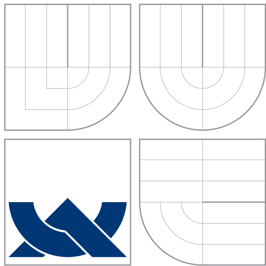
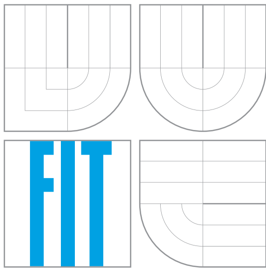


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

POROVNÁNÍ VLASTNOSTÍ A VÝKONNOSTI JADER UC/OS-II A UC/OS-III

COMPARISION OF PROPERTIES AND PERFORMANCE OF UC/OS-II AND UC/OS-III KERNELS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. JÁN LORENC
AUTHOR

VEDOUCÍ PRÁCE — Ing. JOSEF STRNADEL, Ph.D.
SUPERVISOR

BRNO 2016

Abstrakt

Tato diplomová práce se zabývá otestováním výkonnosti operačních systémů reálného času *µC/OS-II* a *µC/OS-III*. Popisuje základní vlastnosti těchto systémů a metody, které se používají k testování výkonnosti operačních systémů reálného času. Vybrané testovací metody jsou implementovány a na základě nich je následně porovnána výkonnost operačních systémů reálného času *µC/OS-II* a *µC/OS-III*.

Abstract

This master's thesis is focused on benchmarking of Real-Time Operating Systems *µC/OS-II* and *µC/OS-III*. It describes the basic features of these systems and metrics used for benchmarking of Real-Time Operating Systems. Selected test methods are implemented and based on them are then compared the performance of Real-Time Operating Systems *µC/OS-II* and *µC/OS-III*.

Klíčová slova

Real-time systém, operační systém reálného času, RTOS, testování výkonnosti, µC/OS-II, µC/OS-III, Rhealstone, Thread-Metric, SSC-benchmark

Keywords

Real-Time system, Real-Time Operating System, RTOS, benchmarking, µC/OS-II, µc/OS-III, Rhealstone, Thread-Metric, SSC-benchmark

Citace

Ján Lorenc: Porovnání vlastností a výkonnosti jader uC/OS-II a uC/OS-III, diplomová práce, Brno, FIT VUT v Brně, 2016

Porovnání vlastností a výkonnosti jader uC/OS-II a uC/OS-III

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Josefa Strnadela, PhD. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Ján Lorenc
25. května 2016

Poděkování

Na tomto místě bych rád poděkoval svému vedoucímu Ing. Josefu Strnadelovi, PhD. za jeho odbornou pomoc, užitečné rady a cenné postřehy při tvorbě této diplomové práce.

© Ján Lorenc, 2016.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Základní pojmy **4**
2.1 Systém reálného času . . . 4
2.1.1 Vlastnosti . . . 4
2.1.2 Klasifikace . . . 5
2.1.3 Inverze priorit . . . 5
2.2 Operační systém reálného času . . . 7
2.2.1 Vlastnosti . . . 7
2.2.2 Jádro RTOS . . . 8
2.2.3 Plánovač . . . 8
2.2.4 Objekty jádra . . . 12

3 Stručná charakteristika jader μC/OS-II a μC/OS-III **14**
3.1 Základní vlastnosti . . . 14
3.2 Plánovač . . . 16
3.3 Úlohy . . . 17
3.4 Systémové úlohy . . . 20
3.5 Správa paměti . . . 21
3.6 Semafory . . . 22
3.7 Mutexy . . . 23
3.8 Schránky zpráv . . . 23
3.9 Fronty zpráv . . . 24

4 Přehled metod pro testování výkonnosti RTOS **25**
4.1 Rhealstone . . . 25
4.2 Thread-Metric . . . 28
4.3 SSC-Benchmark . . . 31
4.4 Metody pro testování výkonnosti RT systémů a HW platforem . . . 33
4.4.1 Dhrystone . . . 33
4.4.2 Whetstone . . . 33
4.4.3 Hartstone . . . 33
4.4.4 SPEC CPU 2006 . . . 34
4.4.5 CoreMark . . . 34
4.4.6 Mibench . . . 34
4.4.7 Pababench . . . 34

5 Realizační prostředky **35**
5.1 Vývojová deska FITkit 3 35
5.1.1 Mikrokontrolér MK60DN512VM10 35
5.1.2 Programovací model 36
5.2 Kinetis Design Studio 40

6 Návrh řešení a popis implementace **41**
6.1 Metody a experimenty 41
6.2 Způsob sběru experimentálních dat 41
6.3 Využití realizačních prostředků 42
6.3.1 Kinetis Design Studio 42
6.3.2 Periferie mikrokontroléru 42
6.4 Implementace testovacích metod 43
6.4.1 Thread-Metric 43
6.4.2 Rhealstone 45
6.4.3 SSC-benchmark 47
6.4.4 Vlastní testy 49

7 Vyhodnocení dosažených výsledků **50**
7.1 Thread-Metric 50
7.1.1 μC/OS-II 50
7.1.2 μC/OS-III 51
7.1.3 Srovnání μC/OS-II a μC/OS-III 51
7.2 SSC-benchmark 52
7.2.1 μC/OS-II 52
7.2.2 μC/OS-III 52
7.2.3 Srovnání μC/OS-II a μC/OS-III 54
7.3 Rhealstone 55
7.4 Vlastní testy 57
7.4.1 μC/OS-II 57
7.4.2 μC/OS-III 57
7.4.3 Srovnání μC/OS-II a μC/OS-III 57
7.5 Vyhodnocení a diskuze 58

8 Závěr **61**

Literatura **63**

Přílohy **66**
Seznam příloh 67

A Obsah CD **68**

Kapitola 1

Úvod

Operační systémy reálného času se vyskytují všude kolem nás. Každý den se s nimi setkáváme, i když si to mnohdy ani neuvědomujeme. Operační systémy reálného času jsou totiž používány například pro řízení vestavěných systémů, mezi něž se řadí i běžné domácí spotřebiče jako pračka, mikrovlnná trouba či myčka nádobí a s těmi jste se již určitě setkali. Cílem vestavěného systému však je, aby jej uživatel nevnímal jako počítač, ale skutečně jako spotřebič.

Systémy reálného času obsahují jednu či více úloh, které mohou mít různé priority a časové meze. Úkolem operačního systému reálného času je pak zajistit běh daných úloh. O pořadí vykonávání úloh rozhoduje plánovač, jenž je součástí operačního systému reálného času.

Výkonnost operačního systému může být důležitým kritériem při jeho volbě. Zvláště v případě systémů reálného času, kde musí úlohy přesně dodržovat časové meze svého vykonání.

Cílem této diplomové práce bylo navrhnout a implementovat testovací sadu úloh pro otestování výkonnosti jader operačních systémů *µC/OS-II* a *µC/OS-III* a na základě výsledků implementovaných metod pak zhodnotit výkonnost obou těchto operačních systémů reálného času.

Kapitola 2 obsahuje úvod do základních pojmů týkajících se systémů reálného času. Je zde vysvětlen pojem systému reálného času, shrnuty jeho základní vlastnosti a popsána klasifikace těchto systémů. Dále je obsažena definice operačního systému reálného času, jeho jádra a objektů poskytovaných jádrem.

V kapitole 3 představím dva konkrétní operační systémy reálného času, kterými jsou *µC/OS-II* a *µC/OS-III*. Nejdříve uvedu popis obou systémů, jejich základních vlastností a poté se zaměřím na služby, které poskytují, např. semafory, fronty zpráv apod.

Součástí kapitoly 4 je přehled metod používaných pro otestování výkonnosti operačních systémů reálného času. Dále jsou zde ve stručnosti popsány metody a benchmarky pro testování výkonnosti systémů reálného času a hardwarových platforem.

Kapitola 5 obsahuje popis realizačních prostředků použitých v této práci. Je popsána použitá vývojová deska, mikrokontrolér, kterým je tato vývojová deska osazena a použité vývojové prostředí. Rovněž je uveden stručný úvod do programovacího modelu procesoru ARM Cortex-M4.

Kapitola 6 je zaměřena na popis vlastní implementace testovacích metod. Kromě implementace testovacích metod je rovněž popsán způsob sběru experimentálních dat.

Obsahem kapitoly 7 je vyhodnocení naměřených dat. Nejprve jsou přehledně uvedena naměřená data, která jsou následně diskutována.

Kapitola 2

Základní pojmy

V této kapitole vysvětlím pojem systém reálného času včetně jeho základních vlastností a klasifikace. Na tomto místě představím také pojem operační systém reálného času, přičemž se zaměřím především na jeho plánovač, jádro a jím poskytované služby. Zde uváděné pojmy využiji v dalších částech této práce.

2.1 Systém reálného času

Systém reálného času, dále jen *RT* (*Real-Time*) systém, je definován jako systém, který reaguje na externí události přicházející z okolního prostředí v reálném čase, přičemž podstatným kritériem je včasnost odezvy. Externí události mohou být synchronní či asynchronní. Synchronní událost je taková událost, jejíž čas výskytu lze dopředu předpovědět. Oproti tomu asynchronní události jsou generovány v předem nepředvídatelných okamžicích. Správná funkce *RT* systému je pak dána nejen správností produkovaných výstupů, ale také včasností jejich produkce [1].

2.1.1 Vlastnosti

- **Časová omezení** – Podstatným kritériem *RT* systému je včasnost odezvy systému. Úloha má běžně nastavenu časovou mez, do jejíhož uplynutí musí dokončit svůj běh a poskytnout výstupy.
- **Korektnost** – Korektnost *RT* systému je dána nejen správností produkovaných výpočetních výstupů, ale také včasností produkce těchto výstupů. Správný výpočetní výsledek nedodaný ve stanovém čase je pak považován za chybný výstup.
- **Bezpečnost** – *RT* systém je bezpečný, pokud při jeho selhání nedojde k žádnému poškození.
- **Konkurentnost** – často přichází více událostí současně. *RT* systém musí zpracovávat tyto události konkurentně, jinak by mohlo dojít ke ztrátě dat událostí, což by mohlo vést k nekorektní práci *RT* systému.
- **Kritičnost úloh** – Tento parametr vyjadřuje náklady způsobené selháním úlohy. Čím vyšší má úloha časovou kritičnost, tím by měla být spolehlivější.
- **Reaktivnost** – Reaktivní systém je takový systém, který reaguje na události z okolního prostředí.

- **Stabilita** – *RT* systém by neměl nikdy selhat, ale měl by pokračovat alespoň zpracováváním kritičtějších úloh, přičemž časové meze nekritických úloh nemusí být dodrženy [2].

2.1.2 Klasifikace

RT systémy můžeme rozdělit dle dvou hledisek, jejichž popis následuje.

Dle způsobu, jakým detekují změny v prostředí a reagují na ně

Podle tohoto kritéria můžeme rozdělit RT systémy do dvou skupin [3]:

- **Spouštěné časem (*Time-Trigerred*)** – jsou řízeny zevnitř. Spouštění úloh je řízeno globálními hodinami *RT* systému.

- **Řízené událostmi (*Event-Driven*)** – jsou řízeny zvnějšku. Úlohy jsou prováděny asynchronně v závislosti na přicházejících událostech.

Dle následků plynoucích z nedodržení časových mezí

Podle tohoto kritéria dělíme *RT* systémy do tří kategorií, *Soft*, *Firm* a *Hard* [4].

- **Soft RT systém** – systém, u něhož při nedodržení časových mezí nedojde k selhání systému, ale pouze k dočasnému snížení výkonu *RT* systému. Příkladem je hlasové ovládání TV, kde nedodržení časových mezí nevede k selhání systému, ale pouze k degradaci výkonu.

- **Firm RT systém** – je systém, u něhož nedodržení několika málo časových mezí nevede k selhání systému. Nedodržení většího počtu časových mezí však už může vést k selhání systému. Příkladem tohoto systému je navigační systém autonomní sekačky na trávu, kde nedodržení časových mezí vede k nekoordinovanému pohybu sekačky, důsledkem čehož jsou místo trávy posekány užitečné rostliny.

- **Hard RT systém** – je takový systém, kde nedodržení byť jen jediné časové meze vede k selhání systému, které může způsobit katastrofální následky. Příkladem je navigační systém autonomního automobilu, kde nedodržení jediné meze odezvy vede k nehodě automobilu, což může mít tragické následky.

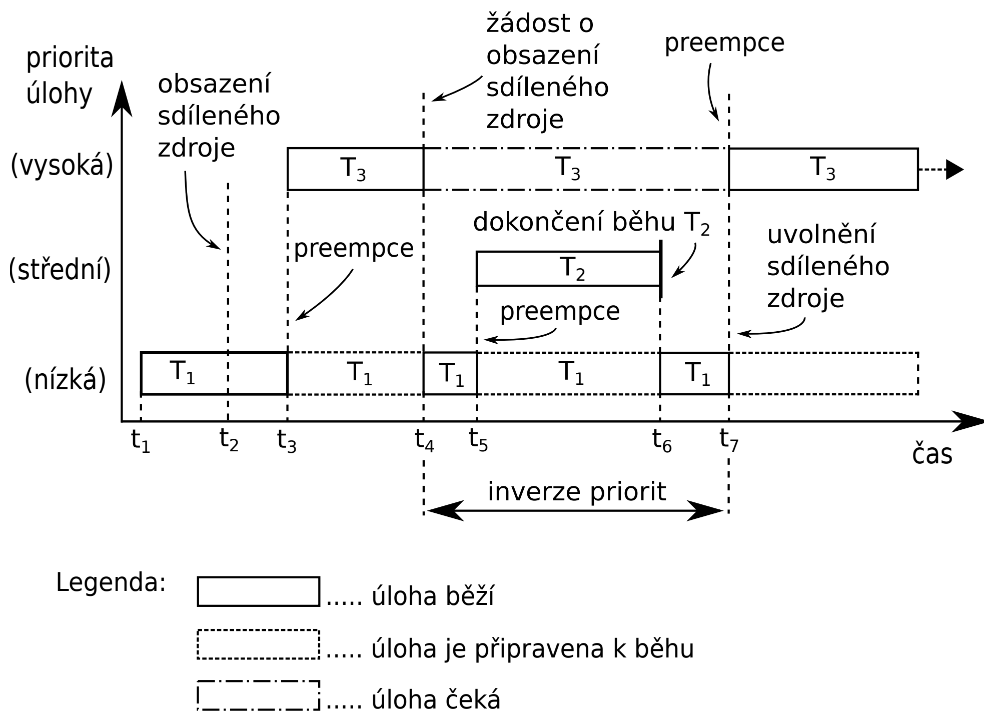
2.1.3 Inverze priorit

Inverze priorit je problém v oblasti *RT* systémů, kdy výšeprioritní úloha je blokována nížeprioritní úlohou v důsledku vlastnictví sdíleného zdroje nížeprioritní úlohou, který chce obsadit právě výšeprioritní úloha [1]. Tuto situaci ilustruje obrázek 2.1.

Na obrázku 2.1 figurují tři úlohy, T_1, T_2 a T_3. Úloha T_1 má nejnižší prioritu, úloha T_2 má vyšší prioritu, než úloha T_1 a úloha T_3 má nejvyšší prioritu.

V čase t_1 je k běhu připravena úloha T_1. Protože má aktuálně nejvyšší prioritu, je ihned spuštěna. V čase t_2 obsadí tato úloha sdílený zdroj.

V čase t_3 je k běhu připravena úloha T_3, která má vyšší prioritu, než aktuálně běžící úloha T_1, proto dojde k přepnutí kontextu a běží tedy úloha T_3. Tato úloha se v čase t_4 pokusí obsadit sdílený zdroj, jenž je však již vlastněn úlohou T_1 a úloha T_3 je tedy pozastavena a čeká na uvolnění sdíleného zdroje. Ihned je spuštěna opět úloha T_1.



Obrázek 2.1: Inverze priorit

V čase t_5 je k běhu připravena úloha T_2, jež má vyšší prioritu než úloha T_1, znovu tedy dojde k přepnutí kontextu. Úloha T_2 dokončí svůj běh v čase t_6. Je tedy spuštěna jiná úloha s aktuálně nejvyšší prioritou ze seznamu připravených úloh, tedy opět úloha T_1.

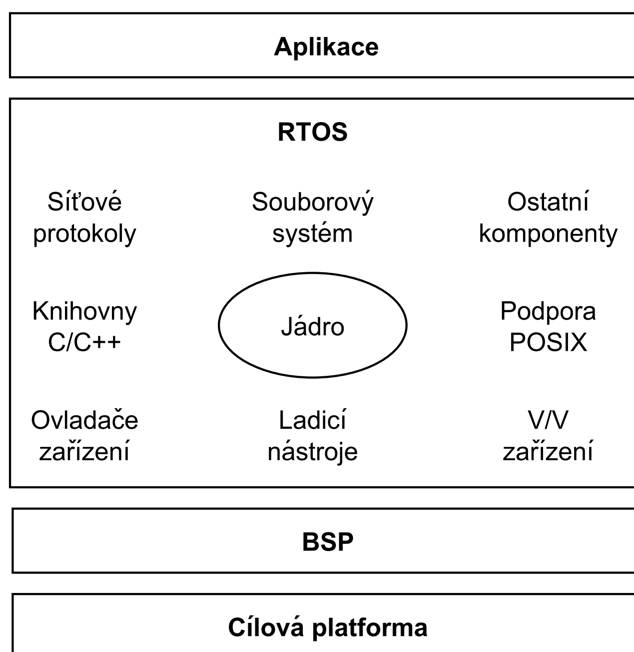
V čase t_7 úloha T_1 dokončí práci se sdíleným zdrojem, následkem čehož je úloha T_3 převedena ze seznamu čekajících úloh do seznamu úloh připravených k běhu. A protože má vyšší prioritu, než úloha T_1, dojde k preempci a je spuštěna úloha T_3, jež tak může přistoupit k již uvolněnému sdílenému zdroji.

V rozmezí časů t_4 až t_7 tedy došlo k jevu inverze priorit, jak je znázorněno i na obrázku 2.1. Tento jev se stává nebezpečným především v případě, kdy středněprioritní úloha T_2 provádí dlouhodobou nepodstatnou činnost, čímž tedy blokuje důležitější úlohu T_3, jež může být klíčová pro chod celého systému. Je důležité si rovněž uvědomit, že mezi časy t_5 až t_6 může být k běhu připraveno více středněprioritních úloh, které tak budou postupně spouštěny, čímž bude neustále odkládáno dokončení úlohy T_1. V nejhorším případě nebude úloha T_1 dokončena nikdy v důsledku existence úloh s vyšší prioritou, než je priorita úlohy T_1, připravených k běhu.

Řešením inverze priorit je dočasné zvýšení priority úlohy, která vlastní sdílený zdroj a blokuje tak výšeprioritní úlohu, na úroveň právě této výšeprioritní úlohy. Konkrétně na obrázku 2.1 to znamená, že v čase t_4 by byla zvýšena priorita úlohy T_1 na úroveň priority úlohy T_3, následkem čehož by úloha T_1 nebyla v čase t_5 přerušena úlohou T_2, ale mohla by dokončit svou práci a uvolnit sdílený zdroj. Poté by byla priorita úlohy T_1 opět snížena na původní úroveň.

2.2 Operační systém reálného času

Operační systém reálného času (*RTOS*, *Real-Time Operating System*) se od konvenčního operačního systému liší tím, že úlohy musí být vykonány v předem definovaných časových intervalech (*deadlines*). Nedodržení těchto časových mezí může mít až katastrofální dopady na okolí v závislosti na systému, jež je *RTOS* řízen. *RTOS* jsou používány například v robotice či v oblasti vestavěných systémů. Obecné schéma *RTOS* znázorňuje obrázek 2.2. Základním stavebním prvkem *RTOS* je jádro [1].



Obrázek 2.2: Obecné schéma RTOS [1]

2.2.1 Vlastnosti

Vlastnosti *RT* systémů uvedené v sekci 2.1.1 můžeme vztáhnout i k operačním systémům reálného času. Mimo tyto vlastnosti požadujeme od *RTOS* také následující vlastnosti:

- **Spolehlivost** – Systém by měl být schopen pracovat delší dobu bez zásahu člověka. Spolehlivý systém je takový systém, který je dostupný, tzn. poskytuje požadované služby, a nezpůsobuje chyby [1].

- **Determiničnost** – Pojem determiničnost je spojen s předvídatelným chováním *RTOS*, kdy k dokončení jednotlivých volání služeb *RTOS* dochází v předem známých časových intervalech [1].

- **Výkonnost** – Jak již bylo řečeno v sekci 2.1.1, *RT* systém musí dodržovat stanovená časová omezení provádění úloh. Čím více je takových časových omezení a čím jsou intervaly mezi nimi kratší, tím musí být *RT* systém výkonnější. Výkonnost *RT* systému je pak dána kombinací výkonnosti HW i SW. V důsledku toho klademe požadavky na výkonnost i na použitý *RTOS*. Výkonnost *RTOS* pak často měříme jako dobu trvání jednotlivých služeb, které *RTOS* poskytuje [1].

- **Kompaktnost** – Kompaktnost je dána návrhovými a cenovými omezeními. Např. mobilní telefon by měl být malý, lehce přenosný a relativně levný. V takových *RT* systémech, kde je použitý HW limitován rozměry a cenou, by měl být *RTOS* pokud možno malý a efektivní [1].
- **Škálovatelnost** – Protože *RTOS* může být použit v různě složitých aplikacích, bylo by vhodné, aby v základu obsahoval pouze nezbytně nutné součásti a poskytované služby, přičemž další funkcionalitu by bylo možné volitelně přidávat a odebírat dle požadavků a potřeb, což povede také k úspoře paměti [1].
- **Preemptivnost** – Při příchodu výšeprioritní úlohy musí být neprodleně přerušena právě běžící nížeprioritní úloha a řízení předáno této výšeprioritní úloze. Doba, po kterou čeká výšeprioritní úloha, než je jí přidělen procesor, se nazývá doba preempce úlohy. Požadavkem je, aby tato doba byla co nejnižší, typicky několik mikrosekund [5].
- **Podpora prioritních hladin** – Operační systém reálného času musí podporovat statické prioritní hladiny, abychom mohli specifikovat důležitost úlohy. Prioritní hladinu nazveme statickou, pokud operační systém samovolně nemění prioritu úlohy, kterou jí přiřadil programátor. Čím vyšší má úloha prioritu, tím více je obvykle důležitá pro chod aplikace a musí být tedy vykonána co nejdříve [5].
- **Podpora generování hodinového signálu a časovačů** – Služby generování hodinového signálu a časovačů s adekvátním rozlišením jsou jedním z nejčastějších problémů v *RT* systémech. Především v *Hard RT* systémech je vyžadováno rozlišení časovačů v řádu několika mikrosekund [5].
- **Rychlá odezva přerušení** – Doba odezvy přerušení je definována jako doba mezi výskytem přerušení a spuštěním odpovídající *ISR* (*Interrupt Service Routine*). Obvykle je rovněž vyžadována podpora vnořování přerušení (*interrupt nesting*), což znamená, že obsluha přerušení může být přerušena vznikem jiného přerušení. Doba odezvy přerušení musí být nižší, než několik mikrosekund [5].

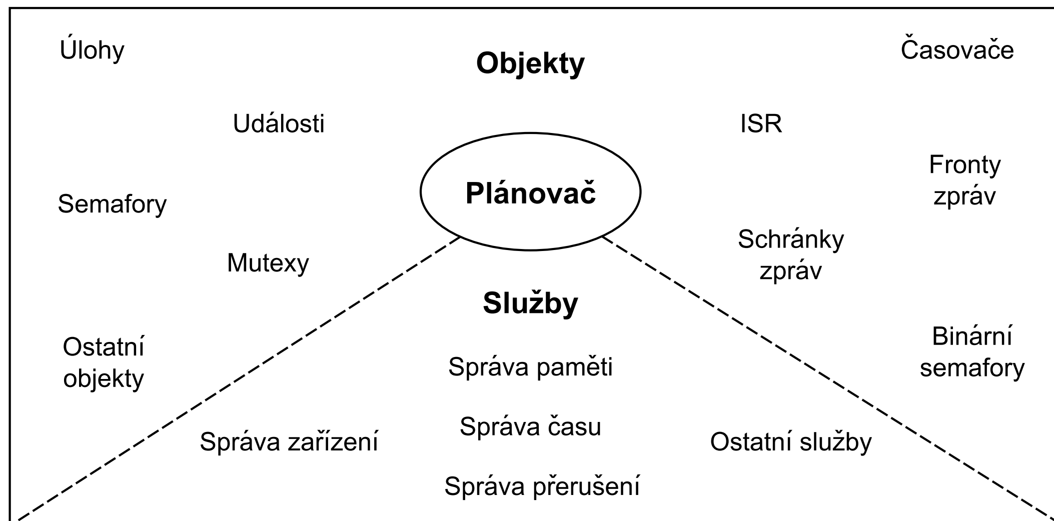
2.2.2 Jádro RTOS

Jak již bylo řečeno v sekci 2.2, jádro můžeme nazvat srdcem *RTOS*. Jádro v sobě zahrnuje mechanismy plánování úloh, prostředky pro správu paměti, synchronizaci a komunikaci úloh a podobně [1]. Schéma jádra je znázorněno na obrázku 2.3.

2.2.3 Plánovač

Plánovač je součástí jádra a obsahuje algoritmy, které určí, kdy bude která úloha vykonávána.

Plánovače můžeme rozdělit na preemptivní a nepreemptivní. Preemptivní plánovač vždy přiřazuje procesor úloze s nejvyšší prioritou. Pokud je připravena k běhu výšeprioritní úloha, preemptivní plánovač ihned odebere procesor aktuálně běžící úloze a naplánuje běh výšeprioritní úlohy. Oproti tomu při nepreemptivním plánování nelze úloze odebrat procesor. Ta se jej musí vzdát sama a až poté může být spuštěna další úloha [1].



Obrázek 2.3: Schéma jádra RTOS [1]

Plánovatelné entity

Předmětem plánování jsou tzv. plánovatelné entity, jež můžeme charakterizovat jako objekty jádra soutěžící o procesorový čas. Nejčastější plánovatelnou entitou v *RTOS* je úloha. Úloha je vlákno obsahující posloupnost plánovatelných instrukcí. V některých *RTOS* se můžeme setkat s pojmem proces. Procesy jsou podobné úlohám s tím rozdílem, že nabízejí lepší prostředky ochrany paměti [1].

Víceúlohovost

Víceúlohovost, neboli *multitasking*, je schopnost operačního systému reálného času zpracovávat více úloh současně. Ve skutečnosti běží vždy pouze jedna úloha, dojem víceúlohového zpracování je vytvářen pomocí rychlého přepínání kontextu mezi jednotlivými úlohami. Přepínání kontextu má na starosti jádro *RTOS*, respektive jedna z jeho částí – plánovač.

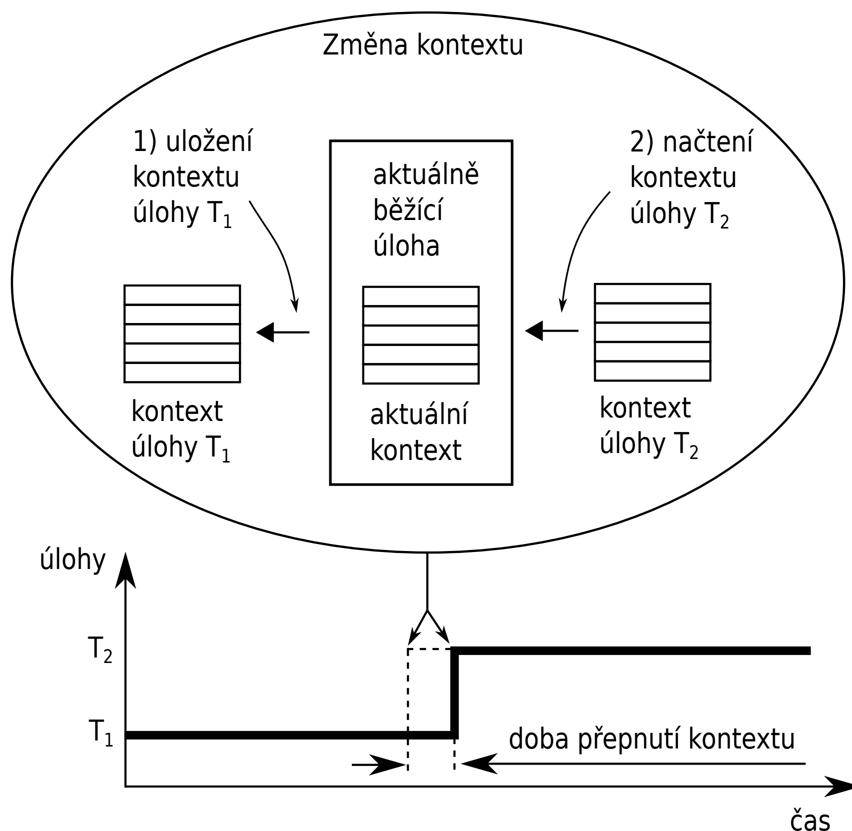
Multitasking zlepšuje využití procesoru, jelikož pokud právě běžící úloha čeká na nějakou událost, je procesor přidělen jiné úloze [1].

Přepnutí kontextu

Kontext úlohy slouží k uchování informací o jejím stavu. Součástí kontextu úlohy je obsah jejího adresového prostoru a obsah registrů, například programového čítače a podobně. Každá úloha má svůj vlastní kontext. K přepnutí kontextu dochází v situaci, kdy plánovač přepne z jedné úlohy na jinou.

Přepnutí kontextu sestává z několika kroků. Mějme dvě úlohy, úlohu T_1 a úlohu T_2. Přepnutí kontextu z úlohy T_1 na úlohu T_2 typicky zahrnuje uložení kontextu úlohy T_1, následně je obnoven kontext úlohy T_2 a poté je úloze T_2 předáno řízení. Schéma změny kontextu zachycuje obrázek 2.4.

Čas potřebný k přepnutí z jedné úlohy na jinou je označován jako doba přepnutí kontextu. Interval mezi přepínáním úloh musíme volit tak, aby potřebná režie k přepnutí nepřevážila užitečný běh úloh, tzn. aby většina procesorového času nebyla spotřebována na přepínání mezi úlohami [1].



Obrázek 2.4: Schéma změny kontextu [1]

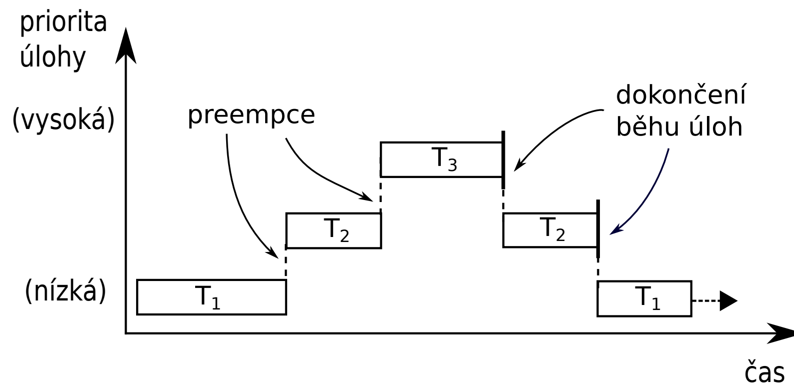
Dispečer

Dispečer je jednou z částí plánovače. Jeho úkolem je zajištění přepnutí kontextu a rovněž je zodpovědný za tok řízení výpočtu. Při vyvolání systémového volání je tok řízení předán jádru pro vykonání některé ze systémových rutin jádra. Po vykonání systémové rutiny dispečer zajistí předání toku řízení některé z aplikačních úloh, kterou vybere plánovač [1].

Plánovací algoritmy

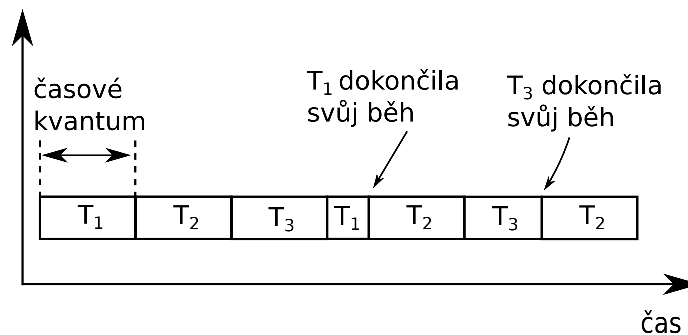
Plánovač vybírá úlohu, která bude spuštěna, na základě plánovacího algoritmu. Existuje celá řada plánovacích algoritmů. Mezi základní patří algoritmus *FCFS/FIFO* (*First Come First Serve/First In First Out*), kdy je procesor přiřazen úloze ze začátku fronty úloh připravených k běhu. Z dalších plánovacích algoritmů můžeme zmínit například algoritmus *SJF* (*Shortest Job First*), jenž přiděluje procesor úloze s nejkratší dobou běhu. V *RTOS* se nejčastěji setkáme s preemptivním plánováním založeném na prioritách úloh či s algoritmem *Round-Robin* [1].

- **Preemptivní plánování založené na prioritách** – Základním předpokladem pro využití tohoto algoritmu je, že každé úloze je přiřazena určitá priorita. Algoritmus pak k běhu naplánuje vždy úlohu s nejvyšší prioritou. Fungování tohoto algoritmu je přehledně vyobrazeno na obrázku 2.5.



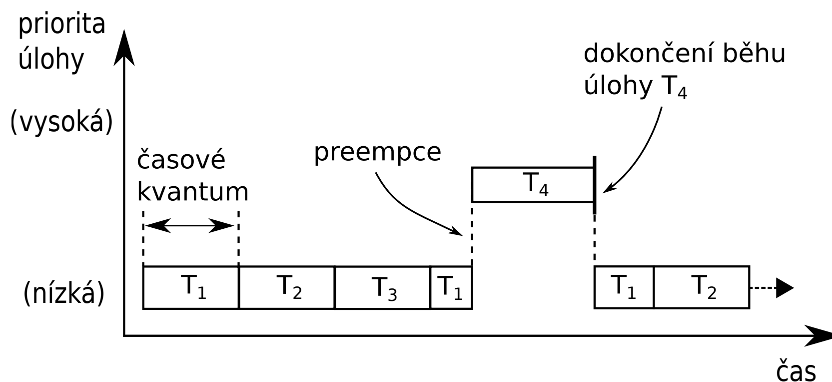
Obrázek 2.5: Preemptivní plánování

- ***Round-Robin*** – Principem tohoto algoritmu je přiřazení časového kvanta všem úlohám, jimž je pak na toto časové kvantum cyklicky přidělován procesorový čas. Velikost časového kvanta může být pro všechny úlohy shodná nebo nastavena individuálně. Algoritmus *Round-Robin* nebere v potaz prioritu úloh, proto je pro použití v oblasti *RTOS* nevhodný, nicméně velice často se používá v kombinaci s preemptivním plánováním. Obrázek 2.6 ukazuje princip tohoto algoritmu.



Obrázek 2.6: Plánování Round-Robin

- **Kombinace *Round-Robin* a preemptivního plánování** – Plánování úloh je prioritně prováděno pomocí algoritmu preemptivního plánování. Může však dojít k situaci, kdy má aktuálně nejvyšší prioritu více úloh připravených k běhu současně. V takovém případě přijde ke slovu algoritmus *Round-Robin*, jenž každé z těchto úloh přiřadí časové kvantum, po které bude této úloze přiřazen procesor. Po vypršení časového kvanta je procesor úloze odebrán, tato úloha je zařazena na konec seznamu úloh připravených k běhu se stejnou prioritou a procesor je přiřazen další úloze v pořadí. Tento proces se cyklicky opakuje, dokud svůj běh nedokončí všechny tyto úlohy nebo není k běhu připravena úloha s vyšší prioritou. V takovém případě se opět uplatní preempce a procesor je neprodleně přiřazen úloze s vyšší prioritou. Schéma tohoto typu plánování úloh zobrazuje obrázek 2.7.



Obrázek 2.7: Kombinace preemptivního plánování a Round-Robin

2.2.4 Objekty jádra

Úlohy

Úlohy, někdy též nazývané vlákna, jsou konkurentní objekty, které soutěží o procesorový čas. Aplikace je většinou rozčleněna do několika menších úloh. Jak již bylo řečeno v sekci 2.2.3, úloha patří mezi plánovatelné objekty.

Úloha je definována několika základními parametry, jako jsou unikátní identifikátor, jméno, velikost zásobníku, adresa vrcholu zásobníku, priorita, stav, *TCB* (*Task Control Block*). V závislosti na systému může být těchto parametrů samozřejmě více.

Každá úloha se vždy nachází v nějakém stavu. Obecně rozlišujeme tři typy stavů úloh:

- ***Ready*** **(připravená)** – úloha je připravena k běhu, nicméně právě běží úloha s vyšší prioritou.
- ***Running*** **(běžící)** – úloha má nejvyšší prioritu, a proto je spuštěna.
- ***Blocked*** **(pozastavená)** – úloha čeká na nějakou událost, například na přidělení systémového prostředku, který však aktuálně není volný. Aby nebyl procesor blokován, je takováto úloha pozastavena a procesor je přidělen jiné úloze.

V konkrétních *RTOS* se můžeme setkat s dalšími stavy úloh, které jsou rozšířením výše uvedených tří základních.

Úloha má vždy jednu z následujících struktur:

- ***Run-to-Completion*** – po vykonání všech příkazů z těla úlohy je tato úloha odstraněna. Pseudokód úlohy tohoto typu je uveden v kódu 2.1.
- **Nekonečná smyčka** – provádění těla úlohy je neustále cyklicky opakováno. Schéma této úlohy zobrazuje kód 2.2.

Semafory

Semafor je základní synchronizační prostředek. Slouží k synchronizaci dvou úloh či k řízení přístupu ke sdílenému zdroji (*mutual exclusion*). Semafor se typicky skládá z hodnoty a seznamu čekajících úloh. Semafory můžeme rozdělit na binární a čítací. Jak již název

```
Úloha () {
  Inicializace
  Uživatelský kód
  Odstranění úlohy
}
```

Kód 2.1: Pseudokód úlohy Run-to-Completion

```
Úloha () {
  Inicializace
  Nekonečný cyklus {
    Tělo cyklu
    Blokující volání
  }
}
```

Kód 2.2: Pseudokód úlohy typu nekonečná smyčka

napovídá, hodnota binárního semaforu může nabývat pouze dvou hodnot (0 a 1). Pokud hodnota semaforu může nabývat více hodnot, jedná se o čítací semafor.

Pro práci se semaforem se používají dvě operace, `wait` a `signal`. `Wait` slouží pro zamknutí semaforu. Po provedení této operace je hodnota semaforu snížena o jedničku za předpokladu, že již není 0. V takovém případě je úloha žádající o semafor zablokována a čeká, dokud není semafor uvolněn. Operace `signal` slouží k odemknutí semaforu a po jejím provedení je hodnota semaforu zvýšena o jedničku [1].

Mutexy

Mutex je speciálním typem binárního semaforu, který zahrnuje mechanismy pro řešení problémů vztahujících se k vzájemnému vyloučení, např. problém vzniku inverze priorit. Stejně jako binární semafor se může nacházet ve dvou stavech, *locked* (0) a *unlocked* (1). Po vytvoření se mutex nachází ve stavu *unlocked*. Jakmile jej úloha zamkne, přechází do stavu *locked*. Po odemknutí se vrací do stavu *unlocked* a může být zamknut jinou úlohou [1].

Schránky zpráv

Schránka zpráv je objekt, který slouží ke komunikaci a výměně dat mezi úlohami či mezi *ISR* (*Interrupt Service Routine*) a úlohou. Schránka zpráv může současně obsahovat pouze jednu zprávu. Jakmile odesílatel zprávu odešle, zpráva je udržována ve schránce zpráv do doby, než si ji příjemce vyzvedne [6].

Fronty zpráv

Fronta zpráv je objekt jádra, který stejně jako schránka zpráv slouží pro zasílání zpráv mezi úlohami či mezi *ISR* a úlohou. Rozdílem oproti schránce zpráv je, že fronta zpráv může v jednu chvíli obsahovat více zpráv. Maximální počet zpráv, které může fronta současně obsahovat je určen délkou fronty.

Součástí fronty zpráv jsou dva seznamy úloh. Jeden z nich slouží pro udržování úloh čekajících na zprávu, pokud je fronta prázdná. Druhý seznam udržuje seznam úloh čekajících na možnost odeslání zprávy, pokud je fronta zpráv zaplněna [1].

Kapitola 3

Stručná charakteristika jader μC/OS-II a μC/OS-III

Tato kapitola obsahuje přehled základních vlastností a charakteristik operačních systémů reálného času *μC/OS-II* a *μC/OS-III*. V další části této kapitoly jsou popsány základní služby poskytované oběma systémy včetně popisu, v čem se liší. Popsány jsou tak například úlohy, semafory, fronty zpráv či správa paměti.

3.1 Základní vlastnosti

μC/OS-II

μC/OS-II [6] je operační systém reálného času, jehož autorem je Jean J. Labrosse. Vznikl v roce 1998 a navázal na svého úspěšného předchůdce *μC/OS*. *μC/OS-II* byl implementován s ohledem na zpětnou kompatibilitu se systémem *μC/OS*.

Charakteristiky

- Byl vytvořen pro použití ve vestavěných systémech.
- Vzhledem k tomu, že byl implementován v jazyce ANSI C, lze jej přenést na celou řadu různých mikroprocesorů.
- Systém je škálovatelný, což znamená, že můžeme používat pouze služby, které potřebujeme a ostatní zakážeme, čímž uspoříme kapacitu *RAM* (*Random Access Memory*) i *ROM* (*Read Only Memory*) paměti.
- Zdrojový kód lze pro nekomerční účely stáhnout z webových stránek[1] výrobce.
- Je preemptivní, vždy tedy běží úloha s nejvyšší prioritou.
- Maximální počet úloh je 255, přičemž každá musí mít unikátní prioritu. Tzn. v systému existuje 255 prioritních hladin.
- Jádro je deterministické. Vždy tedy víme, jak dlouho bude trvat provádění konkrétní služby.

[1] http://micrium.com/

- Poskytuje velké množství systémových služeb, např. semafory, mutexy, schránky zpráv, fronty zpráv apod.
- Velikost zásobníku každé úlohy lze nastavit individuálně dle potřeb, což vede k úspoře paměti *RAM*.

μC/OS-III

μC/OS-III [7] je již třetí verzí operačního systému reálného času *μC/OS*. Je stále relativně nový, vznikl v roce 2009. Jeho autorem je opět Jean. J. Labrosse. Oproti *μC/OS-II* obsahuje řadu vylepšení. Těmi nejzásadnějšími jsou optimalizace plánovače na úrovni assembleru a konfigurovatelnost systému za běhu.

Charakteristiky

- Byl vytvořen pro použití ve vestavěných systémech.
- Byl implementován ve vysoce přenositelném jazyce ANSI-C, lze tedy vytvořit jeho port pro velké množství mikroprocesorů. Pro *μC/OS-III* lze také jednoduše upravit již existující porty *μC/OS-II*.
- Systém je škálovatelný, což znamená, že můžeme používat pouze služby, které potřebujeme a ostatní zakážeme, čímž uspoříme kapacitu *RAM* i *ROM* paměti.
- Zdrojový kód lze pro nekomerční účely stáhnout z webových stránek[2] výrobce.
- Je preemptivní, vždy tedy běží úloha s nejvyšší prioritou.
- Maximální počet úloh v systému není omezen, respektive je omezen pouze kapacitou paměti.
- Maximální počet prioritních hladin v systému není omezen. Pro většinu aplikací však dostačuje počet prioritních hladin v rozmezí 32 až 256. Na jedné prioritní hladině může existovat více úloh.
- Pro plánování úloh se stejnou prioritou používá algoritmus *Round-Robin*, který každé z této úloh přiřadí časové kvantum, po které bude úloze přidělen procesor.
- Jádro je deterministické. Vždy tedy víme, jak dlouho bude trvat provádění konkrétní služby.
- Poskytuje velké množství systémových služeb, např. semafory, mutexy, schránky zpráv, fronty zpráv apod.
- Velikost zásobníku každé úlohy lze nastavit individuálně dle potřeb, což vede k úspoře paměti *RAM*.
- Obsahuje vestavěné nástroje pro měření výkonnosti. Lze měřit např. dobu trvání úlohy, počet běhů úlohy, využití CPU, využití zásobníku každé úlohy a mnoho dalších.
- Oproti *μC/OS-II* je konfigurovatelný za běhu. Všechny objekty jádra jsou alokovány až za běhu systému.

[2] http://micrium.com/

3.2 Plánovač

Obecné informace o plánovači a jeho významu v operačních systémech reálného času jsou uvedeny v sekci 2.2.3.

μC/OS-II

μC/OS-II je založeno na prioritách, procesor je tedy vždy přidělen úloze s nejvyšší prioritou. Plánovač *μC/OS-II* je preemptivní, což znamená, že procesor může být úloze kdykoli odebrán. Děje se tak v případě, kdy je k běhu připravena úloha s vyšší prioritou, než má aktuálně běžící úloha. V tomto případě je tedy procesor neprodleně odebrán úloze s nižší prioritou a přidělen výšeprioritní úloze.

K plánování na úrovni úloh slouží funkce `OS_Sched()`, k plánování na úrovni *ISR* (*Interrupt Service Routine*) slouží funkce `OSIntExit()`. K přepnutí kontextu pak slouží úloha `OS_TASK_SW()`, která je volána vždy v plánovači [6].

μC/OS-III

RTOS *μC/OS-III* je, stejně jako *μC/OS-II*, založen na prioritách a plánovač je preemptivní. Pro plánovač systému *μC/OS-III* tak platí stejné principy jako pro *μC/OS-II*, jež jsou uvedeny v sekci 3.2.

Oproti *μC/OS-II* může v *RTOS μC/OS-III* existovat více úloh se stejnou prioritou. Je tedy nutné plánovač rozšířit o tuto možnost. V systému *μC/OS-III* je k tomuto účelu použito plánování metodou *Round-Robin*. Každé úloze je přiřazeno časové kvantum, po které je jí přidělen procesor a úloha tak běží. Po uplynutí tohoto kvanta je procesor přidělen jiné úloze se stejnou prioritou (za předpokladu, že k běhu není připravena úloha s vyšší prioritou). Časové kvantum přidělené úlohám lze nastavit pomocí volání funkce `OSSchedRoundRobinCfg()`. Každé úloze přitom může být přiděleno jiné časové kvantum, které specifikujeme při vytvoření dané úlohy v systému. Časové kvantum úlohy můžeme měnit také za běhu pomocí funkce `OSTaskTimeQuantaSet()`. Pokud úloha dokončí svou práci dříve a nepotřebuje tak celé přidělené časové kvantum, může se procesoru kdykoli vzdát voláním funkce `OSSchedRoundRobinYield()` [7].

Body plánování

Jsou to body, ve kterých dochází k automatickému plánování úloh.

- Při vytvoření nové úlohy.
- Při smazání úlohy.
- Úloha volá funkci `OSTimeDly()` nebo `OSTimeDlyHMSM()`.
- Úloha změní svou prioritu či prioritu jiné úlohy.
- Úloha je pozastavena pomocí funkce `OSTaskSuspend()`.
- Opětovné spuštění úlohy, která byla pozastavena.
- Plánovač je odemčen voláním funkce `OSSchedUnlock()`.
- Uživatel volá funkci `OSSched()`.

- Úloha se vzdá svého časového kvanta pomocí funkce `OSSchedRoundRobinYield()` – platí pouze pro *µC/OS-III* [7].

3.3 Úlohy

Princip správy úloh je shodný pro *µC/OS-II* i *µC/OS-III*. Úloha musí být implementována buď jako *Run-to-completion*, což znamená, že po provedení své práce se sama odstraní, nebo jako *nekonečná smyčka*, ve které úloha čeká na příchozí události a poté je zpracovává. Častěji se používá druhá varianta [6]. Pojem úloha je vysvětlen v sekci 2.2.4.

µC/OS-II

Schéma každé úlohy je `void Task(void *pdata)`. Je tedy stejné jako u kterékoli funkce jazyka C. Návratová hodnota úlohy musí být vždy `void`. Úloha má jeden parametr typu ukazatel na `void`, pomocí něhož můžeme úloze předávat libovolná data.

Maximální počet úloh v *µC/OS-II* je 255. Každá úloha musí mít unikátní prioritu, z čehož tedy vyplývá, že v *µC/OS-II* existuje celkem 255 prioritních hladin. Nejnižší priorita je vyhrazena pro *Idle* úlohu (3.4), aplikace tak může obsahovat maximálně 254 uživatelských úloh. Autor *µC-OS-II* však doporučuje nepoužívat čtyři nejvyšší a čtyři nejnižší priority pro případ využití těchto prioritních hladin v dalších verzích *µC/OS-II*, čímž se dostáváme na hodnotu 247 vlastních uživatelských úloh.

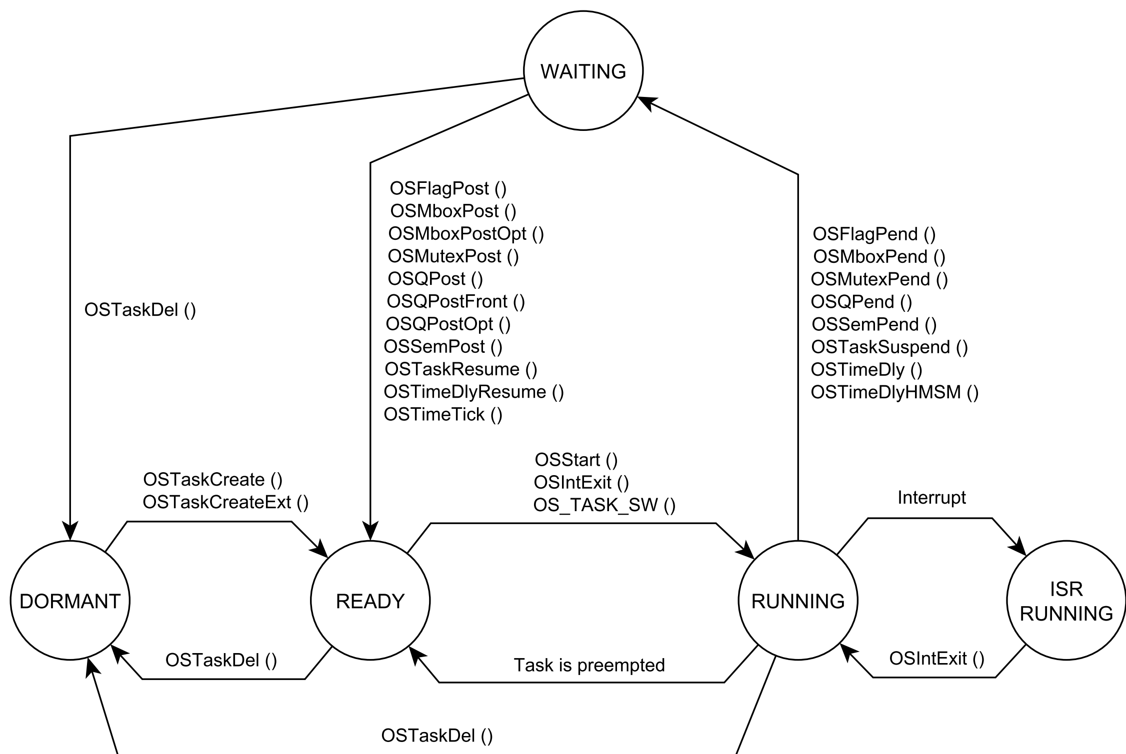
Úlohu lze vytvořit pomocí dvou funkcí, `OSTaskCreate()` a `OSTaskCreateExt()`. Jak již název napovídá, funkce `OSTaskCreateExt()` je rozšířenou verzí funkce `OSTaskCreate()`. Umožňuje nastavení většího množství parametrů úlohy, tedy větší flexibilitu. Zatímco funkce `OSTaskCreate()` vyžaduje čtyři paramtery, `OSTaskCreateExt()` devět parametrů. Úloha může být vytvořena před startem aplikace či za běhu. Po vytvoření úlohy se úloha nachází ve stavu *READY* (připravená). Do stavu *DORMANT* (odložená) se může úloha dostat pomocí volání funkce `OSTaskDel()`, čímž dojde k jejímu odstranění.

Po zavolání funkce `OSStart()` je spuštěn běh aplikace, plánovač vybere úlohu s nejvyšší prioritou a tato úloha je spuštěna, tzn. přejde do stavu *RUNNING* (běžící). Ostatní úlohy ve stavu *READY* pak čekají na přidělení procesoru, dokud všechny úlohy s vyšší prioritou nejsou odstraněny nebo ve stavu *WAITING* (čekající).

Běžící úloha může uspat sebe sama na předem stanovenou dobu pomocí volání funkcí `OSTimeDly()` či `OSTimeDlyHMSM()`. Po zavolání jedné z těchto funkcí daná úloha přejde do stavu *WAITING* a plánovač vybere k běhu úlohu s nejvyšší prioritou z úloh ve stavu *READY*. Po uplynutí doby, na kterou je úloha uspána, je daná úloha opět převedena do stavu *READY* pomocí interní funkce *µC/OS-II* `OSTimeTick()` a opět se může ucházet o přidělení procesoru. Do stavu *WAITING* se může právě běžící úloha dostat také voláním funkcí `OSSemPend()`, `OSMutexPend()`, `OSMboxPend()`, `OSQPend()`.

Pokud jsou v aplikaci povolena přerušení, po jeho vyvolání přejde právě běžící úloha (*RUNNING*) do stavu *ISR RUNNING*. Po obsloužení přerušení je opět spuštěna úloha s nejvyšší prioritou, nemusí jí tedy již být přerušená úloha ve stavu *ISR RUNNING*.

Pokud není žádná úloha připravena k běhu (*READY*), je spuštěna systémová úloha `OS_IdleTask()` [6]. Stavový diagram úlohy v *µC/OS-II* je zobrazen na obrázku 3.1.



Obrázek 3.1: Stavový diagram úloh v µC/OS-II [6]

µC/OS-III

Základní schéma úlohy je stejné jako u *µC/OS-II*, tedy `void Task(void *p_arg)`. Návratová hodnota každé úlohy je `void` a parametr úlohy `p_arg` je typu ukazatel na `void`. Tento parametr slouží k předávání dat úloze. Může být také využit pro vytvoření více identických úloh, které sice mají stejný kód, ale různou dobu běhu.

Oproti *µC/OS-II* není omezen maximální počet úloh v aplikaci, respektive je omezen pouze velikostí paměti, ať už programové či datové. Také počet prioritních hladin je teoreticky neomezený, nicméně v praxi bývá tato hodnota často omezena na 64, jelikož většinou postačuje. Další podstatnou změnou v porovnání s *µC/OS-II* je, že stejná priorita může být přiřazena více úlohám.

Novou úlohu lze vytvořit voláním funkce `OSTaskCreate()`. Po jejím zavolání je inicializován zásobník a *TCB* úlohy. Nová úloha se vždy nachází ve stavu *Ready* (připravená).

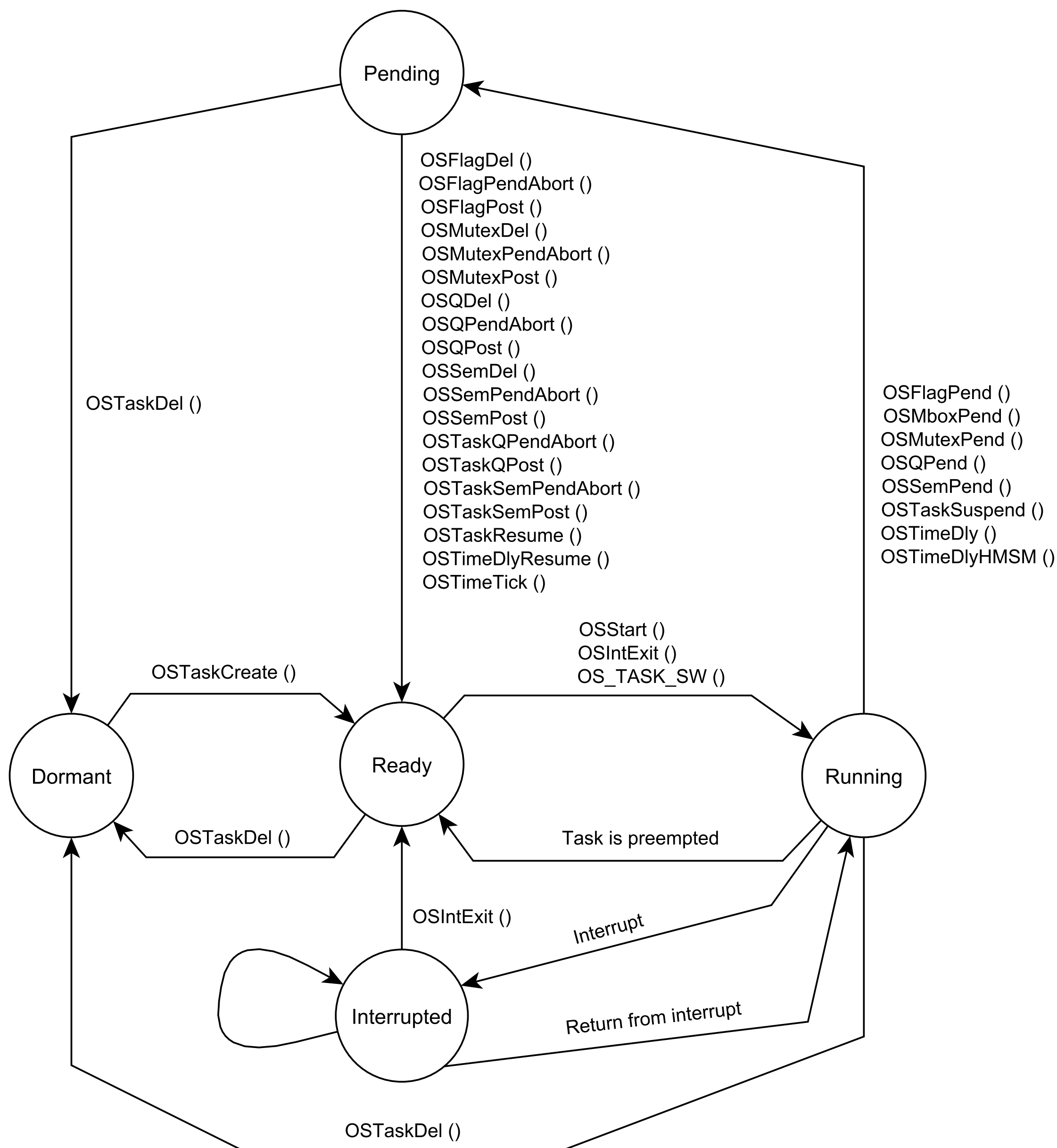
Úloha se nachází ve stavu *Dormant* (odložená), pokud byla odstraněna voláním funkce `OSTaskDel()`. V tomto stavu již úloha není v aplikaci dostupná, nicméně stále je uložena v paměti. Pokud bychom odstraněnou úlohu chtěli v aplikaci znovu využít, musíme ji opětovně vytvořit pomocí funkce `OSTaskCreate()`.

Každá aplikace musí obsahovat právě jedno volání funkce `OSStart()`, která zajistí zahájení *multitaskingu*. Následně plánovač vybere k běhu úlohu s nejvyšší prioritou ze seznamu úloh připravených k běhu (*Ready*).

Pokud je úloha uspána (`OSTaskSuspend()`) nebo čeká na nějakou událost, například na zamknutí semaforu, který je momentálně obsazený jinou úlohou (`OSSemPost()`) či na přijetí zprávy od jiné úlohy, jež ji doposud neodeslala (`OSQPost()`), je úloha převedena do stavu

Pending (čekající) a zařazena do seznamu čekajících úloh. Všechny funkce, jejichž volání vede k zařazení úlohy do seznamu čekajících úloh, jsou znázorněny ve stavovém diagramu na obrázku 3.2.

Při vzniku přerušení přechází právě běžící úloha do stavu *Interrupted* (přerušená). Po jeho obsloužení je spuštěna úloha s nejvyšší prioritou. V *µC/OS-III* může vznik přerušení přerušit jiné přerušení. Na stavovém diagramu (obrázek 3.2) je tato skutečnost znázorněna cyklickým přechodem ve stavu *Interrupted*. Tento děj je nazýván jako vnořování přerušení (*interrupt nesting*), při jehož nesprávném řízení však může velmi snadno dojít k přetečení zásobníku [7]. Stavový diagram úlohy v *µC/OS-III* je zobrazen na obrázku 3.2.



Obrázek 3.2: Stavový diagram úloh v µC/OS-III [7]

Task Control Block

Task Control Block (*TCB*) je datová struktura jádra sloužící k uchování všech informací o každé úloze. Tato struktura je deklarována v souboru `os.h` a je principiálně shodná pro *μC/OS-II* i *μC/OS-III*. Při vytvoření nové úlohy je jí přiřazen její vlastní *TCB*, jenž se ukládá do paměti *RAM*. Při změně kontextu se pracuje právě s *TCB*. Do *TCB* úlohy, která bude pozastavena, jsou uloženy aktuální informace o této úloze a z *TCB* úlohy, která bude následně spuštěna, je obnoven její kontext.

V *TCB* je uchovávána například priorita úlohy, velikost zásobníku, ukazatel na vrchol zásobníku úlohy, stav úlohy apod. V *μC/OS-III* můžeme úlohy pojmenovávat, což usnadňuje ladění programu. Jména úloh jsou pak také samozřejmě uchovávána v *TCB* [6, 7].

3.4 Systémové úlohy

Systémové úlohy jsou úlohy, které jsou přímo součástí daného operačního systému reálného času. Tyto úlohy jsou buď automaticky přidány do každé uživatelské aplikace, nebo je lze volitelně aktivovat.

μC/OS-II

μC/OS-II obsahuje celkem dvě systémové úlohy. Jedna z nich existuje vždy, druhá je volitelná.

Idle úloha

V *μC/OS-II* vždy existuje jedna úloha, která je vykonávána v případě, že k běhu není připravena žádná jiná úloha. Tato úloha se nazývá *idle* úloha (`OS_TaskIdle()`) a její priorita je vždy nastavena na nejnižší hodnotu, čímž je zajištěno, že je opravdu spuštěna pouze v situaci, kdy k běhu není připravena jiná úloha. *Idle* úloha nevykonává žádnou užitečnou funkci a nemůže být nikdy odstraněna pomocí aplikačního softwaru [6].

Statistická úloha

Druhou systémovou úlohou v *μC/OS-II* je statistická úloha (`OS_Task_Stat()`), která poskytuje informace o využití CPU (*Central Processor Unit*). Ve výchozí konfiguraci tato úloha není vytvořena. Je třeba ji povolit nastavením konstanty `OS_TASK_STAT_EN` na hodnotu 1 v souboru `OS_CFG.H`. Pokud je tato úloha povolena, pak každou sekundu zobrazuje statistiku využití procesoru v procentech. Statistická úloha má prioritu o jedničku vyšší, než *idle* úloha [6].

μC/OS-III

μC/OS-III obsahuje pět interních systémových úloh. Dvě z nich jsou vytvořeny vždy při inicializaci a nelze je odstranit. Zbylé tři úlohy jsou volitelné.

Idle úloha

V *μC/OS-III*, stejně jako v *μC/OS-II*, vždy existuje *idle* úloha (`OS_IdleTask()`). Tato úloha má nejnižší prioritu a je spouštěna pouze v případě, kdy k běhu není připravena žádná jiná úloha [7].

***Tick* úloha**

Tato úloha (`OS_TickTask()`) je v systému vytvořena vždy při inicializaci. Je periodická a používá se ke sledování úloh, jež čekají na uvolnění některého z objektů jádra, který má nastaven *timeout*. *Timeout* je parametr objektu jádra definovaný v ticích hodin, po jehož uplynutí je úloze čekající na uvolnění tohoto objektu umožněno pokračovat v běhu. Priorita této úlohy je nastavena na relativně vysokou hodnotu a lze ji uživatelsky měnit pomocí konstanty `OS_CFG_TICK_TASK_PRIO` v konfiguračním souboru `os_cfg_app.h` [7].

***S*tatistická úloha**

Statistická úloha (`OS_StatTask()`) je volitelnou systémovou úlohou, je třeba ji povolit nastavením konstanty `OS_CFG_STAT_TASK_EN` v souboru `os_cfg.h`. Stejně jako v *µC/OS-II* poskytuje statistiku o využití procesoru s tím rozdílem, že rozlišení poskytované hodnoty je v setinách procent oproti jednotkám procent v *µC/OS-II*. Prioritu úlohy lze konfigurovat parametrem `OS_CFG_STAT_TASK_PRIO` v souboru `os_cfg_app.h` [7].

***Timer* úloha**

Timer úloha (`OS_TmrTask()`) je volitelná úloha, je tedy třeba ji aktivovat nastavením konstanty `OS_CFG_TMR_EN` v souboru `os_cfg.h` na hodnotu 1. Priorita této úlohy je typicky nastavena na střední úroveň, lze ji však uživatelsky měnit nastavením konstanty `OS_CFG_TMR_TASK_PRIO` v souboru `os_cfg_app.h` [7].

***ISR Handler* úloha**

Nastavením konstanty `OS_CFG_ISR_POST_DEFERRED_EN` v souboru `os_cfg.h` je vytvořena systémová úloha (`OS_IntQTask()`), která poté odpovídá za provedení služeb volaných z *ISR*. Priorita této úlohy je vždy nastavena na hodnotu 0, což je nejvyšší možná priorita, přičemž žádná jiná úloha už tuto prioritu použít nemůže [7].

3.5 Správa paměti

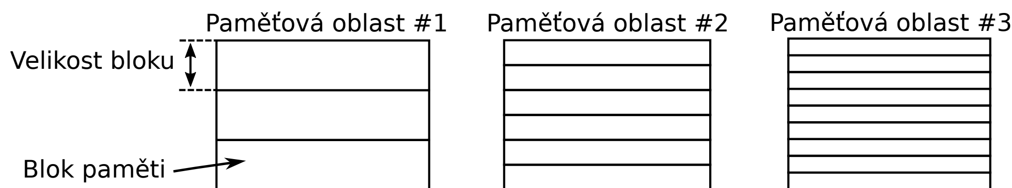
Používání funkcí `malloc()` a `free()` pro alokaci, respektive dealokaci paměti, může být v systémech reálného času nebezpečné z důvodu fragmentace. Rovněž doba trvání zmíněných funkcí je obecně nedeterministická.

Operační systémy reálného času *µC/OS-II* a *µC/OS-III* proto nabízejí alternativu funkcí `malloc()` a `free()` v podobě možnosti alokovat v paměti bloky pevné velikosti. Všechny bloky paměti v jedné paměťové oblasti musí mít stejnou velikost. Lze však vytvořit více paměťových oblastí, tudíž můžeme v aplikaci alokovat bloky různých velikostí. Na obrázku 3.3 je znázorněno několik paměťových oblastí. Operace alokace a uvolnění paměti jsou provedeny v konstantním čase a deterministicky [6, 7].

Funkce pro práci s pamětí

Níže uvedené funkce jsou dostupné v systému *µC/OS-II*.

- `OSMemCreate()` – vytvoření paměťové oblasti.
- `OSMemGet()` – alokace jednoho bloku z paměťové oblasti.



Obrázek 3.3: Paměťové oblasti µC/OS-II a µC/OS-III [6]

- `OSMemPut()` – uvolnění jednoho bloku z paměťové oblasti.
- `OSMemQuery()` – zjištění stavu paměťové oblasti.

V systému *µC/OS-III* již není dostupná funkce `OSMemQuery()`, ostatní funkce jsou zachovány.

3.6 Semafory

Obecně o semaforech pojednává sekce 2.2.4.

µC/OS-II

Semafory v *µC/OS-II* se skládají ze dvou prvků. Z 16-bitové celočíselné bezznaménkové proměnné, která určuje počet volných jednotek a seznamu úloh, které žádají o uzamčení semaforu. Pro práci se semafory je poskytováno celkem šest funkcí, jež jsou popsány v sekci 3.6 [6].

µC/OS-III

Semafory v *µC/OS-III* mají strukturu shodnou se semafory *µC/OS-II*. Rozdílem je, že proměnná určující počet volných jednotek může být 8-bitová, 16-bitová či 32-bitová. Tento parametr je určen datovým typem `OS_SEM_CTR` v souboru `os_type.h`, kde může být nastaven dle potřeby [7].

Funkce pro práci se semafory

Následující funkce jsou dostupné v systému *µC/OS-II*.

- `OSSemAccept()` – zamknutí semaforu (tato funkce je neblokující).
- `OSSemCreate()` – vytvoření semaforu.
- `OSSemDel()` – odstranění semaforu.
- `OSSemPend()` – zamknutí semaforu (tato funkce je blokující).
- `OSSemPost()` – odemknutí semaforu.
- `OSSemQuery()` – zjištění stavu semaforu (vrací aktuální hodnotu semaforu a seznam úloh čekajících na semafor).

V *µC/OS-III* je pro zamknutí semaforu používána jediná funkce `OSSemPend()`, přičemž zda bude funkce blokující či ne lze specifikovat nastavením parametru `opt` této funkce. Druhou změnou je, že v *µC/OS-III* již není dostupná funkce `OSSemQuery()`. Ostatní funkce jsou zachovány. *RTOS µC/OS-III* nabízí oproti *µC/OS-II* ještě několik dalších funkcí pro práci se semafory:

- `OSSemPendAbort()` – zrušení žádosti o zamknutí semaforu.
- `OSSemSet()` – změna aktuální hodnoty semaforu.

3.7 Mutexy

Mutex (*mutual exclusion*) je v systému *µC/OS-II* i *µC/OS-III* speciální typ binárního semaforu, který slouží k řízení výlučného přístupu ke sdílenému zdroji. Pro řízení přístupu ke sdílenému zdroji může být využit i semafor, nicméně se to nedoporučuje, respektive neměl by být používán v případě, že běh úloh musí být dokončen v definovaných časových mezích (*deadlines*), protože může docházet ke vzniku inverze priorit, čímž by mohlo dojít k nedodržení časových mezí úloh. Mutex obsahuje mechanismy zabraňující vzniku inverze priorit [7].

Funkce pro práci s mutexy

Níže uvedené funkce jsou dostupné v systému *µC/OS-II*.

- `OSMutexAccept()` – zamknutí mutexu (tato funkce je neblokující).
- `OSMutexCreate()` – vytvoření mutexu.
- `OSMutexDel()` – odstranění mutexu.
- `OSMutexPend()` – zamknutí mutexu (tato funkce je blokující).
- `OSMutexPost()` – odemknutí mutexu.
- `OSMutexQuery()` – zjištění stavu mutexu (vrací aktuální hodnotu mutexu a seznam úloh čekajících na mutex).

V *µC/OS-III* je pro zamknutí mutexu používána funkce `OSMutexPend()`, přičemž zda bude funkce blokující či ne lze specifikovat nastavením parametru `opt` této funkce. Dalším rozdílem je, že v *µC/OS-III* již není dostupná funkce `OSMutexQuery()`. Ostatní funkce jsou shodné s *µC/OS-II*. *RTOS µC/OS-III* nabízí oproti *µC/OS-II* ještě několik dalších funkcí pro práci s mutexy:

- `OSMutexPendAbort()` – zrušení žádosti o zamknutí mutexu.
- `OSMutexSet()` – změna aktuální hodnoty mutexu.

3.8 Schránky zpráv

Schránky zpráv se nacházejí pouze v systému *µC/OS-II*. V *RTOS µC/OS-III* již nejsou podporovány, jelikož dle tvrzení autorů nejsou třeba. Jsou to objekty, které slouží pro zaslání zprávy mezi dvěmi úlohami či mezi *ISR* a úlohou [6].

Funkce pro práci se schránkami zpráv

- `OSMBoxAccept()` – přijetí zprávy (tato funkce je neblokující).
- `OSMBoxCreate()` – vytvoření schránky zpráv.
- `OSMboxDel()` – odstranění schránky zpráv.
- `OMboxPend()` – přijetí zprávy (tato funkce je blokující).
- `OSMboxPost()` – odeslání zprávy.
- `OSMboxPostOpt()` – odeslání zprávy (umožňuje zaslat zprávu více úlohám současně).
- `OSMboxQuery()` – zjištění stavu schránky zpráv (vrací aktuální obsah schránky zpráv a seznam úloh čekajících na přijetí zprávy).

3.9 Fronty zpráv

Fronty zpráv jsou objekty, které slouží pro zaslání zprávy mezi dvěmi úlohami či mezi *ISR* a úlohou. Rozdílem oproti schránkám zpráv je, že fronta zpráv může obsahovat v jednu chvíli více zpráv, zatímco schránka zpráv pouze jednu [6].

Funkce pro práci s frontami zpráv

Následující funkce jsou dostupné v systému *µC/OS-II*.

- `OSQAccept()` – přijetí zprávy (tato funkce je neblokující).
- `OSQCreate()` – vytvoření schránky zpráv.
- `OSQDel()` – odstranění schránky zpráv.
- `OSQFlush()` – vyprázdnění fronty zpráv.
- `OSQPend()` – přijetí zprávy (tato funkce je blokující).
- `OSQPost()` – odeslání zprávy (zpráva je odeslána na konec fronty).
- `OSQPostFront()` – odeslání zprávy (zpáva je odeslána na začátek fronty).
- `OSQPostOpt()` – odeslání zprávy (umožňuje zaslat zprávu více úlohám současně).
- `OSQQuery()` – zjištění stavu schránky zpráv (vrací aktuální obsah schránky zpráv a seznam úloh čekajících na přijetí zprávy).

V systému *µC/OS-III* byly funkce pro odeslání zprávy (`OSQPost()`, `OSQPostFront()`, `OSQPostOpt()`) sloučeny do jediné funkce `OSQPost()`, přičemž funkcionality odstraněných funkcí `OSQPostFront()` a `OSQPostOpt()` lze dosáhnout nastavením příslušných parametrů funkce `OSQPost()`. Taktéž pro přijetí zprávy je definována pouze jedna funkce (`OSQPend()`). Pro určení, zda bude blokující či neblokující potom slouží parametr `opt` funkce `OSQPend()`. Taktéž byla odstraněna funkce pro zjištění stavu schránky zpráv `OSQQuery()`. Ostatní funkce zůstaly zachovány [7].

Kapitola 4

Přehled metod pro testování výkonnosti RTOS

Tato kapitola obsahuje přehled metod používaných pro testování výkonnosti (*benchmarking*) operačních systémů reálného času. V další části kapitoly jsou pro doplnění přehledu stručně zmíněny také metody a benchmarky pro testování výkonnosti *RT* systémů a HW platforem.

4.1 Rhealstone

Metoda *Rhealstone* [5] je velice používanou metrikou pro testování výkonnosti *RTOS*. Jejím principem je sledování šesti parametrů, které jsou důležité pro *RT* systém. Těmito parametry jsou doba přepnutí úlohy, doba preempce úlohy, doba odezvy přerušení, doba přehození semaforu, doba mimo inverzi priorit a propustnost dat mezi úlohami.

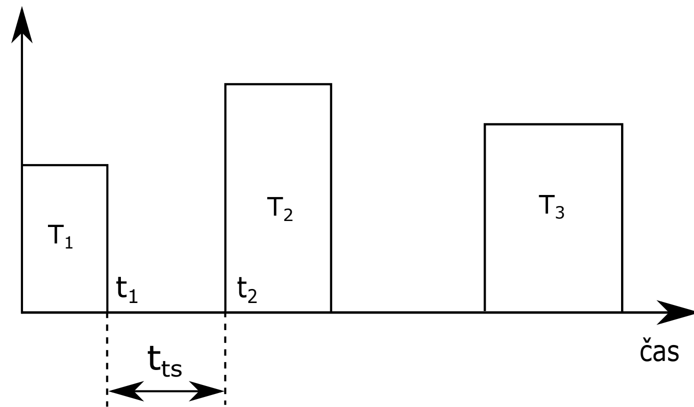
Výsledkem metody *Rhealstone* je jedno číslo, které získáme váhovým součtem změřených parametrů. Každému z výše uvedených šesti parametrů je přiřazena určitá váha, která je zjištěna empiricky. Váha jednotlivých parametrů se tak v každé aplikaci liší. Závisí na tom, jak jsou jednotlivé činnosti zastoupeny v dané aplikaci. Pokud tedy v aplikaci dochází například k častému zasílání zpráv, bude hodnota váhy parametru *Propustnost dat mezi úlohami* vyšší, než v případě nižšího počtu zasílání zpráv. Výsledná hodnota metody *Rhealstone* je tedy dána vzorcem:

$$Rhealstone = a_1 * t_{ts} + a_2 * t_{tp} + a_3 * t_{il} + a_4 * t_{ss} + a_5 * t_{up} + a_6 * t_{dt} \quad (4.1)$$

kde a_1 až a_6 jsou empiricky získané váhy jednotlivých parametrů.

Doba přepnutí úlohy

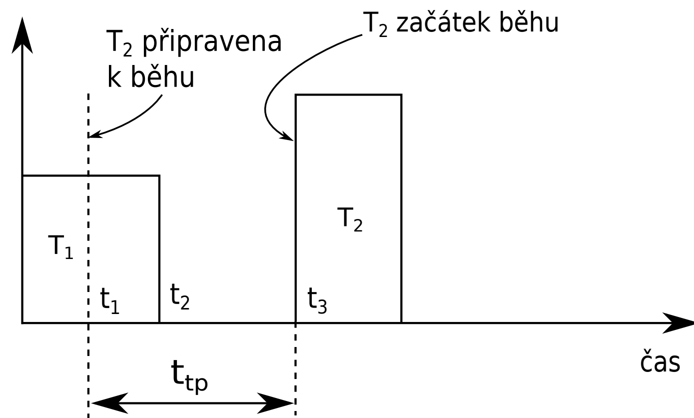
Definována jako doba potřebná k přepnutí kontextu mezi dvěmi úlohami, které mají stejnou prioritu. Podmínkou provedení tohoto testu tedy je, aby daný *RTOS* umožňoval vytvářet více úloh se stejnou prioritou. Jak už tedy víme, tento test je neproveditelný pro *RTOS* *µC/OS-II*. Obrázek 4.1 zobrazuje tři úlohy, T_1, T_2 a T_3, se stejnou prioritou. Doba přepnutí úlohy (t_{ts}) je pak dána jako rozdíl doby, kdy začne běžet úloha T_2 a doby, kdy je dokončen běh úlohy T_1 ($t_{ts} = t_2 - t_1$). Cílem tohoto testu je otestovat efektivitu datové struktury jádra.



Obrázek 4.1: Doba přepnutí úlohy [5]

Doba preempce úlohy

Čas od vzniku pořadavku na spuštění výšeprioritní úlohy, ve vztahu k aktuálně běžící úloze, do doby spuštění výšeprioritní úlohy. Doba preempce úlohy (t_{tp}) je znázorněna na obrázku 4.2. Doba preempce úlohy se skládá ze tří složek, kterými jsou doba přepnutí úlohy (obrázek 4.1), doba potřebná k rozpoznání požadavku běhu výšeprioritní úlohy a doba potřebná k obsloužení tohoto požadavku. Dle výše uvedeného tedy lze předpokládat, že doba preempce úlohy bude vždy vyšší, než doba přepnutí úlohy.



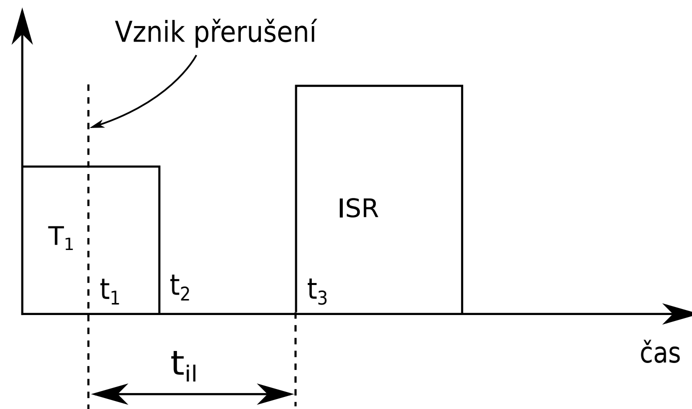
Obrázek 4.2: Doba preempce úlohy [5]

Doba odezvy přerušení

Tento parametr je definován jako doba od vzniku přerušení do doby zahájení obsluhy přerušení. Doba odezvy přerušení (t_{il}) je ilustrována na obrázku 4.3.

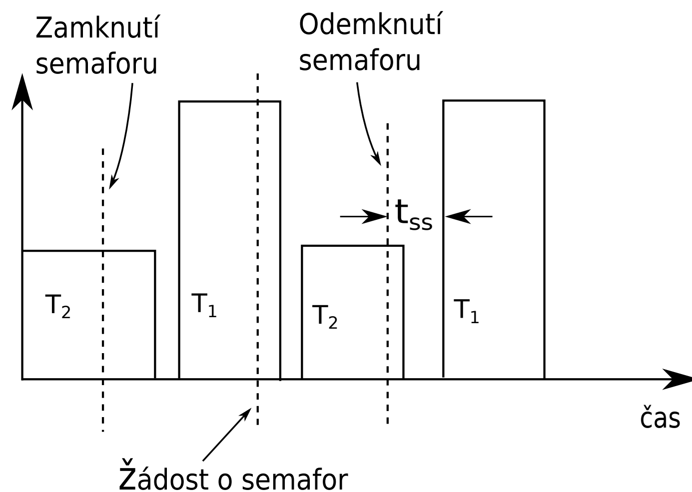
Doba přehození semaforu

Doba od odemknutí semaforu první úlohou do zahájení běhu druhé úlohy, která byla blokována z důvodu čekání na daný semafor. Obě úlohy mají stejnou prioritu. Na obrázku 4.4



Obrázek 4.3: Doba odezvy přerušení [5]

lze vidět, že nejprve běží úloha T_2, která zamkne semafor. Následně dojde k přepnutí kontextu a začne běžet úloha T_1, která se rovněž pokusí zamknout semafor. Ten je nicméně stále v držení úlohy T_2 a úloha T_1 je tedy zablokována, čímž je umožněno pokračování běhu úlohy T_2. Úloha T_2 po nějaké době semafor odemkne, čímž je následně odblokována úloha T_1 a může tak získat semafor.

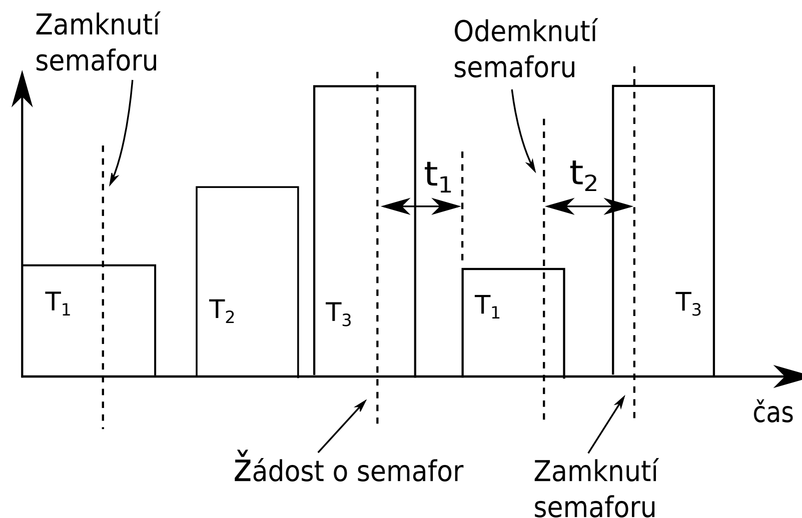


Obrázek 4.4: Doba přehození semaforu [5]

Doba mimo inverzi priorit

Mějme tři úlohy, T_1, T_2 a T_3, přičemž úloha T_1 má nejnižší prioritu, úloha T_2 má vyšší prioritu, než úloha T_1 a zároveň nižší prioritu, než úloha T_3.

Doba mimo inverzi priorit se skládá ze dvou složek, což je znázorněno na obrázku 4.5. Těmito složkami jsou doba od detekce vzniku inverze priorit do znovuspuštění nížeprioritní úlohy (t_1) a doba od uvolnění kritické sekce nížeprioritní úlohou do spuštění úlohy s vyšší prioritou (t_2). Doba mimo inverzi priorit (t_{up}) je pak dána součtem těchto dvou složek ($t_{up} = t_1 + t_2$).



Obrázek 4.5: Doba mimo inverzi priorit [5]

Propustnost dat mezi úlohami

Parametr propustnost dat mezi úlohami (t_{dt}) udává, kolik bytů dat lze zaslat mezi dvěmi úlohami pomocí zasílání zpráv za dobu jedné sekundy. Účelem měření tohoto parametru je ověření efektivity datových struktur mechanismu zasílání zpráv.

4.2 Thread-Metric

Thread-Metric [8] je volně dostupný benchmark pro měření výkonnosti *RTOS*. Zdrojové kódy benchmarku lze zdarma stáhnout z webových stránek[1] firmy Express Logic, Inc., která je tvůrcem tohoto benchmarku. *Thread-Metric* byl původně vytvořen pro měření výkonnosti *RTOS ThreadX*, který je taktéž produktem firmy Express Logic, Inc., nicméně dle tvrzení tvůrce jej lze snadno přizpůsobit i pro jiné *RTOS*. K tomu slouží soubor *tm_porting_layer.c*, ve kterém je třeba implementovat několik funkcí, jež jsou potřebné pro vykonání jednotlivých testů benchmarku. Těmito funkcemi jsou například vytvoření úlohy, vytvoření semaforu, alokace paměti apod.

Principem benchmarku *Thread-Metric* je měření sledovaných parametrů ve stále se opakujících iteracích, výsledky testů jsou pak vypisovány v intervalu každých třicet sekund. Testy jsou prováděny po dvojicích (zamknutí semaforu a odemknutí semaforu, odeslání zprávy a přijetí zprávy apod.). Měřenými parametry jsou:

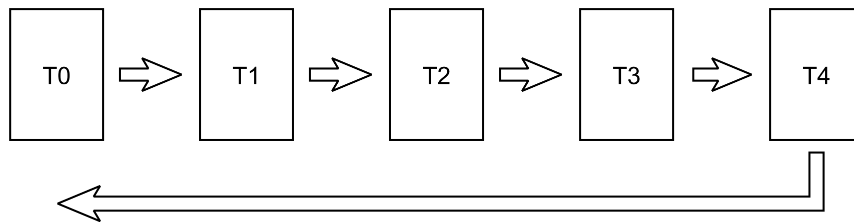
- Kooperativní přepínání kontextu
- Preemptivní přepínání kontextu
- Zpracování přerušení
- Zpracování zpráv
- Zpracování semaforů

[1] http://rtos.com/

- Alokace paměti/dealokace paměti

Kooperativní přepínání kontextu

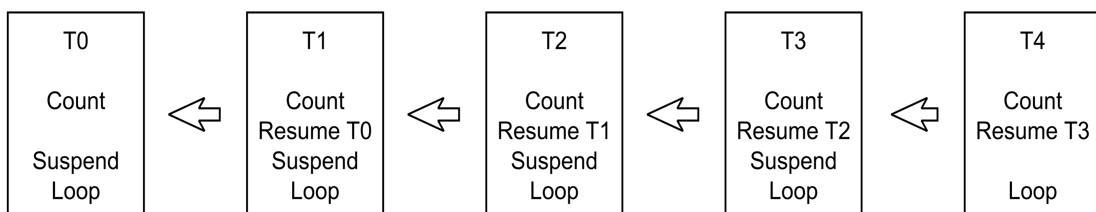
Tento testovací scénář sestává z pěti úloh, které mají stejnou prioritu. Každá úloha vykoná svoji činnost, inkrementuje svůj čítač a předá řízení zpět systému. Plánovač pak vybere k běhu další úlohu, přičemž plánování úloh je prováděno metodou *Round-Robin*. Tento proces se neustále cyklicky opakuje a je schematicky zachycen na obrázku 4.6.



Obrázek 4.6: Kooperativní přepínání kontextu [8]

Preemptivní přepínání kontextu

Test se skládá opět z pěti úloh, jako předchozí test, nyní však mají všechny úlohy různou prioritu. Na začátku testu běží úloha s nejnižší prioritou, ostatní úlohy jsou pozastaveny. Úloha s nejnižší prioritou poté probudí úlohu s druhou nejnižší prioritou a sama je pozastavena. Stejným principem se pokračuje dále, až je provedena úloha s nejvyšší prioritou. Po proběhnutí jednoho cyklu je opět probuzena úloha s nejnižší prioritou a celý postup se cyklicky opakuje. Každá z úloh v rámci svého těla inkrementuje svůj čítač a poté se uspí. Princip tohoto testu ukazuje obázek 4.7.



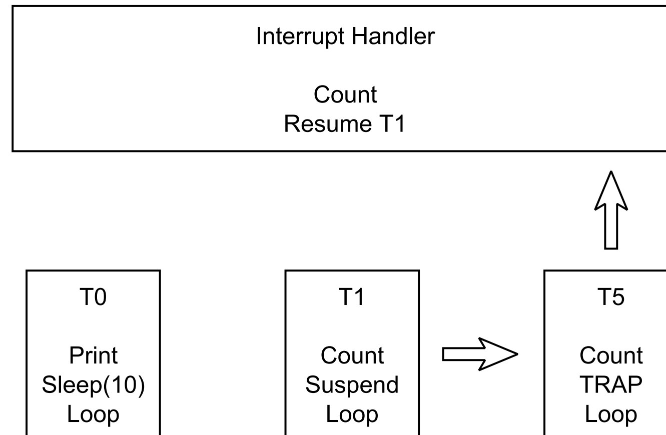
Obrázek 4.7: Preemptivní přepínání kontextu [8]

Zpracování přerušení

Při měření doby zpracování přerušení musíme brát v potaz dvě složky:

- Zpoždění přerušení – Po jakou dobu jsou přerušení zakázána?
- Režii ohledně spuštění úlohy – Jak rychle je úloha schopna odpovědět?

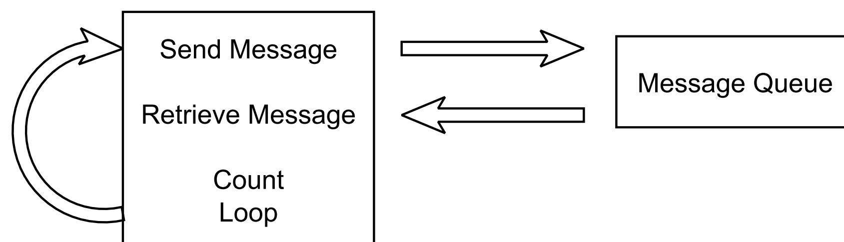
Postup tohoto testu je znázorněn na obrázku 4.8. Na začátku testu běží úloha T_1, která inkrementuje svůj čítač a je pozastavena. Poté je spuštěna úloha T_5, která rovněž inkrementuje čítač, vyvolá přerušení a uspí se. V obsluze přerušení je opět inkrementován čítač a následně probuzena úloha T_1. Úloha T_0 má nejvyšší prioritu a slouží k výpisu výsledků, jenž je prováděn každých 30 sekund.



Obrázek 4.8: Zpracování přerušení [8]

Zpracování zpráv

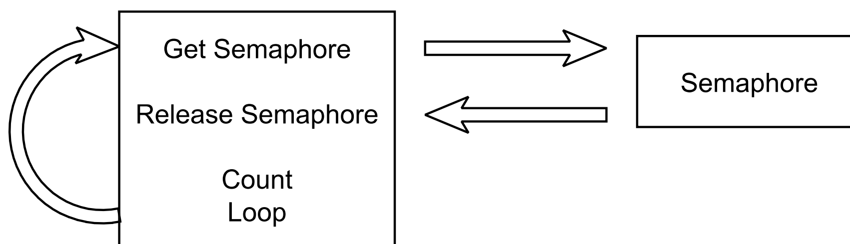
Test zpracování zpráv se skládá z jedné úlohy, která zasílá zprávu o velikosti 16 bytů do fronty zpráv a ta stejná úloha tuto zprávu z fronty zpráv opět přijímá. Po dokončení dvojice operací odeslání/přijetí zprávy úloha inkrementuje čítač. Tento postup se neustále opakuje, což je znázorněno na obrázku 4.9.



Obrázek 4.9: Zpracování zpráv [8]

Zpracování semaforů

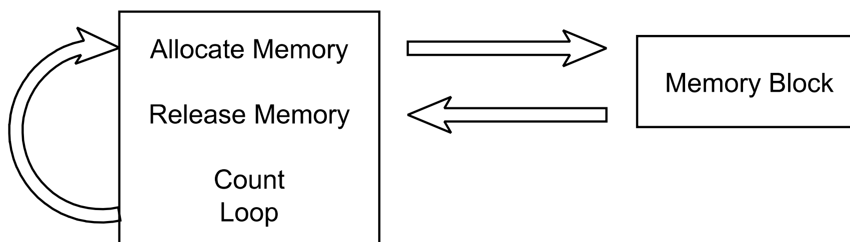
Tento test se stejně jako předchozí skládá z jedné úlohy, která zamkne semafor a neprodleně ho opět odemkne. Po dokončení dvojice operací zamknutí/odemknutí semaforu úloha inkrementuje svůj čítač. Schéma testu je naznačeno na obrázku 4.10.



Obrázek 4.10: Zpracování semaforů [8]

Alokace/dealokace paměti

Princip testu alokace/dealokace paměti je představen na obrázku 4.11. Úloha alokuje jeden blok v paměti o velikosti 128 bytů a po úspěšné alokaci tento blok opět uvolní. Následně inkrementuje svůj čítač a stejný postup se cyklicky opakuje.



Obrázek 4.11: Alokace paměti [8]

4.3 SSC-Benchmark

SSC-Benchmark [9] se zaměřuje na otestování čtrnácti základních služeb jádra. Těmi jsou:

Create/Delete Task

Úloha s nižší prioritou vytvoří výšeprioritní úlohu. Dojde tedy k přepnutí kontextu. Následně se výšeprioritní úloha odstraní a je tedy spuštěna opět nížeprioritní úloha.

Ping Suspend/Resume Task

Úloha s nižší prioritou probudí úlohu s vyšší prioritou, která se následně ihned pozastaví a řízení je vráceno nížeprioritní úloze.

Suspend/Resume Task

Úloha s vyšší prioritou pozastaví úlohu s nižší prioritou a neprodleně ji znovu probudí.

Ping Mutex

Obsahem tohoto testu jsou dvě úlohy, jež střídavě zamykají a odemykají mutex nebo semafor. Tedy první úloha zamkne a ihned odemkne semafor/mutex, druhá úloha následně provede totéž.

Get/Release Mutex

Jedna úloha zamkne mutex/semafor a ihned jej opět odemkne. Měříme dobu, která je k provedení této operace potřebná.

Queue Fill

Test sestává z jedné úlohy, která odešle zprávu do fronty zpráv.

Queue Drain

V tomto testu měříme dobu potřebnou pro přijetí zprávy z fronty zpráv.

Queue Fill/Drain

Jedna úloha odešle zprávu do fronty zpráv a ihned ji sama přijme.

Ping Fill/Drain

Tento test se skládá ze dvou úloh, které mezi sebou komunikují pomocí dvou front zpráv. První úloha odešle zprávu do první fronty zpráv, druhá úloha ji přijme a ihned ji odešle do druhé fronty, odkud ji přijme první úloha.

Allocate/Deallocate Memory

Úloha alokuje paměť a ihned ji opět uvolní.

Time Calls

Získání hodnoty časovače. Lze měřit např. dobu potřebnou pro získání systémového času.

Ping Event

Dvě úlohy se stejnou prioritou čekají na událost generovanou jinou úlohou.

Event Pending

Úloha s vyšší prioritou čeká na událost generovanou nížeprioritní úlohou.

Set Event

V tomto testu se měří doba potřebná k nastavení události.

4.4 Metody pro testování výkonnosti RT systémů a HW platforem

Následující metody a benchmarky jsou používané pro testování *RT* systémů a HW platforem, přičemž nezávisí na použitém *RTOS*. Proto je zde uvedu pouze ve stručnosti pro doplnění přehledu.

4.4.1 Dhrystone

Dhrystone [10] je syntetický benchmark používaný pro testování výkonnosti počítačových systémů, který v roce 1984 vytvořil Dr. Reinhold P. Weicker. *Dhrystone* je zaměřen na výkonnost v celočíselné aritmetice a měří jen několik matematických a základních operací. Příkladem těchto operací jsou sčítání a násobení čísel, práce s řetězci či přístupy do paměti. Výslednou hodnotou měření je počet *Dhrystonů* za sekundu, což vyjadřuje počet provedení měřené operace za jednu sekundu.

4.4.2 Whetstone

Whetstone [11] je syntetický benchmark pro testování výkonnosti počítačových systémů. Jeho prvotními autory jsou Dr. B. A. Wichmann a Dr. H. J. Curnow, kteří v roce 1972 měřili dobu provedení 42 základních výrazů v jazyce ALGOL 60 na padesáti různých počítačích. Mezi měřené operace patřila např. doba přístupu k hodnotě celočíselné proměnné či doba potřebná k uložení reálného čísla do proměnné. Výsledná hodnota měření je pak v tisících *Whetstone* instrukcí za sekundu (kWIPS).

4.4.3 Hartstone

Hartstone [12, 13] je benchmark používaný pro zátěžové (stresové) testování *Hard RT* systémů. Jeho autorem je Nelson Weiderman a název je odvozen z *HArd Real Time* a skutečnosti, že je založen na benchmarku *Whetstone*.

Jsou definovány následující kategorie testů v pořadí od jednodušších ke složitějším:

- **PH Series: Periodic Tasks, Harmonic Frequencies** – Tato sada obsahuje několik úloh, které jsou periodické a harmonické, což znamená, že vykonávání úlohy se v předem daném intervalu neustále opakuje a frekvence každé úlohy je celočíselným násobkem frekvence úlohy s nejnižší frekvencí. Např. sada úloh obsahující čtyři úlohy s frekvencí 10 Hz, 20 Hz, 40 Hz a 200 Hz je harmonická. Důležitost této sady testů je dána tím, že úlohy tohoto typu lze v *RT* aplikacích často nalézt a lze je snadno plánovat.

- **PN Series: Periodic Tasks, Non-Harmonic Frequencies** – Tato sada úloh je podobná sadě *PH Series*, nicméně frekvence úloh je volena dle požadavků aplikace, nikoli požadavků implementace. Jsou hůře plánovatelné, než harmonické úlohy, nicméně odrážejí mnoho reálných situací.

- **AH Series: PH Series with Aperiodic Processing Added** – Sada *PH Series* doplněná o aperiodické úlohy. Reprezentuje aplikaci, ve které systém reaguje na vnější události.

- **SH Series: PH Series with Synchronization** – Synchronizace je používána v řadě *RT* systémů. Sada úloh *SH Series* proto obsahuje úlohy vyžadující synchronizaci. Slouží k otestování efektivnosti synchronizačních mechanismů.
- **SA Series: PH Series with Aperiodic Processing and Synchronization** – Je kombinací všech předcházejících sad úloh. Je tedy nejkomplexnější a slouží k otestování celkové výkonnosti *RT* systému.

4.4.4 SPEC CPU 2006

SPEC CPU 2006 [14] se skládá ze dvou benchmarků. Jeden z nich slouží pro otestování výkonnosti CPU v celočíselné aritmetice, druhý v aritmetice plovoucí řádové čárky. *SPEC CPU 2006* měří výkonnost procesoru několika způsoby. Jedním ze způsobů je měření, jak rychle systém dokončí jednu úlohu – jde tedy o měření rychlosti. Druhým způsobem je měření, kolik úloh je systém schopen dokončit v předem daném časovém intervalu – v tomto případě se jedná o měření propustnosti či kapacity.

Autorem tohoto benchmarku je nezisková organizace SPEC (the Standard Performance Evaluation Corporation), do jejíhož portfolia patří i další benchmarky, např. pro testování výkonnosti souborových serverů (*SPEC SFS2014*), měření grafické výkonnosti (*SPEC viewperf 12*), měření výkonnosti Java serverů (*SPECjbb2015*) a další [15].

4.4.5 CoreMark

CoreMark [16, 17] je benchmark vytvořený pro účely testování výkonnosti procesorového jádra. Jeho autorem je nezisková organizace EEMBC (the Embedded Microprocessor Benchmark Consortium). Zdrojové kódy benchmarku v jazyce ANSI C jsou volně ke stažení z webových stránek výrobce[2]. Obsahem jednotlivých testů benchmarku jsou operace čtení/zápisu, celočíselné operace a řídící operace. Příkladem jsou například různé operace s řetězci či různé algoritmy pracující s poli, nejčastěji maticemi. Řídící operace jsou testovány pomocí konečných automatů. *CoreMark* lze tedy využít pro otestování efektivnosti zřetězeného zpracování instrukcí (*pipelining*), přístupu do paměti a zpracování celočíselných operací.

4.4.6 Mibench

Mibench [18] je benchmark pro testování výkonnosti vestavěných systémů. Skládá se z 35 testovacích aplikací, jež jsou rozděleny do šesti kategorií, jimiž jsou automotive a průmyslové řízení, spotřební elektronika, automatizace kancelářských prací, počítačové sítě, bezpečnost a telekomunikace.

4.4.7 Pababench

Papabench [19] je benchmark založený na volně dostupném projektu *Paparazzi UAV* (*Unmanned Aerial Vehicle*), jehož cílem bylo vytvořit *RT* systém pro řízení dronů. Benchmark byl vytvořen pro experimentální analýzu nejhorších případů dob trvání programu (*WCET*, *Worst Case Execution Time*) a může být rovněž použit pro analýzu plánování podle různých plánovacích algoritmů.

[2] http://www.eembc.org/coremark/index.php

Kapitola 5

Realizační prostředky

Tato kapitola obsahuje popis použitých realizačních prostředků a to jak hardwarových, tak softwarových. Je popsána použitá vývojová deska, mikrokontrolér, kterým je tato vývojová deska osazena a použité vývojová prostředí. Rovněž je uveden stručný popis programovacího modelu procesoru ARM Cortex-M4, jehož znalost je důležitá pro vývoj aplikací na těchto procesorech.

5.1 Vývojová deska FITkit 3

Platforma FITkit 3, neboli Minerva, je vývojová deska vytvořená na Fakultě informačních technologií (FIT) Vysokého učení technického (VUT) v Brně. Je osazena mikrokontrolérem MK60DN512VM10. Součástí vývojové desky je rovněž *FPGA* (*Field Programmable Gate Array*) obvod z rodiny Spartan-6 od firmy Xilinx. Deska dále obsahuje různé periferie, např. *USB* (*Universal Serial Bus*) konektor, *HDMI* (*High-Definition Multimedia Interface*), čtyři LED diody, pět tlačítek, Ethernet port. Vývojová deska FITkit 3 je zachycena na obrázku 5.1.

5.1.1 Mikrokontrolér MK60DN512VM10

Mikrokontrolér MK60DN512VMD10 patří do rodiny mikrokontrolérů K60 firmy Freescale Semiconductor. Je založen na jádru ARM Cortex-M4.

Jeho pracovní frekvence je až 100 MHz, napájecí napětí se pohybuje v rozmezí 1,71 až 3,6 V.

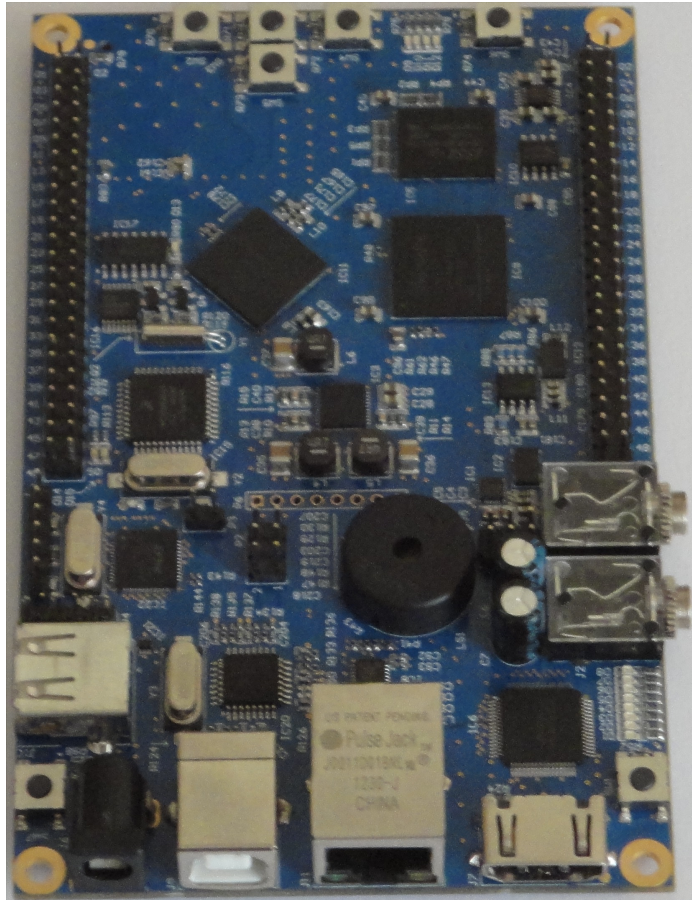
Mikrokontrolér MK60DN512VMD10 obsahuje několik paměťových modulů. Programová FLASH paměť má velikost 512 kB, kapacita *RAM* paměti je 128 kB.

Jako zdroj hodinového signálu může být využit krystalový oscilátor s frekvencí 32 kHZ či 3 až 32 MHz.

Mezi periferie mikrokontroléru patří 16-kanálový řadič *DMA* (*Direct Memory Access*), dva 16-bitové A/D převodníky, dva 12-bitové D/A převodníky, několik časovačů včetně časovače *PWM* (*Pulse Width Modulation*) pro řízení motorů, tři analogové komparátory.

Komunikačími rozhraními jsou Ethernet, USB 2.0 OTG, tři *SPI* (*Serial Peripheral Interface*) moduly, dva *CAN* (*Controller Area Network*) moduly, dvě *I2C* (*Inter-Integrated Circuit*) rozhraní, šest *UART* (*Universal Asynchronous Receiver/Transmitter*) modulů, *SDHC* (*Secure Digital Host Controller*), *I2S* (*Inter-Ic Sound*) modul.

Mikrokontrolér nabízí několik modulů pro bezpečnost a kontrolu integrity:



Obrázek 5.1: Vývojová deska FITkit 3

- Hardwarový *CRC* (*Cyclic Redundancy Check*) modul.
- Hardwarový generátor náhodných čísel.
- Hardwarové šifrování podporující algoritmy *DES*, *3DES*, *AES*, *MD5*, *SHA-1* a *SHA-256*.

Podrobnější informace o tomto mikrokontroléru jsou uvedeny v [20], odkud jsem čerpal.

5.1.2 Programovací model

Tato sekce obsahuje popis programovacího modelu procesoru ARM Cortex-M4, jehož znalost je důležitá pro vývoj aplikací.

Datové typy

Procesor ARM Cortex-M4 podporuje následující datové typy v paměti [21]:

- **Byte** – 8 bitů
- **Halfword** – 16 bitů
- **Word** – 32 bitů

Registry procesoru ARM Cortex-M4 jsou 32-bitové. Instrukční sada tohoto procesoru podporuje následující datové typy uložené v registrech [21]:

- 32-bitové ukazatele
- bezznaménková nebo znaménková 32-bitová celá čísla
- bezznaménková 16-bitová nebo 8-bitová celá čísla, uložená s rozšířením nuly
- znaménková 16-bitová nebo 8-bitová celá čísla, uložená s rozšířením znaménka
- bezznaménková nebo znaménková 64-bitová celá čísla, uložená ve dvou registrech

Módy a stavy procesoru

Procesory ARM Cortex-M4 podporují dva operační módy, *Thread* mód a *Handler* mód. *Thread* mód slouží pro vykonávání aplikačního kódu a procesor do něj vstoupí vždy při resetu či při návratu z obsluhy výjimky. V tomto módu může být kód spouštěn jako privilegovaný či neprivilegovaný. V *Handler* módu se procesor nachází při obsluze výjimky. Kód v tomto módu musí být spouštěn jako privilegovaný. Kód spuštěný jako neprivilegovaný má omezený přístup k některým zdrojům zatímco privilegovaný může využívat všech dostupných zdrojů [22].

Procesor může pracovat ve dvou stavech, *Thumb* stavu a *Debug* stavu. Ve stavu *Thumb* procesor vykonává instrukce. Do stavu *Debug* je procesor převeden debuggerem nebo při dosažení breakpointu. V tomto stavu procesor nevykonává instrukce [22].

Instrukční sada

Procesor ARM Cortex-M4 používá instrukční sadu nazvanou *Thumb-2*. Tato sada obsahuje instrukce předchozí 16-bitové instrukční sady *Thumb*, dále mnoho dalších 16-bitových i 32-bitových instrukcí. Sada *Thumb-2* je vysoce efektivní a přináší významné benefity, pokud jde o velikost kódu či výkonnost [21].

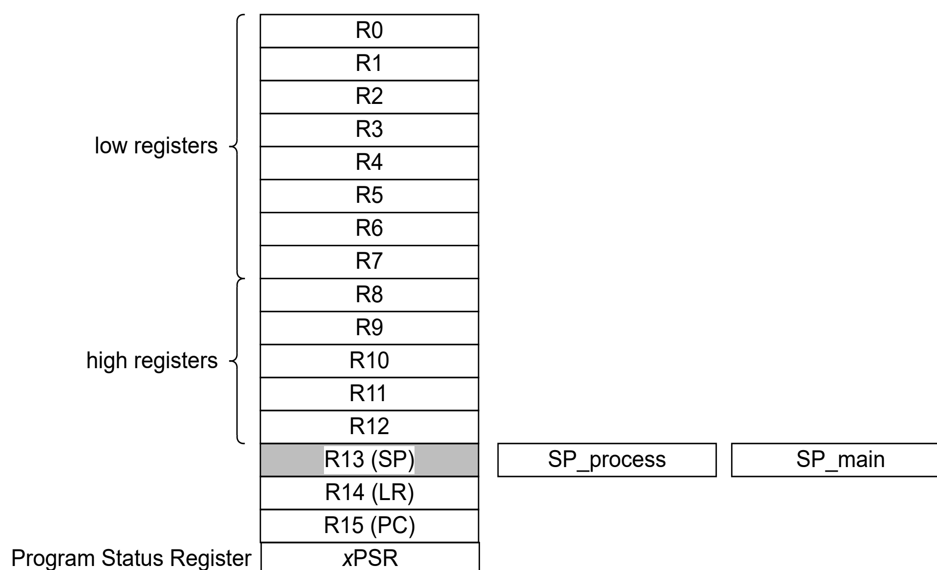
V tradičních procesorech ARM je použita 16-bitová instrukční sada *Thumb* a 32-bitová instrukční sada *ARM*. Pokud je vykonávána instrukce ze sady *Thumb*, nachází se procesor ve stavu *Thumb*. Při vykonávání instrukce ze sady *ARM* musí být procesor přepnut do stavu *ARM*. Efektivita instrukční sady *Thumb-2* tedy spočívá právě v tom, že nedochází k přepínání stavu procesoru při vykonávání různých instrukcí, všechny instrukce jsou vykonávány ve stavu *Thumb* [21].

Registry

Procesor ARM Cortex-M4 obsahuje celkem sedmnáct 32-bitových registrů [22]:

- 13 registrů pro obecné použití
- ukazatel zásobníku
- registr návratové adresy
- progamový čítač
- registr stavu programu

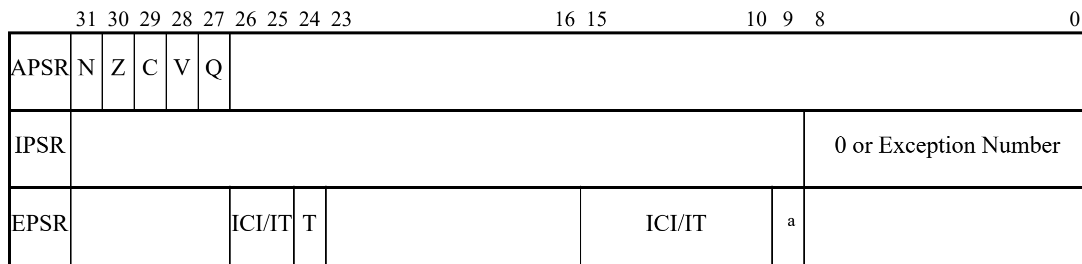
Schéma sady registrů je znázorněno na obrázku 5.2.



Obrázek 5.2: Sada registrů procesoru ARM Cortex-M4 [22]

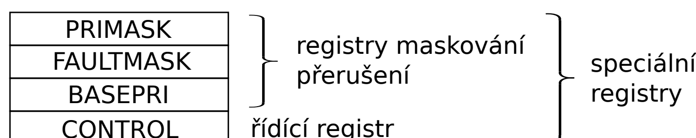
Popis jednotlivých registrů procesoru ARM Cortex-M4 vyobrazených na obrázku 5.2 [22]:

- **Low registers** – obecné registry R0-R7 přístupné všem instrukcím.
- **High registers** – obecné registry R8-R12 přístupné všem 32-bitovým instrukcím.
- **Ukazatel zásobníku** – registr R13 je používaný jako ukazatel zásobníku (*SP*, *Stack Pointer*). *Handler* mód vždy používá *SP_main* (*MSP*, *Main Stack Pointer*), *Thumb* mód lze nakonfigurovat pro použití *SP_main* či *SP_process* (*PSP*, *Process Stack Pointer*).
- **Registr návratové adresy** – registr návratové adresy R14 (*LR*, *Link Register*) slouží k uchování návratové adresy z podprogramu. Je používán také při návratu z obsluhy výjimky.
- **Programový čítač** – registr R15 je programový čítač (*PC*, *Program Counter*).
- **Registr stavu programu** – registr stavu programu (*xPSR*, *Program Status Register*) slouží k uchování stavu programu. Skládá se ze tří podregistrů, *APSR* (*Application Program Status Register*), *IPSR* (*Interrupt Program Status Register*) a *EPSR* (*Execution Program Status Register*). Jeho schéma zobrazuje obrázek 5.3. Bity *N*, *Z*, *C*, *V*, *Q* jsou flagy procesoru. Tedy *Negative*, *Zero*, *Carry/borrow*, *Overflow*, *Sticky saturation*. *ICI/IT* (*Interruptible-Continuable Instruction/If-Then*) slouží pro uložení stavu *If-Then* (*IT*) instrukce nebo pro uložení stavu *exception-continuable* instrukce. Bit *T* (*Thumb* stav) je vždy nastaven na hodnotu 1. Pokus o vynulování tohoto bitu skončí výjimkou. *Exception number* indikuje, kterou výjimku právě procesor obsluhuje, případně obsahuje hodnotu 0, pokud se procesor nachází v *Thread* módu. Bit *a* je rezervován.



Obrázek 5.3: Schéma registru xPSR [21]

Kromě výše uvedených 32-bitových registrů obsahuje ARM Cortex-M4 ještě několik speciálních registrů. Těmi jsou tři registry maskování přerušení (*PRIMASK*, *FAULTMASK*, *BASEPRI*) a řídící registr (*CONTROL*). Jejich schéma zobrazuje obrázek 5.4.



Obrázek 5.4: Speciální registry procesoru ARM Cortex-M4 [21]

Registr *PRIMASK* je jednobitový a slouží k zakázaní všech přerušení, kromě nemaskovatelného přerušení (*NMI, nonmaskable interrupt*) a výjimky*HardFault* [21].

Registr *FAULTMASK* je rovněž jednobitový a jeho funkce je shodná s registrem *PRIMASK* s tím rozdílem, že zakáže i výjimku *HardFault* [21].

Registr *BASEPRI* je také jednobitový a slouží k zakázání všech přerušení s prioritou stejnou a nižší, jako je hodnota uložená v tomto registru. Pokud je tedy obsahem registru hodnota 2, zakáže všechna přerušení s prioritou 2 a nižší. V případě, že je v registru *BASEPRI* uložena hodnota 0, nemá tento registr na přerušení žádný vliv [21].

Registr *CONTROL* je dvoubitový. První bit (*CONTROL[0]*) slouží k nastavení privilegovaného spouštění kódu v *Thread* módu. Druhý bit (*CONTROL[1]*) určuje použitý *SP* (hodnota 0 pro *MSP*, hodnota 1 pro *PSP*) [21].

Výjimky a přerušení

Výjimka může být způsobena vykonáváním instrukce generující výjimky nebo jako odezva na chování systému, jako je například přerušení či ladicí událost [21].

Procesor ARM Cortex-M4 obsahuje řadič přerušení *NVIC* (*Nested Vectored Interrupt Controller*), který se stará o zpracování všech výjimek. Při zpracování výjimek platí následující [22]:

- Všechny výjimky jsou zpracovávány v *Handler* módu procesoru.
- Při vyvolání každé výjimky je automaticky uložen stav procesoru na zásobník a automaticky opět obnoven při ukončení obsluhy výjimky.
- Vektor přerušení je vyčten paralelně s ukládáním stavu procesoru, čímž je zefektivněno zahájení obsluhy přerušení.

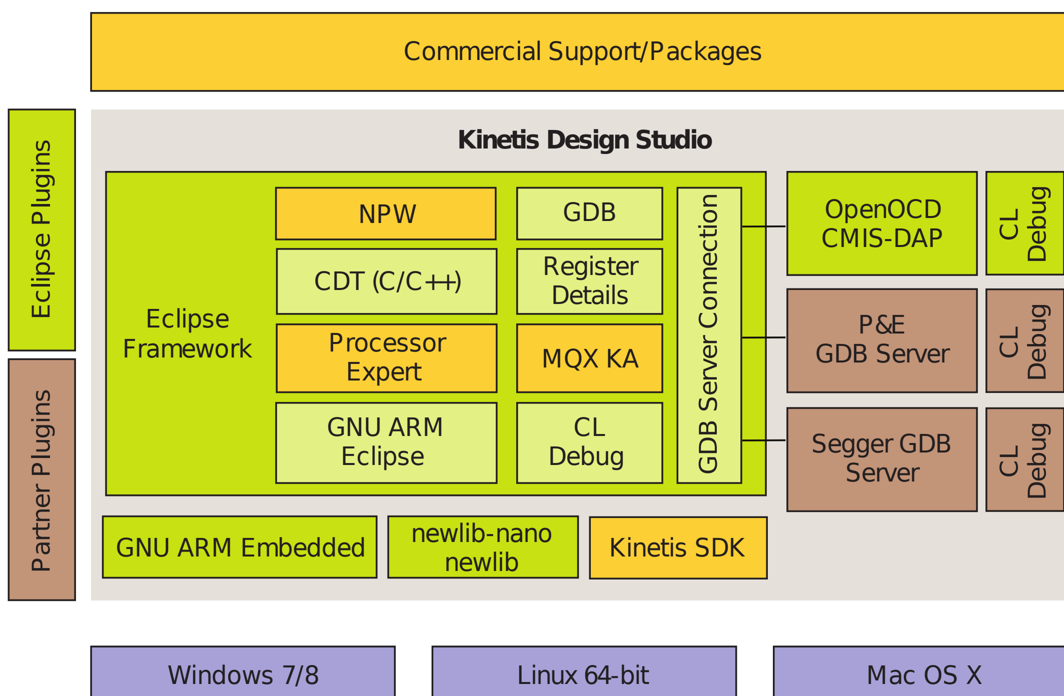
Procesor ARM Cortex-M4 implementuje pokročilý model zpracování výjimek a přerušení. Podrobné informace lze nalézt v [21].

5.2 Kinetis Design Studio

Kinetis Design Studio (*KDS*) je integrované vývojové prostředí od firmy Freescale pro vývoj aplikací pro mikrokontroléry Kinetis. Umožňuje editovat, kompilovat a debuggovat vytvářené aplikace. Je založeno na *Eclipse*, obsahuje sadu překladačů *GCC* (*GNU Compiler Collection*) a debugger *GDB* (*GNU Debugger*) [23].

Vývojové prostředí *KDS* obsahuje integrovaný nástroj *Processor Expert* (*PE*), což je vývojový nástroj pro vytváření, konfiguraci a optimalizaci softwarových komponent, které generují zdrojový kód. *Processor Expert* tak umožňuje vytvářet aplikace ve velmi krátkém čase [23].

Blokové schéma vývojového prostředí *KDS* je zobrazeno na obrázku 5.5.



Obrázek 5.5: Blokové schéma vývojového prostředí KDS [23]

Do tohoto vývojového prostředí lze rovněž integrovat *Kinetis Software Development Kit* (*KSDK*), který obsahuje podpůrné nástroje pro vývoj aplikací na mikrokontrolérech Kinetis. Obsahem tohoto podpůrného balíčku jsou například různé ovladače periferií či porty operačních systémů reálného času [24].

Kapitola 6

Návrh řešení a popis implementace

Obsahem této kapitoly je popis volby implementovaných metod pro otestování výkonnosti jader operačních systémů reálného času *µC/OS-II* a µC/OS-III. Dále je uveden způsob měření, popsáno využití realizačních prostředků, schéma testování a samotná implementace zvolených metod.

6.1 Metody a experimenty

Implementované metody vychází z metod a benchmarků popsaných v kapitole 4. Konkrétně byla zvolena metoda *Thread-Metric* popsaná v sekci 4.2, metoda *Rhealstone*, jejíž bližší popis je uveden v sekci 4.1 a metoda *SSC-benchmark* popsaná v sekci 4.3.

Tyto metody byly vybrány z důvodu, že jsou všeobecně známé a používané pro srovnávání výkonnosti operačních systémů reálného času. Hlavní motivace pro jejich výběr však spočívala v tom, že testují výkonnost *RTOS* z pohledu základních služeb jádra, jako jsou semafory, fronty zpráv apod. Tyto služby jsou používány snad v každé *RT* aplikaci, tudíž jejich výkonnost může hrát důležitou roli při výběru operačního systému reálného času.

Dále bylo implementováno několik dalších testů, jež nejsou součástí metod *Rhealstone*, *Thread-Metric* či *SSC-benchmark*. Tyto testy byly rovněž zaměřeny na základní služby jádra.

6.2 Způsob sběru experimentálních dat

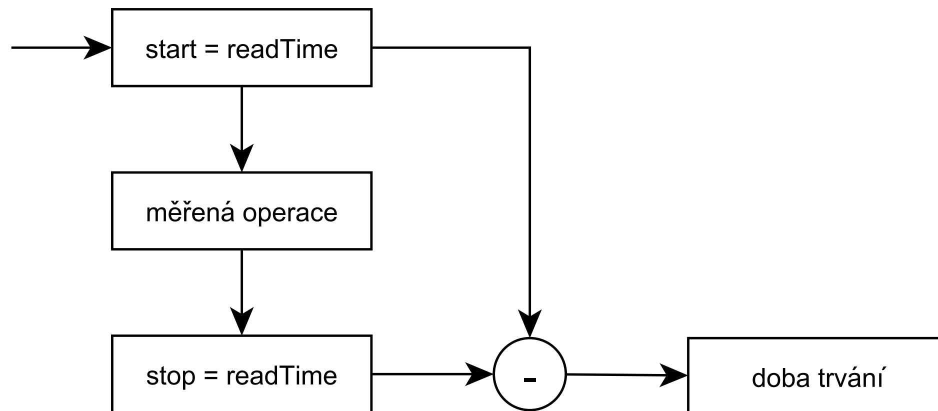
Čítač událostí

Čítač událostí je pro měření výkonnosti používán v metodě *Thread-Metric*. Princip jeho použití je velice jednoduchý. Před začátkem samotného měření je čítač inicializován na hodnotu 0. Po každém jednotlivém provedení měřené operace je pak hodnota čítače zvýšena vždy o hodnotu 1. Jakmile je získán výsledek provedeného testu, čítač je opět vynulován a měření může proběhnout znovu.

Doba trvání

Druhou možností, jak měřit výkonnost prováděných operací, je změřit čas potřebný k provedení dané operace pomocí vestavěného čítače/časovače. Tato metoda je použita u testovací sady *Rhealstone*, *Thread-Metric* a rovněž u vlastních navržených testů. Princip této metody je takový, že odečteme hodnotu čítače/časovače, neprodleně provedeme měřenou

operaci a ihned po jejím dokončení opět odečteme hodnotu čítače/časovače. Odečtením hodnoty čítače/časovače před provedením měřené operace od jeho hodnoty po provedení měřené operace pak získáme dobu trvání měřené operace. Schéma tohoto způsobu měření zachycuje obrázek 6.1. Je třeba však vzít v úvahu rozlišení použitého čítače/časovače.



Obrázek 6.1: Schéma měření doby trvání

6.3 Využití realizačních prostředků

V této sekci je stručně zmíněno, jak byly využity realizační prostředky popsané v kapitole 5 pro implementaci zvolených metod.

6.3.1 Kinetis Design Studio

Testovací metody byly implementovány v jazyce C, tudíž jsem při realizaci využil překladač *GCC* obsažený ve vývojovém prostředí *KDS* a rovněž debugger *GDB*. Netřeba snad ani zmiňovat, že právě debuggeru jsem hojně využíval při ladění implementovaných testů.

Do vývojového prostředí jsem integroval softwarový podpůrný balíček *Kinetis Software Development Kit*, ze kterého jsem využil portů operačních systémů reálného času *μC/OS-II* a *μC/OS-III*.

6.3.2 Periferie mikrokontroléru

Při realizaci testovacích metod bylo využito některých periferií a komunikačních rozhraní zmíněných při popisu mikrokontroléru v sekci 5.1.1.

Časovač FlexTimer

Pro měření doby trvání operací tak, jak bylo popsáno v sekci 6.2 a znázorněno na obrázku 6.1, byl využit čítač/časovač *FlexTimer*, jenž je součástí mikrokontrolérů řady Kinetis K60. *FlexTimer* je 16-bitový čítač/časovač, který může pracovat v několika režimech. V této práci jsem využil režimu *free counter*, tedy volného čítání. V tomto režimu *FlexTimer* čítá od hodnoty 0 do své maximální hodnoty, která činí 65 535.

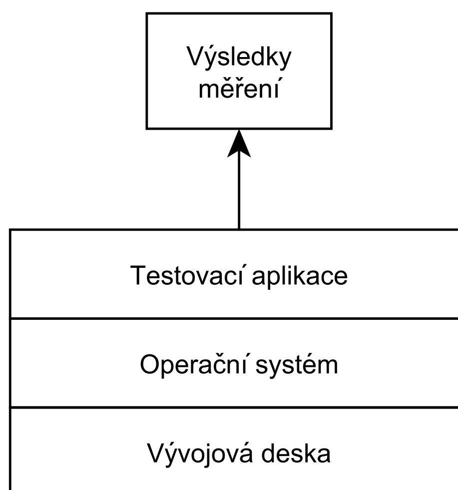
Modul UART

Bylo třeba zvolit způsob, jak budou naměřené výsledky zasílány na výstup. Výstupy testovacích metod jsou zasílány prostřednictvím *UART* rozhraní přes sériovou linku. Následně jsou výsledky zobrazovány v terminálu připojenému k příslušnému sériovému rozhraní.

6.4 Implementace testovacích metod

V této sekci je uveden bližší popis implementace testovacích metod. Pokud nebude řečeno jinak, uváděné ukázky zdrojového kódu platí vždy pro operační systém reálného času *μC/OS-II*.

Obrázek 6.2 ukazuje schéma návrhu testování rozdělené do jednotlivých úrovní. Na nejnižší úrovni je vždy vývojová deska, na které běží příslušný testovaný operační systém reálného času. Nejvyšší vrstva pak obsahuje implementované testovací metody, z nichž jsou po dokončení měření zasílány dosažené výsledky.



Obrázek 6.2: Schéma návrhu testování

6.4.1 Thread-Metric

Implementace této metody vychází z obecného popisu metody uvedeného v sekcii 4.2.

Všechny implementované testovací úlohy metody *Thread-Metric* obsahují úlohu s názvem `main_task()`, jejíž úkolem je vypsat dosažené výsledky v daném testu. Schéma této úlohy je ve všech testovacích úlohách stejné, liší se pouze použitý čítač. Úloha `main_task()` má v rámci aplikace vždy nejvyšší prioritu. Tato úloha je spouštěna každých 30 sekund, vždy vypíše výsledek testu, vynuluje čítač a opět se na 30 sekund uspí. Kód 6.1 znázorňuje tuto úlohu pro testovací úlohu *Zpracování semaforů*.

Kooperativní přepínání kontextu

Tento test se skládá z pěti úloh, `cooperative1()`, `cooperative2()`, `cooperative3()`, `cooperative4()`, `cooperative5()`. Všechny zmíněné úlohy mají stejnou prioritu, což tedy znamená, že tento test nelze provést pro *RTOS μC/OS-II*, jelikož daný *RTOS* neumožňuje

```
unsigned long semaphore_cnt = 0; // čítač událostí

void main_task (os_task_param_t task_init_data) {
  while (1) { // nekonečný cyklus
    OSTimeDlyHMSM(0, 0, 30, 0); // uspání úlohy na 30 sekund
    PRINTF("Pocet iteraci za 30 s: %lu\r\n", semaphore_cnt); // výpis výsledku
    semaphore_cnt = 0; // vynulování čítače
  }
}
```

Kód 6.1: Kód úlohy vypisující výsledky testu

vytvářet více než jednu úlohu na jedné prioritní hladině. Následující popis včetně ukázky zdrojového kódu se tedy týká *RTOS µC/OS-III*.

Pro plánování výše uvedených úloh je využit plánovací algoritmus *Round-Robin* implementovaný v plánovači *RTOS µC/OS-III*. Všechny úlohy jsou připraveny k běhu, plánovač je zařadí do seznamu úloh připravených k běhu. Následně plánovač jednu z nich zvolí a ta je spuštěna. Daná úloha pouze inkrementuje svůj čítač a vzdá se procesoru voláním funkce `OSSchedRoundRobinYield()`. Tím je zařazena na konec seznamu úloh připravených k běhu a plánovač vybere k běhu následující úlohu ze začátku seznamu. Tento cyklus se neustále opakuje.

Na kódu 6.2 je uveden kód jedné z úloh použitých v testu, konkrétně `coopeative1()`. Kód ostatních úloh je v podstatě shodný.

```
void cooperative1 (os_task_param_t task_init_data) {
  INT8U err; // chybový kód

  while (1) { // nekonečný cyklus
    cooperative1_cnt++; // inkrementace čítače
    OSSchedRoundRobinYield(&err); // vzdání se procesoru
  }
}
```

Kód 6.2: Kód úlohy cooperative1()

Preemptivní přepínání kontextu

Test je složen z pěti úloh, `preemptive1()`, `preemptive2()`, `preemptive3()`, `preemptive4()`, `preemptive5()`. Úloha `preemptive1()` má nejnižší prioritu, úloha `preemptive2()` má vyšší prioritu než `preemptive1()` a zároveň nižší než úloha `preemptive3()` atd., až úloha `preemptive5()` má nejvyšší prioritu. Každá z úloh má svůj vlastní čítač událostí.

Nejdříve běží úloha s nejnižší prioritou, tedy `preemptive1()`, která inkrementuje svůj čítač a probudí úlohu `preemptive2()`. Úloha `preemptive2()` probudí úlohu `preemptive3()`, inkrementuje svůj čítač následně se pozastaví. Takto se pokračuje stále dále, až proběhne úloha `preemptive5()`, po které je opět spuštěna úloha `preemptive1()`. Cyklus se tak neustále opakuje. Výsledná hodnota měření je pak dána součtem čítačů událostí všech pěti úloh.

Kód 6.3 ukazuje kód úlohy `preemptive2()`. Zdrojový kód ostatních úloh je schematicky shodný, liší se pouze parametry volaných funkcí.

```
void preemptive2 (os_task_param_t task_init_data) {
  while (1) { // nekonečný cyklus
    OSTaskResume(PREEMPTIVE3_TASK_PRIORITY); // probuzení úlohy preemptive3()
    preemptive2_cnt++; // inkrementace čítače
    OSTaskSuspend(OS_PRIO_SELF); // pozastavení sebe sama
  }
}
```

Kód 6.3: Kód úlohy preemptive2()

Zpracování zpráv

V tomto testu vystupuje jedna úloha, `message_operation()`, která odešle zprávu do fronty zpráv, ihned ji přijme a inkrementuje svůj čítač.

Pro test byla zvolena zpráva o velikosti 4 B (čtyř bajtů). Pro přijetí 16B zprávy z fronty zpráv by bylo nutné volat funkci pro přijetí zprávy z fronty zpráv `OSSemPost()` čtyřikrát. Stejná zpráva však byla využita pro *RTOS µC/OS-II* i *µC/OS-III*, tudíž jsou obě naměřené hodnoty vzájemně porovnatelné.

Zpracování semaforů

Tento test je složen z jedné úlohy s názvem `semaphore_operation()`, která zamkne semafor a neprodleně jej opět odemkne. Po dokončení této dvojice operací inkrementuje svůj čítač událostí.

Alokace/dealokace paměti

Test alokace/dealokace paměti se opět skládá z jedné úlohy, `memory_operation()`. Nejdříve je nutné definovat v paměti prostor, ze kterého následně bude alokována paměť. To se provede pomocí funkce `OSMemCreate()`.

V těle úlohy `memory_operation()` je nejdříve alokován jeden blok v paměti o velikosti 128 B. Následně je tento blok paměti opět uvolněn a je inkrementován čítač.

6.4.2 Rhealstone

Testovací aplikace metody *Rhealstone* byly implementovány dle popisu této metody uvedeného v sekci 4.1. Rovněž bylo využito článku [25], ve kterém autor benchmarku *Rhealstone* popisuje jeho implementaci.

Každý test metody *Rhealstone* obsahuje úlohu `main_task()`, jejíž úkolem je změřit dobu provádění programu bez měřené operace a následně zajistit běh úloh, jež vykonávají měřenou činnost. Kód úlohy `main_task()` pro test propustnosti dat mezi úlohami je zobrazen na kódu 6.4.

Doba přepnutí úlohy

Na počátku měření běží úloha `main_task()`, ve které je změřena doba potřebná k proběhnutí testu bez přepínání kontextu, která bude následně od výsledku odečtena. Po změření této doby úloha `main_task()` sníží svoji prioritu, aby byl umožněn běh úloh `task1()` a `task2()` se shodnou prioritou. Obě úlohy se pouze vzdají procesoru voláním funkce `OSSchedRoundRobinYield()`.

```
void main_task (os_task_param_t task_init_data) {
  INT16U start, startw, end, endw; // začátky, konce měření

  while (1) { // nekonečný cyklus
    /* ZMĚŘENÍ DOBY BEZ POSÍLÁNÍ ZPRÁV */
    startw = FTM_DRV_CounterRead(FSL_FLEXTIMER1); // odečtení počáteční hodnoty
    for (int i = 0; i < CYCLES; i++) {}
    for (int j = 0; j < CYCLES; j++) {}
    endw = FTM_DRV_CounterRead(FSL_FLEXTIMER1); // odečtení koncové hodnoty

    /* ZMĚŘENÍ DOBY S POSÍLÁNÍM ZPRÁV */
    start = FTM_DRV_CounterRead(FSL_FLEXTIMER1); // odečtení počáteční hodnoty
    OSTaskChangePrio(PRIORITY_OSA_TO_RTOS(MAINTASK_TASK_PRIORITY),
      PRIORITY_OSA_TO_RTOS(20U)); // změna priority úlohy, aby mohly bežet testovací
    end = FTM_DRV_CounterRead(FSL_FLEXTIMER1); // odečtení koncové hodnoty
  }
}
```

Kód 6.4: Kód úlohy main_task() pro test propustnosti dat

Doba preempce úlohy

V úloze s nejvyšší prioritou `main_task()` je změřena doba potřebná k běhu aplikace bez preempce a přepnutí kontextu a doba potřebná k probuzení úlohy `task1()`. Následně úloha sníží svou prioritu, čímž dojde ke spuštění úlohy `task1()`. Tato úloha se pouze uspí pomocí funkce `OSTaskSuspend()` a je spuštěna úloha `task2()`, která probudí opět úlohu `task1()`.

Doba přehození semaforu

Test se skládá ze tří úloh. V úloze s nejvyšší prioritou `main_task()` je změřena doba potřebná k proběhnutí testu bez operací zamykání/odemykání semaforu. Následně jsou spuštěny dvě shodné úlohy se stejnou prioritou, `task1()` a `task2()`. První úloha se pokusí zamknout semafor a pokud se jí to podaří, ihned se vzdá procesoru. Tím je spuštněna druhá úloha, která se pokusí rovněž zamknout semafor, který je však v držení pvní úlohy, proto se zablokuje a čeká na uvolnění semaforu. Tím je opět spuštěna první úloha, která semafor uvolní a opět se vzdá procesoru. Následně tedy semafor zamkne druhá úloha, vzdá se procesoru a cyklus se opakuje znovu po stanovený počet iterací.

Doba mimo inverzi priorit

Jako objekt, ke kterému budou úlohy přistupovat, byl zvolen semafor, jelikož mutex obsahuje nástroje pro zabránění vzniku inverze priorit a měření by tak nebylo možné.

Test je složen ze čtyř úloh. Úloha `main_task()`, která má na počátku nejvyšší prioritu, změří čas potřebný k běhu programu bez vzniku inverze priorit. Následně sníží svoji prioritu, čímž je umožněn běh úloh `task1()`, `task2()` a `task3()`. Úloha s nejnižší prioritou `task3()` zamkne semafor a probudí středněprioritní úlohu `task2()`. Tato úloha se pouze pozastaví a probudí úlohu `task1()`, jež má nejvyšší prioritu. Úloha `task1()` se pokusí zamknout semafor, který je však ve vlastnictví úlohy `task3()`. Úloha `task1()` tedy zahájí čekání na semafor, následkem čehož je spuštěna následující úloha s nejvyšší prioritou, kterou je úloha `task3()`. Úloha `task3()` odemkne semafor a následně dojde k preempci, jelikož semafor,

na který čeká úloha `task1`, je již volný a zamkne jej tedy úloha `task1()`.

Propustnost dat mezi úlohami

Jelikož v *µC/OS-III* nejsou k dispozici schránky zpráv, byly k implemetaci tohoto testu využity fronty zpráv.

Test se skládá celkem ze tří úloh. Na počátku má nejvyšší prioritu úloha `main_task()`, ve které je provedena veškerá inicializace, je změřena doba potřebná pro běh aplikace bez zasílání/přijímání zpráv. Následně úloha `main_task()` sníží svou prioritu až za prioritu úloh `task1()` a `task2()`, aby mohly běžet právě tyto úlohy.

Úloha `task1()`, jejíž úkolem je přijmout zprávu z fronty zpráv, má vyšší prioritu, než úloha `task2()`. Tím je zajištěno, že úloha `task1()`, čekající na zprávu z fronty zpráv, bude spuštěna ihned, jakmile úloha `task2()` zprávu odešle.

Po skončení zasílání/přijímání zpráv jsou úlohy `task1()` a `task2()` odstraněny, čímž dojde opět ke spuštění úlohy `main_task()`, která vypíše naměřené výsledky a rovněž se odstraní.

6.4.3 SSC-benchmark

SSC-benchmark byl implementován dle popisu uvedeného v sekci 4.3.

Get/Release Semaphore/Mutex

Test je složen z jedné úlohy, `main_task()`. Nejdříve je odečtena aktuální hodnota čítače, následně úloha zamkne semafor/mutex, ihned jej opět odemkne a znovu je odečtena hodnota čítače. Kód 6.5 ukazuje kód tohoto testu při použití semaforů.

```
void main_task (os_task_param_t task_init_data) {
  INT8U err, err2; // chybové kódy
  INT16U start, end, res; // začátek, konec, výsledek jednoho měření
  INT16U cnt = 0, min_cnt = 0xFFFF, max_cnt = 0; // výsledek všech měření, min, max

  for (int i = 0; i < 100; i++) { // 100 cyklů
    start = FTM_DRV_CounterRead(FSL_FLEXTIMER1); // odečtení počáteční hodnoty
    OSSemPend(semaphore, 0, &err); // zamknutí semaforu
    err2 = OSSemPost(semaphore); // odemknutí semaforu
    end = FTM_DRV_CounterRead(FSL_FLEXTIMER1); // odečtení koncové hodnoty
    res = end - start; // naměřená hodnota
    cnt += res; // pričtení naměřené hodnoty k celkovému výsledku

    if (res > max_cnt) max_cnt = res; // výpočet maximální hodnoty
    if (res < min_cnt) min_cnt = res; // výpočet minimální hodnoty
  }
}
```

Kód 6.5: Kód úlohy main_task() testu Get/Release Semaphore

Create/Delete Task

Úloha `main_task()` s nižší prioritou zahájí měření a vytvoří výšeprioritní úlohu `task1()`, čímž dojde k přepnutí kontextu. Úloha `task1()` se sama odstraní voláním funkce `OSTaskDel()`,

řízení je předáno opět úloze `main_task()` a měření je zastaveno.

Ping Suspend/Resume Task

Úloha `main_task()` s nižší prioritou zahájí měření a probudí výšeprioritní úlohu `task1()`, čímž dojde k přepnutí kontextu. Úloha `task1()` se ihned pozastaví, řízení je předáno opět úloze `main_task()` a měření je zastaveno.

Suspend/Resume Task

Obsahem testu jsou dvě úlohy, výšeprioritní `main_task()` a `task1()`. V úloze `main_task` je pozastavena úloha `task1()` voláním funkce `OSTaskSuspend()`, jenž je ihned znovu probuzena pomocí funkce `OSTaskResume()`. Úloha `task1()` neprovádí žádnou činnost.

Ping Mutex

Úloha `main_task()` zahájí měření, zamkne a neprodleně odemkne mutex a probudí úlohu se stejnou prioritou `task1()`. Úloha `task1()` rovněž zamkne a odemkne mutex. Poté se pozastaví, čímž je opět spuštěna úloha `main_task()`, v níž je zastaveno měření.

Provedení tohoto testu vyžaduje dvě úlohy se stejnou prioritou, proto jej opět nelze provést pro *μC/OS-II*.

Queue Fill, Queue Drain, Queue Fill/Drain

Jde celkem o tři testy, které měří dobu provádění operací týkajících se fronty zpráv. Každý z těchto testů obsahuje jednu úlohu `main_task()`.

V případě testu *Queue Fill* je měřena doba odeslání zprávy do fronty zpráv pomocí funkce `OSQPost()`.

Obsahem *Queue Drain* je změření doby přijetí zprávy z fronty zpráv pomocí funkce `OSQPend()`.

Queue Fill/Drain je pak spojením dvou předcházejících testů. Nejdříve je odeslána zpráva do fronty zpráv, ihned je přijata stejnou úlohou a je změřen čas potřebný k této operaci.

Pro testování byla využita zpráva o velikosti 4 B.

Ping Fill/Drain

Obsahem testu jsou dvě úlohy se shodnou prioritou. Úloha `main_task()` odešle zprávu do první fronty zpráv, ze které ji přijme úloha `task1()`. Úloha `task1()` přijatou zprávu neprodleně odešle do druhé fronty zpráv, ze které ji přijme úloha `main_task()`.

Allocate/Deallocate Memory

Test alokace/dealokace paměti je opět složen z jedné úlohy `main_task()`. V úloze je nejprve alokován jeden blok v paměti o velikosti 128 B pomocí funkce `OSMemGet()`, který je neprodleně opět uvolněn voláním funkce `OSMemPut()`.

Time Calls

Tento test se skládá z jedné úlohy `main_task()`, v níž je změřena doba potřebná k získání hodnoty systémového časovače. Systémový čas lze získat zavoláním funkce `OSTimeGet()`.

6.4.4 Vlastní testy

Bylo implementováno několik dalších testů, které nejsou součástí metod *Thread-Metric*, *Rhealstone* a *SSC-Benchmark*. Zmíněné metody často testují operace po dvojicích, tzn. alokaci a dealokaci paměti, zamknutí a odemknutí semaforu apod. Z výsledků těchto metod tedy není zřejmé, jak dlouho trvá operace zamknutí semaforu, alokace paměti atp. Právě na změření dob trvání těchto operací jsou zaměřeny vlastní navržené testy.

Zamknutí/odemknutí semaforu/mutexu

Test se skládá z jedné úlohy, `main_task()`, ve které je samostatně změřena doba zamknutí a doba odemknutí semaforu i mutexu.

Alokace/dealokace paměti

Tento test je podobný testu *Allocate/Deallocate Memory* z *SSC-benchmarku*, nicméně v úloze `main_task()` je samostané změřena doba alokace a doba dealokace paměti. Opět byl využit blok paměti o velikosti 128 B. Kód měření doby alokace paměti je zobrazen na kódu 6.6.

```
void main_task (os_task_param_t task_init_data) {
  INT8U err, err2; // cgybové kódy
  INT8U *mem_get; // přidělený blok paměti
  INT16U start, end, res; // začátek, konec, výsledek měření
  INT16U min_cnt = 0xFFFF, max_cnt = 0; // minimální, maximální hodnota

  while (1) { // nekonečný cyklus
    start = FTM_DRV_CounterRead(FSL_FLEXTIMER1); // odečtení počáteční hodnoty
    mem_get = OSMemGet(mem, &err); // alokace 1 bloku v paměti
    err2 = OSMemPut(mem, (void *)mem_get); // uvolnění bloku
    end = FTM_DRV_CounterRead(FSL_FLEXTIMER1); // odečtení koncové hodnoty
    res = end - start; // naměřená hodnota

    if (res > max_cnt) max_cnt = res; // výpočet maximální hodnoty
    if (res < min_cnt) min_cnt = res; // výpočet minimální hodnoty
  }
}
```

Kód 6.6: Kód úlohy main_task() pro alokaci paměti

Vytvoření/odstranění úlohy

V úloze `main_task()` je nejprve vytvořena nová úloha `task1()` s nižší prioritou, než má úloha `main_task()`. Je změřena doba potřebná k vytvoření této úlohy a úloha je odstraněna, přičemž se opět změří doba potřebná k odstranění úlohy. Úloha `task1()` není nikdy spuštěna, jelikož má nižší prioritu a po vytvoření je ihned odstraněna.

Probuzení/pozastavení úlohy

Obsahem testu je jedna úloha, `main_task()`, ve které je samostatně změřena doba potřebná k pozastavení úlohy a následně doba potřebná k probuzení úlohy.

Kapitola 7

Vyhodnocení dosažených výsledků

Tato kapitola obsahuje vyhodnocení výsledků, kterých bylo dosaženo měřením výkonnosti operačních systémů reálného času *µC/OS-II* a *µC/OS-III* dle jednotlivých metod. Nejprve jsou v tabulkách prezentovány výsledky měření dle jednotlivých metod samostatně pro *µC/OS-II* a *µC/OS-III*. Dosažené výsledky u obou operačních systémů reálného času jsou následně porovnány. U jednotlivých metod je rovněž popsán způsob měření.

Frekvence mikrokontroléru byla při každém měření nastavena na hodnotu 95,977472 MHz.

7.1 Thread-Metric

V této sekci jsou uvedeny naměřené výsledky dle metody *Thread-Metric*. Nejprve samostatně pro *µC/OS-II* a *µC/OS-III*. Následně jsou dosažené výsledky u obou *RTOS* porovnány.

Každé měření bylo provedeno celkem desetkrát. V tabulkách 7.1 a 7.2 je pak uvedena dosažená minimální, maximální a průměrná hodnota, jenž byla vypočtena jako aritmetický průměr ze všech naměřených hodnot. Dále je uveden rozptyl jednotlivých měření.

7.1.1 µC/OS-II

Tabulka 7.1 obsahuje výsledky měření pomocí *Thread-Metric* testů pro operační systém reálného času *µC/OS-II*.

Měřená operace	**Průměrná hodnota [počet cyklů]**	**Minimální hodnota [počet cyklů]**	**Maximální hodnota [počet cyklů]**	**Rozptyl [počet cyklů]**
Kooperativní přepínání kontextu	-	-	-	-
Preemptivní přepínání kontextu	2 610 771,5	2 610 770	2 610 773	1,25
Zpracování zpráv	5 647 904,3	5 647 904	5 647 905	0,21
Zpracování semaforů	7 724 983,6	7 724 982	7 724 985	0,84
Alokace/dealokace paměti	8 760 806,6	8 760 806	8 760 809	0,86

Tabulka 7.1: Výsledky měření Thread-Metric testů pro µC/OS-II

7.1.2 μC/OS-III

V tabulce 7.2 jsou zobrazeny výsledky měření *Thread-Metric* testů pro *RTOS* μ*C/OS-III*.

Měřená operace	**Průměrná hodnota [počet cyklů]**	**Minimální hodnota [počet cyklů]**	**Maximální hodnota [počet cyklů]**	**Rozptyl [počet cyklů]**
Kooperativní přepínání kontextu	3 449 003,3	3 449 002	3 449 004	0,41
Preemptivní přepínání kontextu	1 826 088,7	1 826 088	1 826 090	0,61
Zpracování zpráv	3 763 863,3	3 763 863	3 763 864	0,21
Zpracování semaforů	6 499 714,1	6 499 706	6 499 726	94,99
Alokace/dealokace paměti	9 931 733,2	9 931 733	9 931 734	0,19

Tabulka 7.2: Výsledky měření Thread-Metric testů pro µC/OS-III

7.1.3 Srovnání μC/OS-II a μC/OS-III

Tabulka 7.3 ukazuje srovnání výsledků měření pomocí *Thread-Metric* testů pro μ*C/OS-II* a μ*C/OS-III*. Toto srovnání je rovněž přehledně vyobrazeno na obrázku 7.1.

Operační systém reálného času μ*C/OS-II* dosáhl lepších výsledků téměř ve všech testech, kromě zpracování paměti. Právě v testu zpracování paměti dosáhly oba systémy na nejvyšší naměřené hodnoty. Naopak nejdelší dobu v testech *Thread-Metric* potřebovaly oba systémy k preemptivnímu přepnutí kontextu.

Nejvyššího rozdílu bylo dosaženo u testu zpracování zpráv, kdy byly tyto operace v μ*C/OS-II* o 50 procent rychlejší, než v μ*C/OS-III*.

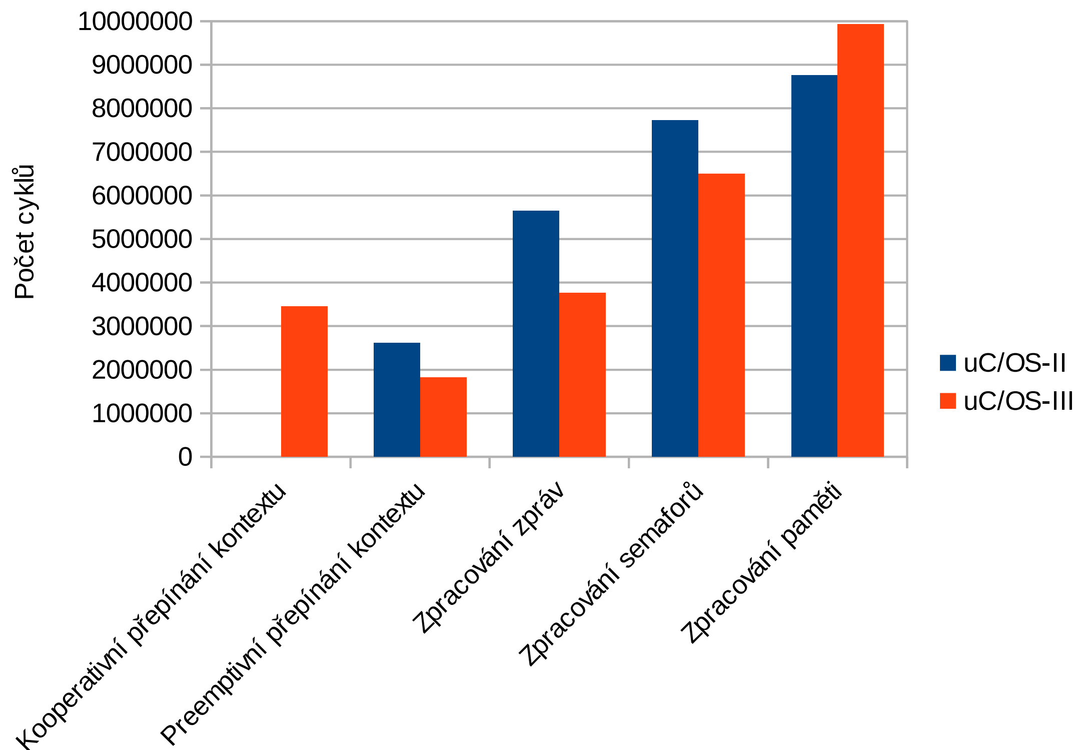
Naopak nejnižší rozdíl v případě, kdy byl μ*C/OS-II* rychlejší, byl pozorován u testu preemptivního přepínání kontextu a to necelých 19 procent.

V případě zpracování paměti dosáhl o 13 procent lepších výsledků *RTOS* μ*C/OS-III*.

Je třeba si však uvědomit metodiku měření *Thread-Metric*. Každý jednotlivý test běží 30 sekund, do výsledku je tak započítána i doba plánování úloh. Při interpretaci výsledků je tak nutné s tímto počítat. Nejedná se však o chybu měření, nýbrž o princip metody *Thread-Metric*. Možné příčiny dosažených výsledků jsou diskutovány v sekci 7.5.

Měřená operace	**µC/OS-II**	**µC/OS-III**
	Průměrná hodnota [počet cyklů]	**Průměrná hodnota [počet cyklů]**
Kooperativní přepínání kontextu	-	3 449 003,3
Preemptivní přepínání kontextu	2 610 771,5	1 826 088,7
Zpracování zpráv	5 647 903,3	3 763 863,3
Zpracování semaforů	7 724 983,6	6 499 714,1
Alokace/dealokace paměti	8 760 806,6	9 931 733,2

Tabulka 7.3: Porovnání µC/OS-II a µC/OS-III pomocí Thread-Metric testů



Obrázek 7.1: Porovnání µC/OS-II a µC/OS-III pomocí Thread-Metric testů

7.2 SSC-benchmark

Obsahem této sekce jsou naměřené výsledky pomocí metody *SSC-benchmark*. Nejprve samostatně pro *µC/OS-II* a *µC/OS-III*. Následně jsou dosažené výsledky u obou *RTOS* porovnány.

Frekvence použitého čítače/časovače *FlexTimer* byla nastavena na hodnotu 47,989 MHz, čímž bylo dosaženo přesnosti měření 20,838 ns.

Každé z jednotlivých měření bylo provedeno celkem stokrát pro eliminaci chyby způsobené rozlišením časovače *FlexTimer*. V tabulkách 7.4 a 7.5 je pak uvedena dosažená minimální, maximální a průměrná hodnota, jenž byla vypočtena jako aritmetický průměr ze všech naměřených hodnot. Dále je uvedena dosažená průměrná hodnota v mikrosekundách, která je však jen orientační, jelikož je zaokrouhlena pouze na dvě desetinná místa.

7.2.1 µC/OS-II

Tabulka 7.4 obsahuje naměřené výsledky pro operační systém reálného času *µC/OS-II* pomocí testů metody *SSC-benchmark*.

7.2.2 µC/OS-III

V tabulce 7.5 jsou obsaženy naměřené výsledky pomocí testů metody *SSC-benchmark* pro operační systém reálného času *µC/OS-III*.

Měřená operace	**Průměrná hodnota [počet tiků]**	**Minimální hodnota [počet tiků]**	**Maximální hodnota [počet tiků]**	**Průměrná hodnota [µs]**
Create/Delete Task	1 558	1 558	1 558	32,47
Ping Suspend/Resume Task	718	718	718	14,96
Suspend/Resume Task	418	418	418	8,71
Ping Mutex	-	-	-	-
Get/Release Semaphore	228	228	228	4,75
Get/Release Mutex	291,02	290	292	6,06
Queue Fill	155,04	154	156	3,23
Queue Drain	179	179	179	3,73
Queue Fill/Drain	280,81	280	284	5,85
Ping Fill/Drain	-	-	-	-
Allocate/Deallocate Memory	207	207	207	4,31
Time Calls	98,01	98	99	2,04

Tabulka 7.4: Výsledky měření SSC-benchmark testů pro µC/OS-II

Měřená operace	**Průměrná hodnota [počet tiků]**	**Minimální hodnota [počet tiků]**	**Maximální hodnota [počet tiků]**	**Průměrná hodnota [µs]**
Create/Delete Task	1 602	1 602	1 602	33,38
Ping Suspend/Resume Task	1 022	1 022	1 022	21,29
Suspend/Resume Task	778	778	778	16,21
Ping Mutex	1 317,82	1 300	1 318	27,46
Get/Release Semaphore	266	266	266	5,54
Get/Release Mutex	305	305	305	6,36
Queue Fill	260,99	260	261	5,44
Queue Drain	224,99	224	225	4,69
Queue Fill/Drain	430,02	430	431	8,96
Ping Fill/Drain	2 542	2 542	2 542	52,97
Allocate/Deallocate Memory	206,02	206	207	4,29
Time Calls	109,04	109	111	2,27

Tabulka 7.5: Výsledky měření SSC-benchmark testů pro µC/OS-III

7.2.3 Srovnání μC/OS-II a μC/OS-III

Tabulka 7.6 ukazuje srovnání výsledků měření pomocí *SSC-benchmark* testů pro *µC/OS-II* a *µC/OS-III*. Toto srovnání je rovněž přehledně vyobrazeno na obrázku 7.2.

Výsledky *SSC-benchmarku* potvrdily naměřené výsledky pomocí metody *Rhealstone*. Opět dosáhl lepších výsledků *RTOS µC/OS-II*, kromě testu alokace/dealokace paměti.

Výsledky testů, které vyžadují k provedení dvě úlohy, jsou v uvozovkách zkresleny dobou trvání přepnutí kontextu. Opět se však nejedná o chybu metodiky měření, metoda *SSC-benchmark* je takto postavena. V této metodě je doba přepnutí kontextu započítána do výsledku cíleně, jelikož například v testu *Create/Delete Task* se nemůže nově vytvořená úloha sama odstranit, aniž by předtím byla spuštěna.

V testech *Create/Delete Task* a *Get/Release Mutex* dosáhly oba operační systémy podobných výsledků, jen mírně lepší byl *µC/OS-II*.

Naopak v testech zpracování zpráv, tedy *Queue Fill*, *Queue Drain* a *Queue Fill/Drain* byly rozdíly výrazné, konkrétně v uvedeném pořadí testů byl rychlejší *µC/OS-II* o 20 %, 40 % a 35 %.

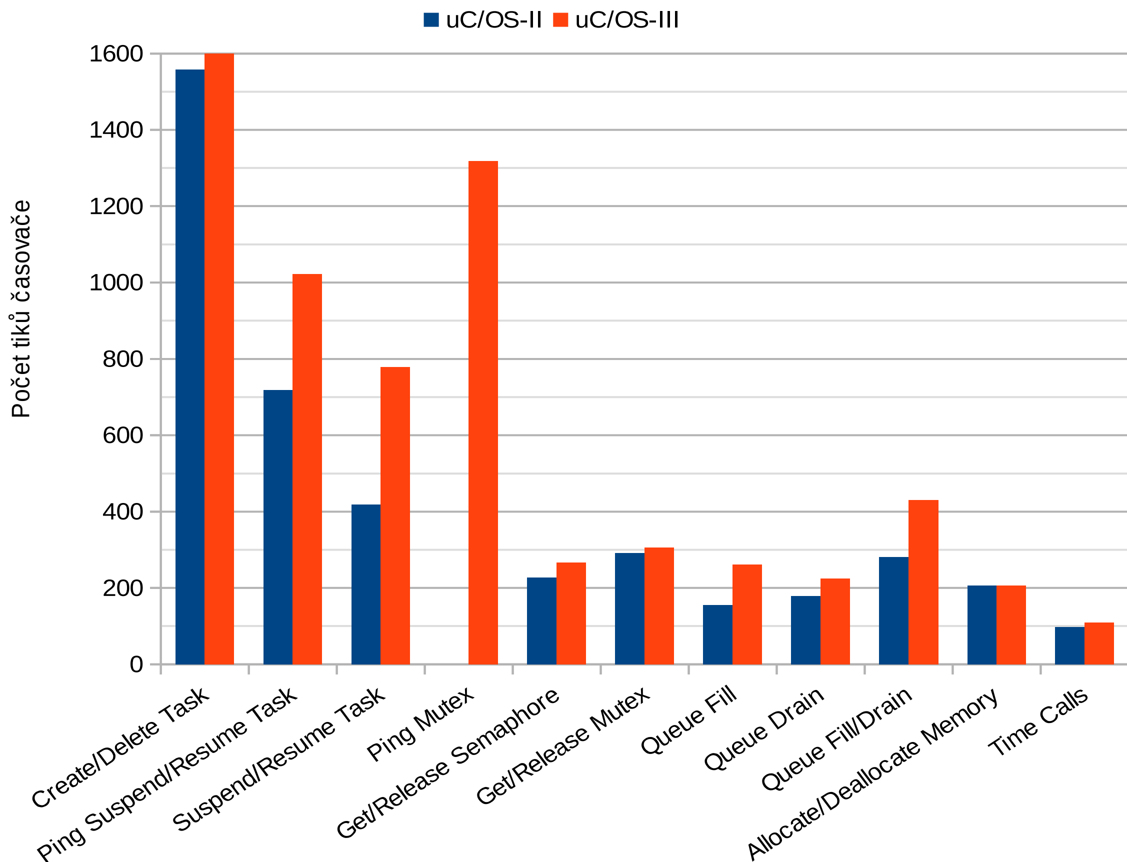
V testu *Allocate/Deallocate Memory* byly potvrzeny výsledky měření metodou *Thread-Metric*, tedy i zde dosáhl lepších výsledků *RTOS µC/OS-III*. Výsledek se nezdá tak výrazný, jako v případě metody *Thread-Metric*, nicméně zde je uveden výsledek jedné iterace alokace/dealokace paměti, zatímco v metodě *Thread-Metric* byla paměť cyklicky alokována a dealokována po dobu 30 sekund.

Nejvyššího rozdílu mezi oběma systémy bylo dosaženo v testu *Suspend/Resume Task*, který zvládl *µC/OS-II* o více než 46 % rychleji, než *RTOS µC/OS-III*.

Možné důvody dosažených výsledků jsou diskutovány v sekci 7.5.

Měřená operace	**µC/OS-II**		**µC/OS-III**	
	Průměrná hodnota [počet tiků]	**Průměrná hodnota [µs]**	**Průměrná hodnota [počet tiků]**	**Průměrná hodnota [µs]**
Create/Delete Task	1 558	32,47	1 602	33,38
Ping Suspend/Resume Task	718	14,96	1 022	21,29
Suspend/Resume Task	418	8,71	778	16,21
Ping Mutex	-	-	1 317,82	27,46
Get/Release Semaphore	228	4,75	266	5,54
Get/Release Mutex	291,02	4,06	305	6,36
Queue Fill	155,04	3,23	260,99	5,44
Queue Drain	179	3,73	224,99	4,69
Queue Fill/Drain	280,81	5,85	430,02	8,96
Ping Fill/Drain	-	-	2 542	52,97
Allocate/Deallocate Memory	207	4,31	206,02	4,29
Time Calls	98,01	2,04	109,04	2,27

Tabulka 7.6: Porovnání µC/OS-II a µC/OS-III pomocí SSC-benchmark testů



Obrázek 7.2: Porovnání µC/OS-II a µC/OS-III pomocí SSC-benchmark testů

7.3 Rhealstone

V této sekci jsou uvedeny naměřené výsledky pomocí metody *Rhealstone*. Oproti ostatním metodám není uvedena minimální a maximální naměřená hodnota, ale pouze průměrná hodnota, jelikož princip této metody je mírně odlišný a výpočet minimální/maximální hodnoty není možný, aniž by ovlivnil výsledky měření.

Frekvence použitého čítače/časovače *FlexTimer* byla nastavena na maximální možnou hodnotu, která je dána polovinou nastavené frekvence mikrokontroléru, tedy 47,989 MHz, čímž bylo dosaženo přesnosti měření 20,838 ns.

Každé z jednotlivých měření bylo provedeno vícekrát pro minimalizaci chyby způsobené rozlišením časovače *FlexTimer*. Konkrétní počet provedení měření byl různý pro každou aplikaci, jelikož musela být brána v potaz maximální hodnota čítače/časovače *FlexTimer*.

Tabulka 7.7 obsahuje výsledky měření pro operační systémy reálného času *µC/OS-II* a *µC/OS-III*. Pro názornost jsou tyto výsledky zaneseny rovněž do grafu 7.3.

Nejvyššího rozdílu mezi oběma systémy bylo dosaženo při měření doby preepmce úlohy, kdy doba preempce úlohy je v *RTOS µC/OS-III* o více než polovinu delší, než v případě *µC/OS-II*. Právě doba preempce úlohy je z pohledu výkonnosti operačního systému reálného času velice důležitá, jelikož k preempci úloh dochází v systému velice často.

Test doby přepnutí úlohy vyžaduje dvě úlohy shodné priority, proto jej nebylo možné provést pro *µC/OS-II*. Pokud porovnáme dobu přepnutí úlohy a dobu preempce úlohy pro

μC/OS-III, zjistíme, že byl potvrzen předpoklad, že doba preepmce úlohy bude vyšší, jelikož v sobě zahrnuje i dobu přepnutí úlohy.

Test přehození semaforu opět nebylo možné provést pro *μC/OS-II* kvůli požadavku na dvě úlohy se stejnou prioritou.

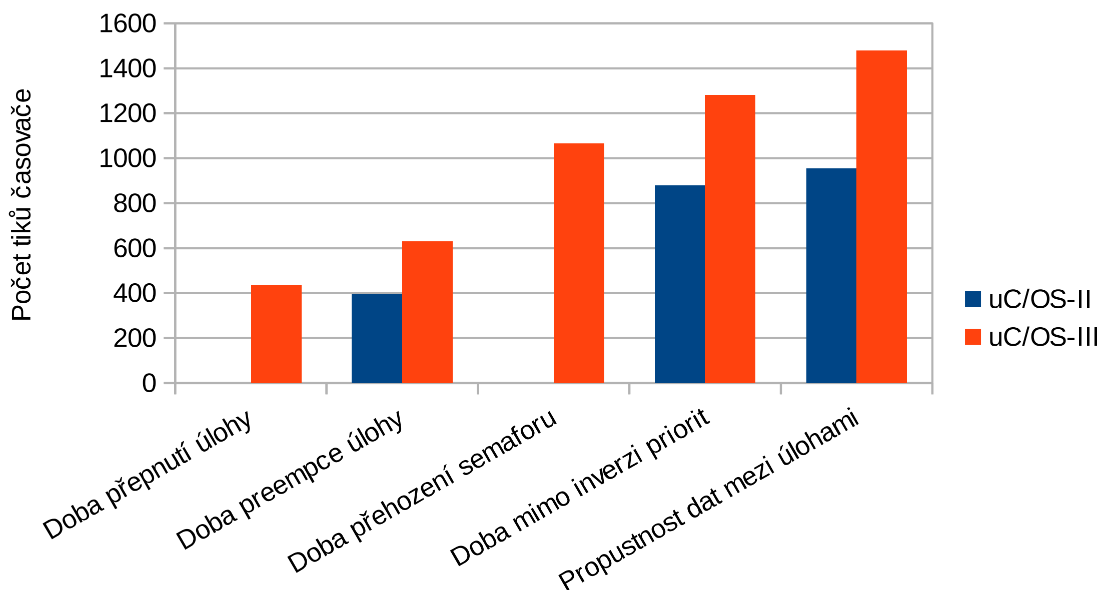
Doba mimo inverzi priorit je o 31 % kratší v systému *μC/OS-II*.

Propustnost dat mezi úlohami lze vypočítat jako podíl velikosti zasílané zprávy a naměřené doby odeslání/přijetí zprávy. Pro *μC/OS-II* činí tedy propustnost dat mezi úlohami přibližně 1,61 Mb/s a pro *μC/OS-III* přibližně 1,04 Mb/s při zasílání zprávy o velikosti 4 B. Jelikož je v době odeslání/přijetí zprávy započteno dvakrát přepnutí kontextu, je hodnota propustnosti dat mezi úlohami závislá na velikosti zprávy.

Možné příčiny dosažených výsledků jsou diskutovány v sekci 7.5.

Měřená operace	**μC/OS-II**		**μC/OS-III**	
	Průměrná hodnota [počet tiků]	**Průměrná hodnota [μs]**	**Průměrná hodnota [počet tiků]**	**Průměrná hodnota [μs]**
Doba přepnutí úlohy	-	-	436,1375	9,09
Doba preepmce úlohy	395,2	8,24	628,65	13,09
Doba přehození semaforu	-	-	1 065,4	22,21
Doba mimo inverzi priorit	879	18,32	1 279,56	26,66
Propustnost dat mezi úlohami	955,575	19,91	1 477,367	30,79

Tabulka 7.7: Porovnání μC/OS-II a μC/OS-III pomocí metody Rhealstone



Obrázek 7.3: Porovnání μC/OS-II a μC/OS-III pomocí metody Rhealstone

7.4 Vlastní testy

V této sekci jsou obsaženy naměřené výsledky pomocí vlastních testů. Nejprve samostatně pro *µC/OS-II* a *µC/OS-III*. Následně jsou dosažené výsledky u obou *RTOS* porovnány.

Frekvence použitého čítače/časovače *FlexTimer* byla nastavena na hodnotu 47,989 MHz, čímž bylo dosaženo přesnosti měření 20,838 ns.

Každé z jednotlivých měření bylo provedeno celkem stokrát pro eliminaci chyby. V tabulkách 7.8 a 7.9 je pak uvedena dosažená minimální, maximální a průměrná hodnota, jenž byla vypočtena jako aritmetický průměr ze všech naměřených hodnot. Dále je uvedena dosažená průměrná hodnota v mikrosekundách, která je však jen orientační, jelikož je zaokrouhlena pouze na dvě desetinná místa.

7.4.1 µC/OS-II

Tabulka 7.8 obsahuje naměřené výsledky pro operační systém reálného času *µC/OS-II* pomocí vlastních testů.

Měřená operace	**Průměrná hodnota [počet tiků]**	**Minimální hodnota [počet tiků]**	**Maximální hodnota [počet tiků]**	**Průměrná hodnota [µs]**
Zamknutí semaforu	152	152	152	3,17
Odemknutí semaforu	140,01	140	141	2,92
Zamknutí mutexu	176,02	176	177	3,67
Odemknutí mutexu	171,04	170	172	3,56
Alokace paměti	205,04	204	206	4,27
Dealokace paměti	137,04	136	138	2,86
Vytvoření úlohy	962	962	962	20,05
Odstranění úlohy	439	439	439	9,15
Pozastavení úlohy	176	176	176	3,67
Probuzení úlohy	297,02	296	298	6,19

Tabulka 7.8: Výsledky měření vlastních testů pro µC/OS-II

7.4.2 µC/OS-III

V tabulce 7.9 jsou obsaženy naměřené výsledky pomocí vlastních testů pro operační systém reálného času *µC/OS-III*.

7.4.3 Srovnání µC/OS-II a µC/OS-III

Výsledky těchto testů ukázaly, některé skutečnosti, které nebyly odhaleny metodami *Thread-Metric* a *SSC-benchmark*, jelikož v těchto metodách byly všechny operace testovány vždy po dvojicích, tedy zamknutí/odemknutí semaforu, vytvoření/odstranění úlohy atd.

Operace zamknutí semaforu, zamknutí mutexu a odstranění úlohy trvají kratší dobu v operačním systému reálného času *µC/OS-III*. Pokud však k dobám trvání těchto operací připočteme dobu provedení příslušné opačné operace, tedy odemknutí semaforu, odemknutí mutexu a vytvoření úlohy, jsou potvrzeny výsledky metody *SSC-benchmark*. Tedy, že *µC/OS-II* dosahuje souhrnně vyšší výkonnosti.

Měřená operace	**Průměrná hodnota [počet tiků]**	**Minimální hodnota [počet tiků]**	**Maximální hodnota [počet tiků]**	**Průměrná hodnota [µs]**
Zamknutí semaforu	146	146	146	3,04
Odemknutí semaforu	178,02	178	180	3,71
Zamknutí mutexu	168,04	167	169	3,49
Odemknutí mutexu	195	195	195	4,06
Alokace paměti	134,99	134	135	2,81
Dealokace paměti	132,99	132	133	2,77
Vytvoření úlohy	1 592	1 592	1 592	33,17
Odstranění úlohy	132,02	132	134	2,75
Pozastavení úlohy	400	400	400	8,34
Probuzení úlohy	437,03	436	439	9,11

Tabulka 7.9: Výsledky měření vlastních testů pro µC/OS-III

Test pozastavení a uspání úlohy potvrdil závěry měření z ostatních metod, že právě v těchto operacích jsou největší rozdíly mezi *µC/OS-II* a *µC/OS-III* ve prospěch *µC/OS-II*.

Možné důvody dosažených výsledků jsou diskutovány v sekci 7.5.

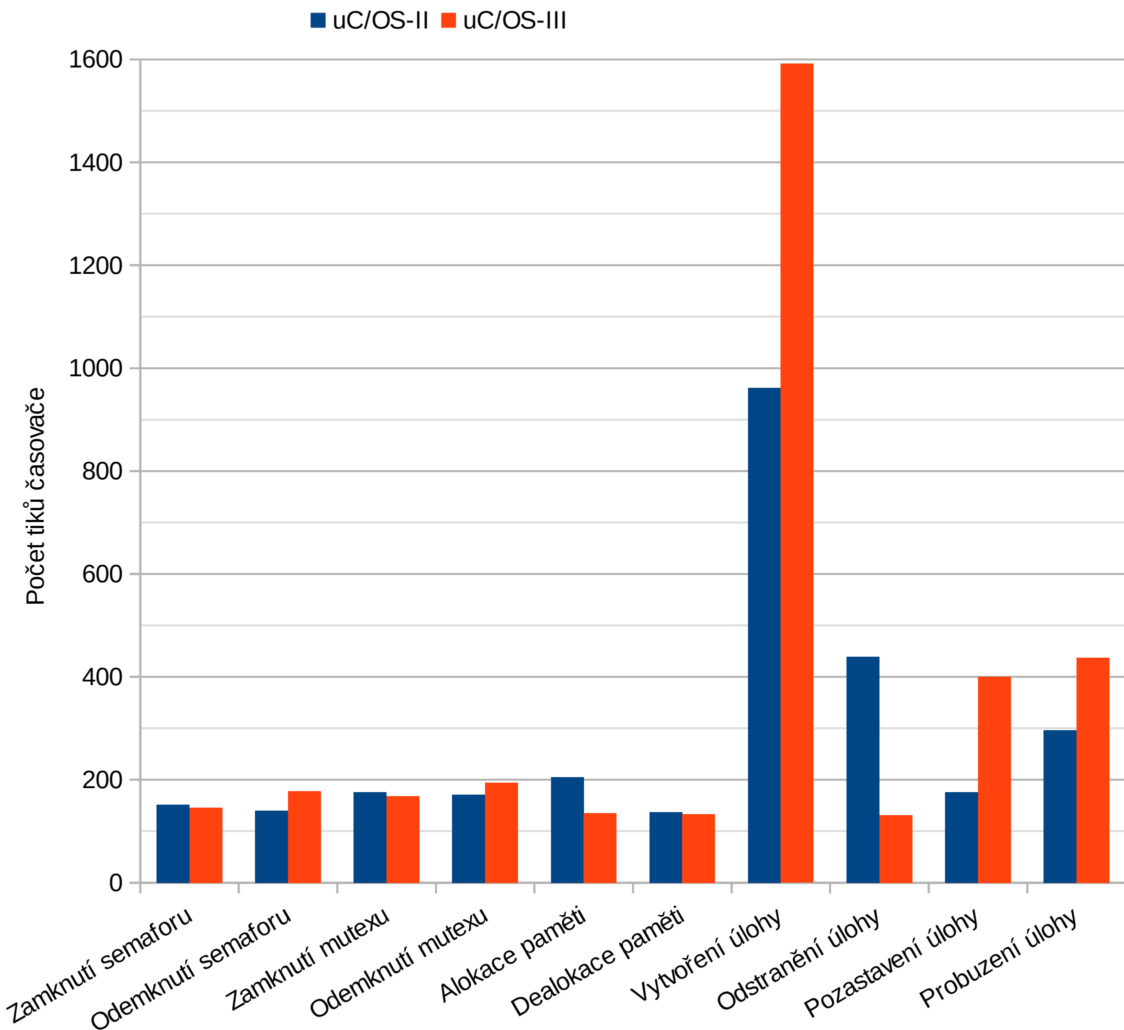
Měřená operace	**µC/OS-II**		**µC/OS-III**	
	Průměrná hodnota [počet tiků]	**Průměrná hodnota [µs]**	**Průměrná hodnota [počet tiků]**	**Průměrná hodnota [µs]**
Zamknutí semaforu	152	3,17	146	3,04
Odemknutí semaforu	140,01	2,92	178,02	3,71
Zamknutí mutexu	176,02	3,67	168,04	3,49
Odemknutí mutexu	171,04	3,56	195	4,06
Alokace paměti	205,04	4,27	134,99	2,81
Dealokace paměti	137,04	2,86	132,99	2,77
Vytvoření úlohy	962	20,05	1 592	33,17
Odstranění úlohy	439	9,15	132,02	2,75
Pozastavení úlohy	176	3,67	400	8,34
Probuzení úlohy	297,02	6,19	437,03	9,11

Tabulka 7.10: Porovnání µC/OS-II a µC/OS-III pomocí vlastních testů

7.5 Vyhodnocení a diskuze

Na základě výsledků všech testovacích metod lze říci, že operační systém reálného času *µC/OS-II* je výkonnější, než *µC/OS-III*. Jelikož je *µC/OS-III* novější verzí, mohly by se výsledky na první pohled zdát překvapivé. Nicméně mnohé o přepokládaných výsledcích již naznačila kapitola 3, ve které byla uvedena stručná charakteristika obou operačních systémů reálného času.

Systém *µC/OS-III* obsahuje pokročilejší plánovač, jelikož umožňuje vytvářet více úloh se stejnou prioritou. Rovněž uchovávání úloh připravených k běhu v paměti je tak u *RTOS*



Obrázek 7.4: Porovnání µC/OS-II a µC/OS-III pomocí vlastních testů

µC/OS-III složitější. Zatímco u *µC/OS-II* jsou ve dvojrozměrném poli dle priorit úloh uloženy ukazatele na úlohy, v *µC/OS-III* jsou ve dvojrozměrném poli dle priorit úloh uloženy pouze ukazatele na oboustranně vázané seznamy, které dále obsahují ukazatele na příslušné úlohy dané priority. Tedy rovněž výběr následující úlohy k běhu je složitější. Tímto jsou vysvětleny výsledky měření, které se týkaly plánování a vytváření úloh, tedy některých testů *Thread-Metric* a *Rhealstone*. Tato skutečnost má vliv rovněž na výsledky dalších provedených testů, které vyžadovaly k běhu více úloh a ve výsledku tak je obsažena i doba přepnutí kontextu. To se především týká testů *SSC-benchmark*.

Co se týče výsledků dalších testů, zaměřených na další objekty jádra, tedy semafory, mutexy a fronty zpráv, vysvětlení lze hledat při pohledu na aplikační programové rozhraní jednotlivých funkcí těchto objektů. Pokud se podíváme na prototypy jednotlivých funkcí, zjistíme, že funkce operačního systému *µC/OS-III* požadují obecně více parametrů. Pohledem do zdrojových kódů lze odhalit, že kód funkcí pracujících s objekty jádra je v *µC/OS-III* rozsáhlejší, než v *µC/OS-II*. Kromě přizpůsobení systému *µC/OS-III* tak, aby byl použitelný pro univerzálnější systémy reálného času je důvodem také sjednocení aplikačního

programového rozhraní v *µC/OS-III*, které se mohlo jevit poměrně nepřehledně v *RTOS µC/OS-II*.

Z naměřených hodnot je zřejmé, že provedení operace po dvojicích (zamknutí/odemknutí semaforu, alokace/dealokace paměti) tak, jak bylo měřeno v testech metody *SSC-benchmark*, trvá vždy kratší dobu, než když každou z operací provedeme samostatně, tedy zvlášť změříme zamknutí a odemknutí semaforu apod. a dosažené výsledky sečteme, což bylo předmětem měření dle vlastních testů. Tento rozdíl je způsoben vlivem vyrovnávací (*cache*) paměti.

Lze tedy říci, že operační systém reálného času *µC/OS-II* je vhodný spíše pro konkrétněji zaměřené systémy reálného času, kde nepotřebujeme například více úloh se stejnou prioritou. V takovém případě může být operační systém reálného času *µC/OS-II* vhodnou volbou z důvodu své výkonnosti.

Na druhou stranu *µC/OS-III* můžeme označit za univerzálnější systém reálného času. Tento systém byl rozšířen o koncepty, které lze nalézt v jiných operačních systémech reálného času, např. *FreeRTOS* či *QNQ*, tak, aby poskytoval vhodnou alternativu právě těchto systémů, které jsou velice používané při vývoji systémů reálného času. Právě možnost univerzálnějšího využití si však vyžádala snížení výkonnosti oproti *µC/OS-II*.

Kapitola 8

Závěr

Operační systémy reálného času jsou využívány při vývoji vestavěných systémů založených na mikrokontrolérech. Pomáhají zkrátit vývojový cyklus, zlepšit udržovatelnost a usnadnit programování. Výkonnost použitého operačního systému reálného času tak může být velice důležitá, zvláště v případě, pokud se jedná o *Hard RT* systémy.

Cílem této diplomové práce bylo porovnat výkonnost jader operačních systémů reálného času *µC/OS-II* a *µC/OS-III*. V rámci řešení bylo implementováno několik standardních metod pro srovnávání výkonnosti operačních systémů reálného času, konkrétně metody *Rhealstone*, *Thread-Metric* a *SSC-benchmark*. Nepovedlo se implementovat dva testy ze zmíněných metod týkající se přerušení. Rovněž bylo navrženo několik dalších testů, které nebyly součástí uvedených standardních metod.

Z dosažených výsledků implementovaných testovacích metod *Rhealstone*, *Thread-Metric* a *SSC-benchmark* vyplynulo, že operační systém reálného času *µC/OS-II* je výkonnější, než *µC/OS-III*. Jedinou oblastí, ve které dosahoval lepších výsledků *RTOS µC/OS-III*, byla práce s pamětí. Vlastní testy však ukázaly, že *RTOS µC/OS-III* zvládá rychleji i operace zamknutí semaforu, zamknutí mutexu a odstranění úlohy. Na druhou stranu operace odemknutí semaforu, odemknutí mutexu a vytvoření úlohy trvají v *µC/OS-III* déle, než v *µC/OS-II*, což tedy potvrzuje výsledky měření metodami *Thread-Metric* a *SSC-benchmark*.

Oba operační systémy reálného času se jeví poměrně stabilní, co se týká doby provádění měřených operací, jelikož naměřené minimální a maximální hodnoty jednotlivých testů se liší pouze nepatrně či dokonce vůbec.

Pokud jde tedy v aplikaci čistě o výkonnost, lze doporučit spíše systém *µC/OS-II*. Naopak pokud jde o flexibilitu a poskytované služby, je lepším systémem *µC/OS-III*, který umožňuje vytvářet více úloh se stejnou prioritou, konfiguraci za běhu a podobně. Rovněž programování aplikací je pohodlnější pro operační systém reálného času *µC/OS-III*, jelikož oproti *µC/OS-II* bylo sjednoceno aplikační programové rozhraní, které bylo v *RTOS µC/OS-II* poměrně nepřehledné.

Během zpracování této diplomové práce jsem nenarazil na žádný zdroj, který by pojednával o srovnání výkonnosti jader *µC/OS-II* a *µC/OS-III*. Výsledky této práce by tak mohly být přínosné širšímu spektru návrhářů systémů reálného času.

Pokračováním této práce by mohlo být testování využití paměti, zátěžové (stresové) testování či testování spotřeby elektrické energie. Zátěžové testování by mohlo být zajímavé z pohledu zda a do jaké míry klesne výkonnost systému při zvyšující se zátěži, případně jakou roli přitom hraje čas. Tedy zda při vyšším zatížení bude výkonnost systému v čase konstantní nebo bude postupně klesat. Velikost paměti je důležitým kritériem při volbě

mikrokontroléru, které významně ovlivňuje jeho cenu. Je tedy dobré vědět, kolik paměti je třeba pro využití daného operačního systému reálného času. Při nedostatečné velikosti paměti nebude možné daný *RTOS* využít, při naddimenzování velikosti paměti zase zaplatíme vyšší cenu za konkrétní mikrokontrolér. Spotřeba elektrické energie je důležitá v systémech reálného času, které jsou napájeny z baterie.

Literatura

[1] Li, Q. a Yao, C. *Real-Time Concepts for Embedded Systems.* 1st ed. Lawrence, Kansas: CMP Books, 2003. 294 s. ISBN 978-1-57820-124-2.

[2] IIT Kharagpur. *Real-Time Systems: Introduction* [online]. Version 2. [cit. 10. prosince 2015]. Dostupné na: <http://nptel.ac.in/courses/Webcourse-contents/IIT%20Kharagpur/Real%20time%20system/pdf/Module1.pdf>.

[3] Madl, G. *Model-Based Analysis of Event-driven Distributed Real-Time Embedded Systems.* Irvine: University of California, 2009. Ph.D. Thesis. Dostupné na: <http://dre.sourceforge.net/madl-dissertation.pdf>.

[4] Laplante, P. A. *Real-Time Systems Design and Analysis.* 3rd ed. New Jersey: IEEE Press, 2004. 480 s. ISBN 978-0-471-64828-4.

[5] IIT Kharagpur. *Real-Time Systems: Commercial Real-Time Operating Systems* [online]. Version 2. [cit. 12. prosince 2015]. Dostupné na: <http://nptel.ac.in/courses/Webcourse-contents/IIT%20Kharagpur/Real%20time%20system/pdf/module5.pdf>.

[6] Labrosse, J. J. *MicroC/OS-IIl: The Real-Time Kernel.* 2nd ed. Lawrence, Kansas: CMP Books, 2002. 606 s. ISBN 978-1-57820-103-7.

[7] Labrosse, J. J. *µC/OS-III: The Real-Time Kernel.* 2nd ed. Weston: Micriµm Books, 2011. 916 s. ISBN 978-0-9823375-3-0.

[8] Express Logic, Inc. *Thread-Metric Benchmark Suite* [online]. [cit. 13. prosince 2015]. Dostupné na: <http://rtos.com/PDFs/MeasuringRTOSPerformance.pdf>.

[9] Gräbner, O., Küfner, H., Stierschneider, O. et al. *Assesing Microsoft Windows CE 3.0 Real-Time Capabilities* [online]. 2001 [cit. 23. ledna 2016]. Dostupné na: <https://msdn.microsoft.com/en-us/library/ms834456.aspx>.

[10] Weiss, A. R. *Dhrystone Benchmark: History, Analysis, "Scores"and Recommmendations* [online]. 2002 [cit. 5. března 2016]. Dostupné na: <http://www.eembc.org/techlit/datasheets/dhrystone_wp.pdf>.

[11] Curnow, H. J. a Wichmann, B. A. A Synthetic Benchmark. *The Computer Journal.* 1976, roč. 19, č. 1. S. 43–49.

[12] Weiderman, N. *Hartstone: Synthetic Benchmark Requirements for Hard Real-Time Applications.* Pittsburgh: Software Engineering Institute, Carnegie Mellon University,

June 1989. 14 s. Technical Report, CMU/SEI-89-TR-023. Dostupné na: <http://www.sei.cmu.edu/reports/89tr023.pdf>.

[13] Donohoe, P., Shapiro, R. a Weiderman, N. *Hartstone Benchmark User's Guide, Version 1.0.* Pittsburgh: Software Engineering Institute, Carnegie Mellon University, March 1990. 92 s. Technical Report, CMU/SEI-90-UG-1. Dostupné na: <http://www.sei.cmu.edu/reports/90ug001.pdf>.

[14] SPEC. *SPEC CPU 2006* [online]. [cit. 21. ledna 2016]. Dostupné na: <https://www.spec.org/cpu2006/>.

[15] SPEC. *SPEC Benchmarks* [online]. [cit. 21. ledna 2016]. Dostupné na: <https://www.spec.org/benchmarks.html>.

[16] EEMBC. *CoreMark: An EEMBC Benchmark: CoreMark FAQ* [online]. [cit. 21. ledna 2016]. Dostupné na: <http://www.eembc.org/coremark/faq.php>.

[17] Gal-On, S. a Levy, M. *Exploring CoreMark – A Benchmark Maximizing Simplicity and Efficacy* [online]. 2010 [cit. 21. ledna 2016]. Dostupné na: <http://www.eembc.org/techlit/coremark-whitepaper.pdf>.

[18] Guthaus, M. R., Ringenberg, J. S., Ernst, D. et al. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop.* Washington, DC: IEEE Computer Society, 2001. S. 3–14. WWC '01. ISBN 0-7803-7315-4.

[19] Nemer, F., Cassé, H., Sainrat, P. et al. PapaBench: a Free Real-Time Benchmark. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06).* Dagstuhl: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2006. OpenAccess Series in Informatics (OASIcs). ISBN 978-3-939897-03-3.

[20] Freescale Semiconductor, Inc. *K60P144M100SF2V2* [online]. June 2012, Rev. 3, 6/2013 [cit. 11. února 2016]. Dostupné na: <http://cache.nxp.com/files/32bit/doc/data_sheet/K60P144M100SF2V2.pdf?fpsp=1&WT_TYPE=Data%20Sheets&WT_VENDOR=FREESCALE&WT_FILE_FORMAT=pdf&WT_ASSET=Documentation&fileExt=.pdf>.

[21] ARM. *ARMv7-M Architecture Reference Manual* [online]. June 2006, revised 2 December 2014 [cit. 14. března 2016]. Dostupné na: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0403e.b/index.html>.

[22] ARM. *ARM Cortex-M4 Processor: Technical Reference Manual* [online]. December 2009, Revision r0p1 [cit. 14. března 2016]. Dostupné na: <http://infocenter.arm.com/help/topic/com.arm.doc.100166_0001_00_en/arm_cortexm4_processor_trm_100166_0001_00_en.pdf>.

[23] Freescale Semiconductor, Inc. *Kinetis Design Studio IDE: Integrated Development Environment (IDE) for Kinetis MCUs* [online]. 2014, Revision 2 [cit. 16. března 2016]. Dostupné na: <http://cache.nxp.com/files/microcontrollers/doc/fact_sheet/

KINDESTDSOFS.pdf?fpsp=1&WT_TYPE=Fact%20Sheets&WT_VENDOR=FREESCALE&
WT_FILE_FORMAT=pdf&WT_ASSET=Documentation&fileExt=.pdf>.

[24] Freescale Semiconductor, Inc. *Kinetis Software Development Kit* [online]. 2014, Revision 5 [cit. 16. března 2016]. Dostupné na:
<http://cache.nxp.com/files/microcontrollers/doc/fact_sheet/
SDKKINETMCUFS.pdf?fpsp=1&WT_TYPE=Fact%20Sheets&WT_VENDOR=FREESCALE&
WT_FILE_FORMAT=pdf&WT_ASSET=Documentation&fileExt=.pdf>.

[25] Kar, R. P. Implementing the Rhealstone Real-Time Benchmark. *Dr. Dobb's Journal.* 1990, roč. 15, č. 4. S. 46–55. ISSN 1044-789X.

Přílohy

Seznam příloh

A Obsah CD **68**

Příloha A

Obsah CD

Přiložené CD obsahuje:

- Technickou zprávu ve formátu PDF
- Zdrojové soubory této technické zprávy
- Projekty obsahující testovací úlohy
- Textový soubor s popisem projektů a jejich spuštění *README.txt*
- Textový soubor s naměřenými výsledky *vysledky.txt*