrokontrolérech.

Abstract

This diploma thesis deals with design of a low level software of the graphical attendance terminal powered by low-resource microcontroller based on ARM architecture using Open Source development tools. The goal is to design and implement system allowing multi-layered access to the hardware components of the terminal. The solution is based on method of abstract layering defining Peripheral Driver API - Device Drivers - Hardware Abstraction Layer (HAL) drivers and the unified interface between the layers of the drivers and real-time operating system layer (OSAL). Developed solution provides the possibility to replace microcontroller or operating system only by changing the HAL. This feature allows to verify faultless operation of the software and collaboration of individual program components in conjunction with the operating system on a personal computer using existing software tools that cannot be run on microcontrollers.

Obsah

1 Úvod

Evidence docházky pracovníků se stala po zavedení hodinové formy mzdy nezbytnou součástí každého většího zaměstnavatele. Jednotlivé záznamy o docházce jsou důležité pro výpočet mzdy ekonomickým úsekem, a také pro případné kontroly přítomnosti osob na pracovišti. Docházkové terminály jsou zařízeními, které pomáhají zajišťovat snadnou a rychlou evidenci docházky. Mohou být realizovány různými způsoby, prostým sešitem s tabulkou, kde zaměstnanci zapisují příchod nebo odchod, mechanické značkovací hodiny nebo počítačové zařízení s číslicovým zpracováním dat.

Pronikání výpočetní techniky do všech odvětví průmyslu se výrazně projevilo i v rozvoji možností docházkových terminálů, kdy se odstranila největší nevýhoda „analogových" zařízení. Podklady pro výpočet mezd bylo nutné zpracovat z výkazů manuálně vždy ke konci měsíce. Zavedením číslicového zpracování dat přímo u docházkových terminálů a ve spolupráci s osobními počítači výrazně urychlilo a usnadnilo evidenci docházky.

Ve své podstatě je docházkový terminál řízený mikrokontrolérem moderní variantou zařízení z dřívějších dob známého pod lidovým názvem „píchačky". Použití mikrokontroléru s číslicovým zpracováním vstupních dat umožňuje obsáhnout více funkcionality, než jeho analogová varianta, obecný princip však zůstává zachován u obou zařízení, liší se jen způsobem provedení. Základní vlastnosti každého docházkového terminálu jsou:

- Identifikace zaměstnance
- Rozlišení příchodu/odchodu a důvodu přítomnosti/nepřítomnosti
- Časová značka události

Způsobů identifikace zaměstnance je vícero množství, od nejčastěji používaných bezkontaktních karet (RFID) po biometrické údaje (otisk prstů, rozpoznání obličeje), případně kombinace obou variant pro zamezení obcházení identifikace jinou osobou držící identifikační kartu.

Příchod a odchod bývá obvykle u digitálních terminálů doplněn volitelným důvodem změny stavu, např. dovolená, lékař. Výhodou digitálního terminálu je možnost okamžité informace pro zaměstnavatele o počtu lidí v zaměstnání, případně důvodu jejich nepřítomnosti. Dalším přínosem je následné počítačové zpracování docházky a automatická tvorba výkazů práce. Docházkový terminál může také plnit funkci elektrického vrátného, kdy úspěšná identifikace zaměstnance je spojená s aktivací elektrického zámku na vstupních dveřích. Pokud terminál disponuje obecnými výstupy, může být využit pro deaktivaci/aktivaci zabezpečovacího zařízení v závislosti na příchodu nebo odchodu zaměstnanců. Aby mohl digitální docházkový terminál plnit výše uvedené funkce, musí obsahovat patřičné programové vybavení, které umožní obsluhu všech připojených zařízení.

2 Stanovení cílů práce

Cílem této práce je navrhnout a vytvořit nízkoúrovňové programové vybavení grafického docházkového terminálu, které umožní na použité hardwarové platformě splnit nejenom základní vlastnosti docházkových terminálů, ale také nabídnout rozšířenou funkcionalitu, která plyne z využití číslicového zpracování. Záměrem práce není vytvořit aplikaci docházka Vála,P.[1], ale navržené programové vybavení musí umožnit její snadnou realizaci.

Programové vybavení musí ve spolupráci s hardwarem grafického terminálu podporovat splnění těchto funkčních vlastností:

- **Komunikace**

  Terminál, resp. použitý mikrokontrolér, by měl být schopen komunikovat s okolím prostřednictvím rozhraní Ethernet, I2C, SPI a UART.

- **Dotykové rozhraní**

  Dotykové rozhraní by mělo být odolné vůči vnějším povětrnostním vlivům, cenově dostupné na výměnu a použitelné za všech ročních období.

- **Grafický displej**

  Grafický výstup by měl být zobrazovaný na barevném LCD TFT displeji s minimální úhlopříčkou 5“ a rozlišením VGA (640 x 480), hloubka barev 16 bpp (bits per pixel - bitů na pixel).

- **Grafická knihovna**

  Programové vybavení by mělo obsahovat grafickou knihovnu pro zobrazení informací na grafickém výstupu tvořeného barevným displejem.

- **Audio**

  Terminál by měl být schopen generovat zvukové signály o proměnlivé frekvenci.

- **Aktualizace firmware a jeho bezpečné nahrání**

  Terminál by měl umožnit provést nahrání nového firmware a vyrovnat se situací, kdy je během jeho nahrávání přerušeno napájení.

- **Souborový systém**

  Programové vybavení by mělo umožnit ukládat a číst data ve strukturované podobě na vyjímatelné médium.

- **Databáze**

  Programové vybavení by mělo podporovat unifikovaný systém ukládaní, čtení a modifikaci dat.

- **Teplotní a napěťové senzory**

  Pro zajištění bezpečného provozu terminálu by mělo být možné znát podmínky okolního prostředí ve kterém je terminál provozován.

- **Ochrana vykonávání programu**

  Programové vybavení by mělo podporovat způsob, který v případě havárie aplikace umožní její obnovu bez uživatelského zásahu.

- **Rozhraní vstup/výstup**

  Kromě komunikačních linek by měl terminál podporovat ovládání vstupu a výstupu (I/O).

- **Hodiny reálného času**

  Terminál by měl být schopen udržovat aktuální datum a čas i při vypnutém napájení. Programové vybavení by mělo podporovat změnu letního a zimního času a jeho editaci.

- **Víceúlohové prostředí (multitasking)**

  Programové vybavení by mělo podporovat prostředí, ve kterém jsou úlohy jsou rozděleny do nezávislých programových procesů s využitím plánovače úloh, namísto cyklické smyčky.

- **Programové rozhraní API**

  Programové vybavení by mělo obsahovat abstraktní uživatelské API k jednotlivých hardwarovým částem terminálu a nabízeným službám.

Před samotným návrhem je nutné rozhodnout, který mikrokontrolér a jaký druh software bude vybrán k realizaci programového vybavení:

- **Výběr mikrokontroléru založeného na architektuře ARM**

  Architektura ARM nabízí široké množství druhů jader, lišící se spotřebou, vlastnostmi a cenou. Cílem je vybrat model, který umožní splnit funkcionalitu kladenou na programové vybavení za předpokladu minimalizace ceny, složitosti návrhu hardware, aj.

- **Výběr open-source programů**

  Open-Source Software (OSS) oblast nabízí mnoho nástrojů a knihoven k realizaci programového vybavení. Bude nutné vybrat OSS, který pokrývá požadavky programového vybavení, případně s mírnými modifikacemi, nebo navrhnout nové řešení. Týká se vývojových nástrojů, knihoven a operačního systému.

Při návrhu programového vybavení bude nutné se zabývat:

- **Příprava open-source nástrojů**

  Open-source nástroje vyžadují přizpůsobení pro platformu grafického terminálu. Na základě vlastností programového vybavení bude nutná jejich modifikace.

- **Návrhem vnitřních struktur a jejich propojení**

  Programové vybavení bude tvořit systém ovladačů, software knihoven a pomocných služeb. Pro snadné využívaní prostředků nabízených p.v. bude nutné navrhnout jeho vhodnou vnitřní strukturu a definovat rozhraní, které bude zajišťovat jeho spolupráci.

- **Koncepcí hardware docházkového terminálu**

  Programového vybavení by mělo zavést požadavky na koncepci hardware terminálu, tj. minimální obsažnost součástkové základny, způsob zavádění systému a jeho inicializace.

- **Návrhem způsobu testování**

  Testování programového vybavení by mělo ověřit jeho bezchybnou funkci a spolupráci jednotlivých programových komponent i v součinnosti s operačním systémem a hardware. Způsob návrhu programového vybavení by měl umožnit využít testovací softwarové nástroje, které jsou doménou osobních počítačů a nelze je aplikovat na mikrokontrolérech.

3 Motivace

Zadání práce vzniklo z požadavku nahradit dosluhující generaci (grafických) docházkových terminálů ve společnosti IMA, s.r.o. (http://www.ima.cz). Společnost IMA (Institut mikroelektronických aplikací), vzniklá privatizací Tesla Eltos se zabývá vývojem a výrobou přístupových a docházkových systémů již od počátku 90. let. Ve svém portfoliu má několik typů docházkových terminálů lišících se svým technickým vybavením.

Docházkový terminál CKP-3gd s grafickým displejem a membránovou klávesnicí uvedený na obrázku č. 1, vlevo, je postaven na jednočipovém mikroprocesoru s jádrem Intel 8052 [2]. Ačkoliv jde v současnosti o stále hojně využívaný typ mikrokontroléru, jeho ISA naráží na výkonové a funkční limity. Rozšíření komunikačních vlastností terminálu se musí řešit pomocí externích převodníků, např. Ethernet-Serial, které zvyšují výslednou cenu produktu. Tato generace terminálů obsahuje programové vybavení napsané v assembleru, jenž omezuje spolupráci mezi vývojáři a při rozšiřování nových funkcionalit.

Grafický docházkový terminál JIN-02 s dotykovým rozhraním uvedený na obrázku č. 1, vpravo, je postaven na minipočítači s architekturou x86 a běžícím prostředím GNU/Linux. Kód pro daný produkt je napsán v jazyce C s využitím API Linuxu. Nevýhodou tohoto terminálu je značný cenový rozdíl kvůli použité platformě oproti CKP-3gd a závislosti na externích subjektech vytvářející Linuxovou distribuci.

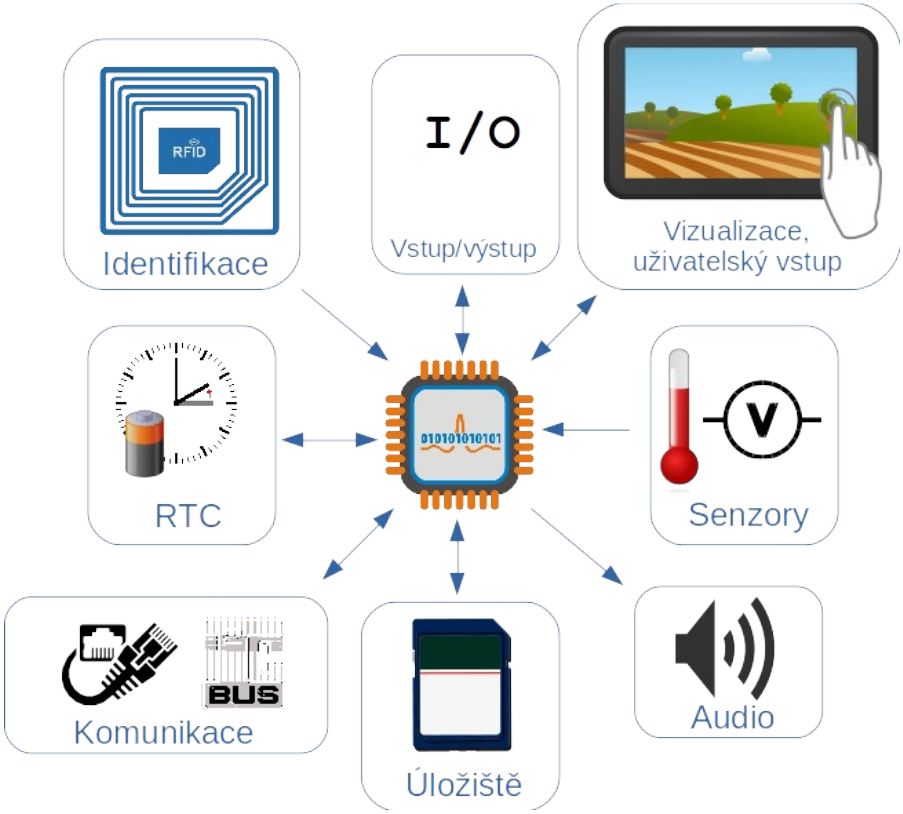
Z pohledu vývojáře jsou oba produkty vzájemně nekompatibilní, ať už z hlediska architektury, tak použitého programovacího jazyka. Splněním cílů deklarovaných v předchozí kapitole se umožní vytvoření nového grafického docházkového terminálu, který bude obsahovat modernější architekturu mikrokontroléru (u terminálu CKP-3gd) a také obsahovat zcela nové způsoby týmové spolupráce při psaní programů, unifikace vývojového prostředí, a to vše s minimálními pořizovacími náklady na vývojové prostředky.

Obrázek 1: Docházkové terminály IMA,s.r.o., vlevo CKP-3gd, vpravo JIN-02

4 Rozbor řešení

Analýza požadavků na funkci grafického docházkového terminálu a jeho programového vybavení vychází z praktických požadavků získaných ve společnosti IMA při provozování současné generace docházkových terminálů. Krom základních vlastností očekávaných od docházkového terminálu, vzešla část požadavků jako podněty na reakce zákazníků. Některé požadavky mohou proto působit nestandardně, ale mají své opodstatnění. Část souvisí s již existující firemní infrastrukturou, která musí být zpětně podporována, jiné byly získány na základě empirických zkušeností.

V této diplomové práci je termín systém definován jako souhrn technických a funkčních vlastností grafického terminálu. Z technického hlediska se jedná o použitý hardware a nízkoúrovňový software, z funkčního hlediska se jedná o možnosti, které je schopné systém nabídnout po stránce technické a jejich způsobu implementace. Řešením návrhu programového vybavení grafického docházkového terminálu, jehož jednotlivé bloky jsou znázorněny na obrázku č. 2, je systém ovladačů, software knihoven a pomocných služeb, vzájemně propojených programovým rozhraním.



Obrázek 2: Funkční bloky grafického terminálu

4.1 Funkční vlastnosti programového vybavení

Vlastnosti programového vybavení jsou rozděleny do kategorií a každá kategorie pokrývá určitou oblast funkcionality. Vlastnosti jsou také zároveň požadavky, které musí programové vybavení splňovat, a ty určují omezující podmínky pro výběr mikrokontroléru, software knihoven a hardware. Cílem požadavků na programové vybavení je umožnit vybrat cenově přijatelný mikrokontrolér obsahující v nejlepším případě všechny potřebné periférie, zjištění dostupnosti open-source knihoven a rozhodnutí, jaký druh software bude nutné vyvinout/nakoupit.

Při stanovování služeb zajišťovaných programovým vybavením je nutné stanovit režim správy uživatelů. Jednotlivé režimy vyžadují odlišné hardwarové vybavení a strukturu programového vybavení.

Z hlediska správy lze docházkové systémy rozdělit na

- Centrálně řízené
- Autonomní
- Kombinované

Centrálně ovládané terminály jsou vhodné zejména ve velkých organizacích nebo při vysokém počtu nasazení zařízení. Řízená správa umožňuje rychle a efektivně reagovat na změny, např. při vysoké fluktuaci zaměstnanců. Rovněž lze při ztrátě identifikačního klíče snadno zablokovat přístup pouhým odstraněním z centrálně uložené evidence. Nevýhody tohoto systému je technická složitost zařízení zajišťující ovládání a ověřování, údržba komunikačních linek a zajištění školené obsluhy. Vyšší pořizovací cenu lze vykompenzovat integrací do firemního účetnictví, kdy lze sledovat docházku pracovníků a následně automaticky zpracovat mzdové podklady. Výhodou systému je odolnost soukromých dat při odcizení terminálu, neboť jsou údaje o příchodu/odchodu zpracovávány a ukládány mimo samotný terminál.

Autonomně řízené terminály fungují zcela nezávisle a ke svému provozu potřebují pouze energetický zdroj. Jsou vhodné v prostředí, kde se neočekává velké nasazení a aktuální stav zaměstnanců není vyžadován. Data o docházce se ze zařízení sbírají obvykle jednou měsíčně pro podklady výplaty mezd. Tyto terminály musí obsahovat datové úložiště, které pojme dostatečný počet záznamů. Při odcizení terminálu hrozí únik soukromých dat o docházce.

Kombinované terminály sdružují výhody předchozích řešení a odstraňují jejich technické nedostatky. Centrálně řízené terminály se stávají v případě výpadku komunikace nefunkčními a autonomní jsou omezeny kapacitně. Kombinované terminály se primárně chovají stejně jako centrálně řízené, pouze v případě výpadku komunikace, či jiné neočekávané situace se přepnou do nezávislého režimu. Zde je uložena sada pravidel pro nouzový režim a jejich používání záleží na specifických podmínkách uživatelů.

4.1.1 Komunikace s nadřazeným systémem, periériemi

Přítomnost komunikační rozhraní vychází z nutnosti spolupráce s centrálním řídicím systémem, pokud je použit a také s periferními zařízeními nacházejícími se uvnitř terminálu.

Pro komunikaci s nadřazeným systémem se nejčastěji používá sběrnice RS485. Jelikož je tato sběrnice obvykle implementována pomocí periférie UART a RS485 budiče, mikrokontrolér musí obsahovat hardwarový UART obvod. Sběrnice RS485 ale neřeší na fyzické vrstvě pokročilou kontrolu konzistence přenášených dat, detekci kolizí a podporu prioritních zpráv, tyto vlastnosti musí být implementovány v rámci softwarového protokolu. Tyto nedostatky by mohly být řešeny přímo na fyzické vrstvě, pokud by MCU podporoval CAN rozhraní.

Rozhraní Ethernet je dnes de-facto standard, neboť s rozvojem Internetu věcí (IoT – Internet of Things [3]) se stává toto rozhraní standardní součástí téměř všech produktů vč. spotřební elektroniky (televize, ledničky, tiskárny aj.). Pokud je rozhraní implementováno, dochází k výraznému nárůstu požadavků na softwarové vybavení, ale zároveň se zvyšují jeho možnosti. Samotný Ethernet, případně Wifi neumožňuje využít všech možností IoT. K tomu je nutné zahrnout software obsluhující rodinu protokolů TCP/IP. S jeho pomocí lze komunikovat s nadřazeným systémem, dálkově spravovat, konfigurovat a diagnostikovat terminál. Mikrokontrolér musí obsahovat minimálně EMAC periférii (Ethernet Media Access Controller). Pro usnadnění vzdáleného přístupu by mělo programové vybavení obsahovat HTTP server umožňující poskytovat statické stránky/soubory, dynamicky generované stránky a přijímat data. Použitím HTML formátu při přenosu dat není nutné pro dálkovou správu vytvářet speciální program, ale postačí běžný WWW prohlížeč.

Pro získání sériového čísla z RFID karty pro účely identifikace se používají RFID čtečky vybavené rozhraním Wiegand [4], jedná se o jednoduchý protokol založený na dvou vodičích H a L. Pro jeho implementaci postačuje mikrokontrolér generující přerušení při detekci vzestupné/sestupné hrany na I/O pinech a časovacím obvodem. V případech, kdy je vyžadována složitější komunikace se čtečkou, např. přístup k vnitřním sektorům na kartě, mívají čtečky sériové rozhraní RS232.

Kromě vstupně/výstupních komunikačních linek musí programové vybavení podporovat ovládání vstupu a výstupu na pinech. Výstupy lze využít k ovládání relé elektrického zámku na dveřích nebo Elektronického Zabezpečovacího Systému (EZS), který se automaticky aktivuje po odchodu posledního zaměstnance. Vstupy lze využít pro detekci neoprávněného zásahu, odejmutí od stěny nebo otevření krabičky, ve které je terminál umístěn (tamper senzory).

4.1.2 Rozhraní mezi koncovým uživatelem a terminálem

U dotykového rozhraní docházkového terminálu není kladen důraz na absolutní přesnost detekce dotyku, dokonce nebývá využívána funkce multi-touch (více-dotykové ovládání). Vstup zahrnuje volbu od uživatele, nastavení zamýšlené akce, které by nemělo obsahovat složité listování v menu, ale být maximálně na dva pohyby/dotyky – volba směru a důvodu.

Resistivní dotykové rozhraní je přes zjevné nevýhody oproti kapacitnímu (případně jiných technologií) cenově i technologicky zatím jedinou vhodnou variantou dostupnou na trhu. Je levné, snadno se vymění nové za poškozený kus a funguje i v zimním období, kdy uživatelé mohou nosit rukavice a kapacitní senzory by nemusely fungovat bezchybně. U resistivní dotykové obrazovky je nutné, aby mikrokontrolér obsahoval ADC převodník. Programové vybavení pro dotykové rozhraní by mělo minimálně podporovat tyto funkce:

- kalibrace
- filtrace naměřených hodnot
- konverze na souřadnicový systém

Grafický výstup by měl být zobrazovaný na barevném LCD TFT displeji s minimální úhlopříčkou 5“ a rozlišením VGA (640 x 480), hloubka barev 16 bpp (bits per pixel - bitů na pixel). U displejů s menší úhlopříčkou se objevuje problém při vstupu od uživatele pomocí dotykové obrazovky. Prvky na obrazovce jsou těsně naskládané vedle sebe a nezbývá mnoho prostoru pro toleranci chyb dotykového rozhraní. Nižší rozlišení způsobuje viditelné kostičkování u obrázků s oblými hranami. Pro snadnou manipulaci s grafickými daty, by měl mikrokontrolér obsahovat LCD rozhraní a odpovídající velikost videopaměti.

Zobrazování grafických informací by mělo být zajištěno grafickou knihovnu obsaženou v programovém vybavení. Tato knihovna by měla podporovat rozšířené funkce grafického prostředí, tj. nejen zobrazovat rastrové obrázky na daných souřadnicích, ale také vytvářet, rušit a spravovat grafické objekty, tzv. widgets [5].

Minimální podporované funkce graf. knihovny:

- podpora českých znaků (fonty)
- podpora grafických formátů (PNG nebo JPEG)
- rozhraní pro vstup (dotyková obrazovka – touchscreen)

Výstup, tj. zobrazení nastavené akce k uživateli, musí být srozumitelné, jednoznačné a obsahovat pouze relevantní informace vztažené k zamýšlené akci.

Terminál musí být schopen generovat zvukové signály o proměnlivé frekvenci. Jednotlivé tóny se přiřazují k akcím od uživatele např. volba příchodu nebo odchodu, což umožňuje ovládat terminál i lidem se zrakovým postižením.

Zvukový výstup by měl podporovat minimálně tyto funkce:

- nastavení hlasitosti
- doba trvání tónu
- volba frekvence

4.1.3 Bezpečnostní a bezúdržbové funkce

Docházkový terminál pracuje v autonomním režimu a jeho provozování by mělo být bezúdržbové s minimálním počtem uživatelských zásahů, pouze zapnutí a vypnutí. Ostatní zásahy, např. resetování zařízení z důvodu nereagování, jsou považovány za nežádoucí stav a programové vybavení musí zahrnout postupy při své činnosti, které se tyto negativní situace eliminují.

Při aktualizaci stávajícího firmware může dojít k situaci, kdy je během aktualizační procedury přerušeno napájení. Po neúspěšném nahrání nového firmware se terminál nesmí dostat do situace, kdy není možné opětovně zahájit nahrání nového firmware nebo je terminál zablokován.

Zastavení chodu vykonávání programu, ať už z důvodu špatného software nebo externími vlivy je nepřípustné. MCU musí podporovat ochranný Watchdog timer nezávislý na vnitřních hodinách mikrokontroléru nebo obsahovat externí vstupní pin RESET pro připojení externího Watchdog obvodu, aby v případě neočekávané chyby programu došlo k resetování systému.

Teplota mimo rozsah pracovních podmínek nebo kmitání napětí může mít za následek poškození dat na paměťovém médiu, anebo se systém stane nestabilním. Mikrokontrolér by měl proto být vybaven ADC převodníkem pro kontrolu stability napájení a měření teploty (např. termistorem).

4.1.4 Souborový systém

Programové vybavení musí umožnit ukládat a číst data ve strukturované podobě na vyjímatelné médium, které bude možné přečíst také na osobním počítači. Tento požadavek, aby data systému byla oddělena od fyzického zařízení, není pouze bezpečnostní, ale také jde o usnadnění výměny vadných zařízení v prostorách zákazníka. Nejvhodnější externí paměťové médium pro vestavěné systémy, v době psaní diplomové práce, v poměru cena/rozhraní/rozměr/kapacita je microSD karta. Mikrokontrolér by měl obsahovat SD/MCI rozhraní pro komunikaci s kartou. Specifikace SD [7] sice umožňuje přepnout kartu do režimu komunikace přes sběrnici SPI, ale dojde tím k omezení vestavěných kontrol na straně mikrokontroléru, zejména hardwarové implementace kontrolních součtů, jenž je součástí SD/MCI rozhraní.

Spolu s požadavkem na výměnu dat mezi terminálem a osobním počítačem pomocí vyjímatelného úložiště, programové vybavení by mělo podporovat unifikovaný systém ukládaní, čtení a modifikaci dat, aby na straně PC nebylo nutné používat specializovaný program pasující pouze ke konkrétnímu typu zařízení. Z tohoto důvodu je nutné použít systém řízení báze dat (SŘBD), který je implementovatelný jak v mikrokontroléru, tak v PC.

4.1.5 Hodiny reálného času

Datum a čas události je jednou ze základních vlastností poskytovanou docházkovým terminálem. Mikrokontrolér by měl proto být vybaven obvodem reálného času s možností alternativního zdroje napájení (baterie, superkapacitor). Programové vybavení musí

podporovat automatickou změnu letního a zimního času a synchronizaci pomocí NTP [6] protokolu.

4.1.6 Víceúlohové prostředí (multitasking)

Ze stanovených cílů pro programové vybavení plyne snadná realizace aplikace docházka. Konečný produkt grafický docházkový terminál se z pohledu software skládá z programu docházka, která zajišťuje logiku aplikace a programového vybavení, které zajišťuje služby využívané aplikací na konkrétním druhu terminálu. Tento pohled vede na rozdělení rolí vývojářů, aplikační a systémové.

Tyto obvykle nesourodé skupiny vývojářů v přístupu k řešení úloh vede k nutnosti použití operačního systému, kdy jednotlivé úlohy jsou rozděleny do nezávislých programových procesů s využitím plánovače úloh, namísto cyklické smyčky [12]. Aby byla zachována minimální odezva funkčního systému z pohledu uživatele, která činí přibližně 100 milisekund [8], nelze zvolit obecné operační systémy typu GNU/Linux či MS Windows, ale je nutné použít RTOS (Realtime Operating System). Grafický terminál není zařízení, které při překročení termínu dokončení úlohy ohrozí život uživatelů. Tato klasifikace umožňuje vyhnout se nutné certifikaci programového vybavení, hardware a také požadavky na RTOS nejsou tak přísné, jako u systémů vyžadující hard-realtime kernel [9].

4.1.7 Aplikační programové rozhraní - API

Programové rozhraní je tvořeno sadou deklarovaných funkcí s definovanými vstupy a výstupy, tím je umožněna záměna vnitřní funkcionality a zároveň jsou skryty složité detaily implementace. API zajišťuje i určitý stupeň abstrakce, kdy zařízení mající podobné vlastnosti jsou sdruženy do skupin. Například z programátorského hlediska jsou paměti typu I2C EEPROM, SPI Flash ekvivalentní, na obě lze nahlížet jako na zařízení, které podporují čtení a zápis bajtů na adresy.

Rozhraní poskytnuté aplikačnímu programátorovi programovým vybavením

- OSAL – Operating System Abstraction Layer

  Abstraktní vrstva rozhraní k operačnímu systému.

- Knihovny

  Rozhraní softwarových knihoven je zachováno stejné, jak jej nabízí poskytovatel knihovny.

- Periferní API

  Abstraktní vrstva rozhraní přístupu k periferiím mikrokontroléru a zařízením

4.2 Výběr mikrokontroléru založeného na architektuře ARM

Výběr mikrokontroléru je strategické rozhodnutí mnohdy ovlivňující výsledek projektu. Podcenění parametrů může vést k nedostatečnému výkonu zařízení, přecenění k navyšování ceny konečného produktu. Dalším kritickým aspektem výběru je výrobce mikrokontroléru.

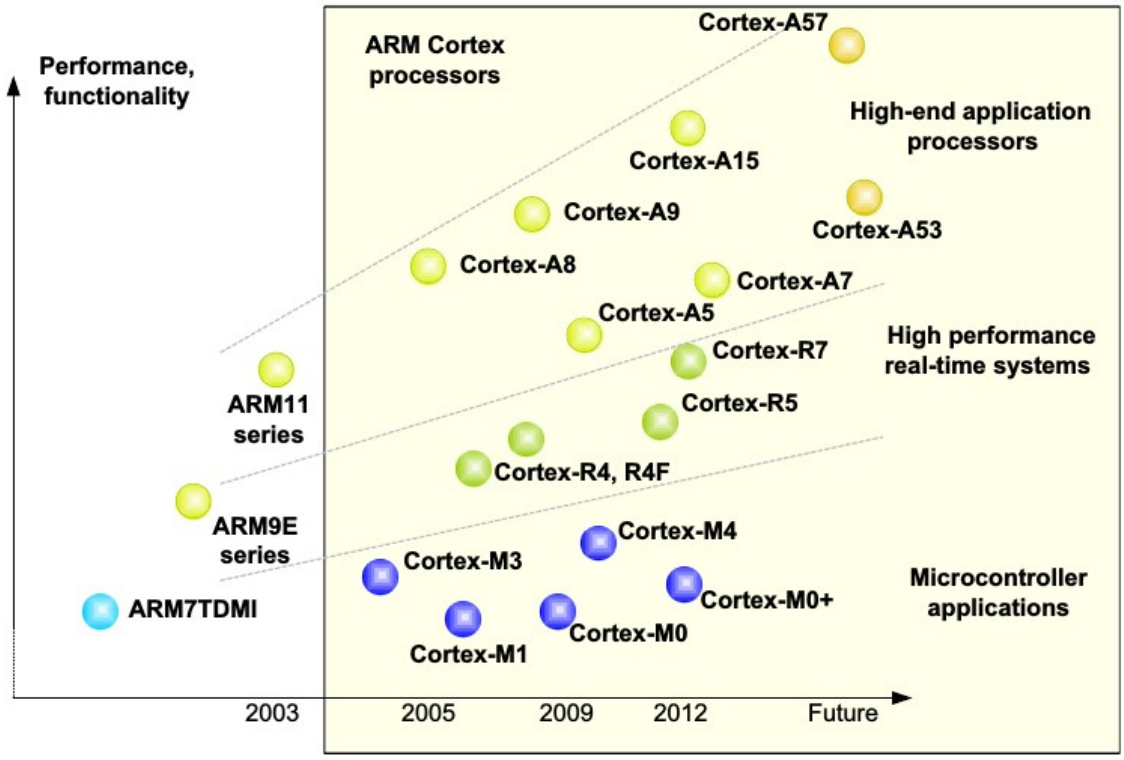
Většina současných velkých výrobců elektronických součástek jako např. Texas Instruments, Atmel, Microchip Technology, Freescale Semiconductor nabízí produktovou řadu mikrokontroléru s proprietární ISA, vhodných pro daný typ aplikace. Výběrem jejich produktu mnohdy dochází k situaci, kdy se zákazník stane závislým na konkrétním výrobci (vendor lock-in) a přechod ke konkurenci, byť nabízející výkonnější, levnější produkt je nemožný z časových a hlavně finančních důvodů. Nejhorším snem pro vývojáře je náhlé zjištění ukončení výroby nebo podpory daného produktu. Tyto nevýhody částečně eliminují mikrokontroléry, které implementují jádro od společností ARM. Společnost ARM sama mikrokontroléry nevyrábí, pouze prodává know-how a podklady pro výrobu ostatním společnostem (syntenizovatelné VHDL, IP soft-core). Výhodou tohoto způsobu podnikání je velký výběr mikrokontrolérů s daným jádrem od mnoha výrobců, dokonce i od technologických lídrů uvedených výše, které se liší pouze implementovanými periferiemi, velikostí pamětí, maximální frekvencí apod. Instrukční sada, architektura zůstává stejná a uživatel může přecházet mezi jednotlivými výrobci bez velkých nákladů na pořízení nového vývojového prostředí a nezbytných školeních. Mikrokontroléry s jádrem ARM se staly de-facto standardem, pokud jde o využití ve víceúčelových vestavěných systémech.

Historie společnosti ARM sahá až do roku 1990 [10]. V roce 1991 byl uvedena řada ARM6, ovšem nejpopulárnější se staly ARM7, ARM9 (1998) a ARM11 (2002). V roce 2004 přichází na trh řadu Cortex-M3, který představil zlom na poli 32-bitových mikrokontrolérů. V současné době, pokud není nutné zachovávat zpětnou kompatibilitu, je výhodné použít řadu Cortex-A/M/R nabízejí vyšší výkon (DMIPS/MHz), nižší spotřebu (mW/MHz) a vylepšenou podporu Debug (CoreSight) oproti řadám ARMx. Výrobci je však stále vyrábějí, ale jedná se spíše o setrvačné tendence kvůli masové rozšířenosti. Podobné příčiny má i výroba mikrokontrolérů s jádrem 8051.

Při výběru mikrokontroléru řady Cortex se nabízí několik variant (profilů) v závislosti na požadavcích zařízení:

- Profil A je určen pro zařízení vyžadující vysoký výpočetní výkon, spouštění komplexních aplikací běžící pod operačním systémem GNU/Linux (Android) nebo MS Windows. Procesory dosahují frekvence řádu Gigahertz, obsahují složitou správu virtuální paměti, správu paměti (MMU) a prostředí pro bezpečné vykonávání aplikací (secure program execution environment). Cílové produkty zahrnují high-end chytré telefony, televize, entertaiment systémy.
- Profil R je určen pro zařízení vyžadující vysoký výpočetní výkon, real-time požadavky, nízkou latenci a vyšší stupeň bezpečnosti a spolehlivosti. Cílové produkty zahrnují řídicí komponenty pevných disků, zpracování komplexních signálů, automotive.
- Profil M je určen pro vestavěná (embedded) zařízení, kde není rozhodující vysoký výkon, ale nízká cena, nízká spotřeba a vysoká responsivita. Cílové produkty zahrnují přesně ty produkty, které nejsou pokryty profily A a R, např. grafické terminály, hračky, audio kodeky.

Obrázek č. 3 (zdroj [10]) znázorňuje rozložení tříd jednotlivých ARM modelů, podle data vydání a cílové aplikace.



Obrázek 3: Historie vývoje ARM jader a oblasti použití

Pro nové produkty je preferovaná řada Cortex, volba profilu závisí na použitém programovém vybavení a deklarované funkčnosti.

4.3 Výběr open-source programů

Dalším aspektem při návrhu programového vybavení, případně jeho částí, je rozhodnutí, zdali využít GNU/Linux nebo začít s vývojem zcela nového (tzv. from scratch). GNU/Linux je plnohodnotný operační systém, tj. zahrnuje již definované struktury pro ovladače, paměť, souborové systémy aj., s rozsáhlou mírou abstrakce pro pokrytí potřeb různého druhů hardware. Tato komplexnost, v roce 2015 je počet řádek kódu ve verzi 4.1 více jak 18 miliónů [13], v API vytváří latence při volání služeb jádra a obsluhy přerušení. Vysoká provázanost s jádrem při psaní ovladačů nutí vývojáře znát nejenom strukturu a chování periférie, ale také i samotného kernelu.

Nasazení GNU/Linux navíc vyžaduje procesor s podporou MMU (Memory Management Unit) [11], která je ale přítomna pouze ve vyšších třídách mikrokontrolérů, např. Cortex-Ax. Existují sice verze GNU/Linux, které MMU nevyžadují, např. uClinux, ale dochází tím k výraznému omezení v oblasti bezpečnosti procesů v rámci paměťových domén [16].

Provázanost jádra spolu se strukturou ovladačů se objevuje také u alternativních open-source RTOS, např. RTEMS (https://www.rtems.org/) nebo eCos (http://ecos.sourceware.org/).

V případě vlastního vývoje programového vybavení není nutné vytvářet veškerý software od začátku, ale lze využít metody vhodného pospojování a přizpůsobení již existujícího OSS do jednoho celku. Tento postup vyžaduje pouze definování vazeb v systému a jejich implementaci.

5 Použité řešení

V této kapitole je popsán výběr se zdůvodněním použitého hardware a software, který byl stanoven na základě deklarovaných funkcí grafického docházkového terminálu. Při výběru mikrokontroléru byla hlavním kritériem obsažnost co největšího počtu periférií uvedených ve vlastnostech programového vybavení v kapitole č. 4.1. Prioritní rolí u vývojových nástrojů je rozšířenost mezi komerčními subjekty, které je využívají ve svých produktech, a také dostupnost vhodných rozšíření určených pro vestavěné systémy. Použité knihovny a RTOS byly vybrány z oblasti OSS, kdy rozhodovacím faktorem byla jejich snadná portace na vestavěná zařízení, tj. nízké nároky na datovou a programovou paměť, nepoužívání rekurzivních funkcí a časově náročných algoritmů a malá provázanost na existující API.

5.1 Výběr typu mikrokontroléru

Na základě požadavků na funkcionalitu byla stanovena následující minimální kritéria pro architekturu CPU:

- frekvence hodin CPU 100 MHz
- interní paměť programu 512 KB
- interní datová paměť 64 KB
- DSP (Digital signal processing) funkce ani FPU (Floating Point Unit) nebudou použity
- nepoužívání ASM v rutinách přerušení
- ochrana paměti proti neoprávněnému přístupu/vykonávání kódu
- podpora virtuální paměti není potřeba
- datová, instrukční cache není potřeba
- více výpočetních jader není potřeba
- nízká spotřeba energie

Dále byla stanovena následující minimální kritéria pro vybavenost periferiemi:

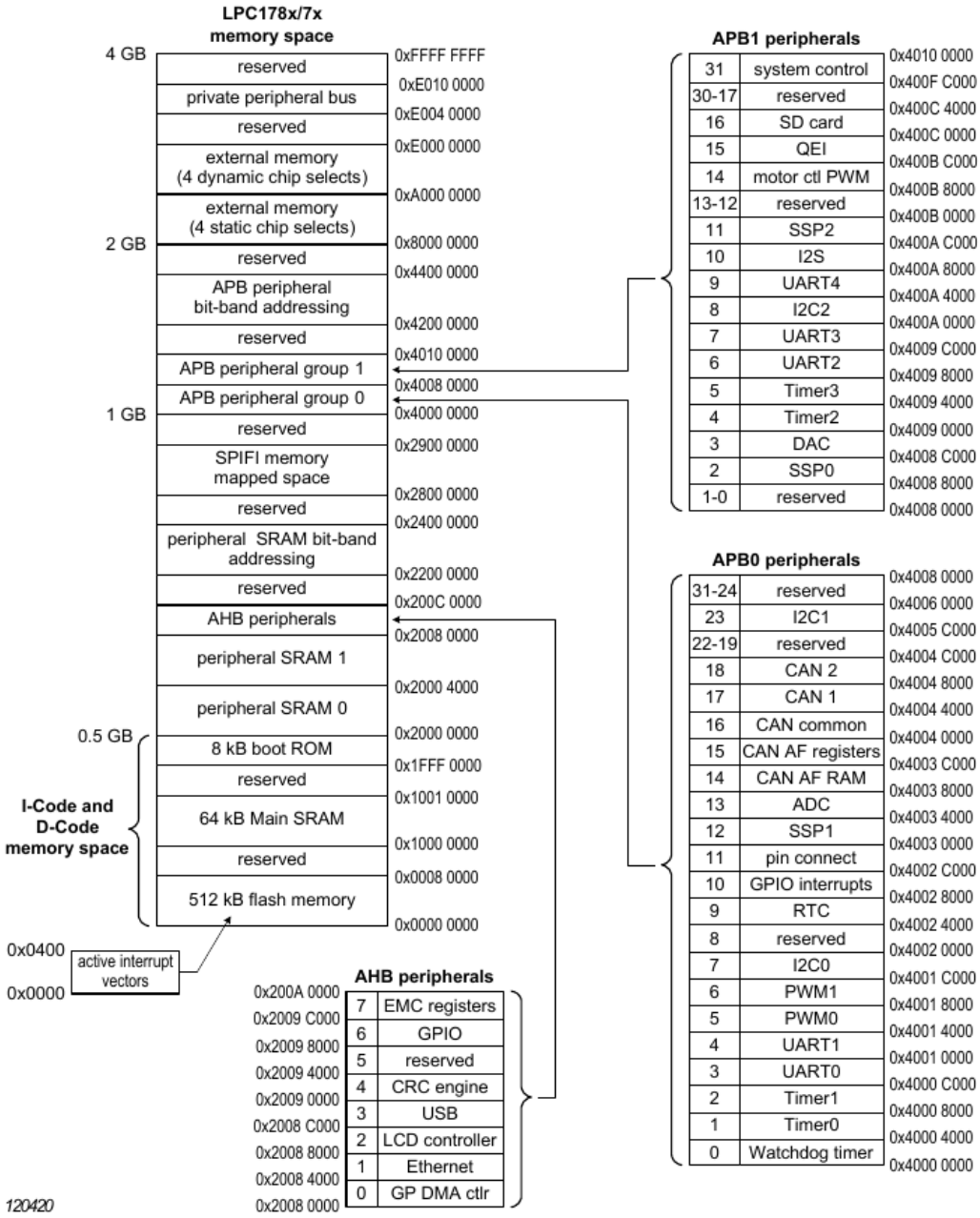
- Rozhraní pro připojení externí paměti (EMC)
- Ethernet (EMAC)
- Rozhraní SD/MMC pro připojení SD karty
- Minimálně 2x UART sériové rozhraní
- Integrovaný grafický LCD kontrolér s paralelním výstupem pro TFT displeje
- Sběrnice CAN výhodou
- Integrovaný Watchdog obvod s nezávislým časovačem

- Periférie PWM pro generování zvuku, tónů
- Rozhraní SPI pro komunikaci s pamětmi
- Periférie ADC pro měření teploty a stavu napájení
- Obvod RTC s podporou nezávislého napájení na hlavním
- Periférie časovačů pro operační systém a potřeby měření času událostí v milisekundovém rozlišení
- Hardwarová ochrana paměti (Memory Protection Unit)

Z požadavku na zobrazovací displej a jeho parametry vyplývá minimální velikost paměti pro video data (frame buffer), která činní 600 KB (640 * 480 * 2 B). Pokud je použita technika double-buffering (viz kapitola č. 7.7.1), vzroste velikost na 1,2 MB. Najít mikrokontrolér s dostačující velikostí interní paměti není možné, a proto je nutné připojit externí datovou paměť SDRAM s mapováním do adresního prostoru mikrokontroléru.

Požadavek na grafické rozhraní a HTTP server obnáší potřebu úložného prostoru, kde budou umístěny HTML soubory, kaskádové styly a obrázky. Velikost prostoru pro data nelze předem přesně odhadnout, mohou se pohybovat až v řádu MB. Z bezpečnostně-funkčního hlediska nelze též uložit tato data na vyjímatelné médium. Funkčnost terminálu by byla výrazně omezena v případě ztráty či poškození SD karty. Připojení externí sériové SPI FLASH paměti znemožňuje využití zbylého prostoru pro programový kód a přináší nutnost serializace/de-serializace dat. Z tohoto důvodu je nutné k mikrokontroléru připojit externí programovou paměť FLASH mapovanou do jeho adresního prostoru. Aby bylo možné využít tuto paměť i pro program a vykonávání zní kódu, je nutné vybrat typ NOR namísto NAND [14].

5.1.1 Vybraný mikrokontrolér



Obrázek 4: Paměťová mapa LPC1788

Mikrokontroléry s výše uvedenými parametry vyrábí všechny renomované firmy, Texas Instrument, ST Microelectronic, NXP aj., liší se velikostmi pamětí a počtem dostupných periférií. Pro grafický docházkový terminál byl vybrán mikrokontrolér LPC1788 [17] od společnosti NXP. Výběr konkrétního typu byl ovlivněn zkušenostmi s předchozí generací mikrokontrolérů této společnosti, LPC2478 (ARM7TDMI-S), kdy se NXP podařilo zachovat

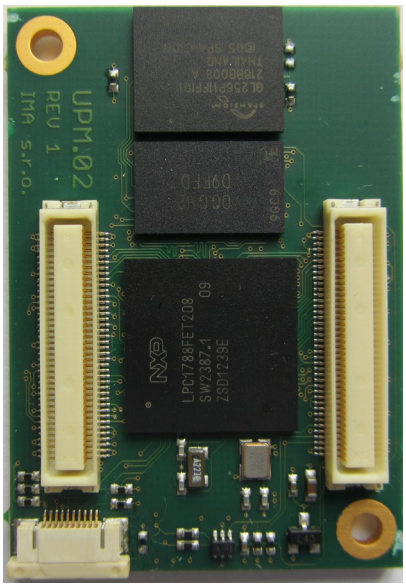
stejné periférie, jejich způsob nastavení a přitom se využilo výhod nového jádra. Současně je LPC2478 pinově kompatibilní s LPC1788.

LPC1788 s jádrem Cortex-M3 je mikrokontrolér s vysokou mírou integrace, nízkým příkonem (spotřeba závisí na konfiguraci, MCU umožňuje vypínat napájení jednotlivý periferiím, pokud nejsou používány) a běžícím až do frekvence 120 MHz. Obsahuje 512 KB programové paměti FLASH, 96 KB datové paměti SRAM, 4 KB EEPROM a veškeré periférie stanovené v požadavcích. Seznam periférií spolu s paměťovou mapou je uveden na obrázku č. 4. Mikrokontrolér umožňuje měnit programový kód přepsáním interní FLASH paměti pomocí IAP (In-Application Programming) funkcí. Kromě JTAG rozhraní obsahuje také nové ladící rozhraní od společnosti ARM, Serial Wire Debug (SWD), Serial Wire Output (SWO) a Emulation Trace Module (ETM) [10].

Spojením požadavků na mikrokontrolér a externí paměť vznikl Universální Procesorový Modul (UPM) zobrazený na obrázku č. 5, který obsahuje:

- Mikrokontrolér LPC1788
- 16 MB SDRAM paměť
- 16 MB NOR Flash paměť
- Krystaly 12 MHz a 32,768 KHz
- JTAG/SWD debug konektor
- Konektory s vyvedenými piny MCU

Modulární řešení terminálu za použití několika desek plošných spojů přináší finanční úsporu v celkové ceně zařízení terminálu. UPM byl navržen na osmivrstvé DPS z důvodu použití BGA pouzder u čipů za účelem úspory místa. Deska s periferiemi, konektory, aj. tak může být dvou nebo čtyřvrstvá. Pokud dojde k poškození jedné z desek, lze je snadno nahradit prostou výměnou poškozené části. Za podmínky neměnné konfigurace pinů u konektorů, lze snadno změnit hw vybavení na modulu, pokud by v budoucnu vznikly nové požadavky, které použitý mikrokontrolér není schopen obsloužit, aniž by byla nutná hardwarová úprava ostatních částí. Nevýhodou modularizace je navýšení ceny součástek o konektory a oddělený výrobní proces při výrobě DPS a osazování.



Obrázek 5: Procesorový modul s externími paměťmi

5.2 Výběr open-source vývojových nástrojů

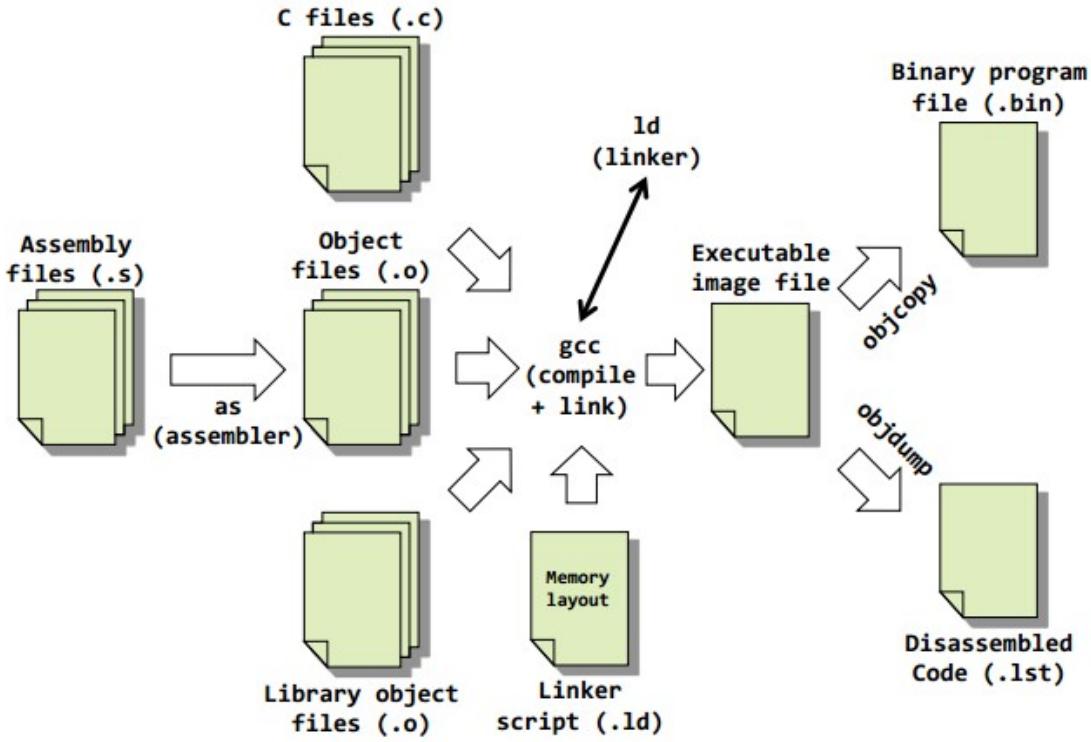
Vývojové nástroje pokrývají kompletní řetězec operací od vytvoření zdrojového kódu, kompilaci, linkování, nahrání do zařízení a ladění. OSS (Open Source Software) poskytuje kvalitní nástroje pokrývající oblast celého řetězce, ale na druhou stranu nemůže soupeřit s komerčními řešeními od společností IAR nebo KEIL. Pokud vývojář oželí nedostupnost okamžité technické podpory, ne úplně kvalitní dokumentace, získá s OSS alternativu, která není řádově horší (vygenerovaný kód, rychlost zpracování) oproti komerčním produktům. Některé společnosti se snaží zmírnit neduhy OSS a nabízejí komerční řešení, které je založeno na open-source nástrojích a kromě podpory přidávají různé modifikace zjednodušující vývoj (Atollic, NXP Code Red, ARM DS-5 CE). Tento model je prospěšný pro oba světy, OSS získá vylepšení díky specialistům ve firmách pracujících na modifikacích, komerční společnost získá velikou komunitu testovacích uživatelů, ačkoliv nevyužívají přímo její produkt.

Při vývoji vestavěného zařízení se používají tyto nástroje:

- Kompilátor a linker – Programy umožňující převod ze zdrojového kódu do binární podoby pro konkrétní MCU.
- Vývojové prostředí – Grafický uživatelský program sloužící k editaci zdrojových kódu, správu projektů a dalších podpůrných činností zjednodušující vývoj programového vybavení.
- JTAG/SWD debugger – Zařízení umožňující ladění a nahrání programu do mikrokontroléru.

5.2.1 Kompilátor (překladač)

V době psaní této práce, je GCC (GNU Compiler Collection) synonymem pro spojení Open-source překladač. V případě GCC se nejedná pouze o překladač, obsahem je kompletní sada nástrojů pro vývoj software, od kompilace zdrojových souborů, generování objektových souborů, jejich linkování a export do binární podoby. Obsahuje též podpůrnou sadu nástrojů pro manipulaci a diagnostiku vygenerovaného spustitelného souboru ELF (Executable Image File). Na obrázku č. 6 (zdroj http://www.bogotobogo.com) je znázorněn průběh operací při generování binárního obrazu ze zdrojových souborů.



Obrázek 6: Průběh operací při vytváření binárního obrazu ze zdrojových kódů

GCC není specializován pouze na ARM, ale umožňuje vytvářet ze stejného zdrojového kódu programy pro mnoho různých platforem, nejen pro vestavěné systémy. Této vlastnosti je využito v kapitole č. 9.1 při ověřování bezchybné funkce programového vybavení. Generování binárního kódu pro Cortex M/R procesory (arm-none-eabig-gcc) udržuje a podporuje přímo společnost ARM.

5.2.2 Integrované vývojové prostředí

OSS nabízí vícero programů zaměřených na vývoj vestavěných systémů, nejpopulárnější jsou NetBeans a Eclipse. Oba programy jsou vyvinuté v Javě, takže je lze provozovat na systémech s operačním systémem MS Windows, GNU/Linux a MacOS. Vzrůstající

popularitu potvrzuje přechod některých společností od vlastních prostředích na některé z výše jmenovaných. Například společnost Microchip postavila nové IDE MPLAB X na IDE Netbeans. V této práci bylo použito vývojové prostředí Eclipse. Kombinaci OSS IDE a překladače využívají některé společnosti (Atollic, NXP Code Red, ARM DS-5 CE, aj.), které nabízejí komerční vývojové prostředí složené z překladače GCC a modifikovaného Eclipse.

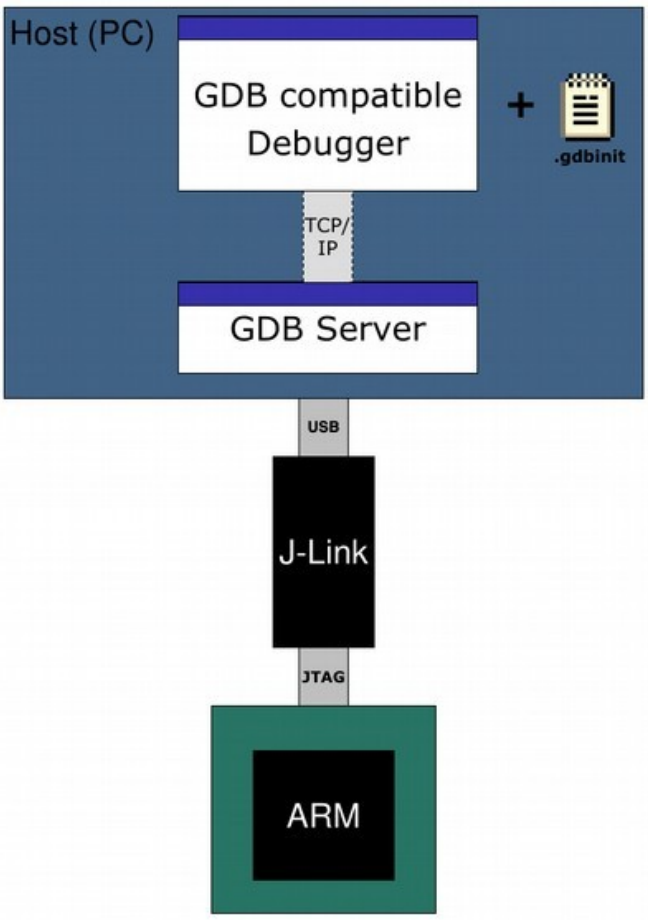
Díky svému návrhu a modularitě je možno Eclipse rozšířit pomocí pluginů pro vytváření kódu vestavěných systémů. Organizace Eclipse přímo nabízí ucelený balík s podporou vývoje v jazyce C/C++, které zahrnuje podporu pro GCC toolchain, ale pouze ve verzi pro PC. Samotné GCC je složené z programů spouštěných z příkazové řádky a Eclipse umožňuje zkompilovat rozsáhlé projekty pouze stisknutím jediného tlačítka.

Eclipse není pouze editor, ale zahrnuje množství vestavěných funkcionalit, např.:

- Správa projektů
- Integrovaná podpora pro verzovací systémy (SVN, Git)
- Syntaktický analyzátor
- Našeptávač kódu

5.2.3 Ladící prostředky - JTAG/SWD debugger

Pro nahrání/ladění programu do/na mikrokontroléru byl použit debugger Jlink od společnosti Segger (https://www.segger.com/). Ačkoliv se jedná o komerční produkt, obsahuje podporu pro OSS vývojové nástroje pomocí programu JlinkGDBServer, který zajišťuje překlad mezi OSS (GDB protokolem) a proprietárním řešením JLink znázorněného na obrázku č. 7 (zdroj: https://www.segger.com/). Další zajímavá vlastnost je funkce „Unlimited Flash Breakpoints“, umístění téměř neomezeného počtu breakpointů v ladícím režimu, tedy i v případě, kdy jsou hardwarové breakpointy vyčerpány nebo je nelze použít z důvodu vykonávání kódu v externí paměti.



Obrázek 7: Propojení komerční ladící sondy s OSS

5.3 Výběr knihoven a operačního systému

V kapitole 4.3 Výběr open-source programů je uváděna vlastnost některých současných real-time operačních systémů, a to přílišná provázanost jejich jádra na strukturu ovladačů a systému. Tato vlastnost byla vyhodnocena jako negativní při návrhu programového vybavení z důvodu možného rizika chyb při implementaci ovladačů, kdy je nutné zkoumat chování jak samotného ovladače, tak i samotného jádra. Preferovaným řešením je RTOS, který integruje pouze přepínání úloh a obsahuje základní primitiva pro práci se sdílenými prostředky a nechává režii ovladačů plně v kompetenci systémového programátora. Takovému chování plně vyhovuje FreeRTOS (http://www.freertos.org/), real-time exekutiva obsahující prioritní preemptivní plánovač a základní podporu pro mutexy, semafory a sdílené fronty.

5.3.1 OSS knihovny

Na základě požadavků na funkcionalitu programového vybavení bylo stanoveno:

- TCP/IP komunikace a HTTP server zajištěno externí knihovnou

- Grafická knihovna zajištěna externí knihovnou
- Bezpečné nahrání aktualizovaného firmware řešeno pomocí zavaděče
- Pro výměnu mezi terminálem a osobním počítačem byl zvolen souborový formát dat FAT, použita externí knihovna
- Pro snadnou manipulaci s daty byl zvoleno použití relační databáze s podporou jazyka SQL, použita externí knihovna
- Podpora letního času na úrovni standardní knihovny jazyka C
- Ladící rozhraní obdobné jako u knihovny Syslog

V ostatních případech byl na každý požadavek vytvořen odpovídající ovladač, případně knihovna. Tabulka č. 1 zobrazuje seznam použitých OSS knihoven.

Funkce	**Název knihovny**	**Odkaz na zdroj**
Grafická knihovna	GRlib	http://software-dl.ti.com/tiva-c/SW-TM4C/latest/index_FDS.html
Podpora PNG obrázků	Libpng	http://www.libpng.org/
TCP/IP stack	LWIP	http://savannah.nongnu.org/projects/lwip/
HTTP Server	Libmicrohttpd	https://www.gnu.org/software/libmicrohttpd/
Souborový systém	FatFS	http://elm-chan.org/fsw/ff/00index_e.html
Databáze	SQLite	https://www.sqlite.org/

Tabulka 1: Seznam OSS knihoven

6 Příprava open-source vývojových nástrojů

Samotné vývojové nástroje nabízejí pouze obecné řešení problému a před použitím pro konkrétní platformu je nutné je parametricky nastavit, případně upravit. Ve fázi zpracování zdrojových souborů a sestavení programu musí překladač znát parametry cílové architektury a linker paměťovou mapu zařízení. Standardní knihovna C, která je dodávána s GCC překladačem, nepodporuje víceúlohové prostředí a musí být znovu sestavena s novými parametry. Grafické IDE v základní verzi nepodporuje použitý překladač a je nutné nainstalovat rozšiřující modul.

6.1 Parametry překladače

Parametry překladače slouží primárně k ovlivnění výsledného kódu z hlediska architektury, velikosti a rychlosti. Seznam všech přepínačů lze nalézt v uživatelské příručce GCC. Grafické IDE je má ve formě zaškrtávacího nebo výběrového menu i s textovým popiskem, což usnadňuje jejich použití. V tabulce č. 2 je uveden příklad některých přepínačů pro mikrokontrolér LCP1788.

-mcpu=cortex-m3 -mthumb	Výsledný kód bude generován pro architekturu ARMv7-M a kód bude vykonáván ve Thumb stavu
-ffunction-sections -fdata-sections	Každá funkce a proměnná (vyjma lokálních) bude uložená ve vlastní sekci. Lze využít při eliminaci typu „dead code“, parametr pro linker „`--gc-sections`“
-fsingle-precision-constant	Konstanty s pohyblivou řádovou čárkou jsou považovány za typ float(32bit) namísto implicitního double(64bit).
-0[0|1|2|3|s]	Úroveň optimalizace kódu, 0 – žádná optimalizace, s – optimalizace na velikost kódu
-Wstack-usage=512	Překladač generuje upozornění, pokud funkce vyžaduje více než 512 B zásobníkové paměti

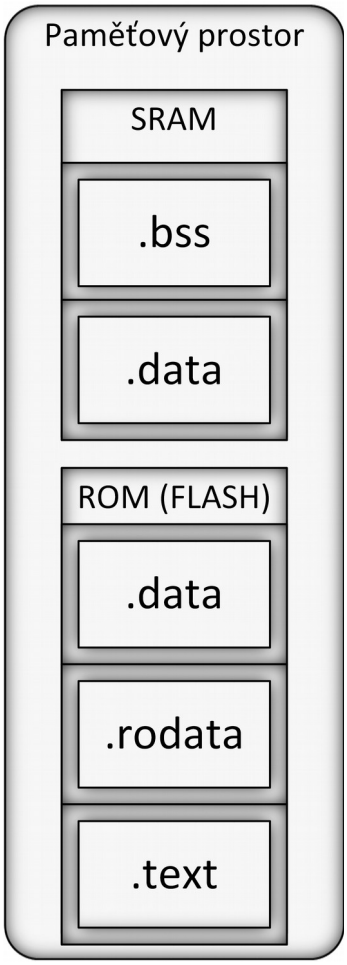
Tabulka 2: Přepínače překladače pro jádro Cortex-M3

Při kompilaci kódu se používají dvě nezávislé sady přepínačů. V Debug konfiguraci je zapnuto generovaní ladících symbolů a zachována struktura odpovídající zdrojovému kódu. Tato konfigurace je vhodná pro ladění aplikace. V Release konfiguraci jsou zapnuty optimalizace, které mohou zcela pozměnit strukturu programu, např. vynecháním určité části kódu, který nemá ze sémantického hlediska vliv na chod programu. Typickým příkladem je čekání ve smyčce na změnu příznaku u proměnné, která je modifikována v rutině přerušení a nebyla deklarována s atributem „volatile“. Překladač provede pouze jedno načtení z paměti

a uloží hodnotu do registru, který již není poté modifikován a kontrola hodnoty se provádí vůči hodnotě v registru. Také je nutné zadefinovat makro NDEBUG, které odstraní z programu „assert“ klauzule.

6.2 Paměťová mapa zařízení

U mikrokontrolérů s architekturou ARMv7-M vykonávající kód z ROM, je paměťové rozdělení programu obdobné jako na obrázku č. 8, oblasti zásobníkové paměti(stack) a dynamické haldy (heap) zde nejsou zobrazeny.



Obrázek 8: Paměťové rozdělení programu

Sekce, nebo také též v tomto kontextu segment, je úsek paměti, kam ukládá linker symbol vygenerovaný překladačem v závislosti na jeho kontextu. Symbol může tvořit funkce, proměnná, případně konstanta. V linker scriptu lze definovat také symboly, jenž nealokují paměťový prostor a jsou přístupné z programového kódu. Každá sekce má dvě adresy VMA (Virtual Memory Address), LMA (Load Memory Address) a tabulku symbolů, které do ní náleží. Pokud není využíváno MMU (memory management unit), VMA a LMA se rovnají

a odpovídají fyzickým adresám, vyjma speciálního .data segmentu, u kterého LMA odpovídá adrese, odkud se nahrávají inicializační data do proměnných umístěných v RAM. Inicializační data se nacházejí ve Flash paměti. Výsledkem postupu inicializace .data segmentu je dvojí výskyt jedné inicializované proměnné v paměti mikrokontroléru. Tato vlastnost někdy bývá zdrojem plýtvání SRAM paměti, pokud se deklaruje proměnná s inicializovanou hodnotou, ale posléze se k ní přistupují jako ke konstantě. Kód na výpisu č. 1 vyhradí v SRAM úsek paměti o délce ~32 bajtů, do kterého se při inicializaci systému překopíruje z FLASH paměti ~16 bajtů.

```
char text[32] = „Plytvani pameti“;
```

Výpis 1: Alokace proměnné v programové a datové paměti

Celková spotřeba paměti činí ~48 bajtů, namísto pouhých ~16ti. Kritické na tomto chybném přístupu je zbytečné alokování SRAM, které bývá řádově méně, než ROM.

V tabulce č. 3 jsou uvedeny ustálené názvy sekcí a jejich zaměření.

Název segmentu (sekce)	**Význam**
.text	Oblast obsahující kód programu, umístěná obvykle v ROM (FLASH)
.data (RAM)	Oblast obsahující inicializované proměnné, umístěná v SRAM
.bss	Oblast obsahující neinicializovaná proměnné, umístěná v SRAM. Norma jazyka C definuje, že tato oblast musí obsahovat hodnotu nula (binární).
.rodata	Oblast obsahující konstanty (read only data), umístěná obvykle v ROM (FLASH)
.data (ROM)	Oblast obsahující inicializační hodnoty pro inicializované proměnné, umístěná obvykle v ROM (FLASH)
.xyz	V linker scriptu lze definovat libovolný název sekce a její umístění v paměti. Této vlastnosti je využito při mapování rozsáhlejších knihoven do externí paměti.

Tabulka 3: Název a popis sekcí generovaných překladačem

```
> arm-none-eabi-objdump -h UPM_02_TST.elf

UPM_02_TST.elf: file format elf32-littlearm

Sections:
Idx Name Size VMA LMA File off Algn
 0 .isr_vector 00000610 00004000 00004000 00004000 2**2
 CONTENTS, ALLOC, LOAD, READONLY, CODE
 1 .inits 00000000 00004610 00004610 00030180 2**2
 CONTENTS
 2 .text 00021fc8 00004610 00004610 00004610 2**3
 CONTENTS, ALLOC, LOAD, READONLY, CODE
 3 .ARM.exidx 00000008 000265d8 000265d8 000265d8 2**2
 CONTENTS, ALLOC, LOAD, READONLY, DATA
 4 .rodata 000047c8 000265e0 000265e0 000265e0 2**3
 CONTENTS, ALLOC, LOAD, READONLY, DATA
 5 .data 00000180 10000000 0002adb0 00030000 2**4
 CONTENTS, ALLOC, LOAD, DATA
 6 .bss 000014b0 10000180 0002af30 00030180 2**2
 ALLOC
 7 .noinit 0000d650 10001630 0002af30 00031630 2**2
 ALLOC
 8 .lpc1788.peripheral_ram 00002000 20000000 20000000 00038000 2**3
 ALLOC
 9 .comment 00000070 00000000 00000000 00030180 2**0
 CONTENTS, READONLY
 10 .debug_aranges 00003988 00000000 00000000 000301f0 2**3
 CONTENTS, READONLY, DEBUGGING
 11 .debug_info 0005ddec 00000000 00000000 00033b78 2**0
 CONTENTS, READONLY, DEBUGGING
 12 .debug_abbrev 0000fd37 00000000 00000000 00091964 2**0
 CONTENTS, READONLY, DEBUGGING
 13 .debug_line 0004335a 00000000 00000000 000a169b 2**0
 CONTENTS, READONLY, DEBUGGING
 14 .debug_frame 0000d89c 00000000 00000000 000e49f8 2**2
 CONTENTS, READONLY, DEBUGGING
 15 .debug_str 00050cc8 00000000 00000000 000f2294 2**0
 CONTENTS, READONLY, DEBUGGING
 16 .debug_ranges 00002ed8 00000000 00000000 00142f5c 2**0
 CONTENTS, READONLY, DEBUGGING
 17 .ARM.attributes 00000031 00000000 00000000 00145e34 2**0
 CONTENTS, READONLY
 18 .debug_macro 000290b4 00000000 00000000 00145e65 2**0
 CONTENTS, READONLY, DEBUGGING
```

Výpis 2: Seznam sekcí obsažených v programu

Výpis sekcí a jejich umístění lze v GCC zjistit pomocí programu objdump. V ukázkovém výpisu č. 2 je uvedeno v sekci .data LMA=0002adb0 a VMA=10000000. Počátek oblasti inicializovaných dat v SRAM začíná od adresy 0x10000000 a inicializační hodnoty jsou uloženy v ROM od adresy 0x0002adb0, což odpovídá i paměťové mapě LPC1788 (viz obrázek č. 4). Velikost oblasti je 0x180 bajtů. Provedení inicializace, tj. nakopírování dat z ROM do SRAM, je součástí Board Support Package (BSP), o kterém pojednává kapitola č. 8.3. Nesoulad mezi LMA a VMA má také .bss sekce, ale u ní je důležité si všimnout sekčních příznaků, které obsahují pouze hodnotu ALLOC, tedy v paměti bude vyhrazeno místo, ale nebudou se do něj nahrávat data.

Rozdělení paměťového prostoru a jeho přidělení jednotlivým sekcím je stanoveno v textovém souboru zvaném linker script [15], který je základním vstupem pro program Linker. Script obsahuje přesné informace o velikosti a typu pamětí, umístění segmentů a jejich parametrů, popisem výstupního formátu, definování symbolů aj. Příklad linker scriptu je uveden v příloze A.

6.2.1 Definování velikosti a typu paměti v linker scriptu

Konfigurační slovo MEMORY určuje rozdělení pamětí v konkrétním mikrokontroléru. Hodnoty pro údaje ORIGIN musí být v souladu s fyzickým rozdělením paměti u mikrokontroléru a linker využívá hodnoty pro generování programových skoků, umístění proměnných v RAM a kontroly, zda se program vejde do pamětí (ROM i RAM).

```
MEMORY
{
 FLASH (rx) : ORIGIN = 0x4000, LENGTH = 496K /* The lpc1788 has 512KB but
the first 16KB is used for bootloader*/
 RAM (rwx) : ORIGIN = 0x10000000, LENGTH = 63K /* The lpc1788 has 64KB but
the last 1KB is used for interrupts*/
 peripheral_ram (rw) : ORIGIN = 0x20000000, LENGTH = 32K
 EXT_FLASH (rx) : ORIGIN = 0x80000000, LENGTH = 16M
}
```

Výpis 3: Definování paměti v linker skriptu

Na výpisu č. 3 je uvedeno rozdělení dle paměťové mapy LPC1788. Ačkoliv interní FLASH začíná od adresy 0x00000000, pro systém je vyčleněna oblast od adresy 0x00004000. První 16KB je vyhrazeno pro program zavaděče (Bootloaderu). Externí ROM paměť je mapovaná na adresu 0x80000000 a periférie EMC musí být konfigurována v souladu s touto hodnotu.

Velikost dostupné (S)RAM paměti je zmenšena o 1 KB oproti skutečné velikosti. Pokud by se stanovila plná velikost, mohlo by dojít k náhodnému přepsání dat v .data nebo .bss segmentu. Při běhu aplikace s využitím RTOS se využívá dvojího ukazatele na zásobník [10], PSP – zásobník jednotlivých úloh, MSP – zásobník pro rutiny obsluhy přerušení. V architektuře ARMv7-M roste zásobník směrem dolů a MSP je nastaven na nejvyšší možnou adresu RAM. Nastavení PSP je nastavováno v rámci RTOS. Pokud by programátor alokoval proměnné umístěné v RAM o celkové velikosti např. 65.000 B, překladač by negeneroval chybu při linkování a při běhu programu dojde k přepsání dat. Snížením hodnoty dostupné paměti se de-facto alokuje 1KB zásobníkové paměti pro rutiny obsluhy přerušení, spolu s kontrolou přetečení uživatelských dat. Pro rutiny platí omezení o maximálním využití zásobníku na 1 KB.

6.2.2 Definování sekcí v linker scriptu

Konfigurační slovo SECTIONS určuje rozmístění kde jsou vstupní sekce umístěny ve výstupních sekcích, jejich pořadí ve výsledném obrazu a do které paměti jsou alokovávány. Vstupní sekce je označení symbolu generovaného překladačem, automaticky podle umístění

proměnné nebo pomocí speciální atributu „section“. Na výpisu č. 4 je příklad výstupní sekce „.text“ umístěná v interní programové paměti, která v sobě zahrnuje symboly ve vstupní sekci „.text“, kód programu. Z této výstupní sekce jsou vyloučeny pomocí direktivy „EXCLUDE_FILE„ všechny vstupní sekce v knihovně libUPM_02_Library.a. Ty jsou umístěny do výstupní sekce „.ext_flash.text“ nacházející se v externí programové paměti. Toto rozdělení umožňuje rozdělovat kód aplikace do různých programových prostorů a vytvářet mezi nimi vazby ve smyslu skoků a návratů z funkcí, reference proměnných, bez nutnosti používat vyhledávací tabulky jako v případě použití dvou různých nezávislých obrazů.

```
   .text :
      {
            CREATE_OBJECT_SYMBOLS
            _ftext = .;
            *(EXCLUDE_FILE   (*libUPM_02_Library.a:)   .text   EXCLUDE_FILE
(*libUPM_02_Library.a:)   .text.*   EXCLUDE_FILE   (*libUPM_02_Library.a:)
.gnu.linkonce.t.*)
            *(.plt)
            *(.gnu.warning)
            *(.glue_7t) *(.glue_7) *(.vfp11_veneer)
            *(.ARM.extab* .gnu.linkonce.armextab.*)
            *(.gcc_except_table)
      } > FLASH
      .ext_flash.text : ALIGN (4)
      {
       __upm02_ext_flash_text_start__ = .;
       *libUPM_02_Library.a:(.text .text.* .gnu.linkonce.t.*)
       *(.ext_flash .ext_flash.*)
       . = ALIGN (4);
       __upm02_ext_flash_text_start_end__ = .;
      } >EXT_FLASH
```

Výpis 4: Definování vstupních a výstupních sekcí v linker skriptu

6.3 Rekompilace standardní knihovny jazyka C

Standardní knihovna jazyka C (Newlib) dodávaná spolu s binární distribucí GNU ARM Toolchain plně nepokrývá požadavky na vývoj vícevláknových aplikací s využitím RTOS a s omezeným paměťovým prostorem a je nutná její rekompilace ze zdrojových kódů. Rekompilace knihovny se provádí z bezpečnostních důvodů z balíku, ze které pochází GCC

překladač, neboť dle dokumentace k vývojovému prostředí prochází daná verze knihovny s danou verzí překladače testováním. Před rekompilací knihovny je potřeba provést její úpravy spočívající v modifikací existujících souborů a přidání nových, kde je implementována podpora pro RTOS nebo opraveny dosud nevyřešené chyby.

6.3.1 Uvolňování prostředků po ukončení úlohy

Newlib při uvolňování prostředků alokovaných pro úlohu (task, thread) v konfiguraci s volbou --enable-newlib-reent-small nezavírá souborové deskriptory alokované pro stdin/stdout/stderr ve funkci _cleanup_r() volané z _reclaim_reent(). Funkce _cleanup_r() musí být modifikována, aby došlo k explicitnímu uvolnění prostředků po skončení úlohy.

```
--------------------file newlib/newlib/libc/stdio/findfp.c-----------------
#if defined(_REENT_SMALL) && defined(_STDIO_CLOSE_PER_REENT_STD_STREAMS)
_CAST_VOID cleanup_func(ptr, ptr->_stdin);
_CAST_VOID cleanup_func(ptr, ptr->_stdout);
_CAST_VOID cleanup_func(ptr, ptr->_stderr);
#endif
_CAST_VOID _fwalk_reent (ptr, cleanup_func);
```

Výpis 5: Úprava funkce _cleanup_r() pro podporu uvolňování alokovaných prostředků

6.3.2 Podpora zamykání sdílených proměnných

Zamykání sdílených prostředků pro architekturu ARM typ bare-metal je řešeno jako NO-OP, tedy nevykonává se žádná operace a zamykací funkce jsou z knihovny vypuštěny. Aby se zamykání aktivovalo, do adresáře newlib/newlib/libc/sys/arm/sys/ musí být nahrán soubor s odkazy na zamykací funkce, které jsou následně implementovány programovým vybavením.

```
    ------------------file newlib/newlib/libc/sys/arm/sys/lock.h-----------------
    #ifndef __SYS_LOCK_H__
    #define __SYS_LOCK_H__

    typedef int _LOCK_T;
    typedef int _LOCK_RECURSIVE_T;

    #include <_ansi.h>

    #define __LOCK_INIT(class,lock) static int lock = 0;
    #define __LOCK_INIT_RECURSIVE(class,lock) static int lock = 0;

    void newlib_lock_init(_LOCK_T * lock);
    void newlib_lock_init_recursive(_LOCK_RECURSIVE_T * lock);
    void newlib_lock_close(_LOCK_T * lock);
    void newlib_lock_close_recursive(_LOCK_RECURSIVE_T * lock);
    void newlib_lock_acquire(_LOCK_T * lock);
    void newlib_lock_acquire_recursive(_LOCK_RECURSIVE_T * lock);
    int newlib_lock_try_acquire(_LOCK_T * lock);
    int newlib_lock_try_acquire_recursive(_LOCK_RECURSIVE_T * lock);
    void newlib_lock_release(_LOCK_T * lock);
    void newlib_lock_release_recursive(_LOCK_RECURSIVE_T * lock);

    #define __lock_init(lock) newlib_lock_init(&(lock))
    #define __lock_init_recursive(lock) newlib_lock_init_recursive(&(lock))
    #define __lock_close(lock) newlib_lock_close(&(lock))
    #define __lock_close_recursive(lock) newlib_lock_close_recursive(&(lock))
    #define __lock_acquire(lock) newlib_lock_acquire(&(lock))
    #define __lock_acquire_recursive(lock) newlib_lock_acquire_recursive(&(lock))
    #define __lock_try_acquire(lock) newlib_lock_try_acquire(&(lock))
    #define __lock_try_acquire_recursive(lock)
newlib_lock_try_acquire_recursive(&(lock))
    #define __lock_release(lock) newlib_lock_release(&(lock))
    #define __lock_release_recursive(lock) newlib_lock_release_recursive(&(lock))

    #endif /* __SYS_LOCK_H__ */
```

Výpis 6: Přidání zamykací funkcionality pro podporu víceúlohového prostředí

6.3.3 Editace systémových parametrů ARM

V současné verzi knihovny 2.20 nejsou v souboru libc/sys/arm/sys/param.h definována makra MIN a MAX, která posléze vyvolají výjimku nedefinované funkce při použití Newlib funkce __big_insert. Tato makra musí být do souboru přidána.

```
   ------------------file newlib/newlib/libc/sys/arm/sys/param.h------------------
/* ARM configuration file; HZ is 100 rather than the default 60 */
#ifndef _SYS_PARAM_H
# define _SYS_PARAM_H

# define HZ (100)
# define NOFILE (60)
# define PATHSIZE (128)

#define BIG_ENDIAN 4321
#define LITTLE_ENDIAN 1234

#ifdef __ARMEB__
#define BYTE_ORDER BIG_ENDIAN
#else
#define BYTE_ORDER LITTLE_ENDIAN
#endif

#define MAXPATHLEN PATH_MAX

#define MAX(a,b) ((a) > (b) ? (a) : (b))
#define MIN(a,b) ((a) < (b) ? (a) : (b))
#endif
```

Výpis 7: Přidání chybějících maker pro architekturu ARM

6.3.4 Chybějící prototyp __cxa_exit

Pro správnou implementaci výjimek v C++, v adresáři sys/arm se nachází soubor __aeabi_atexit.c, který obsahuje volání funkce __aeabi_atexit, která volá funkci __cxa_atexit. V původní verzi Newlib chybí prototyp funkce __cxa_atexit a překladač generuje chybové hlášení.

```
   -------------file newlib/newlib/libc/sys/arm/sys/__aeabi_atexit.c--------------
#include <stdlib.h>

int __cxa_atexit (void (*fn) (void *), void *arg , void *d);
/* Register a function to be called by exit or when a shared library
 is unloaded. This routine is like __cxa_atexit, but uses the
 calling sequence required by the ARM EABI. */
int
__aeabi_atexit (void *arg, void (*func) (void *), void *d)
{
 return __cxa_atexit (func, arg, d);
}
```

Výpis 8: Doplnění prototypu funkce __cxa_atexit()

6.3.5 Konfigurace systému proměnných prostředí

Některé funkce během svého vykonávaní alokují na zásobníku proměnné o velikosti danou konfiguračními parametry. Defaultní parametry jsou převážně z prostředí pro paměťově bohatější mikrokontroléry. Snížením z původních hodnot se zamezí přílišného obsazení zásobníku, na druhou stranu je potřeba více času na vykonání funkce.

Jedná se zejména o tyto parametry:

__FILENAME_MAX__	Maximální podporovaná délka souboru
__FOPEN_MAX__	Maximální počet souborů otevřených najednou
__BUFSIZ__	Velikost bufferu při práci se soubory

Konfigurační parametry se uvádí v souboru newlib/newlib/libc/sys/arm/sys/config.h

6.3.6 Příprava kompilace

Kompilace knihovny se provádí s využitím sestavovacího prostředí GNU Make. Před vlastní kompilací je nutné knihovnu nakonfigurovat pomocí speciálního skriptu configure, kterému jsou předány parametry ovlivňující výsledný obraz knihovny, viz výpis č. 9.

```
   ./configure --target=arm-none-eabi --disable-newlib-supplied-syscalls --enable-
newlib-io-long-long --enable-newlib-reent-small --disable-libgloss --enable-
newlib-io-c99-formats --disable-newlib-atexit-dynamic-alloc --enable-newlib-
global-atexit --enable-newlib-reent-small --disable-newlib-wide-orient --disable-
multilib --enable-newlib-multithread --disable-newlib-register-fini --disable-
newlib-mb --disable-newlib-unbuf-stream-opt --disable-newlib-fvwrite-in-streamio
--disable-newlib-io-long-double
```

Výpis 9: Konfigurační parametry knihovny

Význam jednotlivých parametrů lez nalézt v dokumentaci knihovny a převážná většina parametrů slouží k paměťové redukci knihovny. Vlastní kompilace probíhá pomocí příkazu make se specifickými parametry pro mikrokontrolér LPC1788.

```
   make CFLAGS_FOR_TARGET="-DMALLOC_PROVIDED -DHAVE_ASSERT_FUNC
-DREENTRANT_SYSCALLS_PROVIDED -DSIGNAL_PROVIDED -fstack-usage -ffunction-sections
-fdata-sections -Wstack-usage=512 -mcpu=cortex-m3 -mthumb -std=gnu11 -fno-common
-g3"
```

Výpis 10: Sestavovací parametry knihovny

Dodefinovaná makra snižují paměťovou náročnost knihovny a jejich použití je z důvodu neexistence parametru pro skript configure.

- SIGNAL_PROVIDED

  Programové vybavení nepoužívá signály a jejich funkcionalita není implementována.

- MALLOC_PROVIDED

  Programové vybavení používá vlastní správu dynamické paměti.

- REENTRANT_SYSCALLS_PROVIDED

  Programové vybavení používá vlastní sadu reentrantních systémových funkcí.

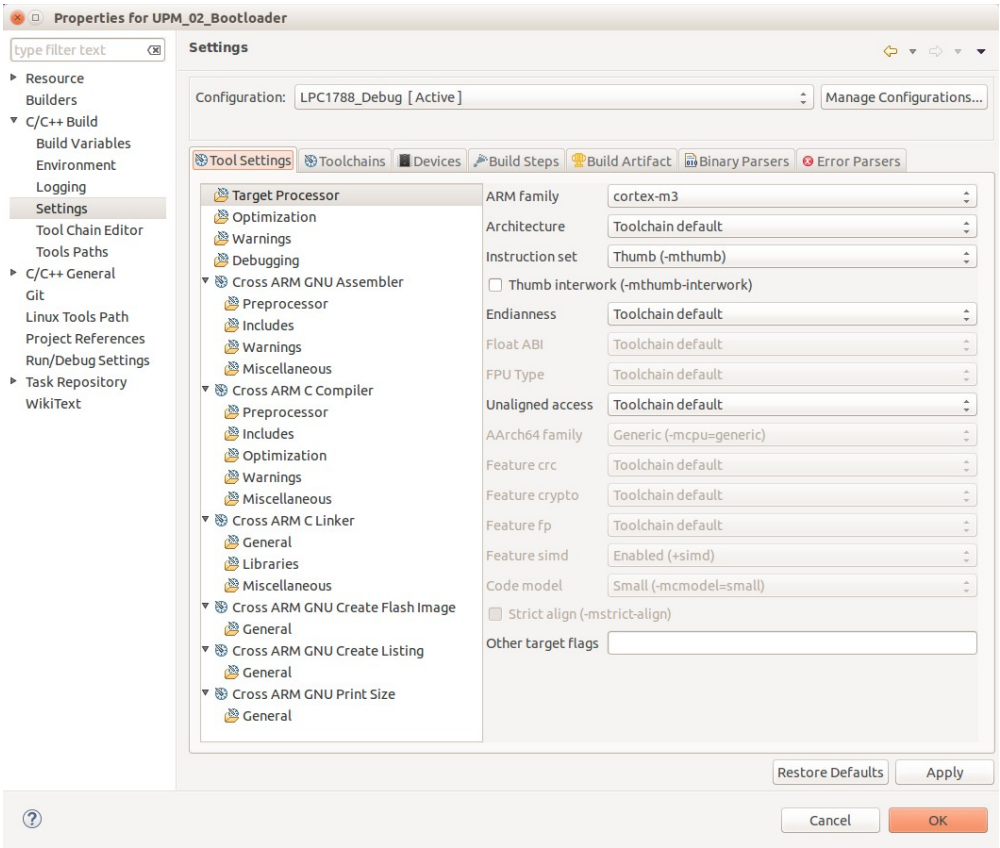
- HAVE_ASSERT_FUNC

  Programové vybavení definuje vlastní formát výpisu aserce.

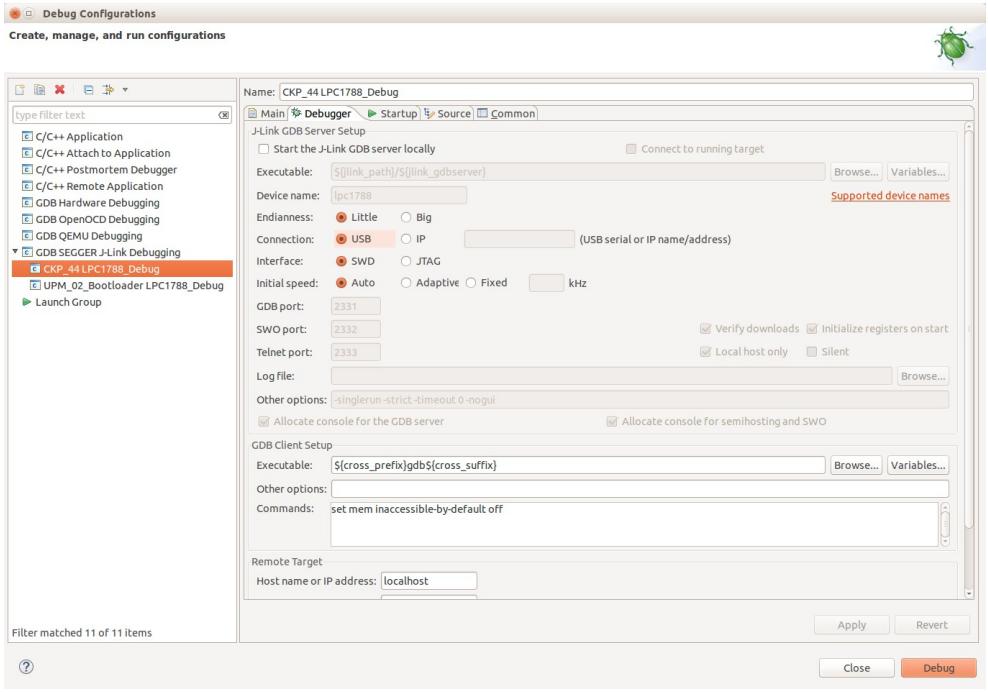
Výsledné vygenerované knihovny libc.a, libm.a a hlavičkové soubory se přehrají do adresářové struktury překladače a nahradí stávající. Nové knihovny obsahují ladící symboly a jsou vhodné pro Debug konfiguraci. Při Release konfiguraci je nutné zapnout optimalizaci pomocí přepínače „-O“.

6.4 Rozšiřující moduly pro IDE

Vývojové prostředí Eclipse v instalačním balíku neobsahuje podporu vývoje programů běžících na procesorech Cortex-M3 s překladačem GCC pro ARM. Pro zahrnutí podpory je nutné nainstalovat plugin GNU ARM Eclipse, který modifikuje prostředí Eclipse, a tak usnadňuje správu projektů určených pro vestavěné systémy. Důležité parametry nastavení překladače a linkeru jsou zaintegrovány do rolovacích menu spolu s popiskem, viz obrázek č. 9. Plugin podporuje také integraci ladící sondy J-Link, obrázek č. 10.



Obrázek 9: Podpora vestavěných systémů v GNU ARM Eclipse pluginu



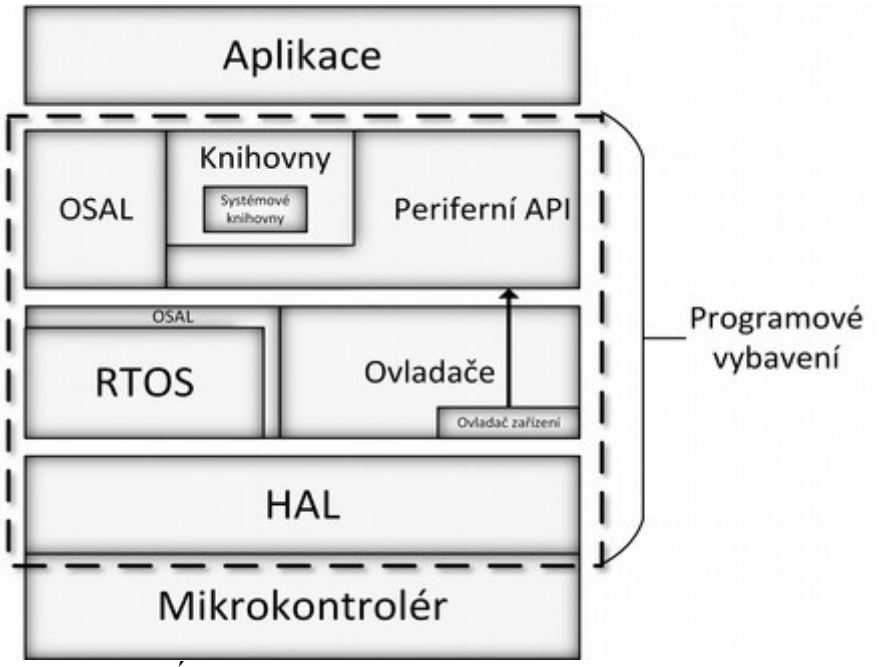
Obrázek 10: Podpora ladění s JLink v GNU ARM Eclipse pluginu

7 Struktura programového vybavení

Vnitřní struktura programového vybavení je tvořena úrovňovým modelem se vzestupnou vertikální abstrakcí. Nejnižší úroveň tvoří mikrokontrolér a jeho periférie, nejvyšší rozhraní je poskytnuté aplikačnímu programátorovi. Jednotlivé úrovně spolu komunikují na vertikální a horizontální úrovni pomocí volání příslušných funkcí. Rozhraní mezi úrovněmi je nazýváno vrstvou. Vertikálně lze využívat služeb pouze nejbližší nižší úrovně. Základní koncept je zobrazen na schématu na obrázku č. 11. Aplikace je oddělená od služeb operačního systému resp. procesoru a jeho periferií abstrakční vrstvou, která je rozdělena na tři části, abstrakční vrstvu operačního systému (OSAL), knihovny a Periferní API. Samotné hardwarové ovládání periferií je pak ještě od ovladačů odděleno HAL vrstvou. Toto uspořádání umožňuje v případě potřeby přejít na jiný operační systém nebo hardware bez nutnosti změn v aplikacích.

Knihovny tvoří balík software vykonávající různé úlohy. S programovým vybavením komunikují pomocí adaptačních vrstev, rozhraní mezi knihovnou a systémem. Systémové knihovny slouží potřebám programového vybavení a nejsou přístupné aplikačnímu programátorovi.

Při vertikální komunikaci mezi úrovněmi je využíváno objektů bez datové struktury, tzv. handles, které odkazují na zapouzdřené detaily objektů nižších úrovní.



Obrázek 11: Úrovně programového vybavení

7.1 Handles

Handle (z důvodu přehlednosti ponechán anglický název) označuje v programovém vybavení objekt bez známé vnitřní struktury, který reprezentuje složitější objekt z vrstev HAL, RTOS nebo Ovladače. Z hlediska aplikačního programu není tato vnitřní struktura není známa a handle je nestrukturovaný objekt, který je pouze předáván mezi volanými funkcemi API. Důvodem použití handle je zapouzdření, tj. oddělení vnějšího chování a rozhraní nějakého objektu od jeho vnitřní struktury. Pokud se změní vnitřní reprezentace a implementace, není nutné provádět změny v uživatelském API. Navíc je programátor odstíněn od přílišných detailů vnitřní struktury objektů, a tím spojenou nutnost zahrnout cesty k jejich hlavičkovým souborům. Handle je v systému globální objekt a není svázán s konkrétní úlohou, která ho vytvořila.

7.1.1 Vnitřní reprezentace handle

Handle tvoří znaménkový celočíselný datový typ (int), jehož číselné vyjádření představuje ukazatel do tabulky seznamu handlů a s ním asociované uložené proměnné. Pozitivní čísla včetně nuly vyjadřují validní handle a negativní jsou rezervovány pro chybové stavy errno. Struktura popisující záznam handle, výpis č. 11, obsahuje proměnnou uchovávající druh a příznak vztahující se k danému handle a pole dat, které lze asociovat s konkrétním handle. Jednotlivé vrstvy mohou provádět typovou kontrolu na základě druhu handle, například vrstva OSAL při práci s mutexem si nejdříve ověří, zda předávaný handle je skutečně druhu MUTEX, jinak je vrácen volajícímu chybový stav.

```
typedef struct
{
      uint32_t iTypeAndFlags;
      HANDLE_DATA handle_data[HANDLE_MAX_VARIABLES];
} HANDLE_DESCRIPTOR;
```

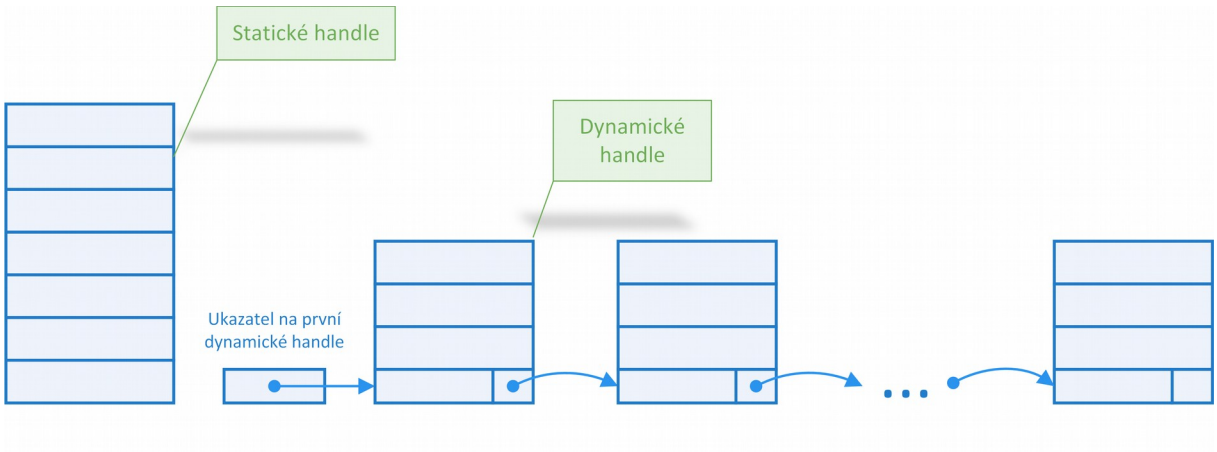
Výpis 11: Struktura záznamu handle

Proměnné ukládané do pole dat mohou být různého typu, proto je datová struktura tvořena typem union, výpis č. 12, který zajistí alokaci podle velikosti největšího typu. Primitivní datové typy jsou ukládány hodnotou, složité struktury pomocí ukazatele.

```
typedef union {
      uint32_t number;
      int   integer;
      unsigned int unsigned_integer;
      void * pointer;
      const void * cpointer;
}HANDLE_DATA;
```

Výpis 12: Podpora typů dat asociovaných s handle

Tabulka pro statické handle je vytvořena v segmentu paměti .bss a její velikost je určena během kompilace. Jelikož nelze předem odhadnout, jaké množství handle bude aplikace vyžadovat, je v případě vyčerpání statických handle alokována sada handle v dynamické paměti, obsahující ukazatel na další dynamickou sadu, pokud je aktuální sada vyčerpána, viz obrázek č. 12. Při uvolnění všech dynamických handle není dynamická paměť dealokována a je a předpokládáno, že dojde k jejich novému použití. Používáním handle pro přístup k objektům nižších vrstev vzniká režie vyhledávání v tabulkách, a proto jsou tabulky pro statickou a dynamickou handle umísťovány v paměti s rychlým přístupem, interní SRAM.



Obrázek 12: Statické a dynamické tabulky handles

7.1.2 Operace s handle

Každý handle je nutné před použitím nejdříve alokovat pomocí volání funkce handle_alloc(). Tato funkce vrátí index do tabulky, na které pozici se nacházel první volný handle. Při rušení se volá funkce handle_free(), která označí danou pozici jako volnou. Pokud je handle alokován, voláním handle_set() se pro daný handle přiřadí druh a příznaky a také proměnné do pole dat. Získat informace lze voláním handle_get().

Programové vybavení podporuje standardní proudy (stream) pro vstup, výstup a chybový výstup z knihovny jazyka C, tj. stdin, stdout a stderr. Proudy se v mapují na makra STDIN_FILENO(0) STDOUT_FILENO(1) a STDERR_FILENO(2), jejichž číselná vyjádření tvoří číslo virtuálního handle, který nemá přidruženou strukturu. Z tohoto důvodu má první dostupné handle hodnotu rovnu číslu 3.

7.2 Abstraktní vrstva operačního systému - OSAL

Důvodem oddělení aplikace a ovladačů zařízení od operačního systému abstrakční vrstvou je možnost nahradit operační systém RTOS, pokud by při vývoji vestavěného systému došlo k přechodu na jiný. Jednotlivé RTOS systémy se také liší v oblastech nabízených funkcí a rozdíly mezi nimi mohou být do-implementovány v této vrstvě. Při alokaci zdroje z RTOS se využívá služeb handles pro jeho uložení spolu s typem zdroje.

Mezi hlavní činnosti vrstvy patří:

- Obsluha plánovače (zastavení, spuštění)
- Obsluha úloh (vytváření, rušení, zastavování)
- Obsluha RTOS synchronizačních primitiv
- Softwarové časovače
- Spouštění odložených procedur

7.3 Vrstva přístupu k perifériím a zařízením - periferní API

Periferní API poskytuje rozhraní pro přístup k periferiím procesoru jako např. LCD displej, Ethernet, sériový port, a další. U rozhraní je snaha o co nejvyšší abstrakci, aby aplikace nebyla závislá na konkrétním typu periferie. Funkce jsou často typu periferie_open(), periferie_read(), periferie_write(), periferie_close(), jak je obvyklé u POSIX operačních systémů a pro konfiguraci periferie_ioctl().

Identifikace periferie v této vrstvě probíhá na základě textového řetězce, který je k ovladači přiřazen během registrace do systému a operace na perifériemi je prováděna pomocí handle, který je získán po zavolání periferie_open(„Označení periférie“). Ovladače v sobě udržují typ skupiny, do které patří a v rámci periferního API je kontrolována příslušnost k dané skupině. Např. ovladač SPI Flash paměti patří do skupiny MEMORY_DEVICE a pokud je chybně zavolána funkce serial_write(handle) s handle asociovaným pro paměť, je vrácen chybový stav.

7.4 Vrstva ovladačů

Vrstva ovladačů obsahuje implementaci ovládání periferií z abstraktního pohledu a zjemňují komplexní činnosti na menší kroky, které jsou vykonány voláním HAL ovladače. Zároveň sdružují součinnost několika ovladačů, pokud jsou vyžadovány pro obsluhu periferie. Jako příklad lze uvést ovladač SPI sběrnice s využitím DMA přenosu dat. Pro činnost zápisu dat na sběrnici je nutné nejdříve nakonfigurovat DMA přenos a posléze SPI sběrnici. Periferním API je tato implementace skryta, což umožňuje měnit ovladače podle konfigurace systému, aniž by se změnila aplikace.

K periferiím se z aplikací běžně přistupuje z různých vláken, další důležitou funkcí ovladačů je ošetření souběžného přístup. K tomu jsou použita synchronizační primitiva poskytovaná OSAL vrstvou, aby byla zachována nezávislost na operačním systému.

Ovladač zařízení je speciální druh ovladače, který řídí externí periferii a pro svoji činnost využívá integrované periférie na mikrokontroléru. Jako příklad lze uvést ovladač SPI Flash paměti. Samotná paměť vyžaduje pro komunikaci dodržení formátu (protokolu) na sběrnici SPI. Ovladač zařízení SPI paměti využívá ovladač periférie rozhraní SPI z periferního API a implementaci protokolu komunikace s pamětí.

Vyšším vrstvám je ovladač přístupný pomocí interface, který obsahuje odkazy na funkce nabízené ovladačem. Interface je asociován s handle a vyšší vrstvy němu získají přístup voláním funkcí device_interface_*().

7.5 Vrstva HAL - Hardware Abstraction Layer

Samotné ovladače ještě neobsluhují přímo konkrétní hardware, ale volají funkce abstrakční hardwarové vrstvy (HAL). Její rozhraní definuje funkce pro určitý typ hardware jako je například UART, Watchdog nebo sběrnice I2C, či GPIO. Tyto funkce jsou implementovány HAL ovladači, které již provádějí postupy specifické pro konkrétní typ mikrokontroléru, procesoru a periferie. Při portaci programového vybavení na jiný typ mikrokontroléru jsou HAL ovladače místem, které je třeba nejvíce upravit pro konkrétní použití. HAL vrstva negarantuje ochranu proti sdílenému přístupu, ovladače se soustřeďují čistě na obsluhu příslušné periferie.

Vyšším vrstvám je Hal ovladač přístupný pomocí interface, který obsahuje odkazy na funkce nabízené ovladačem. Interface je asociován s handle a vyšší vrstvy němu získají přístup voláním funkcí hal_interface_*().

7.6 Struktura ovladačů

Ovladače periferií a zařízení, stejně jako HAL ovladače mají obdobnou strukturu. Obsluha periferie je zajišťována skupinou funkcí, interface, které jsou společné pro jednotlivé druhy periferií, liší se pouze aktuálním stavem, ve kterém se jednotlivá periférie nachází. Aktuální stav je uchováván v pracovním prostoru ovladače, tzv. workspace. Jde o dynamicky alokovanou strukturu vytvářenou během registrace ovladače.

Na výpisu č. 13 je uveden interface pro HAL ovladač periférie SD/MMC komunikující s SD kartou. Pomocí tohoto interface je komunikováno s periferií z vyšší vrstvy. Struktura T_halInterface obsahuje informaci o velikosti workspace a prototyp pro funkci, která jej zinicializuje. Tato struktura musí být přítomna v každém HAL interface, neboť je využívána během registrace ovladače. Funkce registrující ovladač hal_interface_register() přetypovává ukazatel konkrétního HAL interface na typ T_halInterface, který obsahuje odkaz na vstupní inicializační rutinu a velikost pracovního prostoru. Smyslem přetypování je existence pouze jedné funkce registrující HAL interface nezávisle na jeho konkrétním typu. Tento princip vychází z objektového programování, dědičnosti, kdy potomek zachovává vlastnosti předka a pouze přidává svoje vlastní rozšíření.

```
typedef struct
{
// Header
T_halInterface iInterface;
// Functions
int (*Init)(void * aWorkspace, void (*isrCallback)(void * context,
SD_MCI_RESULT_STATUS state), void * context);
int (*SendCmd)(void *aWorkspace, uint32_t CmdIndex, uint32_t Argument,
RESPONSE_TYPE ExpectResp, bool pend, bool waitForCard);
int (*GetCmdResp)( void *aWorkspace, RESPONSE_TYPE ExpectResp,
T_SDMCIResponse * SDMCIResponse);
int (*Transfer)(void *aWorkspace, SD_MCI_DATA_DIRECTION direction,
uint32_t timeout, uint32_t blockSize, uint32_t numBlocks, bool
dma_enable);
int (*ioctl)(void *aWorkspace, int request, ...);
} HAL_SD_MMC;


typedef struct
{
      int (*InitializeWorkspace)(void *aWorkspace, uint32_t
physical_device);
      size_t iWorkspaceSize;
} T_halInterface;
```

Výpis 13: Interface HAL ovladače

V implementaci HAL ovladače je nutné korektně vyplnit údaje potřebné pro jeho registraci, viz výpis č. 14. Jelikož je překladačem exportován pouze symbol název struktury, která obsahuje odkazy na funkce manipulující s registry periferie, je možné tyto funkce deklarovat s klíčovým slovem static, aby jejich symboly nebyly exportovány a překladači to umožní efektivnější optimalizaci.

```
const HAL_SD_MMC LPC1788_HAL_SD_MMC =
{
// Header
{ .InitializeWorkspace = LPC1788_SD_MCI_InitializeWorkspace,
.iWorkspaceSize = sizeof(T_LPC1788_SD_MCI_Workspace),},
// Functions
.Init = LPC1788_MCI_Init,
.SendCmd = LPC1788_MCI_SendCmd,
.GetCmdResp = LPC1788_MCI_GetCmdResp,
.Transfer = LPC1788_MCI_Transfer,
.ioctl = LPC1788_MCI_ioctl, };
```

Výpis 14: Implementace rozhraní HAL ovladače

Pracovní prostor ovladače, workspace, je tvořen strukturou uvedenou na výpisu č.15, která deklaruje stavové proměnné a na prvním místě obsahuje odkaz na implementaci interface pro daný ovladač, vyplněný během registrace. Při manipulaci s ovladačem tak stačí pouze informace o umístění jeho workspace, přístup k funkcím lze umožnit vhodným přetypováním.

```
typedef struct
{
const HAL_SD_MMC *iHAL;
void (*isrCallback)(void * context, SD_MCI_RESULT_STATUS state); // isr
hook function
void * context;
} T_LPC1788_SD_MCI_Workspace;
```

Výpis 15: Struktur workspace pro HAL ovladač SD/MMC

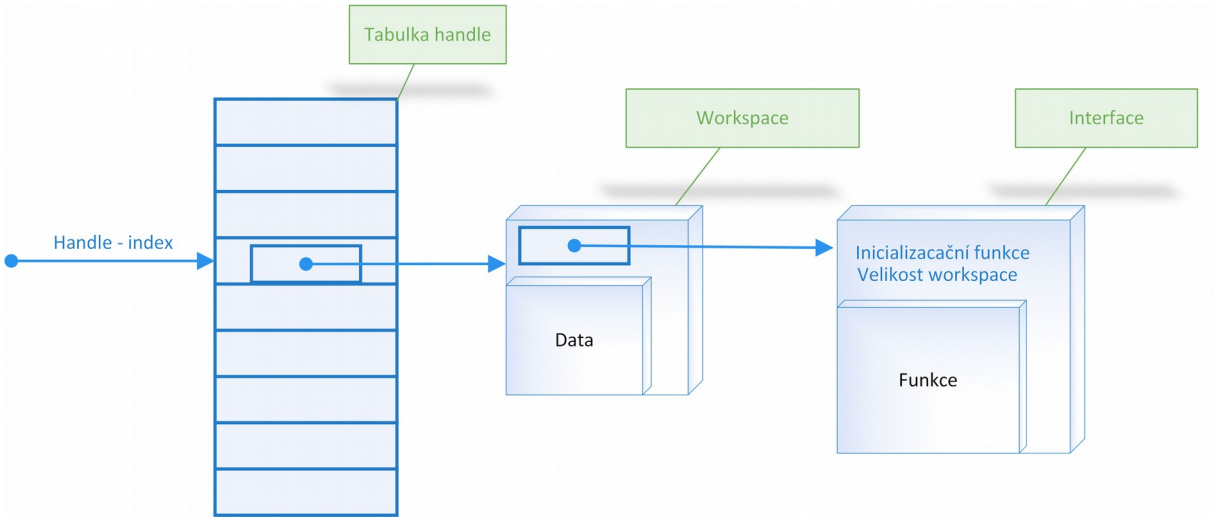
Interface pro ovladač, výpis č. 16, se liší od HAL ovladače pouze přidáním položky iID, která označuje třídu zařízení, do které ovladač patří a změnou typu parametru u funkce inicializujíc worskspace, aby mohl být předán ukazatel s konfigurací ovladače.

```
typedef struct {
 int (*InitializeWorkspace)(void *aWorkspace, void * params);
 uint16_t iWorkspaceSize;
 uint16_t iID; // ID of device.
} DEVICE_INTERFACE;
```

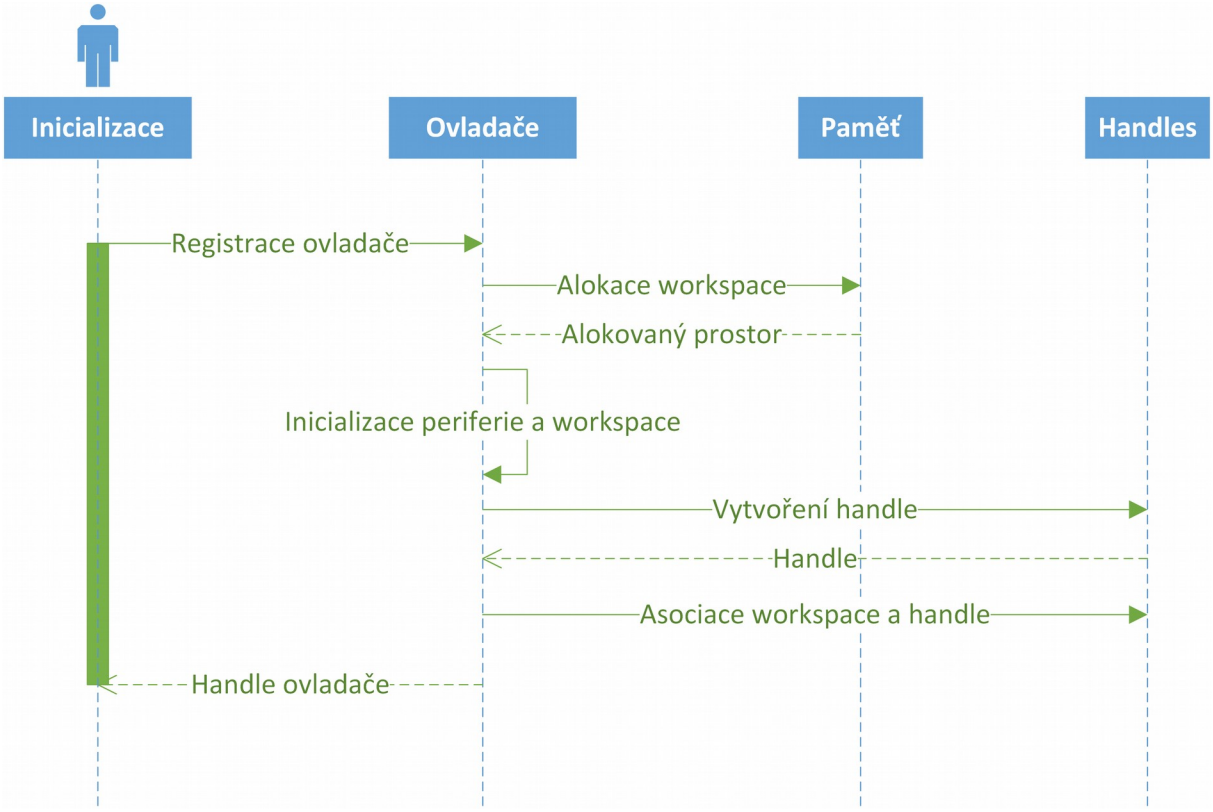
Výpis 16: Interface ovladače

Při registraci obou typů ovladačů se používají textové řetězce, jména, pro jejich identifikaci, které se ukládají do datového pole u handle. Jmenné prostory jsou na sobě nezávislé, např. ovladač pro periferii UART0 může být identifikován jako „UART0“ a HAL ovladač také „UART0“. Rozlišení prostoru je provedeno na základě typu handle, který se ukládá do tabulky.

Paměťová reprezentace ovladačů je znázorněna na obrázku č. 13 a postup registrace ovladačů je znázorněn na obrázku č. 14.



Obrázek 13: Paměťová reprezentace ovladačů



Obrázek 14: Sekvence postupu registrace ovladačů

7.7 Knihovny a systémové knihovny

Aby mohlo programové vybavení zajistit komplexní funkce, které má podporovat, integruje externí OSS knihovny. Knihovny podporující standard jazyka C není nutné upravovat. V místech, kde standard není definován nebo funkce záleží na komunikaci s periférií, obsahují knihovny svůj interface, který je potřeba do-implementovat. Soubor s implementací se nazývá adaptační vrstva a podporuje komunikaci mezi knihovnou a programovým vybavením. Např. u knihovny LWIP je vytvořena adaptační vrstva, která implementuje rozhraní LWIP_send() na periferní API rozhraní Ethernet. Obdobně je postupováno u databázové knihovny SQLite, kde adaptační vrstva obsahuje rozhraní mezi souborovým systémem a rozhraním databáze.

U některých knihoven je nutné, kromě vytvoření adaptační vrstvy, provést úpravy, neboť plně nepodporují požadavky kladené na programové vybavení, např. grafická knihovna. Další skupinu knihoven tvoří systémové knihovny. Ty slouží pro vnitřní potřeby programového vybavení a služby nejsou dostupné aplikačnímu programátorovi. Tvoří funkční subsystém, paměťový a souborový.

7.7.1 Úpravy grafické knihovny

Grafická knihovna Grlib poskytuje sadu grafických primitiv a widgetů pro vytváření uživatelských interface na displejích. Obsahuje též podporu pro vstup z dotykového rozhraní. Část grafických možností knihovny jsou zobrazeny na obrázku č. 15 (zdroj: http://www.ti.com/) . Skládá se ze tří vrstev

- Vrstva grafického displeje

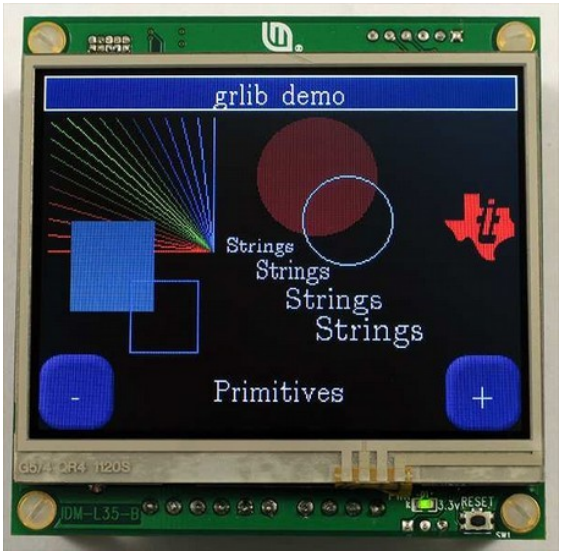
  Obsahuje podporu zobrazování dat na LCD.

- Vrstva grafických primitiv

  Vykresluje na obrazovku základní grafická primitiva jako bod, obdélník, kružnice, čára, fonty, aj.

- Vrstva widgetů

  Zapouzdření jednoho nebo více grafických primitiv do objektu, se kterým lze provádět aplikačně definované operace v závislosti na interakci s uživatelem.



Obrázek 15: Grafické možnosti knihovny Grlib

Nedostatkem této knihovny je metoda řešení zpracování událostí a vykreslování objektů, neboť původní implementace nepočítá s využitím víceúlohového prostředí. V knihovně je využíváno aktivního čekání a události od dotykového rozhraní jsou vykonávány okamžitě v rámci kontextu objektu zasílajícího událost. Modifikace knihovny pro využití v programovém vybavení spočívala v zabalení zpráv a událostí do front nabízených v rámci RTOS, konkrétně s využitím OSAL vrstvy. V nabízeném rozhraní pro aplikaci byl použit model z knihovny GTK+. Aplikační programátor nejdříve vytvoří widgety k zobrazení na displeji včetně jejich atributů a zavolá funkci grlib_main(), která cyklicky čeká na zprávy

a události zasílané z widgetů. Akce jsou vykonávány z kontextu úlohy, která volala funkci grlib_main().

Dalším nedostatkem je existence jevu roztržení obrazu (angl. tearing). Tento jev vzniká při modifikaci frame-bufferu v průběhu vykreslování grafických dat, kdy video paměť již obsahuje nový obrázek, ale původní ještě nebyl kompletně vykreslen. Část nového snímku je vykreslen a zkombinován se starým a vmístě nových dat se zobrazí nenavazující části, roztržení, viz obrázek č. 16 (zdroj: http://www.segger.com).

Obrázek 16: Tearing - Roztržení obrazu

Řešení tohoto problému spočívá ve využití přerušení načtení bázové adresy grafických dat (nebo případně vertikální synchronizace VSYNC) od řadiče LCD a použitím techniky známé jako double-buffering.

Double-buffering využívá dvě sady framebufferů, aktivní a neaktivní. Aktivní framebuffer obsahuje data, která jsou vykreslována na displej, do neaktivního jsou prováděny změny z aplikace. Pokud je potřeba změny vykreslit, v rutině přerušení od LCD řadiče se prohodí adresy framebufferů a z aktivního se stane neaktivní a naopak. U klasického double-bufferingu, bez podpory hardware, je nutné po přepnutí adres framebufferů modifikovat obsah neaktivního překopírováním grafických dat z aktivního. Při rozlišení 640 x 480 a 16 bpp je nutné přenést 600 KB i během změny několika pixelů na obrazovce.

Grafická knihovna umožňuje měnit zobrazení pouze v rámci widgetů, které obsahují souřadnice o umístění. Při změně obsahu widgetu jsou souřadnice poslány do fronty modifikovaných objektů. Tato fronta je využita během kopírování dat z aktivního do neaktivního framebufferu, kdy se kopírují pouze grafická data ze souřadnic z fronty.

7.7.2 Souborový subsystém

Souborový subsystém je v programovém vybavení přístupný pomocí standardních funkcí knihovny jazyka C (fopen, fread, fwrite, atd). Pro propojení knihovních funkcí s fyzickým souborovým systémem jsou v systému definovány funkce (open, close, read, write) v adaptačním souboru, které jsou z knihovny C volány po zpracování požadavků standardních funkcí.

Odkaz na soubor je tvořen řetězcem: „/Přípojný bod/Absolutní cesta k souboru“. Úvodní znak „/“ je povinný a označuje kořenový systém přípojných bodů. Tato konvence je převzata z GNU/Linux operačního systému, kde úvodní znak „/“ značí kořenový adresář. Relativní cesty nejsou podporovány a uživatel musí uvádět absolutní cestu k souboru.

Přípojný bod (mount point) je registrován při inicializaci systému pomocí funkce mount(). Název by měl odpovídat fyzickému umístění, např. „sd“, „usb“, „flash“, ale principiálně může být libovolný. V názvu nesmí být přítomen znak „/“, který slouží jako oddělovač přípojného bodu od cesty k souboru. Absolutní cesta k souboru se skládá z libovolného řetězce znaků, které jsou podporovány konkrétní implementací souborového systému.

Rozhraní souborového systému mezi souborovým subsystémem a jeho konkrétní implementací je zajištěno pomocí struktury devoptab_t, výpis č. 17, která obsahuje odkazy na knihovní funkce, resp. jejich obalovací procedury (wrappery). Struktura se registruje pomocí funkce register_filesystem().

```
typedef struct {
      const char *name;
      int (*open)(int mount_options, const char *path,int flags,int mode);
      int (*close)(int mount_options, int fd);
      int (*write)(int mount_options, int fd,const char *ptr,int len);
      int (*read)(int mount_options, int fd,char *ptr,int len);
      off_t (*lseek)(int mount_options, int fd,off_t pos,int dir);
      int (*fstat)(int mount_options, int fd,struct stat *st);
      int (*stat)(int mount_options, const char *file,struct stat *st);
      int (*link)(int mount_options, const char *existing, const char
*new);
      int (*unlink)(int mount_options, const char *name);
      int (*chdir)(int mount_options, const char *name);
      int (*rename)(int mount_options, const char *old, const char *new);
      int (*mkdir) (int mount_options, const char *pathname, mode_t mode);
      int (*truncate)(int mount_options, int fildes, off_t length);
      int (*sync)(int mount_options, int fd);
      int (*ioctl) (int mount_options, int filedes, int command, ...);
      int (*chmod)(int mount_options, const char *path, mode_t mode);
      int (*getcwd)(int mount_options, char *buf, size_t size);
} devoptab_t;
```

Výpis 17: Popis struktury devoptab_t

Názvy přípojných bodů, souborových systémů nesmí být během využívání služeb souborového systému měněny. Souborový subsystém používá tyto řetězce pro vyhledávání

v jeho interních tabulkách. Sdílení a zamykání přístupu k souborům je řešeno v rámci knihovny jazyka C.

7.7.2.1 Inicializace souborového systému

Aby mohl být souborový systém používán v systému, je třeba provést jeho registraci do interních tabulek programového vybavení. Registrace je prováděna při inicializaci systému, před předáním řízení aplikaci. Při inicializaci souborového systému se musí nejdříve zaregistrovat jeho devoptab_t struktura funkcí register_filesystem(). V systému je vytvořen záznam s uloženým ukazatelem na tuto strukturu. Tento záznam slouží pro pozdější mapování mezi přípojným bodem a implementací souborového systému. V systému nemohou být zaregistrovány dvě devoptab_t struktury popisující implementaci jednoho konkrétního souborového systému. Kontrola vícenásobné registrace je prováděna s využitím pole „devoptab_t->name“.

7.7.2.2 Registrace přípojného bodu

Po registraci devoptab_t struktury probíhá registrace přípojného bodu, resp. názvu zařízení, na kterém se nacházejí soubory. Registrace probíhá pomocí funkce mount(), jejíž parametry jsou název přípojného bodu, název implementace souborového systému a uživatelský parametr. Při registraci je vytvořen systémový handle typu HANDLE_MOUNT_POINT, který si uloží záznam o indexu do pole registrovaných souborových systémů (devoptab_t strukturu), název přípojného bodu a uživatelský parametr.

Při vyplňování devoptab_t struktury mohou zůstat nevyužité ty funkce, které nejsou podporovány danou implementací, resp. jejich hodnota musí být nastavena na NULL. Při volání funkce devoptab_t->open musí každá implementace při úspěšném otevření vytvořit handle typu HANDLE_FILE a v návratové hodnotě vrátit jeho číslo.

7.7.2.3 Diskový subsystém

Fyzický přístup k úložnému médiu je řešen pomocí ovladače náležející do skupiny `STORAGE_DEVICE`. V této třídě jsou implementovány funkce pro čtení/zápis bloků a zjišťování stavu/konfigurace úložného média. V případě SD karty a SDMMC periférie je řetězec následující: SD MMC Storage Device Driver→ SD MMC Device Driver → SD HAL driver.

7.7.2.4 Technika dopředného čtení

Některé Storage ovladače implementují dopředné čtení pro zvýšení výkonu systému s využitím vlastnosti časové a prostorové lokality dat, pokud je k datům datům přistupováno často, je pravděpodobné, že k těmto datům bude přistupováno i v budoucnosti a navíc pokud dochází v aplikaci k sekvenčnímu přístupu, je pravděpodobné, že budou v budoucnu zpracovávaná sousední data. Hlavním přínosem dopředné čtení (Read Ahead Buffer – RAB) je kromě nahrazení kopírování dat z operační paměti namísto pomalé komunikace mezi procesorem a periférií komunikující s paměťovým úložištěm, také zrychlení čtení dat z úložiště. RAB při generovaní čtecích příkazů využívá multi-blokové operace, tj. čtení více bloků najednou v jednom příkazu. Při využití RAB může být čtení urychleno až

několikanásobně oproti samostatným požadavkům, neboť odpadá režie vysílání příkazu pro každý přenášený blok dat. Samotná doba přenosu bývá zanedbatelná oproti času stráveném v čekání na přepnutí kontextu z rutiny přerušení.

RAB z principu nemůže být souvislý paměťový úsek (pole) využívající prostorovou lokalitu. Z důvodu multitaskingu by docházelo ke střídavému zneplatňování dat v bufferu neustálým přepisováním dopředných dat pro nezávislé úlohy. RAB rozděluje paměťový úsek na N clusterů po K blocích. Každý cluster reprezentuje na sobě nezávislou skupinu bloků. Cluster představuje souvislé paměťové místo, ve kterém jsou uložena sousední data ze sektorů úložného média. Velikost RAB v paměti zabírá N*K*velikost sektoru + struktura popisující RAB. Tato struktura obsahuje pole o N prvcích, které obsahuje počáteční adresu sektoru a čítač četnosti využití. Tento čítač je modifikován při každém využití RAB. Pokud se požadována data vyskytovala v clusteru, čítač je inkrementován, v opačném případě je dekrementován. Počet clusterů N a bloků K je volitelný během překladu zdrojového kódu.

RAB se využívá při čtení sektorů pouze za podmínky, že má jeho využití smysl. Pokud uživatelská aplikace vyžaduje souvislé čtení rovno nebo více sektorů, než je počet bloků (K) v clusteru, RAB ztrácí efektivitu. Při rozhodování o využití se porovnává počet požadovaných sektorů s hodnotou K a pokud není menší než K, RAB se nevyužije, dokonce ani v případě, že obsahuje již dříve načtená data. V opačném případě by se mohly vytvořit mezery v kontinuálním čtení sektorů a režie spojená s obsluhou požadavky by stála více času a prostředků než samotné čtení.

Pokud je RAB využit, tj. počet vyžadovaných sektorů je menší než K, započne jejich vyhledávání v bufferu. Jelikož je RAB rozdělen na N nezávislých clusterů, musí se kontrolovat celá struktura, zda neobsahuje požadovaný sektor, což znamená iteraci přes N položek a porovnání, zda leží požadovaný sektor v clusteru. Pokud je přítomen, jsou data přítomna, jsou zkopírována do uživatelského bufferu a zároveň je inkrementován čítač využití clusteru obsahující data, ostatní jsou dekrementovány.

Pokud nebyla všechna data zkopírována z bufferu, RAB nastaví čtecí požadavek na začátek nejnižšího nenalezeného sektoru, počet čtených sektorů na délku K a ukládání dat do RAB. Po vykonání čtení z úložného média se chybějící data zkopírují do uživatelského bufferu. Při hledání volného místa v RAB mají prioritu clustery, jejichž obsah je nevalidní, počáteční adresa sektoru obsahuje speciální hodnotu. Pokud takový cluster není nalezen, vybírají se ty s nejnižší hodnotou čítače. Pokud i ty jsou stejné, vybere se cluster o indexu 0.

Při požadavku na zápis se z bezpečnostních důvodů RAB nevyužívá, resp. se pouze kontroluje, zda zapisované sektory kolidují s uloženými. Pokud ano, jsou uložené sektory zneplatněny, tj. počáteční adresa sektoru pro cluster se nastaví na speciální hodnotu.

7.7.2.5 Podporované implementace souborových systémů

Programové vybavení podporuje dva typy souborových systémů, FAT FS a ROM FS. Souborový systém FAT je podporován s využitím knihovny FatFS. Tato knihovna je určena pro vestavěná zařízení a je kompatibilní s Microsoft FAT souborovým systémem. Tento

souborový systém je primárně určen pro SD kartu. FatFS nepodporuje kontrolu kolizí pro čtení/zápis pokud je soubor vícenásobné otevřen. Vícenásobné otevření je možné pouze pokud bylo každé otevření v režimu read-only. Pokud je porušena tato podmínka, může dojít k poškození dat na médiu.

Souborový systém ROM je podporován s využitím knihovny ROMFS a je primárně určen k uložení neměnných souborů, např. fontů, obrázků aj., respektive těch souborů, které jsou nutné pro běh samotného systému. Tyto soubory jsou součástí binárního obrazu aplikace. Pokud zařízení disponuje Při použití procesorového modulu s externí NOR Flash pamětí, je linkerem zajištěno jejich uložení do externí paměti. Souborový systém tvoří struktura ROM_FILE_TABLE, výpis č. 18, kde name je jméno souboru (vč. popisu adresářové cesty), content je ukazatel na obsah souboru a size je velikost souboru v bajtech. Kód uvedený v příloze B vygeneruje potřebné struktury pro ROMFS ze souborů uložených na disku vč. zachování adresářové struktury. V systému je ROMFS obvykle přístupný s prefixem /rom/, záleží na inicializační části, jaký název mount pointu bude přidělen. V tomto souborovém systému je povoleno pouze čtení. Pokusy o otevření pro zápis skončí chybovým stavem.

```
struct ROM_FILE_TABLE
{
	const char * name;
	const uint8_t * content;
	const size_t size;
};
```

Výpis 18: Struktura popisu souboru umístěného v ROM

7.7.3 Paměťový subsystém

Umístění a rozvržení paměti určené pro proměnné je uvedeno v linker skriptu. Proměnné jsou umísťované automaticky překladačem do .bss nebo .data segmentu v závislosti na kontextu proměnné. Umístění do jiného segmentu je možné pouze s využitím atributů. Definované rozmístění proměnných se nazývá statické rozvržení a nevýhodou je nutnost znát dopředu veškeré paměťové požadavky aplikace během vytváření obrazu. Toto omezení není možné splnit u systémů, které tvoří platformu pro vývoj aplikací, proto programové vybavení nabízí možnost alokovat a dealokovat paměť za běhu uživatelské aplikace s využitím dynamické paměti.

Dynamická alokace je založená možnosti vytvářet a rušit paměťový prostor pro proměnné během činnosti programu na základě požadavků z aplikace. Dynamická paměť je tvořena statickým alokováním bajtového pole, které je a priori považované za prázdné a je spravováno knihovnou paměťového alokátoru. Počet polí a knihoven může být libovolný a kombinací statického umístění a typu knihovny lze reagovat na různé požadavky na rychlosti, velikosti a chování dynamické paměti.

Programové vybavení rozlišuje z pohledu použití tři druhy paměti

- Paměť ovladačů

Paměť určená pro potřeby ovladačů, jsou zde převážně uloženy jejich workspace struktury.

- Paměť operačního systému

  Paměť určená pro potřeby operačního systému, jsou zde převážně uloženy datové struktury pro primitiva (mutex, semafor) a paměť zásobníku jednotlivých úloh.

- Uživatelská paměť

  Paměť určená pro potřeby aplikace, do této paměti jsou přesměrována volání typu malloc/calloc a free.

Vlastní inicializace paměti a mapování do adresního prostoru (statická alokace) je provedena ve speciální části programového vybavení nazývané Board Support Package.

Standardní knihovna jazyka C poskytuje základní funkce malloc() a free() pro práci s dynamickou pamětí, ale neumožňují parametrizovat účel paměti, pro který má být alokována. Programové vybavení nabízí alternativu ke knihovním funkcím a přidává funkce s prefixem BSP_, které obsahují parametr, ze kterého typu paměti má být prostor alokován (ON_CHIP_DRIVERS, ON_CHIP_RTOS, SDRAM).

Dynamická paměť není obecně doporučována na vestavěných zařízeních, resp. při používání RTOS z důvodu časového nedeterminismu alokace požadované velikosti nebo dealokace při spojování bloků. Programové vybavení proto používá statickou alokaci paměti pro kritické úlohy, jinak je použita dynamická paměť. Uživatelská aplikace by měla dodržovat obdobný model použití.

8 Koncepce zařízení docházkového terminálu

Z hlediska hardware je se grafický terminál složen z desky plošných spojů, na které je umístěn mikrokontrolér, externí součástky (rezistory, kondenzátory, externí čipy aj.) a konektory pro připojení externích periférií. Programové vybavení musí být schopno reagovat na hardwarové změny, pokud dojde k aktualizaci součástky, která vyžaduje odlišnou inicializaci, resp. ovládání. Například při výměně grafického displeje za řádkový a dotykového rozhraní za klávesnici je logika aplikace zachována, pouze se mění vizualizační a vstupní část.

U vestavěných zařízení je přílišná obecnost v rozporu s nízkými zdroji, které mohou vestavěné systémy nabídnout. Nelze tedy jednoduše oddělit hardware a software, obě domény jsou na sobě závislé a vzájemně se ovlivňují. Programové vybavení musí definovat minimální hardwarové požadavky, které každé zařízení musí splňovat, aby mohla být umožněna deklarovaná funkcionalita. Souhrn hardwarových požadavků je označen jako koncept UPM, podle procesorového modulu. Například pokud existuje požadavek na bezpečné nahrávání firmware i při výpadku napájení během aktualizační procedury, zařízení musí obsahovat nevolatilní paměť, která bude uchovávat obraz aplikace a kód, který zajistí obnovu systému po výpadku.

Programové vybavení musí také zajistit přípravu prostředí, tj. inicializace paměti, ovladačů, respektive celého systému, aby aplikační programátor pouze využíval služby nabízené programovým vybavením. Tímto způsobem je zajištěna přenositelnost kódu aplikace z různých zařízení na jiná, aniž by byla potřeba její výrazná modifikace.

Z hlediska funkce systému je nutné, aby programové vybavení definovalo, obsahovalo a zajišťovalo:

- minimální hardwarová konfigurace terminálu
- zavaděč systému
- inicializace systému

8.1 Minimální hardwarová konfigurace docházkového terminálu, koncept UPM

Programové vybavení nahlíží na terminál jako na zařízení, které se skládá ze tří hlavních částí, procesorového modulu, základnové a dceřiné desky plošných spojů. Každá část v terminálu může obsahovat různou konfiguraci hardware, respektive chybět v případě dceřiné desky. Kombinace základnové desky a její dceřiných desek tvoří zařízení, mající své unikátní označení a identifikátor, který udává hardwarovou konfiguraci.

Každé zařízení musí též obsahovat součástky:

- SPI Flash paměť

Paměť o velikosti minimálně 1MB, kde 512KB je určeno pro uložení záložního firmware a identifikátorů, zbylých 512KB je určeno pro update firmware u procesorových modulů bez NOR Flash paměti.

- Tlačítko záložního firmware

  Stiskem tlačítka během spouštění systému dojde k přehrání záložního firmware z SPI Flash paměti do procesorového modulu. Tlačítko spíná proti zemi a vstup je na UPM modulu invertován, aby hodnota log 1 znamenala stisknutí tlačítka.

- Diagnostické LED

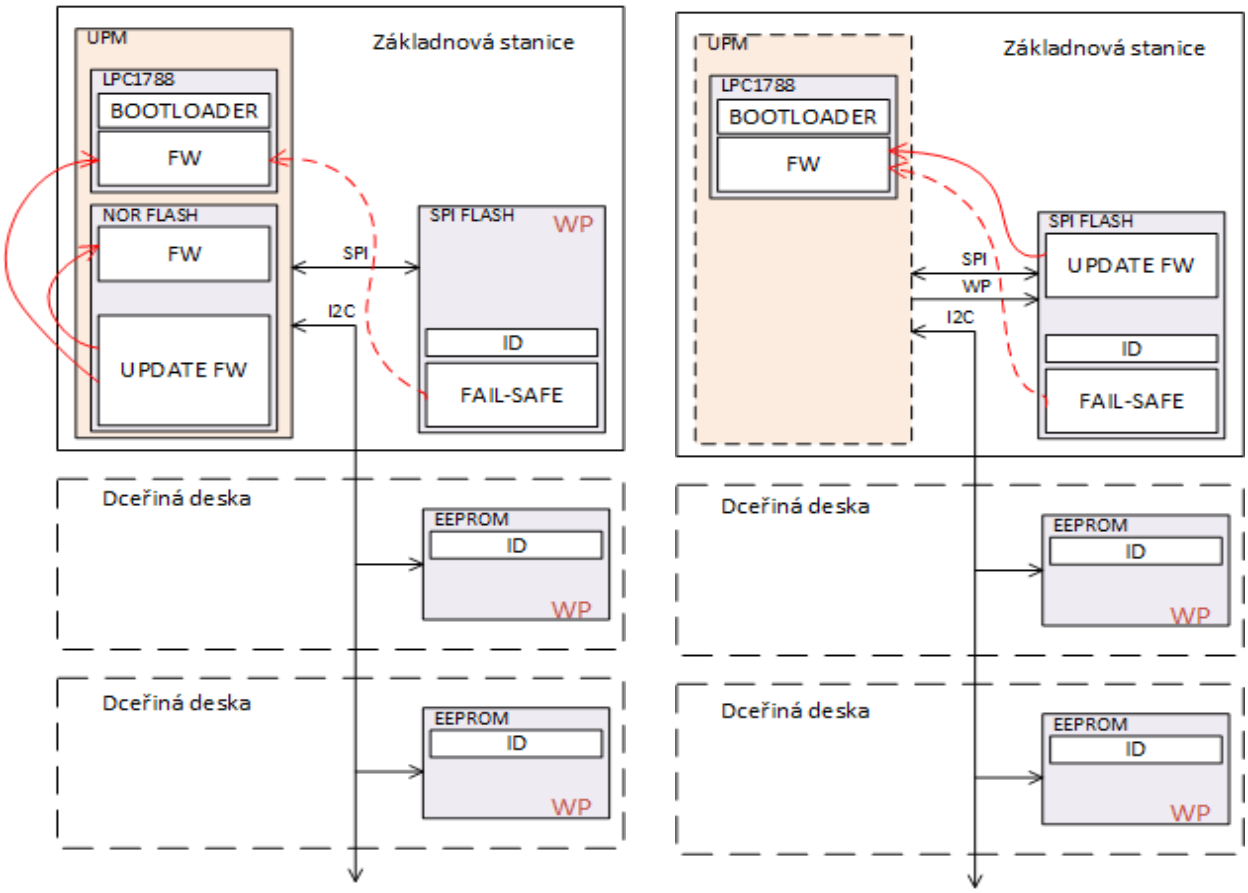
  Zelená a červená LED zobrazují průběh spouštění systému a jsou ovládané přímo z mikrokontroléru, výstupní hodnota log 1 značí rozsvícení diody, log 0 zhasnutí.

- Reset tlačítko

  Tlačítko resetuje (uvede v reset) procesorový modul. Reset ostatních periférií záleží na typu zařízení.

Základnová deska je DPS, která obsahuje konektory pro připojení procesorového modulu a další povinné součástky konceptu UPM, ostatní součástky záleží na typu zařízení. Z bezpečnostních důvodů musí být zapojení SPI Flash hardwarově chráněno pomocí Write Protect ochrany proti přepsání obsahu paměti z aplikace. Tato ochrana je vypnuta, pokud konfigurace procesorového modulu neobsahuje NOR Flash a SPI paměť je využita jako úložiště pro update firmware. Pokud zařízení podporuje dceřiné desky, je vyvedena I2C sběrnice z UPM modulu a zároveň základnová deska obsahuje pull-up rezistory pro identifikační sběrnici I2C.

Dceřiná deska je jedna nebo více DPS, které obsahují přídavné funkce hardware pro určitý typ zařízení. Deska obsahuje I2C EEPROM s identifikačními údaji, která je připojena na identifikační I2C sběrnici vyvedenou ze základnové desky. Pull-up rezistory se na této desce nepřipojují. EEPROM paměť musí být hardwarově chráněna proti zápisu. V případě připojení více dceřiných desek k základnové desce, musí být umožněna změna nastavení I2C adresy paměti. Koncept zařízení UPM z pohledu DPS je znázorněn na obrázku č. 17.



Obrázek 17: Možná konfigurace zařízení konceptu UPM

8.1.1 Pinové zapojení součástek

V tabulce č. 4 je uvedeno zapojení součástek z konceptu UPM na piny procesoru. Spoje na základnové desce musí být vedeny tak, aby byla dodržena specifikace z tabulky. Programové vybavení používá tyto údaje při inicializaci a kontrole konfigurace terminálu.

Popis zařízení	Název pinu u součástky	Pin procesoru
SPI Flash paměť	MOSI	P2[27]
	MISO	P2[26]
	CLK	P2[22]
	/CS	P2[23]
Zelená LED	-	P1[13]
Červená LED	-	P4[31]
Identifikační I2C sběrnice (volitelná)	SCL	P0[28]
	SDA	P0[27]
Tlačítko záložního firmware	-	P4[27]

Tabulka 4: Závazné zapojení periferií na piny mikrokontroléru

8.1.2 Paměťová mapa konceptu UPM

Paměťová mapa stanovuje adresy, na kterých programové vybavení vyhledává informace o terminálu. Tyto informace mají podobu strukturované tabulky a jsou vždy příslušné konkrétnímu zařízení. Údaje ve strukturách jsou chráněny pomocí kontrolního součtu typ CRC-CCITT, kontrolní součtu firmware pomocí CRC-32. Konfigurace CRC vlastností se provádí dle doporučených hodnot z uživatelského manuálu pro LPC1788 [17]. Pokud je záznam větší než jeden byte, používá se pro uložení formátu Little-Endian.

V terminálu může být současně přítomno až pět různých pamětí. Koncept UPM proto rozlišuje paměťovou mapu pro:

- Procesorový modul UPM - Interní Flash mikrokontroléru, externí NOR Flash a SDRAM
- SPI Flash
- I2C EEPROM

Mapování externích pamětí do adresního prostoru mikrokontroléru LPC1788 je uvedeno v tabulce č. 5. Adresace interní Flash paměti je dána výrobcem MCU.

Typ	Velikost	Adresa
LPC1788 Interní Flash	512 KB	0x00000000
NOR FLASH S29GL256P (volitelné)	16 MB	0x80000000
SDRAM MT48LC4M32B2B5 (volitelné)	16 MB	0xA0000000

Tabulka 5: Adresní rozsah interní a externím pamětí

Rozložení firmware v adresním prostoru mikrokontroléru je uvedeno v tabulce č. 6.

Typ	Velikost	Adresa	Umístění
Zavaděč systému (Bootloader)	až 16 KB	0x00000000	LPC1788 interní Flash
Inicializační část programového vybavení + aplikace	až 496 KB	0x00004000	LPC1788 interní Flash
Aplikace	až 7,5 MB	0x80000000	NOR Flash
Primary Master Boot Record UPM NOR Flash	až 16 KB	0x80800000	NOR Flash
Dynamická paměť (Heap)	až 16MB	0xA0000000	SDRAM

Tabulka 6: Rozložení firmware v adresním prostoru

Rozložení kontrolních součtů aplikace v adresním prostoru mikrokontroléru je uvedeno v tabulce č. 7.

Adresa	Velikost [B]	Popis
0x00004020	4	Velikost firmware v interní paměti
0x00004024	4	Kontrolní součet firmware v interní paměti
0x00004028	4	Velikost firmware v externí paměti (volitelné)
0x00004034	4	Kontrolní součet firmware v externí paměti (volitelné)

Tabulka 7: Rozložení kontrolních součtů aplikace

V paměti SPI Flash je uložen záložní firmware a identifikace základnové desky ve struktuře Primary Master Boot Record (PMBR) SPI Flash, tabulka č. 9. Počáteční adresa 0x8000 je zvolena z důvodu nemožnosti uzamknout adresový prostor 0x00000 – 0x7FFFF v některých SPI Flash (S25FL208K). Zařízení, které nedisponují NOR FLASH pamětí na UPM modulu mohou využívat část SPI FLASH pro uložení update firmware, který je popsán ve struktuře Secondary Master Boot Record (SMBR) SPI Flash, tabulka č. 10. Adresy záznamů v paměti SPI Flash jsou uvedeny v tabulce č. 8.

Absolutní adresa	Velikost [B]	Popis
0x00000000	16	Secondary Master Boot Record SPI Flash
0x00080000	20	Primary Master Boot Record SPI Flash

Tabulka 8: Adresy záznamů Boot Record v paměti SPI Flash

Offset	Velikost[B]	Popis	Poznámka
0x00	4	Adresa začátku identifikační části základnové desky v SPI Flash	Adresa musí být <= velikost SPI Flash - velikost identifikační části
0x04	4	Adresa začátku záložního firmware ve SPI Flash	Adresa musí být <= velikost SPI Flash minus velikost záložního firmware
0x08	4	Velikost záložního firmware	Velikost musí být menší než 512KB minus velikost zavaděče a musí být větší jak 0
0x0C	4	Kontrolní součet záložního firmware	CRC-32
0x10	4	Kontrolní součet Primary Master Boot Record SPI Flash	CRC-CCITT

Tabulka 9: Primary Master Boot Record SPI Flash

Offset	Velikost[B]	Popis	Poznámka
0x00	4	Adresa začátku update firmware ve SPI Flash	Adresa musí být <= velikost SPI Flash - velikost záložního firmware
0x04	4	Velikost update firmware	Velikost musí být menší než 512KB – velikost zavaděče a musí být větší jak 0.
0x08	4	Kontrolní součet update firmware	CRC-32
0x0C	4	Kontrolní součet Secondary Master Boot Record SPI Flash	CRC-CCITT

Tabulka 10: Secondary Master Boot Record SPI Flash

I2C EEPROM obsahuje identifikační údaje pro dceřiné desky. Adresa záznamu je uvedena v tabulce č. 11.

Absolutní adresa	Velikost [B]	Popis
0x00000000	12	Identifikační údaje dceřiné desky

Tabulka 11: Adresa identifikačních údajů dceřiné desky

8.1.3 Identifikátory konceptu UPM

Identifikační údaje o deskách využívá programové vybavení pro kontrolu správného typu firmware a jemu příslušnému terminálu. Pro záznamy platí samé nastavení jako u záznamů v paměťové mapě. Údaje uložené v rámci desek a ne interní paměti mikrokontroléru umožňují měnit procesorové moduly mezi terminály bez nutnosti jejich personalizace pro dané zařízení.

Struktura Identifikační údaje základnové desky je uvedena v tabulce č. 12. Nejdůležitější parametr se nachází na offsetu 0x08 - Identifikace zařízení. Tento jednobajtový identifikátor určuje typ terminálu a programové vybavení kontroluje, zda je typ terminálu shodný s typem programového vybavení. Počáteční adresa je určena v PMBR SPI Flash na offsetu 0x0.

Offset	Velikost[B]	Popis	Poznámka
0x00	4	Časová značka (unix timestamp), kdy byla konfigurace nahrána	-
0x04	4	Sériové číslo desky	-
0x08	1	Identifikace zařízení	-
0x09	1	Major verze desky	-
0x0A	1	Minor verze desky	-
0x0B	1	Typ záložního firmwaru uložený ve Flash	0x07 - fw pro LPC1788 (arm v7) 0x71 - fw pro LPC1788 (arm v7), zašifrovaná verze
0x0C	6	Unikátní Ethernet IMA MAC adresa je uložen ve SPI Flash.	6 krát 0x00 – pokud Ethernet není využíván 0xC – první oktet ..... 0x11 – šestý oktet
0x12	1	Externí NOR Flash paměť *)	0x0 - Paměť nemůže být připojena 0x1 - Paměť může být připojena
0x13	1	Nevyužito	Hodnota 0xFF
0x14	4	Kontrolní součet identifikace CRC	CRC(16)-CCITT

Tabulka 12: Struktura Identifikačního údaje základnové desky

*) Zařízení, které nepodporují připojení UPM modulu pomocí dvouřadých konektorů a nemají připojenu Externí NOR Flash paměť musí mít tento bit vynulovány. Na základě této hodnoty Bootloader provádí detekci update firmware v Externí NOR Flash paměti.

Struktura Identifikační údaje dceřiné desky je uvedena v tabulce č. 13. Podobně jako Identifikační údaje základnové desky, tato struktura obsahuje záznam Identifikace zařízení, který je využit pro identifikaci typu terminálu v programovém vybavení. Počáteční adresa se nachází na ofsetu 0x0 v I2C EEPROM.

Offset	Velikost[B]	Popis	Poznámka
0x00	4	Časová značka (unix timestamp), kdy byla konfigurace nahrána	-
0x04	4	Sériové číslo desky v rámci IMA	-
0x08	1	Identifikace zařízení	-
0x09	1	Major verze desky	-
0x0A	1	Minor verze desky	-
0x0B	1	Nevyužito	Hodnota 0xFF
0x0C	4	Kontrolní součet identifikace CRC	CRC(16)-CCITT

Tabulka 13: Struktura Identifikačního údaje dceřiné desky

8.2 Zavaděč systému - Bootloader

Primární činností Bootloaderu je spuštění hlavní aplikace nacházející se ve vnitřní Flash paměti mikrokontroléru. Svou funkcionalitou je podobný vestavěnému Bootloaderu uloženému v Boot ROM LPC1788 [17], kdy přidává navíc rozšířenou kontrolu uživatelské aplikace a možnost obnovení systému po výpadku napájení během upgrade firmwaru. Z pohledu mikrokontroléru se jedná o standardní aplikaci, která musí splňovat podmínky validního kódu [17]. Bootloader poskytuje pouze minimální funkcionalitu a kromě nezbytně nutných periférií pro svoji činnost se nekonfiguruje žádné další zařízení uvnitř/vně procesoru.

Výčet funkcí Bootloaderu:

- Kontrola validity uživatelského firmware a jeho spuštění
- Nahrání záložního firmware, zpracování žádosti o jeho nahrání
- Nahrání update firmware

Bootloader je nahrán do vnitřní paměti mikrokontroléru od adresy 0x00000000 – 0x00003FFF, tím je zajištěno vykonávání programu po každém zapnutí napájení nebo resetu. Z bezpečnostních důvodů je po spuštění hlavní aplikace zablokován adresní prostor firmware zavaděče pomocí MPU vůči čtení, zápisu a vykonávání kódu. Tato ochrana navíc umožňuje odhalit chyby uživatelské aplikaci, např. dereference nulového funktoru, řetězcové funkce s NULL ukazatelem, či běžná chyba vynechání kontroly návratové hodnoty u funkce malloc(). Povolení MPU ochrany zabrání také teplému resetu, kdy se začne opětovně vykonávat kód od adresy 0x00000000, chování programu obdobné jako při zapnutí napájení, ale aniž by byl proveden reset periférií. Při pokusu o čtení, zápisu nebo vykonávání kódu v adresním rozsahu Bootloaderu dojde k vyvolání výjimky MemFault s nastavenými příznaky o porušení ochrany paměti.

Během doby vykonávání Bootloaderu je zakázáno IRQ přerušení a zjišťování příznaků je prováděno pomocí aktivního čekání. Zakázání přerušení umožňuje využít prostor alokovaný

pro vektory přerušení (196 Bajtů) jako programový kód Bootloaderu. Ochrana bezpečnostním časovacím obvodem (Watchdog) při vykonávání zavaděče je aktivována pouze v případech, kdy programová paměť neobsahuje validní uživatelský program a Bootloader provádí nahrání záložního nebo update firmware. Nastavením a aktivováním časovače před spuštěním hlavní aplikace by vedlo k jejímu omezování z časového hlediska.

8.2.1 Spuštění hlavní aplikace

Bootloader před předáním řízení hlavní aplikaci kontroluje, zda se v interní a externí programové paměti nachází validní firmware. Kontrola validního firmware uživatelské aplikace je prováděna na základě hodnot uložených v nevyužitých IRQ vektorech uživatelské aplikace. V tabulce č. 14 je uvedeno umístění součtů a velikostí interního firmware vzhledem k bázové adrese začátku uživatelské aplikace.

Offset	Velikost	Popis
0x20	4	Velikost firmware v interní paměti
0x24	4	Kontrolní součet firmware v interní paměti
0x28	4	Velikost firmware v externí paměti (volitelné)
0x34	4	Kontrolní součet firmware v externí paměti (volitelné)

Tabulka 14: Umístění součtů a velikostí firmware

Bootloader v Debug verzi umožňuje vypnutí kontroly validního firmware, pokud detekuje speciální hodnotu 0xAABBCCDD na pozici kontrolního součtu interního firmware. Vypnutí/zapnutí kontroly je řízeno pomocí makra NDEBUG a v Release verzi je tato funkcionalita deaktivována. Nemá-li se kontrolovat firmware v externí NOR Flash, musí velikost a kontrolní součet externí paměti současně obsahovat hodnotu 0. U kontrolního součtu externí paměti musí existovat shoda s informací z Identifikačního záznamu pro Základnovou stanici. Za porušení validity firmware se považuje situace, kdy Identifikační záznam obsahuje příznak, že externí paměť nemůže být připojena, ale velikost nebo kontrolní součet externí paměti je nenulový.

Začátek uživatelského firmware, pro který se počítá CRC z interní paměti se nachází na offsetu 0x0 od počáteční adresy aplikace (0x00004000). Aby se zabránilo počítání CRC z CRC, které ještě není v průběhu výpočtu známo, offset pozice kontrolního součtu se nezapočítává do výsledné hodnoty. Začátek uživatelského firmware, pro který se počítá CRC z NOR Flash paměti se nachází na offsetu 0x0 od počáteční adresy aplikace (0x80000000). Pro urychlení výpočtu CRC algoritmus počítání vydělí velikost příslušného firmware čtyřmi k získání počtu 32bitových slov, proto musí být velikost interního a externího firmware dělitelná čtyřmi.

Při výpočtu kontrolních součtů firmware používá Bootloader CRC-32. Parametry nastavení kontrolního součtu odpovídají nastavení CRC periférie v procesoru LPC1788.

Po úspěšné validaci uživatelského firmware Bootloader uvede mikrokontrolér do stavu podobném po resetu. Hodnota registru VTOR (Vector Table Offset Register) je nastavena na začátek adresy uživatelské aplikace (0x00004000), aby se vektory přerušení načítaly z oblasti uživatelského firmware. Bootloader nastaví zásobník na hodnotu uloženou na adrese 0x00004000 a provede skok na hodnotu uloženou na adrese 0x00004004, což je chování odpovídající procesorům s Cortex-M3 jádrem.

8.2.2 Nahrání záložního firmware, zpracování žádosti o jeho nahrání

Záložní firmware slouží jako bezpečností pojistka pro situace, kdy dojde k poškození aplikace uložené ve vnitřní paměti mikrokontroléru, a také jako iniciální firmware pro nové zařízení. Tento firmware je uložen v externí SPI Flash paměti.

Nahrávání je aktivováno v těchto případech

- Uživatelský firmware je nevalidní a neexistuje update firmware
- Uživatel stiskl tlačítko nahrání záložního firmware
- Uživatel si vyžádal nahrání záložního firmware

Bootloader se po neúspěšném kroku ověření validity uživatelské aplikace pokusí nalézt a nahrát update firmware, který může být uložen v externí SPI Flash, případně NOR Flash v závislosti na konfiguraci základnové stanice. Kontrola přítomnosti update firmware je prováděna pomocí kontrolních součtů oblasti popisující jeho umístění. Při neúspěšném nahrání update firmware nebo jeho neexistenci se přechází k nahrání záložního.

Tlačítko nahrání záložního firmware slouží převážně k nahrání iniciálního (záložního) firmware daného zařízení pro nové UPM moduly, který by již měl umožnit nahrát uživatelský firmware. Tlačítko je monitorováno po dobu 100 ms od startu Bootloaderu a je provázeno blikáním diagnostických LED v intervalu 500 ms, pokud je drženo v aktivním stavu. Spuštění nahrávání proběhne až po uvolnění tlačítka. Z bezpečnostních důvodů je stanoven maximální limit doby stisku na 16 s, poté je stav vyhodnocen jako nestisknuto. Pokud by nebyl zaveden časový limit na stisknutí, změna elektrických parametrů tlačítka, např. snížením odporu rozpojeného stavu v důsledku kondenzace vodních par, by mohla vést k mylné interpretaci o požadavku na nahrání záložního firmware.

Vstupní rozhraní Bootloaderu zajišťující provedení změnu uživatelské aplikace, funkce IAP(int), umožňuje na základě předaného parametru vyžádat si nahrání záložního firmware namísto update, tj. zavaděč provede nahrání záložního firmware i v případě, pokud existuje v paměti správný update firmware. Tato vlastnost ošetří situaci, pokud je omylem nahrán update firmware, který není platný pro daný typ Základnové Stanice. Pokud BSP z aktualizovaného firmware detekuje hardwarovou konfiguraci, pro kterou není určen, může požádat Bootloader o nahrání záložního firmware. Bez této funkcionality by systém nemohl být uveden korektního stavu. Žádost o provedení update firmware by vedla k opětovnému nahrání nesprávné aplikace. Zneplatnění update firmware pomocí BSP není možné z důvodu

bezpečnosti, neboť BSP nesmí inicializovat piny, které nejsou definované, jelikož by na nich mohl být připojen akční člen.

8.2.3 Nahrání update firmware

Bootloader se snaží po neúspěšné validaci uživatelské aplikace najít validní update firmware v externí paměti SPI případně NOR Flash. Pokud není v identifikačním záznamu základnové stanice, příznak Externí NOR Flash paměť, uvedena možnost připojení externí paměti, zavaděč hledá update firmware v SPI paměti. V opačném případě se nejdříve zkontroluje přítomnost osazení paměti NOR, poté je na základě výsledku kontroly rozhodnuto, zda bude firmware hledán v této paměti, případně v SPI. Dvoufázová příznaková detekce je použita kvůli možnosti připojení akčních členů na piny, kde se nachází NOR Flash. Bez použití příznaků by Bootloader aktivací EMC periférie mohl tyto členy uvést do nedefinovaného stavu.

Hledání je prováděno na základě paměťové mapy UPM pro vybraný typ úložiště update firmwaru. Validní update firmware je detekován spočítáním kontrolního součtu a porovnáním s hodnotou uloženou v Secondary Master Boot Record SPI Flash nebo Primary Master Boot Record UPM NOR Flash. Pokud je nahráván update firmware do NOR Flash, jsou sektory obsahující programový kód ochráněny pomocí Persistent Protection Bits. Při nahrávání firmware bliká zelená diagnostická LED.

Bootloader nabízí uživatelské aplikaci možnost požádat o provedení nahrání update firmware po jeho uložení v SPI nebo NOR Flash pomocí funkce s prototypem void IAP(int), jejíž vstupní bod se nachází na adrese 0x000003FC0, posledních 64 byte adresního prostoru Bootloaderu. Při volání aktualizační funkce pomocí ukazatele je třeba nastavit bit 0 na hodnotu 1, neboť jádro procesoru pracuje pouze v tzv. Thumb režimu a poslední nulový bit by znamenal přechod do režimu ARM, který není podporován a je vyvolána výjimka. Kód na výpisu č. 19 znázorňuje možné použití rutiny pro aktivaci update firmware.

```
#define IAP_LOCATION (0x00003ff0 + 1)
typedef void (*IAP)(int);
IAP iap_entry = (IAP) IAP_LOCATION;
//int load_fail_safe = 0 – force Bootloader to load failsafe image
//int load_fail_safe != 0 – load update image
iap_entry(load_fail_safe);
```

Výpis 19: Vstupní bod procedury aktualizace firmware

Před voláním IAP() funkce je nutné zpřístupnit její adresní prostor v MPU. Po předání řízení do funkce je automaticky vypnuto MPU a provede se smazání prvního sektoru v interní Flash paměti mikrokontroléru uživatelské aplikace, de-facto její zneplatnění. Následuje

restart, po kterém Bootloader provede nahrání update případně záložního firmware na základě předaného parametru.

8.2.4 Kroky vykonané funkcí IAP()

Funkce IAP() je definována s atributem „noreturn“, není předpokládáno vrácení se do volající procedury. Pokud se tak z neočekávaného důvodu stane, BSP či aplikace by okamžitě měla vyvolat reset systému, neboť došlo k chybě hardware.

Následující kroky jsou prováděné funkcí IAP() po jejím zavolání:

- Zakázání přerušení
- Vypnutí MPU ochrany zakázáním periférie
- Nastaví používání MSP jako aktuálního zásobníku
- Nastaví hodnotu zásobníku na hodnotu definovanou Bootloaderem
- Nastavení bázové adresy tabulky vektorů na hodnotu 0x0000000
- Vypnutí PLL a nastavení zdroje hodin na vnitřní IRC
- Zavolání funkce, která zneplatní první sektor aplikace, na základě vstupního parametru
- Parametr 0
  - Smazání prvního sektoru aplikace
  - Nahrání hodnoty 0 do prvního sektoru
  - Reset procesoru
- Parametr != 0
  - Smazání prvního sektoru aplikace
  - Reset procesoru

Po resetu procesoru a spuštění Bootloaderu je zjištěno selhání kontroly firmware aplikace. Na základě obsahu prvního sektoru aplikace Bootloader rozhoduje, který typ firmware (záložní/update) bude nahrán. Při mazání interní Flash paměti jsou všechny bity nastaveny na hodnotu 1, proto funkce IAP() při vstupním parametru různém od hodnoty 0 neprovádí žádné další akce vyjma smazání sektoru.

8.2.5 Ochrana běhu programu zavaděče

Před nahráním záložního nebo update firmware, Bootloader aktivuje Watchdog periférii s maximální možnou periodou, která činí přibližně 134.000 milisekund (2 minuty a 14 sekund), během které musí Bootloader nahrát libovolný firmware do vnitřní Flash, resp. externí NOR Flash programové paměti včetně opakovaných pokusů. Reaktivační sekvence

„FEED“ se provádí pouze ve stavu, kdy Bootloader ukončí svoji činnost zastavením a mezi neúspěšnými pokusy o nahrání záložního/update firmware. Definovaný maximální čas vypršení pro Watchdog postačuje na pokrytí nahrání všech druhů firmwaru. Tabulka č. 15 zobrazuje dobu trvání nahrání maximální povolené velikosti firmware, která činí 496 KB pro záložní a update SPI Flash a 7,98 MB (8.176 KB) pro Update NOR Flash.

Typ přehrání firmware	Čas nahrání [ms]	
	Debug konfigurace	Release konfigurace
Nahrání záložního firmware	25933	24860
Nahrání update firmware z SPI Flash	25749	24676
Nahrání update firmware z NOR Flash	61360	62020

Tabulka 15: Doba trvání nahrání firmware

8.2.6 Ukončení činnosti zavaděče

Bootloader ukončuje svoji činnost předáním řízení uživatelské aplikaci. Pokud při jeho běhu nastanou komplikace, přechází do režimu ukončení, které je reprezentováno virtuálním zastavením nebo resetem procesoru.

Ukončení zastavením představuje pro Bootloader chybový stav, který mu brání v definované činnosti. Zastavení je zajištěno přechodem do nekonečné smyčky v případech, kdy je detekován odlišný typ mikrokontroléru než LPC1788 nebo není možné nahrát záložní firmware. Tato smyčka obsahuje „feedovací“ sekvenci pro případ, že byl aktivován časovač Watchdogu. V prvotní fázi Bootloaderu je kontrolováno identifikační číslo procesoru pomocí interních IAP funkcí [17]. Pokud identifikátor neodpovídá hodnotě pro LPC1788 procesor začne vykonávat smyčku, aniž by aktivoval diagnostické LED, neboť cílový mikrokontrolér může vyžadovat odlišné nastavování pinů než LCP1788. Selhání nahrání záložního firmware může být vyvoláno několika případy

- Chyba komunikace s SPI Flash pamětí
- Nevalidní konfigurace PMBR, Identifikačního záznamu
- Selhání zápisu do interní Flash paměti mikrokontroléru

Tento stav je provázen rozsvícením červené LED. Bootloader se při neúspěšném nahrání firmware snaží provést operaci znova, dokud nedojde limit pro opakované pokusy, který se nastavuje v konfiguračním souboru Bootloaderu. Stav zastavení je možné opustit pouze vynuceným resetováním mikrokontroléru.

Ukončení resetováním je prováděno z důvodu nastavení mikroprocesoru do výchozího stavu. Provádí se po úspěšném nahrání záložního nebo update firmwaru nebo pokud dojde k neočekávané chybě, která je charakterizována odklonem od programového sledu instrukcí, např. návrat z funkce main() nebo generování hard-fault výjimky. Reset procesoru se vykoná

pomocí zápisu do NVIC registru AIRCR. Reset u externích zařízení se provede pouze v případě, kdy je jejich RESET vstup připojen na pin RSTOUT/ u LPC1788.

8.2.7 Ladící výstupy

Výpis ladících informací během provádění Bootloaderu se děje pomocí SWO (Serial Wire Output) výstupu STIMULUS PORT 0, který je součástí debugovací architektury mikrokontroléru. Tyto výpisy je možné zaznamenat pomocí zařízení připojené k JTAG/SWD rozhraní. Bootloader je schopen pomocí stavových bitů mapovaných do adresního prostoru zkontrolovat, zda je debugovací zařízení připojené a v opačném případě je výstup potlačen, a tak nedojde k výraznému časovému ovlivnění běhu programu.

Ladící výpis je reprezentován unikátním stavovým kódem a případně textovým popisem. Nejčastěji používané výpisy jsou reprezentovány pouze textovým popisem a nemají stavový kód. Jedná se zejména o verzi, datum a čas překladu firmware Bootloaderu, informace o stisku tlačítka pro nahrání záložního firmware a předání řízení uživatelské aplikaci. Stavové kódy se dělí do několika skupin identifikovatelné dle první číslice a vzestupně označují závažnost stavu, viz tabulka č. 16.

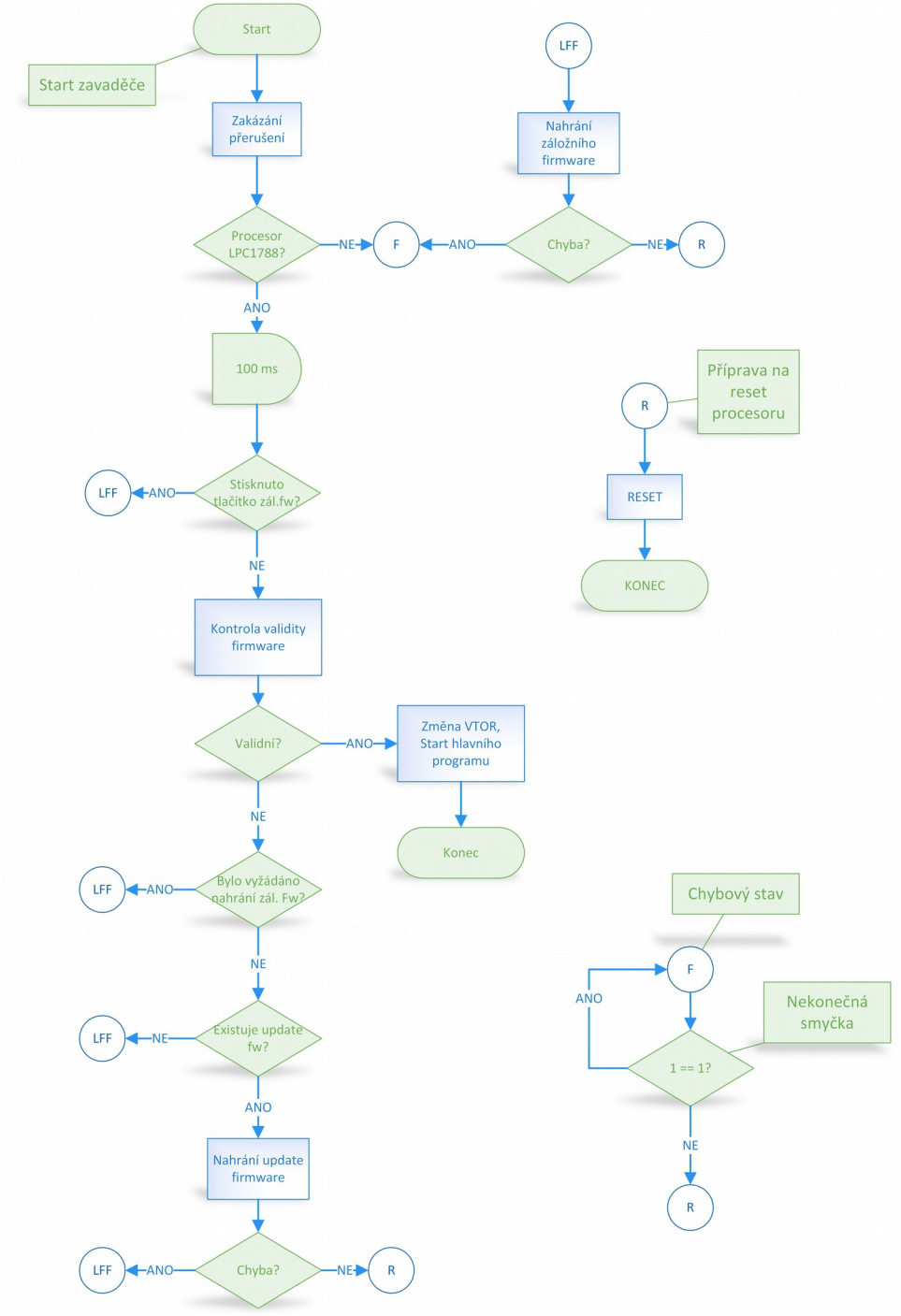
Hodnota	Závažnost	Popis chyby
1	Emergency	Systém je nepoužitelný, nelze pokračovat
2	Critical	Kritická chyba v systému, následuje stav Emergency
5	Warning	Varování, neobvyklá událost
7	Informational	Informační zpráva o stavu Bootloaderu
8	Debugging	Informace využitelné při ladění aplikace

Tabulka 16: Klasifikace závažnosti chyb

Číslice částečně odpovídají hodnotám používaných v syslogu s určitými modifikacemi, kdy hodnoty 3 a 4 byly vynechány z důvodu nejednoznačnosti. Vzhledem k velikosti textů, kdy jejich binární reprezentace zabírá značnou část paměti určenou pro samotný Bootloader, jsou v Release konfiguraci textové řetězce potlačeny, řízeno pomocí makra NDEBUG, a jsou zobrazovány pouze stavové kódy. V Debug konfiguraci se vypisuje stavový kód i textový řetězec popisující stav.

8.2.8 Vývojový diagram zavaděče

Struktura činnosti Bootloaderu je popsána vývojovým diagramem na obrázku č. 18. Z důvodu přehlednosti jsou zde uvedeny pouze základní procesy, které jsou vykonávány. Detaily jednotlivých operací byly popsány v předcházejících kapitolách.



Obrázek 18: Vývojový diagram zavaděče

8.3 Inicializace systému

Programové vybavení musí před skutečným předáním řízení uživatelské aplikaci provést inicializaci systému, která se skládá z přípravy prostředí jazyka C, definice vektorů přerušení, inicializace periférií, ovladačů, dynamické paměti a systémových služeb. Na rozdíl od zavaděče je kód pro inicializaci odlišný pro každý terminál a proto je součástí aplikace. V literatuře je označován názvem Board Support Package, zkratka BSP.

BSP je vztaženo vždy ke konkrétnímu terminálu, má tedy povědomí o konfiguraci desek a použití různých variant procesorového modulu. S využitím konceptu UPM, kde SPI paměť na základní desce obsahuje identifikaci zařízení lze snadno detekovat použití nesprávného BSP.

8.3.1 Příprava prostředí jazyka C

Standard jazyka C vyžaduje, aby neinicializované globální a lokální statické proměnné byly inicializované na hodnotu 0. V GNU GCC překladači jsou tyto proměnné ukládány do segmentu .bss. Pokud jsou proměnné inicializované, GCC je ukládá do segmentu .data, viz kapitola č. 6.2.2. BSP s využitím symbolů z linker skriptu nuluje oblast definovanou segmentem .bss a kopíruje data z interní Flash paměti do SRAM. Výše uvedené inicializace jsou prováděny v proceduře _start(). Po provedení operací je volána funkce main().

Pokud je vráceno řízení z funkce main(), jedná se o vážnou chybu systému jako celku, který může nabýt nedefinovaného stavu. Proto není možné znovu zavolat funkci main(), ale definovaným způsobem ukončit činnost systému. Volba ukončení závisí na konkrétní implementaci BSP. Obvykle je řešeno pomocí resetování procesoru, případně cyklení v nekončené smyčce a čekání na resetování od watchdogu. Pokud to systém umožňuje, uživatel by se měl o této chybě dozvědět, buďto výpisem na ladící konzoli, případně interním záznamem v nevolatilní paměti.

Součástí přípravy prostředí je také definice vektorů přerušení. Procesory s jádrem Cortex-M3 mají umístěné vektory přerušení od fyzické adresy uložené ve VTOR, kdy každá čtyř-bajtová hodnota ukazuje na počátek kódu příslušné obsluhy přerušení. Výjimku tvoří ofset 0x0, kde je uložena počáteční hodnota zásobníku. Tabulku vektorů přerušení tvoří pole ukazatelů na funkce s prototypem void (*pFunc)(void). Každý záznam odkazuje na příslušnou funkci zpracovávající přerušení. Funkce musí být definovány v příslušném HAL ovladači. Procesor LPC1788 navíc požaduje, aby se na offsetu 0x1C nacházel kontrolní součet předchozích vektorů [17]. V opačném případě je spuštěn vnitřní NXP Bootloader. V případě využití Bootloaderu není tato podmínka nutná, neboť tuto hodnotu definuje samotný Bootloader, ale je výhodné ji uvádět, zejména pokud se jedná o testování samotného BSP a zavaděč není nahrán do interní paměti mikrokontroléru. Koncepce UPM navíc vyžaduje, aby se na offsetech od bázové adresy aplikace, viz tabulka č. 7, vyskytovaly kontrolní součty a velikosti interního firmware.

8.3.2 Provázanost BSP s programovým vybavením

Programové vybavení využívá pro svou funkci několik pomocných knihoven, např. LWIP, FatFS, Grlib aj. Většina knihoven obsahuje konfigurační volby definovatelné při překladu, které určují počet a velikost prostředků alokovaných danou knihovnou. Nastavit univerzální konfiguraci, tak aby vyhovovala všem terminálům je téměř nemožné.

Programové vybavení tento problém pomocí konfiguračních souborů definovaných BSP. Při kompilaci systémových knihoven programového vybavení je potřeba změnit nastavení projektu a přidat odkaz na místo, kde se konfigurační soubory nacházejí.

Jedná se o tyto soubory

- ffconf.h

  Konfigurace souborového systému

- lwipopts.h

  Konfigurace TCP/IP stacku

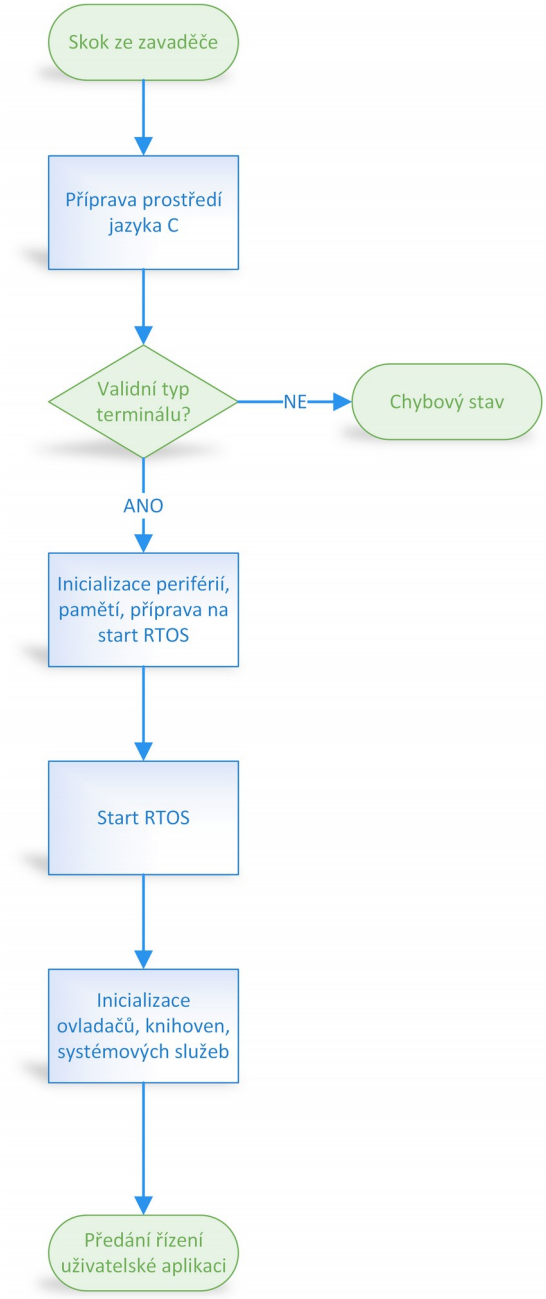
- system_config.h

  Konfigurace systému

Část systémových funkcí v programovém vybavení závisí na použitém hardwaru na desce, a proto nemohou být definovány přímo v knihovnách systému (alokace paměti, nastavení hodin, resetování periférií, aj.). Programové vybavení deklaruje tyto funkce, mající prefix BSP_, ve speciálním hlavičkovém souboru a každá funkce musí být v BSP definována, resp. musí obsahovat minimální implementaci vracející chybový kód. Uživatelská aplikace je od tohoto vnitřního volání odstíněná, neboť využívá tyto funkce přes standardní API.

8.3.3 Inicializační pořadí BSP

Paměťové rozložení aplikace (tj. včetně BSP) odpovídá požadavkům pro Cortex-M3 procesory, pouze posunuté o offset velikosti Bootloaderu. Po předání řízení Bootloaderem aplikaci se začne vykonávat kód, jehož začátek definuje vektor ležící na offsetu 0x4 od počátku adresního prostoru aplikace. Tento vektor ukazuje na obsluhu výjimky reset, funkci __lpc1788_isr_reset. Ta obsahuje skok na funkci _start(), která provede přípravu C prostředí a zavolá funkci main() vyskytující se v systémové knihovně. Ve funkci main() se zinicializují proměnné nutné pro samotný systém a následuje inicializace za pomoci BSP kódu ve dvou fázích. V první fázi je volána funkce BSP_init_PRE_OS(), kdy ještě není spuštěn operační systém. BSP musí minimálně nakonfigurovat dynamickou paměť pro potřeby operačního systému. Ve druhé fázi je volaná funkce BSP_init_POST_OS(), kdy operační systém je aktivní a plně funkční. V této fázi by měly být zavedeny ovladače a inicializována podpůrná funkcionalita, např. časové pásmo, standardní vstup/výstup. Poslední krokem inicializace systému je zavolání funkce Main(int,void*), vstupní bod do uživatelského prostoru. Vývojový diagram inicializace je znázorněn na obrázku č. 19.



Obrázek 19: Inicializace systému v rámci BSP

8.3.4 Jednotlivé kroky vykonávané v BSP

V této kapitole je prezentována šablona BSP pro terminál s procesorem LPC1788. Každé zařízení má své BSP, a proto zde uvedený příklad slouží pouze jako reference. Jednotlivé kroky jsou většinou navzájem nezávislé, existují ale výjimky, např. inicializace externí paměti. Tato činnost nemůže být zahájena dříve, než inicializace pinových multiplexerů.

8.3.4.1 Inicializace systému před spuštěním OS

Tato inicializace probíhá v BSP definované funkci BSP_init_PRE_OS(). V této fázi ještě není RTOS aktivní a není možné volat funkce, které RTOS poskytuje. Zároveň je garantována nepřerušitelnost vykonávání kódu a nehrozí race condition. Nelze zatím volat některé funkce ze standardní knihovny jazyka C, resp. ty, které závisí na existenci dynamické paměti.

- **Ochrana adresního prostoru Bootloaderu kompletním zakázáním přístupu vč. možnosti vykonávání instrukcí pomocí MPU**

V adresním prostoru Bootloaderu se nachází kritické funkce pro manipulaci s interní Flash pamětí mikrokontroléru. BSP by mělo zajistit, aby uživatelská aplikace nemohla přistoupit k těmto funkcím a chránit tak uložený systém před náhodným zničením dat. Výhodou je také možnost detekovat standardní chyby v C programech, kdy dochází k dereferenci nulového ukazatele, případně se přistupuje k adrese 0x00000000. Při provádění uživatelského firmware update je poté nutné vypnout ochranu a zavolat funkci Bootloaderu, která zajistí nahrání nového firmware. Bootloader umisťuje tuto funkci na adresu 0x3FC0, 64 bajtů pod horní hranici svého adresního prostoru. Vlastnost MPU umožňuje předefinovat přístupová práva, která překrývají již existující region. BSP tedy může povolit přístup a vykonávání kódu pro region začínající na adrese 0x3FC0 s velikostí 64 bajtů, aniž by musel odemykat celou oblast Bootloaderu.

- **Detekce a zjištění typu základnové desky z SPI Flash**

BSP před započetím inicializace pinových multiplexerů, driverů aj. musí mít garanci o správné verzi zařízení, pro které je určeno. V opačném případě by mohlo dojít k aktivaci pinů, které řídí akční členy, např. relé napojené na elektrický zámek. Zároveň jde o kontrolu nahrání korektní aplikace při provádění firmware update. BSP kontroluje parametry z identifikačního záznamu Základnové stanice, konkrétně device ID, major a minor verze. Pokud parametry neodpovídají očekávaným hodnotám, záleží na implementaci BSP, jak se zachová. Jedna z možností je zavolat funkci Bootloaderu, aby byl nahrán záložní firmware.

- **Detekce a zjištění typu dceřiných desek z I2C EEPROM, pokud je dané zařízení obsahuje**

Při detekci validní základnové desky se ověřuje přítomnost dceřiných desek. Průběh a ukončení tohoto kroku je obdobný jako u detekce a zjištění základnové desky.

- **Nastavení frekvence hodin pro CPU, Paměti, Periférie**

Procesor LPC1788 je po zapnutí napájení řízen hodinami z interního RC oscilátoru o frekvenci ~12 MHz. Pokud aplikace požaduje jiný zdroj hodin, odlišnou frekvenci, lze toto nastavení měnit pomocí PSP_clock_set() funkce.

- **Zapnutí napájení pro periférie uvnitř mikrokontroléru**

Ve výchozím stavu, resp. po zapnutí napájení, je většině periferiím vypnuto napájení z důvodu úspory energie. Ve vybraném mikrokontroléru je napájení řízeno pomocí PCONP

registru. BSP zná přesně, které periférie MCU budou využity a pouze tyto jsou zapnuty. Přínosem je snížení proudového odběru.

- **Inicializace pinových multiplexerů**

V procesoru LPC1788 může mít vstupně/výstupní pin několik funkcí a vlastností v závislosti na nastavení pinového multiplexeru. BSP zná rozmístění a funkci jednotlivých pinů a provede jejich inicializaci. Piny použité pro povinné součástky konceptu UPM musí zůstat ve stavu, ve kterém je zanechal Bootloader, případně mít stejné nastavení. Inicializace musí probíhat z bezpečnostních důvodů pouze po detekci správné Základnové, případně dceřiné desky.

- **Inicializace dynamické paměti pro RTOS, drivery**

Ještě před spuštěním OS je potřeba vytvořit prostor pro dynamické alokování prostředků pro RTOS a ovladače. Záleží čistě na BSP, jaké zvolí prostředky pro dynamickou alokaci. Programové vybavení při alokaci/uvolňování paměti volá funkce s prefixem BSP_mem_*(). V systému jsou rozlišovány tři druhy pamětí:

ON_CHIP_DRIVERS

: Paměť určená pro potřeby ovladačů, jsou zde převážně uloženy jejich workspace struktury. Paměť je obvykle alokovaná ve vnitřní SRAM paměti mikrokontroléru.

ON_CHIP_RTOS

: Paměť určená pro potřeby operačního systému, jsou zde převážně uloženy datové struktury pro primitiva (mutex, semafor) a paměť zásobníku jednotlivých úloh. Paměť je obvykle alokovaná ve vnitřní SRAM paměti mikrokontroléru.

SDRAM.

: Paměť určená pro potřeby aplikace, do této paměti jsou přesměrována volání typu malloc/calloc. Paměť je obvykle alokovaná v externí SDRAM paměti.

BSP může libovolně mapovat všechny typy paměti např. do jednoho adresního poolu nebo pokud není připojena externí paměť, přemapovat SDRAM do ON_CHIP_RTOS.

- **Inicializace a konfigurace externí paměti**

Inicializace externí paměti, např. NOR Flash a SDRAM, smí proběhnout pouze po inicializaci pinových multiplexerů. V této fázi se provádí konfigurace paměťového kontroléru, inicializace externí paměti pro využití funkcí malloc() je prováděna až po startu operačního systému.

- **Nastavení přístupu do adresního prostoru externích pamětí s využitím MPU**

Podobně jako oblast Bootloaderu, BSP může nastavit různá omezení pro adresní rozsahy externí paměti. Pro NOR Flash je vhodné povolit pouze čtení s možností vykonávání kódu. Pro SDRAM čtení/zápis se zákazem vykonávání kódu. V každém případě musí BSP dodržovat jednotnou politiku v oblasti správy paměti. Pokud nastaví prostor NOR Flash pouze

pro čtení, musí před být před případným nahráním update firmware v adresním prostoru opět povolen i zápis.

8.3.4.2 Inicializace systému po spuštění OS

V této fázi je mikrokontrolér připraven vykonávat základní operace s využitím operačního systému. Programové vybavení vytvoří v systému úlohu, která po své aktivaci zavolá funkci BSP_init_POST_OS(). Následně je zavolána rutina aktivující samotný operační systém.

- **Inicializace debug konzole**

Pokud to hardware zařízení umožňuje, BSP by mělo co nejdříve inicializovat debug konzoli, např. UART, do které bude zapisovat stavové informace o průběhu své činnosti. V krajním případě lze využít SWO funkcionalitu u mikrokontroléru. Nevýhoda tohoto řešení je nutnost připojení JTAG/SWD debuggeru k procesoru, namísto jednoduchého UART data záznamového zařízení.

- **Výpis konfigurace**

Tento výpis je čistě informačního charakteru pro případné pozdější rozlišení terminálu a jeho nastavení.

- **Inicializace dynamické paměti SDRAM**

Při alokaci dynamické paměti typu SDRAM a její umístění do adresního prostoru externí paměti, je třeba mít na zřeteli možné využití této paměti také pro jiné periférie, např. pro LCD, které zde umisťuje svoji videopaměť. Pokud je LCD použito, BSP musí alokovat dynamickou paměť od adresy za hranicí videopaměti. V rámci inicializace je vhodné provést rychlý test paměti.

- **Detekce a inkrementace čítačů druhů resetu**

Mikrokontrolér LPC1788 umožňuje detekovat, z jakého důvodu byl procesor resetován, např. výpadek napájení, aktivace watchdogu apod. Tyto informace mohou sloužit pro pozdější diagnostiku zařízení, je-li zvýšený počet resetů z důvodu aktivace watchdogu, může se jednat o chybu v aplikaci, která zablokuje celé zařízení.

- **Inicializace ovladačů pro detekci základnové a dceřiné desky**

Programové vybavení nabízí uživatelské API pro zjištění typu základnové a dceřiné desky. Informaci o typu terminálu může aplikace využít pro své interní potřeby, např. odlišný způsob zobrazení výstupních dat.

- **Registrace HAL ovladačů**

HAL (Hardware Abstraction Layer) ovladače komunikují přímo s registry periférií v mikrokontroléru. Registrací v programovém vybavení jsou alokovány nezbytné struktury pro práci, workspace, proběhne inicializace periférie a případně aktivace přerušení. Také se vytvoří záznam v tabulce prostředků, který je asociován s textovým jménem HAL ovladače. Toto jméno je využíváno jako index při získávání hledání workspace ovladače. Jméno musí být uloženo v read-only paměti, resp. nesmí dojít k jeho přepisu. HAL ovladače jsou na sobě

nezávislé a nevyužívají služeb operačního systému. Jmenný prostor HAL ovladačů je nezávislý na jmenném prostoru Device driver ovladačů.

- **Registrace Device Driver ovladačů**

Device drivery tvoří rozhraní mezi uživatelským API a HAL ovladači. Ovladač je spojen buďto s HAL ovladačem, např. HAL DMA a Device Driver DMA, nebo ovládá externí zařízení připojené k mikrokontroléru pomocí sběrnice, např. Device Driver AT24C04 EEPROM, který volá služby Device Driveru I2C, jenž je propojen s HAL I2C. Registrací v v programovém vybavení jsou alokovány nezbytné struktury pro práci, workspace, a proběhne inicializace zařízení, případně registrace callbacků obsluhující přerušení od periférie. Také se vytvoří záznam v tabulce prostředků, který je asociován s textovým jménem Device driver ovladače. Toto jméno je využíváno jako index při získávání hledání workspace ovladače. Jméno musí být uloženo v read-only paměti, resp. nesmí dojít k jeho přepisu. Device driver ovladače mohou být na sobě závislé a využívají služeb operačního systému. Jmenný prostor Device driver ovladačů je nezávislý na jmenném prostoru HAL ovladačů. Většina ovladačů vyžaduje konfigurační strukturu, která se předává při registraci. Její definice je uvedena v hlavičkovém souboru daného Device driveru.

- **Uvolnění prostředků registrovaných HAL ovladačů**

Při registraci Device Driver ovladačů a jimi využívaných HAL ovladačů se z důvodu rychlosti přístupu k funkcím ukládá HAL workspace ukazatel přímo do workspace Device Driver (DD) ovladače. Při následném volání funkcionality HAL ovladače není potřeba vyhledávat HAL workspace v tabulce handles. Pokud již nebude jméno HAL ovladače využíváno, lze uvolnit záznam v tabulce. Uvolněním záznamu nedojde ke zneplatnění HAL workspace, pouze ke ztrátě vztahu HAL jméno – HAL workspace, který je ale již uložen v příslušném ovladači.

- **Inicializace softwarových knihoven**

Poslední fází je inicializace softwarových knihoven, které jsou obsaženy v programovém vybavení. Výčet knihoven, které budou inicializovány závisí na typu terminálu. Pokud např. terminál obsahuje řádkový LCD, BSP nebude obsahovat inicializaci grafické knihovny GrLib.

9 Testování programového vybavení

Ověření bezchybné funkce programového vybavení je provedeno testováním kódu na dvou různých platformách nabízejících odlišné podpůrné možnosti ladění. Prvotní testování probíhá na osobním počítači s využitím simulátoru. V kontextu vývoje softwaru pro vestavěná zařízení jsou simulátory nástroje k testování zdrojového kódu bez použití cílového zařízení. Důvodů proč se používají tyto nástroje je několik. Aplikace se typicky vyvíjejí na stolním počítači a je rychlejší a pohodlnější spouštět vyvíjený kód přímo na něm, protože tím odpadá přenos výsledného programu do zařízení při každé sebemenší změně a následná manipulace s ním.

Simulátory také poskytují pokročilejší ladící a diagnostické funkce, jejichž implementace do reálného zařízení by byla především ve vestavěných systémech velmi časově a finančně náročná. Další výhodou těchto nástrojů je, že jejich implementace může být značně rychlejší než vývoj reálného hardwaru. V takových případech je pak možné vytvářet paralelně hardware a software zařízení a zkrátit tak celkový čas na jeho vývoj.

Simulátor může usnadnit a urychlit testování především v počáteční fázi vývoje, ale nikdy nemůže zcela nahradit testování na reálném zařízení. Výkon jednotlivých částí systému není nikdy možné nasimulovat přesně a stejně tomu je u chyb hardwaru nebo operačního systému. Proto je vždy nutné, minimálně na konci vývojového cyklu, provést testy na samotném zařízení. Proto je programové vybavení testováno v druhé fázi, přímo na mikrokontroléru.

9.1 Testování programového vybavení na osobním počítači

Na osobní počítač lze nahlížet jako nový typ procesoru a operačního systému. Rozvrstvením programového vybavení a definováním rozhraní mezi úrovněmi je možné pro portaci na novou platformu změnit pouze vrstvy HAL a OSAL, ostatní součásti programového vybavení zůstávají beze změny.

Při simulaci chování aplikace na osobním počítači se využívá vlastnosti Toolchainu GCC, která umožňuje generovat kód jak pro ARM architekturu, tak i pro PC. Veškeré parametry a speciální konstrukty jazyka mohou zůstat beze změny. Na osobním počítači, zejména pak pod operačním systémem GNU/Linux je možné využití existujících softwarových nástrojů, které nelze spustit přímo na mikrokontrolérech, např. Valgrind pro paměťovou analýzu, GCC profilace kódu, GDB ladící podpora aj.

Kód testovací aplikace a programového vybavení je zkompilován pro verzi spustitelnou na PC. OSAL vrstva je implementována s využitím knihovny pthread a HAL ovladače generují zprávy pro aplikaci Semulátor, rovněž spouštěnou na osobním počítači, která simuluje chování mikrokontroléru a externích zařízení. Struktura protokolu výměny mezi dat mezi HAL ovladači a aplikací Semulátor je znázorněna na výpisu č. 20. Struktura obsahuje atribut packed, informaci pro překladač, aby nevkládal výplňové bajty pro zarovnání přístupu, neboť testovaná aplikace a Semulátor mohou být spouštěny na odlišných počítačích.

```
struct SYSTEM_SIMULATOR_PROTOCOL
{

      // Preamble to synchronize packets.
      uint8_t preamble[PREAMBLE_SIZE];
//-------------- four bytes boundary-------------
      // Size of data in bytes (exclude this header)
      uint32_t size_of_data;
      //-------------- four bytes boundary-------------
      // A timestamp of message when it was created.
      uint32_t timestamp;
      //-------------- four bytes boundary-------------
      // Unique ID of a packet message
      uint16_t unique_id;

      // Peripheral ID, see PERIPHERAL_xxx
      uint8_t peripheral_id;

      // Physical number of device. E.g. serial 0, 1, 2,...
      uint8_t peripheral_physical;
      //-------------- four bytes boundary-------------
      // index to hal function, see a hal for every peripheral or
HAL_INDEX_INTERRUPT (with a combination of SOURCE_SIMULATOR)
      uint8_t index_to_hal;

      // Source of protocol message. See SOURCE_X define
      uint8_t protocol_source;

      // Message status - CONFIRMATION, REQUEST
      uint8_t message_status;

      // Crc of the packet header.
      uint8_t xor_crc_header;

      // Crc of the packet data.
      uint8_t xor_crc_data;

      //-------------- four bytes boundary-------------
      // Reserved for a further use.
      uint8_t reserved[3];
      //-------------- four bytes boundary-------------
      // Pointer to data section. Determined by size_of_data item.
      uint8_t data[];

}__attribute__ ((packed));
```

Výpis 20: Struktura protokolu mezi HAL ovladači a aplikací Semulátor

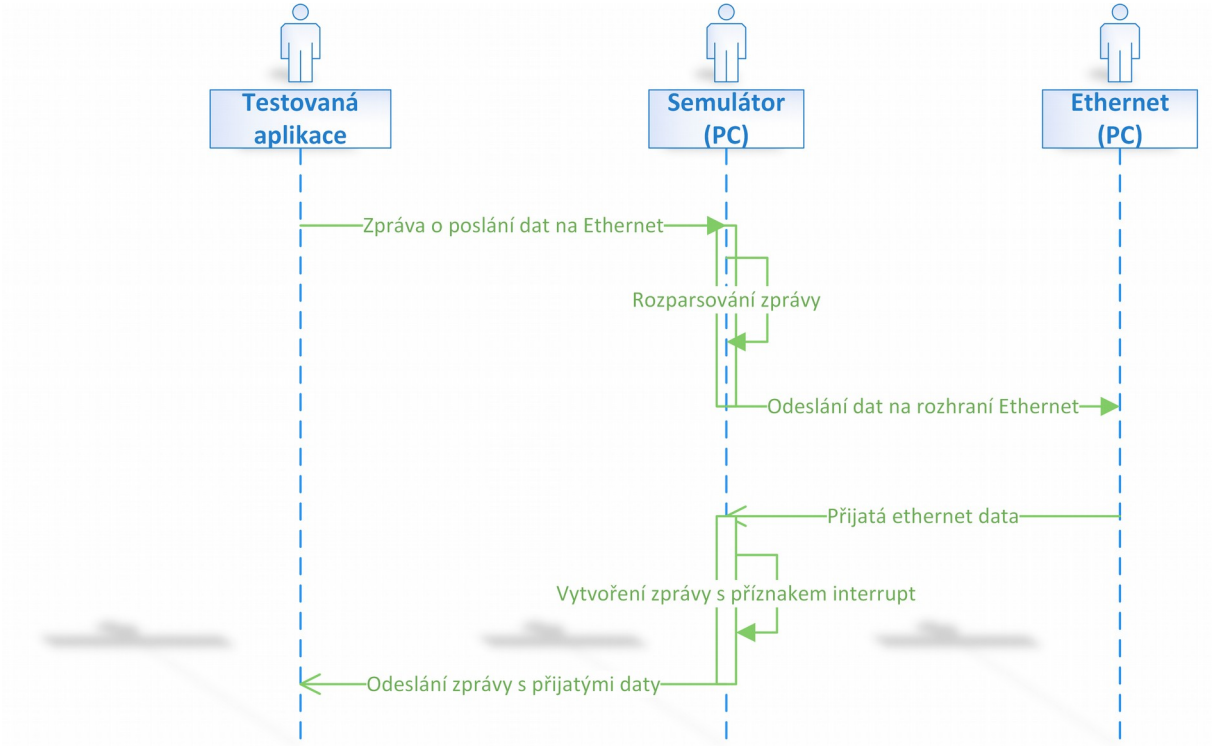
Hlavička protokolu začíná preambulí, kterou je určena pro zasynchronizování zprávy na straně přijímací aplikace. Následuje velikost datového bloku, který bude přenášen po hlavičce. Časová značka a identifikační číslo zprávy je určena převážně pro kontrolu duplicit u příjemce. Následuje identifikace perfiérie, která vygenerovala protokol (např. UART) spolu s číslem udávající její index (např. UART0 nebo UART1). Další položka představuje index funkce z HAL interface, aby mohl příjemce rozlišit druh funkce, která má být vykonána.

Označení odesílatele, testovaná aplikace nebo Semulátor a status zprávy, je převážně z důvodu snadného rozlišení zpráv při analýze zachyceného protokolu. Na závěr hlavičky jsou uvedeny kontrolní součty dat protokolu a dat. Použitý kontrolní mechanismus je velmi jednoduchý, operace xor s bajty zprávy, a je spoléháno na sofistikovanější způsoby ochrany u přenosového kanálu.

Zprávy posílané z testované aplikace do Semulátoru a naopak tvoří vyplněná serializovaná hlavička plus příslušná data, pokud jsou využita. Serializace je prováděna kopírováním proměné z paměti, proto je důležité, aby testovaná aplikace a Semulátor byly spouštěny na architektuře mající stejnou endianitu. Na přenos zpráv mezi HAL ovladači a Semulátorem lze využít UDP síťové pakety. Komunikační kanál lze volit libovolně, nicméně je důležité, aby byla podporována duplexita, asynchronost a zajištěna nízká latence doručování zpráv.

Na schématu zobrazeného na obrázku č. 20 je znázorněna komunikace testované aplikace a Semulátoru při využívání rozhraní Ethernet. V testované aplikaci je knihovnou LWIP vygenerován TCP/IP paket zabalený v Ethernet rámci. V adaptační vrstvě je pomocí periferního API zavolána funkce Ethernet ovladače pro odeslání Ethernet rámce. Ovladač zpracuje rámec a volá funkce Ethernet HAL ovladače deklarovaných v HAL interface. HAL ovladač vygeneruje a vyplní hlavičku zprávy, přidá data a odešle aplikaci Semulátor, která rozparsuje zprávu a na základě informací z hlavičky, zpráva z periferie Ethernet, HAL interface index, ji pošle na lokální Ethernet v rámci PC.

Při přijmu zprávy z PC Ethernetu je Semulátorem vygenerována zpráva, vyplněna hlavička, přidána data z rámce a poslána testované aplikaci. Na straně testovací aplikace je provedeno rozparsování zprávy a zavolána rutina přerušení pro periferii Ethernet. HAL ovladač zpracuje přerušení a informuje ovladač Ethernetu o nové zprávě, která je propagována až do knihovny LWIP, která data z rámce zpracuje.



Obrázek 20: Sekvenční diagram testované aplikace a Semulátoru

Postup zpracování dat v ovladači, periférním API a knihovna je stejný, jako kdyby byl použita periferie Ethernet na mikrokontroléru LPC1788.

9.2 Testování programového vybavení na mikrokontroléru

Testování programového vybavení spouštěného na mikrokontroléru je prováděno metodou bottom-up, tedy od nejnižší vrstvy abstrakce k vyšší. Postupným testováním jednotlivých programových komponent dochází u vyšších vrstev také k testování jejich spolupráce s nižšími v součinnosti s operačním systémem. Pro každou periferií je vytvořena sada tří druhu testů, která ověřuje její korektní činnost. Nejdříve je zkoušen HAL ovladač a volají se všechny funkce poskytované HAL interface. Následně je obdobným způsobem testován ovladač a periferní API. U ovladačů se testuje kompletní řetězec, tedy jeho registrace a inicializace workspace.

Dalším stupněm testování je spuštění testovací aplikace, stejné jako v případě testování na simulátoru a porovnáním odlišných výsledků lokalizovat příčinu chyby.

10 Závěr

Tato práce se zabývala návrhem nízkoúrovňové programového vybavení pro grafický docházkový terminál, provozovaný na mikrořadiči s jádrem ARM a využitím open-source vývojových nástrojů. Nejdříve bylo nutné stanovit koncepci a požadavky na funkci programového vybavení, aby bylo možné vybrat konkrétní typ mikrokontroléru, vývojových nástrojů. Na základě rozboru řešení byl vybrán mikrokontrolér LPC1788 s jádrem Cortex-M3 od společnosti NXP. Jelikož požadavky programového vybavení, zejména z důvodu použití obrazové paměti a ne-volatilního datového úložiště, převyšovaly dostupné fyzické možnosti mikrokontroléru, musely být připojeny externí paměti SDRAM a NOR Flash. Pro vývoj software byl vybrán překladač GCC a integrované prostředí Eclipse, ladění kódu pomocí komerčního JTAG/SWD Debugeru Segger s návazností na OSS GDB.

Z požadavků programového vybavení vyplynula modifikace a nastavení open-source vývojových nástrojů. Pro překladač byl vytvořen linker skript podporující uložení programového kódu i v externí paměti a standardní knihovna jazyka C musela být znova zkompilována, aby byly sníženy paměťové požadavky, obsahovala podporu pro víceúlohové prostředí a byly provedeny opravy chyb způsobující úniky paměti (memory leaks). Do editoru Eclipse byl přidán plugin podpory vývoje vestavěných zařízení.

Při realizaci programového vybavení byl využit úrovňový model se vzestupnou vertikální abstrakcí. Jednotlivé úrovně spolu komunikují na vertikální a horizontální úrovni pomocí volání rozhraní, které poskytuje každá vrstva. Nejnižší úroveň, HAL – hardware abstraction layer, tvoří ovladače periférií mikrokontroléru. Ovladače na této úrovni modifikují registry mikrokontroléru a nevyužívají služeb operačního systému. Přístup k nejnižší úrovni je prováděn pomocí HAL interface, který je využíván vyšší úrovní – Ovladače, které obsahují implementaci ovládání periferií z abstraktního pohledu a zjemňují komplexní činnosti na menší kroky. Na stejné úrovni se také nachází implementace RTOS. Pro využívání jeho služeb existuje vrstva OSAL, Operating System Abstraction Layer, která umožňuje jeho záměnu bez nutnosti změny ostatního kódu a dodefinování funkcionalit stanovených programovým vybavením. Nejvyšší úroveň tvoří rozhraní, které může využívat aplikační programátor. Zde se nachází periferní API, OSAL a knihovní funkce. Pro potřeby programového vybavení byly vytvořeny systémové knihovny, zahrnující souborový a paměťový subsystém.

Součástí programového vybavení je také koncept hardware docházkového terminálu, který stanovuje minimální hardwarovou konfiguraci terminálu, struktury a fixní paměťová adresace rozložení programového kódu a dat. Tento koncept umožňuje provádět bezpečnou aktualizaci firmware i při výpadku napájení. Zároveň stanovuje způsob inicializace zařízení, od zavaděče (Bootloader) až po Board Support Package (BSP), software pro přípravu prostředí programového vybavení.

Díky úrovňovému modelu mohlo být programové vybavení testováno na osobním počítači, stačilo pouze nahradit vrstvy HAL a OSAL implementací pro PC. Ostatní kód zůstal beze změny. Pro simulování periferií mikrokontroléru a externích čipů byla vytvořena

aplikace Semulátor, která komunikuje s HAL ovladači přes protokol zpráv. Testováním na PC a pod OS GNU/Linux se otevřely možnosti využít pokročilé ladící nástroje typu Valgrind nebo GCC profilace kódu. Pro testování na mikrokontroléru byly vytvořeny sady testů periferií zahrnující každou úroveň a porovnávaly se výsledky se simulací.

Všechny stanovené cíle se podařilo splnit. Výsledky práce umožnily zavést ve společnosti IMA, s.r.o. nové způsoby týmové spolupráce při vývoji aplikací pro docházkové terminály, unifikaci vývojového prostředí, rozdělení práce mezi embedded a aplikační vývojáře a použití programovacího jazyka C namísto assembleru. Obecný návrh programového vybavení umožnil jeho použití nejen v docházkových terminálech, ale také v přístupových systémech a výdejních automatech.

11 Literatura

[1] VÁLA, Petr. *Docházkový systém pro malé a středně velké firmy*. 2006. Bakalářská práce. ČVUT - FEL Katedra počítačů.

[2] *Intel 8052*. Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2016-01-04]. Dostupné z: https://en.wikipedia.org/wiki/Intel_MCS-51

[3] GUBBI, Jayavardhana, et al. Internet of Things (IoT): *A vision, architectural elements, and future directions*. Future Generation Computer Systems, 2013, 29.7: 1645-1660.

[4] Rozhraní Wiegand. DH Servis [online]. [cit. 2016-01-05]. Dostupné z: http://www.dhservis.cz/dalsi_1/wiegand.htm

[5] *Grafický element widget*. Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2016-01-06]. Dostupné z: https://en.wikipedia.org/wiki/Widget_(GUI)

[6] MILLS, D., et al., *Network Time Protocol Version 4*, IETF, RFC 5905, June 2010.

[7] *SD Part 1, Physical Layer Simplified Specification*, Version 4.10 [online]. SD Association [cit. 2016-01-06]. Dostupné z: https://www.sdcard.org/downloads/pls/part1_410.pdf

[8] *Response Times: The 3 Important Limits* [online]. NIELSEN, Jakob. 1993 [cit. 2016-01-06]. Dostupné z: https://www.nngroup.com/articles/response-times-3-important-limits/

[9] SOJKA, Michal. *Real-Time operating systems overview : Přednášky z předmětu Real-Time systems programming* [online]. 2014 [cit. 2016-01-06]. Dostupné z: https://support.dce.felk.cvut.cz/psr/prednasky/other-os-en.pdf

[10] YIU, Joseph. *The definitive guide to ARM® Cortex®-M3 and Cortex-M4 processors*. Third edition. Amsterdam: Elsevier, Newnes, 2014, xxxv, 818 pages. ISBN 01-240-8082-0.

[11] SLOSS, Andrew N, Dominic SYMES a Chris WRIGHT. *ARM system developer's guide: designing and optimizing system software*. Amsterdam: Elsevier Morgan Kaufmann, 2004, xiii, 689 s. ISBN 15-586-0874-5.

[12] OSHANA, Robert a Mark KRAELING. Software engineering for embedded systems: methods, practical techniques, and applications. Maltham, MA: Newnes, 2013, xlix, 1150 pages. ISBN 01-241-5917-6.

[13] *Linux kernel*. Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2016-01-06]. Dostupné z: https://en.wikipedia.org/wiki/Linux_kernel

[14] *NAND vs. NOR Flash Memory* [online]. Toshiba America Electronic Components, inc., 2006 [cit. 2016-01-06]. Dostupné z: http://umcs.maine.edu/~cmeadow/courses/cos335/Toshiba NAND_vs_NOR_Flash_Memory_Technology_Overviewt.pdf

[15] CHAMBERLAIN, Steve. *The GNU linker* [online]. 1999 [cit. 2016-01-06]. Dostupné z: http://www.scoberlin.de/content/media/http/informatik/gcc_docs/ld_toc.html

[16] KHUSAINOV, Vladimir. *Practical Advice on Running uClinux on Cortex-M3/M4* [online]. In: . 2012 [cit. 2016-01-06]. Dostupné z: http://electronicdesign.com/embedded/practical-advice-running-uclinux-cortex-m3m4

[17] *LPC178x/7x User manual: UM10470* [online]. NXP Semiconductors, 2014 [cit. 2016-01-06]. Dostupné z: www.nxp.com/documents/user_manual/UM10470.pdf

12 Obsah přiloženého CD

- Adresář \Text\Pdf - Text diplomové práce ve formátu PDF

13 Příloha A – Linker skript

Linker skript s podporou nahrávání kódu do externí programové paměti

```
OUTPUT_FORMAT ("elf32-littlearm", "elf32-bigarm", "elf32-littlearm")
SEARCH_DIR(.)

/* Library configurations */

MEMORY
{
 FLASH (rx) : ORIGIN = 0x4000, LENGTH = 496K /* The lpc1788 has 512KB but
the first 16KB is used for bootloader*/
 RAM (rwx) : ORIGIN = 0x10000000, LENGTH = 63K /* The lpc1788 has 64KB but
the last 1KB is used for interrupts*/
 peripheral_ram (rw) : ORIGIN = 0x20000000, LENGTH = 32K
 EXT_FLASH (rx) : ORIGIN = 0x80000000, LENGTH = 16M
}

/*
 * The entry point is informative, for debuggers and simulators,
 * since the Cortex-M vector points to it anyway.
 */
ENTRY(__lpc1788_isr_reset)

PROVIDE(__stack = ORIGIN(RAM) + LENGTH(RAM) + 1024); /* The last 1KB is
used for interrupts*/
__stack_checksum = __stack;

/* Calculate the checksum for nxp bootloader. The "6" is because symbols
are even and should be odd. The thumb mode.*/
PROVIDE(__lpc17xx_checksum = -(ABSOLUTE(__stack_checksum) +
ABSOLUTE(__lpc1788_isr_reset) + ABSOLUTE(__lpc1788_isr_nmi) +
ABSOLUTE(__lpc1788_isr_hard_fault) + ABSOLUTE(__lpc1788_isr_mpu_fault) +
ABSOLUTE(__lpc1788_isr_bus_fault) + ABSOLUTE(__lpc1788_isr_usage_fault) +
6));


/* Used for pre init check */
_region_start_rom = ORIGIN(FLASH);
_region_start_ram = ORIGIN(RAM);
_region_end_ram = ORIGIN(RAM) + LENGTH(RAM);

SECTIONS
{
 /*
 * For Cortex-M devices, the beginning of the startup code is stored in
 * the .isr_vector section, which goes to FLASH.
 */
 .isr_vector : ALIGN(4)
 {
      CREATE_OBJECT_SYMBOLS
 FILL(0xFF)
```

```
__vectors_start__ = ABSOLUTE(.) ;
KEEP(*(.isr_vector))   /* Interrupt vectors */
/* Make sure we pulled in an interrupt vector. */
     ASSERT (. != __vectors_start__, "No interrupt vectors");


      . = ALIGN(4);
      KEEP(*(.isr_exception))
} >FLASH

.inits : ALIGN(4)
{
     CREATE_OBJECT_SYMBOLS

     PROVIDE(__INIT_SECTION_START__ = .);
/*
* These are the old initialisation sections, intended to contain
* naked code, with the prologue/epilogue added by crti.o/crtn.o
* when linking with startup files. The standalone startup code
* currently does not run these, better use the init arrays below.
*/
           KEEP(*(.init))
           KEEP(*(.fini))

           . = ALIGN(4);

           /*
* The preinit code, i.e. an array of pointers to initialisation
* functions to be performed before constructors.
*/
           PROVIDE_HIDDEN (__preinit_array_start = .);

/*
* Used to run the SystemInit() before anything else.
*/
           KEEP(*(.preinit_array_sysinit .preinit_array_sysinit.*))

/*
* Used for other platform inits.
*/
           KEEP(*(.preinit_array_platform .preinit_array_platform.*))

/*
* The application inits. If you need to enforce some order in
* execution, create new sections, as before.
*/
           KEEP(*(.preinit_array .preinit_array.*))

           PROVIDE_HIDDEN (__preinit_array_end = .);

           . = ALIGN(4);

           /*
* The init code, i.e. an array of pointers to static constructors.
*/
           PROVIDE_HIDDEN (__init_array_start = .);
           KEEP(*(SORT(.init_array.*)))
```

```
            KEEP(*(.init_array))
            PROVIDE_HIDDEN (__init_array_end = .);

            . = ALIGN(4);

            /*
 * The fini code, i.e. an array of pointers to static destructors.
 */
            PROVIDE_HIDDEN (__fini_array_start = .);
            KEEP(*(SORT(.fini_array.*)))
            KEEP(*(.fini_array))
            PROVIDE_HIDDEN (__fini_array_end = .);

            /*
 * The constructors and destructors.
 */
            . = ALIGN(0x4);
            KEEP (*crtbegin.o(.ctors))
            KEEP (*(EXCLUDE_FILE (*crtend.o) .ctors))
            . = ALIGN(0x4);
            KEEP (*crtbegin.o(.dtors))
            KEEP (*(EXCLUDE_FILE (*crtend.o) .dtors))
            KEEP (*(SORT(.dtors.*)))
            KEEP (*crtend.o(.dtors))

            PROVIDE(__INIT_SECTION_END__ = .);
            ASSERT((__INIT_SECTION_START__ == __INIT_SECTION_END__), "This
system does not support constructor/destructors, init/fini functions or
semihosting");

 } >FLASH

      .text :
      {
            CREATE_OBJECT_SYMBOLS
            _ftext = .;

            *(EXCLUDE_FILE (*libUPM_02_Library.a:) .text EXCLUDE_FILE
(*libUPM_02_Library.a:) .text.* EXCLUDE_FILE (*libUPM_02_Library.a:)
.gnu.linkonce.t.*)
            *(.plt)
            *(.gnu.warning)
            *(.glue_7t) *(.glue_7) *(.vfp11_veneer)
            *(.ARM.extab* .gnu.linkonce.armextab.*)
            *(.gcc_except_table)

      } > FLASH

      .eh_frame : ALIGN (4)
      {
       *(.eh_frame*)
       *(.eh_frame_entry .eh_frame_entry.*)
      } >FLASH

      . = ALIGN(4);

      .ARM.extab :
```

```
      {
            *(.ARM.extab* .gnu.linkonce.armextab.*)
      } > FLASH

      . = ALIGN(4);

      __exidx_start = .;
      .ARM.exidx :
      {
            *(.ARM.exidx* .gnu.linkonce.armexidx.*)
      } > FLASH
      __exidx_end = .;

      __etext = .;

      .rodata : ALIGN (4)
      {
       /* *(EXCLUDE_FILE (*libUPM_02_Library.a:) .rodata EXCLUDE_FILE
(*libUPM_02_Library.a:) .rodata.* EXCLUDE_FILE (*libUPM_02_Library.a:)
.gnu.linkonce.r.*)*/
       *(EXCLUDE_FILE (*libUPM_02_Library.a:) .rodata.* EXCLUDE_FILE
(*libUPM_02_Library.a:) .gnu.linkonce.r.*)
       . = ALIGN(4);
      } > FLASH

 /*
 * The initialised data section.
 *
 * The program executes knowing that the data is in the RAM
 * but the loader puts the initial values in the FLASH (inidata).
 * It is one task of the startup to copy the initial values from
 * FLASH to RAM.
 */
 /*
 * This address is used by the startup code to
 * initialise the .data section.
 */
 _sidata = LOADADDR(.data);

 .data : ALIGN(16)/* Align to 16 for encryption */
 {
      FILL(0xFF)
 /* This is used by the startup code to initialise the .data section */
 __data_start__ = . ;
            *(.data_begin .data_begin.*)

            *(.data .data.* .gnu.linkonce.d.*)

            PROVIDE(__JCR_SECTION_START__ = .);
            KEEP(*(.jcr))
            PROVIDE(__JCR_SECTION_END__ = .);
            ASSERT((__JCR_SECTION_START__ == __JCR_SECTION_END__), "This
system does not support JRC");

            *(.data_end .data_end.*)
       . = ALIGN(16); /* Align to 16 for encryption */
```

```
      /* This is used by the startup code to initialise the .data section
*/
 __data_end__ = . ;

 } >RAM AT>FLASH

 /* The primary uninitialised data section. */
 .bss (NOLOAD) : ALIGN(4)
 {
 __bss_start__ = .;
 *(.bss_begin .bss_begin.*)

 *(.bss .bss.*)
 *(COMMON)

 *(.bss_end .bss_end.*)
       . = ALIGN(4);
 __bss_end__ = .;
 } >RAM AT>FLASH

 .noinit (NOLOAD) : ALIGN(4)
 {
      __noinit_start__ = .;   /* standard newlib definition */
 *(.noinit .noinit.*)
 . = ALIGN(4);
 __noinit_end__ = .;
 } > RAM

      __end__ = .;
      end = __end__;

      .lpc1788.peripheral_ram (NOLOAD) : ALIGN (8)
      {
            __lpc1788_peripheral_ram_start__ = .;
      *(.peripheral_ram .peripheral_ram.*)
      __lpc1788_peripheral_ram_end__ = .;
      } >peripheral_ram

      .ext_flash.text : ALIGN (4)
      {
       __upm02_ext_flash_text_start__ = .;
       *libUPM_02_Library.a:(.text .text.* .gnu.linkonce.t.*)
       *(.ext_flash .ext_flash.*)
       . = ALIGN (4);
       __upm02_ext_flash_text_start_end__ = .;
      } >EXT_FLASH

      .ext_flash.rodata : ALIGN (4)
      { libUPM_02_Library.a
            __upm02_ext_flash_rodata_start__ = .;
            *(.rodata)
       *libUPM_02_Library.a:(.rodata .rodata.* .gnu.linkonce.r.*)
       . = ALIGN(16); /* Align to 16 for encryption */
       __upm02_ext_flash_rodata_end__ = .;
      } >EXT_FLASH

 /* Stabs debugging sections. */
```

```
 .stab 0 (NOLOAD): { *(.stab) }
 .stabstr 0 (NOLOAD): { *(.stabstr) }
 .stab.excl 0 : { *(.stab.excl) }
 .stab.exclstr 0 : { *(.stab.exclstr) }
 .stab.index 0 : { *(.stab.index) }
 .stab.indexstr 0 : { *(.stab.indexstr) }
 .comment 0 : { *(.comment) }

 /* DWARF debug sections.
 * Symbols in the DWARF debugging sections are relative to
 * the beginning of the section so we begin them at 0.
 */
 /* DWARF 1 */
 .debug 0 : { *(.debug) }
 .line 0 : { *(.line) }
 /* GNU DWARF 1 extensions */
 .debug_srcinfo 0 : { *(.debug_srcinfo) }
 .debug_sfnames 0 : { *(.debug_sfnames) }
 /* DWARF 1.1 and DWARF 2 */
 .debug_aranges 0 : { *(.debug_aranges) }
 .debug_pubnames 0 : { *(.debug_pubnames) }
 /* DWARF 2 */
 .debug_info 0 : { *(.debug_info .gnu.linkonce.wi.*) }
 .debug_abbrev 0 : { *(.debug_abbrev) }
 .debug_line 0 : { *(.debug_line) }
 .debug_frame 0 : { *(.debug_frame) }
 .debug_str 0 : { *(.debug_str) }
 .debug_loc 0 : { *(.debug_loc) }
 .debug_macinfo 0 : { *(.debug_macinfo) }
 /* DWARF 2.1 */
 .debug_ranges 0 : { *(.debug_ranges) }
 /* SGI/MIPS DWARF 2 extensions */
 .debug_weaknames 0 : { *(.debug_weaknames) }
 .debug_funcnames 0 : { *(.debug_funcnames) }
 .debug_typenames 0 : { *(.debug_typenames) }
 .debug_varnames 0 : { *(.debug_varnames) }

 .note.gnu.arm.ident 0 : { KEEP (*(.note.gnu.arm.ident)) }
 .ARM.attributes 0 : { KEEP (*(.ARM.attributes)) }
 /DISCARD/ : { *(.note.GNU-stack) }
}
```

14 Příloha B – Generování ROMFS struktur

Zdrojový kód generátoru datových struktur pro ROMFS. Kód je kompilovatelný pod GNU/Linux OS pomocí překladače GCC.

Formát příkazu: jméno_programu ZDROJOVÝ_ADRESÁŘ CÍLOVÝ_ADRESÁŘ,

ZDROJOVÝ_ADRESÁŘ – umístění souborů, které budou zahrnuty do ROMFS

CÍLOVÝ_ADRESÁŘ – adresář pro umístění vygenerovaných struktur

```
#define _GNU_SOURCE
#include <stdint.h>
#include <sys/types.h> /* Type definitions used by many programs */
#include <stdio.h> /* Standard I/O functions */
#include <stdlib.h> /* Prototypes of commonly used library functions,
 plus EXIT_SUCCESS and EXIT_FAILURE constants */
#include <unistd.h> /* Prototypes for many system calls */
#include <errno.h> /* Declares errno and defines error constants */
#include <string.h> /* Commonly used string-handling functions */
#include <dirent.h>
#include <ftw.h>

#define NEW_LINE_AFTER_N_DAT (16)
#define MAX_FILENAME_PATH (64)
#define ROM_FILE_TABLE_FILE "romfiletable.h"
#define INITIAL_TABLE_RECORDS (128)


static char * src_dirpath;
static char * dst_dirpath;
static FILE * file_table;

/*
 * The table records will be lexicographically ordered
 *
 */
typedef struct
{
      char name[MAX_FILENAME_PATH];
      char content[MAX_FILENAME_PATH];
      char size[MAX_FILENAME_PATH];
}ROM_FILE_TABLE;

/*
 * Rom FS file table
 * Will be allocated dynamically
 */
ROM_FILE_TABLE * rom_file_table;
size_t rom_file_table_size = 0;
size_t rom_file_records = 0;


static int filename_lexico_compare(const void * a, const void * b)
```

```
{
	const ROM_FILE_TABLE * aa =   (const ROM_FILE_TABLE *) a;
	const ROM_FILE_TABLE * bb =   (const ROM_FILE_TABLE *) b;

	return strcmp(aa->name, bb->name);
}

/*
 * Generate the include list
 */
static void generate_rom_file_include_table(const char * filepath, const
char * commentary, FILE * output_file)
{
	fprintf(output_file, "#include \"%s\" \t\t// %s\n", filepath,
commentary);
}

/*
 * Generate the file table item
 */
void generate_rom_file_table(const char * filepath, const char * c_data,
FILE * output_file)
{
	rom_file_table_size = INITIAL_TABLE_RECORDS;
	strncpy(rom_file_table[rom_file_records].name,
filepath,MAX_FILENAME_PATH);
	strncpy(rom_file_table[rom_file_records].content,
c_data,MAX_FILENAME_PATH);
	snprintf(rom_file_table[rom_file_records].size,MAX_FILENAME_PATH,
"sizeof(%s)", c_data);
	rom_file_records++;
	//fprintf(output_file, "{\"%s\", %s, sizeof(%s)},\n", filepath,
c_data, c_data);
}

/*
 * Generate the file content as C array
 */
void generate_file_content(const char * filepath, const char *
c_data_name, const uint8_t * data, size_t len, FILE * output_file)
{

	// put header
	fprintf(output_file, "/* This content was generated from file
%s */ \n\n", filepath);
	fprintf(output_file, "static const uint8_t %s[] = {\n",
c_data_name);

	// put data.
	for (size_t i = 0; i < len; i++)
	{
		fprintf(output_file, "0x%02x, ", data[i]);

		// generate new line.
		if (0 == (i % NEW_LINE_AFTER_N_DAT) && (i > 0))
		{
			fprintf(output_file, "\n");
```

```
            }

      }
      // put footer
      fprintf(output_file, "};\n");
}

/* Function called by nftw() */
static int generate_list_files(const char *pathname, const struct stat
*sbuf, int type, struct FTW *ftwb)
{
      char filename[MAX_FILENAME_PATH];
      switch (type)
      {
      /* Print file type */
      case FTW_F:
            snprintf(filename, sizeof(filename) / sizeof(filename[0]),
"%s.%ld.h", &pathname[ftwb->base], (long) sbuf->st_ino);
            generate_rom_file_include_table(filename, pathname,
file_table);
            break;
      default:
            break;
      }

      return 0; /* Tell nftw() to continue */
}

/* Function called by nftw() */
static int generate_rom_files(const char *pathname, const struct stat
*sbuf, int type, struct FTW *ftwb)
{
      char filename[128];
      char c_filename[MAX_FILENAME_PATH];
      char * pch;
      switch (type)
      {
      case FTW_F: // Regular file
      {
            snprintf(filename, sizeof(filename) / sizeof(filename[0]),
"%s/%s.%ld.h", dst_dirpath, &pathname[ftwb->base], (long) sbuf->st_ino);
            // get c name of array
            snprintf(c_filename, sizeof(c_filename) /
sizeof(c_filename[0]), "%s_%ld_h", &pathname[ftwb->base], (long) sbuf-
>st_ino);
            // replace all dot with underscore
            pch = c_filename;
            while (NULL != (pch = strchr(c_filename,'.')))
            {
             // replace with _
                  *pch = '_';
            }
            /*
             * Variables used in this CASE.
             */
            FILE * generated_header_file = NULL;
            FILE * input_file = NULL;
```

```
            uint8_t * input_file_content = NULL;

            // create the output file.
            generated_header_file = fopen(filename, "wb");
            if (NULL == generated_header_file)
            {
                  fprintf(stderr, "Failed to open %s:%s ", filename,
strerror(errno));
            }
            // open input file.
            input_file = fopen(pathname, "rb");
            if (NULL == generated_header_file)
            {
                  fprintf(stderr, "Failed to open %s:%s ", pathname,
strerror(errno));
            }

            if (NULL != generated_header_file && (NULL != input_file))
            {

                  // allocate space for input file content
                  input_file_content = malloc(sbuf->st_size);
                  if (NULL != input_file_content)
                  {

                        // read the content
                        size_t read_bytes = fread(input_file_content,
sizeof(uint8_t), (size_t) sbuf->st_size, input_file);
                        if (read_bytes == (size_t) sbuf->st_size)
                        {
                              // Generate file content as C array
                              generate_file_content(pathname, c_filename,
input_file_content, (size_t) sbuf->st_size, generated_header_file);
                              // Generate record in the table.

      generate_rom_file_table(&pathname[strlen(src_dirpath) + 1],
c_filename, file_table);
                        }
                        else
                        {
                              fprintf(stderr, "Failed to read %lu bytes
from %s\n", (size_t) sbuf->st_size, pathname);
                        }
                  }
                  else
                  {
                        fprintf(stderr, "malloc: Failed to allocate
%lu\n", (size_t) sbuf->st_size);
                  }
            }

            // Deallocate resources.
            if (NULL != generated_header_file)
            {
                  fclose(generated_header_file);
            }
            if (NULL != generated_header_file)
```

```
            {
                  fclose(input_file);
            }
            free(input_file_content);

            break;
      }
      default:
            break;
      }

      return 0; /* Tell nftw() to continue */
}


// Main entry point
int main(int argc, char *argv[])
{
      int return_error = EXIT_SUCCESS;
      int flags;
      char filename[128];
      DIR * directory;

      if (argc != 3)
      {
            fprintf(stderr, "Invalid usage: %s SRC_DIR DST_DIR\n",
argv[0]);
            exit(EXIT_FAILURE);
      }

      src_dirpath = strdup(argv[1]);
      dst_dirpath = strdup(argv[2]);

      /*
       * Check if source and destination dirs exist
       */
      directory = opendir(src_dirpath);
      if (NULL == directory)
      {
            perror("Opening source directory");
            return_error = EXIT_FAILURE;
            goto main_error_handler;
      }
      closedir(directory);
      directory = opendir(dst_dirpath);
      if (NULL == directory)
      {
            perror("Opening destination directory");
            return_error = EXIT_FAILURE;
            goto main_error_handler;
      }
      closedir(directory);

      // Allocate table
      rom_file_table = (ROM_FILE_TABLE *)malloc(INITIAL_TABLE_RECORDS *
sizeof(ROM_FILE_TABLE));
      if (NULL == rom_file_table)
```

```
      {
            perror("Creating file table");
            return_error = EXIT_FAILURE;
            goto main_error_handler;
      }
      // Init to null
      rom_file_records = 0;
      rom_file_table_size = INITIAL_TABLE_RECORDS;
      memset(rom_file_table, 0, INITIAL_TABLE_RECORDS *
sizeof(ROM_FILE_TABLE));

      /*
       * Remove last '/' from directory name to create proper name in
generated files
       */
      if ('/' == src_dirpath[strlen(src_dirpath) - 1])
      {
            src_dirpath[strlen(src_dirpath) - 1] = '\0';
      }
      if ('/' == dst_dirpath[strlen(dst_dirpath) - 1])
      {
            dst_dirpath[strlen(dst_dirpath) - 1] = '\0';
      }

      /*
       * Create the output file with file table
       */
      snprintf(filename, sizeof(filename) / sizeof(filename[0]), "%s/%s",
dst_dirpath, ROM_FILE_TABLE_FILE);
      file_table = fopen(filename, "wb");
      if (NULL == file_table)
      {
            perror("Creating file table");
            return_error = EXIT_FAILURE;
            goto main_error_handler;
      }
      fprintf(file_table, "/* This content was generated automatically
*/ \n\n");

      /*
       * First walk to get list of included files.
       */
      flags = FTW_MOUNT | FTW_PHYS; // do not cross over another fs, do
not deref. symb. links.
      if (nftw(src_dirpath, generate_list_files, 10, flags) == -1)
      {
            perror("nftw");
            return_error = EXIT_FAILURE;
            goto main_error_handler;
      }

      fprintf(file_table, "\n\n");

      /*
       * Second walk to create file content and rom fs struct.
       */
      fprintf(file_table, "static const struct ROM_FILE_TABLE
```

```
romfs_image[] = {\n");

      if (nftw(src_dirpath, generate_rom_files, 10, flags) == -1)
      {
            perror("nftw");
            return_error = EXIT_FAILURE;
            goto main_error_handler;
      }

      // Lexico sort rom table
      qsort (rom_file_table, rom_file_records, sizeof(ROM_FILE_TABLE),
filename_lexico_compare);

      // Print sorted table content into file.
      for(size_t i = 0; i < rom_file_records; i++)
      {
       fprintf(file_table, "\t{\"%s\", %s, %s},\n",rom_file_table[i].name,
rom_file_table[i].content, rom_file_table[i].size);
      }

      // Add termination bracket
      fprintf(file_table, "};\n");

      main_error_handler:
      if (NULL != file_table)
      {
            fclose(file_table);
      }
      free(src_dirpath);
      free(dst_dirpath);
      free(rom_file_table);

      exit(return_error);
}
```