TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Test operačních systémů pro IoT v minimální hardwarové konfiguraci

Diplomová práce

Studijní program: N2612 – Informační technologie
Studijní obor: 1802T007 – Informační technologie

Autor práce: **Bc. Zdeněk Rindt**
Vedoucí práce: Ing. Lenka Kosková-Třísková

Liberec 2017

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

Test of operating systems for IoT in minimal hardware configuration

Master thesis

Study programme: N2612 – Information technology

Study branch: 1802T007 – Information technology

Author: **Bc. Zdeněk Rindt**

Supervisor: Ing. Lenka Kosková-Třísková

Liberec 2017

Technická univerzita v Liberci
Fakulta mechatroniky, informatiky a mezioborových studií
Akademický rok: **2016/2017**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Zdeněk Rindt**
Osobní číslo:	**M14000177**
Studijní program:	**N2612 Elektrotechnika a informatika**
Studijní obor:	**Informační technologie**
Název tématu:	**Test operačních systémů pro IoT v minimální hardwarové konfiguraci**
Zadávající katedra:	**Ústav nových technologií a aplikované informatiky**

Zásady pro vypracování:

1. Seznamte se s operačními systémy označovanými jako "systémy pro IoT" a proveďte jejich srovnání podle zvolených vlastností.
2. Vyberte tři nejvnodnější systémy pro srovnávací test.
3. Zprovozněte zvolené operační systémy na testovacím hardware.
4. Navrhněte a zprovozněte srovnávací úlohu (měření dat a bezdrátová komunikace).
5. Vyhodnoťte výsledky měření.

Rozsah grafických prací: **dle potřeby**

Rozsah pracovní zprávy: **40 - 60 stran**

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

Seznam odborné literatury:

[1] Mc Even, Adrian, Cassimally, Hakim: Designing the Internet of Things, John Wiley & Sons, 2013, ISBN: 9781118430620
[2] Hersent, Olivier, Boswarthick, David, Elloumi, Omar: Internet of Things Key applications and protocols, John Wiley & Sons, 2011, ISBN: 9781119958345
[3] Fortier, Paul; Michel, Howard: Computer Systems Performance Digital Press 2003, ISBN: 9780080502601

Vedoucí diplomové práce: **Ing. Lenka Kosková - Třísková**
Ústav nových technologií a aplikované informatiky

Datum zadání diplomové práce: **20. října 2016**

Termín odevzdání diplomové práce: **15. května 2017**

Technická univerzita v Liberci

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

prof. Dr. Ing. Jiří Maryška, CSc.
vedoucí ústavu

V Liberci dne 20. října 2016

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Současně čestně prohlašuji, že tištěná verze práce se shoduje s elektronickou verzí, vloženou do IS STAG.

Datum: 15. 5. 2017

Podpis:

Abstrakt

Práce srovnává operační systémy určené pro vestavné systémy s omezenou hardwarovou konfigurací – procesory, kterým často chybí podpora pro privilegovaný a neprivilegovaný režim, ochranu paměti, dokonce i výpočty v plovoucí řádové čárce. Navíc mají relativně nízkou taktovací frekvenci, málo programové paměti a ještě méně operační paměti.

Práce vymezuje pojem operační systém a vestavný systém a nabízí letmé srovnání architektury vestavných systémů a specifik vývoje pro tyto systémy s architekturou osobních počítačů a vývojem pro ně. Následuje přehled některých operačních systémů pro vestavné systémy a výběr tří z nich pro další srovnání. Druhá polovina práce je věnována procesu zprovoznění těchto systémů na vybraných vývojových kitech, a to od přípravy prostředí pro vývoj, přes vlastní přenos, pokud byl potřeba, řešené problémy a implementaci jednoduché testovací aplikace. Na závěr jsou tyto operační systémy zhodnoceny podle vybraných kritérií.

Klíčová slova

IoT, Internet věcí, Operační systémy, RIOT, mbed, ARM, MCU, Vestavné systémy, Embedded Software, STM, Freescale, C, C++

Abstract

This thesis compares operating systems for embedded systems with limited hardware capabilities such as missing support for privileged and unprivileged mode, memory protection and even floating point calculations in some cases. Processors in these systems have relatively low clock frequency, small program memory and even smaller RAM.

In the first part, this thesis defines what an operating system and embedded systems are and points out differences in architecture of embedded systems and software development for them as opposed to personal computers' architecture and software development. It then gives an overview of few operating systems for embedded and three of them are selected for further comparison. Second half of this thesis is dedicated to the process of making the operating systems run on selected evaluation kits, i.e. setting up the development environment, porting the system if neccessary, dealing with problems and creating simple program for comparison. In the end the systems are evaluated based on selected criteria.

Keywords

IoT, Internet of Things, Operating systems, RIOT, mbed, ARM, MCU, Embedded Software, STM, Freescale, C, C++

Poděkování

Rád bych poděkoval Ing. Lence Koskové-Třískové za její vedení a konzultace během naší spolupráce.

Dále bych chtěl poděkovat svým kolegům ze společnosti JABLOTRON ALARMS, a. s., Karlu Vladařovi, Ing. Radoslavu Nejedlovi a Ing. Martinu Kopalovi, za zapůjčení dodatečného hardware, měřících přístrojů a technické konzultace.

Zejména bych rád poděkoval své rodině a svým přátelům, kteří mi byli a jsou oporou.

Obsah

Seznam obrázků

Seznam tabulek

Seznam ukázek zdrojového kódu

Seznam zkratek

API	Application Programmable Interface
BSS	Block Started by Symbol, sekce neinicializovaných dat
CLI	Command Line Interface
FIFO	First In First Out
FPU	Floating Point Unit, jednotka pro práci s čísly s plovoucí řádovou čárkou
GCC	GNU Compilers Collection, kolekce překladačů GNU
GDB	GNU Debugger
GPIO	General Purpose Input/Output
HAL	Hardware Abstraction Layer, vrstva pro abstrakci hardware
I^{2}C	Inter-integrated Circuit
IDE	Integrated Development Environment, integrované vývojové prostředí
IoT	Internet of Things
IRQ	Interrupt Request
ISR	Interrupt Service Routine, obsluha přerušení
MMU	Memory Management Unit, jednotka pro správu paměti
MSYS	Minimal System
OpenOCD	Open On-chip Debugger
OS	Operační systém
PC	Program Counter, ukazatel na instrukci v programovém kódu, nebo také Personal Computer, osobní počítač
PCB	Process Control Block, řídící blok procesu
RAM	Random Access Memory, paměť s náhodným přístupem
ROM	Read-only Memory, paměť pouze ke čtení
RTOS	Real-time Operating System, operační systém reálného času
SP	Stack Pointer, ukazatel na vrchol zásobníku
SPI	Serial Peripheral Interface
SSD	Solid State Disk
UART	Universal Asynchronous Receiver/Transmitter
USART	Universal Synchronous/Asynchronous Receiver/Transmitter

1 Úvod

Mezi lety 2008 a 2009 překonal počet zařízení připojených k Internetu počet lidí na naší planetě (Evans, 2011). V roce 2015 už bylo k Internetu připojeno 12,5 miliardy zařízení, na konci roku 2016 to bylo 16 miliard a odhaduje se, že v roce 2022 by to mohlo být 29 miliard zařízení (Ericsson AB, 2016).

Podíl stolních počítačů, notebooků a tabletů je k roku 2016 kolem 10 %, chytré telefony představují zhruba 46 %. Asi 35 % jsou chytrá zařízení, „věci“ v Internetu věcí. V roce 2022 by tato zařízení mohla představovat 62 % ze všech zařízení připojených k síti Internet.

Co je to „věc“ v pojmu Internet věcí? Může se jednat o celkem malé a nevýkonné zařízení, jehož připojení k síti je realizováno bezdrátově, může se napájet z baterie a nebo si tvoří energii samo. Vybaveno bude senzory pro měření dat. Takové zařízení může být třeba uzel senzorové sítě, zařízení pro dálkový odečet elektroměru, lokalizační obojek pro domácího mazlíčka, nebo alarm do auta. Ale jsou tu i věci velké, například autonomní vozidla, chytré lednice, nebo chytré budovy.

Mnoho aplikací, třeba zmíněné chytré budovy, zároveň spadá do oborů automatizace a průmyslového nasazení minipočítačů. Myšlenka Internetu věcí tedy není nová, ale rozšířila se jako pojem a díky cenové dostupnosti diskrétních součástek, elektronických stavebnic a také jejím popularizátorům, se nyní dostává k širším masám lidí.

Také výrobci se předhání v uvádění věcí, připojených k Internetu, přestože se stále zpravidla jedná o navzájem nekompatibilní ostrovní systémy. Tento hektický vývoj pro výrobce znamená zkrátit vývojový cyklus zařízení.

V oblasti konstrukce lze využít 3D tisku, kterým je možné ladit počáteční fáze návrhu těla zařízení, než jsou vyrobeny drahé formy pro vstřikovací lis. U hardware lze využít referenčních vývojářských desek osazených procesorem a případně i dodatečnými senzory a dalšími komponenty. Na nich lze testovat možnosti platformy a počáteční návrh software.

Celý produkční cyklus od analýzy trhu až po uvedení zařízení může trvat třeba i 2 roky. Co když ale v průběhu vývoje dojde třeba ke změně procesoru. Pokud byl software psán „na tělo“ zvolené platformě, pak jeho přepsání zdržuje vývoj, přináší další náklady a pozdní uvedení na trh může mít na firmu katastrofální dopad.

A zde se může uplatnit vývoj nad operačním systémem, který aplikační kód odstíní od konkrétního procesoru a jeho periferií tím, že poskytuje uniformní rozhraní pro přístup k hardware.

2 Vymezení pojmů

Internet of Things nebo také Internet věcí je propojení vestavných, často jednoúčelových, systémů do sítě Internet. Podle ITU-T (2012) se jedná o: „Globální infrastrukturu informační společnosti umožňující vznik pokročilých služeb propojením (fyzických i virtuálních) věcí vzniklých využitím existujících i vyvíjejících se informačních a komunikačních technologiích.“.

Vestavný systém je jednoúčelové zařízení obsahující řídící počítač zabudovaný přímo v těle zařízení. Na rozdíl od počítačů PC se jedná o kombinaci hardware a software optimalizovanou pro konkrétní úlohu. Jejich softwarová výbava může být neměnná, například u průmyslových vestavných systémů, jiná zařízení ale mohou umožnit aktualizaci programu a s ní rozšíření systému o nové funkce.

Toolchain je sada programů pro překlad programu z jazyka vyšší úrovně do strojového kódu pro konkrétní platformu a operační systém. Například toolchain GCC se skládá z překladačů jazyků C a C++ (a dalších), sestavovacího programu (linkeru), programu pro ladění GDB (debugger) a dalších doprovodných nástrojů pro vlastní proces sestavení.

3 Operační systém

Operační systém je software řídící hardware počítače. Primární funkcí je správa systémových prostředků a poskytování rozhraní k těmto prostředkům aplikacím. Mezi další funkce a služby operačního systému patří:

- Plánování a rozdělování procesorového času pro spuštěné úlohy, jedná-li se o víceúlohový systém.
- Správa paměti.
- Řízení hardware a periferií. K tomu systém potřebuje ovladače, vyvíjené typicky výrobcem daného hardware.
- Souborový systém nad úložištěm dat.
- Síťová vrstva.
- Zabezpečení.

Díky abstrakci přístupu k hardware a všem jmenovaným službám, je možné vyvíjet aplikace nikoliv pro konkrétní hardware, ale pro operační systém. Takovou aplikaci lze pak v podstatě beze změn přenášet mezi různým hardware, běží-li na něm operační systém, pro který byla aplikace naprogramována.

3.1 Víceúlohové systémy

Operační systémy, které známe ze stolních počítačů a serverů jsou víceúlohové (multi-task), dokáží obsluhovat několik zároveň běžících aplikací.

Jednojádrový procesor může vykonávat kód jediné úlohy. Operační systém spravuje seznam spuštěných procesů – úloh – a zajišťuje jejich střídání tak, aby měl uživatel dojem, že všechny spuštěné procesy běží najednou.

Střídání úloh vykonává *plánovač*, který procesorový čas přiděluje v závislosti na prioritě úloh, jejich aktuálním stavu, držených a požadovaných prostředcích a dalších parametrech.

3.1.1 Kooperativní multi-tasking

Běžící proces musí dostatečně často provádět systémové volání a tím dát operačnímu systému šanci spustit jiný proces z fronty.

Jsou-li úlohy dobře naprogramované, jde o přímočarou metodu implementace víceúlohového systému. Pokud ovšem proces nevolá operační systém, ať už úmyslně, nebo chybou programového kódu, může dojít k uváznutí systému a nutnosti systém restartovat.

3.1.2 Preemptivní multi-tasking

Moderní operační systémy používají pre-emptivní multi-tasking. Plánovač přidělí úloze procesorový čas, po jehož vypršení dojde automaticky k systémovému volání. Řízení opět přebere operační systém, který rozhodne, kterou úlohu spustí jako další.

Automatické přepnutí řízení může být implementované například hardwarovým časovačem, po jehož přetečení dojde k vyvolání přerušení. Procesor v této situaci začne vykonávat kód obsluhy tohoto přerušení, kterým je kód operačního systému zajišťující změnu aktivního procesu.

3.2 Systémy reálného času

V kritických aplikacích mohou být kladeny zvláštní požadavky na odezvu systému, garance maximální doby odezvy a dodržení termínu zpracování úlohy. Tyto systémy, anglicky označované jako real-time, se dělí na dvě skupiny:

3.2.1 Tvrdé systémy reálného času

V aplikacích, jako je zdravotnictví, robotika, nebo zbraňové systémy, může být po systému požadováno, aby úlohy byly zpracovány do určitého termínu, a to včetně režie operačního systému. Takový systém funguje správně, pokud dodává správné výsledky ve stanovených časových mezích. Nedodání výsledku včas se rovná nedodání výsledku vůbec.

Takové systémy postrádají funkce, které do času zpracování přidávají nejistotu, stejně jako další funkce obsažené v běžných víceúlohových systémech, jako je například virtuální paměť (Abraham et al., 2002, str. 170).

Jednou z klíčových částí je plánovač, který na základě požadavku úlohy na termín zpracování je schopen dopředu rozhodnout, zda úlohu akceptuje a zvládne jí zpracovat v termínu, nebo odmítne.

3.2.2 Měkké systémy reálného času

Jiné aplikace, jako jsou multimédia, také vyžadují garance. Tyto je ale možné implementovat ve víceúlohovém systému pomocí priorit. Kritická úloha pak dostává více procesorového času, přičemž o něj připravuje jiné. Dochází tím ale k prodlevám

dokončení úloh s nižší prioritou, nebo dokonce i k tzv. hladovění některých procesů, které se dlouho nedostávají k procesoru.

3.2.3 Inverze priorit

Problém nastává, pokud úloha s vysokou prioritou vyžaduje prostředek, který zrovna drží úloha s nízkou prioritou. Kvůli nízké prioritě se této úloze nedostává tolik procesorového času, jako jiným, což prodlužuje dobu k jejímu dokončení, ale především to způsobuje, že úloha s vysokou prioritou čeká. A jelikož čeká na úlohu s nižší prioritou, hovoříme o *inverzi priorit*.

Tomu by měl operační systém zabránit. Jedním z řešení je, že úloha držící prostředky požadované úlohou s vysokou prioritou dočasně zdědí prioritu této úlohy, aby mohla být rychleji dokončena a prostředky uvolnit.

3.3 Proces

Procesem rozumíme spuštěný program, který se skládá z programového kódu, vstupních parametrů a řídících dat. Mezi řídící data patří například kontext – nastavení stavových registrů programu, nebo počítadlo programu (známé jako *Program Counter*), které ukazuje na vykonávanou instrukci programu.

Když dochází k přepínání procesů, musí operační systém uložit kontext aktuálně běžícího procesu a obnovit kontext dalšího procesu, aby pokračoval přesně v tom místě, kde skončil.

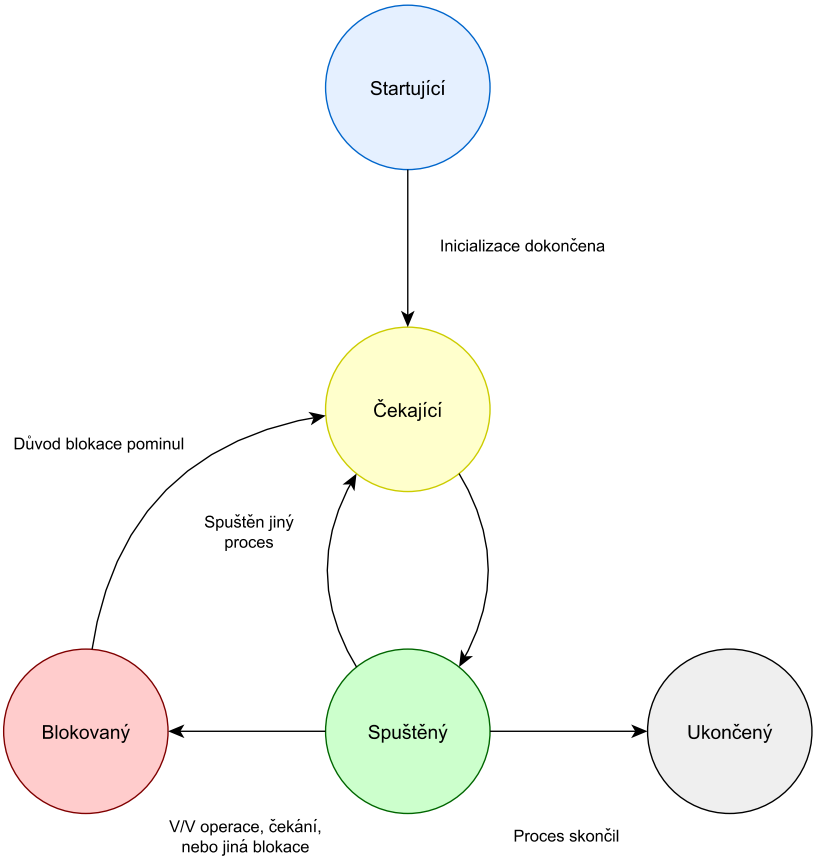
3.3.1 Životní cyklus procesu

Proces má několik stavů, ve kterých se může nacházet. Přechody mezi těmito stavy jsou znázorněny na obrázku 3.1. Stavy procesu jsou následující:

1. *Startující* proces je operačním systémem nahrávaný do operační paměti.
2. *Čekající* proces je schopný běžet, ale nebyl mu přidělen procesor.
3. *Spuštěný* proces aktuálně běží.
4. *Blokovaný* proces nemůže běžet, protože čeká třeba na dokončení vstupně-výstupní operace, nebo uvolnění kritické sekce.
5. *Ukončený* čeká na své uklizení operačním systémem.

Proces může být blokován například dlouhotrvající operací, nebo čekáním na data od jiného procesu, vstup uživatele, apod.

Spuštěný proces může být přerušen operačním systémem, když mu vyprší přidělený procesorový čas, nebo když přijde třeba přerušení od externí periferie, které odblokuje proces s vyšší prioritou. Přestože nebyl dosud běžící proces ničím blokován, je jeho stav změněn na *Čekající* a spuštěn je jiný proces.



Obrázek 3.1: Životní cyklus procesu

3.3.2 Řídící blok procesu

Operační systém si drží informace o každém spuštěném procesu. Této struktuře se říká *Process Control Block* a uchovává nejen zmíněný kontext, ale také další parametry (Tanenbaum, 2001, str. 79-80):

- *Program Counter*, stavový registr, ukazatel zásobníku, a další registry procesoru.
- Ukazatele na segmenty paměti programu, dat a zásobníku.
- Stav procesu, viz 3.3.1.
- Priorita procesu a další parametry pro plánovač.
- Operační systémy jako jsou Microsoft Windows nebo Linux ale o procesech uchovávají mnohé další informace, například statistiky běhu, využití procesorového času, dobu od spuštění a další.

3.3.3 Paměť procesu

Spuštěný program v paměti zabírá více místa, než odpovídá velikosti jeho kódu. To proto, že běžící program se neskládá pouze z programového kódu, ale potřebuje také paměť pro svůj běh, kterou je třeba alokovat v paměti RAM. Zejména se jedná o paměť pro:

- Zásobník, na kterém jsou při běhu ukládány lokální proměnné, návratové adresy volání funkcí a argumenty funkcí.
- Programovou paměť, kam je zkopírován kód programu. Tato paměť, pokud to architektura a operační systém podporují, je určena pouze ke čtení. Některé procesory pro mikropočítače, například STM32 architektury ARM od STMicroelectronics, mají zvlášť paměť typu Flash pro program a konstantní data, a zvlášť operační paměť typu SRAM pro běhová data. Její velikost je zpravidla zlomkem velikost paměti Flash (viz tabulka 5.1).
- Paměť konstant, kam jsou zkopírovány například řetězce a další konstantní proměnné, které nelze v programu modifikovat. Stejně jako sekce programové paměti, je i tato určena jen ke čtení.
- Inicializovanou paměť, kde se nachází všechny globální a statické proměnné, které mají přiřazenu hodnotu. Tato paměť umožňuje nejen čtení, ale také zápis.
- Neinicializovanou paměť, která obsahuje globální a statické proměnné, které neměly přiřazenu hodnotu. V jazyce C jsou statické proměnné umístěny v sekci BSS, která je vyplněna nulami.
- A nakonec halda, tedy paměť sloužící k dynamické alokaci paměti.

3.4 Vlákna

Pokud chceme na systémech s více procesory dosáhnout paralelismu, musíme práci rozdělit mezi více procesů, které mohou běžet každý na svém procesoru (nebo procesorovém jádře). Každý proces má vlastní paměť (viz 3.3.3) a komunikovat spolu mohou pouze s užitím prostředků operačního systému, např. sdílenou pamětí.

Zjednodušením je využití tzv. vláken, která spolu sdílí jeden paměťový prostor procesu, v rámci kterého byla vytvořena. Každý proces je spuštěn s jedním vláknem vytvořeným operačním systémem. Program ale může vytvořit další vlákna, mezi která rozdělí práci.

Každé vlákno má svůj *Program Counter*, zásobník a zálohované registry procesoru. Paměť (a tedy například globální proměnné) a prostředky ale sdílí s ostatními vlákny. To vyžaduje jistou disciplínu a správný návrh programu, jelikož paralelní zpracování více vláken může vést k problematickým situacím právě při práci s pamětí, viz 3.5.

Ačkoliv vnáší vlákna komplikace do návrhu aplikace skrze nutnost synchronizovat data a udržovat je konzistentní, mají vlákna samozřejmě několik výhod:

- Umožňují rozdělit problémy na dílčí části a ty paralelizovat, a přitom sdílet paměť i bez asistence operačního systému.
- Vytvořit nové vlákno je v některých systémech až 100× rychlejší, než vytvořit nový proces (Tanenbaum, 2001, str. 85).

3.5 Paralelismus a sdílené prostředky

Pomocí procesů a vláken můžeme docílit paralelismu a lépe využít procesorový čas. Paralelismus ale také přináší nové problémy, a to při přístupu ke sdíleným prostředkům.

3.5.1 Problém souběhu

Mějme dvě vlákna `A` a `B`. Tato spolu sdílí paměť a přistupují ke globální proměnné `P` typu `int`, jejíž hodnota na počátku je `0`. Obě vlákna provádí jednoduchý kód, viz 3.1.

```
if (P == 0) {
  P = P + 1;
}

printf("%d\n", P);
```

Ukázka kódu 3.1: Ukázka pro souběh vláken

1. Vlákno `A` přečte hodnotu `P == 0`, podmínka je vyhodnocena pravdivě.
2. Čas přidělený vláknu `A` vypršel a dojde k aktivaci vlákna `B`.
3. Vlákno `B` přečte hodnotu `P == 0` a podmínka je vyhodnocena pravdivě.
4. Vlákno `B` inkrementuje hodnotu proměnné `P`, ta je nyní rovna 1.
5. Vlákno `B` vypíše na výstup hodnotu proměnné, tedy 1, a skončí.
6. Nyní se aktivuje vlákno `A` a pokračuje kde skončilo, tedy uvnitř podmínky, a inkrementuje proměnnou `P`, hodnota je nyní rovna 2.
7. Vlákno `A` vypíše na výstup hodnotu, tedy 2, a skončí.

Mohli bychom vymyslet i další možné scénáře. V některých by by byla proměnná `P` inkrementována pouze jednou, pokud by nedošlo k přerušení prvního vlákna, v jiných by se mohlo změnit pořadí, v jakém vlákna skončila. Dokonce by mohlo dojít

i k nesprávnému výpisu, pokud by vlákno bylo přerušeno v průběhu volání funkce `printf`.

Problém s uvedeným kódem je, že operace porovnání a následné inkrementace není atomická. K tomu bychom potřebovali získat exkluzivní přístup k proměnné `P`.

3.5.2 Exkluzivní přístup

Operační systémy naštěstí implementují množství synchronizačních primitiv, z nichž nejjednodušším je *semafor*.

Semafor slouží k ohraničení sekce kódu, do které chceme vpustit jen limitované množství vláken. V podstatě jde o jednoduché počítadlo, doprovázené funkcemi `zamkni` a `odemkni`. Hodnota počítadla, s jakou semafor inicializujeme, udává, kolik vláken je vpuštěno do dané sekce.

- Před vstupem do kritické sekce volá vlákno funkci `zamkni`. Pokud je hodnota počítadla nenulová, pak je snížena o 1 a vlákno pokračuje dále.
- Na konci kritické sekce vlákno musí volat funkci `odemkni`, která hodnotu počítadla opět inkrementuje.
- Pokud byla hodnota počítadla při volání `zamkni` nulová, pak se vlákno zastaví a nepokračuje dále, dokud nebude počítadlo nenulové.

Vlastní implementace semaforů je ale složitější. Popis dává tušit, že kostrou by mohl být kód podobný ukázce 3.1, který by tak trpěl opět na souběh. Synchronizační primitiva jsou proto implementována za využití prostředků operačního systému, který může zabránit přepnutí vláken po čas zamykání a odemykání a zajistit tak atomicitu. Do hry navíc kromě vláken vstupují také přerušení procesoru. K těm může dojít kdykoliv a může je dočasně vypnout pouze operační systém, viz sekce 3.7.1 o režimu běhu procesoru.

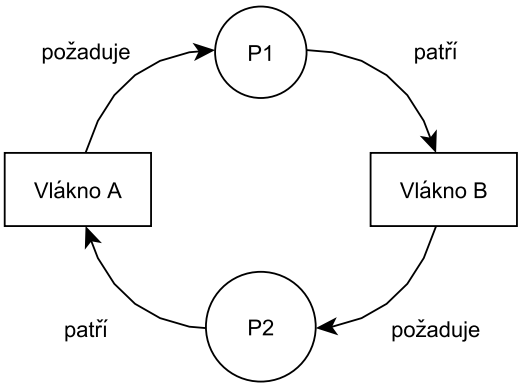
3.5.3 Uváznutí

Exkluzivním přístupem je možné zabránit souběhu, ale při nesprávném návrhu aplikace může dojít k jinému nepříjemnému problému.

Pokud proces nebo vlákno `A` drží prostředek `P1` a čeká na získání prostředku `P2`, zatímco vlákno nebo proces `B` drží prostředek `P2` a čeká na získání prostředku `P1`. V tu chvíli není ani jedno z vláken z pohledu operačního systému běhuschopné a hovoříme o *uváznutí*, angl. *deadlock*. Viz obrázek 3.2 znázorňující situaci.

Podmínky uváznutí

Situace může být samozřejmě podstatně komplikovanější a aktéry uváznutí může být více vláken. Důležité je, že čekají jedno na druhé, až uzavřou kruh. To je jednou z podmínek, které jsou následující:



Obrázek 3.2: Uváznutí dvou procesů

1. Proces nebo vlákno, které požaduje přístup k prostředku, k němuž má exkluzivní přístup jiný proces nebo vlákno, musí vyčkat na jeho uvolnění.
2. Čekající proces nebo vlákno již drží nějaký prostředek.
3. Prostředky nelze procesu nebo vláknu odebrat, dokud se jich proces nebo vlákno sami nevzdají.
4. Procesy nebo vlákna mohou při čekání vytvořit kruh.

Řešení uváznutí

Existují postupy, kterými je možné uváznutí předejít, například eliminace některé z podmínek vzniku uváznutí. Může být například definován protokol, kterým se musí řídit alokace zdrojů procesem. Každý typ zdroje bude mít přiřazen unikátní identifikátor, přirozené číslo. Procesy pak musí zdroje alokovat tak, aby pořadí identifikátorů bylo vždy vzestupné. Tím je zabráněno možnosti vzniku kruhových závislostí procesů (Abraham et al., 2002, str. 252).

Operační systémy mohou implementovat algoritmy předcházející uváznutí, takové ale vyžadují informace od procesů o požadovaných zdrojích. Většina operačních systémů, včetně Unixu, ale uváznutí nepředchází (Abraham et al., 2002, str. 266). Další možností je nepředcházet uváznutí, ale umět jej detekovat a zotavit se z něj, například restartem systému, nebo ukončením uvázlých procesů.

3.5.4 Paralelismus ve vestavných systémech

Jedno ze specifik vývoje software pro vestavný systém je, že se s pseudo-paralelismem setkáváme i bez použití víceúlohového operačního systému, při programování aplikace přímo pro cílovou architekturu. Při běhu programu totiž může dojít ke změně kontextu vlivem **přerušení** z vnějšího vstupu, nebo od periferie. V tu chvíli se začne vykonávat kód obsluhy přerušení a po jeho dokončení dojde k obnovení kontextu a pokračování v programu na původním místě.

Pokud jednu část paměti, třeba globální proměnnou, využívá hlavní smyčka programu, a zároveň stejnou část paměti využívá obsluha přerušení, pak se bude programátor potýkat se všemi jmenovanými problémy. Typickým řešením je vypnout po dobu vykonávání kritické sekce přerušení procesoru. Po skončení kritické sekce program přerušení opět zapne a procesor začne obsluhovat všechna přerušení, ke kterým mezitím došlo.

Problému **uváznutí** vestavné systémy, které zpravidla nemají dostatek prostředků, neumí předcházet, přesto je klíčové, aby byl schopen se systém zotavit. Z toho důvodu jsou procesory pro vestavné systémy (a mikrokontroléry obecně) vybaveny speciálním časovačem, kterému se říká *watchdog*.

Program musí počítadlo časovače periodicky nulovat. Pokud by došlo k jeho přetečení, provede watchdog reset systému.

3.6 Správa paměti

Úlohou operačního systému je správa operační paměti, která je cenným zdrojem, zejména v případě minipočítačů. Jednoúlohové operační systémy (například MS-DOS) měly část operační paměti vyhrazenu pro sebe a část adresního prostoru patřila procesu. Programy využívaly absolutní adresování.

Ve víceúlohovém operačním systému je to ale problém. Operační systém musí při spouštění procesů pro tyto vyhradit část paměti, do které nahraje program. Musí tedy vědět, které části paměti už vyhradil kterým procesům a které jsou volné. Pokud by navíc procesy používaly absolutní adresování, musel by být takový program vždy nahrán na stejné místo, jinak by mohl číst a zapisovat do paměti cizího procesu. Je tedy nutné, aby program mohl být ve fyzické paměti umístěn kdekoliv a přesto fungoval.

Dalším problémem je, že proces může v průběhu běhu alokovat další paměť a kapacita fyzické operační paměti nestačí na pojmutí všech spuštěných procesů.

3.6.1 Virtuální paměť

Jedním z řešením těchto problémů je virtuální paměť, vyžaduje ale podporu jak operačního systému, tak procesoru. Adresám, které se v programu vyskytují jako operandy pro instrukce čtení, zápisu nebo skoku, říkáme virtuální adresy. Namísto toho, aby virtuální adresa putovala přímo na adresovou sběrnici, je nejdříve zpracována *Jednotkou správy paměti* (MMU), která virtuální adresu převede na fyzickou adresu.

V kombinaci se stránkováním (viz 3.6.2) je možné přidělovat procesu více paměti, než jaká je skutečná velikost operační paměti.

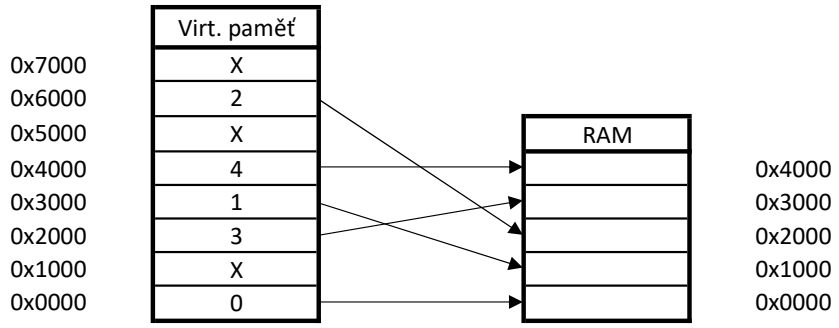
Virtuální adresní prostor znamená, že proces může být fyzicky v paměti umístěn kdekoliv, i bez nutnosti relokace (tj. přepočítávání adres), a virtuální adresy jsou z hlediska programu stále stejné.

3.6.2 Stránkování

S virtuální pamětí úzce souvisí metoda stránkování. Virtuální adresní prostor je rozdělený na bloky stejné velikosti, které nazýváme stránky.

Protože je operační paměť zpravidla menší, než je celý virtuální adresní prostor procesu, uchovává operační systém seznam stránek, které jsou v operační paměti načteny, a informace o tom, kde jsou umístěny.

To znázorňuje obrázek 3.3. První sloupec představuje virtuální adresní prostor procesu, buňky reprezentují jednotlivé stránky o velikosti 4 KiB a hodnota buňky je číslo rámce stránky – index stejně velkého bloku v operační paměti, ve kterém je fyzicky stránka umístěná.



Obrázek 3.3: Mapování stránek virtuálního adresního prostoru do operační paměti

Překlad adresy

Řekněme, že je zpracovávána instrukce odkazující se na virtuální adresu `0x309C`. Dle obrázku 3.3 spadá tato adresa do stránky č. 3, která je mapována na rámec stránky č. 1. Adresa bude přeložena na hodnotu `0x109C`.

Výpadek stránky

Nyní je zpracovávána instrukce obsahující virtuální adresu `0x5F10`. Tato adresa spadá do stránky č. 5, která je ale na obrázku 3.3 označena symbolem křížku a není tedy obsažena v operační paměti.

V tomto případě dojde k tzv. *výpadku stránky*, je vyvoláno přerušení a řízení přebírá operační systém. Jeho úlohou je zajistit uvolnění některého rámce stránky v operační paměti (která je v příkladu zaplněna) a chybějící stránku do paměti načíst, například z pevného disku.

Metodě, kdy jsou stránky ukládány a načítány z pevného disku, se říká *swapování*. Protože jsou pevné disky s rychlostmi v řádech 100 MiB/s a, v případě magnetických disků, odezvou v řádech milisekund podstatně pomalejší, než operační paměť s rychlostmi v řádech jednotek až desítek GiB/s[1], je swapování při nedostatku operační paměti obrovskou výkonnostní brzdou.

[1]Moduly DDR4-1600, značící 1.600 přenosů za sekundu s šířkou datové sběrnice 64 bitů, dosahují teoretické přenosové rychlosti až 12.800 MiB/s.

3.7 Jádro

Pojmem operační systém myslíme kompletní software, včetně dodaných aplikací, nástrojů, ale i uživatelského rozhraní, apod. Základem operačního systému je ale jádro, které zajišťuje popsanou základní funkcionalitu, tedy přístup k hardware počítače, správu procesů (úloh), správu paměti a další.

3.7.1 Režim běhu

Procesory architektury x86, některé procesory architektury ARM, ale i mnohé další architektury, podporují alespoň dva režimy běhu, které označujeme pojmy privilegovaný a neprivilegovaný.

Privilegovaný režim Kód běžící v privilegovaném režimu má přístup ke všem prostředkům počítače, tj. může číst a zapisovat do všech registrů procesoru, využívat všechny instrukce procesoru, nebo číst a zapisovat do paměti v celém jejím rozsahu.

Neprivilegovaný režim Kód procesu spuštěného v neprivilegovaném režimu má naopak přístup k hardware limitovaný. Pokud chce spustit instrukci, která v neprivilegovaném režimu není povolena, nebo přistoupí k paměti mimo dovolený rozsah, dojde k aktivaci jádra operačního systému, které rozhodne o vykonání instrukce.

Implementace režimů běhu je samozřejmě platformě závislá a může se lišit mezi architekturami. Podporu musí mít nejen hardware, ale také operační systém. Kód jádra operačního systému pak běží v privilegovaném režimu, zatímco uživatelské procesy běží v režimu neprivilegovaném.

Pokud by chtěl jeden proces pozměnit (ať už úmyslně, nebo vinou chyby v programovém kódu) paměť, která mu nepatří, a tím narušit data nebo kód jiného procesu, nebo dokonce samotného jádra operačního systému, pak dojde k systémovému volání a jádro může této změně zabránit. Tímto způsobem je zajištěna základní bezpečnost a integrita operačního systému.

3.7.2 Monolitické jádro a mikrojádro

Základní části operačního systému, jako je plánovač, správa procesů, meziprocesová komunikace, nebo obsluha přerušení, běží v privilegovaném režimu. Jak je to ale s dalšími částmi operačního systému?

Monolitické jádro

V případě monolitického jádra běží všechny části operačního systému v režimu jádra, včetně ovladačů, souborového systému, síťové vrstvy a dalšího. Výhodou je, že v rámci těchto částí nedochází k přepínání režimu procesoru. Chyba ve kterémkoliv modulu ale může vést k pádu systému.

Mikrojádro

Kontrastem k monolitickému jádru je mikrojádro, které implementuje pouze nejnutnější služby operačního systému. Ostatní části mohou být implementovány jako samostatné procesy běžící v neprivilegovaném režimu a komunikují spolu skrze meziprocesovou komunikaci.

Výhodou mikrojádra je jeho malá velikost. Pokud dojde k pádu služby, nemusí to mít vliv na ostatní služby ani jádro systému. Vzhledem k častým změnám režimu běhu procesoru je mikrojádro pomalejší, než monolitické jádro.

4 Vestavný systém

Vestavným systémem rozumíme takový systém, který byl jak po softwarové, tak po hardwarové stránce navržen pro konkrétní úlohu. Rozšiřitelnost takových systémů může být omezená, některé jednoduché a jednoúčelové systémy mohou mít program uložený v paměti ROM, která je naprogramována z výroby a není možné jí přepsat. V některých aplikacích k tomu ani není důvod.

Několik příkladů vestavných systémů:

Bankomaty Ačkoliv může být bankomat vybaven klidně i operačním systémem, jako jsou Microsoft Windows, běžícím na hardware nepříliš vzdáleném od stolních počítačů, je také doplněn spoustou dalších periferií, od klávesnice a čtečku karet až k počítačce bankovek. Celek je pak uzavřen v kompaktním těle a jelikož je přizpůsoben konkrétnímu účelu, hovoříme o něm jako o vestavném systému.

Nápojové a podobné automaty nevyžadují nikterak veliký výpočetní výkon, manipulují ale s penězi, což klade zvláštní nároky na kvalitu programu a opět je vybaven netradičními periferiemi.

Systémy pro řízení výrobních linek nebo jejich částí, například lisů, dopravníků, apod. Takové systémy nemusí nutně vyžadovat možnost změny programu a může postačovat nízký výpočetní výkon. Může být ale vyžadována zvýšená odolnost proti prachu, nečistotám, vodě, nebo elektromagnetickému záření. Jistě pak bude vyžadována schopnost trvalého běhu bez uváznutí, případně se zařízení musí z takového stavu umět zotavit.

Zábavně-informační systémy v automobilech poskytující informace palubního počítače, autorádio, navigaci, vše v jednom systému.

Senzorové sítě skládající se z minipočítačů, jejichž úlohou je měření a odesílání dat do sítě. Mohou být provozovány z baterií a mají tedy nároky na nízkou spotřebu.

4.1 Třídy vestavných systémů

Nároky jednotlivých aplikací se liší a podle toho se liší také platformy. Zatímco uzel senzorové sítě si vystačí s trochou paměti a procesorem s taktovací frekvencí pár jednotek MHz a stráví většinu času uspaný, aby měl co nejnižší spotřebu,

zábavně-informační systém v automobilu, který shromažďuje data z různých řídících jednotek, přehrává rádio a filmy na displeji s vysokým rozlišením, naviguje a telefonuje, bude vyžadovat podstatně více výkonu i paměti.

Podle Hahm et al. (2016) se zařízení dělí na třídy 0, 1 a 2 podle výkonnosti zařízení:

Třída 0 jsou nejmenší zařízení s méně než 10 kB RAM a méně než 100 kB nevolatilní paměti (typicky Flash). Tato zařízení mohou být uzly senzorových sítí s jednoduchými komunikačními protokoly.

Třída 1 jsou zařízení se zhruba 10 kB RAM a 100 kB paměti Flash, které umožňují implementaci složitější protokolů a aplikací s větším množstvím funkcí.

Třída 2 mají zdrojů ještě více, stále se ale jedná o limitovaná zařízení v porovnání s takovými, jako je například Raspberry Pi.

4.2 Architektura

Základní řídící jednotkou jednodeskového počítače, minipočítače a podobných vestavných systémů je, stejně jako u PC, procesor. V mnohých ohledech se však od procesorů v osobních počítačích liší a tyto odlišnosti mají zásadní dopad na způsob vývoje programů pro tyto systémy.

Existuje mnoho architektur procesorů pro vestavné systémy. Jednou z nejznámějších architektur můžeme považovat ARM s redukovanou instrukční sadou (RISC). Tyto procesory mají typicky operační paměť typu SRAM pro běhová data a podstatně větší paměť typu Flash, ve které je uložen, a ze které je prováděn, program.

4.3 Rozdíly proti PC

Procesory architektury ARM, ale i jiných architektur, se skládají z licencovaného procesorového jádra, například ARM Cortex-M, a periferií. Jednotlivé periferie si, stejně jako sběrnice mezi nimi, procesorem a pamětmi, navrhuje konkrétní výrobce.

Periferie s procesorovým jádrem v podstatě tvoří „systém na čipu“ (*System-on-Chip*). A zde je největší rozdíl mezi procesorem architektury x86 pro PC a procesorem pro vestavný systém. Ten má totiž velkou sadu periferií dostupnou přímo v čipu, zatímco u stolních počítačů jsou tyto periferie přítomny buď až na základní desce počítače, nebo dokonce jen s rozšiřujícími kartami.

Několik příkladů takových periferií:

- Univerzální vstupy a výstupy. Vstupy mohou být konfigurovány, aby generovaly přerušení v případě změny (výskyt hrany), výstupy mohou být obecné, s možností připínání pull-up a pull-down rezistorů, nebo mohou být, stejně jako vstupy, vázané na některou z periferií.

- Rozhraní I^2C, SPI, CAN, UART, USART, často s dalšími režimy funkce, například periferie USART u procesorů STM32L476 podporuje protokoly LIN, Smartcard, nebo IrDA.
- Čítače a časovače s funkcí generování PWM.
- Analogově digitální a digitálně analogové převodníky.
- Kontroléry pro LCD TFT displeje, nebo kapacitní dotykové rozhraní.
- Rozhraní pro karty SD a MMC.
- Audio rozhraní, USB rozhraní včetně režimu host u některých procesorů, Ethernet.
- Hodiny reálného času, často s možností napájení z externí baterie. Procesory STM mají v periferii hodin reálného času i několik registrů, jejichž obsah je při připojené baterii udržován i když je zbytek procesoru vypnutý.
- Obvod watchdog, který umožňuje resetovat procesor, pokud dojde k uváznutí programu.
- Detektory podpětí (*brown-out*), které resetují procesor a udržují jej v resetu, dokud napětí nedosáhne minimální hodnoty pro zaručenou funkčnost procesoru.
- Kontroléry pro přímý přístup do paměti (DMA) umožňující přesuny dat mezi periferiemi a pamětí.

Téměř všechny z vyjmenovaných periferií navíc dokáží generovat pro různé události, v závislosti na softwarové konfiguraci, **přerušení procesoru**. To umožňuje psát programy řízené událostmi, namísto neustálého manuálního kontrolování, zda se něco nezměnilo (polling).

4.4 Specifika vývoje

Dalšími odlišnostmi ve vývoji programu pro vestavný systém je překlad programu a způsob, jakým se provádí ladění programu pro cílovou architekturu.

4.4.1 Překlad

Pro překlad zdrojového kódu jazyka vyšší úrovně do strojového kódu, kterému rozumí procesor, potřebujeme překladač. Jde o program, který je součástí *toolchainu*, balíku nástrojů pro překlad, sestavení a ladění.

Procesory ARM Cortex-M jsou zcela odlišné architektury s jiným instrukčním souborem v porovnání s architekturami x86 nebo amd64 známých ze stolních počítačů.

Pro překlad programu určeného pro ARM je tak třeba použít *křížový překladač*. O křížovém překladu hovoříme, když je architektura hostitelského systému (na kterém probíhá překlad) odlišná od architektury cílového systému (na kterém program poběží), případně když překládáme program pro stejnou architekturu, ale jiný operační systém.

Linker

Součástí překladače je linker. Jeho úlohou je rozmístit v paměťovém prostoru sekce programového kódu, konstantních dat, neinicializovaných dat a sekce pro zásobník a dynamickou alokace paměti. Linker překladače GCC k rozmístění používá soubor skriptu – *Linker Script*. Jím je možné určit, které sekce budou v paměti RAM a které budou v paměti Flash, a na jakých adresách. Toto rozmístění je nutné provádět se znalostí rozvržení paměti konkrétního procesoru.

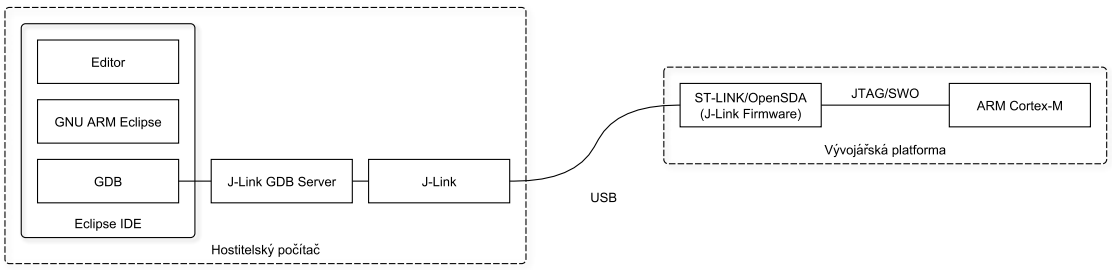
4.4.2 Nahrávání kódu a vzdálené ladění

Po překladu programu pro minipočítač chceme výsledný program odladit na cílovém systému. Ještě předtím jej potřebujeme na tento systém nahrát, k čemuž slouží ladící rozhraní (debugger) kombinované s programátorem, připojené mezi vývojářskou stanici a procesor systému, který chceme programovat, nebo ladit.

Procesorová jádra Cortex-M mají standardizované rozhraní pro ladění, díky kterému je možné číst a zapisovat do operační i programové paměti procesoru, pozastavovat jádro, krokovat program po jednotlivých instrukcích, nebo vkládat a odebírat z programu body přerušení – breakpointy.

Prototypovací desky a vývojářské kity některých výrobců jsou přímo z výroby osazeny debuggerem. Ten potřebuje vlastní procesor a další související součástky, čímž se náklady na takové desky zvyšují.

Stejně jako při ladění aplikace pro stolní počítač můžeme využít ladícího programu GDB. Ten komunikuje s debuggerem a skrz něj řídí procesor cílového systému. Na obrázku 4.1 je znázorněno propojení vývojového prostředí Eclipse IDE, ladícího programu GDB, debuggeru na vývojovém kitu a procesoru systému.



Obrázek 4.1: Propojení mezi vývojovým prostředím a procesorem s laděným kódem

Při vývoji zakázkového systému na desce debugger není. Ladění probíhá pomocí externího debuggeru, který se kabelem připojuje k ladícímu rozhraní vyvedenému

z procesoru na pinovou lištu. Zpravidla je také vyvedeno na testovací body. V sériové výrobě se tato lišta často ani neosazuje a procesory jsou programovány pomocí jehlového pole právě skrze testovací body.

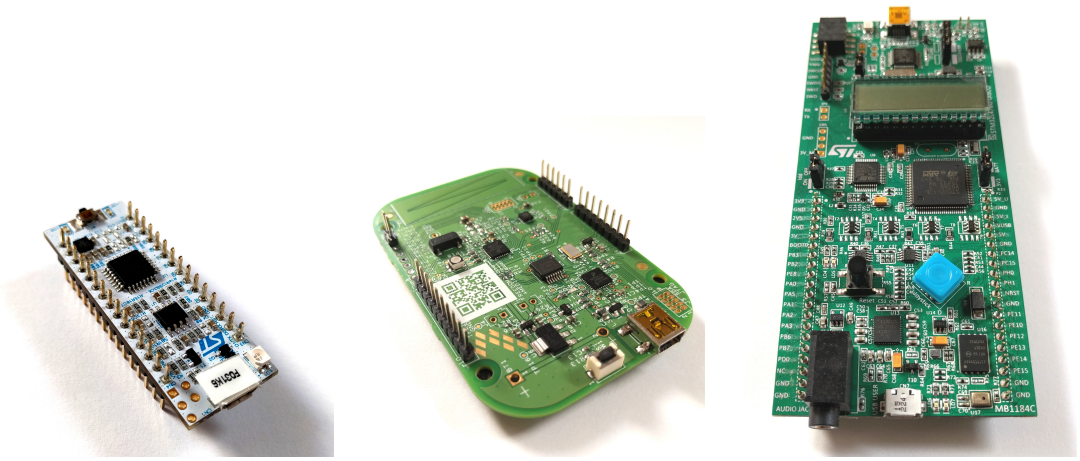
5 Rešerše

5.1 Zvolený hardware

Vedoucí práce byly nabídnuty a dodány tři vývojářské platformy od dvou společností:

- Nucleo F031K6 od STMicroelectronics (obr. 5.1a),
- FRDM KL05Z od Freescale (obr. 5.1b)
- a Discovery L476G, opět od STMicroelectronics (obr. 5.1c).

Všechny procesory obsahují jádro ARM Cortex-M, ale periferie, jejich koncepce a registry se již liší. Osazené procesory mají oddělenou paměť RAM pro běhová data a nevolatilní paměť typu Flash, ze které je prováděn program a kde jsou také uchována neměnná data.



(a) Nucleo F031K6 (b) FRDM KL05Z (c) Discovery L476G

Obrázek 5.1: Vybrané prototypovací platformy

Malé Nucleo F031K6 je možné přirovnat desce Arduino Nano, na rozdíl od Arduina je však (stejně jako ostatní vybrané desky) vybavena programátorem a ladícím rozhraním. Deska FRDM-KL05Z je fyzicky větší než Nucleo a je pinově kompatibilní

s Arduinem. Svými parametry (viz tabulka 5.1) desky odpovídají třídě 0, jak byly definovány v sekci 4 (STMicroelectronics (2017a), Freescale Semiconductor, Inc. (2014)).

Discovery L476G, sloužící také jako záloha pro případ, že by některý z operačních systémů měl větší nároky na paměť, je na rozhraní tříd 1 a 2 (STMicroelectronics, 2015a).

Tabulka 5.1: Specifikace použitých prototypovacích platforem

	Nucleo F031K6	FRDM KL05Z	Discovery L476G
Hodnocení výkonu	Třída 0	Třída 0	Třída 1-2
Jádro	ARM Cortex M0	ARM Cortex M0+	ARM Cortex M4
Frekvence	48 MHz	48 MHz	80 MHz
Paměť RAM	4 kB	4 kB	128 kB
Pameť Flash	32 kB	32 kB	1 MB
Jednotka FPU	-	-	●
Ochrana paměti	-	-	●
Ladící rozhraní	ST-LINK/V2	OpenSDA	ST-LINK/V2-1

(●) podporováno, (-) nepodporováno, (?) nepodařilo se zjistit.

5.2 Omezení platformy

5.2.1 Ochrana paměti

Procesory ve vybraných deskách, s výjimkou Discovery L476G (viz tabulka 5.1), nemají žádnou ochranu paměti. Tu by mohl operační systém využít pro omezení přístupu vláken k částem adresního prostoru (a tedy i periferiím, jejichž registry jsou mapovány do adresního prostoru).

5.2.2 Správa paměti

Procesory nedisponují jednotkou pro správu paměti a tedy nepodporují ani virtuální adresní prostor (sekce 3.6.1). Programy jsou přeloženy s absolutními adresami. Vzhledem k umístění programu v paměti Flash, bez nutnosti jeho kopírování do

RAM, toto omezení většinou nepředstavuje problém. Skutečným problémem může být fragmentace paměti RAM.

Speciální aplikace ale mohou být zkopírovány do paměti RAM a z ní běžet. Tomuto procesu se říká relokace. Zároveň musí být nastaven registr procesoru obsahující adresu vektorů přerušení. Relokace se používá například v případě, kdy je třeba smazat a přepsat obsah paměti Flash. Logicky pak nemůže být smazán ten sektor paměti, ze kterého je vykonáván kód provádějící smazání (nebo to může vést k nedefinovanému stavu).

5.2.3 Počítání v plovoucí řádové čárce

Chybějící jednotka výpočtů v plovoucí řádové čárce, FPU, představuje razantní zpomalení zpracování čísel s plovoucí řádovou čárkou. Při použití datových typů `float` nebo `double` musí být k programu připojen kód pro softwarové zpracování výpočtů v plovoucí řádové čárce, což navyšuje velikost programu. U malých programů, a zejména pak na malých procesorech, může být tento nárůst razantní.

Napsal jsem jednoduchý program pro Nucleo F031K6, který inkrementuje a násobí dvě bezznaménková 16bitová čísla, a pokud je jejich součet menší, než konstanta, změní stav výstupního pinu. Druhá varianta pak tyto operace provádí nad 16bitovým číslem a datovým typem `float`. Program byl přeložen pro procesory STM32F031K6 bez FPU (emulována softwarově) a STM32L476VG s hardwarovou FPU. Porovnání velikostí se nachází v tabulce 5.2.

Tabulka 5.2: Nárůst velikosti programu při použití výpočtů v plovoucí řádové čárce

	Celočíselné operace	Plovoucí řád. č.	Rozdíl
STM32F031K6	768 B	1.552 B	+784 B
STM32L476VG	1.828 B	1.856 B	+28 B

Rozdílná velikost je způsobena rozdílným inicializačním kódem, větší tabulkou vektorů přerušení a inicializací většího prostoru pro dynamicky alokovanou paměť. Velikost připojených rutin pro emulaci (před optimalizacemi překladačem) je podle linker mapy 1.410 B.

5.3 Operační systémy pro vestavný systém

Cílem této práce bylo vybrat pro porovnání 3 operační systémy určené pro Internet věcí. Bylo nutné brát v úvahu možností hardware, jeho limitace, zejména pak kapacitu paměti a architekturu. Počáteční seznam jsem sestavil na základě doporučení vedoucí práce a Hahm et al. (2016), kteří se zabývali přehledem operačních systémů pro IoT, omezil jsem se však pouze na systémy s otevřeným zdrojovým kódem.

Tabulka 5.3: Potenciální operační systémy pro IoT

	uClinux	Contiki	TinyOS	RIOT OS	FreeRTOS	ARM mbed	
						Verze 2.0	Verze 5.4
Jazyk	C	C	nesC	C, C++	C	C, C++	
Vícevláknový	●	●[1]	●	●	●	-	●
RTOS	-	-	-	●	●	-	●
Plánovač	Preemptivní	Kooperativní	Kooperativní	Preemptivní	Preemptivní	-	Preemptivní
HAL	●	●	?	●	-	●	●
Minimum ROM	> 600 kB	< 30 kB	< 4 kB	~5 kB	5-10 kB	?	?
Minimum RAM	> 200 kB	< 2 kB	< 1 kB	~1,5 kB	~0,5 kB[2]	?	?

(●) podporováno, (-) nepodporováno, (?) nepodařilo se zjistit.
[1] Nepoužívá plnohodnotná vlákna s vlastním zásobníkem, ale pseudo-vlákna Protothreads
[2] Pouze jádro systému bez zásobníku pro jakékoliv vlákno

Nároky na paměť ROM a RAM uvedené v tabulce 5.3 jsou orientační a vycházejí z informací uvedených na webových stránkách jednotlivých projektů. V případě uC-linuxu se jedná o nároky minimální konfigurace pro procesor STM32F429 (Emcraft Systems).

5.3.1 Výběr systémů

Přestože je pro aplikace Internetu věcí důležitá podpora síťových protokolů, ať už 6LoWPAN, nebo IPv6, při výběru systému pro testovací aplikace na jejich podporu nebyl kladen důraz. Komunikace aplikace by měla probíhat po co nejjednodušším protokolu.

Na operační systém pro vybraný hardware byly kladeny následující požadavky:

- Musí být schopen běžet na hardware bez MMU, kterým není vybaven ani jeden z procesorů.
- Musí být přeložitelný s toolchainem GNU ARM Embedded Toolchain, dostupný z: `https://launchpad.net/gcc-arm-embedded`.
- Musí mít minimální nároky na paměť
- Musí poskytovat abstrakci nad hardware – HAL.

Dle těchto kritérií jsem hned z kraje vyloučil **uClinux**, který sice nevyžaduje pro běh MMU, ale jeho paměťové nároky se stále pohybují příliš vysoko. **TinyOS** by měl mít ze všech systémů nejnižší nároky na paměť, používá ale dialekt jazyka C s vlastním překladačem, nesC, čímž znemožňuje využití známých nástrojů pro překlad a ladění, jako je toolchain GCC a ladící nástroj GDB. **FreeRTOS** je jedním z nejznámějších RTOS, nemá ale žádné uniformní API nad hardware. Každý

jeho přenos pro cílovou platformu vyžaduje napsání vlastních ovladačů hardware, případně využití HAL od výrobce, pokud jej dodává.

5.3.2 RIOT OS

Prvním vybraným systémem byl RIOT OS pro nízké deklarované nároky na paměť a podporu podobných procesorů od STM a Freescale, ze které jsem předpokládal o něco jednodušší proces přenosu na procesory vybraných platforem.

Historie

Systém vyvíjený od roku 2013 byl postaven na základech jádra FireKernel, vyvíjeného pro bezdrátové senzorové sítě. Chlubí se přepnutím kontextu v méně než 100 instrukčních cyklech na architektuře ARM, malými nároky na paměť, událostmi (z přerušení) řízeným plánovačem, který nevyžaduje časovač pro preempci, a podporou dynamické alokace paměti (Hahm et al., 2016). Samotné jádro ale striktně využívá pouze staticky inicializovanou paměť.

Shrnutí

- Mikrojádro s oddělenými moduly platformě závislého kódu pro procesor a jeho periferie, to vše implementující jednotné API. Stejně tak jsou oddělené moduly ovladačů externích periferií, jako jsou rádiové čipy, senzory, apod.
- Jádro systému napsané v jazyce C. Deklaruje téměř konstantní čas přepnutí kontextu.
- Podpora pro C++.
- Oddělené moduly ovladačů senzorů, rádií a síťové vrstvy.
- Nízké deklarované nároky na paměť.
- Podpora pro přepínání do režimů nízké spotřeby v podobě API.

5.3.3 ARM mbed

Dalšími vybranými systémy byly mbed 2.0 a 5.4, které zaštiťuje samotná společnost ARM. Systém se orientuje na procesory ARM. Na rozdíl od mnoha jiných systémů, mbed certifikuje hardware splňující požadavky systému logem „mbed Enabled“.

Historie

Vývoj mbed započal v roce 2009. Původně byla platforma mbed SDK šířena formou již přeložené knihovny. K otevření mbed SDK pod licencí Apache 2.0 došlo v roce 2013. V roce 2014 se mbed SDK (verze 2.0) transformovalo do mbed OS (verze 3.0) a v roce 2016 došlo ke spojení obou verzí do řady 5, která je aktivně vyvíjena.

Zajímavostí je, že mbed je dostupný také ve formě online vývojového prostředí, ve kterém lze program napsat, přeložit a stáhnout. Certifikované desky se po připojení hlásí jako paměťové médium, do kterého stačí program zkopírovat a tím jej naprogramovat.

Pro pár vybraných procesorů je dostupná také knihovna uVisor, která přináší podporu pro ochranu paměti a možnosti vytváření boxů pro kód. Tyto boxy pak mají přístup k paměti a periferiím limitovaný podle programátorem předem daných pravidel. Komunikace mezi boxy a sdílení dat je možné pouze programátorem zveřejněným rozhraním.

Shrnutí

- Vybraný hardware je certifikován a podporován v základu. Teoreticky bez nutnosti zasahovat do kódu operačního systému a jeho ovladačů.
- Online vývojové prostředí.
- Podpora pro C++.
- Vrstva abstrakce hardware implementována v jazyce C i C++ v podobě tříd.
- Systém je doplnitelný o knihovnu uVisor pro ochranu paměti a přístupu k hardware s podporou pro privilegovaný a neprivilegovaný kód.
- Vývoj systému svou značkou zaštiťuje ARM Ltd.

6 Návrh řešení

Cílem práce je pokusit se na třech vybraných vývojových kitech (viz 5.1) zprovoznit vybrané operační systémy RIOT OS a ARM mbed ve verzích 2.0 a 5.4 a implementovat testovací úlohu.

6.1 Testovací úloha

Pro test desek a operačních systémů byla navržena testovací úloha implementace jednoduchého uzlu senzorové sítě. Ta se skládá ze dvou funkčních částí:

Senzorový uzel ke kterému bude připojen senzor teploty a rádiový modul. Uzel by měl po spuštění začít periodicky měřit teplotu vzduchu a tu bezdrátově odesílat koncentrátoru.

Koncentrátor který bude mít stejný rádiový modul a navíc bude připojen k PC. Bude sloužit jako přijímač a měřenou teplotu bude odesílat do počítače k zaznamenání.

6.2 Postup řešení

Pro začátek se pokusím o zcela základní oživení jednotlivých desek, nahrání jednoduchého programu, vyzkoušení ladícího programu pro krokování aplikace, čtení paměti a registrů. Tento krok bude vyžadovat stažení a konfiguraci vývojového prostředí, toolchainu a ovladačů pro jednotlivé ladící rozhraní.

Pro implementaci samotné úlohy bude třeba:

- Vybrat vhodný senzor teploty vzduchu.
- Vybrat vhodný rádiový modul pro senzorový uzel.
- Vybrat druhý kus hardware se stejným rádiovým modulem, který bude schopen přijímat data a předávat je jakýmkoliv způsobem PC.
- Navrhnout zapojení, navrhnout a implementovat software senzorového uzlu pro vybrané systémy a vybrané desky a software pro přijímač/koncentrátor.

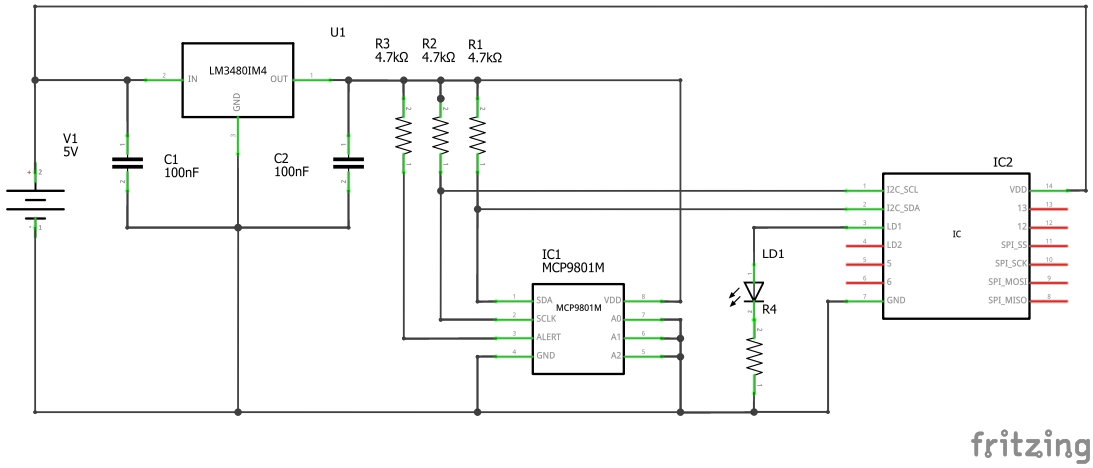
6.2.1 Použitý hardware

Z nerealizovaného projektu mi zbyl teplotní senzor **Microchip MCP9801**. Jedná se o digitální senzor komunikující po sběrnici I²C (někdy označované jako 2-Wire). Mezi další možné teplotní senzory, které by bylo možné využít, patří například **Dallas 18B20**, taktéž digitální, využívající sběrnici 1-Wire. Stejně tak by bylo možné využít analogový senzor **TMP36**. Tam už je nutné měřenou teplotu vzorkovat pomocí A/D převodníku samotného procesoru.

Sběrnice 1-Wire senzoru Dallas 18B20 by vyžadovala plně softwarové řízení komunikace. Analogový senzor TMP36 by pro tuto aplikaci postačoval, nakonec jsem se ale rozhodl využít MCP9801. Všechny procesory na vybraných deskách mají pro řízení sběrnice I²C periferii. Senzor navíc umožňuje nastavit teploty pro alarm, po jejichž dosažení může vyvolat přerušení procesoru, přestože tato funkce nebude využita.

6.3 Schéma zapojení úlohy

Na obrázku 6.1 je schéma zapojení teplotního senzoru k desce, ta je reprezentována zjednodušenou součástkou MCU. Na všech deskách je osazena LED, která bude aplikací využita, proto se nachází i ve schématu.



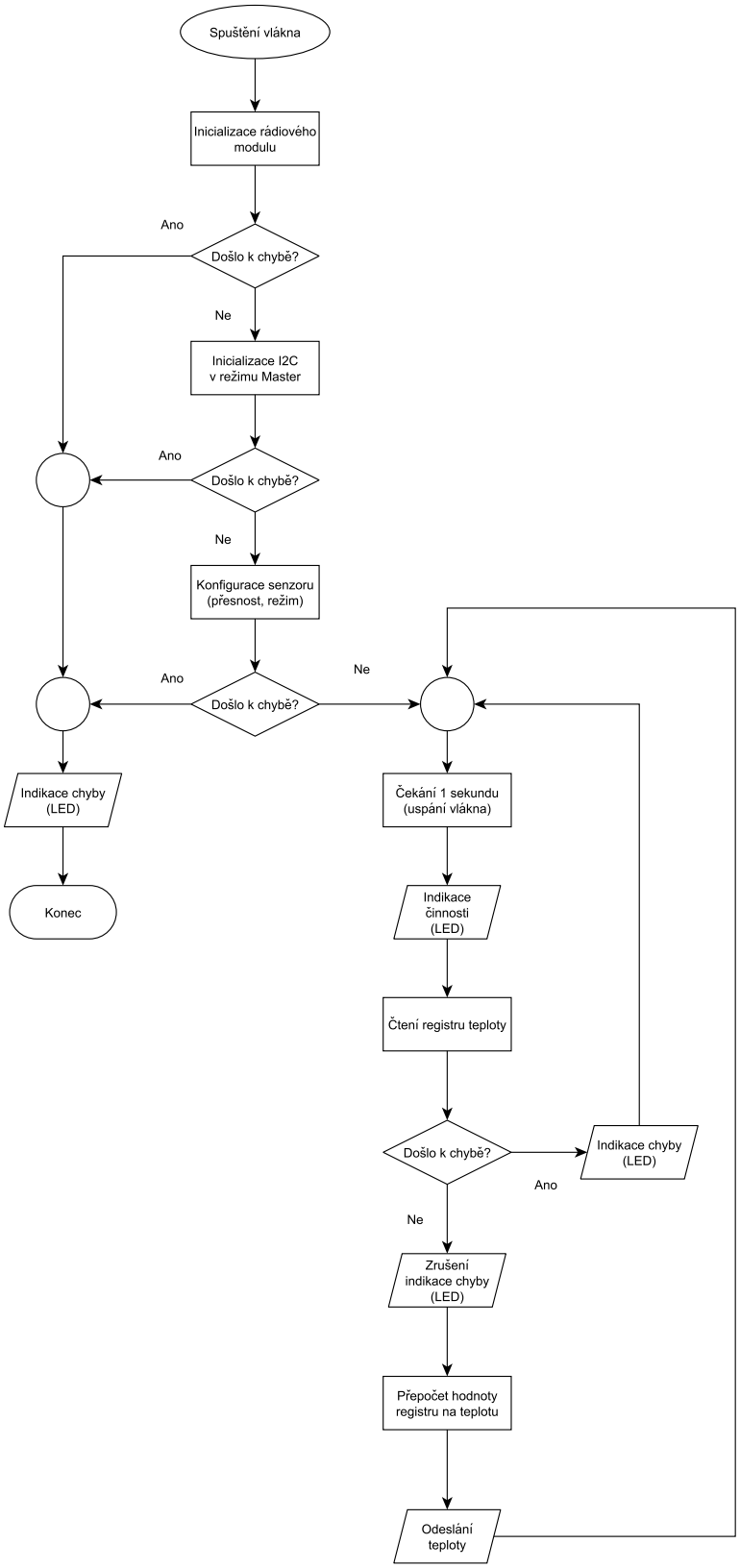
Obrázek 6.1: Schéma zapojení se senzorem MCP9081

- Datové vodiče `SDA` a `SCL` sběrnice I²C a `ALERT`, ač nevyužitý, musí být opatřeny pull-up rezistory (Microchip Technology Inc., 2010).
- Součástí zapojení se později stal také regulátor napětí z 5 V na 3.3 V, jelikož jsem se u desky FRDM KL05Z potýkal s problémy při komunikaci a výstupní napětí na pinu 3.3 V bylo kolem 2.9 V, což je napětí blízko spodní hranici napájecího napětí pro senzor.

6.4 Návrh aplikace

Aplikace senzorového uzlu je popsána vývojovým diagramem na obrázku 6.2.

1. Všechny desky disponují alespoň 1 LED, která bude použita pro indikaci činnosti střídavým blikáním.
2. Další LED bude použita pro indikaci chyby inicializace nebo chyby komunikace.
3. Cyklus čtení teploty bude nekonečný.
4. Komunikace pomocí rádiového modulu bude jednosměrná a tedy nepotvrzovaná.



Obrázek 6.2: Vývojový diagram aplikace měřící teplotu

7 Řešení

Ze všeho nejdříve bylo třeba nainstalovat veškeré nutné softwarové vybavení a ovladače pro oživení jednotlivých desek.

Pro vývoj používám počítač s operačním systémem Microsoft Windows 10. Tomu je uzpůsobena volba software a popis kroků k jeho instalaci, nebo konfiguraci.

7.1 Instalace software

7.1.1 Ovladače

Všechny tři desky jsou vybaveny rozhraním USB. Po připojení k počítači jsou registrována tři zařízení:

- Virtuální port COM, na který je skrz ladící rozhraní přesměrována periferie UART.
- Úložiště, na kterém je HTML soubor vedoucí na stránky ARM mbed. V případě desky FRDM-KL05Z slouží toto úložiště zároveň pro nahrávání programu metodou táhni a pusť.
- Ladící rozhraní, ST-LINK v případě desek STM, nebo OpenSDA u Freescale.

Zatímco s rozpoznáním prvních dvou zařízení nemá operační systém Windows větší problém, ovladače ladících rozhraní ST-LINK a OpenSDA je zpravidla nutné stáhnout a nainstalovat ručně, a to ze stránek výrobců.

Po instalaci ovladačů bylo v mém případě potřeba přiřadit ve *Správci zařízení* správný ovladač ručně.

7.1.2 Ladící rozhraní

Pro ladění je potřeba program komunikující se samotným zařízením. Ten umožňuje připojit se skrz ladící rozhraní k procesoru, číst a zapisovat do jeho paměti, včetně paměti Flash, a krokovat aplikaci.

- Rozhraní ST-LINK je podporováno programem **OpenOCD**, který je ke stažení na GitHubu projektu *GNU ARM Eclipse*, dostupném z adresy: `https://github.com/gnuarmeclipse/openocd/releases`

- Rozhraní OpenSDA na desku FRDM-KL05Z jsem přehrál alternativním firmware (sekce 7.1.10), a tak z něj udělal rozhraní kompatibilní s debuggerem **J-Link**, které využívá stejnojmenný program od firmy SEGGER. Ten je ke stažení na stránkách firmy SEGGER z adresy:
`https://www.segger.com/downloads/jlink`

7.1.3 Toolchain

Všechny platformy jsou osazeny procesory architektury ARM. Pro překlad programu je třeba křížový překladač. Použil jsem pro systém Windows dostupný *GNU ARM Embedded Toolchain* ve verzi 5.4.1 20160919 (dostupné v době instalace).

7.1.4 Nástroje pro sestavení

RIOT OS používá pro sestavení sadu Makefile skriptů, které vyžadují Linuxový program *make*. Pro systém Windows lze *make* získat, spolu s dalšími nástroji z Linuxu, v rámci MSYS2 nebo například přibalený k toolchainu MinGW.

Důležité je, aby byl adresář s programem *make* přidán do uživatelské proměnné *Path* a bylo jej možné spustit z jakéhokoliv adresáře.

7.1.5 Eclipse CDT

Jako vývojové prostředí jsem se rozhodl využít Eclipse CDT ve verzi Neon.1. Jedná se o multiplatformní prostředí, verze CDT je určena pro vývoj C/C++ aplikací, bez dalších zásuvných modulů ale není připravena pro vývoj software pro vestavné systémy.

Vývojovému prostředí Eclipse CDT chybí:

- Podpora pro toolchain použitý ke křížovému překladu.
- Schopnost vzdáleného ladění pomocí debuggerů ST-LINK a OpenSDA, kterými jsou desky vybaveny.
- Možnost prohlížení registrů procesoru, a to jak obecných registrů, tak speciálních funkčních registrů a registrů periferií.

RIOT OS i ARM mbed používají pro překlad svůj vlastní sestavovací program, samotný překlad pomocí Eclipse tak není klíčový.

Schopnost ve vývojovém prostředí krokovat na desce běžící program a mít k tomu možnost prohlížet registry procesoru, představuje pro vývojáře obrovskou výhodu. Ladění komplikovanějších programů na jednoduchých deskách typu Arduino, které nejsou vybaveny ladícím rozhraním, je komplikované až téměř nemožné.

7.1.6 Zásuvné moduly do Eclipse CDT

Všechny chybějící funkce je možné přidat se zásuvnými moduly projektu *GNU ARM Eclipse*. Potřeba je zejména modul podpory OpenOCD debuggeru pro komunikační rozhraní ST-LINK, ale také podpora pro SEGGER J-Link, viz 7.1.10.

Postup instalace je následující

1. Otevřít *Help > Install New Software*.
2. Přidat adresu repozitáře modulu klikem na tlačítko *Add*.
3. *Location* vyplnit `http://gnuarmeclipse.sourceforge.net/updates`, vyhledat balíčky a zvolit:
   - GNU ARM C/C++ Cross Compiler
   - GNU ARM C/C++ Cross J-Link Debugging
   - GNU ARM C/C++ Cross OpenOCD Debugging
   - GNU ARM C/C++ Packs (Experimental)
4. Cestu k nainstalovanému toolchainu nastavte v *Window > Preferences > C/C++ > Build > Global Tools Paths*. Pokud byl toolchain nainstalovan například do *C:/ARM-GCC*, pak *Toolchain folder* musí být nastaven na *C:/ARM-GCC/bin*.
5. Pokud jste instalovali program OpenOCD, nastavte v *Run/Debug > OpenOCD* cestu k adresáři *bin* ve složce instalace. Jako *Executable* nastavte *openocd.exe*.
6. Pokud jste instalovali program J-Link, nastavte v *Run/Debug > SEGGER J-Link* kořenovou složku instalace programu. Jako *Executable* nastavte *JLinkGDBServerCL.exe*.

7.1.7 Stažení definic procesorů

Zásuvný modul zobrazující periferie a registry procesoru potřebuje soubory s jejich definicemi. Jejich stažení provedeme v Eclipse následovně:

1. Otevřeme *Window > Perspective > Open Perspective > Other* a vybereme *Packs*.
2. Na záložce *Devices* najdeme výrobce procesoru a po rozbalení jsou zobrazeny modelové řady.
3. Po výběru řady se v prostřední části okna zobrazí dostupné sdružené balíčky. Ty je dále možné rozbalit a zobrazit konkrétní verze.
4. Kliknutím pravým tlačítkem lze balík stáhnout a nainstalovat volbou *Install*.

7.1.8 Ostatní software

Mezi další důležitý software patří:

- Verzovací systém **Git** je třeba pro stažení a k aktualizaci repozitáře RIOT OS.
- Verzovací systém **Mercurial** je používán programem *mbed CLI* systému ARM mbed pro aktualizaci.
- Python 2.7 pro *mbed CLI*.

7.1.9 Podpora pro procesory

Důležitá byla také podpora pro jednotlivé procesory. Výrobci zpravidla dodávají kromě datasheetů a důležitých referenčních manuálů také hlavičkové soubory (v jazyce C) s deklaracemi a definicemi struktur periferií procesoru a jeho registrů.

K vyhledání je třeba znát přesné označení procesoru. Například řada procesorů STM32F0 se skládá z modelů s různými pouzdry, a tedy i počty pinů, množstvím periferií a velikostí pamětí. Naštěstí ale stačí najít produktovou stránku desky, kde jsou ke stažení příklady a právě i podpora.

Nucleo F031K6

- Referenční manuál STMicroelectronics (2017b)
- Podpora ve formě *STM32CubeF0* ke stažení ze stránek společnosti STMicroelectronics v sekci věnované desce.

Discovery L476G

- Referenční manuál STMicroelectronics (2017d)
- Podpora ve formě *STM32CubeL4* ke stažení ze stránek společnosti STMicroelectronics v sekci věnované desce.

FRDM-KL05Z

- Referenční manuál Freescale Semiconductor, Inc. (2012)
- Podpora ve formě *FRDM-KL05Z Sample Code Package (Rev. 3)* ke stažení ze stránek společnosti NXP v sekci věnované desce.

7.1.10 Alternativní firmware pro ladící rozhraní

V zaměstnání používáme komerční debuggery **SEGGER J-Link**. Firma SEGGER poskytuje pro některé vývojářské desky alternativní firmware, které z jejich ladících rozhraní udělají rozhraní kompatibilní s debuggerem J-Link. Pro všechny vybrané desky je tento alternativní firmware dostupný.

Motivací pro jeho využití byla chybějící podpora rozhraní OpenSDA na desce FRDM-KL05Z programem OpenOCD. V případě desek STM s rozhraním ST-LINK pak chybějící podpora desky Discovery L476G programem OpenOCD, která přišla až ve verzi 0.10.0 dostupné koncem ledna 2017.

Přehráním firmware se ladící rozhraní začne reprezentovat jako SEGGER J-Link a je možné použít stejnojmenné ovladače a software. Při práci s některými deskami, například STM Discovery F429i, dosahuje tento firmware vyšších rychlostí přepisu paměti Flash a subjektivně rychlejšího krokování při ladění programu.

7.2 RIOT OS

První jsem začal s přenosem operačního systému RIOT OS. Vývoj probíhal na verzi z 21. listopadu 2016. K tomuto datu neměl systém přímou podporu pro žádnou z vybraných desek, existovala ale podpora pro jiné řady na deskách osazených procesorů. To, jak jsem doufal, by mohlo do jisté míry proces přenosu usnadnit.

7.2.1 Stažení

Ke stažení projektu je třeba mít nainstalován verzovací systém Git. Pak už stačí pouze otevřít konzoli a naklonovat repozitář projektu na disk:

```
git clone https://github.com/RIOT-OS/RIOT.git RIOT
```

Systém je nyní stažen ve složce *RIOT/*. Jeho adresářová struktura je následující:

boards
: Moduly s definicemi pro podporované desky. Deska má přiřazen modul procesoru a obsahuje konfiguraci periferií a pinů. Součástí modulu desky je inicializační kód, který konfiguruje na desce osazené vstupy a výstupy, například tlačítka a LED. Při překladu projektu je vybírán cíl, kterým je právě některý z modulů desek.

core
: Obsahuje zdrojové kódy jádra systému.

cpu
: Moduly s definicemi pro podporované procesory. Tyto moduly obsahují ovladače pro periferie procesoru, kód inicializace procesoru (inicializace hodin a periferií), definice vektorů přerušení a skripty pro linker.

dist
: Podpůrné nástroje, skripty Makefile pro ladící rozhraní.

drivers
: Moduly ovladačů pro (externí) periferie, senzory, rádiové moduly a mnohé další.

examples
: Ukázkové aplikace postavené nad operačním systémem.

pkg
: Externí balíčky a knihovny, například síťový stack lwIP.

sys
: Moduly čítačů a časovačů, semafory, dynamický alokátor paměti, souborový systém, rozhraní POSIX a další.

7.2.2 Příprava prostředí

Začal jsem vytvořením nové aplikace ve složce *examples*, a to z kopie složky *hello-world*. Tu jsem pojmenoval *playground*, jelikož sloužila pro testování.

Vytvoření projektu v Eclipse

Abych mohl celý projekt spravovat v Eclipse CDT, bylo nutné vytvořit projekt. Samotný RIOT OS má v Makefile podporu pro vytvoření konfiguračního XML souboru pro Eclipse, který importujeme do projektu. Ten obsahuje definice maker, a složky, ve kterých bude překladač vyhledávat hlavičkové soubory, označované jako *include directories*.

K jeho vytvoření je třeba spustit ve složce projektu následující příkaz:

```
make BOARD=nazev_desky eclipsesym
```

Když už máme vytvořen *eclipsesym.xml*, můžeme vytvořit vlastní projekt v Eclipse CDT:

1. Vytvořte projekt v *File > New > Makefile Project with Existing Code*
2. Vyplňte jméno projektu a jako *Existing Code Location* zvolte kořenovou složku RIOT OS
3. Projděte zbylými kroky průvodce a otevřete vlastnosti projektu skrz *Project Explorer* v levé části okna.
4. V *C/C++ Build* nejdřív zvolte *Manage Configurations* a vytvořte novou konfiguraci pro desku. Konfigurace představuje nastavení projektu, konfiguraci překladače, makra preprocesoru a složky pro hledání hlavičkových souborů. Pro každou desku vytvořím novou konfiguraci projektu, jelikož se liší právě makry preprocesoru a proměnnými, které nastavíme v dalším kroku. První konfiguraci jsem pojmenoval *Nucleo F031K6*.
5. Na záložce *Builder Settings*:
    - Ověřte, že není zaškrtnuto *Use default build command*

- Nastavte hodnotu *Build command* na:
  `${cross_make} BOARD=${RIOT_BOARD}`
- Nastavte hodnotu *Build directory* na:
  `${workspace_loc:/RIOT}/${RIOT_APPLICATION}`

6. V *C/C++ Build > Build Variables* vytvořte následující proměnné:
   - `RIOT_APPLICATION` s hodnotou odpovídající názvu složky projektu, v mém případě `playground`
   - `RIOT_BOARD` s hodnotou odpovídající názvu složky, ve které je definice desky, v mém případě to bude `nucleo-f031`
7. V *C/C++ Build > Settings > Toolchains* zvolte *GNU Tools for ARM Embedded Processors*, zkontrolujte, zda *Toolchain path* odkazuje do podsložky *bin* ve složce, kde máte nainstalován toolchain.
8. Na záložce *Devices* zvolte procesor, pro Nucleo F031K6 je to *STM32F031K6*. V tomto okně jsou vidět jen procesorové řady instalované skrze zásuvný modul *Packs*, viz 7.1.7.
9. V *C/C++ Build > Paths and Symbols* importujte výše vytvořeno XML soubor s konfigurací tlačítkem *Import Settings*.

7.2.3 Překlad projektu

V levé části okna otevřete složku *RIOT/examples/playground* a rozbalte položku *Build Targets*, která má následující prvky:

build Přeloží projekt pro desku nastavenou ve vlastnostech projektu proměnnou `RIOT_BOARD`. Soubory překladu a výstupní binární soubor určený k nahrání do desky jsou v podsložce projektu *bin/nazev_desky/*.

clean Odstraní soubory překladu a vynutí tak čistý překlad.

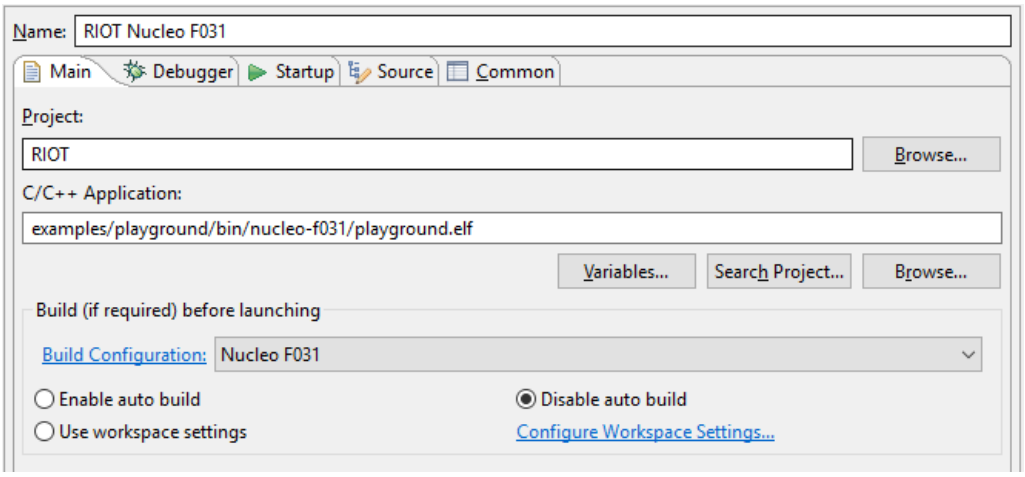
flash Použije program daný definicí desky (například OpenOCD nebo J-Link) pro nahrání výstupního binárního souboru aplikace do procesoru.

term Slouží pro otevření terminálu a připojení se k desce, na které běží program.

7.2.4 Vytvoření ladícího sezení

V kontextové nabídce projektu v *Debug As > Debug Configurations* se vytváří konfigurace pro ladění. Ta se dělí podle cílového ladícího rozhraní na J-Link a OpenOCD.

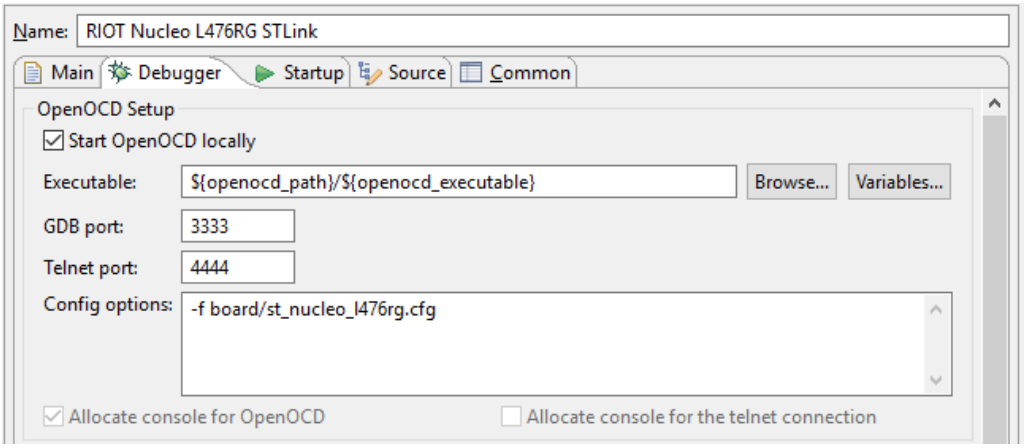
Společná konfigurace je na záložce *Main*, kde je třeba vybrat vytvořený projekt, a v *C/C++ Application* vyplnit relativní cestu k výstupnímu souboru. Viz obrázek 7.1.



Obrázek 7.1: Společná konfigurace ladícího sezení v Eclipse CDT

OpenOCD

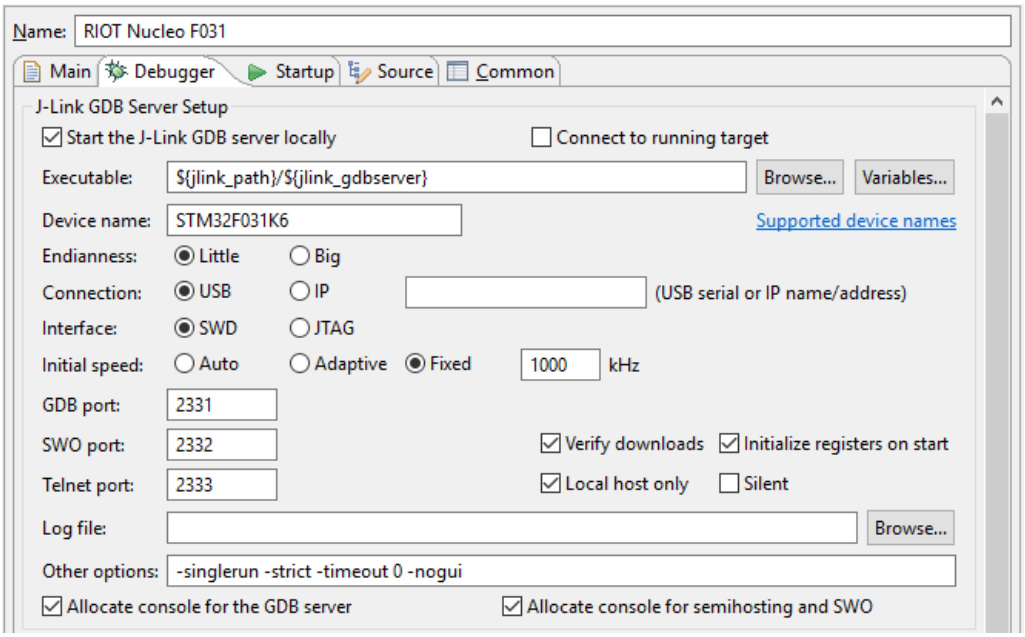
Pro OpenOCD je třeba na záložce *Debugger* nastavit správnou cestu ke skriptu desky, viz obrázek 7.2. OpenOCD 0.10 již podporuje obě desky od STM.



Obrázek 7.2: Konfigurace OpenOCD ladícího sezení v Eclipse CDT

J-Link

V případě J-Link na záložce *Debugger* nastavujeme pro změnu typ procesoru. Ten musíme vyčíst z datasheetu desky, případně přímo z pouzdra procesoru. Konfigurace je zobrazena na obrázku 7.3.



Obrázek 7.3: Konfigurace J-Link ladícího sezení v Eclipse CDT

7.2.5 Úprava sestavovacích skriptů systému

Sestavení RIOT OS pomocí Makefile pod Windows přináší hned několik problémů. Po dokončení překladu cílem *flash* (sekce 7.2.3) se v závislosti na definici desky vykonává jeden ze dvou skriptů. Jeden určený pro obsluhu OpenOCD, druhý pro J-Link. Umístění obou skriptů je následující:

- *dist/tools/jlink/jlink.sh* pro J-Link
- *dist/tools/openocd/openocd.sh* pro OpenOCD

Od obou skriptů jsem udělal kopie, které jsem upravil pro prostředí systému Windows.

J-Link

Skript pro J-Link má ve své výchozí podobě dva problémy, které znemožňují nahrání programu do desky.

1. Obsahuje nesprávné názvy programů J-Link a GDB Server.
2. Nevypořádá se s Windows formátem cesty k souboru.

První problém lze vyřešit **úpravou projektu** v Eclipse. Názvy programů jsou definovány na počátku skriptu a při volání skriptu je možné skriptu předat jejich alternativní hodnoty. Ve vlastnostech projektu v *C/C++ Build* na záložce *Builder Settings* přidejte na konec *Build command* následující řetězec:

```
JLINK=JLink.exe JLINK_SERVER=JLinkGDBServer.exe
```

Následně se vyskytl **druhý jmenovaný problém**. Skript spustí J-Link, který se připojí k desce a následuně skončí s chybou *Failed to open file*. Problém je ve formátu cesty k souboru, která je předávána programu J-Link, ta má Linuxový tvar.

Bylo potřeba skript upravit tak, aby cestu formátu */d/Workspace/RIOT* převedl na *D:/Workspace/RIOT*, čehož jsem docílil vytvořením nové proměnné, která byla předávána programu J-Link.

```
# Fix path by replacing "/d/Path/To/File.bin" with "d:/Path/To/File.bin"
# It is used in burn.seg file
JLINK_HEXFILE=$(echo ${HEXFILE} | sed 's/^\/\([a-z]\)/\1:/g')
```

Ukázka kódu 7.1: Oprava formátu cesty k souboru pro J-Link

OpenOCD

U OpenOCD byl stejný problém s cestou k souboru k flashování. Vyzkoušel jsem stejnou úpravu cesty, jako v ukázce 7.1, výsledkem ale byla chybová hláška:

```
Error: couldn't open d;C:MinGWmsys?.0WorkspaceRIOTexamplesplayground?
inucleo-l476playground.hex
```

Řešením tohoto problému bylo zdvojit lomítka v cestě.

```
# Fix path by replacing "/d/Path/To/File.bin" with "d://Path//To//File.bin"
# when running in Windows environment
HEXFILE=$(echo ${HEXFILE} | sed 's/^\/\([a-z]\)\(.*\)/\1:\2/g' \
  | sed 's/\//\/\//g')
```

Ukázka kódu 7.2: Oprava formátu cesty k souboru pro OpenOCD

7.2.6 Implementace testovací aplikace

Nejdříve jsem implementoval zjednodušenou verzi aplikace (viz kap. 6.4), která namísto rádiového modulu měřenou teplotu odesílala na UART.

RIOT OS po spuštění aplikace vytvoří dvě vlákna. Hlavní vlákno, které spouští kód funkce `main`, a vlákno s nízkou prioritou, jehož úlohou je přepnutí procesoru do co nejhlubšího spánku v závislosti na používaných periferiích. Implementace usínání je platformě závislá a pro použité procesory nebyla k dispozici.

Zkrácený zdrojový kód této aplikace je v ukázce C.1.

7.2.7 Přenos pro Nucleo F031K6

Procesor STM32F031K6 spadá do rodiny procesorů STM32F0, pro které měl systém podporu, včetně ovladačů pro některé z periferií. Tato konkrétní varianta procesoru ale podporována nebyla.

Přidání linker scriptu

Podle vzoru procesoru STM32F030R8 jsem vytvořil linker script (viz 4.4.1). Liší se velikostmi pamětí RAM a Flash reprezentovanými sekcemi `ram` a `rom`, které používá RIOT OS.

Další sekce, zejména vektory přerušení, jsou vkládané ze sdíleného linker scriptu pro architekturu ARM Cortex-M, umístěného v modulu *cortexm_common* v souboru *ldscripts/cortexm_base.ld*.

Soubor linker scriptu je umístěn v *cpu/stm32f0/stm32f031k6.ld*.

Definiční soubor procesoru

Dále bylo nutné přidat hlavičkový soubor *stm32f031x6.h* z balíku STM32CubeF0 s definicemi periferií procesoru. Musel jsem udělat drobnou úpravu v tomto souboru, kvůli odlišnému pojmenování některých funkcí obsluhy přerušení.

Vytvoření modulu desky

Jako šablonu jsem využil existující modul desky Nucleo F030, *nucleo-f030*, a vytvořil nový modul *boards/nucleo-f031*.

V kořenovém adresáři jsem modifikoval skripty Makefile, jelikož měla deska nahraný SEGGER J-Link firmware a místo OpenOCD skriptu tedy bylo potřeba vkládat skript pro obluhu rozhraní J-Link. Pro ladění jsem navíc potřeboval pomocí parametrů (flagů) instruovat překladač, aby překládal s optimalizacemi programu pro ladění a nikoliv velikost.

Vstupy a výstupy

Deska je osazena jednou uživatelskou LED připojenou na port B, pin 3. Jelikož jsou LED přítomné na mnoha vývojářských deskách, definuje RIOT OS v hlavičkovém souboru *drivers/include/led.h* několik maker pro jejich ovládání:

- `LEDx_ON`
- `LEDx_OFF`
- `LEDx_TOGGLE`

Standardně tato makra, kde `x` je číslo diody od 0 do 7, nic nedělají. Pokud je deska osazena LED, pak jsou v jejím modulu tato makra definována a ovládají konkrétní piny procesoru. Tyto definice se nachází v souboru *include/board_common.h*.

Definice periferií

V souboru *include/periph_conf.h* se nachází konfigurace procesoru a jeho periferií. Na základě informací z datasheetu (STMicroelectronics, 2017a) a referenčního manuálu jsem konfiguroval:

- Taktovací frekvenci procesoru,
- periferii UART připojenou na virtuální COM port, na který bude přesměrován standardní výstup,
- a periferii I²C kde je připojen teplotní senzor MCP9801.

Tabulka 7.1: Konfigurace pinů desky Nucleo F031K6

Periferie	Funkce	Port	Pin	Alternativní funkce
USART1	RX	A	15	AF1
	TX	A	2	AF1
I2C0	SCL	B	6	AF1
	SDA	B	7	AF1

Tabulka 7.1 ukazuje, jak byly konfigurovány piny procesoru. Mnoho pinů procesoru je univerzálních, mohou být použity jako obecné vstupy nebo výstupy, případně mohou být přiřazeny různým periferiím. Pak mluvíme o alternativní funkci pinu. Některé piny mají až 8 alternativních funkcí, jejich tabulka je v datasheetu procesoru (STMicroelectronics, 2017a, str. 31).

Periferie I²C

Původně v modulu *stm32f0* chyběl ovladač periferie řídící sběrnici I²C. Jelikož mají procesory STM napříč řadami podobné, někdy dokonce stejné periferie, využil jsem ovladače I²C z modulu *stm32f3* s pár úpravami.

- Upravil jsem inicializační funkci sběrnice, `i2c_init_master`, aby bylo možné nastavit nastavit nejnižší rychlost sběrnice – 100 kHz.
- Opravil jsem přiřazení neexistující obsluhy přerušení pro událost chyby. Periferie má pro všechny události jediný vektor přerušení.

Inicializace desky

V souboru *board.c* je implementována funkce `board_init` sloužící k inicializaci desky. Její úlohou je:

1. Inicializovat vstupy a výstupy, jako jsou uživatelská tlačítka a LED.
2. Inicializovat procesor voláním funkce `cpu_init`. Tato funkce je implementována v modulu konkrétního procesoru, v tomto případě *stm32f0*.

7.2.8 Přenos pro FRDM-KL05Z

Na desce osazený procesor Freescale MKL05Z32 vychází z rodiny Kinetis. RIOT OS má podporu pro jiné procesory této rodiny, a tak jsem vyšel z ovladačů pro ně. Přesto byly potřeba další úpravy.

Modul procesoru

Nejdříve jsem vytvořil modul procesoru *cpu/mkl05z4*. Ovladače periferií čítače a časovače, I^2C a UART jsem převzal z modulu *kinetis_common* k dalším úpravám.

- Z balíku *FRDM-KL05Z Sample Code Package* (viz 7.1.9) jsem převzal hlavičkový soubor s definicemi pro procesor.
- Podle datasheetu jsem vytvořil část linker scriptu doplněného o upravený linker script z modulu *kinetis_common*.
- Podle šablony jiného Kinetis procesoru, konkrétně *k60*, a na základě hlavičkového souboru definice procesoru, jsem vytvořil tabulku vektorů přerušení.
- Přidal jsem speciální sekci v paměti Flash (Freescale Semiconductor, Inc., 2012, str. 389), kde se nachází například konfigurace zabezpeční modulu paměti Flash procesoru (Freescale Semiconductor, Inc., 2012, str. 419). Ta je naplněna ze souboru *flashcfg.c*. Tato sekce byla přidána také do linker scriptu.
- Upravil jsem skripty Makefile, ve kterých jsou vkládány moduly periferií, definován model procesoru a jeho řada.

Konfigurace hodin procesoru

Zatímco u jader STM32F0 již byla inicializace procesoru hotová, pro MKL05Z4 jsem implementoval vlastní. Ta sestává z funkce `cpu_init` a její úlohou je:

- Inicializovat procesorové jádro Cortex-M voláním `cortexm_init`.
- Nakonfigurovat taktovací frekvenci jádra procesoru, sběrnic a periferií.

Konfigurace hodin se provádí nastavením registrů periferie *Multi-purpose Clock Generator* (MCG), která umožňuje využít externí oscilátor, nebo interní oscilátor. Obvod FLL v závislosti na nastavení MCG z frekvence zvoleného oscilátoru generuje frekvenci, která dále je skrze děličky využívána právě jádrem procesoru, periferiemi a sběrnicemi.

```
/* Disable watchdog timer (Computer Operating Properly) */
SIM->COPC = 0;

/* Prescalers */
SIM->CLKDIV1 = SIM_CLKDIV1_OUTDIV1(0)    /* DIV_1 for Core, Platform
                                            and System clocks */
             | SIM_CLKDIV1_OUTDIV4(1);   /* DIV_2 for Bus and Flash
                                            clocks */

/* Switch to FEI (FLL Engage Internal) Mode */
MCG->C1 = MCG_C1_CLKS(0)         /* Clock source: FLL */
        | MCG_C1_FRDIV(0)        /* Reference divider: DIV_1
                                    (because C2_RANGE0 == 0) */
        | MCG_C1_IREFS_MASK      /* Internal reference is used */
        | MCG_C1_IRCLKEN_MASK;   /* Enable internal reference clock */

MCG->C2 = MCG_C2_RANGE0(0);      /* Low frequency range */

/* DMX32=1, DRST_DRS=1, reference 32768 kHz -> DCO 48 MHz */
MCG->C4 = MCG_C4_DMX32_MASK | MCG_C4_DRST_DRS(1);

/* Enable external reference clock */
OSC0->CR = OSC_CR_ERCLKEN_MASK;

/* Check that the source of the FLL reference clock is the internal
 * reference clock. */
while ((MCG->S & MCG_S_IREFST_MASK) == 0) { }
/* Wait until output of the FLL is selected */
while ((MCG->S & 0x0C) != 0) { }
```

Ukázka kódu 7.3: Inicializace hodin procesoru MKL05Z

Jelikož nemá deska externí oscilátor (je osazena krystalem, který je ale používán procesorem ladícího rozhraní), je potřeba nakonfigurovat MCG do režimu *FLL Engaged Internal* (Freescale Semiconductor, Inc., 2012, str. 346, 355), aby využíval interní oscilátor o taktu 32,768 kHz a z něj vytvořil frekvenci 48 MHz pro jádro procesoru a další frekvence pro periferie a sběrnice. Kód provádějící tuto inicializaci je v ukázce 7.3.

Ovladač GPIO

Pro tento procesor jsem napsal malý ovladač pro univerzální vstupně výstupní piny. Jeho funkčnost je omezená, neumožňuje pro vstupní piny nakonfigurovat přerušení, jinak je ale možné konfigurovat pin jako vstup (analogový nebo digitální) nebo výstup, připojit vnitřní pull-up nebo pull-down rezistory a nastavit alternativní funkci – přiřazení pinu periferii.

Úpravy ovladačů

Ovladače periferií, převzaté z modulu *k60* procesoru Kinetis K60, jsem musel pro procesor MKL05Z upravit, jednalo se například o:

- Ovladač UART, používající nesprávné registry. Procesor MKL05Z má úspornou variantu této periferie nazvanou *UARTLP* (low-power), navíc nedisponuje zásobníkem FIFO.
- Ovladač pro čítače a časovače využívá pouze periferii PIT (Periodic Interrupt Timer), přestože procesor dále obsahuje také periferii LPTMR (Low-power Timer). Kód ovladače musel být dále upraven, jelikož využíval například bit-banding (ARM Limited, 2010, str. 44), funkci tímto procesorem nepodporovanou.

Vytvoření modulu desky

Po dokončení modulu procesoru byl vytvořen modul pro desku v *boards/frdm-kl05z*. Jelikož bylo ladící rozhraní přehráno firmwarem J-Link, bylo opět nutné upravit skripty Makefile.

Na rozdíl od ostatních desek je FRDM-KL05Z osazena RGB LED, nemá tedy pouze jednu, ale tři diody. Makra pro jejich ovládání jsou opět definována v souboru *include/board.h*. Na které piny procesory jsou tyto diody připojeny je uvedeno v tabulce 7.2. Konfigurace desky se nachází v souboru *include/periph_conf.h* a obsahuje:

- Taktovací frekvence procesoru 48 MHz, v případě této desky daná konfigurací MCG,
- konfigurace periferie čítačů a časovačů PIT,
- UART, vyvedený na pinovou lištu,
- I^2C, použitý teplotním senzorem.

Tabulka 7.2: Konfigurace LED na desce FRDM-KL05Z

Barevný kanál	Port	Pin
Červená	B	8
Zelená	B	10
Modrá	B	9

Tabulka 7.3: Konfigurace pinů desky FRDM-KL05Z

Periferie	Funkce	Port	Pin	Alternativní funkce
UARTLP	RX	B	1	AF3
	TX	B	2	AF3
I2C0	SCL	B	3	AF2
	SDA	B	4	AF2

7.2.9 Přenos pro Discovery L476G

Procesor STM32L476G osazený na desce Discovery L476G neměl ze strany systému podporu. Při přenosu jsem vycházel z ovladačů pro procesory řady STM32F4, jejich periferie se ale vzájemně dost liší, řada STM32L4 je určená pro zařízení s nízkou spotřebou.

Modul procesoru

Vytvořil jsem modul *cpu/stm32l4*, jako šablonu jsem použil modul *stm32f4*.

- Hlavičkový soubor s deklaracemi registrů a periferií procesoru jsem převzal z *STM32CubeL4* (viz 7.1.9).
- Tabulků vektorů přerušení jsem vytvořil podle seznamu přerušení v ukázkovém projektu *STM32CubeL4*.
- Hlavičkový soubor *include/periph_cpu.h* byl z velké části přepracován kvůli podpoře jiného modelu DMA přenosů.

Konfigurace hodin

Notně přepracována musela být konfigurace hodin procesoru. K tomu slouží periferie *Reset and Clock Control* (RCC). Ta se mimo jiné používá také pro zapnutí jednotlivých periferií, včetně portů GPIO.

Procesor STM32L476G může jako zdroj hodinového taktu používat vnější oscilátor s frekvencí od 4 do 48 MHz, nebo jeden ze dvou vnitřních oscilátorů. Jeden s fixní frekvencí 16 MHz, označovaný jako HSI (High-speed Internal), a druhý s nastavitelnou frekvencí, označovaný jako MSI (Multi-speed Internal). Jako zdroj hodin pro watchdog a hodiny reálného času je tu ještě interní 32kHz oscilátor. K vytvoření taktů pro procesorové jádro, sběrnice a periferie je možné použít PLL (STMicroelectronics, 2017d, str. 182).

Implementována byla konfigurace využívající MSI oscilátor, nastavený na takt 4 MHz a kalibrovaný externím krystalem 32,768 kHz, která pomocí PLL a děliček generovala:

- 80 MHz pro procesorové jádro,
- 80 MHz pro vysokorychlostní sběrnice (AHB, APB2),
- 40 MHz pro nízkorychlostní sběrnici periferií (APB1).

V závislosti na taktovací frekvenci procesorového jádra je nutné konfigurovat také latenci paměti Flash – vkládání prázdných cyklů, nebo-li *wait states* (STMicroelectronics, 2017d, str. 83).

Dodaná konfigurace není vhodná pro bateriový provoz. V případě provozu na baterie je vhodné taktovací frekvenci snížit na nutné minimum, odpojit nepoužívané periferie a implementovat modul *lpm_arch.c* pro přepínání mezi stavy nízké spotřeby a uspávání procesoru.

Ovladače periferií

Jednotlivé ovladače převzaté z modulu *stm32f4* byly upraveny, aby byl modul přeložitelný, což znamenalo opravit všechny nesprávně pojmenované registry, nebo bitové masky periferií.

Struktura procesorů z řady STM32L4 se proti řadě STM32F4 liší (STMicroelectronics, 2017c). Ať už je to rozdělení paměti, dvě sběrnice pro periferie, periferie odlišně implementované nebo s jinými registry, a přepracované kanály pro přímý přístup do paměti (DMA).

- Ovladače AD převodníku a GPIO byly upraveny, jelikož jsou periferie připojené na jiné sběrnici a používají jiné registry.
- Pro periferii I^2C byla přidána podpora rychlosti 100 kHz.

Ovladač UART

Procesory řady STM32L4 mají některé periferie přepracované, jsou komplexnější, mají rozdílné registry a masky, a především řízení přímého přístupu do paměti (DMA) je u řady STM32L4 přepracované. A jelikož ovladač periferie UART pro řadu procesorů STM32F4, ze kterého jsem vycházel, DMA využíval, bylo nutné jej upravit.

Procesor obsahuje dva DMA kontroléry po 7 kanálech a ke každému z nich je pevně přiřazeno až 7 periferií. V registrech DMA kanálu je vybrána periferie, směr komunikace (čtení z periferie do paměti, zápis z paměti do periferie), velikost a množství dat, priorita a další parametry (STMicroelectronics, 2017d, str. 298).

Periferie UART podporuje přenosy pomocí DMA v obou směrech, je tedy možné přijímat data do předem alokované paměti, nebo je z paměti odesílat. K obojímu je ale nutné předem znát jejich velikost, proto ovladač implementuje pouze odesílání pomocí DMA přenosů.

Byly implementovány základní funkce pro podporu DMA kanálů jako náhradu za DMA proudy (streams) implementované v řadě STM32F4. Přepracována byla konfigurace kanálů při odesílání dat, obsluha přerušení DMA kanálů a konfigurace rychlosti rozhraní – baud rate.

Ovladač SPI

Upraven musel by také ovladač SPI použitý s rádiovým modulem. Byla upravena tabulka děliček pro všechny podporované rychlosti sběrnice a inicializační funkce, kde je nově konfigurována velikost dat a režim komprese dat v přijímací frontě. Periferie totiž podporuje přenosy od 4 do 16 bitů a pokud je velikost rámce menší nebo rovna 8 bitům, je použit *Data packing*. Při příjmu lichého počtu takto malých rámců, musí být nastaven zvláštní příznak, aby byla generována událost příjmu (a vyvoláno přerušení, pokud bylo konfigurováno) a bylo zachováno správné zarovnání ukazatelů na frontu dat.

Při ladění komunikace se právě 8bitové rámce ukázaly jako problematické, a to kvůli automatickému sdružování dat (*Data packing*). Při zápisu 8bitového rámce do datového registru periferie docházelo k odeslání 16 bitů. Tento problém je dále rozebírán a řešen v kapitole 7.5.2.

Modul desky

Vytvořil jsem pro desku modul *boards/stm32l4discovery*. Stejně jako u předchozích desek, i pro tuto byly upraveny skripty Makefile, jelikož bylo ladící rozhraní přehráno firmwarem J-Link.

Deska je osazena dvěma uživatelskými LED a 5 tlačítky v podobě joysticku. Jejich zapojení je uvedeno v tabulce 7.5. Konfigurace pinů pro sběrnice je uvedena v tabulce 7.4.

Tabulka 7.4: Konfigurace pinů desky Discovery L476G

Periferie	Funkce	Port	Pin	Alternativní funkce
USART2	RX	A	3	AF7
	TX	A	2	AF7
I2C1	SCL	B	6	AF4
	SDA	B	7	AF4

Tabulka 7.5: Konfigurace vstupů a výstupů na desce Discovery L476G

Vstup/Výstup	Port	Pin
LED 0	B	2
LED 1	E	8
Joystick střed	A	0
Joystick vlevo	A	1
Joystick nahoru	A	3
Joystick vpravo	A	2
Joystick dolů	A	5

7.3 ARM mbed

Druhým systémem byl ARM mbed. Na rozdíl od RIOT OS měly všechny desky certifikaci pro tento systém a tedy podporu od počátku, bez nutnosti psát nebo upravovat ovladače.

7.3.1 Příprava prostředí

Před stažením systému je třeba nainstalovat program *mbed CLI* spuštěním následujícího příkazu v konzoli:

```
pip install mbed-cli
```

Instalovaný program mbed CLI slouží nejen k vytvoření projektu, ale také k jeho překladu, ke správě závislostí v podobě externích knihoven a exportu nastavení pro vývojová prostředí, včetně Eclipse CDT.

Vytvoření mbed 2.0 projektu

Pro starší desky a desky s menším množstvím paměti, jako jsou právě Nucleo F031K6 a FRDM-KL05Z, je určen starší mbed 2.0. Ten je distribuován v podobě již přeložené knihovny, což komplikuje ladění. Získání zdrojových kódu je o něco složitější.

1. Nejdříve vytvoříme složku s projektem:

```
mbed new radiotemp_frdm_kl05z --create-only
cd radiotemp_frdm_kl05z
```

2. Přidáme *mbed-src* jako závislost projektu. Ta obsahuje zdrojové kódy systému mbed, namísto již přeložené knihovny.

```
mbed add https://developer.mbed.org/users/mbed_official
  /code/mbed-src/
```

3. Vytvoříme složku pro aplikaci a následně vlastní aplikaci, následující krok totiž vygeneruje skript Makefile obsahující také seznam souborů k překladu. Později vytvořené soubory do něj musí být přidány ručně.

```
mkdir src
```

4. Exportujeme projekt pro vývojové prostředí Eclipse CDT. Za parametrem `m` následuje název cílového hardware. V tomto případě se jedná o desku FRDM-KL05Z. Seznam podporovaných cílů je ve složce *.temp/tools* v souborech *latest_targets.json* a *legacy_targets.json*.

```
mbed export -i eclipse_gcc_arm -m KL05Z
```

Vytvoření mbed 5 projektu

Postup pro vytvoření projektu pro nejnovější verzi mbed 5 je jednodušší. V konzoli spustíme následující příkaz pro vytvoření složky nového projektu, kam budou staženy zdrojové kódy:

```
mbed new radiotemp_disco_l476g
cd radiotemp_disco_l476g
```

Nyní pokračujeme od 3. kroku pro verzi mbed 2.0.

Vytvoření projektu v Eclipse

Takto připravený projekt stačí v Eclipse CDT importovat. V kontextovém menu *Project Explorer* zvolte *Import* a vyberte složku, kde je projekt umístěn.

Ve vlastnostech projektu (v kontextové nabídce *Project Explorer*) jděte do *C/C++ Build > Settings* na záložce *Devices* zvolte procesor. Pro desku FRDM-KL05Z je to MKL05Z32xxx4.

7.3.2 Překlad projektu

Pro ladění doporučuji překládat přímo ve vývojovém prostředí Eclipse. V kontextovém menu projektu zvolte *Build Project*. Výstupní soubory projektu budou ve složce *BUILD*. Soubory *bin* a *hex* jsou určeny pro naprogramování zařízení, soubor *elf* pak pro ladění v ladícím sezení (viz 7.2.4).

Nefunkční překlad

V době psaní tohoto textu se v sestavovacím skriptu Makefile vyskytovala chyba, kvůli které nebylo možné projekt přeložit na desce FRDM-KL05Z a překlad končil s hláškou:

```
arm-none-eabi-gcc.exe: fatal error: c:/arm-gcc/arm-none-eabi/lib
/nano.specs: attempt to rename spec 'link' to already defined
spec 'nano_link'
```

Pro opravu stačí otevřít soubor *Makefile* a provést následující změny:

- Z proměnné `LD` odstranit:
  - `'--specs=nano.specs'`
  - `'-Wl,--wrap,_malloc_r'`
  - `'-Wl,--wrap,_free_r'`
  - `'-Wl,--wrap,_realloc_r'`
  - `'-Wl,--wrap,_calloc_r'`
  - `'-Wl,--wrap,exit'`
- Z proměnné `LD_FLAGS` odstranit:
  - `-Wl,--wrap,_malloc_r`
  - `-Wl,--wrap,_free_r`
  - `-Wl,--wrap,_realloc_r`
  - `-Wl,--wrap,_calloc_r`
  - `-Wl,--wrap,exit`

Pozor, parametr `specs` v proměnné `LD_FLAGS` musí zůstat zachován. Nyní by měl být projekt přeložitelný.

Překlad s optimalizacemi

Překládat je možné také programem mbed CLI. Nejdříve doporučuji nastavit pro projekt výchozí toolchain, cestu k němu a cílové zařízení. V příkazové řádce ve složce projektu spusťte následující příkazy. Parametr *target* uzpůsobte použitému zařízení.

```
mbed toolchain GCC_ARM
mbed target DISCO_L476VG
```

Pro sestavení projektu **mbed 2.0** s optimalizacemi nyní stačí spustit příkaz:

```
mbed compile --profile .temp/tools/profiles/small.json
```

Pro sestavení projektu **mbed 5.4** s optimalizacemi pak příkaz:

```
mbed compile --profile mbed-os/tools/profiles/release.json
```

Výstupní soubory jsou umístěny ve složce *BUILD/nazev_cile*. I v tomto případě jsem se setkal s nefunkčním překladem a bylo nutné upravit soubor předávaný jako parametr *profile* stejným způsobem, jako soubor skriptu Makefile.

7.3.3 Implementace testovací aplikace

Stejně jako v případě RIOT OS i pro ARM mbed byla implementována zjednodušená verze aplikace (viz kap. 6.4), která měřenou teplotu odesílá na UART. Viz ukázku C.2.

7.4 Rádiový modul SPSGRF-868

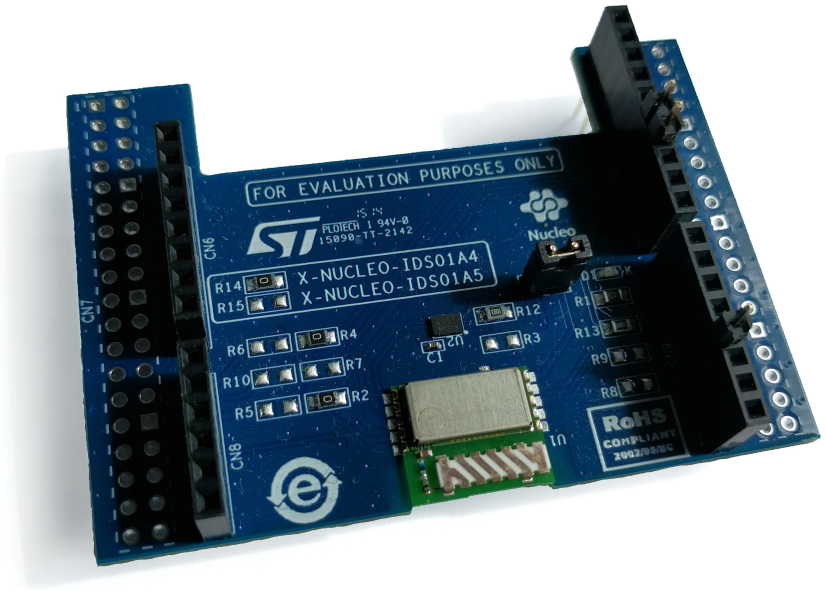
Vývojový kit X-NUCLEO-IDS01A4 s rádiovým modulem SPSGRF-868 od firmy STMicroelectronics, určený pro ty desky Nucleo, které jsou pinově kompatibilní s deskami Arduino, mi zapůjčili kolegové z JABLOTRON ALARMS a. s.

Jelikož desky Nucleo F031K6 ani Discovery L476G nejsou pinově kompatibilní s tímto kitem, poskytli mi také desku Nucleo L476RG, která má stejný procesor, jako deska Discovery L476G, jen v menším poudře a s méně vývody, a také desku Nucleo F030R8 s procesorem velice podobným tomu z desky Nucleo F031K6, ale s 64 kB paměti Flash namísto 32 kB.

Rádiový modul sestává z rádiového čipu SPIRIT1, antény a pasivních součástek. STMicroelectronics dodává knihovnu HAL a pár ukázkových aplikací.

- Bezlicenční frekvenční pásmo 868 MHz.
- Komunikace s procesorem na desce po sběrnici SPI.
- 4 programovatelné piny GPIO využitelné například pro přerušení procesoru v případě události.

- Podpora v podobě referenčního manuálu a balíčku *X-CUBE-SUBG1* s knihovnou *SPIRIT1_Library* a ukázkovými aplikacemi.



Obrázek 7.4: Rádiový modul SPSGRF pro desku Nucleo

Rádiový čip SPIRIT1 umožňuje nejen odesílat surová data, ale implementuje také dva typy protokolů s autonomním potvrzováním přijatých paketů, ověřování integrity pomocí kontrolních součtů, opakované vysílání při vypršení časového limitu pro potvrzení příjemcem. Dokáže také šifrovat data algoritmem AES-128 a podporuje CSMA/CA pro zabránění kolizí při vysílání.

Pro události, například příjem paketu, odeslání paketu, události týkající se kvality signálu, nebo změny stavu, lze konfigurovat přerušení na 4 pinech GPIO. Je tak možné uspat procesor a při příjmu platného paketu jej skrze GPIO probudit.

SPIRIT1 má několik provozních režimů, včetně 2 režimů snížené spotřeby a režimu úplného vypnutí (pomocí pinu GPIO), a vnitřní stavový automat pro přechod mezi nimi (STMicroelectronics, 2016a, str. 32). Přechod mezi těmito režimy je řízen z rozhraní pro procesor.

7.4.1 Rozhraní pro procesor

Konfigurace probíhá pomocí registrů (STMicroelectronics, 2016a, str. 82), příchozí a odchozí data jsou uložena v interních frontách FIFO. Pro čtení a zápis do registrů, nebo front, případně další operace, slouží příkazy (STMicroelectronics, 2016a, str. 34). Jedná se například o příkazy pro:

- Započetí příjmu nebo vysílání.

- Ukončení příjmu nebo vysílání.
- Přechod do režimu připravenosti, pohotovostního režimu nebo režimu spánku.
- Řízení šifrování AES.
- Vymazání přijímací nebo odesílací fronty FIFO.
- Reset čipu.

7.4.2 Ovladač pro RIOT OS

Ovladač pro systém RIOT jsem psal jako API tvořící vrstvu mezi systémem (aplikací) a knihovnou *SPIRIT1_Library* dodávanou výrobcem, a to z důvodu komplexnosti nízkoúrovňového ovladače samotného modulu. Nachází se v modulu *drivers/spsgrf*.

Při návrhu jsem analyzoval ukázkovou implementaci aplikace s rádiovým modulem pro Nucleo F411. STMicroelectronics ve všech příkladech využívá vrstvu HAL, kterou k procesorům dodává, pro inicializaci a ovládání periferií procesoru. Protože má RIOT OS vlastní platformě nezávislé ovladače pro periferie, bylo nutné implementovat funkce využívané knihovnou rádiového modulu tak, aby využívaly ovladače systému, a zároveň implementovat API, které bude zaobalovat volání knihovních funkcí.

Vývoj jsem začal na desce Nucleo L476RG, pro kterou jsem nejdříve vytvořil modul *boards/nucleo_l476*. Ten stavěl na mnou již vytvořeném modulu pro desku Discovery L476G a lišil se přiřazením pinů pro periferie.

Rozhraní SPI

Nejdříve jsem implementoval funkce rozhraní rádia a procesoru deklarované knihovnou, týkající se samotné komunikace po sběrnici SPI. Tyto využívají ovladače RIOT OS pro sběrnici SPI a jsou tak nezávislé na použité desce a procesoru.

- `RadioSpiWriteRegisters` pro zápis dat do vnitřního registru čipu,
- `RadioSpiReadRegisters` pro čtení dat z vnitřního registru čipu,
- `RadioSpiCommandStrobes` pro odeslání příkazu (STMicroelectronics, 2016a, str. 34),
- `RadioSpiWriteFifo` zapisující data do odesílací fronty FIFO,
- `RadioSpiReadFifo` pro čtení dat z přijímací fronty FIFO.

Aplikační rozhraní

Další na řadě bylo aplikační rozhraní. Tyto funkce z větší části tvoří mezivrstvu ke knihovně SPIRIT1, navíc se ale starají o zámky pro exkluzivní přístup k rádiovému modulu. Důležité jsou funkce týkající se inicializace rádia a typu paketů.

Pro konfiguraci slouží struktura `spsgrf_init_t`, kde uživatel konfiguruje, které piny procesoru jsou připojeny k pinům rádiového modulu a dále konfiguraci rádiového modulu, jako je frekvence, šířka kanálu, nebo modulace. Inicializační funkce `spsgrf_init` nakonfiguruje piny podle požadavků rádia, provede inicializaci SPI sběrnice a inicializaci rádiového modulu.

Registrace modulu do systému

Vytvořený modul ovladače je třeba přidat do skriptů Makefile ve složce *drivers*. Do *Makefile.dep* byl přidán modul a jeho závislosti – ovladače GPIO a sběrnice SPI. Do *Makefile.include* potom byla přidána složka *include* modulu rádia.

Hlavičkový soubor s deklaracemi aplikačního rozhraní ovladače se nachází v *drivers/include/spsgrf.h*.

7.4.3 Testování modulu

Upravil jsem srovnávací aplikace pro měření teploty a přidal jsem inicializaci rádiového modulu na desce Nucleo L476RG. Konfigurace pinů desky je v tabulce 7.6. Pin `SDN` slouží k zapnutí nebo vypnutí modulu, `SPI_CS` pro volbu slave a zahájení komunikace.

Tabulka 7.6: Konfigurace vstupů a výstupů rádiového modulu na desce Nucleo L476RG

Pin modulu SPSGRF	Procesor STM32L476RG		
	Port	Pin	Poznámka
SDN	A	10	Výstup, pull-up
SPI_CS	B	6	Výstup, pull-up
SPI_MOSI	A	7	AF5
SPI_MISO	A	6	AF5
SPI_CLK	B	3	AF5

Piny definovány v manuálu STMicroelectronics (2015b), str. 16.
AFx je alternativní funkce pinu dle STMicroelectronics (2015a), tabulka 16.

Problémy s SPI

Hned na počátku se objevily problémy při komunikaci po sběrnici SPI. Rádiový modul používá 8bitové datové rámce. Periferie SPI procesoru STM32L476RG podporuje rámce od 4 do 16 bitů, přestože však byla konfigurovaná na 8bitové rámce, při odeslání 1 byte došlo k odeslání 2 bytů, přičemž druhý byl nulový. Tento problém je dále rozebrán v kapitole 7.5.2.

Uváznutí inicializace rádia

Bohužel se mi rádiový modul nepodařilo zprovoznit. Inicializační funkce ovladače rádia, `spsgrf_init`, provádí inicializaci GPIO a SPI, restart rádiového modulu a volá funkci `SpiritRadioInit` z knihovny SPIRIT1.

Ta provádí výpočty hodnot konfigurace podle inicializační struktury a jejich zápis do konfiguračních registrů a následně volá funkci `SpiritRadioSetFrequencyBase`. Na jejím konci se volá funkce `SpiritManagementWaVcoCalibration` provádějící ruční kalibraci VCO (oscilátoru), která řeší chybu v návrhu čipu (STMicroelectronics, 2016b, str. 3).

Jakmile program kalibrace provede 4. bod v postupu k obejití této chyby, dojde k uváznutí při volání funkce, která čeká na synchronizaci stavu rádia a programu. Ta v nekonečném cyklu čte stavové slovo rádia, dokud se nerovná očekávanému stavu v programu.

Při krokování procesu inicializace jsem zjistil, že chyba pravděpodobně vzniká ještě ve funkci `SpiritRadioInit`, když se zapisuje konfigurace registrů digitálního bloku, `MOD1`, `MOD0`, `FDEV0` a `CHFLT`. Po jejich zápisu je ve stavovém slově hodnota bytu, reprezentujícího stav čipu, mimo rozsah a tedy neplatná (STMicroelectronics, 2016a, str. 33, pozn. 1 pod tabulkou). Lze předpokládat, že chování čipu po zbytek inicializační procedury je nedefinované.

7.5 Další řešené problémy

V této části bych rád zmínil několik problémů a úskalí, na které jsem narazil. Některé souvisí s hardware a jeho zapojením, jiné s jeho špatnou konfigurací v software. Nesprávné zapojení obvodů nebo jen použití nevhodných součástek může mít až nečekaný vliv na software a jeho běh. Mohou se začít vyskytovat sporadické chyby, které se jen těžko hledají, a jejich původ může vývojáře zaskočit.

7.5.1 Frekvence, délka vedení a odpory pro sběrnici I²C

V průběhu oživování teplotního senzoru jsem využíval zapůjčený osciloskop připojený na vodiče SDA a SCL. Frekvence sběrnice byla nastavena na 400 kHz a komunikace fungovala. Později jsem sondy osciloskopu odpojil a obvod přestal komunikovat. Po konzultacích jsem problém vyřešil následovně:

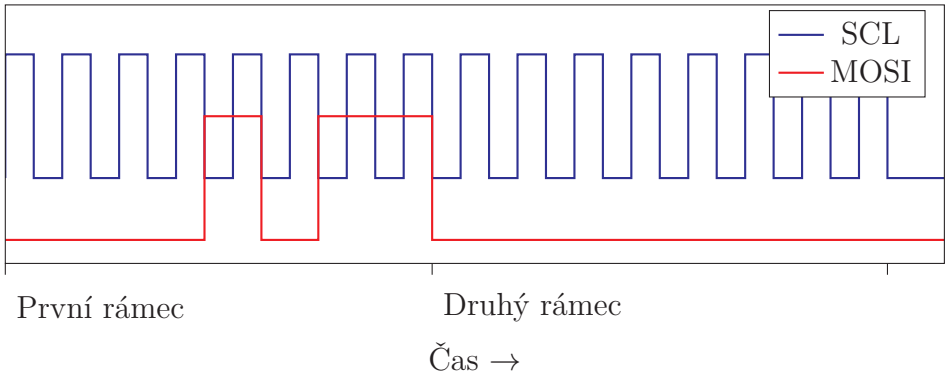
- Kabelové propojky datových vodičů mezi deskou a nepájivým polem byly zkráceny.

- Frekvence byla snížena na 100 kHz.
- Původní pull-up rezistory o hodnotě 10 kΩ byly nahrazeny rezistory 4,7 kΩ.
- Po problémech s komunikací na desce FRDM-KL05Z, která měla na výstupu místo 3,3 V zhruba 2,8-2,9 V, byl přidán regulátor napětí 5 V na 3,3 V.

7.5.2 Velikost datových rámců SPI na procesorech STM

Při ladění ovladače SPI a ovladače rádiového modulu SPSGRF-868 na desce Nucleo L476RG jsem narazil na problém s konfigurovanou a skutečnou velikostí datových rámců.

Modul SPSGRF-868 používá 8bitové datové rámce. Periferie SPI podporuje rámce o velikostech od 4 do 16 bitů, přestože však byla konfigurovaná na 8bitové rámce, při odeslání 1 byte došlo k odeslání 2 bytů, přičemž druhý byl nulový. Obrázek 7.5 zobrazuje, jak na bylo osciloskopu zachyceno odeslání bytu `0x0B`.



Obrázek 7.5: Průběh signálů na sběrnici SPI s 32bitovým přístupem k datovému registru

Sdružování dat

Referenční manuál (STMicroelectronics, 2017d, str. 1302) definuje pojem *Data packing*. Pokud je velikost datového rámce menší nebo rovna 8 bitům, což v mém případě byla, pak je sdružování dat automaticky zapnuto, kdykoliv je proveden 16bitový přístup k datovému registru periferie SPI. Při zápisu 16bitového slova do datového registru je do odesílací fronty zařazen spodní byte a poté horní byte.

Přístup k datovému registru

Přestože hovoří referenční manuál pouze o 16bitovém přístupu, deklaruje struktura `SPI_TypeDef` datový registr `DR` jako 32bitový bezznaménkový integer. Pro přiřazení hodnoty do takto deklarovaného registru použije překladač instrukci `STR`, která slouží pro uložení 32bitového slova.

A právě tato instrukce představuje problém. Zápisem jednoho byte provede překladač jeho rozšíření na 4 byty doplněné nulami. Z pohledu periferie se tak uplatní 16bitový přístup a do fronty je vložen spodní byte, tedy původní data, a horní byte, tedy nulové rozšíření.

Řešení problému

Pokud je periferie konfigurovaná pro 8bitové datové rámce, musí být zajištěn 8bitový přístup k datovému registru. Stačí pouhé přetypování datového registru a poté jeho dereferencování:

```
*(uint8_t *)&SPI1->DR = (uint8_t)data;
```

Ukázka kódu 7.4: Zajištění 8bitového přístupu k datovému registru SPI

Pro takový kód překladač použije instrukci `STRB`, která uloží pouze jeden byte, namísto `STR`, čímž je problém zápisu vyřešen.

7.5.3 Uváznutí UART přerušení na desce Nucleo F031

Na počátku vývoje jsem měl UART konfigurován na piny vyvedené na pinovou lištu a připojen externí převodník FTDI. Ten byl totiž k počítači připojen neustále, takže i při odpojení desky zůstal port otevřen a při výměně desky se číslo portu nezměnilo.

Sporadicky ale začal program na RIOT OS zamrzat v obsluze přerušení UART. Ve stavovém registru `ISR` byl nastaven příznak `RXNE` příjmu dat a přijatý znak odpovídal naposledy odeslanému znaku. Zároveň byly nastaveny chybové příznaky *Overrun* a *Framing Error*, indikující příjem dalšího znaku bez zpracování předchozího a chybu rámce. Když k tomuto problému došlo, bylo to vždy při odesílání znaku tečky v řetězci s měřenou teplotou.

Problém a jeho řešení

Protože byl UART určen pouze pro vysílání, bylo přerušení příjmu vypnuto, což vysvětluje chybu *overrun*. Zároveň jsem udělal následující chyby:

- Nezapojil jsem vodič RX do desky,
- a nenastavil jsem pull-up rezistor pro RX pin.

Jelikož byl tedy pin RX plovoucím vstupem, byl s největší pravděpodobností ovlivňován ostatními hradly, a přestože provádí periferie UART až 16násobné vzorkování jednotlivých bitů (kvůli deviaci hodin) s hlasováním o majoritě tří vzorků, muselo docházet k detekci start bitu. Další plavání vstupu kolem rozhodovací úrovně pak způsobilo *Framing Error*.

Řešením bylo připojit k RX pull-up rezistor. Později jsem UART přesměroval na virtuální COM port zprostředkovaný ladícím rozhraním.

7.6 Srovnání systémů

Srovnány budou systémy, které se podařilo přenést a zprovoznit na zvoleném hardware. Jedná se o:

- RIOT OS, verze z 21. listopadu 2016,
- ARM mbed 2.0
- a ARM mbed 5.4

7.6.1 Náročnost na hardware

Zatímco RIOT OS již předem deklaroval orientační požadavky na hardware (Hahm et al., 2016), pro systém mbed se mi nepodařilo oficiální čísla najít. Provedl jsem tedy porovnání velikostí srovnávací aplikace s identickou funkčností. Překlad obou aplikací probíhal v režimu *release*, tedy s nejvyšší mírou optimalizací, jakou umožňoval sestavovací systém.

Deska Discovery L476VG bohužel není podporována v mbed 2.0, proto bylo na této desce provedeno srovnání RIOT OS a mbed 5.4. Výsledky jsou shrnuty v tabulce 7.7.

Tabulka 7.7: Velikost srovnávací aplikace mezi zvolenými operačními systémy na desce Discovery L476G

	Paměť RAM				Paměť Flash		
	Program	Zásobník	Systém	Celkem	Program	Systém	Celkem
RIOT OS	0 B[1]	1.536 B	1.392 B[2]	**2.928 B**	291 B	12.541 B	**12.832 B**
ARM mbed 5.4	392 B	1.024 B	10.868 B	**12.284 B**	372 B	73.896 B	**74.587 B**

[1] Všechny proměnné jsou alokovány na zásobníku
[2] Včetně 256 B pro zásobník vlákna řídícího úsporný režim

Z menších desek byla pro srovnání vybrána deska FRDM-KL05Z. Na ní byly srovnána velikost úlohy implementované na systémech RIOT OS a mbed 2.0. Výsledky jsou shrnuty v tabulce 7.8.

Úroveň optimalizací

Parametry pro překladač a linker se u sestavovacích skriptů liší, oba systémy ale shodně používají parametr `-Os` pro linker (optimalizace na výslednou velikost) a parametry `-ffunction-sections -fdata-sections -fno-builtin` pro překladač.

Tabulka 7.8: Velikost srovnávací aplikace mezi zvolenými operačními systémy na desce FRDM-KL05Z

	Paměť RAM				Paměť Flash		
	Program	Zásobník	Systém	Celkem	Program	Systém	Celkem
RIOT OS	0 B[1]	1.536 B	1.240 B[2]	**2.776 B**	301 B	10.831 B	**11.132 B**
ARM mbed 2.0	144 B	128 B	836 B	**1.236 B[3]**	372 B	17.376 B	**17.748 B**

[1] Všechny proměnné jsou alokovány na zásobníku
[2] Včetně 256 B pro zásobník vlákna řídícího úsporný režim
[3] Včetně 128 B pro dynamickou alokaci paměti

Metodika

Výpočty nároků jsem prováděl na základě výstupu překladu v podobě linker mapy. Nároky na velikost paměti RAM jsou dány součtem velikostí všech sekcí umístěných linkerem do adresního prostoru RAM:

- *Program* je součet proměnných z objektového souboru *main.o*.
- *Zásobník* je v u systému RIOT alokován staticky a lze jej vyčíst ze souboru linker mapy. U systému mbed 5.4 je zásobník alokován za běhu. Vyšel jsem tedy ze zdrojových kódů, kde je jeho velikost pro procesory Cortex-M nastavena ve výchozím stavu na (v případě STM32L476 dvojnásobek) 512 bytů.
- *Systém* je rozdíl mezi celkovým využitím a využitím srovnávací aplikací.

Podobně je tomu v případě nároků na programovou paměť Flash:

- *Program* je součet velikosti kódu a symbolů v objektovém souboru *main.o*. Přičteny jsou také symboly ze sekcí *.data* a *.rodata* umístěné po startu do paměti RAM. Tato inicializovaná data totiž musí být uchována v programové paměti.
- *Systém* je rozdíl mezi celkovým využitím paměti Flash, které je dané součtem velikostí sekcí umístěných v adresním prostoru Flash a navíc velikostí sekce *.data*.

7.6.2 Zprovoznění systému a úlohy

Přenos

V tomto konkrétním případě bylo zprovoznění systému mbed podstatně jednodušší, díky podpoře všech použitých desek alespoň některými verzemi systému. Naopak zprovoznění systému RIOT bylo složitější o nutnost jeho přenesení na zvolený

hardware, což obnáší implementaci modulů pro podporu procesorů, jeho periferií a následně desek.

U obou systémů byly potřeba úpravy sestavovacích skriptů, ačkoliv u mbed to bylo z spíše z důvodu chyby v dané verzi.

Rozhraní API

Zprovoznění srovnávací úlohy spočívalo především v odladňování ovladačů. API použité v této jednoduché aplikaci bylo až na drobné rozdíly přímočaré. Zatímco RIOT OS používá pro jádro systému a ovladače pouze jazyk C, ačkoliv nijak nebrání v použití C++ pro kód aplikace, u systému mbed je implementováno množství ovladačů v podobě C++ tříd nad C funkcemi.

Využití C++ může přinášet pozitiva i v segmentu vestavných zařízení. Třída zaobalující ovladač zařízení může v destruktoru obstarávat správnou deinicializaci a automatické uvolnění objektu alokovaného na zásobníku tak deinicializaci provede za programátora (princip RAII). Nároky na paměť jsou zejména při využití tříd a dědičnosti vyšší, ale při vhodné optimalizaci, jako je vypnutí výjimek, běhových informacích o typech (RTTI) a vhodné optimalizaci nemusí být problém.

7.6.3 Ovladače a knihovny

RIOT i ARM mbed mají přibalenu také sadu ovladačů pro některá externí zařízení a knihovny třetích stran. Několik příkladů:

- RIOT má množství ovladačů pro různé senzory, jakou jsou PIR, senzory osvětlení, vlhkostní a teplotní senzory, a rádiové moduly, například NRF24L01P nebo XBee.
- mbed má systém souborů FAT. Další knihovny jsou dostupné na webu a je možné je přidat v podobě externích závislostí pomocí mbed CLI.
- Oba systémy mají IP stack v podobě knihovny lwIP a další.

7.6.4 Řízení spotřeby

Systém RIOT spouští automaticky jedno vlákno s nejnižší možnou prioritou, jehož jediným úkolem je zajistit přepnutí do režimu s nejnižší možnou spotřebou.

K řízení spotřeby je definováno LPM (Low-power Management) API. Jelikož je ale řízení spotřeby platformě závislé, implementace tohoto API musí být součástí modulu procesoru.

Stejně tak systém mbed používá zvláštní vlákno, které ve smyčce volá funkci `sleep` uspávající procesor.

Přepnutí do hlubokého spánku u některých procesorů musí předcházet korektní deinicializace některých periferií. Opětovné probuzení pak může znamenat nutnost znovu inicializovat periferie. Tyto faktory komplikují řízení spotřeby, potýkat se s nimi ale musí všechny operační systémy.

7.6.5 Ostatní

Multi-tasking

Víceúlohové systémy RIOT a mbed 5.4 mají kromě standardních synchronizačních primitiv, jako je mutex, také API pro **komunikaci mezi vlákny**, kterou představují „poštovní schránky“ (Mailbox u RIOT, mail u mbed).

Se synchronizačními primitivy souvisí také problémy paralelismu – inverze priorit. Implementace mutexu v systému **mbed**, na rozdíl od RIOT OS, **umí řešit inverzi priorit** pomocí dědění priorit. Pokud vlákno zamyká mutex, který drží vlákno s nižší prioritou, pak je tomuto vláknu přidělena priorita prvního vlákna, a to až do uvolnění mutexu.

Dokumentace

Oba projekty disponují dokumentací v podobě průvodců pro rychlý start a tutoriály a dále popisu modulů systému a jejich API.

Dokumentace mbed v podobě průvodce prvními kroky je podle mého názoru o něco lépe navržena, při dalším hledání některých problémů, například export projektu do Eclipse, jsem ale nacházel starší kousky dokumentace, které již neodpovídaly aktuálnímu stavu.

V případě systému RIOT je dokumentace poněkud roztříštěna. Část dokumentace s popisy modulů systému a rozhraní API je na webu projektu, další část v podobě Wiki se nachází v repozitáři projektu na GitHub.

Komunita a vývoj

Oba projekty mají repozitáře na GitHub a velice aktivní. Počet přispěvovatelů tvoří u repozitáře systému RIOT téměř 150 lidí, u mbed je to téměř 300 lidí, ne všichni jsou však aktivními přispěvovateli.

V tuto chvíli se oba projekty vyvíjí a u obou přibývají změny do repozitáře prakticky každý den. Historie repozitářů obou projektů sahá do roku 2013. Samotné projekty jsou ale ještě starší, RIOT vychází z jádra FireKernel a mbed byl vyvíjen s uzavřeným zdrojovým kódem, distribuovaný ve formě již přeložené knihovny.

Za systémem mbed stojí značka ARM a aktivně do něj přispívají i někteří vývojáři ARM Ltd., můžeme tedy předpokládat, že mbed se bude i nadále vyvíjet. RIOT OS pro změnu podporují Svobodná univerzita Berlín, Univerzita Hamburk a INRIA.

Systém mbed má pro vlastní komunitu svá diskusní fóra. RIOT žádné diskusní fórum nemá, nabízí ale mailing list zvlášť pro vývojáře a uživatele, a chat IRC.

7.6.6 Celkové srovnání

V tabulce 7.9 najdete přehled vybraných srovnávacích kritérií.

Tabulka 7.9: Srovnání operačních systémy pro IoT na desce Discovery L476G

	RIOT OS	ARM mbed	
		Verze 2.0	Verze 5.4
Jazyk	C, C++	C, C++	
Vrstva abstrakce hardware	●	●	●
Certifikace hardware	-	●	●
Multi-tasking			
Vícevláknový	●	-	●
Typ plánovače	Preemptivní	-	Preemptivní
Podpora real-time	●	-	●
Meziprocesová komunikace	●	-	●
Řeší inverzi priorit	-	-	●
Subjektivní hodnocení			
Náročnost přenosu	3	-	-
Složitost zprovoznění úlohy	1	1	1
Kvalita dokumentace	2	2	2
Komunita	3	1	1
Spolehlivost vývojářů	2	1	1
Zabezpečení			
Podpora ochrany paměti	-	-	Volitelně uVisor[1]

(●) podporováno, (-) nepodporováno, (?) nepodařilo se zjistit. Hodnocení známkováno jako ve škole.

[1] uVisor podporuje ochranu paměti a implementuje rozhraní pro výměnu dat mezi privilegovaným a neprivilegovaným kódem aplikace. Podporuje některé procesory Cortex-M3 a Cortex-M4.

8 Závěr

Záměrem práce bylo seznámení se s operačními systémy pro IoT a vybrané tři systémy zprovoznit na dodaném hardware. Úloha srovnávající tyto systémy byla měření teploty a její bezdrátové odesílání na koncentrátor.

Vybrány byly tyto operační systémy:

- RIOT OS,
- ARM mbed ve verzích 2 a 5
- a FreeRTOS

Operační systémy a srovnávací úloha byly přenášeny a spouštěny na následujících vývojových kitech:

- Freescale FRDM-KL05Z,
- STMicroelectronics Nucleo F031K6,
- Discovery L476VG
- a Nucleo L476RG, používané při vývoji pro rádiový modul, oba od téhož výrobce.

Pro měření teploty byl použit senzor **Microchip MCP9801** a pro bezdrátovou komunikaci rádiový module **SPSGRF-868** s čipem SPIRIT1 od STMicroelectronics.

8.1 Problémy při realizaci

Nepodařilo se plně zprovoznit rádiový modul SPSGRF-868. Byl implementován ovladač modulu pro systém RIOT OS, inicializace modulu ale selhávala v nízkoúrovňové knihovně dodávané výrobcem (viz sekce 7.4.3 o modulu). S tím souvisí chybějící implementace koncentrátoru pro PC. Přenos hodnot do počítače probíhá pouze textově skrz rozhraní UART.

8.2 Výstupy a výsledky

8.2.1 RIOT OS

Na začátku práce chyběla systému RIOT OS podpora pro všechny vybrané desky. Přidat podporu pro desku znamenalo znamenalo **vytvořit modul pro procesor** a doplnit jej ovladači periferií, inicializačním kódem pro zdroj hodinového signálu, linker skriptem a skripty Makefile.

- Pro všechny procesory jsem připravil linker script, kde jsou sekcím přiřazeny adresní rozsahy v adresním prostoru procesoru a skripty Makefile, s výjimkou procesoru STM32F031K6, který byl pouze doplněn do již existujícího modulu řady STM32F0.
- Pro všechny procesory, s výjimkou STM32F0 z desky Nucleo F031K6, jsem připravil inicializační kód pro konfiguraci procesoru a zdroje hodinového signálu.
- Pro procesor MKL05Z32 z desky FRDM-KL05Z jsem implementoval jednoduchý ovladač GPIO a uzpůsobil existující ovladače procesorů Freescale řady Kinetis pro čítače a časovače, sběrnici I^2C, SPI a UART. Zároveň jsem pro tento procesor připravil tabulku vektorů přerušení.
- Do modulu procesorů STMicroelectronics řady STM32F0 jsem přidal podporu pro procesor STM32F031K6. Přidal jsem chybějící ovladač periferie I^2C z modulu STM32F3, kterému jsem upravil tabulku děliček a přidal podporu pro rychlost 100 kHz.
- Pro procesory STM32L476 jsem využil existujících ovladačů z modulu procesorů řady STM32F4, které jsem musel upravit z důvodů velkých odlišností mezi těmito řadami. Procesory se liší uspořádáním registrů periferií, jejich množstvím a také přerušeními periferií. Zcela jsem přepsal podporu pro DMA, použitou například u ovladače UART pro odesílání dat. Opravil jsem 8bitové rámce u periferie SPI (viz 7.5.2) a upravil tabulku děliček pro rychlosti této sběrnice. Zároveň jsem pro tento procesor připravil tabulku vektorů přerušení.

Následně byly vytvořeny **moduly pro jednotlivé desky**, které používají již vytvořené moduly procesorů. Definice desek konfigurují jednotlivé periferie a přiřazují k nim piny na desce, vstupy a výstupy v podobě LED a tlačítek. Modul desky zároveň obsahuje inicializační kód desky a skripty Makefile.

Pro všechny čtyři desky byly vytvořeny moduly s alespoň základní podporou periferií použitých ve srovnávací aplikaci. Navíc byly přidány upravené skripty Makefile pro podporu sestavení RIOT OS na systémech Microsoft Windows.

8.2.2 ARM mbed

Systém mbed, pro který měly certifikaci všechny desky, se v nejnovější verzi 5.4 podařilo zprovoznit pouze na desce Discovery L476VG, která měla pro provoz tohoto systému dostatek prostředků.

Na deskách Nucleo F031K6 a FRDM-KL05Z se podařilo zprovoznit pouze starší verzi mbed 2.0. Ta bohužel nepředstavuje operační systém s podporou vláken, jako spíše knihovnu poskytující jednotné rozhraní k hardware.

8.2.3 Srovnávací aplikace

Na obou systémech byla implementována aplikace vyčítající teplotu z teplotního senzoru připojeného na sběrnici I^2C. Teplota byla v podobě formátovaného řetězce odesílána na rozhraní UART do připojeného PC.

8.2.4 Srovnání systémů

Na závěr jsem srovnal systémy podle vybraných kritérií (viz 7.6), mimo jiné také podle velikosti výsledné přeložené aplikace implementované pro všechny systémy. RIOT OS se ze všech srovnávaných systémů ukazuje jako systém s nejnižšími nároky na programovou paměť a v nárocích na paměť RAM jej předběhne pouze mbed 2.0, který nemá podporu vláken.

Co do přenosu mohu srovnávat pouze RIOT OS, který neměl podporu pro žádnou z desek. ARM mbed naopak desky podporoval v základu, přestože ve své nejnovější verzi podporuje pouze Discovery L476VG. Nejkomplikovanější pro přenos byla deska Discovery L476VG. Přestože jsem vycházel z ovladačů pro řadu STM32F4, bylo nutné udělat množství změn pro zprovoznění těchto ovladačů na novější řadě STM32L4. Naopak přidání podpory pro STM32F031, co by dalšího modelu již implementované řady, bylo podstatně jednodušší.

Při výběru systému bych se osobně rozhodoval podle konkrétní aplikace. Pokud bych měl desku s parametry třídy 0 nebo 1 (Hahm et al., 2016), preferoval bych RIOT OS. Moduly procesorů a desek pro tento systém se mně zdají, v porovnání s moduly pro systém mbed, jednodušší pro případný přenos nové platformy. Systém mbed bych volil až pro platformu s více prostředky, ideálně s logem „mbed Enabled“, a pro aplikace, které mohou využít potenciál v podobě volitelné ochrany paměti (v tuto chvíli pouze pro pár vybraných procesorů), C++ rozhraní API pro přístup k hardware, nebo souborový systém.

Literatura

ABRAHAM, S. – GALVIN, P. B. – GAGNE, G. *Operating System Concepts (6th Edition)*. John Wiley & Sons, Inc., 2002. ISBN 0-471-41743-2.

ARM Limited. *Cortex-M3 Technical Reference Manual* [online]. 2010. [cit. 2017-04-28]. Rev. r2p0. Dostupné z: `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337h/DDI0337H_cortex_m3_r2p0_trm.pdf`.

Emcraft Systems. *What is the Minimal Footprint of uClinux?* Dostupné z: `http://www.emcraft.com/stm32f429discovery/what-is-minimal-footprint`. [online], [cit. 2017-04-17].

Ericsson AB. *Ericsson Mobility Report* [online]. Listopad 2016. [cit. 2017-04-15]. Dostupné z: `https://www.ericsson.com/assets/local/mobility-report/documents/2016/ericsson-mobility-report-november-2016.pdf`.

EVANS, D. *The Internet of Things, How the Next Evolution of the Internet Is Changing Everything* [online]. Cisco IBSG, Duben 2011. [cit. 2017-04-15]. Dostupné z: `http://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf`.

Freescale Semiconductor, Inc. *Kinetis KL05 32 KB Flash, 48 MHz Cortex-M0+ Based Microcontroller* [online]. Březen 2014. [cit. 2017-04-05]. Doc. ID KL05P48M48SF1 Rev. 4. Dostupné z: `http://www.nxp.com/assets/documents/data/en/data-sheets/KL05P48M48SF1.pdf`.

Freescale Semiconductor, Inc. *KL05 Sub-Family Reference Manual* [online]. Listopad 2012. [cit. 2017-04-05]. Doc. ID KL05P48M48SF1RM Rev. 3.1. Dostupné z: `http://www.nxp.com/assets/documents/data/en/reference-manuals/KL05P48M48SF1RM.pdf`.

HAHM, O. et al. *Operating Systems for Low-End Devices in the Internet of Things: A Survey. IEEE Internet of Things Journal*. 2016, 3, 5, s. 720–734. ISSN 23274662. doi: 10.1109/JIOT.2015.2505901.

ITU-T. *ITU-T Recommendation Y.2060: Overview of the Internet of Things*. Technical report, International Telecommunication Union, Červen 2012. Dostupné z: `https://www.itu.int/rec/T-REC-Y.2060-201206-I`.

Microchip Technology Inc. *MCP9800/1/2/3, 2-Wire High-Accuracy Temperature Sensor* [online]. Listopad 2010. [cit. 2017-04-18]. Doc. ID DS21909D Rev. D. Dostupné z: ww1.microchip.com/downloads/en/DeviceDoc/21909d.pdf.

STMicroelectronics. *SPIRIT1, Low data rate, low power sub-1GHz transceiver* [online]. Říjen 2016a. [cit. 2017-05-01]. DocID022758 Rev. 10. Dostupné z: http://www.st.com/resource/en/datasheet/spirit1.pdf.

STMicroelectronics. *STM32F031x4 STM32F031x6 ARM®-based 32-bit MCU with up to 32 Kbyte Flash, 9 timers, ADC and communication interfaces* [online]. Leden 2017a. [cit. 2017-04-05]. DocID025743 Rev. 5. Dostupné z: http://www.st.com/resource/en/datasheet/stm32f031c4.pdf.

STMicroelectronics. *STM32L476xx Ultra-low-power ARM® Cortex®-M4 32-bit MCU+FPU, 100DMIPS, up to 1MB Flash, 128 KB SRAM, USB OTG FS, LCD, analog, audio* [online]. Prosinec 2015a. [cit. 2017-04-05]. DocID025976 Rev. 4. Dostupné z: http://www.st.com/resource/en/datasheet/stm32l476je.pdf.

STMicroelectronics. *SPIRIT1 device limitations* [online]. Říjen 2016b. [cit. 2017-05-02]. DocID023165 Rev. 7. Dostupné z: www.st.com/resource/en/errata_sheet/dm00053990.pdf.

STMicroelectronics. *Reference manual STM32F0x1/STM32F0x2/STM32F0x8 advanced ARM®-based 32-bit MCUs* [online]. Leden 2017b. [cit. 2017-04-05]. DocID018940 Rev. 9. Dostupné z: http://www.st.com/resource/en/reference_manual/dm00031936.pdf.

STMicroelectronics. *Migrating from STM32F401 and STM32F411 lines to STM32L4 Series microcontrollers* [online]. Únor 2017c. [cit. 2017-04-28]. DocID027151 Rev. 4. Dostupné z: www.st.com/resource/en/application_note/dm00144612.pdf.

STMicroelectronics. *Reference manual STM32F0x1/STM32F0x2/STM32F0x8 advanced ARM®-based 32-bit MCUs* [online]. Leden 2017d. [cit. 2017-04-05]. DocID018940 Rev. 9. Dostupné z: www.st.com/resource/en/reference_manual/dm00031936.pdf.

STMicroelectronics. *Getting started with the Sub-1 GHz expansion board based on SPSGRF-868 and SPSGRF-915 modules for STM32 Nucleo* [online]. Červen 2015b. [cit. 2017-05-01]. DocID027622 Rev. 2. Dostupné z: http://www.st.com/resource/en/user_manual/dm00168396.pdf.

TANENBAUM, A. S. *Modern Operating Systems (2nd Edition)*. Prentice Hall, 2001. ISBN 0130313580.

A Textové přílohy

A.1 Obsah přiloženého CD

radiotemp_disco_l476vg.zip
: Obsahuje složku projektu s mbed 5.4 pro desku Discovery L476VG.

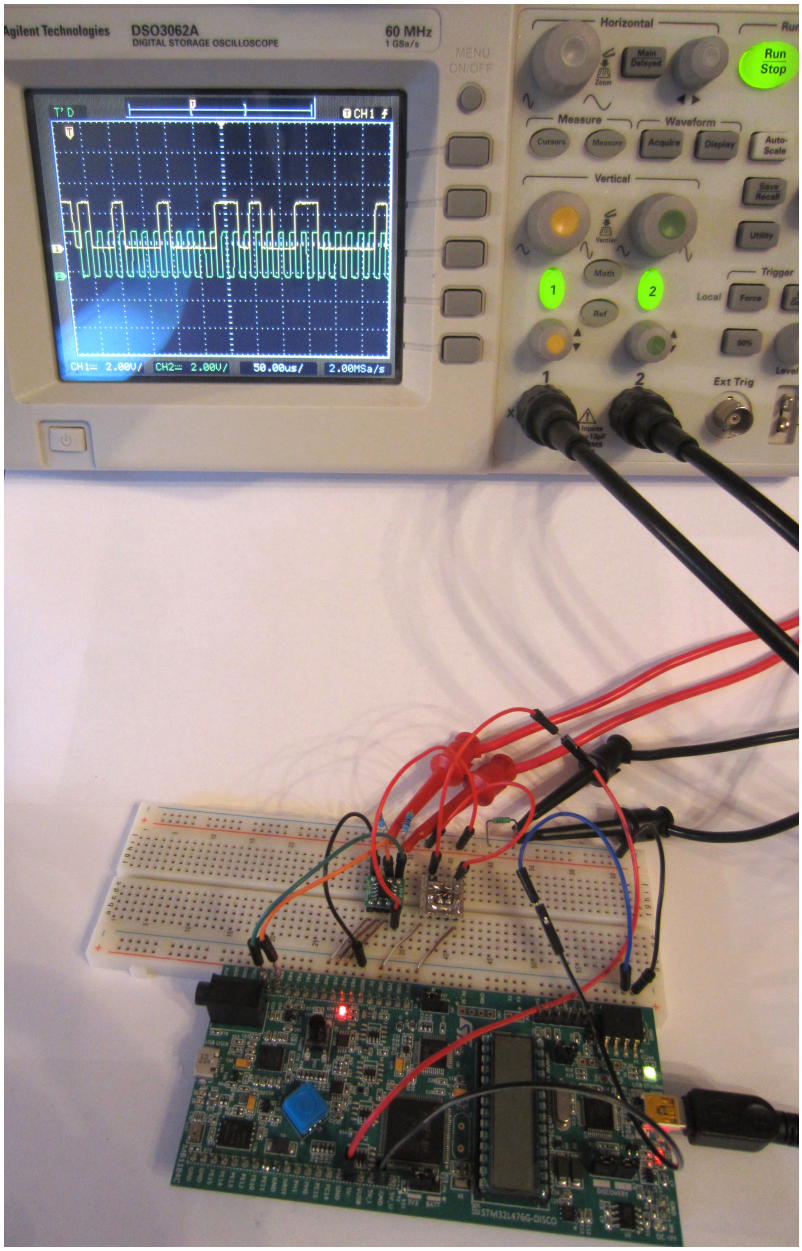
radiotemp_mbedlib_frdm_kl05z.zip
: Obsahuje složku projektu s mbed 2.0 pro desku FRDM-KL05Z.

radiotemp_mbedlib_nucleo_f030k6.zip
: Obsahuje složku projektu s mbed 2.0 pro desku Nucleo F031K6.

RIOT.zip
: Obsahuje složku projektu s RIOT OS. Projekt je sdílený pro všechny desky, pro každou z nich je vytvořena jedna *Build Configuration*. Vlastní aplikace je umístěna v *examples/playground*.

Všechny složky projektu jsou přímo importovatelné do vývojového prostředí Eclipse CDT Neon.1 a vyšší, s nainstalovanými zásuvnými moduly (viz 7.1.6).

B Obrázkové přílohy



Obrázek B.1: Discovery L476VG měřící teplotu pomocí MCP9801 po sběrnici I^{2}C. Komunikace po této sběrnici byla zachycena na osciloskopu.

C Ukázky kódu

Následují ukázky kódu v podobě zkráceného zdrojového kódu srovnávací aplikace pro systém RIOT a systém ARM mbed 2.0 i 5.4.

```c
#define THERMOMETER_ADDR            0x48    /* I2C address, 0b1001000 */
#define REG_TEMPERATURE             0x00
#define REG_CONFIG                  0x01
#define MCP_CONFIG_RES_11BIT        (2 << 5)

static int initialize(void)
{
    if (i2c_init_master(I2C_0, I2C_SPEED_LOW) < 0) {
        return -1;
    }
    if (i2c_write_reg(I2C_0, THERMOMETER_ADDR, REG_CONFIG,
            MCP_CONFIG_RES_11BIT) < 0) {
        return -1;
    }
    if (i2c_write_byte(I2C_0, THERMOMETER_ADDR, REG_TEMPERATURE) < 0) {
        return -1;
    }
    return 0;
}

static void loop(void)
{
    uint32_t last_wakeup = xtimer_now();
    uint8_t temp[2];

    while (1) {
        xtimer_periodic_wakeup(&last_wakeup, 1 * SEC_IN_USEC);

        LED0_TOGGLE;

        if (i2c_read_bytes(I2C_0, THERMOMETER_ADDR, &temp, 2) < 0) {
            LED1_ON;    /* Error indication on */
            continue;
        }

        LED1_OFF;
        printf("%d.%03u\r\n", temp[0], (1000 * temp[1]) / 256);
    }
}

int main(void)
{
    if (initialize() == 0) {
        loop();
    }
    else {
        LED1_ON;
    }
    return 0;
}
```

Ukázka kódu C.1: Testovací aplikace pro RIOT OS

```
// Thermometer I2C address, 4 most significant bits are set from factory to 0b1001
#define THERMOMETER_ADDR        0x48        // 0b1001000
// mbed API uses 8bit addresses, address must be left shifted before passing
#define MBED_THERMOMETER_ADDR   (THERMOMETER_ADDR << 1)

#define REG_TEMPERATURE         0x00
#define REG_CONFIG              0x01

#define MCP_CONFIG_RES_SHIFT    5
#define MCP_CONFIG_RES_11BIT    (2 << MCP_CONFIG_RES_SHIFT)


static DigitalOut led1(LED1);
static DigitalOut led2(LED2);
static Serial serial(USBTX, USBRX);
static I2C i2c(D14, D15);

int main()
{
    char buffer[2];

    // Set frequency and create start condition
    i2c.frequency(100000);

    // Write configuration register
    buffer[0] = REG_CONFIG;
    buffer[1] = MCP_CONFIG_RES_11BIT;
    if (i2c.write(MBED_THERMOMETER_ADDR, buffer, 2) < 0)
    {
        serial.puts("Failed to write configuration register\r\n");
        led2 = 1;
        while (1);
    }

    // Switch to temperature register
    buffer[0] = REG_TEMPERATURE;
    if (i2c.write(MBED_THERMOMETER_ADDR, buffer, 1) < 0)
    {
        serial.puts("Failed to switch to temperature register\r\n");
        led2 = 1;
        while (1);
    }

    while (1)
    {
        wait_ms(1000);
        led1 = !led1;

        if (i2c.read(MBED_THERMOMETER_ADDR, buffer, 2) < 0)
        {
            led2 = 1;
            serial.puts("Failed to read temperature\r\n");
        }
        else
        {
            led2 = 0;
            serial.printf("%d.%03u\r\n", buffer[0], (1000 * buffer[1]) / 256);
        }
    }
}
```

Ukázka kódu C.2: Testovací aplikace pro ARM mbed