VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

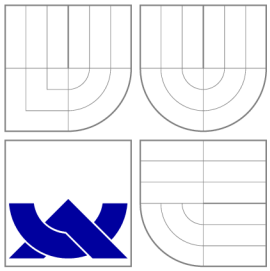
POROVNÁNÍ RT VLASTNOSTÍ 8-BITOVÝCH A 32-BITOVÝCH IMPLEMENTACÍ JÁDRA UC/OS-II

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

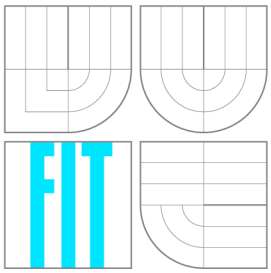
AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ ŠUBR

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

POROVNÁNÍ RT VLASTNOSTÍ 8-BITOVÝCH A 32-BITOVÝCH IMPLEMENTACÍ JÁDRA UC/OS-II

COMPARING RT PROPERTIES OF 8-BIT AND 32-BIT IMPLEMENTATIONS OF THE UC/OS-II KERNEL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ ŠUBR

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JOSEF STRNADEL, Ph.D.

BRNO 2013

Abstrakt

Tato práce se zabývá testováním vlastností systému μC/OS-II na odlišných architekturách mikrokontolérů. Popisuje jádro uC/OS-II a možnosti jeho testování různými sadami testů. Vybrané testy jsou implementovány a jsou porovnávány vlastnosti mikrokontrolerů rozdílných architektur.

Abstract

This thesis concerns of benchmarking μC/OS-II systems on different microcontroller architectures. The thesis describes COS-II microcontroller core and possible series of benchmark tests which can be used. Selected tests are implemented and measured properties of microcontrollers with different architecture are compared.

Klíčová slova

Testování, operační systémy reálného času, RTOS, uC/OS-II, ucos, Freescale Flexis, Thread-Metric.

Keywords

Benchmarking, Real-time operating system, RTOS, uC/OS-II, ucos, Freescale Flexis, thread metric.

Citace

Jiří Šubr: Porovnání RT vlastností 8-bitových a 32-bitových implementací jádra uC/OS-II, diplomová práce, Brno, FIT VUT v Brně, 2013

Porovnání RT vlastností 8-bitových a 32-bitových implementací jádra uC/OS-II

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Josefa Strnadela, Ph.D.

.....
Jiří Šubr
22. května 2013

Poděkování

Rád bych poděkoval panu Ing. Josefu Strnadelovi, Ph.D. za ochotu a odbornou pomoc při vytváření této práce.

Obsah

1 Úvod **3**

2 Real-time operační systémy **4**
2.1 RT systémy . . . 4
2.2 Základní pojmy a definice . . . 4
2.3 Rozdělení RT systémů . . . 5
2.4 RTOS jádro . . . 6
2.4.1 Víceúlohové zpracování . . . 6
2.4.2 Preemptivita . . . 6
2.4.3 Komunikační a synchronizační prostředky . . . 6
2.4.4 Správa paměti . . . 7
2.4.5 Plánovač . . . 8
2.4.6 Výběr jádra . . . 9

3 μC/OS-II **10**
3.1 Struktura jádra . . . 11
3.1.1 Kritická sekce . . . 11
3.1.2 Úlohy . . . 11
3.1.3 Task Control Blocks (TCB) . . . 12
3.1.4 Plánování úloh . . . 13
3.1.5 Změna kontextu . . . 13
3.1.6 Systémové úlohy . . . 13
3.2 Komunikační a synchronizační služby . . . 13
3.2.1 Semafory . . . 14
3.2.2 Mutexy . . . 14
3.2.3 Poštovní schránky . . . 14
3.2.4 Fronty zpráv . . . 14
3.3 Hook funkce . . . 14
3.4 Portace . . . 15
3.4.1 Soubor `os_cpu.h` . . . 15
3.4.2 Soubor `os_cpu_c.c` . . . 16
3.4.3 Soubor `os_cpu_a.asm` . . . 17

4 Platforma Freescale Flexis **20**
4.1 Řada QE128 . . . 20
4.2 8-bitový mikrokontrolér MC9S0QE128 . . . 20
4.2.1 Programovací model . . . 21
4.3 32-bitový mikrokontrolér MCF51QE128 . . . 21

4.3.1 Programovací model . . . 22
4.4 Programování a přenositelnost aplikací . . . 22
4.5 Portace . . . 22
4.5.1 MCF51QE128 . . . 22
4.5.2 MC9S08QE128 . . . 23

5 Metody pro testování vlastností (benchmarking) RTOS **26**
5.1 Testovací metody . . . 27
5.1.1 Rhealstone benchmark . . . 27
5.1.2 Thread-Metric . . . 29
5.1.3 SSC benchmark . . . 31
5.2 Testování robustnosti RTOS . . . 32
5.2.1 CRASHME . . . 32
5.2.2 Fuzz . . . 32
5.2.3 Ballista . . . 32

6 Sada implementovaných testovacích úloh **33**
6.1 Způsob měření . . . 33
6.1.1 Doba trvání . . . 33
6.1.2 Čítač událostí . . . 34
6.2 Implementace . . . 34
6.2.1 Nízkoúrovňové testy . . . 34
6.2.2 Thread-Metric benchmark . . . 35
6.3 Logování výsledků . . . 35

7 Vyhodnocení testů **36**
7.1 Nízkoúrovňové testy . . . 36
7.2 Thread-Metric . . . 36
7.3 Maximální frekvence systémových hodin . . . 37

8 Závěr **38**

A Obsah CD **40**

Kapitola 1

Úvod

Dnes už pomalu každé elektronické zařízení počínaje mikrovlnou troubou, přes televizní přijímač až po mobilní telefon obsahuje mikroprocesor, na kterém běží více či méně složitý operační systém. Například parkovací asistent automobilu. V náraznících jsou umístěny senzory, které měří vzdálenost automobilu od překážky. Řídící jednotka vhodným způsobem reaguje, například zabrzděním auta před překážkou, na vstupní podněty od senzorů. Doba trvání takovéto odezvy jistě nemůže být nekonečně dlouhá a musí mít definovánu maximální délku zpoždění. Na tomto příkladu je ukázán typický real-time systém, který má shora ohraničenou dobu odezvy reakce na vstupní podněty, přičemž se předpokládá správnost takovéto reakce.

Cílem této diplomové práce je navrhnout a implementovat sadu testů, která otestuje chování systému μC/OS-II na dvou odlišných hardwarových architekturách mikrokontrolérů a tyto výsledky následně, co možná nejobjektivněji, vyhodnoti.

V následující kapitole 2 jsou vysvětleny základní pojmy týkající se oblasti real-time operačních systémů. Obsahuje definici real-time systému, popisuje jejich nejdůležitější vlastnosti jako je determinismus, včasnost odezvy a další. Nachází se zde také rozdělení real-time systémů podle požadavků kladených na splnění časových mezí. Tato kapitola obsahuje také popis jádra real-time operačního systému.

Následuje kapitola 3, která se podrobněji zabývá jádrem RTOS μC/OS-II. Je zde uvedena charakteristika systému a popsány služby jádra především jeho komunikační a synchronizační prostředky. Dále jsou zde uvedeny kroky pro přenos tohoto systému na jinou procesorovou architekturu.

Kapitola 4 je zaměřena na platformu Freescale Flexis a na způsoby jejího programování. Dále jsou zde detailněji popsány 8-bitové a 32-bitové architektury mikroprocesorů této platformy. Jsou zde popsány kroky, které je nutné udělat, aby bylo možné spustit μC/OS-II na obou verzích těchto mikrokontrolérů.

V kapitole 5 jsou popsány vybrané sady testů používané k porovnávání jader RTOS. Detailněji jsou popsány testy Rhealstone 5.1.1 a Thread-Metric 5.1.2.

Výběr a implementace sady testů obsahuje kapitola 6. Obsahem kapitoly jsou použité metriky jednotlivých testů. Vyhodnocení výsledků testů jsou shrnuty v kapitole 7.

Kapitola 2

Real-time operační systémy

V této kapitole se seznámíme se základní terminologií používanou v oblasti real-time (RT) operačních systémů, budou zde popsány základní vlastnosti real-time operačních systémů (RTOS) a nejpoužívanější dělení RT systémů. V poslední části kapitoly je uveden základní popis struktury jádra RTOS.

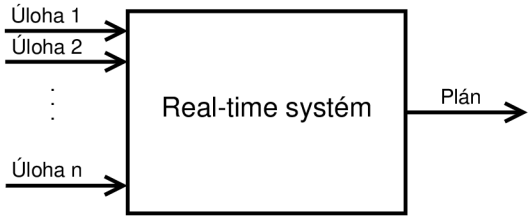
2.1 RT systémy

Systémy pracující v reálném čase jinak také nazývané real-time (RT) systémy jsou reaktivní systémy, které musí být schopny správně reagovat na asynchronní události přicházející z okolního prostředí. Správná reakce je dána nejen správností produkovaných výstupů, ale i včasností poskytnuté odezvy. Tradiční způsob modelování RT systémů je ukázán na obrázku 2.1. Vstupem systému je množina úloh tvořící uživatelskou aplikaci a reagující na vstupní podněty. Výstupem systému je plán/rozvržení těchto úloh. To znamená, že RT systém plánuje spuštění úloh.

2.2 Základní pojmy a definice

V následujícím výčtu budou uvedeny hlavní pojmy a definice týkající se oblasti RT systémů. Tyto definice jsou převzaty z [13].

- **RT systém** je systém, který musí v explicitně stanovených mezích splňovat omezení kladená na dobu své odezvy na vstupní podněty nebo riskovat vážné důsledky plynoucí z jejich nesplnění.
- Jelikož reakce RT systému na vstupní podněty není okamžitá, musíme počítat s určitým zpožděním. Doba od výskytu vstupních podnětů na vstupech systému do prove-



Obrázek 2.1: Logický model RT systému.

dení požadované odezvy, a to včetně výskytu daných hodnot na všech požadovaných výstupech, se nazývá **doba odezvy systému**.

- K **selhání systému** dojde právě tehdy, když systém nedodrží některé z navrhovaných omezení stanovených formální specifikací systému.
- Dalším pojmem je **včasnost** odezvy RT systému. Tento důležitý parametr vyplývá z omezení týkající se správnosti výše uvedené odezvy RT systému. Některé systémy vyžadují schopnost reakce na vstupní podněty v jednotkách mikrosekund, jiným systémům stačí jednotky minut. Aby byl nějaký systém považován za RT systém stačí, aby dodržel časové meze kladené na dobu jeho odezvy.
- Systém je **deterministický**, pokud pro libovolnou kombinaci množiny vstupů a libovolný stav systému lze určit následující stav systému a množinu výstupních odezev. Pokud jsou u takového systému známy i doby všech jeho odezev, pak tento systém projevuje známky *temporálního (časového) determinismu*. Temporální determinismus je významnou vlastností RT systému.
- **Činitel zatížení** je definován jako poměr mezi časem stráveným vykonáváním užitečné práce v rámci jedné periody a délkou periody. Pomocí této jednoduché metriky lze zjistit důležité informace o vlastnostech systému. Příliš nízká hodnota činitele zatížení ukazuje na použití příliš výkonného procesoru, naopak hodnota blížící se 1 může vést k nedodržení časových mezí odezev systému.
- Pod pojmem **robustnost** si můžeme představit několik různých věcí. Například můžeme toto slovo nahradit pojmy jako obrovský nebo těžký. V oblasti RT systémů je tento pojem používán v jiném smyslu. Robustnost systému je možno definovat jako odolnost systému při změně vnitřních nebo vnějších okolností nebo také jako snahu o co nejlepší zvládnutí kritických situací. Robustní systém musí být připraven na případné poruchy jeho částí. Příkladem může být raketa vypálená z letadla, které se v průběhu letu poškodí její letové prvky. Řídící systém je schopen detekovat poškození rakety a upravit její naváděcí systém tak, aby i přesto byla schopna zasáhnout svůj cíl. Typickou oblastí využití robustních systémů je právě vojenství [10].
- Vznik **události** v systému je zapříčiněn reakcí na podnět ze sledovaného prostředí. Reakce na podnět způsobí změnu toku řízení programu. Událost je definována jako změna toku řízení programu. V systému může nastat několik možných typů událostí. Základní dělení těchto událostí je založeno na předvídatelnosti vzniku události. Výskyt *synchronní* události lze předem předvídat (podmíněné skoky, výskyt vnitřního přerušení, atd.). Události, které nastávají v předem nepředvídatelných okamžicích jsou nazývány *asynchronními* událostmi. Zdrojem asynchronních událostí jsou obvykle podněty přicházející z okolí řízeného systému.

2.3 Rozdělení RT systémů

Nejčastěji se RT systémy rozdělují podle toho, jaké následky plynou z nedodržení časových odezev systému. Je určitě rozdíl v tom, když rezervační systém na letišti odmítá několik minut vydat cestujícímu jízdenku a systémem, který řídí chlazení jaderného reaktoru a při překročení určité teploty nezačne chladit palivové tyče. V prvním případě dojde maximálně k tomu, že zákazník zmešká odlet letadla a musí čekat na další, případně zvolit jiný způsob

přepravy. V případě řídícího systému chlazení jaderného reaktoru může nastat situace, že v jaderném reaktoru dojde k úniku radiace a tím k ohrožení životů lidí nacházejících se v okruhu několika kilometrů od reaktoru.

- **Soft** RT systém je takový systém, který při nedodržení časových mezí degraduje svůj výkon, ale funkčnost tohoto systému není omezena.
- Další kategorií je **Firm** RT systém. Tento systém se vyznačuje tím, že nedodržení několika časových mezí nevede k selhání systému. Nedodržením většího počtu časových mezí, dochází k selhání systému, které může mít katastrofální následky.
- Nejvyšší nároky jsou kladeny na **Hard** RT systém. Pokud u tohoto systému nastane překročení nějaké časové meze, tak dojde k nevratnému poškození systému, které má katastrofální následky.

2.4 RTOS jádro

Real-time operační systémy typicky podporují víceúlohové zpracování (multitasking). Jádro poskytuje také mechanismy k vzájemné komunikaci úloh, synchronizaci a zajištění přístupu ke sdíleným prostředkům (periferie, paměť a další) úloh.

2.4.1 Víceúlohové zpracování

Víceúlohové zpracování je zajištěno přepínáním kontextu mezi jednotlivými úlohami. Takto lze dosáhnout dojmu paralelního zpracování úloh. V jeden čas přitom na procesu běží pouze jedna úloha. Rozhodování o tom, která úloha bude spuštěna má na starosti plánovač. Více o plánovačích pojednává sekce 2.4.5.

Realizovat víceúlohovost lze například pomocí cyklického provádění úloh, kde jednotlivé úlohy jsou prováděny sekvenčně v nekonečné smyčce. V RTOS jádrech je vhodnější použít systém využívající přerušení. Plánovač je volán v periodických intervalech softwarovým přerušením. Další situace, kdy může dojít k přeplánování, je při výskytu vnějšího přerušení, kdy dojde k uložení kontextu právě prováděné úlohy a následně k zavolání obsluhy přerušení. Na konci této obsluhy se musí nacházet mechanismus výběru úlohy, která bude po opuštění přerušení pokračovat ve svém běhu.

2.4.2 Preemptivita

Další důležitou vlastností RTOS jádra je preemptivita. Preemptivní RT systémy převládají. Systém je preemptivní pokud jádro RTOS může úloze odebrat systémové prostředky a přidělit je jiné úloze. Pokud se systém spoléhá na uvolnění systémových prostředků úlohou (například po dokončení jejího běhu), jedná se o *nepreemptivní* systém.

2.4.3 Komunikační a synchronizační prostředky

Komunikační a synchronizační prostředky jsou důležitou součástí jádra RTOS. Umožňují sdílení dat mezi úlohami a řízení přístupu ke sdíleným prostředkům systému.

Semafory

Jedná se o nejčastěji používaný synchronizační prostředek obsahující celočíselný čítač. Je implementován pomocí dvou atomických operací `wait` a `signal`. Když je semafor dostupný (hodnota čítače je větší než 0) a úloha provede operaci `wait`, dojde k dekrementaci hodnoty čítače a úloha může pokračovat dál. Jestliže je ale hodnota čítače semaforu rovna nule, dojde k uspání úlohy, a dojde k zařazení této úlohy do fronty. Uvolnění semaforu je reprezentováno operací `signal`, která inkrementuje hodnotu čítače. Výběr čekajících úloh na uvolnění semaforu může být buď podle priority úlohy a nebo podle času zařazení do fronty [11].

Mutexy

Binární semafor se nazývá mutex. Může nabývat hodnot 0 nebo 1. Mutex je používán pro řízení přístupu do kritické sekce.

Poštovní schránky

Poštovní schránkou je možné zaslat pouze jednu zprávu. Nejčastěji se jedná o ukazatel na data, a proto je nutné, aby nejen odesílatel ale i příjemce věděl, na jaký typ dat ukazatel umístěný ve schránce ukazuje. K jedné schránce může přistupovat více úloh a tak je každé schránce přidělena fronta čekajících úloh. Pokud je schránka prázdná, úlohy čekající ve frontě jsou uspány. Obvykle jádro poskytuje možnost nastavení maximální čekací doby (timeout) ve schránce, aby se zamezilo uváznutí. Po vypršení této doby je úloha jádrem přepnuta do stavu *připravena* (ke spuštění) a funkce pro vybrání zprávy ze schránky vrací chybový kód [13].

Fronty zpráv

Fronta zpráv je v podstatě pole poštovních schránek. Používá se tedy tam, kde je potřeba zaslání jedné a více zpráv. Úlohy nebo obsluhy přerušení pomocí systémových funkcí ukládají zprávy(ukazatel) do fronty. Jedna nebo více úloh pak mohou přijmout zprávy z fronty. Obecně je první zpráva vložená do fronty také jako první z fronty vybrána. Jde tedy o princip FIFO, ale některé RTOS umožňují čtení zpráv v opačném pořadí než v jakém do fronty přišly (LIFO) [13].

Globální proměnné

Nejjednodušší a nejrychlejší způsob komunikace mezi úlohami. Je zde ale problém s řízením přístupu k této paměti. Musíme v rámci programu zaručit exklusivní přístup k této paměti například pomocí kritické sekce.

2.4.4 Správa paměti

Implementace správy paměti má velký vliv na doby odezev jádra RTOS. Dynamické přidělování (alokace) paměti je časově náročné, hlavně když dochází k výpadkům stránek nebo když dochází k fragmentaci. Pokud už v době překladu je známá paměťová náročnost aplikace, je lepší paměť přidělovat staticky.

Zásobník

Víceúlohové RTOS potřebují prostředky pro ukládání/obnovu kontextu jednotlivých úloh. To nám zaručuje zásobník. V jádře RTOS se může nacházet jeden globální zásobník a nebo více zásobníků. Příkladem může být jádro, ve kterém má každá úloha svůj vlastní zásobník, což je použito u komplexnějších RTOS.

Přidělování paměťových bloků pevné velikosti

Dostupný adresový prostor je rozdělen na stejně či různě velké alokační bloky. Základní jednotkou paměti je *paměťový blok*. Úloze je přidělován paměťový blok staticky přímo při vytváření a nebo dynamicky až za běhu RT aplikace. Dynamické přidělování je vhodné pro systémy s měnícím se počtem úloh v čase. Problémem při dynamickém přidělování/uvolňování paměti je její fragmentace. Více o tomto problému se lze dočíst v [13]. Pokud vznikne fragmentace, nelze zaručit přidělování paměťových bloků v konstantním čase, což může být příčinou nedeterminičnosti RTOS jádra.

Stránkování

Tato metoda přístupu k paměti je u RTOS jader málo používaná. Zde se odezvě systému neblaze projevují výpadky stránek. Některé RTOS proto nabízejí techniku *zamykání paměti*. Touto technikou lze celou úlohu nebo její část uzamknout v paměti a tím se zabrání ukládání takto zamknutých dat do sekundární paměti.

2.4.5 Plánovač

Plánovač je základní funkcí jádra. Snaží se přitom o to, aby byly dodrženy časové meze pro každou úlohu tvořící RT aplikaci. Výsledkem činnosti plánovače je plán/rozvrh předepisující časování stavových přechodů jednotlivých úloh. Plánovací algoritmy je možné rozdělit do následujících skupin:

- on-line/off-line,
- preemptivní/nepreemptivní,
- plánování s nejlepší snahou,
- centralizované/distribuované.

Off-line plánování probíhá před spuštěním úloh a lze jej efektivně implementovat. Výsledkem je statický plán, který nemůže reagovat na podněty okolí a změny parametrů úloh. Naproti tomu **on-line** plánování umožňuje vybrat úlohu na základě výskytu události nebo analýzy aktuálně běžících úloh. Nevýhodou tohoto plánování je implementační náročnost.

U **nepreemptivního** plánování nedochází k přerušení běžící úlohy. Nevýhodou tohoto přístupu je možnost nedodržení časových odezev úloh. Přístup do kritické sekce se u této metody plánování nemusí řešit, což může být výhoda. **Preemptivní** plánování umožňuje plánovači přerušit běžící úlohu a přiřadit procesor naléhavější úloze. Preemptivní plánování je možné pouze u preemptivních úloh [13].

Plánování s nejlepší snahou se snaží při plánování *soft* RT úloh co nejlépe využít času procesoru k dodržení časových mezí úloh, přičemž je tolerována jistá míra chyb v časování. U plánování *hard* RT úloh musí být všechny časové meze dodrženy a chyby v časování nejsou tolerovány.

Centralizované plánování je implementováno výhradně na jedné centralizované architektuře, zatímco **distribuované** plánování je rozděleno mezi několik spolu vzájemně komunikujících uzlů.

Teorie plánování je velmi obsáhlá. Zde jsou uvedeny pouze některé základní vlastnosti plánovačů, přičemž podrobnější informace včetně konkrétních plánovacích algoritmů lze nalézt v [13].

2.4.6 Výběr jádra

Při výběru RTOS je nutné nutné zohlednit různé požadavky jak technické, tak i komerční. Mezi základní kritéria patří:

- včasnost odezvy,
- odolnost proti přetížení,
- předvídatelnost,
- odolnost proti poruchám,
- udržovatelnost [12].

Pro výběr vhodného RTOS, ale tato kritéria většinou nepostačují. Jsou příliš obecná, a proto se k těmto kritériím přidávají další, které zohledňují požadavky kladené na aplikaci. Například RTOS použitý v pračce nemusí být tak odolný proti chybám jako systém použitý v letadle. I cena obou RTOS bude určitě rozdílná. Systém použitý v letadle musí mít certifikace, které jej opravňují používat v letectví a musí splňovat další přísné normy. Další rozhodující kritéria pro výběr RTOS:

- cena,
- dostupnost zdrojových kódů,
- možnost rozšíření systému o různé moduly (podpora síťových protokolů, grafické rozhraní, apod.)
- maximální počet úloh,
- podpora ze strany výrobce,
- podpora ze strany překladačů,
- podpora pro ladění RT aplikace za jejího běhu,
- podpora různých procesorů,
- paměťová náročnost,
- podpora plánovacích mechanismů,
- a mnohé další požadavky [12].

Podrobnější informace o výběru RTOS lze nalézt v [12]

Kapitola 3

μC/OS-II

Tato kapitola obsahuje informace o real-time operačním systému μC/OS-II. Je zde stručně popsán obecný postup pro provedení přenosu systému (portace) na libovolnou procesorovou architekturu. Dále jsou zde popsány důležité struktury jádra a služby, které systém poskytuje. Jde spíše jen o stručný popis podrobnější informace v knize věnované tomuto RTOS jádru uvedené v seznamu literatury [11].

Real-time operační systém americké firmy Micrum[1] autorem tohoto RTOS je Jean J. Labrosse. Jak už název napovídá jedná se o druhou verzi tohoto RTOS jádra, jehož první verze byla publikována v roce 1992. Od první verze se tento systém odlišuje rozšířením služeb poskytovaných systémem. Byly přidány funkce jako odstranění úlohy, kontrola zásobníku, volání uživatelsky definovaných funkcí v jádru μC/OS-II a další. Aplikace napsaná v první verzi je lehce přenositelná do druhé verze, stačí jen změnit hlavičkové soubory ve zdrojových kódech.

μC/OS-II je napsaný v programovacím jazyce ANSI C [9]. Jeho zdrojové kódy jsou pro nekomerční učely volně ke stažení z firemních internetových stránek. Vyznačuje se tím, že pro přenos systému na jinou procesorovou architekturu potřebuje minimum kódu symbolických instrukcí (assembly language).

Díky tomu je snadná jeho portace na jiné procesory. Aby mohl být systém naportován na nějakou procesorovou architekturu, musí překladač obsahovat nízkoúrovňové instrukce pro práci se zásobníkem. Důkazem jednoduché portace je existence portů μC/OS-II na více než 100 rozdílných procesorových architektur přes 8-bitové až po 64-bitové.

Charakteristika μC/OS-II:

- μC/OS-II je navržen pro vestavěné aplikace, lze ho jednoduše vložit do vytvářeného systému.
- Systém je škálovatelný. To znamená, že je navržen tak, aby programátor měl možnost vybrat si pouze některé z nabízených služeb systému (vybrat si pouze ty služby systému, které využije v navrhované aplikaci). To umožňuje minimalizovat paměťovou náročnost systému (RAM tak i ROM paměť).
- Obsahuje plně preemptivní jádro. To znamená, že μC/OS-II zaručuje, že systém vždy přepne kontext na připravenou úlohu s nejvyšší prioritou. Může nastat situace, že během provádění nějaké úlohy dojde ke změně seznamu připravených úloh tak, že je na něj přidána úloha s vyšší prioritou než priorita aktuálně běžící úlohy. V tomto okamžiku dojde k přeplánování úloh a spuštění připravené úlohy s nejvyšší prioritou.

[1] http://www.micrium.com/

- Jádro je deterministické. To znamená, že známe dobu trvání jednotlivých služeb jádra. Kromě jedné funkce, doba provádění funkcí nezáleží na počtu úloh v systému.
- Každá úloha má svůj vlastní zásobník. Velikosti zásobníků jednotlivých úloh se mohou lišit. Jednou z vlastností systému je také kontrola zásobníku. Je možné určit, jak velký zásobník úloha potřebuje.
- Jádro podporuje plánování až 254 úloh (do verze 2.8 64 úloh). Každá úloha pracuje na jedinečné prioritní úrovni. To znamená, že cyklická obsluha není podporována.
- Poskytuje mnoho systémových služeb jako třeba poštovní schránky, semafory, správu paměťových bloků pevné velikosti a další.
- Systém obsahuje správu přerušení. Pokud nastane přerušení, systém přeruší aktuálně běžící úlohu. Jestliže během přerušení dojde k probuzení výšeprioritní úlohy, tak bude tato úloha po konci obsluhy přerušení spuštěna.
- Je certifikován FFA pro komerční použití a splňuje náročné požadavky tohoto standardu pro použití v letectví. Splnění požadavků tohoto standardu ukazuje, jak je tento RTOS robustní a bezpečný.

3.1 Struktura jádra

Tato sekce obsahuje strukturní vlastnosti jádra μC/OS-II, které jsou důležité pro pochopení základních vlastností jádra μC/OS-II. Jsou zde popsány základní datové struktury jádra, řešení přístupu do kritické sekce, vysvětlen význam pojmu úloha, změna kontextu a vlastnosti plánovače jádra μC/OS-II.

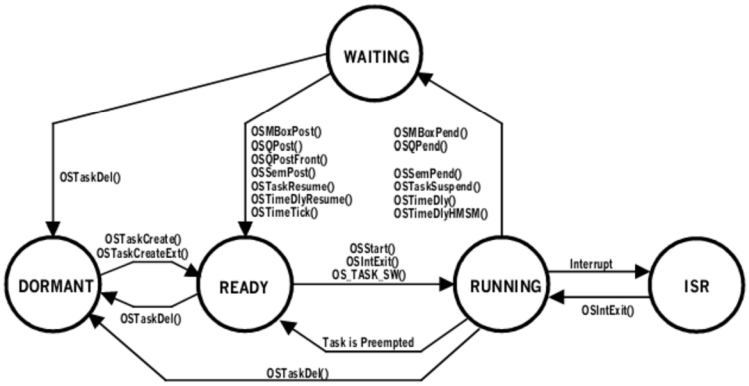
3.1.1 Kritická sekce

μC/OS-II stejně jako ostatní real-time operační systémy potřebuje zakázat přerušení, aby mohl přistoupit ke kritické sekci a povolit přerušení při jejím opuštění. Tímto způsobem předchází současnému přístupu úloh nebo obsluh přerušení ke kritickým úsekům kódu. Doba, po kterou jsou přerušení zakázána, je jedním z nejdůležitějších údajů každého real-time jádra, protože ovlivňuje *reaktivnost* tohoto RT systému.

μC/OS-II se snaží tuto dobu snížit na minimum, ale větší podíl na době, kdy jsou přerušení zakázána, má architektura procesoru a kvalita strojového kódu vygenerovaného překladačem. Některé překladače umožňují povolovat nebo zakazovat přerušení přímo v jazyce C, jiné zase umožňují vložit kód jazyka symbolických instrukcí do programu napsaném v jazyce C. Pro skrytí způsobu této implementace různými kompilátory jsou v μC/OS-II použita makra `OS_ENTER_CRITICAL()` a `OS_EXIT_CRITICAL()`. Jelikož jsou tato makra procesorově specifická, tak jsou definována v souboru `os_cpu.h`, který je vytvářen při portaci systému. Portaci se věnují části 3.4 a ukázka pro konkrétní procesor je zde 4.5.

3.1.2 Úlohy

Aplikace vyvíjená pod μC/OS-II je tvořena úlohami. Úloha je obvykle nekonečná smyčka implementovaná pomocí `for(;;)` nebo `while(1)`. Na konci této smyčky je většinou úloha na určitý čas uspána.



Obrázek 3.1: Stavy úlohy. Zdroj [11].

Struktura úlohy je stejná jako kterékoli jiné funkce jazyka C. Hlavička funkce vypadá takto: `void uloha(void *pdata)`. Jedná se tedy o funkci, jejíž návratová hodnota je typu `void` a parametrem funkce je ukazatel, pomocí kterého lze úloze předávat odkaz na libovolná data (typ dat ale musí úloha znát). Kód samotné funkce lze rozdělit do dvou částí, před nekonečnou smyčkou je část inicializační a v nekonečné smyčce je užitečný kód úlohy. Na obrázku 3.1 je ukázán stavový diagram úlohy. Úloha nacházející se v programové paměti je ve stavu *odložená* (`DORMANT`), pokud se nevyskytuje v systému. V μC/OS-II je možné vytvořit úlohu (zavedení úlohy do systému) dvěma způsoby. Slouží k tomu tyto dvě funkce `OSTaskCreate()` a `OSTaskCreateExt()`.

Po zavolání jedné z těchto funkcí, je úloha vytvořena a nachází se ve stavu *připravená* (`READY`). Úlohu je možné vytvořit před startem samotného systému nebo lze úlohu zavést dynamicky za běhu systému. Úlohy nacházející se v systému lze odstranit pomocí funkce `OSTaskDel()`. Úloha bude odstraněna ze systému, ale pořád bude uložena v programové paměti.

Ve stavu *spuštěná* (`RUNNING`) se může nacházet pouze jedna úloha, a to ta která je naplánovaná plánovačem úloh.

Do stavu *čekající* `WAITING` se úloha může dostat více způsoby. Zavoláním funkcí pro uspání úlohy na určitý čas `OSTimeDly()` nebo `OSTimeDlyHMSM()`. Další možností je zavolání funkcí `OSSemPend()`, `OSMboxPend()` nebo `OSQPend()`, které čekají na výskyt události.

Běžící úlohu lze přerušit mimo úlohy se zakázaným přerušením. Obsluha přerušení může vyvolat události, při kterých se z čekajících úloh stanou úlohy připravené k běhu. Pokud je připravena k běhu úloha s prioritou vyšší než byla priorita přerušené úlohy, dojde k přeplánování úloh a je spuštěna úloha s nejvyšší prioritou. Jestliže se mezi připravenými úlohami nevyskytuje úloha s vyšší prioritou, pokračuje v běhu úloha, která byla přerušena.

3.1.3 Task Control Blocks (TCB)

Každé vytvořené úloze je přiřazen její vlastní *task control block*. Jedná se o datovou strukturu `OS_TCB`, ve které jádro μC/OS-II udržuje informace o aktuálním stavu (kontextu) úlohy nacházející se v systému. Pokaždé, když dojde k přepnutí kontextu úloh, je využívána tato struktura. U přerušené úlohy dochází k aktualizaci aktuálního ukazatele na vrchol zásobníku úlohy `OSTCBPtr`, u spuštěné ulohy je tento ukazatel využíván k obnově stavu úlohy před jejím přerušením. Pokud je úloha vytvořena funkcí `OSTaskCreateExt()`, obsahuje struktura TCB ukazatel na uživatelem definované datové rozšíření. TCB jednotlivých úloh v systému

jsou uchovány v paměti RAM a tvoří obousměrný vázaný seznam.

Struktura TCB obsahuje například výše zmíněný ukazatel vrcholu zásobníku, prioritu úlohy, stav úlohy, ukazatel na předešlý a následující TCB. Pro usnadnění přístupu k položce ukazatele vrcholu zásobníku z jazyka symbolických instrukcí je tato položka umístěna ve struktuře jako první, více v části 3.4.

3.1.4 Plánování úloh

Plánovač `OSSched()` vybírá z úloh nacházejících se ve stavu `READY`, úlohu s nejvyšší prioritu. Tato úloha přejde do stavu `RUNNING` a tím dojde k jejímu spuštění. Čas výběru úlohy plánovačem implementovaným v jádru μC/OS-II není závislý na počtu úloh v systému.

Plánovač nejprve vybere úlohu s nejvyšší prioritou, pokud je stejná jako aktuálně běžící úloha, k přeplánování nedochází. Pokud se mezi připravenými úlohami nachází úloha s vyšší prioritou, dojde ke změně kontextu. Celá funkce je považována za kritickou sekci. Pokud dojde k zavolání plánovače v obsluze přerušení nebo když je přeplánování zakázáno, tak se plánovač ukončí, aniž by provedl přeplánování.

3.1.5 Změna kontextu

Funkce pro změnu kontextu `OS_TASK_SW()` je volána pouze v plánovači a je jeho posledním krokem. Má na starosti uložení kontextu aktuálně běžící úlohy a obnovu kontextu úlohy, která byla vybrána plánovačem k běhu. Funkce simuluje přerušení a je závislá na architektuře procesoru. Proto je jako jedna z mála funkcí implementována při portaci systému. Podrobnější informace o činnosti této funkce jsou uvedeny v části 3.4 a ukázka pro konkrétní procesor je zde 4.5.

3.1.6 Systémové úlohy

V μC/OS-II se vždy nachází *idle úloha*. Tato úloha má nejnižší prioritu ze všech úloh nacházejících se v systému a neobsahuje žádný užitečný kód. V systému je z důvodu zajištění přepnutí kontextu i v případě, že žádná uživatelem definovaná úloha není připravena ke spuštění.

Další úlohou, jejíž zavedení do systému lze zakázat, je *statistická úloha*. Úkolem této úlohy je zjišťování vytíženosti systému během časového úseku jedné sekundy běhu systému. Při inicializaci systému se spočítá počet cyklů systému za vteřinu. Každou následující vteřinu je vypočítáno zatížení systému jako poměr mezi počtem cyklů strávených v uživatelských úlohách a hodnotou zjištěnou v předešlém kroku.

3.2 Komunikační a synchronizační služby

Jádro μC/OS-II podporuje mechanismy pro komunikaci a synchronizaci uvedené v části 2.4.3. Pokud aplikace některé z těchto prostředků nepoužívá, je možné je zakázat v konfiguračním souboru `os_cfg.h`. Tímto způsobem lze ušetřit místo v programové paměti.

Úlohy a ISR mohou komunikovat s jakoukoli jinou úlohou. Každý takovýto podnět je považován za událost, která má svoje svoje jméno. Správa událostí je v jádře řešena pomocí datové struktury nazývané *Event Control Block* (ECB). Volitelným parametrem úlohy je maximální doba čekání (timeout) na příchod dané události. Pokud čeká na výskyt události více úloh, pak je vybrána pouze ta s nejvyšší prioritou. Pouze tato úloha přejde do stavu připravená. Pokud úlohou definovaný timeout vyprší, je odebrána ze seznamu čekajících

úloh tohoto ECB. Každý komunikační prostředek (semafory, poštovní schránky, fronty a jiné) má svůj vlastní ECB.

ECB obsahuje typ události, frontu čekajících úloh na prostředek a také ukazatel na data. V případě poštovní schránky ukazatel na zasílaná data, u fronty je to ukazatel na pole ukazatelů.

3.2.1 Semafory

μC/OS-II semafor se skládá ze dvou částí: 16-bitového celého čísla uchovávající hodnotu čítače a seznamu úloh čekajících na hodnotu čítače semaforu větší než 0. Před použitím semaforu musí být semafor vytvořen a inicializován. Vytvoření semaforu obstarává funkce `OSSemCreate()` a inicializaci provedeme uložením počáteční hodnoty semaforu. Dalšími funkcemi pro práci se semaforem jsou funkce `OSSemDel()`, `OSSemPend()`, `OSSemPendAbort()`, `OSSemPost()`, `OSSemSet()`, `OSSemAccept()` a `OSSemQuery()`.

3.2.2 Mutexy

Jedná se speciální typ semaforu, který může nabývat pouze hodnot 0 nebo 1. V μC/OS-II je u mutexů implementován mechanismus pro omezení inverze priorit (semafory toto implementované nemají) [13]. Dojde-li k takové situaci, že výšeprioritní úloha požádá o semafor vlastněný nížeprioritní úlohou, tak nížeprioritní úloha získá po dobu vlastnění semaforu prioritu výšeprioritní úlohy. Metody pro práci s mutexy jsou: `OSMutexCreate()`, `OSMutexAccept()`, `OSMutexDel()`, `OSMutexPend`, `OSMutexPost` a `OSMutexQuery`.

3.2.3 Poštovní schránky

Poštovními schránkami je možné v μC/OS-II poslat ukazatel na libovolně velká data. Komunikující úlohy musí znát typ posílaných dat. Schránka je prázdná, pokud má ukazatel na data hodnotu `NULL`, jinak je schránka plná. Funkce pracující se schránkou jsou `OSMboxCreate()`, `OSMboxDel()`, `OSMboxPend()`, `OSMboxPendAbort()`, `OSMboxPost()`, `OSMboxPostOpt()`, `OSMboxAccept()` a `OSMboxQuery()`.

3.2.4 Fronty zpráv

Omezením poštovních schránek je možnost poslat pouze jednu zprávu. Fronta zpráv nás tohoto omezení zbavuje tím, že obsahuje pole poštovních schránek. Velikost této fronty je možné upravit v konfiguračním souboru systému `os_cfg.h`. Práci s frontou zpráv poskytují funkce: `OSQCreate()`, `OSQDel()`, `OSQPend()`, `OSQPendAbort()`, `OSQPost()`, `OSQPostFront()`, `OSQPostOpt()`, `OSQAccept()`, `OSQFlush()` a `OSQQuery()`.

3.3 Hook funkce

μC/OS-II obsahuje podporu pro tzv. *hook* funkce. Tyto funkce jsou vhodně vloženy do zdrojových souborů jádra a umožňují rozšiřovat nebo pozměňovat chování systému, aniž by programátor musel zasahovat do zdrojových souborů jádra. Tyto funkce lze také používat například i k měření dob vykonávání služeb jádra.

Programátor má v konfiguračním souboru `os_cfg.h` možnost povolit/zakázat používání těchto uživatelsky definovaných funkcí. Pokud uživatel povolí nějakou hook funkci, musí ji i definovat, jinak se překlad μC/OS-II nezdaří.

3.4 Portace

Portace μC/OS-II je docela jednoduchá za předpokladu, že programátor zná do detailu architekturu procesoru a kompilátoru jazyka C. Vlastní zdrojový kód portace se v závislosti na procesoru skládá z 50 až 300 řádek kódu.

Požadavky kladené na kompilátor:

- Kompilace programu napsaném v jazyce C pro daný procesor,
- musí podporovat programování v jazyce symbolických instrukcí,
- musí podporovat povolení/zakázání přerušení pomocí jazyka C nebo vložení in-line příkazů jazyka symbolických instrukcí do zdrojového kódu s funkcemi napsanými v jazyce C.

Porty pro všechny procesorové architektury musí obsahovat tyto dva nebo tři soubory: `os_cpu.h`, `os_cpu_c.c` a volitelně `os_cpu_a.asm`. Soubor s kódem jazyka symbolických instrukcí je volitelný, protože některé překladače umožňují vkládat tento kód přímo do souboru s C funkcemi `os_cpu_c.c`.

3.4.1 Soubor `os_cpu.h`

Obsahem tohoto souboru jsou implementačně specifické konstanty makra a vytváření nových datových typů pomocí operátoru `typedef`.

Nastavení datových typů nezávislých na překladači

Různé mikrokontroléry mají různou délku slova. Proto μC/OS-II obsahuje řadu definic typů pro zajištění přenositelnosti. μC/OS-II nepoužívá typy jazyka C. Datové typy `short`, `int` nebo `long` nejsou přenositelné, jelikož mají jinou bytovou délku na 8, 16, 32-bitových architekturách procesoru. Místo těchto datových typů používá μC/OS-II vlastní datové typy `INT8X`, `INT16X` nebo `INT32X`, čísla u typů označují počet bitů datového typu. Místo písmene X je buď `S` pro znaménkový typ nebo `U` pro bezznaménkový typ. Přetypování datových typů jazyka C na datové typy používané v μC/OS-II je realizováno pomocí příkazu typedef. Na následujícím řádku je ukázán příklad, kde na nějakém 32-bitové procesoru je datový typ `INT16U` definován jako `unsigned short`.

```
typedef unsigned int   INT16U;
```

Datový typ položky zásobníku

Dalším krokem je definice datového typu položky zásobníku `OS_STK`. Tuto informaci lze nalézt v dokumentaci překladače. Každý zásobník úlohy musí být deklarován jako datový typ `OS_STK`.

Růst zásobníku

Některé typy mikrokontrolérů mají růst zásobníku implementován od vyšší adresy k nižší jiné zase naopak. O směru růstu zásobníku informuje konstanta `OS_STK_GROWTH`:

```
#OS_STK_GROWTH 0 /*od nižší adresy k vyšší*/
#OS_STK_GROWTH 1 /*od vyšší adresy k nižší*/
```

Makra pro řízení přístupu do kritické sekce

Jedná se o dvojici maker, které je nutno volat v páru. Při vstupu do kritické sekce se volá makro `OS_ENTER_CRITICAL()` a při opuštění kritické sekce se volá makro `OS_EXIT_CRITICAL()`. Implementace tohoto makra je možná třemi různými způsoby:

- **Disable/Enable** – Tato metoda je jednoduchá a efektivní. Využívá přímého nastavování povolení/zakázání přerušení, takže každé makro většinou obsahuje pouze jednu instrukci. Problém této metody ale spočívá v tom, že po opuštění kritické sekce bude přerušení vždy povoleno navzdory stavu nastavení přerušení před vstupem do kritické sekce.
- **Push/Pop** – U této metody je pro uchování stavu přerušení používán zásobník. Tělo makra `OS_ENTER_CRITICAL()` obsahuje nejprve instrukce pro uložení aktuální hodnoty masky globálního přerušení a až poté dojde k zakázání přerušení. Po opuštění kritické sekce je ze zásobníku původní hodnota globální masky přerušení obnovena. Výhodou této metody je uchování stavu přerušení, nevýhodou může být větší režie.
- **Save/Restore** – Poslední metoda je podobná předchozí. Rozdíl je ve způsobu uložení masky globálního přerušení. U tohoto přístupu je pro uložení globální masky přerušení použita lokální proměnná funkce, ve které je kritická sekce volána. Implementace této metody záleží na použitém překladači a procesoru.

Záleží na programátorovi, kterou metodu použije. Pokud potřebujeme omezit čas strávený v kritické sekci a nezajímá nás hodnota globální masky po vystoupení z kritické sekce, použijeme první metodu.

Přepnutí kontextu na úrovni úloh

Makro `OS_TASK_SW()` je vyvoláno pokud dochází k přepnutí mezi nížeprioritní úlohou a výšeprioritní úlohou. Jiný mechanismus přepínání kontextu je `OSIntExit()`, který je volán pokud v průběhu přerušení přešla do stavu *připravená k běhu* úloha, která má vyšší prioritu než úloha přerušená přerušením.

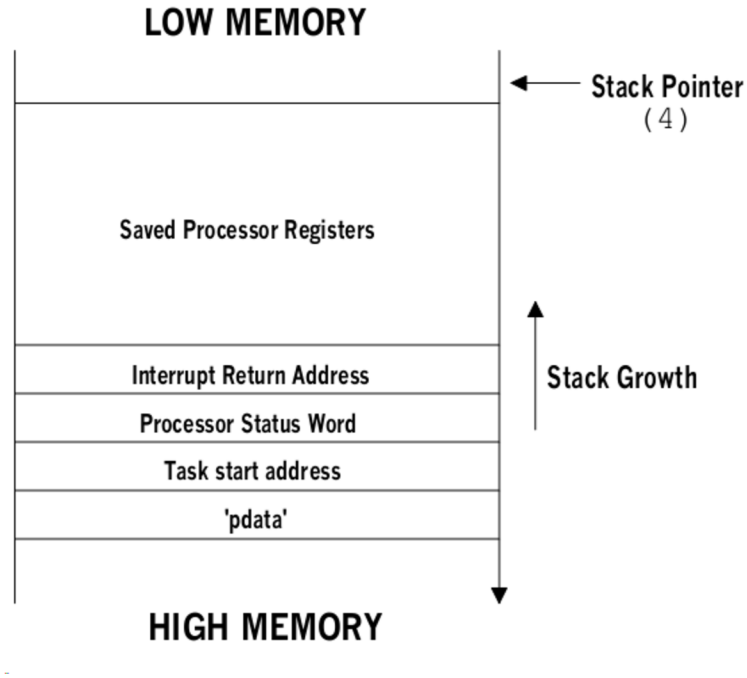
Přepnutí kontextu může být implementováno vygenerováním softwarového přerušení, jehož obsluha `OSCtxSw()` provede procesorově specifické kroky nutné k uložení kontextu přerušené úlohy a obnově kontextu spuštěné úlohy. Pokud procesor neumožňuje softwarové přerušení, tak makro `OS_TASK_SW()` jednoduše zavolá funkci, která simuluje přerušení. Přepnutí kontextu je pak stejné jako obsluha přerušení `OSCtxSw()`.

3.4.2 Soubor `os_cpu_c.c`

V tomto souboru se nacházejí funkce napsané v jazyce C. Nacházejí se zde funkce: `OSTaskStkInit()`, `OSTaskCreateHook()`, `OSTaskDelHook()`, `OSTaskSwHook()`, `OSTaskStatHook()` a `OSTimeTickHook()`. Jedinou funkcí, která musí být povinně definována je funkce `OSTaskStkInit()`. Ostatní funkce musí být pouze deklarovány (mohou mít prázdné tělo funkce).

`OSTaskStkInit()` – Inicializace zásobníku

Tato funkce je volána při vytváření úlohy pomocí funkce `OSTaskCreate()` nebo `OSTaskCreateExt()`. Inicializuje zásobník úlohy tak, aby vypadal jako při právě nastalém



Obrázek 3.2: Ukázka stavu zásobníku po inicializaci. Zdroj [11].

přerušení. To znamená, že se na zásobník uloží jednotlivé registry procesoru ve stejném pořadí jako při příchodu přerušení. Navíc je jako první uložen na zásobník ukazatel na data předávaná úloze při jejím vytváření. Simuluje se tím vlastně předávání parametrů funkce přes zásobník. Jak může vypadat zásobník po inicializaci ukazuje obrázek 3.2.

3.4.3 Soubor `os_cpu_a.asm`

Port μC/OS-II vyžaduje napsání několika jednoduchých funkcí v jazyce symbolických instrukcí. Pokud překladač podporuje vkládání těchto instrukcí přímo do zdrojového souboru jazyka C, můžeme kód napsaný pomocí jazyka symbolických instrukcí vložit do souboru `os_cpu_c.c`.

`OSStartHighRdy()` – Spuštění úlohy s nejvyšší prioritou

Tato funkce je volána funkcí `OSStart()` a jejím úkolem je zahájit spuštění úlohy s nejvyšší prioritou. Před spuštěním `OSStart()` se musí v systému nacházet alespoň jedna uživatelská úloha. Tělo funkce musí postupně provést tyto kroky:

1: Zavolání uživatelské funkce OSTaskSwHook(),
2: nastavení proměnné OSRunning na TRUE,
3: zjištění ukazatele na TCB úlohy s nejvyšší prioritou,
4: získání ukazatele na vrchol zásobníku této úlohy, obnovení procesorových registrů, zavolání intstrukce pro návrat z přerušení.

Struktura TCB usnadňuje získání ukazatele na vrchol zásobníku tím, že je tento ukazatel umístěn na začátku struktury. Obnovou procesorových registrů je myšleno obnovení všech registrů potřebných k pokračování úlohy, kromě těch, které jsou ukládány na zásobník při

přerušení. Proto musí programátor dobře znát architekturu procesoru a funkci přerušovacího podsystému.

`OSCtxSw()` – Přepínání kontextu na úrovni úloh

Tato funkce je volána v rámci obsluhy přerušení, které je generováno makrem `OS_TASK_SW()`. Musí být proto na příslušnou pozici tabulky vektorů přerušení mikroprocesoru přidán ukazatel na tuto funkci. Kroky nutné k implementaci této funkce jsou:

1: Uložení procesorových registrů aktuálně běžící úlohy,
2: uložení ukazatele vrcholu zásobníku aktuálně běžící úlohy do TCB,
3: zavolání uživatelské funkce OSTaskSwHook(),
4: OSTCBCur = OSTCBHighRdy,
5: OSPrioCur = OSPrioHighRdy,
6: získání ukazatele na vrchol zásobníku úlohy, která bude obnovena,
7: obnovení procesorových registrů,
8: zavolání instrukce pro návrat z přerušní.

Proměnná `OSTCBCur` reprezentuje ukazatel na TCB aktuálně běžící úlohy. V proměnné `OOSTCBHighRdy` se nachází ukazatel na TCB úlohy, která bude obnovena (připravená úloha s nejvyšší prioritou). V proměnné `OSPrioCur` je priorita běžící úlohy a `OSPrioHighRdy` je priorita obnovované úlohy.

`OSIntCtxSw()` – Přepnutí kontexu po dokončení poslední z obsluh přerušení

Tato funkce je volána z funkce `OSIntExit` a to jen pokud jsou všechna přerušení obsloužena a je k běhu připravena úloha s vyšší prioritou, než kterou měla přerušená úloha.

1: Zavolání uživatelské funkce OSTaskSwHook(),
2: OSTCBCur = OSTCBHighRdy,
3: OSPrioCur = OSPrioHighRdy,
4: získání ukazatele na vrchol zásobníku úlohy, která bude obnovena,
5: obnovení procesorových registrů,
6: zavolání instrukce pro návrat z přerušení.

Jak je vidět, tato úloha pouze obnovuje kontext úlohy s nejvyšší prioritou. Proto se musí při přerušení běžící úlohy v rámci obsluhy přerušení ukládat kontext přerušované úlohy. Pseudokód obslužné rutiny přerušení je možno nalézt v následujícím odstavci, kde je ukázáno na konkrétním případě obsluhy systémového času.

Obsluha přerušení systémového času

μC/OS-II vyžaduje periodický hodinový signál. Lze jej generovat například s pomocí čítače/časovače, pokud je součástí periferií mikrokontroléru. Hodinový signál by měl být generován 10-krát až 100-krát za sekundu.

1: Uložení procesorových registrů přerušené úlohy,
2: zavolání systémové funkce OSIntEnter() nebo inkrementace proměnné OSIntNesting,
3: **if** OSIntNesting == 1 **then**
4: ulož ukazatel zásobníku přerušené úlohy do TCB,
5: **end if**
6: zavolání systémové funkce OSTimeTick() (provedení obsluhy přerušení),
7: zavolání systémové funkce OSIntExit(),

8: obnovení registry přerušené úlohy,
9: zavolání instrukce pro návrat z přerušení.

Toto je obecné schéma pro obsluhu přerušení. Stačí nahradit volání funkce `OSTick` voláním uživatelem definované obsluhy přerušení.

Kapitola 4

Platforma Freescale Flexis

Platforma Freescale Flexis je rodina 8-bitových a 32-bitových mikrokontrolérů, která se snaží o co největší unifikaci jejich periferií, pouzder a funkčnosti jejich jader při snaze zachovat jejich výkonostní rozdíly. Výhodou je snadná přenositelnost aplikace napsané v jazyce C mezi těmito mikrokontroléry více v sekci 4.4.

Mezi výhody této platformy patří:

- snadná přenositelnost (mezi 8-bitovým a 32-bitovým mikrokontrolérem),
- jednotné vývojové prostředí (vývoj pro oba mikrokontroléry pod stejným programem),
- optimalizovaná architektura s knihovnou periferií,
- nízký příkon.

4.1 Řada QE128

Tato řada mikrokontrolérů se vyznačuje tím, že oba dva typy mikrokontrolérů mají stejná pouzdra a vývody těchto pouzder jsou vzájemně kompatibilní. Je určena pro použití v aplikacích, kde je vyžadována nízká spotřeba. Typicky se jedná o zařízení napájené pomocí baterií.

- frekvence jádra 50 MHz,
- frekvence sběrnice 25 MHz,
- 128 kB paměti Flash
- 8 kB rychlé paměti SRAM
- ladění (debugging) pomocí 1-pinového rozhraní BDM.

Protože testovací programy budou vyvíjeny na demonstrační desce DEMOQE128, tak se zaměříme na podrobnější popis mikrokontroléru, které lze s touto deskou využít.

4.2 8-bitový mikrokontrolér MC9S0QE128

Doba pro vykonání jedné instrukce u tohoto jádra je 2 takty CPU. Paměť je organizována do stránek.

- architektura CISC,
- 8-bitová datová sběrnice
- 16-bitová adresová sběrnice, lineární ukazatel do paměti,
- 32 vyjímek přerušení/reset v jedné úrovni, v HW není podporováno vícenásobné přerušení,
- ladění (debugging) pomocí 1-pinového rozhraní BDM.

4.2.1 Programovací model

- 8 bitový univerzální registr (A),
- 16-bitový index registr (H:X),
- 16-bitový ukazatel zásobníku (SP),
- 16-bitový programový čítač (PC),
- 8-bitový stavový registr příznaků (CCR). Index registr se skládá ze dvou 8-bitových registrů H a X. Kde H registr je uložen ve vyšším bytu a X registr je uložen v nižším bytu. Mnoho instrukcí používá X registr jako druhý univerzální registr.

4.3 32-bitový mikrokontrolér MCF51QE128

Jedná se o výkonnější mikrokontrolér platformy Flexis. Obsahuje 32-bitové jádro architektury ColdFire V1. Periferie jsou shodné s předchozím 8-bitovým mikrokontrolérem. Rychlost provádění instrukcí je rovna rychlosti CPU. To je rozdíl oproti předchozímu mikrokontroléru, které má rychlost provádění instrukcí poloviční (zdroj AN3629 str.3). Níže jsou popsány některé parametry jádra.

- architektura RISC s proměnnou délkou instrukce,
- 16 uživatelských 32-bitových registrů,
- 32-bitová datová sběrnice s 24-bitovou adresovou sběrnicí, přímo adresovatelná paměť,
- instrukční set ColdFire revize C,
- dvě zřetězené linky, jedna pro načítání instrukcí a druhá pro načítání operandů,
- 256 vyjímek přerušení/reset, které jsou rozděleny do sedmi úrovní
- ladění (debugging) pomocí 1-pinového rozhraní BDM.

4.3.1 Programovací model

MCF51QE128 podporuje dva programovací modely. O použitém modelu rozhoduje bit S stavového reegistru SR.

- **Supervisor** – Určen pro řídící systém, který implementuje funkce vyhrazené operačnímu systému jako jsou I/O operace a správa paměti.
- **Uživatelský** – Určen pro běh uživatelské aplikace.

Programovací model v uživatelském režimu:

- 16 uživatelských 32-bitových registrů (D0–D7,A0–A7),
- 32-bitový programový čítač (PC)
- 8-bitový stavový registr příznaků (CCR).

Registry D0–D7 jsou datové nebo se mohou použít k indexaci. Registry A0–A7 jsou adresové a registr A7 zároveň slouží jako ukazatel na vrchol zásobníku.

Supervisor model navíc obsahuje tyto registry:

- 16-bitový status registr (SR),
- 32-bitový ukazatel na supervisor zásobíku (SSP),
- 32-bitový bázovou adresu vektorů přerušení (VBR),
- 32-bitový konfigurační registr CP (CPUCR).

4.4 Programování a přenositelnost aplikací

Vývojář programuje aplikaci pro oba mikrokontroléry ve stejném prostředí, které se nazývá CodeWarrior. Využívá-li přitom knihovní funkce tohoto prostředí, které přiřazují fyzickým adresám registrů a periferií symbolické názvy, zajistí tím snadnou přenositelnost svého programu. Pokud vývojář například zjistí, že výkon 8-bitového mikrokontroléru pro jeho aplikaci již nedostačuje, stačí ve vývojovém prostředí změnit typ mikrokontroléru na 32-bitový a tím se zároveň změní knihovní funkce na požadovaný typ mikrokontroléru. Programátor tedy nemusí ve svém programu nic měnit. Má to ale i svá omezení. Programátor, který použije ve své aplikaci nízkoúrovňový kód (assembler), musí tento kód přepsat pro konkrétní mikrokontrolér. Dále musí používat takové periferie, které se nacházejí na všech mikrokontrolérech, mezi kterými chce přecházet. Pokud přecházíme z méně výkonného mikrokontroléru na výkonnější, nemusíme tento požadavek řešit.

4.5 Portace

V následujícím textu jsou popsány implementační detaily portace jádra μC/OS-II pro výše zmíněné mikrokontroléry.

4.5.1 MCF51QE128

Port μC/OS-II pro tento mikrokontrolér vyžaduje běh v režimu supervisor. Růst zásobníku je od vyšší adresy k nižší. Při vyvolání přerušení se na zásobník ukládá pouze návratová adresa a status registr (SR).

Definice datových typů

Položka zásobníku je 32-bitů široká a stavový registr procesoru je 16-bitový. Další datové typy jsou uvedeny níže.

```
typedef unsigned  char  BOOLEAN;
typedef unsigned  char  INT8U;    /* Unsigned  8 bit quantity           */
typedef signed    char  INT8S;    /* Signed    8 bit quantity           */
typedef unsigned  short INT16U;   /* Unsigned 16 bit quantity           */
typedef signed    short INT16S;   /* Signed   16 bit quantity           */
typedef unsigned  long  INT32U;   /* Unsigned 32 bit quantity           */
typedef signed    long  INT32S;   /* Signed   32 bit quantity           */
typedef float           FP32;     /* Single precision floating point    */
typedef double          FP64;     /* Double precision floating point    */


typedef unsigned  long  OS_STK;    /* Each stack entry is 32-bit wide    */
typedef unsigned  short OS_CPU_SR; /* Define size of CPU status register */
```

Implementace kritické sekce

Pro portaci byla využita metoda Save/restore viz 3.4. Zde je nutné znát, jak překladač implementuje předávání parametrů funkce a její návratové hodnoty. V prostředí CodeWarrior je možná volba způsobu tohoto předávání. Parametry funkcí lze předávat buď přes zásobník nebo přes registry. Další možností je kombinace těchto dvou variant. V tomto způsobu portace byla použita metoda předávání parametrů přes zásobník. Pokud chceme použít kritickou sekci ve svých programech, musíme před vstupem do kritické sekce definovat proměnnou typu `OS_CPU_SR` a inicializovat ji na nulu.

```
OS_CPU_SR cpu_sr = 0;
```

Funkce pro vstup do kritické sekce OS_ENTER_CRITICAL() vrací stav globálního přerušení přes registr D0. OS_EXIT_CRITICAL() má jako parametr stav přerušení před vstupem do kritické sekce. Tento parametr je předáván přes zásobník. Nachází se hned za návratovou adresou [2].

Inicializace zásobníku

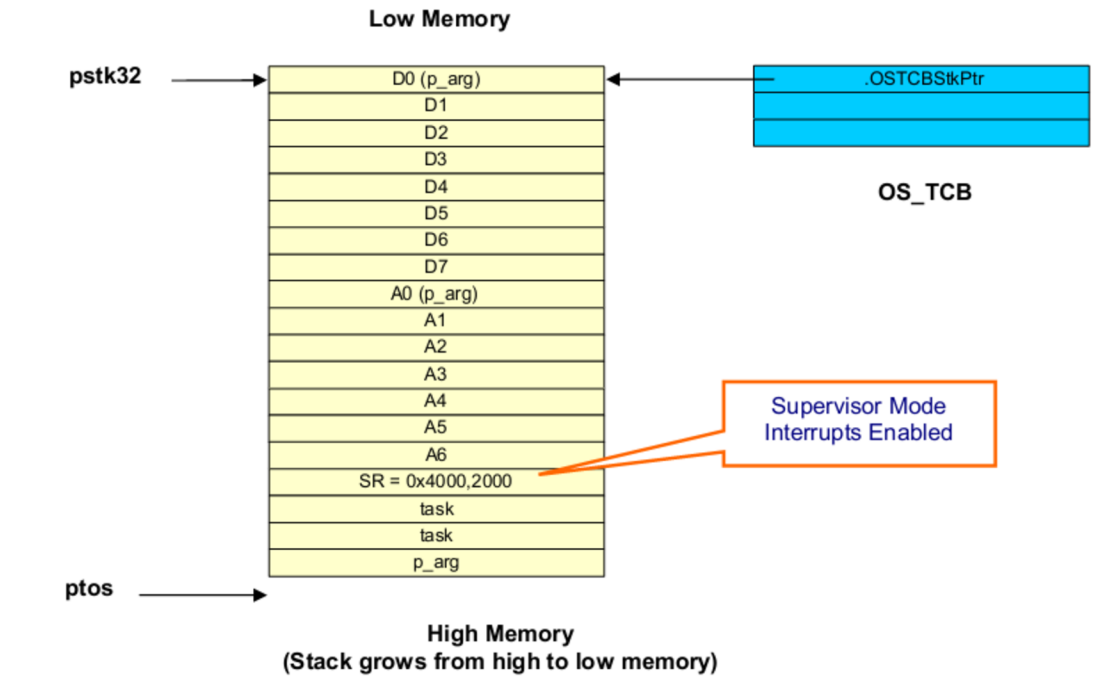
Při vytváření úlohy je inicializován i její zásobník. Pořadí uložení jednotlivých registrů je ukázáno na obrázku 4.1.

Přepínání kontextu

Přepínání kontextu je realizováno pomocí softwarového přerušení. Uložením kontextu jak je uvedeno v 3.4, je v tomto případě myšleno uložení registrů A0–A6 a D0–D7.

4.5.2 MC9S08QE128

Od předchozího 32-bitového mikrokontroléru se tento typ odlišuje chybějícím supervisor režimem. Růst zásobníku je od vyšší adresy k nižší. Při vyvolání přerušení se na zásobník ukládají data v tomto pořadí: návratová adresa, index registr (pouze jeho část (X)), univerzální registr (A) a nakonec stavový registr příznaků.



Obrázek 4.1: Pořadí uložení jednotlivých dat na zásobník. Zdroj [2]

Definice datových typů

Na rozdíl od předchozího mikrokontroléru je položka zásobníku 8-bitů široká stejně jako stavový registr procesoru. Další datové typy jsou uvedeny níže.

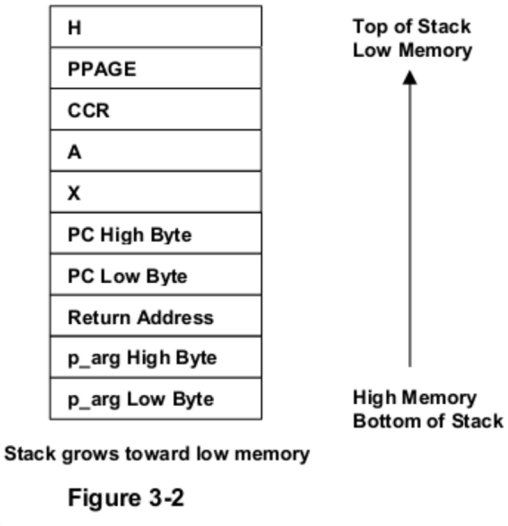
```
typedef unsigned char  INT8U;    /* Unsigned  8 bit quantity           */
typedef signed   char  INT8S;    /* Signed    8 bit quantity           */
typedef unsigned int   INT16U;   /* Unsigned 16 bit quantity           */
typedef signed   int   INT16S;   /* Signed   16 bit quantity           */
typedef unsigned long  INT32U;   /* Unsigned 32 bit quantity           */
typedef signed   long  INT32S;   /* Signed   32 bit quantity           */
typedef float          FP32;     /* Single precision floating point    */
typedef double         FP64;     /* Double precision floating point    */

typedef unsigned char  OS_STK;    /* Each stack entry is 8-bit wide     */
typedef unsigned char  OS_CPU_SR; /* Define size of CPU status register */
```

Implementace kritické sekce

I zde byla pro portaci využita metoda Save/restore viz 3.4. Jediným rozdílem je předávání parametrů funkce a její návratové hodnoty. Odlišně od předchozího mikrokontroléru překladač předává 8-bitové parametry funkce přes univerzální registr (A). Stejně tak návratová hodnota funkce je předávána přes univerzální registr (A) [1].

Inicializace zásobníku

Pořadí uložení jednotlivých registrů je ukázáno na obrázku 4.2. Počet ukládaných položek



Obrázek 4.2: Pořadí uložení jednotlivých dat na zásobník. Zdroj [1]

na zásobník se značně odlišuje od předchozího mikrokontroléru. Zde je jich potřeba mnohem méně.

Přepínání kontextu

Přepínání kontextu je realizováno pomocí softwarového přerušení. Ukládá se pouze číslo stránky (PPAGE) a horní polovina index registru (H).

Kapitola 5

Metody pro testování vlastností (benchmarking) RTOS

Důležitou součástí vývojového cyklu návrhu real-time systému je výběr RTOS a hardwaru, jehož prostředky ke svému běhu RTOS využívá. Pro zjišťování různých vlastností počítačových systémů existují sady programů (benchmarks).

Nejznámějšími jsou metriky pro zjištění výpočetního výkonu procesoru. Příkladem může být benchmark **LINPACK** [4]. Tato sada programů napsaných v jazyce Fortran slouží ke změření hodnoty počtu operací v plovoucí řadové čárce za jednu vteřinu (FLOPS). Mezi další obdobnou metriku výkonnosti počítačového systému patří zjištění hodnoty počtu provedených operací za jednu vteřinu (IPS). Tuto hodnotu lze získat benchmarkem **SPECint** [5].

Pro oblast RT systémů je, jak již bylo uvedeno v předchozích kapitolách, jedna z nejdůležitějších vlastností včasnost odezvy. Na této době se podílí mimo jiné služby a funkce jádra RTOS, které systém potřebuje ke svému běhu. Jádra RTOS jsou většinou prioritně preemptivní, takže jednou z úloh jádra je přepínat kontext 2.4.1. Doba odezvy přepnutí kontextu má tedy podíl na době odezvy systému. Proto benchmark zaměřený na testování doby odezvy systému by měl obsahovat i test na zjištění doby přepnutí kontextu. Dále může obsahovat měření dob odezev synchronizačních a komunikačních prostředků jádra. Příkladem je benchmark Rhealstone 5.1.1.

Nejen doba odezvy RT systému je důležitá. Příklady požadavků, které jsou kladeny na jádro RTOS, jsou uvedeny v části 2.4.6. Existují benchmarky, které se zaměřují na důkladné otestování pouze jedné vlastnosti systému, na příklad robustnosti. Stručný popis vybraných testů robustnosti je uveden v části 5.2. Komplexnější benchmarky se nezaměřují jen na jednu vlastnost systému, ale snaží se zjistit více informací o chování RTOS jádra.

Dalším možným způsobem dělení benchmarků je:

- Testování softwarových vlastností (vlastnosti plánovače, zamezení vzniku inverze priorit a další).
- Testování systému jako celku. Real-time systém je možno rozdělit na řízený systém a řídící systém. Zde se jedná o testování řídícího systému. Řídící systém je složen z programu (RTOS a aplikace nad ním běžící) a hardwaru, na kterém daná aplikace běží.
- Testování systémových služeb, testování maximálního zatížení systému + jaký vliv má zátěž systému na doby odezev systému.

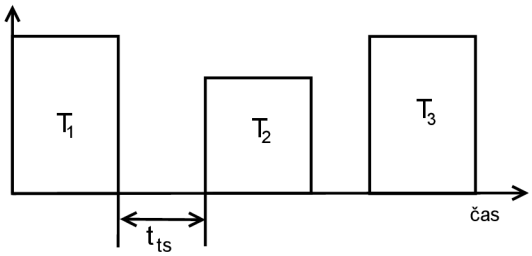
V další části jsou popsány vybrané metody pro testování vlastností RTOS systémů.

5.1 Testovací metody

5.1.1 Rhealstone benchmark

Skládá se z šestice programů napsaných v jazyce C, které se zaměřují na důležité parametry RT systému:

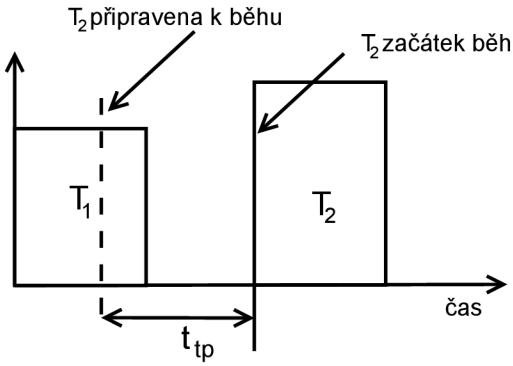
- **Doba přepnutí úlohy** – Čas potřebný k přepnutí mezi dvěma úlohami se stejnou prioritou. Na obrázku 5.1 jsou ukázány tři úlohy T_1, T_2 a T_3 se stejnou prioritou. Jsou



Obrázek 5.1: Čas potřebný k přepnutí mezi úlohami se stejnou prioritou.

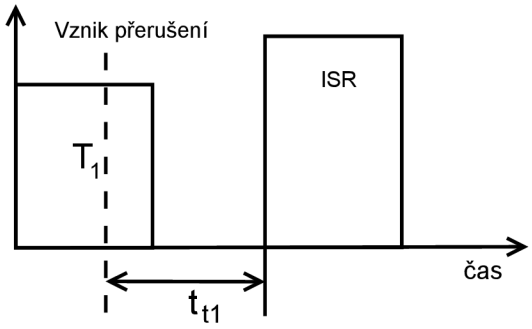
plánovány mechanismem round-robin. Parametr t_{ts} reprezentuje dobu od uplynutí časového kvanta (blokování, ukončení) úlohy T_1 do spuštění úlohy T_2. Tento test je určen k otestování efektivity datových struktur jádra RTOS.

- **Doba preempce úlohy** – Čas od vzniku požadavku na spuštění (obnovu) výše prioritní úlohy, během běhu úlohy s nižší prioritou, do doby vlastního spuštění této úlohy. Ukázka doby preempce úlohy je vidět na obrázku 5.2. Tato doba je složena ze tří složek: doby přepnutí úlohy t_{ts} 5.1, času pro rozpoznání připravenosti úlohy s vyšší prioritou T_2 a čas na jeho zpracování.



Obrázek 5.2: Doba preempce úlohy.

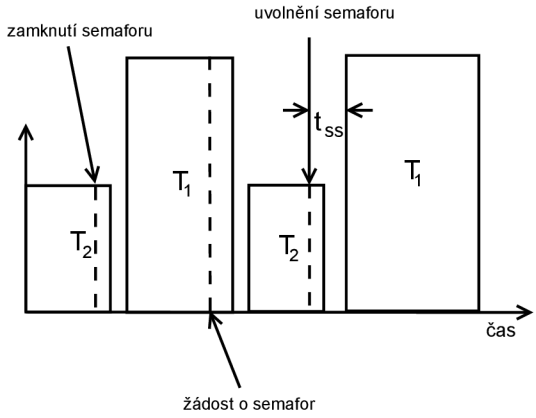
- **Doba odezvy přerušení** – Doba od vzniku přerušení do doby zahájení obsluhy přerušení. Obrázek 5.3 ilustruje vznik přerušení během běhu úlohy T_1. Složky systému, které se na této době podílejí:



Obrázek 5.3: Doba odezvy přerušení.

  - Čas potřebný pro rozpoznání, že došlo k přerušení, určení zdroje přerušení, výběr vektoru přerušení, výběr první instrukce přerušení,
  - doba běhu nejdelší instrukce,
  - nejdelší čas, po který jsou přerušení maskována,
  - čas potřebný k uložení kontextu právě přerušené úlohy.

- **Doba přehození semaforu** – Doba od odemknutí semaforu úlohou s nižší prioritou do zamknutí semaforu úlohou s vyšší prioritou. Na obrázku 5.4 úloha T_2 drží semafor. Úloha T_1 zažádá o semafor a dojde k jejímu blokování. Doba od uvolnění semaforu úlohou T_2 a spuštěním úlohy T_1 je dobou přehození semaforu t_{ss}.

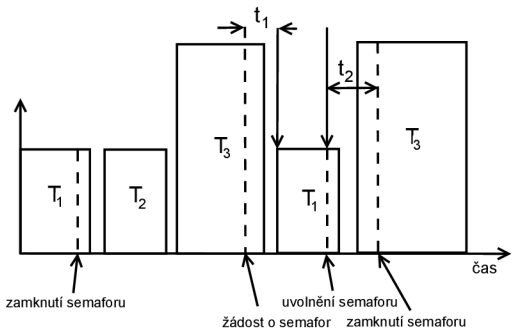


Obrázek 5.4: Doba přehození semaforu.

- **Doba mimo inverzi priorit** – Tato doba se skládá ze dvou složek jak je ukázáno na obrázku 5.5. Doby od rozpoznání vzniku inverze priorit t_1 až do znovuspuštění nížeprioritní úlohy T_1 nacházející se v kritické sekci a doby t_2 od uvolnění nížeprioritní úlohy T_1 až do spuštění úlohy s vyšší prioritou T_3.

- **Propustnost dat mezi úlohami** – Počet bytů, které je možné přesunout mezi dvěma úlohami pomocí zasílání zpráv, za jednu sekundu.

Výsledkem těchto testů je jedna hodnota, která je dána součtem těchto parametrů, přičemž každému z parametrů je přidělena určitá váha.



Obrázek 5.5: Doba mimo inverzi priorit.

Výhodou tohoto testu je komplexní otestování hardware i jádra RTOS. Nevýhodou je omezení testování pouze těchto šesti parametrů. Mezi další nevýhody patří volba vah parametrů, jejichž hodnoty jsou získány empiricky. Pro RT systémy je důležitým parametrem nejhorší možná doba odezvy, ale parametry získány tímto testem jsou pouze průměrné hodnoty získané z několika běhů [3, 14].

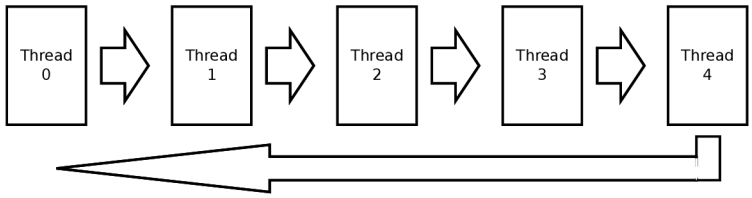
5.1.2 Thread-Metric

Dalším komplexním testem vlastnosti RTOS systémů je Thread-Metric benchmark [6]. Tento benchmark vznikl tvůrci RTOS ThreadX®z firmy Expres Logic Inc.. Jako referenční hodnota výkonu je brána výkonnost ThreadX®. Podle tvrzení tvůrců je tento benchmark lehce přenositelný na ostatní RTOS. Co se měří:

- Čas spotřebovaný RTOS k vykonání specifických služeb jinak nazývaný jako režie
- Aby byl použitelný pro srovnání více RTOS, byla vybrána množina běžných služeb poskytujících jádry. Reprezentativní množina napříč službami a navíc vyskytující se běžně v RTOS systémech.
- Kooperativní přepínání kontextu
- Preemptivní přepínání kontextu
- Zasílání zpráv
- Semafory
- Alokace/dealokace paměti
- Zpracování přerušení s preempcí
- Zpracování přerušení bez preempcí

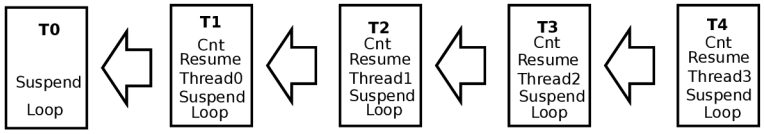
Měření služeb probíhá v iteracích, 30ti vteřinových hlášeních. Vykonání služby a její protislužby. Například zaslání zprávy a její přijetí. Alokace a následná dealokace paměti. Uchování počtu iterací v lokální proměnné. Zasílání výsledků vyhodnocovacímu procesu každých 30 vteřin.

- **Spolupracující úlohy** – Úlohy běží dokud neukončí svoji část a poté samy zavolají plánovač, ten pak vybere další úlohu připravenou k běhu a spustí její vykonání. Test se skládá z pěti úloh se stejnou prioritou. Každá úloha vykoná svoji činnost a sama od sebe předá řízení zpět systému. Úlohy běží v tzv. módu cyklické obsluhy. Každá úloha inkrementuje svůj čítač a předává obsluhu dalšímu procesu.



Obrázek 5.6: Spolupracující úlohy.

- **Preemptivní přepínání kontextu** – Měření preemptivnosti probíhá jako v předešlém testu za pomocí pěti úloh. Každá z úloh má odlišnou prioritu. Všechny úlohy mimo úlohy s nejnižší prioritou jsou na začátku uspány. Úloha s nejnižší prioritou probudí úlohu s druhou nejnižší prioritou. Díky preemptivnosti plánovače se tato úloha dostane do stavu běžící a probudí úlohu s následující prioritou a tak dále až se provede úloha s nejvyšší prioritou. Každá z úloh při svém běhu inkrementuje čítač a uspí se. Až se vykonávání vrátí úloze s nejnižší prioritou, inkrementuje svůj čítač a znovu probudí úlohu s následující vyšší prioritou - běh se opakuje znovu.

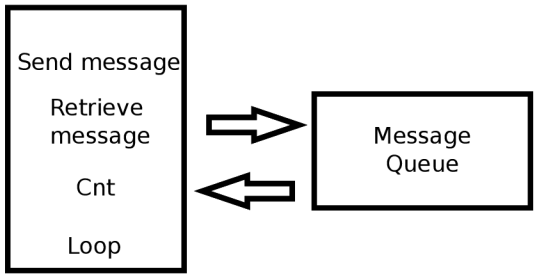


Obrázek 5.7: Preemptivní přepínání kontextu.

- **Zpracování přerušení** – Musíme vzít v úvahu dvě složky. Zpoždění přerušení - Jak dlouho bylo přerušení zakázáno. Režie aktivace úlohy - za jak dlouhou dobu je úloha schopna odpovědět. Nízká odezva přerušení, ale opožděná aktivace úlohy - Může být služba RTOS volána z obsluhy přerušení? Dvoúrovňová obsluha přerušení - nízké úrovně přerušení jsou určeny pro přerušení vyvolaná z vnějšku (pro odpovědi HW) a vysoké úrovně určené pro úlohy/obsluhy přerušení volající služby RTOS. Jedna úloha zpracovává tyto požadavky, ostatní úlohy využívají softwarové přerušení pro spuštění preempce. Úloha s nejvyšší prioritou "vytiskne"výsledky a uspí se. Úloha 5 inkrementuje čítač a vyvolá přerušení, v obsluze se opět inkremetuje čítač a probudí úlohu s vyšší prioritou než má úloha 5.

- **Zpracování zpráv** – V tomto měření je použita jedna úloha, která zasílá zprávu do fronty zpráv a ta samá úloha zprávu zároveň i přijímá. Po dokončení těchto komplementárních funkcí je inkrementován čítač.

- **Zpracování semaforů** – Semafory jsou systémovými zdroji starající se o garanci exklusivního přístupu ke kritickým systémovým zdrojům a synchronizaci asynchronních činností. Více o semaforech v sekci zabývající se komunikačními a synchronizačními



Obrázek 5.8: Zasílání zpráv.

prostředky 2.4.3. Stejně jako u předešlého testu i zde je pouze jedna úloha, která získává a uvolňuje získaný semafor. Pro zaznamenání počtu iterací je opět po těchto operacích inkrementován čítač v těle smyčky úlohy.

- **Alokace paměti** – Některé RTOS jsou vybaveny dynamickou alokací paměti pro znovupoužití paměti a tím i vyhnutí se problému s určením maximální velikosti paměti potřebné pro všechny úlohy v systému. Tento test se skládá z alokace 128 bytů paměti a následného uvolnění tohoto bloku paměti. Opět je použit čítač, který je inkrementován po provedení těchto dvou systémových volání.

5.1.3 SSC benchmark

Tento benchmark se zaměřuje na otestování dob trvání běžně používaných služeb jádra [8]. Obsahuje následující testy:

- **Create/Delete Task** – Úloha s nižší prioritou vytvoří novou úlohu s vyšší prioritou. Nově vytvořená úloha má vyšší prioritu, proto je po vytvoření spuštěna. Úloha nedělá nic, pouze sama sebe ukončí. Po jejím ukončení dojde k návratu do předchozí úlohy.
- **Ping Suspend/Resume Task) uspáním/probuzením úlohy** – Úloha s nižší prioritou probudí uspanou úlohu s vyšší prioritou. Tato úloha se po svém spuštění ihned sama uspí a dojde k návratu k úloze, která ji probudila.
- **Suspend/Resume Task** – Úloha s vyšší prioritou uspí a hned zase probudí níže prioritní úlohu.
- **Ping Mutex** – Dvě úlohy se stejnou prioritou střídavě zamykají a uvolňují semafor/mutex.
- **Get/Release Mutex** – Jedna úloha zamkne a ihned odemkne mutex.
- **Fill/Empty Queue** – Úloha naplní a potom zase vyprázdní frontu zpráv. Po dobu provádění musí být přerušení zakázána.
- **Queue Fill/Drain** – Úloha odešle sama sobě zprávu a ihned poté ji přijme.
- **Ping Fill/Drain** – Dvě úlohy mezi sebou komunikují pomocí dvou front zpráv. První úloha odešle pomocí první fronty zprávu druhé úloze, která ji přijme. Potom druhá úloha přijmutou zprávu odešle přes druhou frontu zpráv úloze první.

- **Allocate/Deallocate Memory** – Měření času potřebného k alokaci/dealokaci paměti.
- **Time Calls** – Zavolání funkce pro získání systémového/hardwarového času.
- **Ping Event** – Dvě úlohy se stejnými prioritami čekají na událost generovanou úlohou s nižší prioritou.
- **Event Pending** – Úloha s vyšší prioritou čeká na událost generovanou úlohou s nižší prioritou.
- **Set Event** – Měří čas potřebný k nastavení události.

5.2 Testování robustnosti RTOS

Tyto testy se používají v takových odvětvích, kde potřebujeme stabilní systémy za všech možných okolností. Tyto systémy se musí vypořádat jak se změnou vnitřní struktury systému, tak i s nečekanými změnami svého okolí.

5.2.1 CRASHME

Principem metody je zápis náhodných hodnot na libovolné místo do paměti. Tím dochází ke změnám některých instrukcí ve funkcích vykonávajícího programu. Mohou se změnit, jak funkce systémové, tak i uživatelské programy. Metoda spoléhá na to, že všechny funkce jsou volány se stejnou pravděpodobností. Pozměněné funkce způsobují zhroucení systému.

5.2.2 Fuzz

Tato metoda také zapisuje libovolné hodnoty do paměti, ale omezuje se pouze na specifické části systému a rozhraní. Tím se odlišuje od předchozí metody, která používá zcela náhodný přístup.

5.2.3 Ballista

Testuje rozhraní systému zasíláním, jak validních tak nedefinovaných dat.

Kapitola 6

Sada implementovaných testovacích úloh

Na základě informací z předchozí kapitoly byly vybrány testovací úlohy zaměřující se na doby trvání jednotlivých systémových funkcí. Tato sada úloh se v následujícím textu označuje jako nízkoúrovňový test.

Pro otestování výkonnosti jádra μC/OS-II na různých hardwarových platformách byl zvolen benchmark Thread-Metric 5.1.2. Další testovanou vlastností bylo zjištění maximální frekvence systémových hodin, při které je systém schopen běžet a korektně zpracovávat úlohy. Maximální frekvence se zjišťuje na sadě úloh Thread-Metric benchmarku.

RT aplikace potřebují reagovat na okolní podněty, pro generování těchto podnětů je pro jednoduchost vhodné zvolit čítač/časovač nacházející se na mikrokontroléru. Ten simuluje vznik vnějších periodických událostí. Pro otestování vlivu různé priority přerušení od vnějších událostí byly zvoleny dvě varianty testu. Jedna varianta generuje přerušení s vyšší prioritou než je priorita přerušení generátoru systémových hodin, druhá varianta má naopak prioritu nižší.

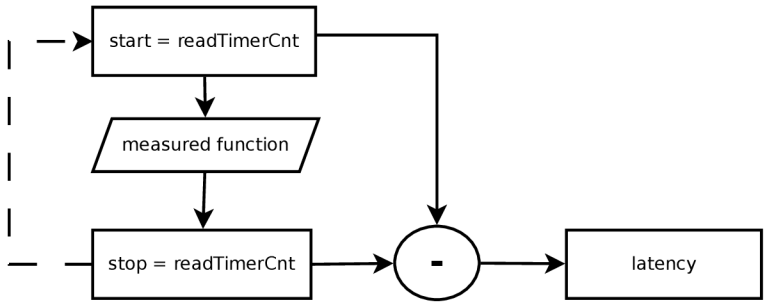
6.1 Způsob měření

Pro vyhodnocení testů jsou použity následující metriky.

6.1.1 Doba trvání

Měření času jednotlivých funkcí lze realizovat více způsoby. Jedním z nich je využití I/O portů. Před spuštěním měřeného úseku kódu se změní hodnota na výstupním portu, to samé se provede po ukončení měřeného úseku. K takto měněnému portu můžeme připojit osciloskop a sledovat doby mezi změnami logických hodnot na portu. Výhodou takto změřených časů může být jemnější časové rozlišení a tím i dosažení vyšší přesnosti. Nevýhodou těchto měření je latence způsobená nastavováním portu a projevení této změny na výstupu.

Další možností je použití vestavěného čítače/časovače, způsob měření je znázorněn na následujícím obrázku 6.1. Na začátku měřeného kódu odečteme hodnotu čítače, to samé uděláme na konci měřeného úseku. Latence bude rozdíl těchto hodnot. Výhodou tohoto způsobu měření je, že odpadá nutnost použití externích přístrojů a odpadá latence změny logické hodnoty výstupního portu. Nevýhoda v tomto případě může být horší rozlišení času než u osciloskopu.



Obrázek 6.1: Měření doby trvání.

6.1.2 Čítač událostí

Čítače událostí se používají v Thread-Metric benchmarku. Výsledná hodnota jednotlivých úloh je dána součtem všech čítačů událostí za určitý čas. Čítače jsou zvoleny tak, aby rozdíl mezi jednotlivými čítači nebyl větší než jedna. Tato vlastnost slouží k otestování správnosti vykonání testu.

6.2 Implementace

Pro implementaci byl zvolen programovací jazyk C [9]. Tato volba byla dána implementací jádra μC/OS-II. Pro portaci systému a implementaci obslužných rutin přerušení bylo potřeba použít jazyk symbolických instrukcí.

Jako vývojové prostředí byl na základě zkušeností autora s vývojovým prostředí Eclipse zvolen CodeWarrior (CW) ve verzi 10.2, který je na základech Eclipse postaven. Obě portace jádra μC/OS-II byly vytvořeny pro starší verzi CW konkrétně 6.2. Bohužel nešlo tyto projekty přímo importovat do novějšího CW, takže se musel pro každý mikrokontrolér vytvořit nový projekt a ten poté podle původního prostředí nastavit. Kromě toho se upravovala velikost zásobníku v konfiguračním souboru linkování.

U 32-bitového mikrokontroléru nastal problém v nastavení způsobu předávání parametrů při volání funkce. V novém vývojovém prostředí zmizela v konfiguraci překladače tato položka. Nově se implicitně používá předávání parametrů přes registry. Pokud chce programátor předávat funkci parametry přes zásobník, musí to pomocí makra umístěného v definici funkce sdělit. Bylo potřeba upravit port pro 32-bitový mikrokontrolér, který počítal s implicitním nastavení parametrů předávaných přes zásobník.

6.2.1 Nízkoúrovňové testy

Pro implementaci těchto testů se využívá obvykle jedna úloha, která spouští cyklicky jednotlivé testy. Způsob měření je znázorněn na obrázku 6.1.1. Pouze pro měření doby preempce a změny kontextu je použito druhé úlohy. Nejdříve se změří testy nepotřebující druhou úlohu a pro potřeby složitějších úloh je poté v systému vytvořena druhá úloha.

Výsledky jednotlivých testů jsou ukládány do struktury. Která může být na konci každé iterace použita pro výpis výsledků například pomocí serivoé linky.

Pokud dojde během měření k přetečení čítače, tak se tato událost zaznamená do proměnné uchovávající počty přetečení. U většiny testů k této věci nedochází, ale je potřeba vědět zda k takové události vůbec došlo.

Pro nízkoúrovňové testy byly vybrány úlohy, které se nevyskytují v Thread-Metrix benchmarku. Jedním z testů je i doba startu systému měřená od restartu mikrokontroléru. Skládá se z inicializace periferií, vytvoření jedné úlohy až po samotné spuštění této úlohy. Dalším testem je čas potřebný k vytvoření nové úlohy za běhu systému. Další měření se snaží nasimulovat přepnutí kontextu. Úlohas vyšší prioritou se sama uspí a plánovač obnoví běh úlohy s menší prioritou. Pro změření času preempce je níže prioritní úlohou obnovena úloha s prioritou vyšší. V obou těchto testech se nachází navíc režie spojená s uspáváním a probouzením úloh.

6.2.2 Thread-Metric benchmark

Při implementaci této sady testů byl použit upravený kód firmy Express Logic, Inc dostupný na [6]. Archiv obsahuje implementované samotné testy a podobně jako je tomu u jádra μC/OS-II soubor **tm_porting_layer.c** usnadňující portaci benchmarku pro různé systémy. Z důvodu nedostatku paměti RAM bylo pole v základním testu zmenšeno na polovinu proti původní velikosti 1024 položek.

Portace pro jádro μC/OS-II si ale vyžádala zásahy i do ostatních zdrojových souborů. Bylo potřeba upravit funkce pracující se semafory, frontou zpráv a s úlohami (uspávání, probouzení).

Řešení problému stejných priorit

Dále byl potřeba vyřešit problém s přiřazením stejných priorit několika úlohám. Tuto možnost μC/OS-II nepodporuje. Nakonec toho bylo dosaženou úpravou funkce **tm_thread_relinquish**. Tato funkce nahrazuje původní plánovač. Je využito toho, že jednotlivé funkce si předávají řízení cyklicky mezi sebou, tak je při volání této funkce předán ukazatel na TCB následující úlohy. Funkce musí ale ještě hlídat úlohu s nejvyšší prioritou, sloužící pro výpis výsledku běhu testů. Tato úloha je probouzena jednou za 30 vteřin. Funkce tedy musí zjistit stav této úlohy a zavolat ji, pokud je úloha připravena k běhu.

Vyvolání přerušení

ěkteré testy potřebují pro svůj běh vyvolat softwarové přerušení. Na 8-bitovém mikrokontroléru se nachází jediné softwarové přerušení a to je použito pro přepínání kontextu mezi úlohami. Proto bylo využito pro emulaci přerušení vnější přerušení IRQ. Aby testy využívající toto přerušení fungovaly správně, je nutné propojit na patici vývojové desky piny PTB1 a PTA5.

Alokace paměti

Jádro μC/OS-II nepodporuje dynamickou alokaci paměti, z toho důvodu nebyl tento test implementován.

6.3 Logování výsledků

Vzhledem k problémům při ladění portů a následně i testů nedošlo k implementaci funkcí, které by výsledky testů zasílali pomocí seriového rozhraní.

Kapitola 7

Vyhodnocení testů

Tato kapitola je zaměřena na vyhodnocení výsledků provedených testů na 8-bitovém a 32-bitovém mikrokontroléru. Veškeré zde uvedené výsledky byly získány pomocí ladící části vývojového prostředí. Test byl přerušen v úloze, která měla kontrolovat správnost výsledků testu a zasílat výsledky testů pomocí seriové linky.

7.1 Nízkoúrovňové testy

Hodnoty naměřené v tabulkách odpovídají počtu tiků čítače umístěného na mikrokontroléru. Jeho frekvence je polovinou frekvence mikrokontroléru. Pro objektivní výsledky testů jsou oba mikrokontroléry nastaveny na stejnou frekvenci.

Podle výše uvedené tabulky je start systému u 32-bitového mikrokontroléru o polovinu než u 8-bitového. Rozdíl mezi preempcí a přepnutím kontextu není moc velký. K těmto dobám je také nutné přičíst dobu potřebnou pro uspání a probuzení úloh. Největší rozdíl při porovnání obou mikrokontrolérů je ve vytváření úlohy. Při volání této funkce dochází k inicializaci datových struktur jak například zásobníku úlohy.

7.2 Thread-Metric

Pro komplexní otestování jádra byl vybrán tento test. V tabulce 7.2 jsou zobrazeny výsledky jednotlivých testů. Jedná se o hodnotu jednoho čítače použitého v rámci dotyčného testu. Ostatní čítače nabývají hodnoty lišící se maximálně o jedna.

Úpravy této sady testů pro μC/OS-II byly popsány v sekci 6.2.2 věnující se implementaci. Například test spolupracujících úloh byl implementován specificky pouze pro tento test.

Testy	**Mikrokontrolér**	
	HCS08	ColdFire V1
Kritická sekce	49	36
Zavedení systému	x 124 086	64 056
Přepnutí kontextu	771	263
Doba preempce	761	253
Vytvoření úlohy	17 566	1 797

Tabulka 7.1: Výsledky měření nízkoúrovňových testů

Testy	Mikrokontrolér		Zrychlení
	HCS08	ColdFire V1	
Základní	2877	xxx	xxx
Spolupracující úlohy	222177	xxxx	xxx
Preemptivní přepínání kontextu	101992	356541	3,5
Zpracování přerušení	416876	1026725	2,46
Přerušení s preempcí	333234	980009	2,94
Zprávy	xxx	xxx	
Semafory	1521931	4442300	2,92

Tabulka 7.2: Výsledky měření Thread-Metric testů.

HCS08	ColdFire V1
4000	37500

Tabulka 7.3: Maximální frekvence systémového tiku.

Pro porovnání vlastností mikrokontrolérů, což je náplní této práce je taková implementace dostačující. Ale nelze takto získané hodnoty porovnat s jinými RTOS jádry [7]. Z výsledků jednotlivých testů vyplývá, že mikrokontrolér s jádrem ColdFire V1 je schopen zpracovávat 3 krát více požadavků než jednoduší mikrokontrolér s jádrem HCS08. Nejmenší rozdíl mezi mikrokontroléry byl ve schopnosti zpracovávat přerušení to lze vysvětlit složitějším přerušovacím podsystémem 32-bitového mikrokontroléru. Největšího rozdílu bylo dosaženo u testu preemptivního plánování, který je zaměřen spíše na porovnání instrukčních sad. Lepšího výsledku by mohla dosáhnout implementace portace systému μC/OS-II používající registry.

Testy u nichž nejsou výsledky se na daném mikrokontroléru nepodařilo spustit nebo jejich výstup nebyl správný, někdy se je dokonce nepodařilo ani nahrát na mikrokontrolér.

7.3 Maximální frekvence systémových hodin

Tento test je zaměřen na otestování instrukční sady společně s datovou propustností systému. Maximální frekvence se zjišťovala na úlohách Thread-Metric 7.2 testů. Podmínkou bylo úspěšné vykonání testů a dodržení reportování výsledků každých 30 vteřin. S nekoretní dobou spouštění úlohy pro výpis výsledků se potýkalo hlavně jádro HCS08. Z tabulky 7.3 je vidět, jak velký je rozdíl maximálních frekvencí systémových hodin na jednotlivých mikrokontrolérech. Tento rozdíl neodpovídá výsledkům naměřeným v Thread-Metric 7.2. Limitujícím faktorem v tomto testu je optimalizace instrukcí jádra HCS08 na zpracování dat o velikosti 1 byte.

Experimentálně bylo zjištěno, že pro testovanou konfiguraci systému je pro 8-bitový mikrokontrolér hodnota počet tiků za vteřinu 4000. Tato hodnota odpovídá 250 ms mezi systémovými tiky. Frekvence mikrokontroléru je 50 Mhz tzn. jednou za 12583 tiků procesoru dojde k přepnutí kontextu.

Kapitola 8

Závěr

Navržené testy byly zaměřeny na rychlost vykonání jak samostatných služeb μC/OS-II, tak i otestování rychlosti komplexnějších úloh. Například při měření doby potřebné pro vytvoření úlohy byl vidět výraznější rozdíl v rychlosti práce se systémovými proměnnými. Výkonnější mikrokontrolér je schopen běžet na vyšší systémové frekvenci a v důsledku toho je vhodný pro náročnější RT aplikace. Oba mikrokontroléry mají deterministické chování tj. nemají žádnou instrukční cache ani neumožňují zřetězené zpracování. To zaručuje pro opakované volání stejného bloku kódu stejnou dobu vykonání. Mikrokontroléry mají doby trvání jednotlivých funkcí μC/OS-II odlišné.

Tato práce byla zaměřena hlavně na dobu odezvy. Podle očekávání měl obecně kratší odezvy 32-bitový mikrokontrolér. Je to dáno jeho odlišnou architekturou a tedy i instrikční sadou, ale také širší datová sběrnice umožňující přenášet větší množství dat. Toto jádro podle získaných časů je vhodné pro náročnější hard RT aplikace s požadavkem na krátké doby odezvy. Pomalejší 8-bitový mikrokontrolér se hodí spíše na RT aplikace, kde nejsou kladeny takové nároky na dobu odezvy.

Nepovedli se implementovat všechny testy Thread-Metric a bohužel nezbyl čas na implementaci zasílání výsledků přes seriovou linku na terminál.

Pokračováním této práce by mohlo být rozšíření testů například o testování robustnosti systému a nebo testování spotřeby jednotlivých mikrokontrolérů.

Literatura

[1] *µC/OS-II and µC/Probe on the Freescale MC9S08QE128* [online]. http://micrium.com/page/downloads/ports/freescale/8-16_bits.

[2] *µC/OS-II and µC/Probe on the Freescale MCF51QE128* [online]. http://micrium.com/page/downloads/ports/freescale/32_bits.

[3] *Commercial Real-Time Operating Systems.* http://nptel.iitm.ac.in/courses/Webcourse-contents/IIT%20Kharagpur/Real%20time%20system/pdf/module5.pdf, [cit. 2011-12-10].

[4] *LINPACK* [online]. http://www.netlib.org/linpack/, [cit. 2012-01-09].

[5] *The SPEC Benchmarks* [online]. http://msdn.microsoft.com/en-us/library/ms834456.aspx, [cit. 2012-01-09].

[6] *Thread-Metric Benchmark Suite* [online]. http://rtos.com/images/uploads/Thread_Metric.zip, [cit. 2013-05-15].

[7] *AVIX-RT: Performance* [online]. http://www.avix-rt.com/Products/Performance/performance.html, [cit. 2013-05-16].

[8] Gräbner, O.; Küfner, H.; Stierschneider, O.; aj.: *Assessing Microsoft Windows CE 3.0 Real-Time Capabilities* [online]. http://msdn.microsoft.com/en-us/library/ms834456.aspx, [cit. 2011-12-10].

[9] Herout, P.: *Učebnice jazyka C.* České Budějovice: Kopp, 2009, ISBN 9788072323838.

[10] Kukal, J.: *Robustnost.* http://www.automatizace.cz/article.php?a=2285, 2008.

[11] Labrosse, J.: *MicroC/OS-II: the real-time kernel.* CMP Books, 2002, ISBN 9781578201037.

[12] Laplante, P. A.: *Real-time systems design and analysis.* John Wiley & Sons, 2004, ISBN 9780471228554.

[13] Strnadel, J.: *Real-time operační systémy (ROS)* [online]. 2006-10-10 [cit. 2011-12-10].

[14] W. A. Halang, e. a.: *Measuring the Performance of Real-Time Systems.* 2000.

Příloha A

Obsah CD

- 8bit_benchmark – testy pro jádro HCS08
- 32bit_benchmark – testy pro jádro ColdFire
- text – složka se zdrojovými soubory tohoto textu
- info.txt – popis spuštění jednotlivých testů