VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

SADA APLIKACÍ PRO DEMONSTRACI REAL-TIME VLASTNOSTÍ μC/OS-III NA PLATFORMĚ FITKIT 3

SET OF APPLICATIONS FOR DEMONSTRATION OF REAL-TIME PROPERTIES OF μC/OS-III ON FITKIT 3 PLATFORM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **VÍT KOUTNÝ**
AUTHOR

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

BRNO 2017

Zadání bakalářské práce/19437/2016/xkoutn21

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Koutný Vít**
Obor: Informační technologie
Téma: **Sada aplikací pro demonstraci real-time vlastností uC/OS-III na platformě FITkit 3**
Set of Applications for Demonstration of Real-Time Properties of uC/OS-III on FITkit 3 Platform
Kategorie: Operační systémy

Pokyny:

1. Přehledově zdokumentujte architekturu platformy FITkit 3 - zaměřte se na prostředky mikrokontroléru Freescale Kinetis K60 s jádrem ARM Cortex-M4 (dále jen K60) na FITkit 3 a principy tvorby aplikací založených na K60.
2. Přehledově shrňte základní pojmy z oblasti real-time (RT) operačních systémů (RTOS) a vytvořte přehled RTOS dostupných pro FITkit 3.
3. Zdokumentujte architekturu a služby jádra uC/OS-III, procesorově specifické soubory a ověřte funkčnost jimi poskytovaných funkcí.
4. Demonstrujte a zdokumentujte funkčnost jednotlivých služeb uC/OS-III, poté sady zvolených RT aplikací běžících nad tímto jádrem. Změřte doby provádění vybraných částí jádra či aplikací a při činnosti aplikací využijte prvky dostupné na platformě FITkit 3.
5. Dosažené výsledky shrňte, interpretujte jejich význam a diskutujte jejich přesnost.
6. Zhodnoťte vlastnosti portu uC/OS-III pro FITkit 3.

Literatura:

- NXP Semiconductors [on-line]. Dokument dostupný na http://www.nxp.com
- Micrium - Empowering Embedded Systems [on-line]. Dokument dostupný na http://www.micrium.com.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Strnadel Josef, Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstrakt

Tato bakalářská práce seznamuje čtenáře se základními pojmy z oblasti real-time systémů a real-time operačních systémů. Popisuje operační systém μC/OS-III, zejména jeho jádro, základní datové struktury a služby aplikačního rozhraní. Nachází se zde také popis metod, jakými je možné testovat operační systémy reálného času. Některé z těchto metod byly naimplementovány, na jejich základě bylo provedeno měření a výsledky jsou v závěru práce diskutovány.

Abstract

This Bachelor's thesis provides an introduction to fundamental terms of real-time systems and real-time operating systems. It describes operating system μC/OS-III, focuses on its kernel, data structures and application programming interface. There is also a description of methods which can be used to test real-time operating systems. Some of these methods were implemented, the measurement was performed afterwards and results were written at the end of this thesis.

Klíčová slova

μC/OS-III, operační systémy reálného času, benchmarking, Rhealstone, Thread-Metric, FITkit 3, real-time systémy, RTOS

Keywords

μC/OS-III, real-time operating system, benchmarking, Rhealstone, Thread-Metric, FITkit 3, real-time systems, RTOS

Citace

KOUTNÝ, Vít. *Sada aplikací pro demonstraci real-time vlastností μC/OS-III na platformě FITkit 3*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Strnadel Josef.

Sada aplikací pro demonstraci real-time vlastností μC/OS-III na platformě FITkit 3

Prohlášení

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

.....
Vít Koutný
7. května 2017

Poděkování

Rád bych velmi poděkoval panu Ing. Josefu Strnadelovi, PhD. za vstřícnost a ochotu poskytovat mi cenné rady v průběhu řešení celé bakalářské práce.

Obsah

Kapitola 1

Úvod

Operační systémy reálného času se vyskytují v mnoha zařízeních okolo nás. Můžeme je najít v real-time systémech, které jsou součástí vestavěných systémů v automobilech, letadlech, měřicích přístrojích, v zařízeních pro monitorování životních funkcí pacienta a na mnohých jiných místech. Společným jmenovatelem všech těchto zařízení je požadavek na správnou a bezchybnou činnost, ale také na včasnost reakcí na příchozí podněty. Operačních systémů reálného času existuje celá řada. Takovým operačním systémem je například RT-Linux, VxWorks, MQX, Windows CE nebo μC/OS-III. Posledně zmíněným operačním systémem se bude zabývat tato bakalářská práce, konkrétně jeho portem na platformu FITkit 3.

Cílem této práce je navrhnout a implementovat sadu aplikací, které budou na platformě FITkit 3 demonstrovat real-time vlastnosti operačního systému μC/OS-III. Práce je rozdělena do sedmi kapitol.

V kapitole 2 se nachází stručný popis starších verzí platformy FITkit, následuje popis samotné platformy FITkit 3 a v závěru jsou zmíněny způsoby, jakými lze na FITkit vytvářet aplikace.

V kapitole 3 je zmíněna základní teorie z oblasti real-time operačních systémů doplněná o real-time operační systémy použitelné na platformě FITkit 3.

Kapitola 4 pojednává o samotné architektuře jádra operačního systému μC/OS-III. Je zde popsán blok řízení úlohy (TCB), stavy, do kterých se úloha může dostat, seznamy připravených a blokovaných úloh a způsob plánování úloh.

V kapitole 5 se nachází popis implementace sady aplikací, kterými byla ověřena funkčnost služeb μC/OS-III. Tato sada aplikací se skládá z implementace Rhealstone a Thread-Metric benchmarku doplněná o sadu vlastních testů.

Kapitola 6 obsahuje výsledky všech provedených testů ve formě tabulek a grafů. Tyto výsledky jsou zde také diskutovány a v samotném závěru kapitoly se nachází zhodnocení vlastností portu μC/OS-III pro platformu FITkit 3.

Kapitola 7 obsahuje závěr, zhodnocení celé práce a nástin možného dalšího pokračování.

Kapitola 2

Architektura platformy FITkit

FITkit je samostatná platforma pro praktickou podporu výuky v hardwarově orientovaných předmětech na Fakultě informačních technologií Vysokého učení technického v Brně. Tento kit si může vypůjčit každý student Fakulty informačních technologií. V dnešní době existuje více verzí platformy FITkit:

- FITkit verze 1.0,
- FITkit verze 1.2,
- FITkit verze 2.0,
- FITkit verze 3.0.

2.1 Popis starších verzí platformy FITkit

FITkit verze 1.x [4] se skládá ze 16-bitového mikrokontroléru MSP430 od firmy Texas Instruments. Tento mikrokontrolér má k dispozici 48kB FLASH paměti a 2kB RAM paměti. Tento kit dále obsahuje FPGA XC3S50-4PQ208C řady Spartan 3 od firmy Xilinx, USB převodník FT2232C, jednořádkový LCD displej, maticovou klávesnici o 16 klávesách, konektory PS2, rozhraní VGA, konektor RS232 a rozšiřující konektory.

U FITkitu 2.0 [4] došlo k nahrazení původního mikrokontroléru z FITkitu verze 1.x výkonnějším modelem MSP430F2616, který má více FLASH paměti (92kB) a také více RAM paměti (8kB). Tento mikrokontrolér obsahuje vylepšené sériové rozhraní podporující nezávislý běh SPI/I2C a UART na jednom kanálu, výkonný audio kodek TLV320AIC23B a propojku J5 sloužící k aktivaci přerušení z FPGA.

2.2 Celková architektura platformy FITkit 3

FITkit verze 3 je nejvyšší verzí platformy FITkit, která v současné době existuje. Oproti předchozím verzím došlo k mnoha změnám. Na obrázku A.1 je zachycen FITkit 3 s vyznačenými důležitými komponentami. [8]

FITkit 3 na rozdíl od starších verzí nemá klávesnici ani LCD displej. Konektorová výbava zahrnuje HDMI konektor (1) A.1, Ethernet RJ45 konektor (2) A.1, konektor USB typ B (3) A.1, konektor pro externí napájení (4) A.1, konektor USB typ A (5) A.1, audio konektory (6) A.1, rozšiřující konektory (7) A.1 a slot pro SD kartu (nacházející se na spodní straně kitu).

Integrované obvody nacházející se na kitu jsou následující: rozhraní zobrazení DVI Texas Instruments (TFP410PAP) (8) A.1, LAN řadič SMSC 10/100 Ethernet (8720A) (9) A.1, 4-port USB Hub Texas Instruments (TUSB2046B) (10) A.1, převodník USB/-COM FTDI Vinculum-II (VNC2) (11) A.1, stereo Audio kodek Freescale (SGTL5000) (12) A.1, řadič napájení Texas Instruments (TPS65251RHA) (13) A.1. MCU Freescale HCS08 (MC9S08JM60) je mikrokontrolér s 8 bitovou sběrnicí, 64kB FLASH paměti, 4kB RAM paměti a jádrem S08 taktovaným na frekvenci 48MHz (14) A.1. Na desce FITkitu 3 se také nachází programovatelné hradlové pole XC6SLX9 řady Spartan 6 od firmy Xilinx (16) A.1. Hlavním mikrokontrolérem na kitu je Freescale Kinetis MK60DN512ZVMD10 (17) A.1. Tento 32 bitový mikrokontrolér obsahuje jádro ARM Cortex-M4 na frekvenci 100MHz, 512kB FLASH paměti, 128kB RAM paměti, rozhraní USB, UART, SPI, I2C a CAN. Na FITkitu 3 se dále nachází paměť DRAM o velikosti 64MB, která je taktována na frekvenci 333MHz (18) A.1. Konkrétně se jedná o paměť typu DDR2 SDRAM (IS43DR16320B). Firmware kitu se nachází v paměti typu NAND flash od výrobce STMicroelectronics s 8 bitovou sběrnicí a velikostí 4Mbit (19) A.1.

Ostatní vybavení kitu obsahuje oddělovače, linkové budiče (SN74HCT125D) (17) A.1, sadu 7 tlačítek (21) A.1, 2 sady LED diod (23) (20) A.1 a reproduktor (22) A.1.

FITkit verze 2 a 3 lze doplnit o rozšiřující moduly. Jedním z těchto modulů je modul Ethernet pro FPGA. Tento modul obsahuje 20 pinový konektor, kterým lze připojit k rozšiřujícím konektorům FITkitu. Součástí modulu je konektor RJ45 a tento modul je kompatibilní s 10Mbit/100Mbit standardem v sítích LAN/WAN. Dalším rozšiřujícím modulem je modul RF 2.4GHz, který umožnuje FITkitu komunikovat v bezlicenčním pásmu 2.4GHz. Modul obsahuje integrovanou anténu a ke kitu se připojuje pomocí 20 pinového konektoru.

2.3 Princip tvorby aplikací

Existuje vícero možností, jak lze vytvářet aplikace na platformu FITkit. Záleží na typu a rozsahu aplikace, kterou chceme naprogramovat, na zkušenostech programátora a v neposlední řadě také na verzi FITkitu. V následujících odstavcích budou krátce zmíněny možnosti, jak programovat FITkit verze 1.0 až 2.0 [4]. Následně bude zmíněno programování FITkitu verze 3.

2.3.1 QDevKit

Multiplatformní nástroj s názvem QDevKit byl vytvořen na Fakultě informačních technologií a značně zjednodušuje práci s FITkity verze 1.x a 2.x. Tento program nabízí uživateli přehledné grafické rozhraní, kde je v pravé části možné vidět připojený FITkit (může jich být i více) a v levé části je seznam projektů, kterými lze FITkit naprogramovat. Tento seznam lze naplnit již vytvořenými projekty z SVN repozitáře. Následně lze označit konkrétní FITkit a pravým tlačítkem kliknout na program, který si uživatel přeje nahrát a přes zobrazené kontextové menu FITkit naprogramovat.

2.3.2 fkterm, fcmake, fkflash

fkterm

Uživatelé, kteří nechtějí používat QDevKit s grafickým uživatelským prostředím, si mohou nainstalovat fkterm. Fkterm je program, který umožňuje komunikaci s FITkitem přes příkazový řádek.

fcmake

Program fcmake slouží ke generování Makefile souborů pro syntézu z popisového souboru project.xml. Tento program je součástí QDevKitu.

fkflash

Utilita fkflash slouží k naprogramování mikrokontroléru a FPGA na platformě FITkit. Hlavní výhodou tohoto programu oproti QDevKitu je urychlení naprogramování. Mikrokontrolér MSP430F168 na FITkitu verze 1.x není kompatibilní s mikrokontrolérem MSP430F2617, který se nachází u FITkitu verze 2.x. Z tohoto důvodu je nutné programu fkflash ve vstupních parametrech specifikovat 2 různé HEX soubory a v průběhu programování tato utilita vybere jeden z nich na základě verze FITkitu, na který se programuje.

2.3.3 MSP430

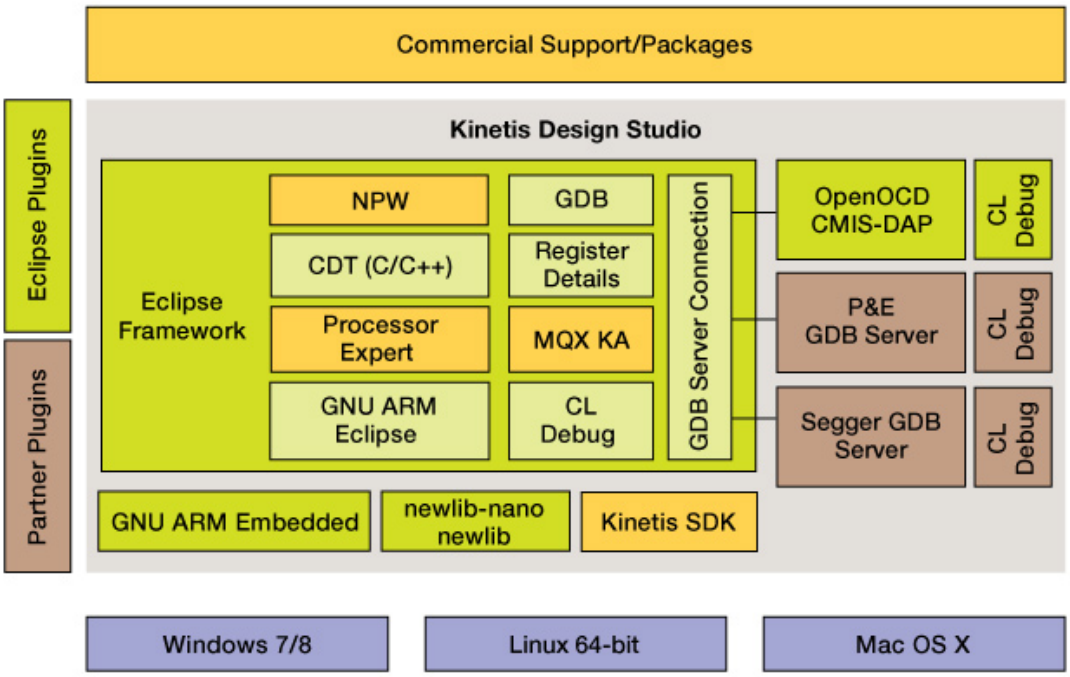
MSP430 je sada utilit, které překládají kód napsaný v assembleru nebo v jazyce C a následně tento přeložený kód programují na FITkit. Nevýhodou je nízká rychlost a možnost programovat pouze MCU a nikoli FPGA.

2.3.4 Kinetis Design Studio

K programování složitějších a větších aplikací, které budou běžet pod nějakým operačním systémem reálného času, je téměř nutností použít nějaký pokročilejší nástroj, který programátorovi ulehčí práci. Tímto nástrojem může být například Kinetis Design Studio (dále jen KDS) od společnosti NXP.

KDS obsahuje robustní integrované vývojové prostředí (IDE) založené na open source platformě Eclipse. KDS umožňuje vytváření zdrojových kódů, následně jejich překlad pomocí kompilátoru GNU Compiler Collection (GCC), pokročilé možnosti ladění kódu skrze GNU Debugger (GDB), Processor Expert pro urychlení vývoje aplikací pro určitý mikrokontrolér. Společnost NXP dále nabízí Source Development Kit (SDK) pro KDS. Tento SDK si může každý uživatel nakonfigurovat podle svých potřeb na stránkách společnosti NXP[1]. Zde je možné si vybrat daný mikrokontrolér, pro který bude aplikace vyvíjena, součástí SDK je také sada ovladačů periferií a v neposlední řadě si zde uživatel také může zvolit, který operační systém reálného času bude používat. Po vygenerování příslušného SDK si jej uživatel stáhne a nainstaluje do KDS.

[1] Zdroj: https://mcuxpresso.nxp.com/en/welcome



Obrázek 2.1: Kinetis Design Studio - blokový diagram[2]

Alternativou ke KDS je vývojové prostředí CodeWarrior. Výhodou KDS je však jeho multiplatformnost a také podpora nejnovějších mikrokontrolérů Kinetis architektury ARM. Na obrázku 2.1 je zobrazen blokový diagram vývojového prostředí KDS. Pro vývoj demonstračních aplikací pro platformu FITkit 3 v rámci této bakalářské práce bude využito prostředí KDS s SDK vygenerovaným pro Kinetis K60 včetně operačního systému μC/OS-III.

[2]Zdroj: http://www.nxp.com/assets/images/en/block-diagrams/31846_KDS_IDE_BD.jpg

Kapitola 3

Operační systémy reálného času

Operační systémy reálného času (dále jen RTOS) se využívají u složitějších real-time systémů, ve kterých běží více úloh. Těmto úlohám je nutné přidělovat procesorový čas a systémové prostředky v závislosti na jejich účelu a prioritách. RTOS poskytují robustní a současně flexibilní prostředky pro víceúlohové prostředí s cílem splnit všechny real-time požadavky daného systému. Tato kapitola čerpá z knihy Real-time operační systémy [3].

3.1 Jádro a správa procesů

Nejdůležitější částí RTOS je jeho jádro. Jádro operačního systému reálného času slouží pro plánování, komunikaci a synchronizaci real-time úloh. Existuje několik typů jader operačních systému reálného času.

3.1.1 Pseudojádro

Pokud je nutné v navrhovaném RT systému zajistit víceúlohovost, avšak navrhovaný systém není tak složitý, aby bylo nezbytně nutné použít nějaký RTOS, je vhodné využít mechanismu pseudojádra v některé z jeho implementací, které budou nastíněny dále. Jednou z hlavních výhod využití pseudojádra je velmi dobrá analyzovatelnost výsledného RT systému a mnohdy také jeho vyšší výkonnost, jelikož RT systém není zatížen režií RTOS. Typy pseudojader:

- vyzývací smyčka,
- synchronizovaná vyzývací smyčka,
- cyklické provádění,
- stavově řízený kód,
- spolupracující úlohy.

Vyzývací smyčka

Nejjednodušším typem pseudojádra je vyzývací smyčka. Vyzývací smyčka se skládá s nekonečného cyklu, ve kterém se stále dokola testuje, zda došlo nebo nedošlo k očekávané události. Pokud k události došlo, tak se zavolá příslušná obslužná úloha.

Synchronizovaná vyzývací smyčka

Synchronizovaná vyzývací smyčka je velmi podobná klasické vyzývací smyčce. Obsahuje navíc časovou prodlevu mezi začátkem události a její obsluhou, což je vhodné použít například k potlačení zákmitů při stisku tlačítka nebo sepnutí spínače.

Cyklické provádění

Cyklické provádění úloh spočívá v sekvenčním volání jednotlivých úloh v nekonečné smyčce. Tento přístup poskytuje iluzi současného běhu úloh. Pro případnou komunikaci mezí úlohami je nutné využít globální proměnné nebo seznamy parametrů. V případě potřeby volání některé úlohy častěji než ostatních úloh, je nutné úlohu zavolat více než jedenkrát za jeden cyklus. Hlavní nevýhodou tohoto přístupu je, že jen málokdy jsou úlohy volané uvnitř smyčky stejně časově náročné.

Stavově řízený kód

Stavově řízený kód bývá implementován v podobě konečného automatu. Tento přístup napomáhá víceúlohovosti systému. Výhodou je možnost zvýšení efektivity výsledné aplikace formální optimalizací daného konečného automatu. Stavově řízený kód nelze použít jako obecný přístup k návrhu pseudojádra, jelikož ne každá úloha může být popsána konečným automatem.

Spolupracující úlohy

Posledním a nejkomplikovanějším typem pseudojádra je implementace ve formě spolupracujících úloh. Každá úloha je implementována pomocí stavově řízeného kódu, přičemž je rozdělena na několik částí. Po skončení každé fáze dojde k volání plánovače. Plánovač musí být programátorem implementován jako součást pseudojádra. Plánovač si musí uložit kontext právě skončené úlohy, vybrat další úlohu, která poběží a načíst její kontext. Komunikace úloh mezi sebou se zužuje na využití globálních proměnných.

3.1.2 Jádro založené na využití přerušovacího systému

U tohoto typu jádra je plánovač realizován pomocí obslužných rutin HW či SW přerušení. Přerušení je vyvoláno nějakým zdrojem hodinového signálu, následně se řadičem přerušení nebo programově vyhodnotí, který požadavek je nejdůležitější. Plánovač poté pozastaví aktuálně probíhající úlohu a spustí úlohu odpovídající příchozímu požadavku.

Hlavní tělo cyklu je tvořeno nekonečnou smyčkou. Běh této smyčky je přerušen pouze tehdy, pokud přijde nemaskované přerušení. Po příchodu takového přerušení se provede ISR (interrupt service routine) pro toto přerušení a řízení programu se vrátí zpět do nekonečné smyčky.

Výhodou tohoto přístupu ke tvorbě RT aplikací je přehlednost kódu a krátké doby odezev na příchozí události. Naopak nevýhodou je nutnost detailně znát přerušovací systém HW platformy, globálně zakazovat přerušení při provádění kritické sekce, které může vést až ke špatné reakci RT systému, dále plýtvání časem v nekonečné smyčce, obtížnost poskytování náročnějších služeb apod. Z těchto důvodů se jádro založené na využití přerušovacího systému příliš nepoužívá.

3.1.3 Systémy pracující v popředí/pozadí

Systémy pracující v popředí/pozadí (anglicky foreground/background systems, FBS) jsou vylepšené systémy založené na využití přerušovacího systému. Hlavní tělo kódu zde také tvoří nekonečná smyčka, ve které se ale vykonává užitečná práce. Tato užitečná práce tvoří tzv. pozadí systému. Popředí systému tvoří úlohy vyvolané přerušeními. Tyto úlohy mají vysokou prioritu narozdíl od úloh běžících v pozadí. Úlohy běžící v pozadí by neměly vykonávat žádnou časově kritickou činnost a nejsou vyvolávány přerušeními.

FBS zlepšuje využitelnost CPU. Výhodou těchto systémů je krátká odezva na události. Nevýhodou je nutnost reimplementace rozhraní, jejich ovladačů atd.

3.1.4 TCB model

Tento model je založen na tzv. blocích řízení úlohy (anglicky task-control block, TCB). Každé úloze je přiřazen právě jeden TCB, což je datová struktura uchovávající informace o identifikaci, prioritě a stavu úlohy. Dále obsahuje také hodnotu programového čítače, hodnoty ostatních registrů a ukazatel na další TCB.

Bloky řízení úloh si většinou jádro uchovává ve dvou jednosměrně vázaných lineárních seznamech. Jeden seznam obsahuje bloky připravených úloh a druhý seznam uchovává bloky blokovaných úloh. Samotné jádro v tomto modelu běží jako úloha s nejvyšší prioritou.

Tento model je velmi populární, protože umožňuje za běhu RT aplikace měnit počet RT úloh. TCB model je využíván i v operačním systému μC/OS-III.

3.2 Správa paměti

Správa paměti je kritickou částí každého operačního systému. Každá úloha, která má běžet pod operačním systémem, potřebuje k tomuto běhu určitou paměť. Aby víceúlohový RT systém běžel podle očekávání, je nutné, aby správa paměti byla vhodně naimplementována. Špatná implementace může vést k narušení determinismu systému, případně může dojít k selhální systému jako celku. V několika následujících odstavcích budou popsány možné přístupy ke správě paměti a budou zhodnoceny jejich výhody a nevýhody.

3.2.1 Zásobník

Jednou z možností správy paměti ve víceúlohovém RTOS je využití jednoho čí více zásobníků pro ukládání a obnovování kontextů jednotlivých úloh. Zásobník se využívá u jednodušších RT systémů založených pouze na přerušeních či u foreground/background systémů. Při využití zásobníku je velmi důležité stanovit jeho správnou velikost tak, aby nedošlo k jeho přetečení ani v nejhorším možném případě.

3.2.2 TCB

U složitějších RT systémů lze využít TCB modelu pro správu paměti. Kontext každé úlohy se ukládá v rámci bloků řízení úlohy (TCB), které se následně spojují do jednoho nebo více lineárních seznamů. V případě jednoho seznamu obsahujícího TCB všech úloh již v době generování RT systému se jedná o tzv. statický seznam TCB. Do statického seznamu nelze za běhu RT systému vkládat nové úlohy, ani z něj odebírat již nepotřebné úlohy. V tomto

případě není nutná žádná správa paměti. Dále existuje i tzv. dynamický TCB model. V případě dynamického modelu je nutné zabývat se správou paměti, jelikož lze úlohy přidávat i odstraňovat.

3.2.3 Swap

V případě využití odkládacího prostoru (anglicky swap) se v hlavní paměti nachází pouze jádro RTOS a aktuálně běžící úloha. Ostatní úlohy jsou odkládány do sekundární paměti, kterou obvykle představuje pevný disk. Vzniká zde problém s velkou časovou ztrátou při přepnutí kontextu, jelikož přístup na disk je časově náročný. Tento model není v praxi příliš využíván, jeho výhodou je však nízká náročnost kladená na primární paměť.

3.2.4 Překrývání

Pokud není k dispozici dostatek systémové paměti pro běh určité úlohy, lze tuto úlohu rozčlenit na dílčí části a v jednom okamžiku mít v hlavní paměti pouze jeden segment této úlohy. Ostatní segmenty úlohy jsou uloženy v sekundární paměti. Tento model také není v praxi příliš využíván z důvodů zhoršení RT vlastností systému a složitější implementace úloh, které mohou být rozčleněny na menší části.

3.2.5 Stránkování a správa bloků paměti

Paměťový prostor je možné rozdělit na bloky a ty poté přidělovat a odebírat jednotlivým úlohám podle jejich potřeb. V tomto modelu vzniká problém s následnou fragmentací paměti, který musí být řešen speciální úlohou zajišťující defragmentaci úlohy. Další možností je využití stránkování. Vzniká zde ale další režie v případě výpadku stránky a následného přístupu do paměti. V praxi se tento model používá velmi zřídka. Výkonnost systému v případě využití stránkování lze vylepšit technikou zamykání paměti, kdy jsou části úloh či celé úlohy uzamknuty v paměti a nedochází tím pádem k jejich odložení do sekundární paměti.

3.3 Správa souborů

U souborových systémů vzniká problém s fragmentací stejně jako u primární paměti. Dále je vhodné ukládat soubory na disk takovým způsobem, aby po sobě následovaly alokační bloky příslušného souboru. Tímto způsobem dojde ke snížení fragmentace i ke zrychlení přístupu k souborům na disku.

3.4 Základní pojmy z oblasti RT systémů a RTOS

V této podkapitole budou přehledově sepsány a vysvětleny základní pojmy z oblasti real-time systémů a real-time operačních systémů.

Nejprve je nutné definovat, co představuje RT systém. **RT systém** je systém, který musí správně reagovat na vstupní podněty a navíc musí být tato reakce provedena v určitém čase. Pokud je reakce RT systému špatná či pomalá, následkem takové reakce může být **selhání systému** jako celku. RT systém je obvykle součástí nějakého vestavěného systému. **Vestavěný systém** (embedded system) je systém vykonávající jednoúčelovou službu. Takový systém obsahuje řídicí počítač, který je zabudován do zařízení, které obsluhuje. Jedním z klíčových pojmů z oblasti RT systémů je **doba odezvy systému**. Je

to doba od výskytu vstupních podnětů na vstupech systému do provedení příslušné reakce systému, včetně výskytu výsledných hodnot na výstupech systému. Při návrhu RT systémů je nutné znát doby odezev na různé podněty a zejména mít přehled o nejhorších možných případech, aby nedošlo v důsledku příliš dlouhé doby odezvy k selhání systému.

Všechny RT systémy však nemusí nutně selhat při nedodržení jedné či více předem stanovených časových mezí. Takovými RT systémy jsou tzv. **soft RT systémy**, jejichž výkon je v důsledku nedodržení časových mezí degradován, ale nevede nutně k selhání celého systému. Dále existují **firm RT systémy**, u kterých nedodržení několika časových odezev nevede k selhání celého systému, ale při větším počtu nedodržení časových mezí již k selhání dojde. Nejstriktnějším RT systémem z hlediska časových odezev a jejich dodržování jsou **hard RT systémy**. Nedodržení jediné časové meze v tomto systému vede k jeho kompletnímu selhání.

Důležitou činností RT systémů je reakce na příchozí události. **Událost** je podnět, který způsobí skokovou změnu toku řízení programu. Určité události v systému lze předvídat. Jedná se o tzv. **synchronní události**. Příkladem může být změna řízení toku způsobená podmíněným skokem. Naopak události, které nelze předem očekávat, se nazývají **asynchronní události** a jsou obvykle způsobeny zdrojem, který se nachází vně systému. Z hlediska časů příchodů je možné události rozdělit na periodické, aperiodické a sporadické. **Periodická událost** je událost, ke které dochází v pravidelných intervalech. **Aperiodická událost** je událost, ke které nedochází v pravidelných intervalech a **sporadická událost** je událost, ke které dochází jen velmi vzácně.

Každý systém se může dostat do stavu, který se vymyká kontrole. Aby bylo možné se z této situace dostat, musí v RT systému existovat **jednotka pro zachování činnosti systému**. Ke správné funkci této jednotky je nutné mít oddělenou paměť programu a paměť dat. Další podmínkou pro zotavení činnosti systému je jeho determinismus. **Deterministický systém** je takový systém, pro jehož libovolný stav a libovolnou množinu vstupních podnětů lze jednoznačně určit následující stav a množinu výstupních odezev. Pokud v takovém systému známe také doby odezev, potom se jedná o **temporálně (časově) deterministický systém**. Takový systém garantuje správnou reakci na podněty v předem známém časovém intervalu a z hlediska RT systémů představuje velmi významnou vlastnost.

3.5 Real-time operační systémy pro FITkit 3

Real-time operačních systémů existuje celá řada. Některé jsou šířeny jako svobodný software, jiné jsou proprietární software. Při výběru konkrétního operačního systému reálného času pro výslednou real-time aplikaci je nutné vzít do úvahy několik faktorů. Nejdůležitější je určit, zda k naprogramování zadané aplikace je vůbec potřeba operační systém reálného času, nebo jestli je možné se spokojit například pouze s pseudojádrem založeným na vyzývací smyčce. Jestliže programátor dojde k závěru, že je nutné použít nějaký RTOS, tak nastává problém s výběrem konkrétního operačního systému. Vhodný operační systém by měl poskytovat všechny nezbytné funkce, které bude aplikační programátor potřebovat. Na druhou stranu by neměl poskytovat zbytečné funkce, které budou zhoršovat celkovou odezvu systému. Dále je nutné vzít do úvahy cílovou platformu, na které se bude daný RTOS provozovat. Zejména jestli bude dostatek operační paměti pro běh RTOS a také pro běh konkrétních úloh. V neposlední řadě je nutné vybrat operační systém, který bude kompatibilní s procesorem na dané platformě, případně vytvořit port tohoto operačního systému tak, aby byl kompatibilní.

Seznam operačních systémů vhodných pro FITkit 3:

- μC/OS
- μC/OS-II
- μC/OS-III
- MQX
- BeRTOS
- FreeRTOS
- Nucleus RTOS

Kapitola 4

Architektura jádra μC/OS-III

Tato kapitola pojednává o jádru operačního systému μC/OS-III. Je zde popsána základní datová struktura uchovávající informace o každé úloze, stavy do kterých se úlohy mohou dostat, způsob jakým si operační systém μC/OS-III uchovává úlohy připravené k běhu a jakým způsobem vybírá další úlohu, která poběží. Při psaní této kapitoly bylo čerpáno z knihy μC/OS-III: The Real-Time Kernel [2].

4.1 Blok řízení úlohy

Blok řízení úlohy (anglicky task control block, TCB) je datová struktura ve které si jádro operačního systému uchovává informace o jednotlivých úlohách. Každá úloha má svůj vlastní TCB. Jednotlivé bloky si jádro operačního systému μC/OS-III uchovává ve dvou dvousměrně vázaných seznamech. Jeden seznam je tzv. ready-list, který obsahuje bloky úloh připravených k běhu a druhý seznam je tzv. pend-list obsahující bloky úloh, které čekají na přidělení určitého zdroje nebo na vypršení svého časovače. Programátor RT aplikací nemá přímý přístup k položkám této datové struktury, pouze jádro OS může k těmto položkám přistupovat a upravovat je. Samotná datová struktura, tak jak je definovaná v operačním systému μC/OS-III (`os.h`), je velmi rozsáhlá a proto zde bude uveden jen krátký výčet nejdůležitějších položek této struktury s krátkým popisem jejich významu. Kompletní podobu této struktury je možné nalézt v [2], str. 114-115.

```
struct os_tcb {
    CPU_STK          *StkPtr;
    void             *ExtPtr;
    CPU_STK          *StkLimitPtr;
    OS_TCB           *NextPtr;
    OS_TCB           *PrevPtr;
    OS_CHAR          *NamePtr;
    CPU_STK          *StkBasePtr;
    OS_STATE         TaskState;
    OS_PRIO          Prio;
    CPU_STK_SIZE     StkSize;
    void             *MsgPtr;
    OS_MSG_SIZE      MsgSize;
    OS_NESTING_CTR   SuspendCtr;
    OS_CPU_USAGE     CPUUsage;
    OS_CPU_USAGE     CPUUsageMax;
 }
```

StkPtr

Každá úloha v rámci μC/OS-III může mít svůj vlastní zásobník libovolné velikosti. StkPtr uchovává ukazatel na aktuální vrchol zásobníku dané úlohy.

ExtPtr

Ukazatel sloužící k rozšíření TCB o uživatelem definovaná data. Takové rozšíření je možné realizovat při vytváření úlohy skrze 11. argument funkce `OSTaskCreate()`.

StkLimitPtr

Tento ukazatel ukazuje na určité místo v zásobníku úlohy. Toto místo je určeno hodnotou `stk_limit` při volání funkce `OSTaskCreate()`. Účelem je chránit zásobník před přetečením a to buď hardwarovou cestou, která je efektivnější, nebo softwarovou cestou.

NextPtr

Ukazuje na další TCB v seznamu bloků připravených úloh (tzv. ready list).

PrevPtr

Ukazuje na předchozí TCB v seznamu bloků připravených úloh (tzv. ready list).

NamePtr

Ukazatel na jméno úlohy. Toto jméno může být libovolně dlouhé.

StkBasePtr

Ukazatel na bázovou adresu zásobníku příslušné úlohy.

TaskState

Obsahuje aktuální stav úlohy. Celkový počet stavů úlohy je 8 a každá úloha se nachází právě v jednom z těchto stavů.

Prio

Položka uchovávající prioritu úlohy. Nabývá hodnoty 1 až `OS_CFG_PRIO_MAX-2`. Nižší číslo značí vyšší prioritu.

StkSize

Položka obsahující velikost zásobníku v CPU_STK jednotkách.

MsgPtr

Pokud je úloze zaslána zpráva, MsgPtr je ukazatelem na tuto zprávu.

MsgSize

MsgSize obsahuje velikost zprávy v bytech, která byla zaslána dané úloze.

SuspendCtr

SuspendCtr je počítadlo, kolikrát byla úloha pozastavena (stav `suspended`). Pokud je zavolána funkce `OSTaskSuspend()`, tak se `SuspendCtr` zvýší o 1. Při volání funkce `OSTaskResume()` je naopak hodnota `SuspendCtr` snížena o 1. Pokud je hodnota rovna nule, tak úloha přechází do stavu `ready` (za předpokladu že již nečeká na přidělení jiných zdrojů). Toto počítadlo je nutné z důvodu možnosti vícenásobného pozastavení úlohy.

CPUUsage

Obsahuje vytížení CPU danou úlohou v procentech vynásobenou konstantou 100 (např. 10000 reprezentuje 100.00%).

CPUUsageMax

Obsahuje maximální vytížení CPU danou úlohou v procentech vynásobenou konstantou 100 (např. 10000 reprezentuje 100.00%).

4.2 Úlohy

Úloha je základní stavební jednotka real-time systémů. Vhodným vytvářením a skládáním úloh buduje programátor výslednou aplikaci, která bude běžet pod nějakým operačním systémem reálného času. V našem případě to bude operační systém μC/OS-III a proto bude v této podkapitole popsána obecná struktura úlohy a dále možnost vytvoření takové úlohy v tomto OS.

4.2.1 Obecná struktura úlohy

Úloha v rámci operačního systému μC/OS-III vypadá jako běžná funkce napsaná v jazyce C. Tato funkce však musí být typu `void` a tudíž nesmí nic vracet. Může mít ve svém těle deklarovány lokální proměnné a má také jeden vstupní argument `p_arg`. Argument `p_arg` je ukazatel na datový typ `void` a je to univerzální prostředek jak úloze při jejím prvotním spuštění předat potřebné informace (např. adresu proměnné, struktury nebo i funkce). Z těla úlohy je možné také volat další funkce. Tyto funkce musí být napsány v jazyce C nebo v jazyce symbolických instrukcí. Jednu a tu samou funkci je možné volat z různých úloh za předpokladu, že je daná funkce reentrantní. Z těla úlohy však není možné volat jinou úlohu. Volání resp. spouštění úloh je plně v režii operačního systému μC/OS-III. Dále se také nedoporučuje používat rekurzivní volání funkcí z důvodů omezené kapacity zásobníku a hrozby přetečení zásobníku vedoucí na obtížně odhalitelné chyby. Existují dva typy úloh: `run-to-completion` a `infinite loop`.

`Run-to-completion` je úloha, jejíž tělo se vykoná pouze jednou a úloha je poté odstraněna ze systému. Na konci svého těla úloha zavolá funkci `OSTaskDel()`, kterou se odstraní. Tento typ úloh se příliš často nepoužívá. Obvykle je úloha navržena za účelem periodického vykonávání dané činnosti.

`Infinite loop` je úloha, v jejímž těle se nachází nekonečná smyčka. V těle této smyčky se provádí užitečná práce, pro kterou byla tato úloha navržena a naprogramována. Na konci smyčky musí tato úloha zavolat nějakou službu operačního systému μC/OS-III a skrze toto volání čekat na určitou událost (např. vypršení časovače nebo přidělení požadovaného zdroje). Při čekání na tuto událost nespotřebovává úloha čas CPU. V případě nezavolání služby μC/OS-III by se z úlohy stala skutečná nekonečná smyčka. Tento typ úloh je mnohem častěji využíván při tvorbě vestavěných systémů.

4.2.2 Vytváření úlohy

Aby operační systém μC/OS-III o úloze, kterou má spouštět, vůbec věděl, je nutné ji nejprve správně vytvořit. K vytváření úloh slouží funkce `OSTaskCreate()`. Při volání této funkce je specifikován TCB, který bude přidělen této úloze. Dále se určí adresa a velikost zásobníku, priorita a ostatní parametry úlohy. V dalším textu bude naznačen a popsán prototyp funkce `OSTaskCreate()`.

```
void OSTaskCreate (
    OS_TCB        *p_tcb,       (1)
    OS_CHAR       *p_name,      (2)
    OS_TASK_PTR   p_task,       (3)
    void          *p_arg,       (4)
    OS_PRIO       prio,         (5)
    CPU_STK       *p_stk_base,  (6)
    CPU_STK_SIZE  stk_limit,    (7)
    CPU_STK_SIZE  stk_size,     (8)
    OS_MSG_QTY    q_size,       (9)
    OS_TICK       time_slice,   (10)
    void          *p_ext,       (11)
    OS_OPT        opt,          (12)
    OS_ERR        *p_err        (13)
 )
```

1. adresa TCB, který má být přiřazen této úloze,
2. funkce `OSTaskCreate()` umožňuje přiřadit každé úloze jméno o jakékoli délce sloužící k jednoduššímu ladění systému,
3. adresa těla úlohy,
4. argument v podobě ukazatele na data, který dostane úloha při svém prvním spuštění,
5. priorita úlohy,
6. bázová adresa zásobníku úlohy,
7. ukazatel na určité místo v zásobníku, které bude monitorováno za účelem zabránění přetečení zásobníku,
8. určuje velikost zásobníku úlohy v jednotkách `CPU_STK`,
9. specifikuje velikost volitelné interní fronty zpráv,
10. časové kvantum v hodinových cyklech (v případě povoleného round robin plánování),
11. ukazatel na uživatelem specifikovaná data (obvykle na strukturu) rozšiřující TCB,
12. slouží ke specifikaci přídavných možností (např. `OS_OPT_TASK_STK_CLR` znamená, že zásobník bude při inicializaci vynulován),
13. ukazatel na proměnnou, do které bude uložen chybový kód.

4.3 Stavy úloh

Každá úloha se v rámci operačního systému μC/OS-III nachází v některém z 8 možných stavů. Aktuální stav úlohy je uložen v jejím TCB v položce `TaskState`. Informace o stavu každé úlohy je kritická pro operační systém, jelikož na základě této informace zajišťuje přemístění TCB příslušné úlohy mezi seznamem blokovaných úloh a seznamem úloh připravených k běhu. Teprve až když se úloha nachází ve stavu připravená (ready), může být plánovačem vybrána na základě priority ke spuštění. Úloha může přecházet mezi stavy na základě konečného automatu zobrazeného na obrázku A.2. V dalším textu budou stručně představeny jednotlivé stavy úlohy.

(0) Ready

Úloha se nachází v tomto stavu v případě, že má všechny potřebné zdroje, které ke svému běhu potřebuje, nachází se v seznamu úloh připravených k běhu a čeká na přidělení CPU od plánovače.

(1) Delay

Ve stavu `delay` se úloha nachází v momentě, kdy čeká než uplyne nějaký čas. Tento čas může být specifikován voláním funkce `OSTimeDly()` nebo `OSTimeDlyHMSM()`. Čekání úlohy může být předčasně ukončeno voláním funkce `OSTimeDlyResume()`. Ve chvíli kdy skončí doba, po kterou měla úloha čekat, nebo po zavolání funkce `OSTimeDlyResume()`, se tato funkce vrací do stavu připravená k běhu (`ready`).

(2) Pend

Úloha se nachází ve stavu `pend`, pokud čeká na výskyt určité události. Do tohoto stavu se může úloha dostat po volání funkce `OSFlagPend()`, `OSMutexPend()`, `OSQPend()`, `OSSemPend()`, `OSTaskQPend()` nebo `OSTaskSemPend()`.V tomto stavu úloha čeká po neomezenou dobu na výskyt této události. Úloha se vrací do stavu `ready` v momentě, kdy tato událost nastane. Další možností jak se vrátit do stavu `ready` je zrušení tohoto čekání jinou úlohou nebo smazání objektu, na který úloha čeká.

(3) Pend Timeout

Tento stav je velmi podobný stavu `pend` s tím rozdílem, že v tomto případě úloha nečeká po neomezeně dlouhou dobu, ale pouze po předem daný časový interval. Po vypršení tohoto časového intervalu se úloha vrací do stavu `ready`. Po vypršení časovače a navrácení úlohy do stavu `ready`, je úloha informována o tom, že očekávaná událost nenastala.

(4) Suspend

Do stavu `suspend` se úloha dostane po zavolání funkce `OSTaskSuspend()`. Tuto funkci může zavolat úloha sama na sebe, nebo ji může zavolat jiná úloha. Aby se úloha mohla vrátit do stavu `ready`, je nutné, aby jiná úloha zavolala funkci `OSTaskResume()` na tuto úlohu.

(5) Delay Suspended

Tento stav vzniká kombinací stavů `delay` a `suspended`. Pro přechod do stavu `ready` je nutné, aby jiná úloha zavolala funkci `OSTaskResume()` na tuto úlohu. Dále úloha také musí vyčkat na uplynutí doby, po kterou má čekat. Teprve po splnění těchto podmínek se může úloha vrátit do stavu `ready`.

(6) Pend Suspended

Tento stav vzniká kombinací stavů `pend` a `suspended`. Úloha čeká na volání funkce `OSTaskResume()` od jiné úlohy a současně čeká na výskyt určité události. Teprve poté se může dostat do stavu `ready`.

(7) Pend Timeout Suspended

Posledním stavem je stav `pend timeout suspended`. Tento stav vzniká kombinací tří stavů a platí pro něj stejná pravidla, jako pro dva předchozí stavy. Úloha musí čekat na vypršení svého časovače, výskyt očekávané události a volání funkce `OSTaskResume()` od jiné úlohy.

4.4 Seznam připravených úloh

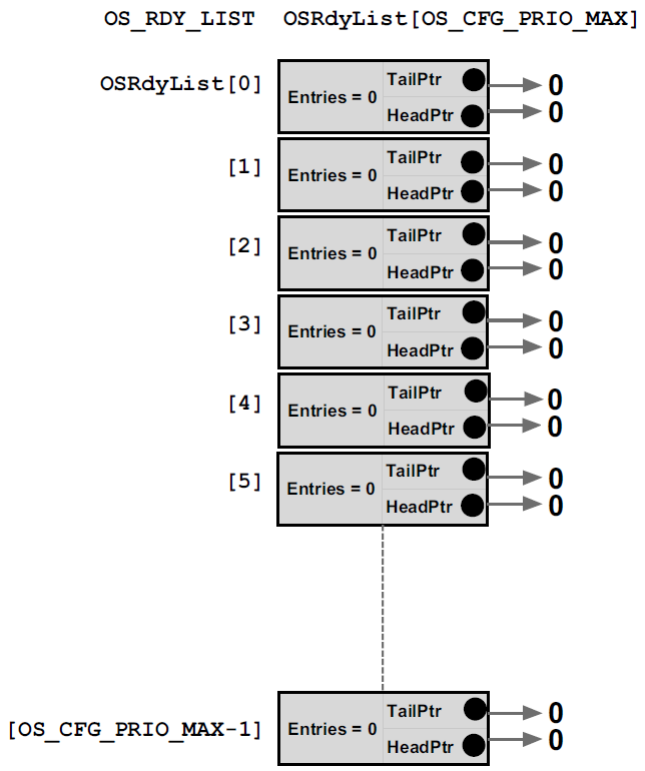
Úlohy připravené k běhu jsou vkládány do tzv. seznamu připravených úloh (anglicky the ready list). Tento seznam se skládá ze dvou částí. První část tvoří bitmapa obsahující informace o tom, pod kterými prioritami se skrývá jedna či více připravených úloh. Druhou částí je poté tabulka ukazatelů na připravené úlohy. Cílem je co nejrychleji najít připravenou úlohu s nejvyšší prioritou.

4.4.1 Bitmapa

Pokud je jedna nebo více úloh se stejnou prioritou připraveno k běhu, vloží se do bitmapy logická jednička na index odpovídající dané prioritě (například k úloze s prioritou 3 bude nastaven 3. bit v bitmapě na jedničku za předpokladu, že je tato úloha připravena k běhu). Úrovně priorit jsou číslovány zleva doprava, přičemž se číslovka udávající prioritu zvyšuje, což znamená snižování reálné priority. Tento přístup umožňuje využití speciální instrukce CLZ (Count Leading Zeros) k nalezení prvního nenulového bitu v této bitmapě. Stejně tak mohou být využity instrukce pro nastavení konkrétního bitu na logickou jedničku a instrukce pro vynulování konkrétního bitu. V případě že procesor tyto instrukce neobsahuje, je nutné zůstat u softwarové implementace těchto operací v jazyce C.

4.4.2 Tabulka

Ukazatelé na připravené úlohy se vkládají do tabulky nazývané the Ready List. The Ready List je pole OSRdyList[] obsahující OS_CFG_PRIO_MAX položek. Každá položka je typu OS_RDY_LIST a skládá se z Entries, TailPtr a HeadPtr. Entries obsahuje počet úloh připravených k běhu na jedné konkrétní prioritní úrovni. TailPtr a HeadPtr jsou ukazatelé sloužící k vytvoření dvojsměrně vázaného seznamu připravených úloh stejné priority. OSRdyList[] je tedy defacto pole seznamů připravených úloh, kde index jednotlivých položek pole udává prioritu dané úlohy či úloh.



Obrázek 4.1: Prázdný seznam připravených úloh[1]

[1] Zdroj: [2] str. 144

4.5 Plánování úloh

Plánovač slouží k výběru úlohy, která poběží jako další v pořadí. Jádro operačního systému μC/OS-III je preemptivní, což znamená, že v tomto OS vždy poběží úloha s nejvyšší prioritou. Pokud nastane událost, jejímž důsledkem přejde úloha, která má vyšší prioritu než současně běžící úloha, do stavu připravená, dojde k přepnutí kontextu a poběží úloha s vyšší prioritou. Plánování je plně v režii jádra μC/OS-III. Uživatel tedy nemůže ovlivnit, kdy k plánování dojde. Jedinou výjimkou je volání funkce OSSched(), skrze kterou si může uživatel plánování vynutit.

4.5.1 Round-Robin

μC/OS-III narozdíl od svých předchůdců umožňuje více úlohám přidělit stejnou prioritu a proto podporuje plánování Round-Robin. Při tomto způsobu plánování je úloze přiděleno určité časové kvantum. Po tuto dobu má úloha k dispozici CPU a po vypršení tohoto kvanta je úloze CPU odebrán. Úloha se sama od sebe může vzdát procesoru ještě před vypršením daného časového kvanta. Aplikační programátor může měnit implicitní hodnotu časového kvanta (i za běhu), povolovat a zakazovat plánování Round-Robin a přidělovat jednotlivým úlohám různá časová kvanta (i za běhu).

Kapitola 5

Popis metod a jejich implementace

V úvodu této kapitoly budou popsány metody pro měření a porovnání výkonnosti různých operačních systémů reálného času. Dále bude popsán způsob, jakým bude měřena rychlost provádění služeb jádra μC/OS-III. Následovat bude výpis nejdůležitějších služeb jádra μC/OS-III doplněný o získané hodnoty za různých podmínek. Při psaní této kapitoly bylo čerpáno z oficiální dokumentace společnosti Micrium [6].

5.1 Benchmarking operačních systémů reálného času

Benchmarking u operačních systémů reálného času slouží k otestování a porovnání výkonnosti různých implementací těchto operačních systémů. Naměřené výsledky by měly usnadnit aplikačnímu programátorovi volbu konkrétního operačního systému pro konkrétní použití. V následujících odstavcích budou zmíněny a přehledově popsány dvě metody pro testování výkonnosti.

5.1.1 Rhealstone

U Rhealstone benchmarku [1] se sleduje šest parametrů. Těmito parametry jsou:

1. průměrná doba přepnutí mezi dvěma úlohami se stejnou prioritou (Task Switching Time),
2. průměrná doba přepnutí z úlohy s nižší prioritou na úlohu s vyšší prioritou (Preemption Time),
3. průměrná doba mezi výskytem přerušení v procesoru a provedením první instrukce obsluhy tohoto přerušení (Interrupt Latency),
4. průměrná doba mezi odemčením semaforu jednou úlohou a spuštěním další úlohy, která čekala na odemčení tohoto semaforu (Semaphore Shuffling Time),
5. průměrná doba vyřešení uváznutí, které nastane ve chvíli, kdy je úloha s nižší prioritou vlastnící určitý zdroj přerušena a je spuštěna úloha s vyšší prioritou, která tento zdroj potřebuje ke svému běhu (Deadlock Breaking Time),
6. průměrné zpoždení, které nastane při zaslání zprávy mezi dvěma procesy (Intertask Messaging Latency).

Výsledkem této metody je jedno číslo získané váhovým součtem převrácených hodnot těchto šesti naměřených parametrů.

$$Rhealstone = \sum_{i=1}^{i=6} w_i * \frac{1}{t_i} \tag{5.1}$$

Tento způsob měření výkonnosti s sebou nese i určité problémy. Jedním z problémů je určení váhových koeficientů. Tyto koeficienty se určují empiricky v závislosti na tom, jak jsou dané operace zastoupeny v konkrétní aplikaci. Určení koeficientů je tedy zatíženo určitou nepřesností. Dalším problémem je skutečnost, že získané výsledky jsou průměrem naměřených hodnot. V oblasti RT systémů a RTOS je mnohem cennější informace o nejhorším možném průběhu dané aplikace.

5.1.2 Thread-Metric

Thread-Metric [7] je benchmark od firmy Express Logic, Inc., který iterativně provádí vybrané služby zkoumaného RTOS a každých třicet sekund vypisuje naměřené výsledky. Vybranými službami jsou:

- kooperativní přepínání kontextu (Cooperative Context Switching),
- preemptivní přepínání kontextu (Preemptive Context Switching),
- zpracování přerušení (Interrupt Handling),
- zasílání zpráv (Message Processing),
- zpracování semaforů (Semaphore Processing),
- alokace a dealokace paměti (Memory Allocation).

Tyto testy jsou prováděny po dvojicích (alokace a dealokace paměti, apod.)

Podobných benchmarků existuje daleko více - například SSC-Benchmark, MiBench, Hartstone, SNU Real-Time Benchmarks, PapaBench, RT_STAP Benchmark a další. Tyto benchmarky se však velmi často zaměřují na testování výkonnosti dané hardwarové platformy a nikoli čistě pouze na testování operačního systému.

5.2 Doby provádění služeb μC/OS-III

Aby bylo možné co nejpřesněji měřit doby provádění služeb operačního systému μC/OS-III, je nutné tyto doby měřit na nejnižší úrovni, tedy na úrovni jednotlivých taktů procesoru. Samotný operační systém μC/OS-III obsahuje softwarové prostředky pro měření těchto dob. Nicméně takovéto softwarové měření je nepřesné a hodí se spíše k měření dob provádění celých úloh s přesností na milisekundy. V našem případě bude nutné sáhnout po hardwarovém řešení tohoto problému.

Součástí procesoru ARM Cortex-M4 je tzv. jednotka DWT [5] (Data WatchPoint and Trace Unit). Tato jednotka obsahuje sadu registrů sloužící k ladění výsledné aplikace. Předtím než lze jednotku DWT použít, je nutné nastavit v registru DEMCR (Debug Exception and Monitor Control Register, adresa 0xE000EDFC) 24. bit na hodnotu 1. Následně

je možné s touto jednotkou pracovat. Před začátkem měření je nutné vynulovat registr DWT_CYCCNT (Cycle Count Register, adresa 0xE0001004), který slouží k počítání jednotlivých taktů. Aby počítání taktů probíhalo, musí být v registru DWT_CONTROL (Control Register, adresa 0xE0001004) nastaven první bit na hodnotu 1. Od této chvíle je hodnota uložená v registru DWT_CYCCNT zvýšena s každým taktem o 1. Po provedení určitého úseku kódu, jehož dobu provádění chceme zjistit, přečteme hodnotu z registru DWT_CYCCNT a zjistíme tak, kolik taktů trvalo provedení tohoto úseku. Toto měření s sebou přináší jistou režii, která se projeví ve výsledku. Výsledná naměřená hodnota je obvykle o 11 až 18 taktů vyšší. Tento rozptyl je způsoben přeskládáním instrukcí (tzv. Out-of-order execution).

5.3 Rhealstone benchmark

Rhealstone benchmark se skládá ze šesti úloh (viz 5.1.1). Pro každou úlohu byl vytvořen jeden projekt, ve kterém byla příslušná úloha naimplementována. Měření doby provádění každé úlohy se dělí na dvě části. V první části se doba vykonávání úlohy měří bez kódu, jehož rychlost provedení má být změřena. Ve druhé části již úloha probíhá včetně měřeného kódu. Na konci měření jsou tyto dvě hodnoty od sebe odečteny a výsledkem je doba provádění dané úlohy bez režie ostatního kódu, který není součástí měření (doba jeho provedení byla změřena v první části), ale je nezbytný k běhu úlohy. Každá úloha proběhne 100 000krát a na konci je vypočítána průměrná hodnota doby jejího provádění v taktech procesoru. Výsledkem každého meření je tedy počet rhealstonů, což je hodnota, kolikrát se měřená úloha stihne vykonat za dobu jedné sekundy. V úplném závěru budou všechny hodnoty rhealstonů sečteny s váhovými koeficienty 1 (tzn. žádné měření nebude upřednostňováno před jiným).

5.3.1 Task Switching Time

V prvním měření se zjišťuje doba, za kterou dojde k přepnutí mezi dvěma úlohami. Celkem se v tomto testu nachází 3 úlohy. Úloha `AppTaskStart`, která má za úkol vytvořit zbylé dvě úlohy se stejnou prioritou (`Task1` a `Task2`), mezi kterými se bude následně přepínat. Tato úloha také změří provedení testu bez přepínání mezi úlohami. Před samotným měřením je také nutné povolit Round robin plánování voláním funkce `OSSchedRoundRobinCfg()`.

Listing 5.1: AppTaskStart

```
START_CNT()

for (int i = 0; i < ITERATIONS; i++) {}
for (int j = 0; j < ITERATIONS; j++) {}

STOP_CNT()
```

Úloha `Task1` spustí časovač a poté se cyklicky vzdává procesoru ve prospěch úlohy `Task2`. Úloha `Task2` se naopak cyklicky vzdává procesoru ve prospěch úlohy `Task1`. Po provedení daného počtu iterací první úloha časovač zastaví, dojde k výpisu naměřených hodnot a test skončí.

Listing 5.2: Task1

```
START_CNT()

for (int i = 0; i < ITERATIONS; i++)
{
        OSSchedRoundRobinYield(&Task1Err);
}

STOP_CNT()
```

5.3.2 Preemption Time

V tomto testu se měří doba, za kterou dojde k preempci úlohy s nižší prioritou a následnému spuštění úlohy s vyšší prioritou. Stejně jako v předchozím testu se i zde nachází úloha s nejvyšší prioritou `AppTaskStart`, která je zodpovědná za vytvoření úlohy `Task1` a `Task2`. Úloha `Task1` má vyšší prioritu než `Task2`. Úloha `AppTaskStart` změří dobu běhu programu bez preempce, následně je tato úloha pozastavena a začne se provádět úloha `Task1`. Tato úloha spustí časovač a poté se cyklicky uspává. Po uspání úlohy `Task1` je spuštěna úloha `Task2`, která v cyklu vždy probudí úlohu `Task1` a tím pádem dojde k preempci a začne se znovu provádět úloha `Task1`. Tato úloha také po 100 000. iteraci zastaví časovač a test skončí. Implementace se velmi podobá prvnímu testu.

5.3.3 Semaphore Shuffle Time

Třetí test se zaměřuje na semafory, resp. na zpoždění mezi odemčením semaforu jednou úlohou spuštěním druhé úlohy, která na toto odemčení čekala. Opět se zde objevuje úloha `AppTaskStart` s nejvyšší prioritou, která vytváří úlohy `Task1`, `Task2`, a semafor `Semaphore`. Tyto úlohy mají stejnou prioritu. Po zahájení testu uzamkne úloha `Task1` semafor a vzdá se procesoru. Následně je spuštěna úloha `Task2`, která také potřebuje uzamknout tento semafor. Protože tento semafor je již uzamčen úlohou `Task1`, je úloha `Task2` přerušena a znovu se spustí úloha `Task1`. Tato úloha odemkne semafor, následně je přerušena a k běhu se dostane `Task2`, která uzamkne semafor, vzdá se procesoru a situace se periodicky opakuje.

5.3.4 Intertask Messaging

V tomto testu se měří jak velká časová prodleva nastane mezi odesláním zprávy jednou úlohou a přijetím této zprávy druhou úlohou. Nejprve úloha `AppTaskStart` vytvoří úlohu `Task1`, `Task2` (tato úloha má vyšší prioritu než `Task1`), frontu zpráv a změří dobu, za kterou se test vykoná bez zasílání zpráv. Následně se spustí úloha `Task2`, která čeká na příchod zprávy. Protože ale zatím žádná zpráva nepřišla, je spuštěna úloha `Task1`, která zašle úloze `Task2` zprávu. Ihned po odeslání této zprávy je úloha `Task1` pozastavena a spustí se úloha `Task2`, která zprávu přijme a poté znovu čeká na další zprávu.

5.3.5 Deadlock Break Time

V předposledním testu se zjišťuje, jak dlouho trvá jádru operačního systému vyřešit uváznutí (deadlock). Úloha `AppTaskStart` vytvoří úlohu `Task1`(nejnižší priorita), `Task2`(střední priorita), `Task3`(nejvyšší priorita), semafor `Semaphore`, uspí úlohy `Task2` a `Task3` a změří dobu provádění testu bez vzniku uváznutí. Na začátku testu běží úloha s nejnižší prioritou `Task1`. Tato úloha uzamkne semafor a probudí úlohu se střední prioritou `Task2`, ta pouze

probudí úlohu `Task3` s nejvyšší prioritou, která uspí úlohu `Task2` a pokusí se uzamknout semafor. Tento semafor je však již uzamčen úlohou s nejnižší prioritou `Task1`. Tato úloha je tedy probuzena jádrem operačního systému, ihned po svém probuzení odemkne semafor, následně je znovu přerušena a je spuštěna úloha `Task3`. Tato úloha uzamkne semafor, ihned jej odemkne a uspí se. Následuje nová iterace tohoto testu.

5.3.6 Interrupt Latency

V posledním testu je měřena doba od výskytu přerušení do provedení první instrukce obsluhy tohoto přerušení. V tomto testu figuruje pouze jedna úloha a to úloha `Task1`. Kromě implementace této úlohy bylo nutné upravit funkci `OSTimeTickHook` (os_cpu_c.c). Při volání této funkce dojde k uložení aktuální hodnoty čítače cyklů do proměnné `COUNT`, přičtení této hodnoty do celkové sumy těchto cyklů `SUM` a ke zvýšení počitadla iterací `ITERATIONS`. Dále byla upravena konfigurace jádra, konkrétně zvýšení výskytu přerušení systémového časovače na nejvyšší možnou frekvenci 1000Hz (os_cfg_app.h). Úloha `Task1` nejprve voláním funkce `INT_SYS_DisableIRQ()` vypne všechna přerušení specifická pro dané zařízení (Device specific interrupts v MK60D10.h). Poté se úloha dostane do cyklu, ve kterém pouze dokola nuluje počitadlo cyklů. Za určitou chvíli ale dojde k výskytu přerušení od systémového časovače, který na začátku své obsluhy zavolá funkci `OSTimeTickHook`, která obsahuje kód pro uložení aktuální hodnoty časovače do globální proměnné `SUM`. Tímto způsobem se čeká na 100 000 příchodů přerušení od systémového časovače a následně dochází k měření doby od příchodu přerušení po začátek obsluhy tohoto přerušení.

5.4 Thread-Metric benchmark

Thread-Metric benchmark je složen ze šesti testů (viz 5.1.2). Interrupt Handling test byl kvůli problematické implementaci vynechán. Všechny testy mají podobnou strukturu implementace. Jako první je vždy spuštěna úloha s nejvyšší prioritou `task0`. Tato úloha vypíše na výstup název testu, který se bude provádět a počet iterací tohoto testu (jedna iterace trvá 30 sekund). Poté tato úloha provede inicializaci příslušného testu, například vytvoří semafor, frontu, probudí nebo pozastaví jiné úlohy, které se budou na testu podílet. Následuje cyklus na jehož začátku se úloha `task0` uspí na 30 sekund a začne se vykonávat samotný benchmark. Po třiceti sekundách se tato úloha probudí a vypíše na výstup naměřené hodnoty z předcházející iterace testů. Poté tyto hodnoty přičte do proměnné uchovávající sumu naměřených hodnot za celý benchmark a vynuluje patřičné čítače pro běh další iterace testů. V úplném závěru je vypsána zpráva o konci benchmarku doplněná o výpis celkových naměřených hodnot.

Každá úloha, která se přímo podílí na provádění benchmarku (tedy všechny úlohy v daném testu kromě `task0`), obsahuje ve svém těle proměnnou, která slouží jako počitadlo provedení příslušné operace, která bude měřena. Před spuštěním každého testu jsou tyto proměnné vynulovány.

Listing 5.3: task0

```
void task0(void *p_arg)
{
  /* Vypis nazvu testu a poctu iteraci. */
  printf("Thread-Metric Benchmark\r\n");
  printf("Start of ... Test\r\n");
  printf("Iterations = %d, 30 seconds per iteration.\r\n", ITERATIONS);

  /* Inicializace testu.
  ...
  */

  /* Cyklus for - provadeni benchmarku. */
  for (int i = 1; i <= ITERATIONS; i++)
  {
    /* Uspani teto ulohy na 30s */
    OSTimeDlyHMSM(0, 0, 30, 0, OS_OPT_TIME_HMSM_STRICT, &err_task0);

    /* Vypis vysledku za posledni iteraci
    ...
    */

    /* Pricteni namerenych vysledku do sum celkovych vysledku.
    ...
    */

    /* Vynulovani citacu pro dalsi iteraci testu.
    ...
    */
  }

  /* Vypis konce testu, pocet iteraci, celkove namerene hodnoty
  ...
  */

  /* Ukonceni benchmarku */
  OSTaskSuspend(NULL, &err_task0);
}
```

5.4.1 Cooperative Context Switching

V testu kooperativního (nepreemptivního) přepínání kontextu se vyskytuje 5 úloh o stejné prioritě. Po spuštění první úlohy se zvýší její počitadlo o 1 a poté dojde k volání funkce `OSSchedRoundRobinYield`. Tímto voláním se úloha dobrovolně (nepreemptivně) vzdá procesoru a následně je spuštěna další úloha, která se chová úplně stejně.

5.4.2 Preemptive Context Switching

Test preemptivního přepínání kontextu se podobá testu kooperativního přepínání kontextu. I v tomto případě je test složen z 5 úloh (`task1` až `task5`), avšak tentokrát mají úlohy rozdílné priority. Na začátku testu běží úloha `task5`, která má nejnižší prioritu. Tato úloha zvýší své počitadlo spuštění o 1 a probudí úlohu `task4`. Jelikož má ale nově probuzená úloha `task4` vyšší prioritu než úloha `task5`, dojde k preempci a začne se vykonávat úloha `task4`. Úloha `task4` nejprve zvýší své počitadlo, uspí úlohu `task5` (aby byl test připraven na další

iteraci) a probudí úlohu `task3`. Situace se opakuje až do spuštění úlohy `task1`, která má nejvyšší prioritu. Tato úloha kromě zvýšení svého počitadla spuštění, probudí úlohu `task5`, která má však nejnižší prioritu a proto se ještě musí úloha `task1` sama uspat. Následně pokračuje test od začátku.

5.4.3 Message Processing

V tomto testu se nachází pouze jedna úloha `task1`, která v cyklu zasílá zprávu do fronty zpráv (voláním funkce `OSQPost()`), následně tuto zprávu přečte (`OSQPend()`) a zvýší své počitadlo.

5.4.4 Semaphore Processing

Úloha `task1` uzamkne semafor (pomocí funkce `OSSemPend()`), následně jej odemkne (funkcí `OSSemPost()`) a zvýší své počitadlo. Tato sekvence operací se cyklicky opakuje.

5.4.5 Memory Allocation

Poslední test také obsahuje pouze jednu úlohu `task1`, která si cyklicky alokuje blok paměti o velikosti 128 bytů voláním funkce `OSMemGet()`, ihned tento blok uvolní (funkcí `OSMemPut()`) a nakonec zvýší své počitadlo.

5.5 Vlastní testy

Bylo naimplementováno celkem 7 testů, ve kterých jsou měřeny doby provádění 14 služeb jádra μC/OS-III. Každý test obsahuje jednu až dvě úlohy, jejichž prostřednictvím jsou změřeny doby provádění dvou služeb, které spolu souvisí (například `OSTaskCreate()` a `OSTaskDel()`). Každý test byl proveden 100 000krát a jeho výstupem byla maximální a minimání doba provádění dané služby. Před začátkem měření byla vždy vypnuta všechna přerušení voláním funkce `INT_SYS_DisableIRQGlobal()`, aby nedošlo k ovlivnění měření.

Listing 5.4: obecná struktura testů

```
/* Vyber sluzby, ktera se bude merit */
#define SLUZBA1 0
#define SLUZBA2 1

/* Deklarace a inicializace ostatnich promennych (TCB, stack, ...)*/

void task1(void *p_arg) {
  /* Zjisteni kolik cyklu zabere rezie mereni. */

  for (;;) {

    /* Vypnuti vsech preruseni */

    for (int i = 0; i < ITERATIONS; i++) {

      if (SLUZBA1)
        START_CNT()

      /* Volani prvni sluzby */

      if (SLUZBA1)
        STOP_CNT();


      if (SLUZBA2)
          START_CNT();

      /* Volani druhe sluzby */

      if (SLUZBA2)
        STOP_CNT()

      /* Ulozeni vysledku. */
    }

    /* Povoleni preruseni */

    /* Vypis vysledku. */

    /* Konec testu. */
  }
}
```

5.5.1 Vytvoření a odstranění úlohy

V prvním testu bylo v úloze `task1` změřeno, jak dlouho trvá vytvoření (`OSTaskCreate()`) a odstranění (`OSTaskDel()`) úlohy. Úloha `task1` nejprve vytvoří úlohu `task2`, kterou následně ihned odstraní. Úloha `task2` se tedy nikdy nezačne provádět. Podle nastavení maker `CREATE` a `DEL` je rozhodnuto o tom, která z těchto služeb bude měřena v rámci testu.

5.5.2 Probuzení a uspání úlohy

Na začátku testu je vytvořena úloha `task1` a `task2`. Úkolem úlohy `task1` je, změřit jak dlouho trvá úlohu `task2` uspat (`OSTaskSuspend()`) a znovu probudit (`OSTaskResume()`). Úloha `task2` se, stejně jako v předchozím testu, nikdy nezačne provádět. Ke zvolení služby, která se bude měřit slouží makra `SUSPEND` a `RESUME`.

5.5.3 Uzamykání a odemykání plánovače

Ve třetím testu byla vytvořena pouze úloha `task1`. Tato úloha měří dobu, jak dlouho trvá uzamknout plánovač (`OSSchedLock()`) a znovu jej odemknout (`OSSchedUnlock`). K výběru měřené služby se v testu nachází makra `LOCK` a `UNLOCK`.

5.5.4 Alokace a dealokace paměti

V tomto testu je vytvořena úloha `task1`, která si naalokuje 40 bajtů místa v paměti (`OSMemGet()`) a následně tuto paměť zase uvolní (`OSMemPut()`). Na základě nastavení maker `GET` a `PUT` je rozhodnuto o tom, která služba bude měřena.

5.5.5 Uzamykání a odemykání semaforu

V pátem testu je prostřednictvím úlohy `task1` měřena doba, jakou trvá uzamknout (`OSSemPend()`) a odemknout (`OSSemPost()`) semafor. Makra `PEND` a `POST` slouží k výběru měřené služby.

5.5.6 Uzamykání a odemykání mutexu

Předposlední test se velmi podobá přechozímu testu. Úloha `task1` uzamyká `OSMutexPend()` a odemyká `OSMutexPost()` mutex (binární semafor) a měří doby těchto operací na základě nastavených maker, která jsou shodná jako v předchozím testu.

5.5.7 Vstup a výstup z kritické sekce

V posledním testu byla úlohou `task1` změřena doba vstupu do kritické sekce (`OS_CRITICAL_ENTER()`) a výstupu z kritické sekce (`OS_CRITICAL_EXIT()`). Stejně jako ve všech předchozích testech se i zde nachází 2 makra, sloužící k výběru měřené služby, konkrétně `CRIT_ENTER` a `CRIT_EXIT`.

Kapitola 6

Shrnutí a interpretace výsledků

V této kapitole budou prezentovány dosažené výsledky z měření popsaných v předchozí kapitole. Výsledky budou shrnuty v tabulkách a grafech doplněných o komentář. Naměřené hodnoty byly nejprve zpracovány programem Microsoft Excel. Následně byly takto zpracované hodnoty přepsány do tabulek vytvořených přímo v LaTeXu. Hodnoty byly také pro přehlednost zaneseny do grafů pomocí programu gnuplot. Tyto grafy byly následně vloženy do této práce v podobě obrázků. Při provádění všech měření byl nastaven takt jádra procesoru ARM Cortex-M4 na 99 998 720Hz.

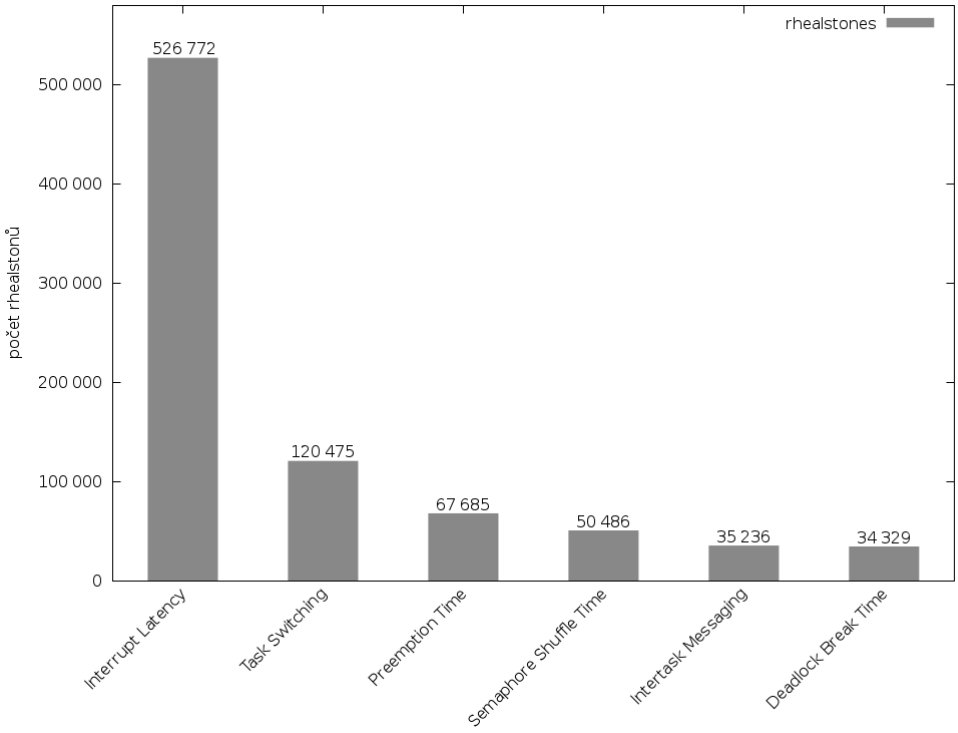
6.1 Rhealstone

Podstata Rhealstone benchmarku byla popsána v kapitole 5 v sekci 5.1.1 a implementace v sekci 5.3. V tabulce 6.1 jsou výsledky všech šesti měřených úloh seřazených od nejdéle trvající úlohy po úlohu trvající nejkratší dobu. Každá úloha byla měřena 100 000krát a výsledek je dán průměrem všech měření. U každé úlohy byla změřena průměrná doba provádění v hodinových cyklech procesoru, v mikrosekundách a v Rhealstonech, což je hodnota reprezentující kolikrát se daný test stihne za jednu sekundu. Ve skutečnosti by měla být výsledkem tohoto benchmarku pouze jedna hodnota daná váhovým součtem všech polí ve sloupci „Rhealstones“. Jedním z problému je, jak určit dané váhové koeficienty. Neexistují žádná přesná pravidla, jakých hodnot by měly koeficienty nabývat. V tomto případě byly všechny koeficienty stanoveny na hodnotu 1 a nedošlo tedy jejich vlivem ke zkreslení celkového výsledku.

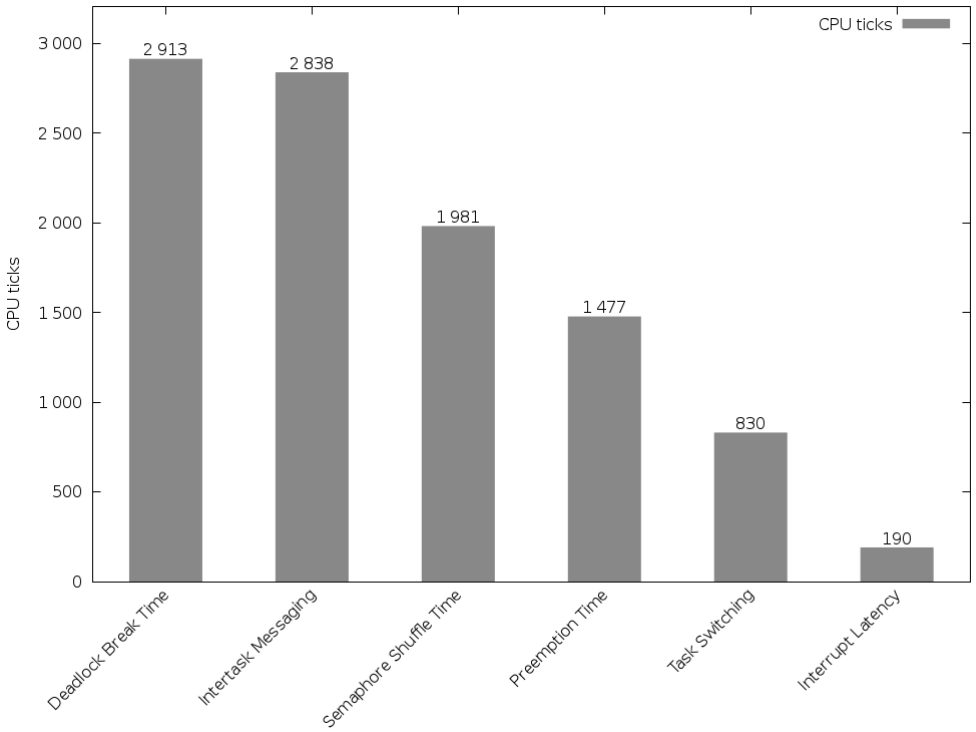
měřená úloha	**CPU ticks**	**μs**	**Rhealstones**
Deadlock Break Time	2 913	29,130	34 329
Intertask Messaging	2 838	28,380	35 236
Semaphore Shuffle Time	1 981	19,807	50 486
Preemption Time	1 477	14,774	67 685
Task Switching	830	8,300	120 475
Interrupt Latency	190	1,900	526 772
suma			**834 983**

Tabulka 6.1: Rhealstone

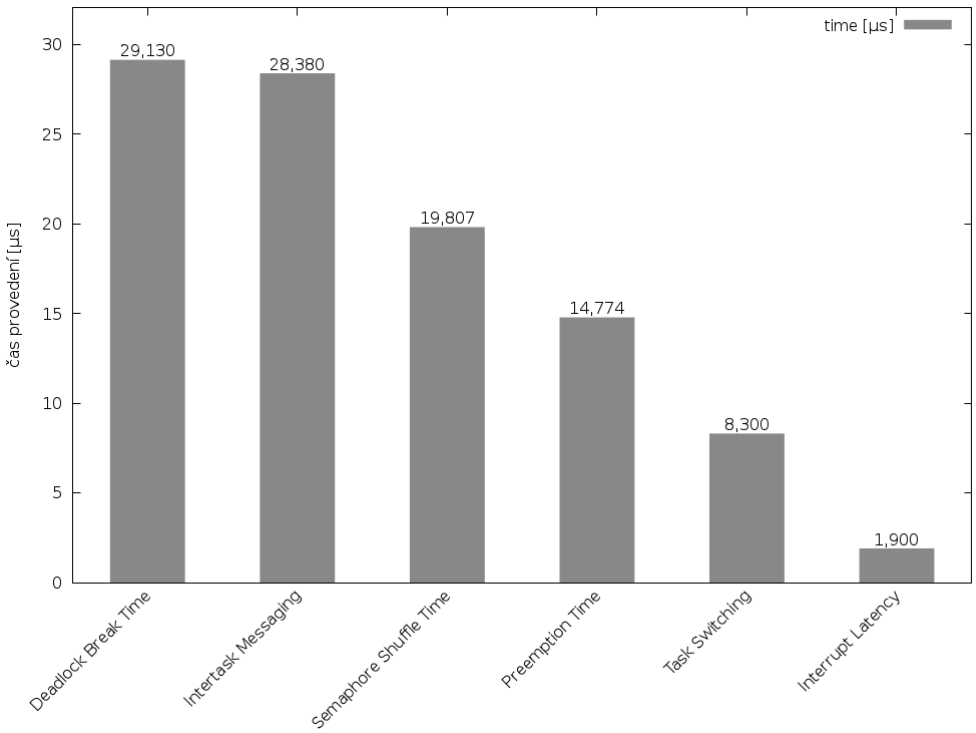
Po dosazení naměřených hodnot do rovnice 5.1 je výsledkem tohoto benchmarku hodnota 834 983 rhealstonů. Jak je vidět z tabulky, zcela nejrychlejší částí tohoto benchmarku je test Interrupt Latency, který má hodnotu 526 772 rhealstonů. Tento test výrazným způsobem ovlivňuje celkovou sumu rhealstonů a z tohoto důvodu jsou v tabulce ponechány i dílčí naměřené hodnoty benchmarku v taktech procesoru a mikrosekundách.



Obrázek 6.1: Rhealstone – rhealstones



Obrázek 6.2: Rhealstone – hodinové takty



Obrázek 6.3: Rhealstone – čas

6.2 Thread-Metric

Thread-Metric benchmark je složen ze šesti úloh, které byly popsány v kapitole 5 v sekci 5.1.2. Popis implementace se nachází v sekci 5.4. Výsledkem tohoto benchmarku jsou hodnoty, kolikrát se příslušný test stihne vykonat za 30 sekund, tedy čím vyšší číslo, tím lepší výsledek. Výsledky byly seřazeny do tabulky sestupně od nejlepšího k nejhoršímu. Interrupt Handling test nebyl z důvodu náročné implementace naprogramován a v tabulce výsledků tedy chybí.

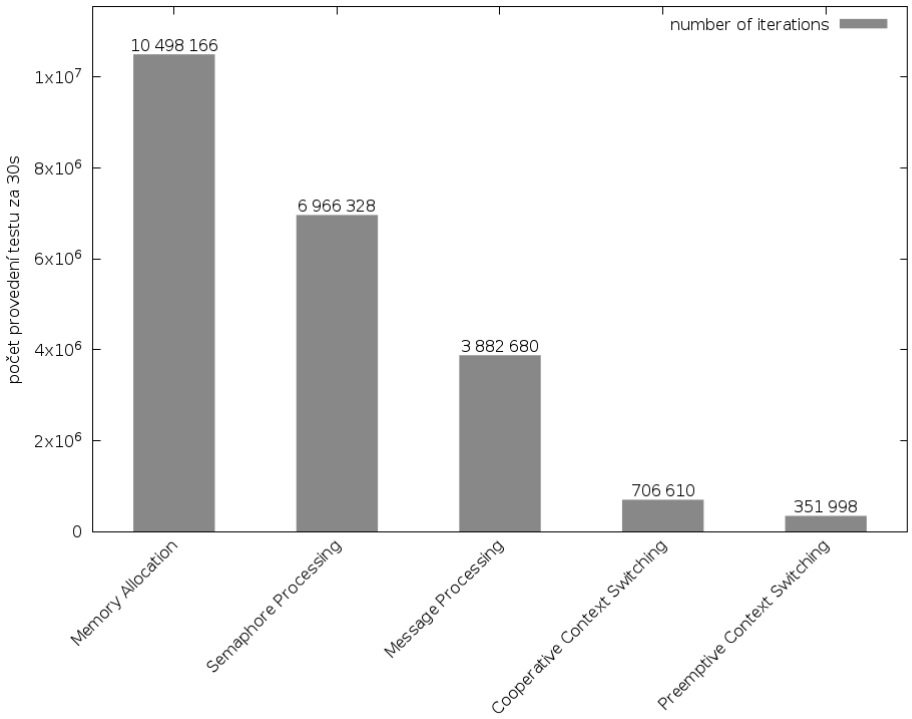
měřená úloha	**počet cyklů**
Memory Allocation	10 498 166
Semaphore Processing	6 966 328
Message Processing	3 882 680
Cooperative Context Switching	706 610
Preemptive Context Switching	351 998

Tabulka 6.2: Thread-Metric

Jak lze vidět, nejvícekrát za 30 sekund se provedl test alokace a dealokace bloku paměti o velikosti 128 bytů. V operačním systému μC/OS-III si lze vytvořit alokovatelný úsek paměti voláním funkce `OSMemCreate()`. V argumentech této funkce je specifikována adresa začátku této paměti, počet alokovatelných sekcí a velikost jedné sekce. Při alokaci paměti voláním funkce `OSMemGet()` je vrácen ukazatel na jednu sekci z připraveného úseku paměti. Dealokace, resp. navrácení bloku paměti, voláním funkce `OSMemPut()` se odehrává taktéž přes ukazatel. Z této skutečnosti plyne, že velikost alokovaného bloku nemá vliv na výsledek tohoto testu v rámci operačního systému μC/OS-III. Velmi podobná situace nastává u Message Processing testu, ve kterém dochází k předávání zpráv skrze ukazatele a tím pádem velikost zprávy neovlivňuje potřebnou dobu k zaslání zprávy jednou úlohou a jejímu přijetí druhou úlohou.

Nejnáročnějšími z těchto pěti testů jsou Cooperative a Preemptive Context Switching testy. Z výsledků je patrné, že pokud se úloha sama vzdá procesoru, dojde k přepnutí na další úlohu za téměř poloviční čas než v případě, kdy musí být jedna úloha přerušena operačním systémem a místo ní je spuštěna jiná úloha.

Výsledky Thread-Metric benchmarku byly také vyneseny pro přehled do následujícího grafu.



Obrázek 6.4: Thread – Metric

6.3 Vlastní testy

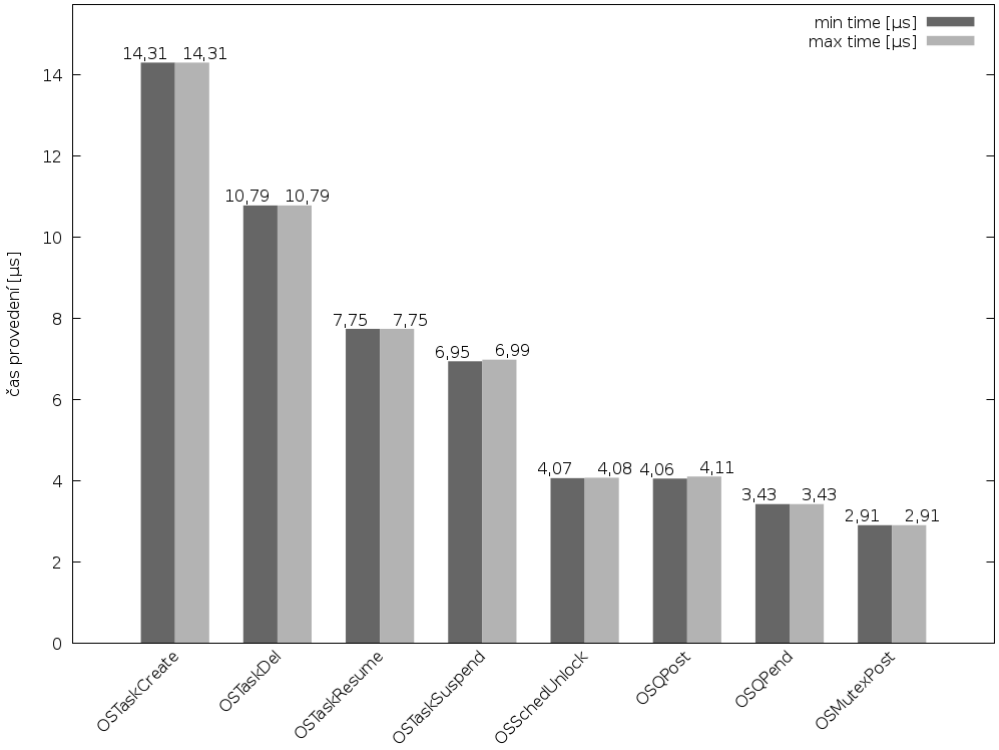
V samotném závěru byla naimplementována sada vlastních testů, které nespadají pod žádný benchmark. Popis implementace se nachází v sekci 5.5. Výsledky těchto testů se nachází v tabulce 6.3 a byly seřazeny sestupně od nejdéle trvající služby. Tabulka obsahuje čtyři údaje ke každé službě - nejkratší a nejdelší dobu provádění v taktech procesoru a v mikrosekundách.

	CPU ticks		čas [μs]	
měřená úloha	**min**	**max**	**min**	**max**
OSTaskCreate	1 431	1 431	14,31	14,31
OSTaskDel	1 079	1 079	10,79	10,79
OSTaskResume	775	775	7,75	7,75
OSTaskSuspend	695	699	6,95	6,99
OSSchedUnlock	407	408	4,07	4,08
OSQPost	406	411	4,06	4,11
OSQPend	343	343	3,43	3,43
OSMutexPost	291	291	2,91	2,91
OSSemPost	239	239	2,39	2,39
OSMutexPend	232	236	2,32	2,36
OSSemPend	195	195	1,95	1,95
OSSchedLock	159	160	1,59	1,60
OSMemGet	149	160	1,49	1,60
OSMemPut	136	136	1,36	1,36
OS_CRITICAL_EXIT	27	47	0,27	0,47
OS_CRITICAL_ENTER	17	20	0,17	0,20

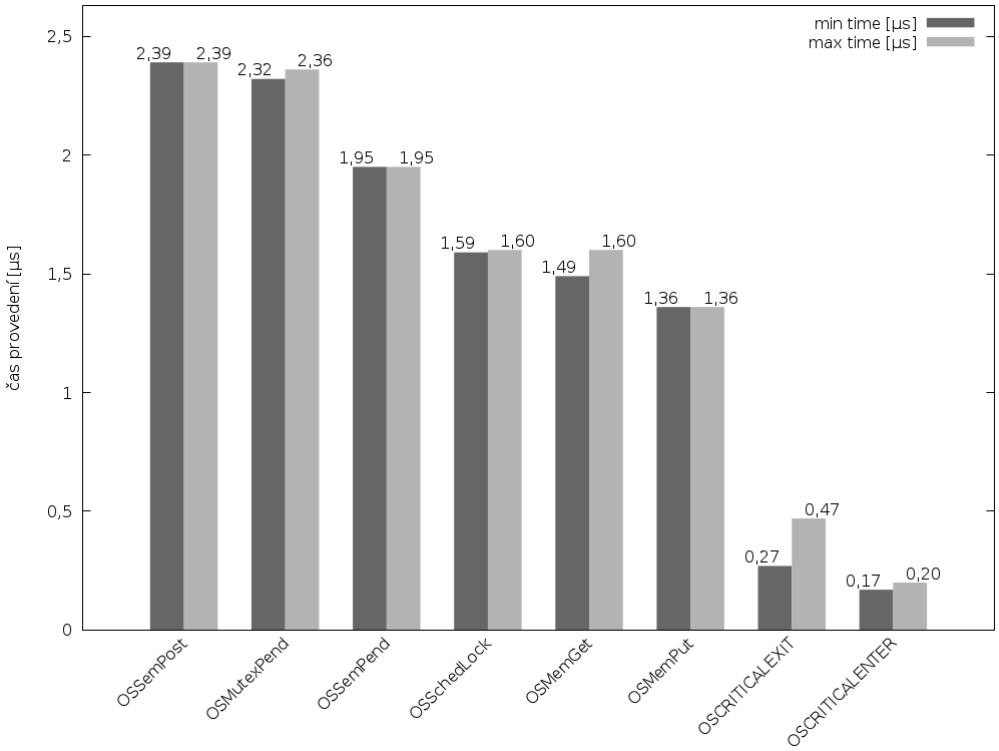
Tabulka 6.3: Vlastní testy

Z výsledků lze vidět, že rozdíl mezi maximální a minimální dobou provádění jedné služby je velmi malý až nulový. Z hlediska hodnocení vlastností operačních systémů reálného času je tato skutečnost velmi významná. Měření tedy dosáhlo vysoké preciznosti, nicméně je problematické určit pravdivost z důvodu neexistence referenčních výsledků daných benchmarků na platformě FITkit 3. Dále lze vidět, že i rozdíl v dobách provádění dvou souvisejících služeb je také relativně malý. Například u služeb pro alokaci (`OSMemGet()`) a dealokaci (`OSMemPut()`) paměti je maximální rozdíl 80 taktů neboli 0,8 μs. Největší rozdíl v době provádění dvou souvisejících služeb je možné pozorovat mezi uzamykáním (`OSSchedLock()`) a odemykáním (`OSSchedUnlock()`) plánovače. Odemknutí plánovače trvá 2,55 krát delší dobu než jeho uzamknutí. Takto velký časový rozdíl je způsoben voláním plánovače ihned po jeho odemknutí (`OSSchedUnlock()`). Stejná situace s voláním plánovače nastává po provedení funkcí `OSSemPost()` a `OSMutexPost()`. Avšak u těchto dvou funkcí lze specifikovat při volání volbu `OS_OPT_POST_NO_SCHED`, která zabrání volání plánovače. Tuto volbu nelze specifikovat u funkce `OSSchedUnlock()` a tím pádem je do výsledku měření započítán i čas potřebný k provedení plánování.

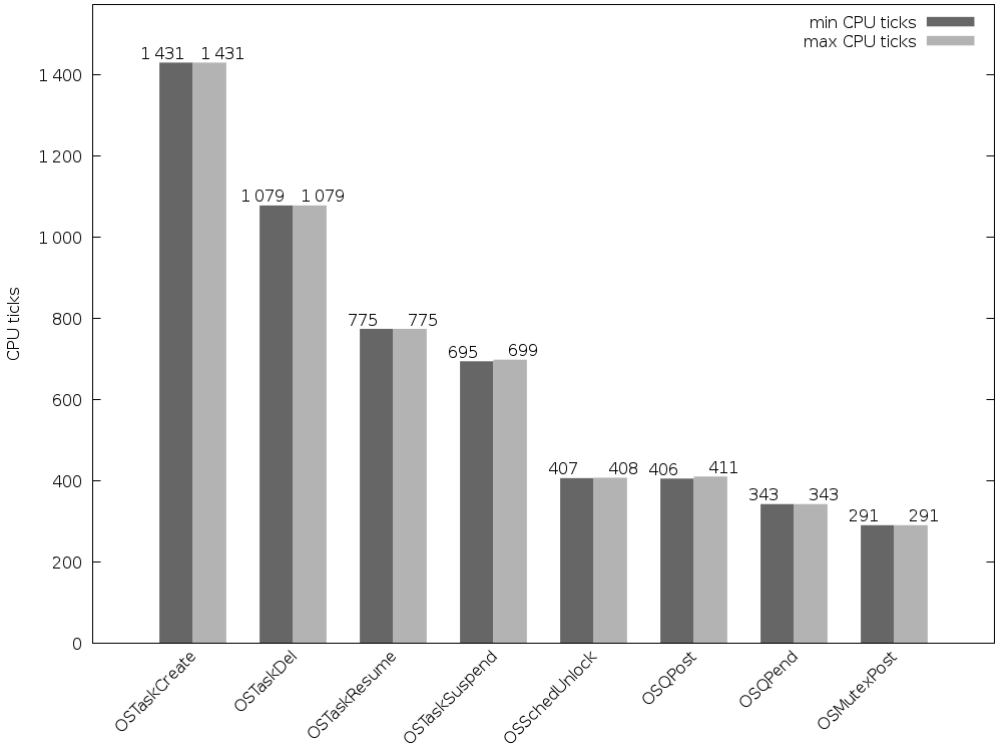
Hodnoty z tabulky byly také vyneseny do čtyř grafů z důvodů větší přehlednosti. Tyto služby jsou seřazeny sestupně od nejdéle trvající služby. V prvních dvou grafech je na ose y čas v mikrosekundách, ve třetím a čtvrtém grafu je na ose y vynesen počet hodinových taktů procesoru.



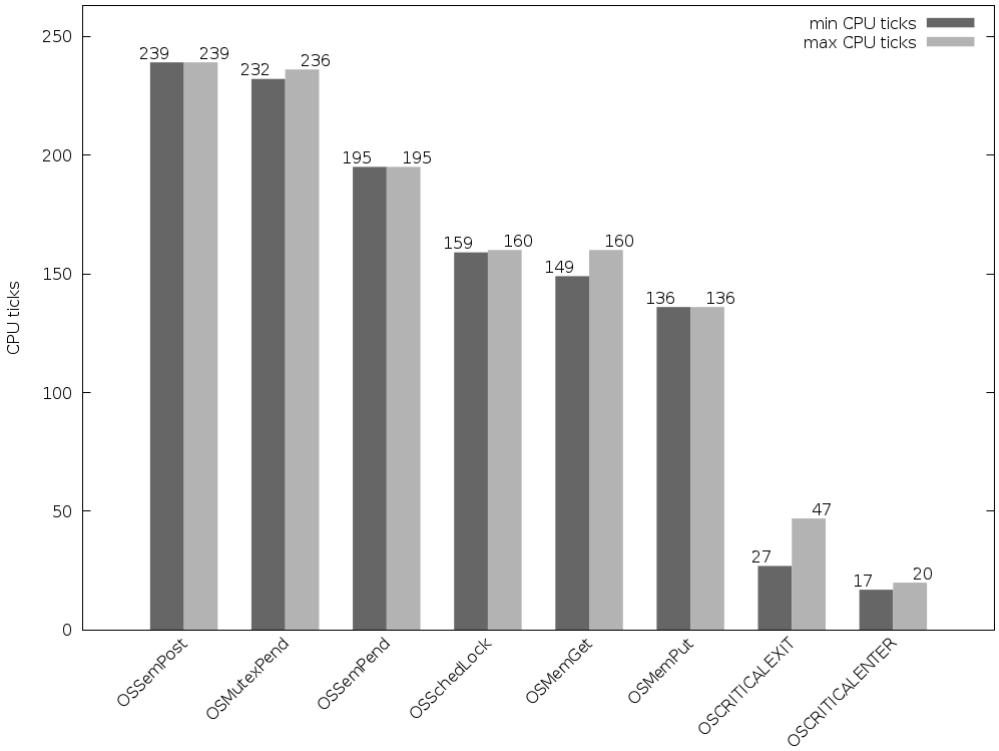
Obrázek 6.5: Vlastní testy – čas (1/2)



Obrázek 6.6: Vlastní testy – čas (2/2)



Obrázek 6.7: Vlastní testy – hodinové takty (1/2)



Obrázek 6.8: Vlastní testy – hodinové takty (2/2)

Kapitola 7

Závěr

Cílem této práce bylo prostudovat architekturu platformy FITkit 3, teorii okolo operačních systémů reálného času, operační systém μC/OS-III, navrhnout sadu aplikací, které by ověřily real-time vlastnosti a výkonnost tohoto operačního systému a naměřené výsledky zhodnotit.

Operační systémy reálného času jsou zpravidla součástí vestavěných systémů okolo nás. Vyskytují se například v tiskárnách, směrovačích, přepínačích, automobilech (ABS), ale také v letadlech čí jaderných elektrárnách. Klíčovými vlastnostmi, které musí operační systém reálného času v rámci vestavěného systému zajistit jsou - správnost a včasnost reakce na příchozí podněty.

Pro ověření funkčnosti služeb operačního systému μC/OS-III byly naimplementovány dva standardizované benchmarky (Rhealstone a Thread-Metric) doplněné o sadu vlastních testů a bylo provedeno otestování na platformě FITkit 3. Nejvyšší skóre z Rhealstone benchmarku bylo naměřeno u Interrupt Latency testu, který sleduje dobu, za kterou se začne provádět obsluha příchozího přerušení, což je velmi důležitá vlastnost systémů, které mají co nejrychleji reagovat na podněty z okolí. Z výsledků vlastních testů plyne skutečnost, že rozdíl mezi maximální a minimální dobou provádění jedné služby je minimální až nulový a dokonce i rozdíl mezi dobami provádění dvou souvisejících služeb je velmi malý. Takové výsledky jsou pro každý operační systém reálného času velmi příznivé, protože na jejich základě lze velmi jednoduše předvídat chování takového operačního systému za různých podmínek a je tedy možné jej využít i pro řízení časově kritických činností v rámci hard real-time systémů.

Při hodnocení z pohledu přívětivosti pro uživatele je vhodné poukázat na velmi čistý zdrojový kód operačního systému μC/OS-III, dále na sjednocené API, které kromě jednotného pojmenování API funkcí, také dodržuje konvence týkající se předávání parametrů.

Vhodným rozšířením této práce by mohlo být provedení zmíněných benchmarků a testů na jiných mikrokontrolérech. Jedním z nich by mohl být mikrokontrolér, jehož procesor by nebyl tak výkonný jako ARM Cortex-M4 a nepodporoval by instrukce umožňující optimalizaci běhu plánovače. Další možností by mohlo být otestování operačního systému μC/OS-III na některém procesoru z rodiny ARM Cortex-R. Procesory této řady byly navrženy jako real-time procesory právě pro nasazení v hard real-time systémech.

Literatura

[1] Gumzej, R.: *Real-time Systems' Quality of Service*. Springer-Verlag London, 2010, ISBN 978-1-4471-5758-8.

[2] Labrosse, J. J.: *uC/OS-III: The Real-Time Kernel*. Micrium Press, 2009, ISBN 978-0-9823375-2-3.

[3] Strnadel, J.: *Real-time operační systémy*. FIT VUT v Brně, 2006.

[4] Vašíček, Z.: Hardware. FIT VUT v Brně, 2006 [cit. 2016-12-06], [Online; navštíveno 6.12.2016].
URL http://merlin.fit.vutbr.cz/FITkit/uvod.html

[5] WWW stránky: *ARM Information Center*. [Online; navštíveno 8.2.2017].
URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0439b/BABJFFGJ.html

[6] WWW stránky: Micrium Doc. [Online; navštíveno 17.4.2017].
URL https://doc.micrium.com/

[7] WWW stránky: Thread-Metric Benchmark Suite. [Online; navštíveno 11.2.2017].
URL http://rtos.com/PDFs/MeasuringRTOSPerformance.pdf

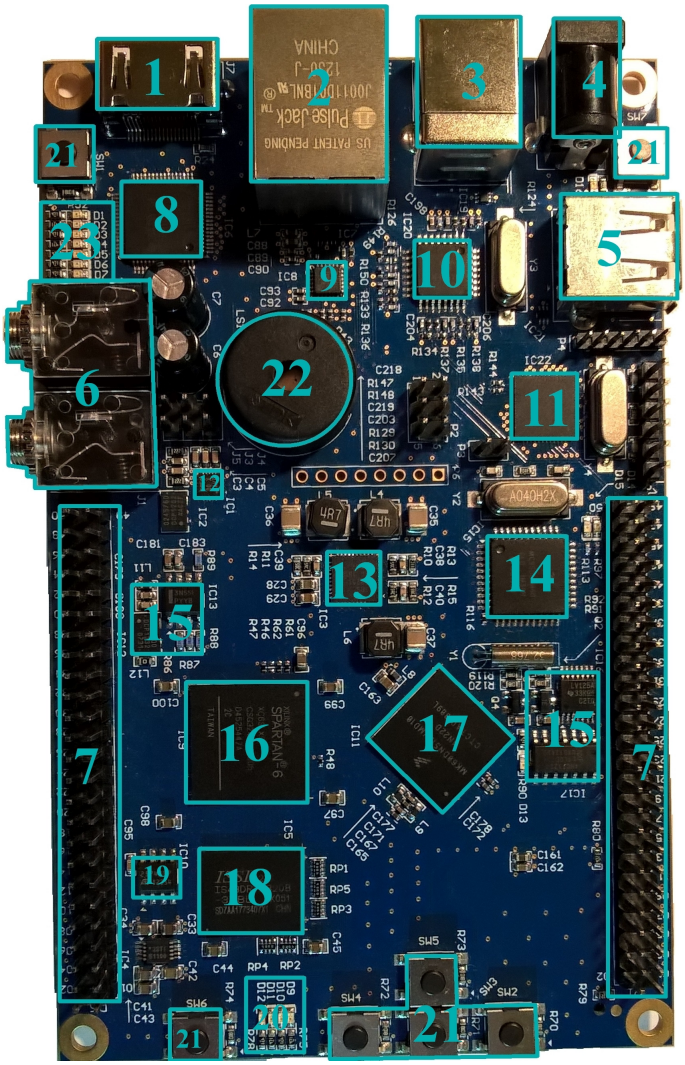
[8] WWW stránky: Minerva kit. FIT VUT v Brně, 2012 [cit. 2016-12-04], [Online; navštíveno 4.12.2016].
URL http://minerva.php5.cz/minerva/popis-kitu.php

Přílohy

Příloha A

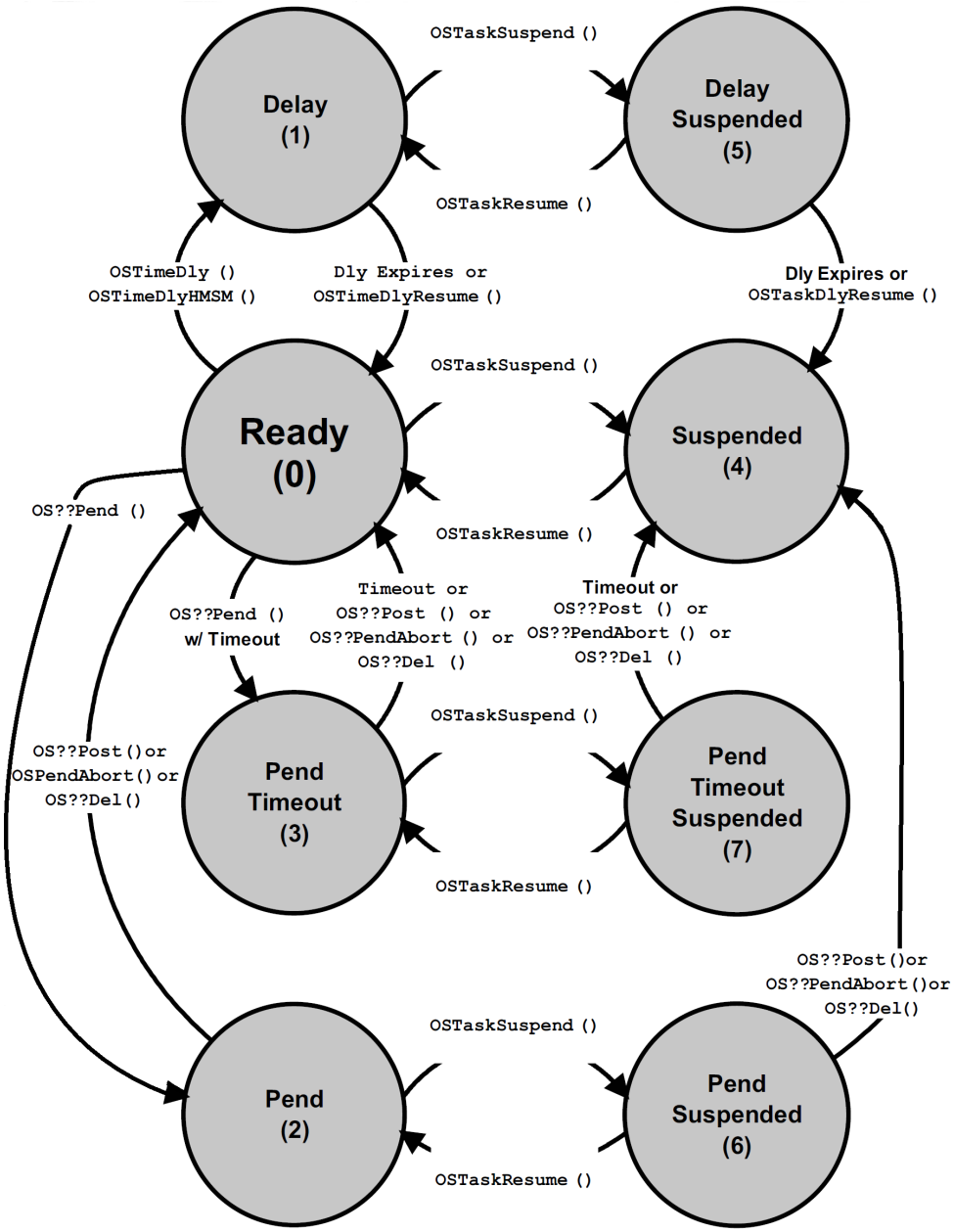
Schémata a obrázky

A.1 Schéma platformy FITkit 3



Obrázek A.1: Schéma platformy FITkit 3

A.2 Konečný automat stavů a přechodů úlohy



Obrázek A.2: Konečný automat stavů a přechodů úlohy[1]

[1] Zdroj: [2] str. 112

Příloha B

Obsah DVD

Přiložené DVD obsahuje:

- technickou zprávu ve formátu pdf
- zdrojové soubory této technické zprávy
- zdrojové soubory k naimplementované sadě aplikací
- soubory s naměřenými výsledky ve formátu xlsx
- soubor README.txt obsahující dodatečné informace a popis spuštění aplikací