ZÁPADOČESKÁ UNIVERZITA V PLZNI
FAKULTA ELEKTROTECHNICKÁ

Katedra aplikované elektroniky a telekomunikací

DIPLOMOVÁ PRÁCE

Analýza a porovnání RTOS dostupných pro mikrokontroléry řady STM32

Bc. Ondřej Štok **2014**

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta elektrotechnická
Akademický rok: **2013/2014**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Ondřej ŠTOK**
Osobní číslo:	**E12N0058P**
Studijní program:	**N2612 Elektrotechnika a informatika**
Studijní obor:	**Elektronika a aplikovaná informatika**
Název tématu:	**Analýza a porovnání RTOS dostupných pro mikrokontroléry řady STM32**
Zadávající katedra:	**Katedra aplikované elektroniky a telekomunikací**

Zásady pro vypracování:

1. Prostudujte problematiku operačních systémů reálného času (RTOS), požadované nároky na jejich parametry a funkce.
2. Seznamte se s mikrokontroléry řady STM32. Prostudujte možnosti a parametry dostupných mikrokontrolerů řady STM32 z hlediska jejich vhodnosti pro použití s RTOS.
3. Prostudujte možnosti a parametry vybraných RTOS portovaných pro zvolený mikrokontrolér řady STM32.
4. Navrhněte a vhodně implementujte abstrakční vrstvu pro vybrané RTOS.
5. Navrhněte a realizujte testovací aplikaci vhodnou pro srovnání jednotlivých vybraných RTOS z hlediska jejich kritických parametrů.

Na základě teoretických podkladů a výsledků testů proveďte srovnání použitých RTOS.

Rozsah grafických prací: **podle doporučení vedoucího**

Rozsah pracovní zprávy: **30 - 40 stran**

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

Seznam odborné literatury:

Student si vhodnou literaturu vyhledá v dostupných pramenech podle doporučení vedoucího práce.

Vedoucí diplomové práce: **Ing. Petr Krist, Ph.D.**
Katedra aplikované elektroniky a telekomunikací

Datum zadání diplomové práce: **14. října 2013**

Termín odevzdání diplomové práce: **12. května 2014**

Doc. Ing. Jiří Hammerbauer, Ph.D.
děkan

Doc. Dr. Ing. Vjačeslav Georgiev
vedoucí katedry

V Plzni dne 14. října 2013

Anotace

Diplomová práce se zabývá operačními systémy reálného času RTOS, jež jsou dostupné pro mikrokontroléry STM32 a jejich porovnáním. Porovnání je provedeno na základě výsledků získaných z realizovaných aplikací. V práci je vysvětlena základní teorie RTOS včetně způsobu návrhu ovladačů I/O. Dále jsou v ní identifikovány vlastnosti STM32, jimiž je podporován běh RTOS a je zde popsán způsob porovnávání RTOS Rhealstone. Na konec jsou zde popsány aplikace použité pro porovnání RTOS. Cílem této práce je zjistit důležité vlastnosti RTOS a následně na základě těchto zjištění porovnat vybrané operační systémy dostupné pro mikrokontroléry STM32.

Klíčová slova

RTOS, STM32, mikrokontrolér, Rhealstone

Abstract

The diploma thesis is focused on real-time operating systems RTOS available for STM32 microcontroller family and their comparison. The comparison is based on results gained from realized applications. Basic theory of the RTOS including design of I/O drivers is explained in the thesis. Attributes of the STM32 used by the RTOS are identified. Then Rhealstone benchmarking for the RTOS is covered. Finally the applications used for the benchmarking are described. Thesis objective classifies important attributes of the RTOS and then makes comparison of chosen RTOS available for STM32.

Key words

RTOS, STM32, microcontroller, Rhealstone

Prohlášení

Předkládám tímto k posouzení a obhajobě diplomovou práci, zpracovanou na závěr studia na Fakultě elektrotechnické Západočeské univerzity v Plzni.

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně, s použitím odborné literatury a pramenů uvedených v seznamu, který je součástí této diplomové práce.

Dále prohlašuji, že veškerý software, použitý při řešení této diplomové práce, je legální.

V Plzni dne 4.5.2014 Ondřej Štok

……..

Poděkování

Tímto bych rád poděkoval vedoucímu diplomové práce Ing. Petru Kristovi, Ph.D. a konzultantovi Ing. Zdeňku Nývltovi z ST Microelectronics za cenné profesionální rady, připomínky a metodické vedení práce.

Obsah

Úvod

Předkládaná práce se zaměřuje na porovnání a analýzu operačních systémů reálného času RTOS, které jsou dostupné pro mikrokontroléry STM32.

Motivace pro výběr tohoto tématu diplomové práce byla následující. Z důvodu neustálého zvětšování výpočetní síly i jednoduchých mikročipů dochází v dnešní době k používání RTOS i v aplikacích, jež jsou dostupné běžnému člověku. Vzhledem k jejich množství vzniká nutnost určit, který RTOS je pro zvolenou aplikaci nejvhodnější.

V první kapitole je definován pojem RTOS, jeho základní vlastnosti a jak se odlišuje od klasických operačních systémů. Následně je rozebráno jádro RTOS. To zahrnuje vysvětlení plánovače, jež přiděluje procesorový čas a popis základních objektů. Následuje vysvětlení důležitých vlastností RTOS, to jak je zde realizováno přerušení a jak je vyřešena kritická část programu. Dále jsou uvedeny dva nejčastější problémy při realizaci aplikací a jak je RTOS řeší. Na konec je popsána architektura ovladačů I/O periferií v prostředí RTOS.

Druhá kapitola se zabývá vlastnostmi, jimiž jsou mikrokontroléry STM32 vhodné pro použití s RTOS. Nejdříve jsou vyjmenovány základní vlastnosti STM32. V této části je vysvětlení způsobu, jakým instrukční sada Thumb 2 napomáhá ke zrychlení a determinismu programu. Následuje popis řadiče NVIC a jak probíhá přerušení na jádrech Cortex M-3 a M-4. Dále je popsáno využití čítače SYSTICK a výjimky jádra PendSV v RTOS. Nakonec je zmíněna jednotka MPU a přístup k jednotlivým bitům pomocí Bit-Bandu.

Ve třetí kapitole se práce věnuje výběru vhodných RTOS. Dojde zde ke zmínění dostupných RTOS. V této kapitole je uveden výčet kritérií, podle nichž došlo k výběru porovnávaných RTOS CoOS a FreeRTOS. Vlastnosti těchto dvou systémů jsou následně popsány detailněji.

V následující čtvrté kapitole je popsáno co a jak je možné v RTOS měřit. Jedná se především o měření latence ve vykonávání funkcí systému a latence přerušení. Je zde také zmíněno měření využití paměti jednotlivými systémy. Konec kapitoly se zaměřuje na způsob porovnávání RTOS Rhealstone.

V páté kapitole jsou definovány výsledné testovací aplikace a platforma, na níž probíhá testování.

V poslední kapitole jsou zobrazeny výsledky z měření. Ty jsou v závěru zhodnoceny.

Cílem této diplomové práce je na základě zjištěných důležitých vlastností porovnat zvolené RTOS.

Seznam symbolů

RTOS	[]……..	Operační systém reálného času (Real-Time Operating System)
TCB	[]……..	Řídící blok úlohy (Task Control Block)
SCB	[]……..	Řídící blok semaforu (Semaphor Control Block)
QCB	[]……..	Řídící blok fronty zpráv (Queue Control Block)
FIFO	[]……..	První dovnitř první ven (First In First Out)
LIFO	[]……..	Poslední dovnitř první ven (Last In First Out)
ISR	[]……..	Obslužná rutina přerušení (Interupt Service Routine)
NVIC	[]……..	Vestavěný vektorový řadič přerušení (Nested Vectored Interrupt Controler)
NMI	[]……..	Nemaskovatelné přerušení (Non-Maskable Interupt)
IRQ	[]……..	Požadavek na přerušení (Interupt Request)
MPU	[]……..	Jednotka ochrany paměti (Memory Protection Unit)
LSB	[]……..	Nejméně významný bit (Last Significant Bit)
GPIO	[]……..	Obecně použitelné vstupně výstupní piny (General-Purpose Inputs Outputs)
MEMS	[]……..	Mikro elektricko-mechanické systémy (Micro-Electro-Mechanical Systems)
USB	[]……..	Universální sériová sběrnice (Universal Serial Bus)
LED	[]……..	Světelná dioda (Light-Emitting Diode)
SPI	[]……..	Sériové periferní rozhraní (Serial Peripheral Interface)
UART	[]……..	Universální asynchronní přijímač a vysílač (Universal Asynchronous Receiver and Transmitter)

1 Operační systém reálného času RTOS

Operační systémy reálného času, dále jen RTOS (Real-Time Operating System), jsou dnes velmi často používány v mnoha embedded (specializovaných) aplikacích. Tím nejsou myšleny jen aplikace používané v armádě, vědě nebo lékařství, ale dnes se s nimi lze setkat i v domácnostech a běžném životě.

1.1 Vlastnosti

Operační systémy RTOS se od běžně používaných operačních systémů (např. Windows) liší ve většině případů odlišnou cílovou platformou a hlavně svými vlastnostmi.

Zavádí totiž do operací determinismus. To znamená, že kromě správného provedení nějaké operace je důležitý i čas, za jaký se provede. Každá úloha v systému s RTOSem má časový úsek (angl. deadline), jehož překročení může mít různé následky. Záleží na aplikaci a jejích požadavcích, jaké a jak vážné následky budou. Od chvilkového rozpadnutí obrazu v DVD přehrávači až po havárii a možnou ztrátu na životech v případě autopilota. Od toho se odvíjí rozdělení aplikací využívající RTOS na Hard a Soft [1].

1.1.1 Hard RTOS

U Hard RTOS má nedodržení časového milníku pro systém, kterého se týká, kritický význam.

Například u již zmíněného autopilota. Systém, který kontroluje reliéf krajiny, musí data zpracovávat a odesílat systému řízení výšky dostatečně rychle. Systém řízení výšky pak musí na tyto data dostatečně rychle reagovat. Nedodržení jakéhokoliv milníku by znamenalo zřícení letadla.

Porušení časového milníku je u těchto systémů nepřípustné!

1.1.2 Soft RTOS

Zde jsou požadavky na dodržování časových milníků méně přísné a jejich nedodržení nemá kritické následky. V Soft systémech je možné akceptovat občasné nedodržení milníku.

Příkladem je DVD přehrávač, konkrétně dekódování obrazu. Pokud zde dojde k nedodržení milníku, nemusí divák zaznamenat žádný efekt nebo pouze zaznamená chvilkové snížení kvality obrazu.

Tyto systémy musí na překročení časového milníku správně reagovat.

1.2 Jádro RTOS

Každý RTOS je tvořen jádrem (angl. kernel). Toto jádro obsahuje plánovač, objekty

a služby. Objekty jsou myšleny úlohy, semafory a fronty zpráv. Tyto objekty jsou součástí každého RTOS. Některé RTOS obsahují ještě další objekty např. signály, roury (angl. pipes), událostní registry (angl. Event register) a další. Jádro dále poskytuje služby pro práci s těmito objekty, obsluhu přerušení, správu paměti, práci s I/O zařízeními a správu času. Jádro stejně jako v případě objektu může u různých RTOS obsahovat ještě další služby [1].

1.2.1 Plánovač

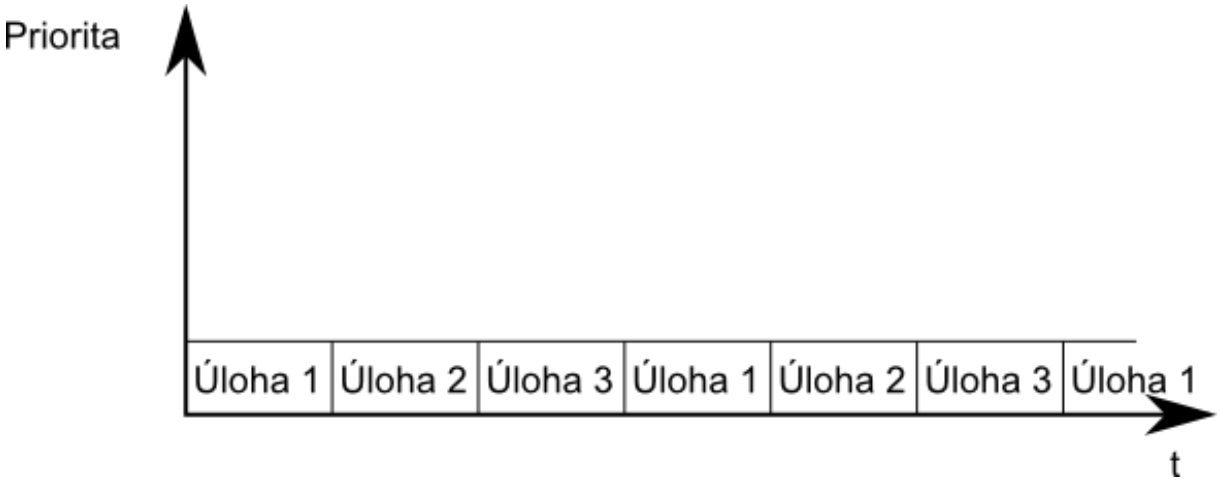
Ve většině případů obsahuje embedded systém pouze jeden mikrokontrolér s jedním jádrem. V jeden okamžik tak může běžet pouze jedna úloha (angl. task). Plánovač zajišťuje, aby běžela vždy ta správná. Plánovač se stará o přepínání úloh a ukládání jejich dat, tak aby mohly v případě, že na ně opět přijde řada, začít vykonávat svou funkci [1].

Existuje několik filozofií, jak plánovač vybírá úlohu, která se bude vykonávat. Následně jsou popsány tři nejběžnější.

1.2.1.1 Round Robin

Plánovač, používající Round robin k přidělovaní procesorového času úlohám, používá jednoduchou metodu. Každé úloze je přidělen časový úsek, kdy se vykonává. Tento úsek je většinou nastaven globálně na stejnou hodnotu. Některé RTOS kromě toho umožňují přidělit každé úloze, při jejím vytvoření, jiný časový úsek. Po uplynutí tohoto času plánovač přepne na další úlohu. Tento proces se neustále opakuje a úlohy se pravidelně střídají ve využívání procesoru [1].

Následující obrázek č. 1 znázorňuje příklad Round robin. Jsou zde tři úlohy, které se po určité době střídají ve využívání procesoru.



Obr. č. 1Plánovač s Round robin

Nevýhodou této filozofie je pomalá reakce na asynchronní akce. V případě výskytu

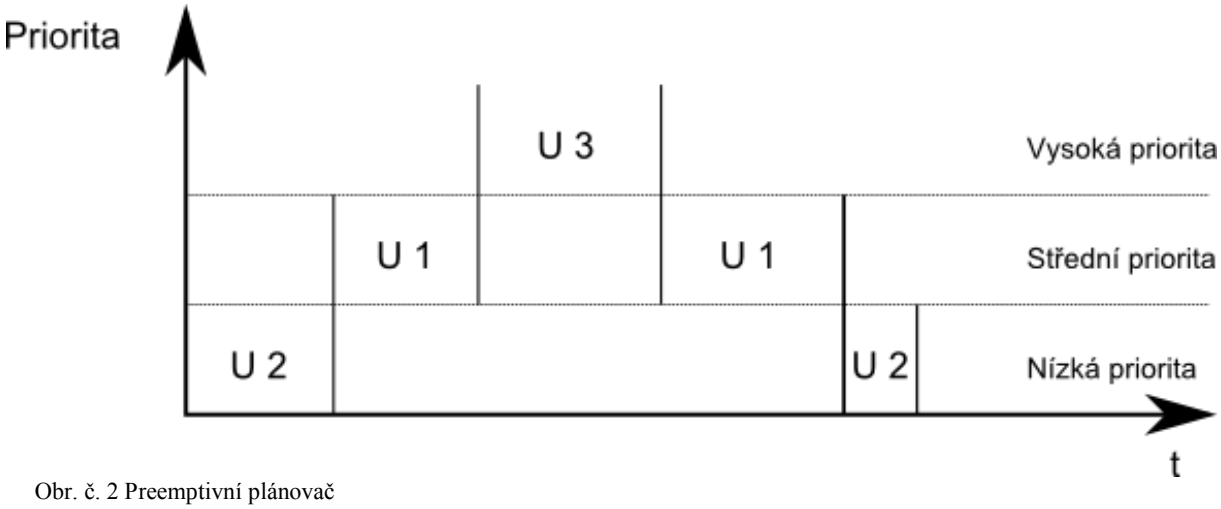
přerušení (průběh přerušení v systémech s RTOS bude vysvětlen dále) může dojít ke zpoždění v odezvě procesoru. Přerušení sice zastaví vykonávání právě probíhající úlohy, ale úloha, kterou odblokuje (kapitola 1.3), přijde na řadu nejdříve až po skončení časového úseku přiděleného přerušené úloze.

1.2.1.2 Preemptivní

Mnohem častěji je používán Preemptivní plánovač. Zde je namísto daného množství procesorového času určena každé úloze priorita. Prioritu dostane úloha při svém vytvoření.

Princip je pak následující. Vyskytne-li se úloha čekající na své vykonání s vyšší prioritou, než je priorita právě probíhající úlohy, plánovač přeruší vykonávání úlohy s nízkou prioritou a uloží její data do zásobníkové paměti. Poté načte data úlohy s vyšší prioritou a nechá ji vykonávat. Není-li tato úloha přerušena úlohou s ještě vyšší prioritou, pak po dokončení její činnosti, pokračuje ve své práci dříve přerušená úloha s nižší prioritou [1].

Na obrázku č. 2 jsou tři úlohy. Úloha 2 má nízkou, úloha 1 střední a úloha 3 nejvyšší prioritu. V průběhu vykonávání úlohy 2 ji přeruší úloha 1. Tu pak přeruší úloha 3. Po jejím skončení pokračuje úloha 1 a naposled je dokončena úloha 2.



Obr. č. 2 Preemptivní plánovač

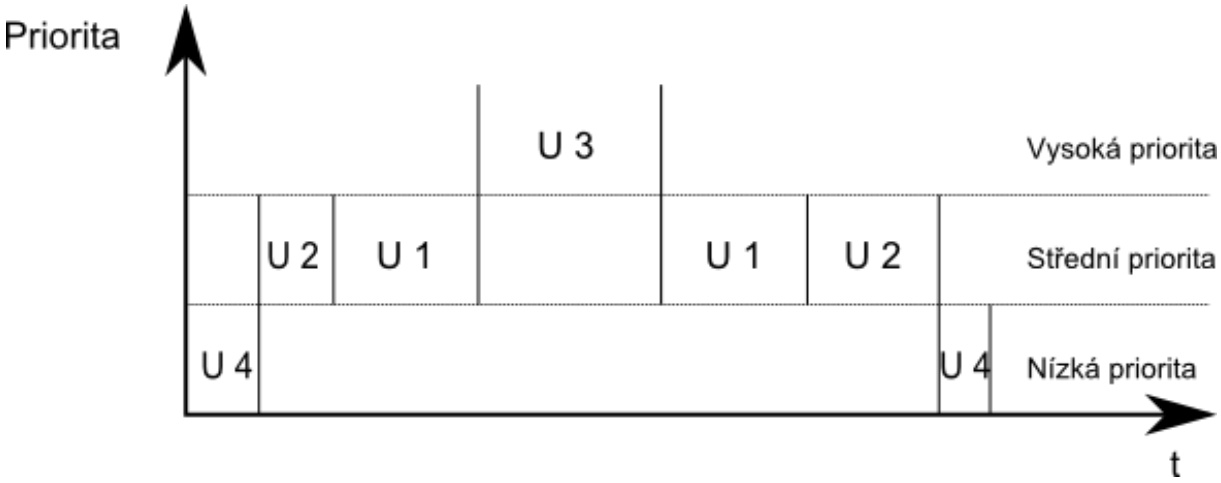
Preemptivní plánovač mnohem lépe reaguje na přerušení, než již zmíněný Round robin. Počet dostupných priorit závisí na zvoleném RTOS.

1.2.1.3 Kombinovaný

Jedná se o kombinaci dvou předchozích způsobů. Úlohám je opět přiřazena priorita. Úlohy se stejnou prioritou se, na rozdíl od čistě preemptivní filozofie, ve vykonávání střídají v rámci časových úseků, stejně jako u Round robin [1].

Obrázek č. 3 znázorňuje opět několik úloh s různými prioritami. Úlohy 1 a 2 sdílejí stejnou prioritu. Úloha 2 přeruší vykonávání úlohy 4. V průběhu jejího vykonávání se úloha 1

také stane připravenou běžet, vystřídá se s úlohou 2 po uplynutí časového úseku. Před skončením svého úseku je však přerušena úlohou 3 s vyšší prioritou. Úloha 1 poté pokračuje ve svém průběhu po skončení úlohy 3. Nakonec se pokračuje ve vykonávání úlohy 4.



Obr. č. 3 Kombinace Round robin a preemptivního plánovače

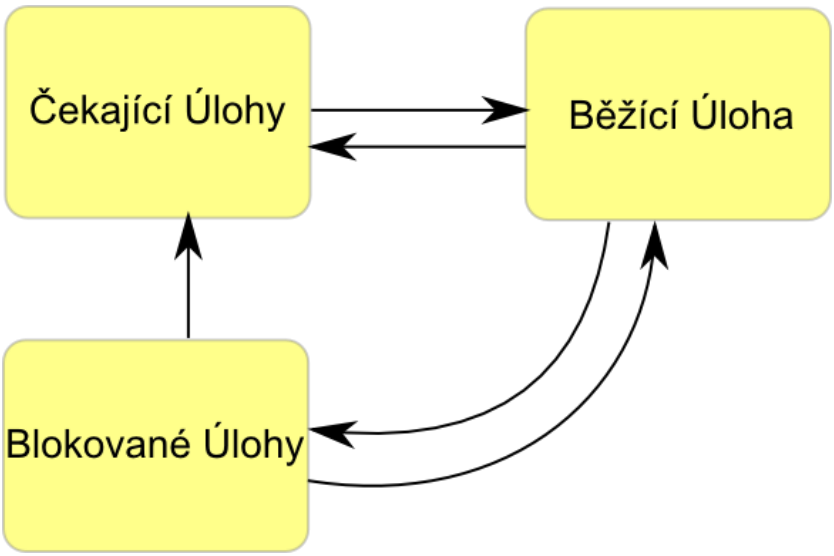
Mnoho RTOS dává na výběr mezi čistě preemptivním nebo kombinovaným plánovačem. Množství dostupných priorit také závisí na vybraném RTOS.

1.2.2 Objekty

1.2.2.1 Úlohy

Každá aplikace pracující pod RTOS se skládá z úloh. Každá úloha vykonává nějakou činnost. Při jejím vytvoření se spolu s ní vytvoří také řídící blok úlohy TCB (angl. Task Control Block). Úloze je přiřazena priorita, identifikátor a zásobníková paměť. V TCB jsou uloženy informace o úloze, jedná se obecně o její ID, prioritu, stav, odkaz na zásobník.

Každá úloha se během svého životního cyklu pohybuje v několika stavech. Jak ilustruje obrázek č. 4, úloha buď běží, čeká na své vykonávání nebo je blokovaná.



Obr. č. 4 Stavy úloh

Běžící úloha může být, v systému s jedním procesorovým jádrem, vždy pouze jedna a to úloha s nejvyšší prioritou. V případě Round Robin běží úlohy podle pořadí, v jakém byly vytvořeny. Tato úloha vykonává svůj programový kód. O tom, která úloha v danou chvíli poběží, rozhoduje plánovač.

Čekající úlohy jsou ty, které by chtěly běžet, ale mají nižší prioritu než právě vykonávaná úloha. Plánovač si uchovává seznam všech úloh v tomto stavu. Úlohy jsou v něm seřazeny podle jejich priority. Na prvním místě je úloha s nejvyšší prioritou mezi čekajícími úlohami, ale stále nižší prioritou než má úloha v běžícím stavu. Plánovač, v případě skončení právě probíhající úlohy, vybere tuto úlohu jako další pro přechod do běžícího stavu. Plánovač aktualizuje tento seznam, vždy když dojde k přepnutí úloh, vytvoření nové úlohy nebo k odblokování úlohy. Úlohy se po svém vytvoření nachází v tomto stavu.

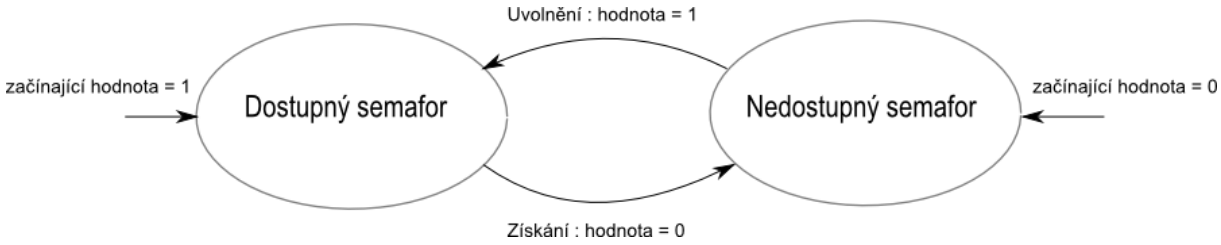
Posledním stavem, ve kterém se mohou úlohy nacházet, je blokování. Tyto úlohy se nevykonávají a ani se neucházejí o procesorový čas. Právě to umožňuje běh i úlohám s nízkými prioritami. Úloha může v blokovaném stavu zůstat po určitý časový úsek, čekat na nějakou akci (např. uvolnění semaforu) nebo kombinaci obou (úloha čeká na akci pouze po nějakou dobu).

I když programátor nenaprogramuje v aplikaci žádné úlohy, RTOS sám po spuštění plánovače, vždy jednu úlohu vytvoří. Nazývá se Idle Task. Idle Task se vytvoří i při existenci jiných úloh, přičemž má vždy nejnižší možnou prioritu. Tato úloha může být upravena programátorem a obsahovat jeho vlastní kód [1].

1.2.2.2 Semafory

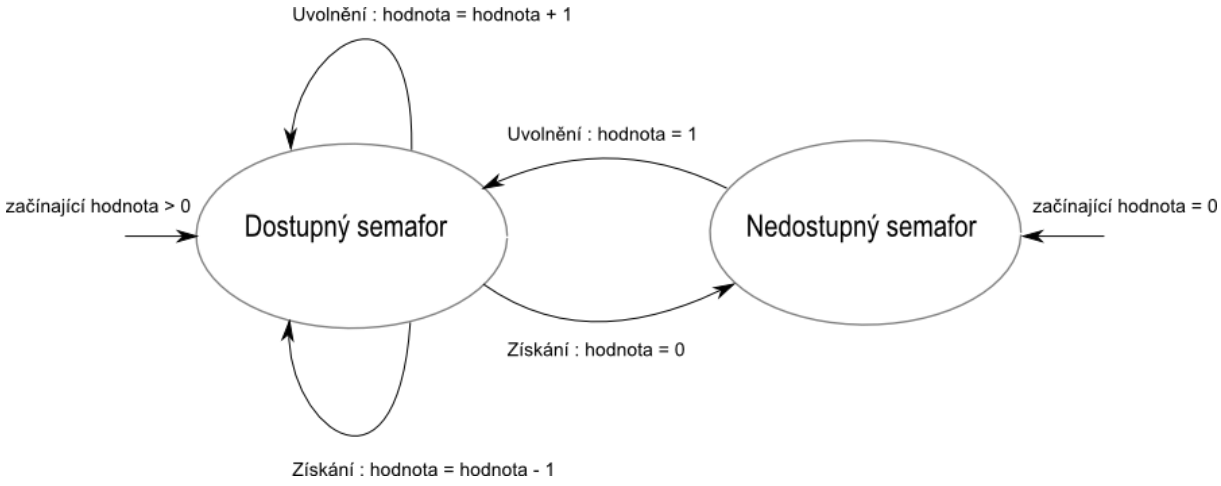
Dalším důležitým objektem RTOS jsou semafory. Ty slouží především ke dvěma účelům, a sice k synchronizaci mezi úlohami nebo přerušením a úlohou a k hlídání přístupu ke sdíleným zdrojům (paměti, periferiím). Při vytvoření semaforu se spolu s ním vytvoří, podobně jako v případě úlohy, řídící blok SCB (Semaphore Control Block) a přidělí se mu identifikátor. Na rozdíl od úlohy má semafor po vytvoření také přiřazenou hodnotu a seznam úloh, které na tento semafor čekají. Úlohy mohou být v seznamu seřazeny buď podle toho, v jakém pořadí se o semafor přihlásily (FIFO First In First Out) nebo podle jejich priorit. Existují tři hlavní druhy semaforů.

Binární semafor je nejjednodušší druh. Používá se převážně k synchronizaci úloh. Má pouze dva stavy, jak je vidět na obrázku č. 5. Semafor je buď dostupný, pokud je jeho hodnota rovna 1, nebo nedostupný při hodnotě 0. Pokud je nedostupný, může na něj úloha počkat v blokovaném stavu. Může na něj čekat pouze určitý čas, nebo dokud nebude semafor opět dostupný. Úloha se také objeví v seznamu úloh čekajících na tento semafor. Binární semafor se nepoužívá k přístupu ke sdíleným zdrojům, protože ho může uvolnit jakákoliv úloha. Tato úloha ho nemusela předtím získat [1]!



Obr. č. 5 Binární semafor

Dalším druhem semaforu je čítací (angl. counting) semafor. Tento semafor na rozdíl od binárního může být získán i uvolněn vícenásobně. Jeho hodnota se tedy může rovnat více než 1. Pokud je jeho hodnota rovna 0, je semafor nedostupný. Jak je patrné z obrázku č. 6, dochází při získání semaforu úlohou, stejně jako u binárního, k dekrementaci hodnoty. Není-li však tato hodnota rovna 0, zůstává semafor stále dostupný. Tento semafor může sloužit k čítání nějakých akcí, které aplikace provede, nebo k počítání dostupných sdílených zdrojů. V tomto případě je jeho počáteční hodnota rovna dostupným zdrojům. Vždy, když úloha zabere nějaký zdroj, hodnota semaforu se sníží o 1. Vzhledem k tomu, že opět může semafor uvolnit i úloha, která ho nezískala, je v tomto případě nutná spolupráce se semaforem typu mutex [1].



Obr. č. 6 Čítací semafor

Posledním semaforem je semafor typu mutex (z angl. Mutual Exclusion). Jedná se vlastně o binární semafor. Mutex se však od něj liší v jedné hlavní věci - podporuje vlastnictví. V případě binárního semaforu ho mohla uvolnit jakákoliv úloha, to se u mutexu stát nemůže. Úloha, která ho získala, ho také musí uvolnit. Tato vlastnost se velmi hodí při přístupu ke sdíleným zdrojům. S každým takovým zdrojem, ať už s pamětí nebo periférií, je spojen vždy určitý mutex. Pokud o daný zdroj zažádá úloha a daný mutex je dostupný, úloha získá jeho vlastnictví a přístup ke zdroji. Pokud by o ten samý zdroj požádala jiná úloha, tak kvůli nedostupnému mutexu přístup nezíská a může se rozhodnout na daný mutex počkat. Po skončení práce se zdrojem je mutex úlohou uvolněn a zdroj může využít jiná čekající úloha [1].

1.2.2.3 Fronta zpráv

Mnoho aplikací vyžaduje, aby si úlohy mezi sebou vyměňovaly informace. K tomu slouží objekt fronta zpráv (angl. message queues). Fronta zpráv dovoluje výměnu dat mezi úlohami. Opět má svůj řídící blok QCB (angl. Queue Control Block), identifikátor. Záleží jen na programátorovi, požadavcích aplikace a množství dostupné paměti, kolik front si vytvoří. Důležitou vlastností fronty je její délka a šířka. Délka určuje počet zpráv, které je možné do fronty poslat. Šířka poté určuje maximální velikost jedné zprávy v Bitech. Také jsou k ní přiřazeny dva seznamy pro čekající úlohy. Jeden je pro úlohy, které čekají na poslání zprávy při plné frontě, druhý pro úlohy čekající na zprávu. Úlohy se v těchto seznamech opět mohou řadit podle pořadí, v jakém přišly, nebo podle svých priorit. Fronta zpráv má také informaci o počtu zpráv, které jsou v ní obsaženy. Tato hodnota určuje, zda je fronta prázdná nebo plná. Zprávy se mohou ve frontě řadit různými způsoby. Řadí se podle pořadí poslání, buď zepředu dozadu a nebo obráceně (FIFO a LIFO). Pokud je nutné poslat zprávu, která je větší než

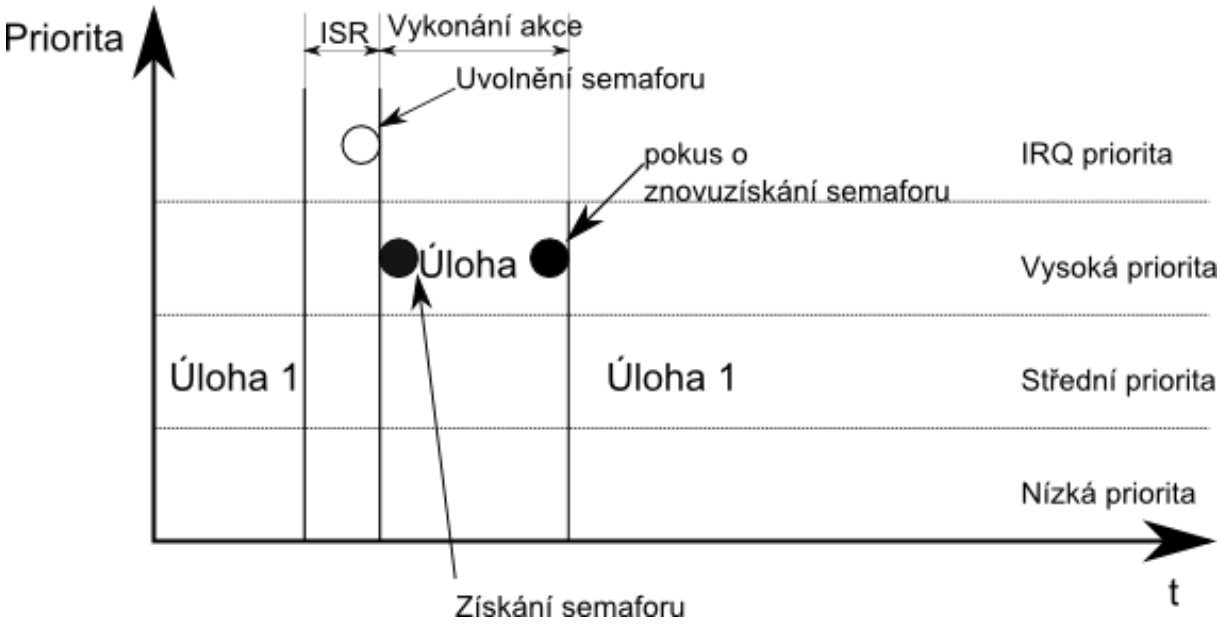
povolená šířka fronty, používá se pointer. Existuje několik způsobů použití fronty zpráv.

Jedna úloha data posílá a jedna nebo více úloh tyto data přijímají (producer a customer). Čekající úlohy nijak nepotvrzují přijetí dat. Posílající úloha nijak nečeká na potvrzení. Data posílá neustále, pouze pokud by se zaplnila fronta, tak bude čekat na její opětovné uvolnění.

Dalším příkladem může být komunikace s potvrzením přijetí. Úloha, která zprávu dodává, čeká na potvrzení jejího úspěšného doručení. Teprve poté vyšle další zprávu. Toto potvrzení může být ve formě uvolnění binárního semaforu. To provede přijímací úloha. Vysílací úloha poté semafor získá, vyšle data a pokusí se opět získat semafor. Protože semafor dosud nebyl uvolněn, úloha se zablokuje [1].

1.3 Přerušení (Interupt)

U RTOS je velmi důležité, aby přerušení proběhlo co možná nejrychleji. Jak již bylo řečeno, RTOS zavádí determinismus. Přerušení má vyšší prioritu, než probíhající úlohy. Při začátku jeho vykonávání tak přeruší průběh právě vykonávané úlohy bez ohledu na velikost její priority. Dobu, kdy se přerušení vyskytne, také často nelze předvídat. Je tedy nutné co nejvíce zkrátit dobu, jakou systém stráví v rutině přerušení ISR (angl. Interupt Service Routine). Z tohoto důvodu není v ISR povoleno žádné čekání (čekací smyčky). Řešení podle obrázku č. 7 je takové, že v ISR se provede jen nezbytný kód potřebný pro zpracování přerušení a uvolnění semaforu. Samotné provedení akce, která je od daného přerušení očekávána, je svěřeno obslužné úloze. Právě aktivní akce by tvořila většinu času, kterou by systém strávil v ISR. Tato úloha s vysokou prioritou je blokována na již zmíněném semaforu a čeká na jeho uvolnění. Po provedení se úloha pokusí semafor znovu získat. Semafor je však v nedostupném stavu a úloha se vrátí k čekání na jeho opětovný výskyt. Takto stráví aplikace v ISR minimální nutnou dobu a determinismus systému není téměř narušen [1].



Obr. č. 7 Průběh přerušení u RTOS

1.4 Kritická část programu

Někdy je potřeba, aby část programu v úloze nebyla přerušena. To znamená, že by nemělo dojít v průběhu této části k výskytu přerušení nebo vystřídání jinou úlohou. Jako příklad může posloužit zápis do registrů nebo příjem dat. V případě přerušení těchto operací by mohlo dojít k znehodnocení dat. Existují dva způsoby jak tento problém vyřešit - zakázání přerušení a zamknutí plánovače.

Zakázání přerušení zamezí zpracování přerušení. Některé architektury mikroprocesorů umožňují zakázání jen přerušení s určitými prioritami, jiné mohou zakázat pouze všechny. Protože vnější události jsou řešeny přes přerušení, dojde ke zpomalení odezvy aplikace na jejich výskyt. Je nezbytné zabránit tomu, aby se úloha v kritické části kódu zablokovala. Tento způsob vnáší do systému s RTOS nedeterminismus a je nutné jej využívat pouze v nezbytných případech.

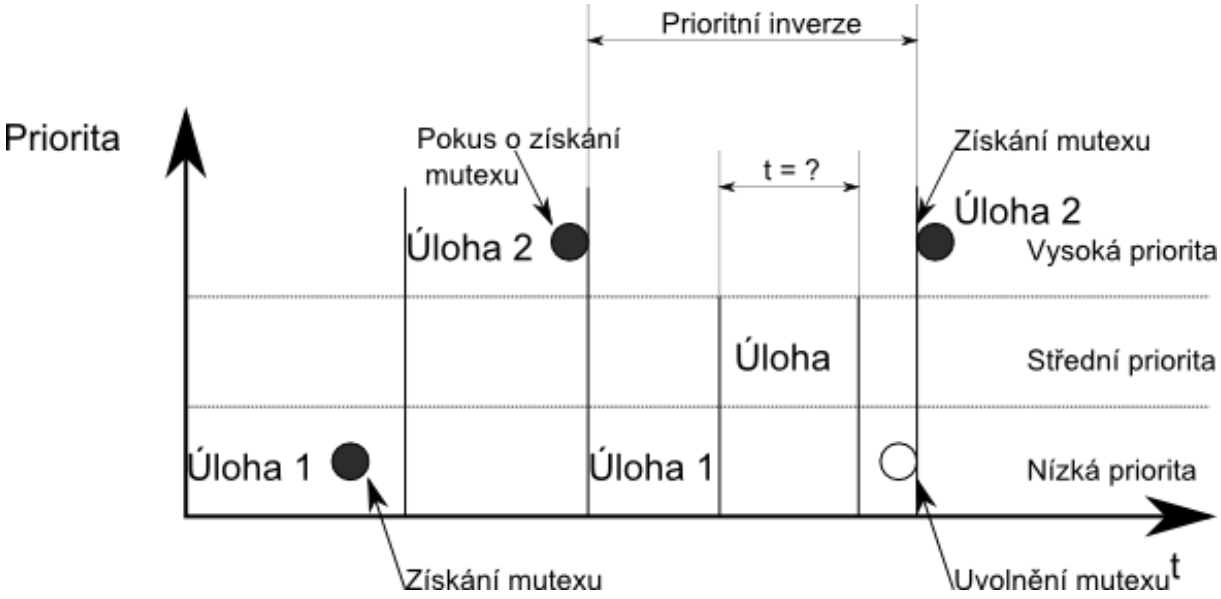
Zamknutí plánovače je druhým způsobem ochrany kritické části. Jak již název napovídá, tento mechanismus zabrání přepnutí na další úlohu. Bohužel i tento způsob má své problémy. Může zde dojít k prioritní inverzi. Přerušení jsou sice povolena, ale jejich hlavní kód se většinou vykonává v úloze s vysokou prioritou. Ta se ale nemůže vykonat z důvodu znemožněného přepínání úloh. Přerušení však nejsou ignorována jako v předchozím případě a jejich úlohy mohou být vykonány po skončení kritické sekce [1].

Většina RTOS nabízí jednu z těchto možností. Některé umožňují vybrat si, jaký způsob v dané situaci použít.

1.5 Prioritní inverze

Prioritní inverze nastává v případě, sdílí-li stejný zdroj úlohy s různými prioritami. Jedná se o případ, kdy jak už název napovídá, přestává platit prioritní plánování.

Jako příklad může sloužit následující. Úloha 1 s nízkou prioritou a úloha 2 s vysokou prioritou sdílí jeden společný zdroj. Nejdříve se vykonává úloha 1, ta získá zdroj. Ještě předtím než tento zdroj pustí, je vystřídána úlohou 2. Ta se během svého průběhu pokusí získat přístup ke zdroji. Ten ale vlastní úloha 1. Úloha 2 zablokuje a nechá vykonávat úlohu 1, dokud ta neuvolní zdroj. Pak by proběhlo plánování a aktivní by byla opět úloha 2. Doba, po kterou úloha 2 nechá běžet úlohu 1, je prioritní inverzí. Tento stav je ještě žádoucí, protože úloha 2 potřebuje přístup ke zdroji co nejdříve a to je možné pouze, pokud úloha 1 tento zdroj uvolní. Problém nastává, jak lze vidět na obrázku č. 8, je-li v aplikaci více úloh s hodnotou priorit mezi úlohami 1 a 2. V tomto případě může dojít k tomu, že úloha 1 ještě předtím, než stihne uvolnit sdílený zdroj, bude vystřídána nějakou jinou úlohou. Toto se v případě více úloh může navíc opakovat. Doba, za jakou úloha 1 uvolní zdroj, pak není jistá. Úloha 2 s vysokou prioritou je tak blokována po neznámou dobu. To naprosto ruší význam prioritního plánování.



Obr. č. 8 Prioritní inverze

Jedním z možných a často používaných způsobů, jak dobu trvání prioritní inverze omezit, je prioritní dědičnost. V tomto případě, pokud se úloha 2 pokusí získat sdílený zdroj, je priorita úlohy 1 dočasně zvýšena na hodnotu priority úlohy 1. Nemůže tak dojít k vystřídání úlohy 1 úlohou s nižší prioritou než je priorita úlohy 2. V prioritní inverzi je tak stráveno nutné minimum času pro uvolnění sdíleného zdroje [1].

1.6 Deadlock

Deadlock je takovým stavem systému, ke kterému dochází, sdílí-li dvě a více úloh několik zdrojů. Přístup úloh ke zdrojům je řízen mutexem. Deadlock nastane tehdy, když úloha 1 získá mutex a přístup ke zdroji 1. V jejím průběhu je ale vystřídána úlohou 2 s vyšší prioritou. Ta při svém vykonávání získá zdroj 2 a následně se pokusí získat zdroj 1. Ten je však v držení úlohy 1. Úloha 2 přejde do blokovaného stavu a plánovač opět nechá běžet úlohu 1. Úloha 1 se ještě před uvolněním zdroje 1 se pokusí získat zdroj 2, který má v držení úloha 2. Tím dojde k blokaci úlohy 1. Úlohy 1 a 2 se takto navzájem zablokují a znemožní správné vykonávání programu.

Pro menší aplikace lze deadlocku zabránit pozorným nastavením přístupu úloh ke sdíleným zdrojům. RTOS pak disponují algoritmy pro vyhledávání a řešení dreadlocku. Algoritmus sice nezabrání jeho výskytu, ale umožní jeho včasné rozpoznání a vyřešení [1].

1.7 Architektura I/O Ovladačů

Správně napsaný ovladač periferie realizujících vstupně/výstupní operace může v aplikaci využívající RTOS velmi přispět k rychlé a bezproblémové výměně dat s okolím systému. Jedná se o kolekci funkcí (Inicializace, Poslání, Příjem…) a obslužných rutin přerušení ISR. Funkce jsou volány z úloh běžících pod RTOS a ISR se spouštějí v reakci na přerušení od dané periferie. Jak je vysvětleno v kapitole 1.3, přerušení v prostředí RTOS pracují v kombinaci s úlohami vysoké priority. Tyto úlohy jsou také součástí ovladače.

I/O zařízení patří vedle paměti k nejčastějším sdíleným zdrojům. Jak je uvedeno v kapitole 1.2.2.2, může mít přístup k sdílenému zdroji vždy jen jedna úloha. Proto je zapotřebí mutex. Mutex je spolu s ostatními objekty RTOS nutnými pro běh ovladače vytvořen při inicializaci ovladače a je jeho vnitřní součástí (nepoužívá se mimo funkce ovladače). Mutex je získán vždy, když úloha zahájí pomocí funkce ovladače I/O operaci, a k jeho uvolnění dojde po jejím ukončení.

I/O ovladače lze rozdělit na synchronní a asynchronní. Rozdíl spočívá v tom, že u synchronního volající úloha čeká na výsledek I/O operace, zatímco v případě asynchronního se vykonává dál v době, kdy ovladač provádí operaci [2].

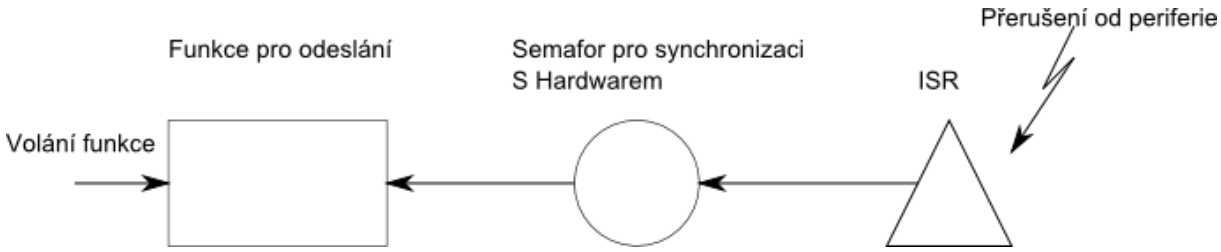
1.7.1 Synchronní

Synchronní ovladač je postaven kolem mechanismu, jež zabrání ve vykonávání volající úlohy do té doby, než jsou připravena požadovaná data. Samotný program ale zablokovaný není, v této době mohou běžet jiné úlohy. Jako takový mechanismus může sloužit binární semafor, na němž se úloha zablokuje. Tento semafor je také jako v případě

mutexu vytvořen při inicializaci ovladače. Je vytvořen prázdný. Pokud se ho úloha při vykonávání I/O funkce pokusí získat, tak se dostane do blokovaného stavu.

Jako příklad lze uvést užití funkce pro příjem dat. Úloha zavolá funkci vykonávající příjem. V této funkci vyšle žádost hardwaru, aby provedl operaci čtení, následně se pokusí získat již zmíněný semafor a zablokuje se. Až hardware dokončí danou operaci, vyvolá přerušení. V tomto případě není nutné postupovat podle kapitoly 1.3, protože od přerušení je vyžadována ta samá činnost, a tou je uvolnění semaforu. Po jeho uvolnění se odblokuje volající úloha a vyčte získaná data. Naprosto shodně pracuje i vysílání dat. Tento princip zobrazuje obrázek č. 9. Ukázka synchronního ovladače (včetně testovacího projektu pro vypsání znaků z klávesnice na PC) pro UART je na přiloženém CD v adresáři UART Driver\CoOS_synch a UART Driver\FreeRTOS_synch. Samotný ovladač jsou soubory UART_CoOS.c (UART_CoOS.h) a UART_FreeRTOS.c (UART_FreeRTOS.h).

Synchronní ovladače bývají většinou jednodušší než asynchronní [2].



Obr. č. 9 Synchronní ovladač (odesílání i příjem)

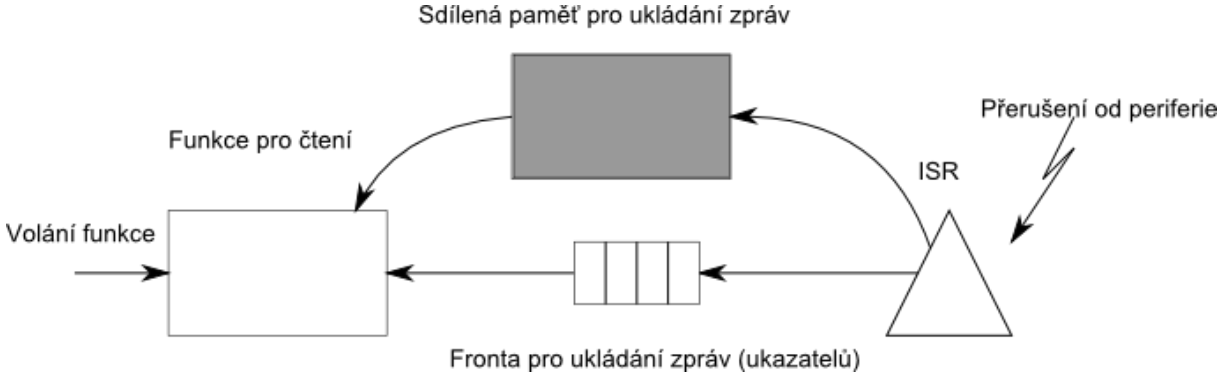
1.7.2 Asynchronní

Asynchronní ovladače nevyžadují od volající úlohy, aby čekala na dokončení I/O činnosti hardwaru. Zatímco synchronní ovladač je postaven kolem synchronizačního semaforu, asynchronní využívá bufferu mezi volanou funkcí a ISR částí ovladače. Buffer slouží k uchovávání dat připravených ke čtení nebo zápisu. Jako buffer slouží buď fronta zpráv, nebo společně sdílená paměť. Buffer je vytvořen při inicializaci ovladače. V případě asynchronních ovladačů je od ISR vyžadována složitější činnost a je již vhodné kombinovat ISR s obslužnou úlohou podle kapitoly 1.3.

Pro krátké zprávy stačí fronta zpráv. Hardware může běžet nezávisle na probíhající úloze a ukládat do fronty výsledná data tak, jak přicházejí. Volaná funkce pak může při svém zavolání tyto data vyzvednout. Frontu je nutné dimenzovat a data vyzvedávat tak často, aby nedocházelo k jejímu úplnému zaplnění. V případě rozsáhlých zpráv je možné zkombinovat sdílený paměťový prostor s frontou zpráv a do ní ukládat ukazatele do tohoto prostoru.

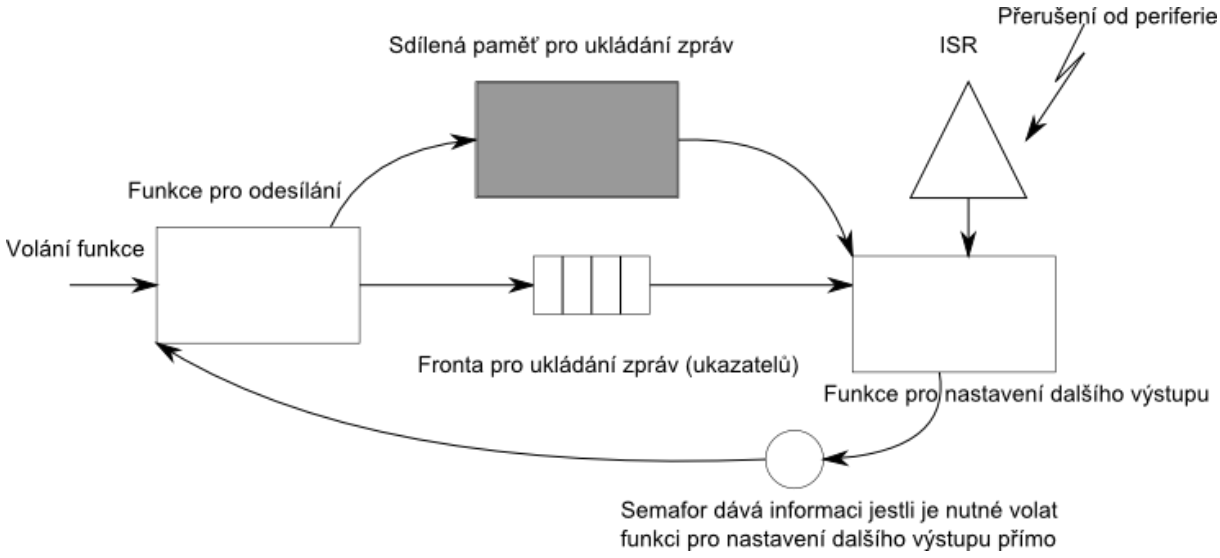
Tento způsob přijímání zpráv je zobrazen na obrázku č. 10. Úloha zavolá funkci,

ta vyzvedne zprávu nebo ukazatel z fronty zpráv. V případě, že vyzvedla ukazatel, tak z dané části paměti získá zprávu a danou část paměti uvolní pro další využití. ISR část programu (na obrázcích č. 10 a 11 jsou ISR a obslužná úloha brány jako jedna část) ukládá zprávy do fronty tak, jak přicházejí. V případě použití ukazatelů nejdříve alokuje požadované místo v paměti, do něj uloží zprávu a ukazatel uloží do fronty zpráv.



Obr. č. 10 Asynchronní ovladač (příjem)

Při posílání dat je situace o něco složitější. ISR postupně vybírá data z bufferu a posílá je. Dojdou-li v bufferu data, hardware již nevygeneruje další přerušení, protože v obsluze přerušení není možné čekat (ani v obslužné úloze) viz. kapitola 1.3. Hardware nezačne generovat přerušení ani, když by poté do fronty přišla nová data. Tento problém řeší zavedení dalšího binárního semaforu a zabalení činnosti probíhající v ISR do funkce. Semafor informuje volající úlohu o tom, zda je přerušení pozastaveno. Ve volané funkci se po odeslání dat (ukazatele) do fronty pokusí úloha získat semafor. Pokud se úloze podaří jej získat, volá funkci vykonávající odeslání dat přímo, pokud ne, hardware stále generuje přerušení a funkce končí. Po zavolání funkce přímo začne hardware generovat přerušení a odesílat sám. V případě, že by ISR zjistila prázdnou frontu, uvolní semafor a pozdrží generaci přerušení [2]. To je zobrazeno na obrázku č. 11.



Obr. č. 11 Asynchronní ovladač (vysílání)

2 Mikrokontroléry STM32

STM32 představuje úspěšnou řadu 32 bitových mikrokontrolérů. STM32 jsou postaveny na mikroprocesorech s jádry řady ARM Cortex M. Konkrétně řadách M-0, M-3 a M-4. Disponují širokou škálou periferií. Z pohledu RTOS je nejdůležitější právě jádro mikrokontroléru. Cortex řady M disponuje mnoha řešeními, které podporují determinismus a i jinak přispívají k hladkému chodu RTOS.

2.1 ARM Cortex M

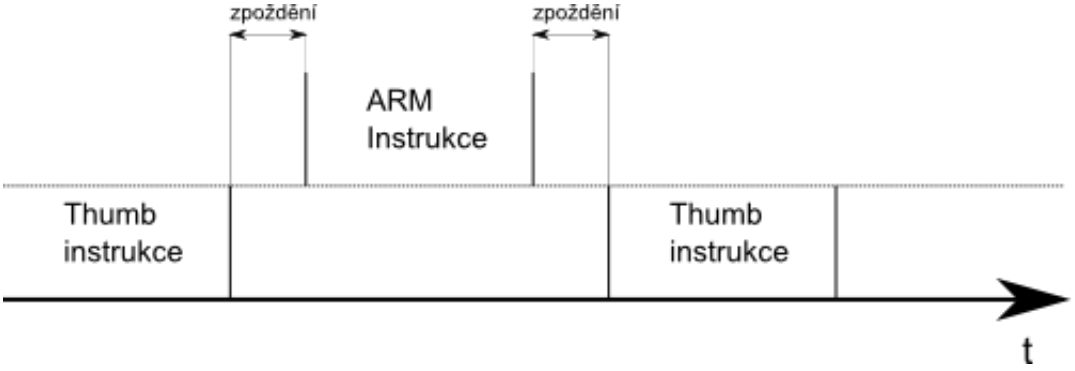
2.1.1 Základní vlastnosti

Mikroprocesory řady Cortex-M jsou postaveny na architektuře RISC. To znamená, že mají několikastupňovou pipeline. Všechny Cortexy používané v mikrokontrolérech STM32 mají 3 stupňovou pipeline . Cortexy M3 a M-4 disponují Harwardskou architekturou sběrnice. Mají dvě sběrnice - jednu pro načítání instrukcí a druhou pro načítání a ukládání dat. Cortex M0 má na rozdíl od nich Von-Neumanovu architekturu sběrnice. Instrukce i data zde sdílí jednu sběrnici. Cortexy-M následně disponují NVIC kontrolérem pro obsluhu přerušení, Thumb 2 instrukční sadou, MPU jednotkou pro ochranu přístupu k paměti, SYSTICK čítač pro spouštění přerušení OS a Bit-band pro mapování přístupu k jednotlivým bitům.

2.1.2 Thumb 2

Jádra Cortex disponují instrukční sadou Thumb 2. Ta se skládá jak z 32 bitových, tak 16 bitových instrukcí. Starší řady jader ARM měly 16 bitovou instrukční sadu Thumb a 32 bitovou sadu ARM. To znamenalo, že v průběhu vykonávání programu docházelo k přepínání

mezi těmito dvěma sadami. Takové neustálé přepínání zhoršuje determinismus a zpomaluje procesor. Jak ukazuje následující obrázek č. 12, přepínání mezi ARM a Thumb sadou není okamžité.



Obr. č. 12 Zpoždění při přepínání instrukčních sad

Po vytvoření sady Thumb 2 toto přepínání odpadlo. To umožňuje zrychlení jádra a zlepšení jeho deterministických vlastností [3].

2.1.3 NVIC

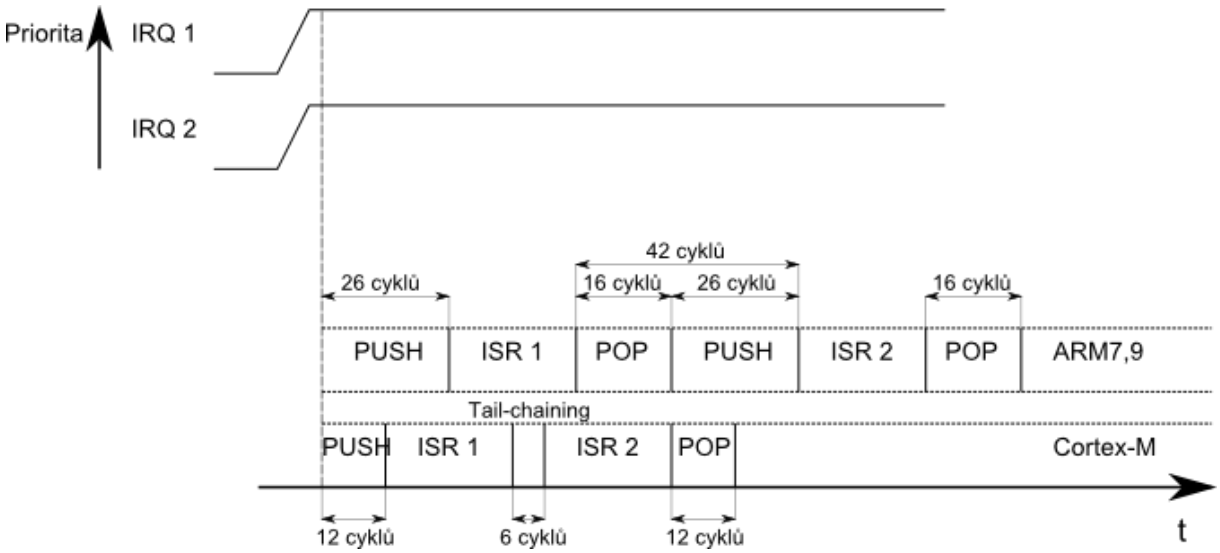
Vestavěný vektorový řadič přerušení dále jen NVIC (angl. Nested Vectored Interrupt Controler) je vnitřní periferií jádra. NVIC podporuje 1 – 240 vnějších přerušení a také 16 systémových výjimek. Umožňuje rychlý a deterministický přechod do obslužné rutiny přerušení. Tento kontrolér slouží k řízení přerušení. Lze pomocí jeho registrů přiřadit prioritu jednotlivým typům přerušení. Těch může být až 256 úrovní. V případě mikrokontrolérů STM32 jsou použity jen horní 4 bity. Má tedy 16 úrovní přerušení. To je velmi důležitá vlastnost z pohledu programátora pro systémy RTOS. NVIC nabízí také další výhody. Obsahuje registry pro ovládání SYSTICK. Může zamaskovat některé druhy přerušení, nebo úplně zakázat všechny kromě NMI [3].

2.1.3.1 Průběh přerušení

Primární výhodou a účelem NVIC je jeho rychlost a determinismus při přechodu do ISR. Tím není myšlena celková doba, kterou systém v přerušení stráví. To záleží na programátorovi, jak ISR naprogramuje. Doba přechodu je doba nutná pro uložení obsahu registrů mikroprocesoru do zásobníkové paměti a načtení počáteční instrukce ISR. Díky NVIC, Harwardské architektuře jádra a paralelnímu ukládání registrů do zásobníku při načítání adresy ISR, může klesnout až na 12 cyklů [3].

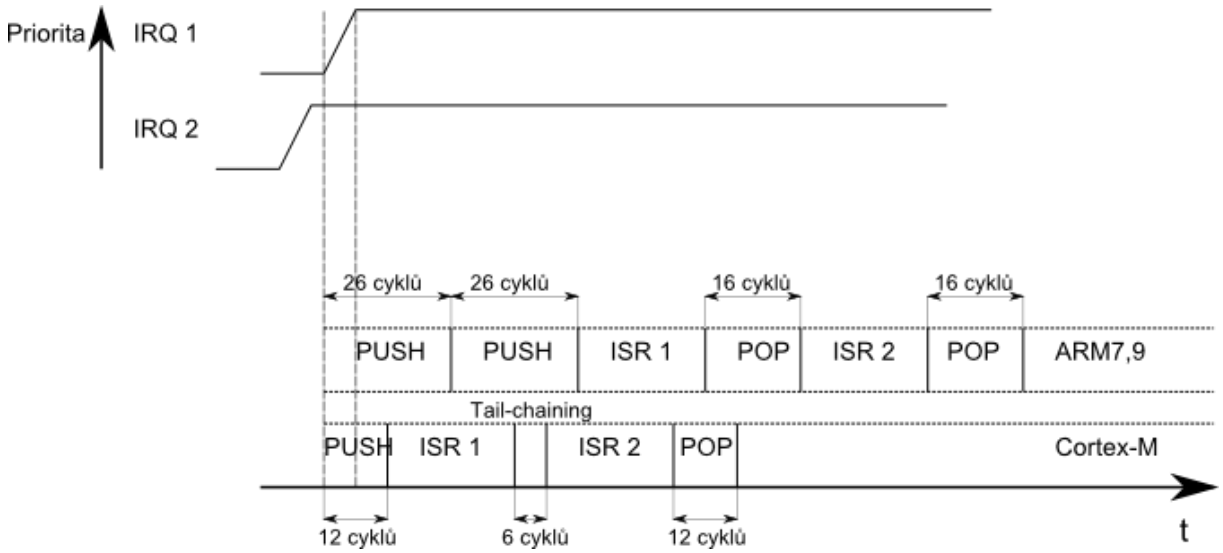
Další výhodou NVIC při přechodech do ISR je, přijde-li žádost o přerušení v průběhu obsluhy jiného přerušení. Má-li nižší prioritu, než má právě probíhající přerušení, musí jen počkat na jeho dokončení. Ve starších architekturách by došlo nejdříve k obnovení obsahu

z právě dokončeného přerušení a jeho následnému znovu-uložení. Tato operace je zdlouhavá. NVIC v Cortexu-M umožňuje Tail-chaining. Ten zajistí, že při přechodu z jedné ISR do druhé je nutné jen 6 strojových cyklů pro načtení adresy právě načítané rutiny ISR. Na obrázku č. 13 je vidět rozdíl rychlosti zpracování jak při samotném přechodu do ISR (na obrázku PUSH), tak při přechodu mezi ISR 1 a ISR 2.



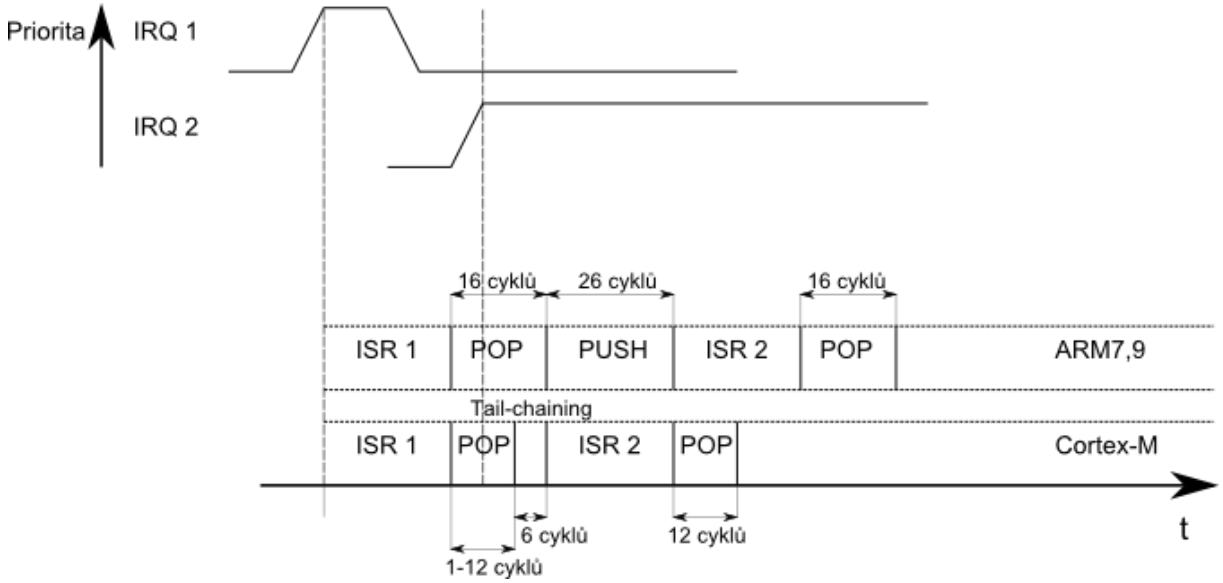
Obr. č. 13 Rozdíl odezvy IRQ u starších architektur a Cortex-M

Další zrychlení poskytuje NVIC při zpracování pozdního příchodu žádosti o přerušení. Ten nastává tehdy, přijde-li během ukládání zásobníku pro první IRQ žádost s vyšší prioritou. Jak ukazuje obrázek č. 14. Jsou zde opět dvě žádosti o přerušení, IRQ 2 s nižší prioritou a IRQ 1 s prioritou vyšší. U starších verzí ARMu bylo nejdříve dokončeno ukládání pro první IRQ 2 a následně byl uložen do zásobníku obsah ISR 2. Až poté začala obslužná rutina ISR 1. Po jejím dokončení byly ze zásobníku obnoveny registry pro vykonání ISR 2. Pokud se stejný případ zpožděného výskytu objeví u Cortexu-M, normálně se uloží obsah registrů. Pouze se změní adresa z ISR 2 na ISR 1. Tím je opět možné zkrátit přechod do ISR až na 12 cyklů. Pro přechod mezi rutinami je opět použit Tail-chaining.



Obr. č. 14 Pozdní příchod IRQ

Posledním případem snížení času a zlepšení determinismu je vyskytnutí žádosti o přerušení při obnově dat ze zásobníku po přerušení s nižší prioritou. V tomto případě, jak ukazuje obrázek č. 15, dojde k vrácení ukazatele zásobníku na původní hodnotu a následný vstup do ISR 2 trvá stejně jako Tail-chaining 6 cyklů. Přechod do nové obslužné rutiny tedy trvá mezi sedmi až devatenácti cykly. Zatímco u starších architektur by muselo dojít k úplnému obnovení dat ze zásobníku a následnému uložení dat zpět do zásobníkové paměti pro vykonání ISR 2 [3].



Obr. č. 15 IRQ při obnově dat ze zásobníku

Všechny tyto vlastnosti Cortexu-M a jeho NVIC kontroléru zkracují dobu nutnou ke zpracování přerušení a přispívají tak k vyššímu determinismu při běhu pod RTOS.

2.1.3.2 Využití registrů NVIC

NVIC disponuje několika registry, které jsou z hlediska využití RTOS zajímavé. Jsou to registry PRIMASK, FAULTMASK a BASEPRI.

Jak je uvedeno výše, jedním ze způsobů ochrany kritické sekce kódu, je zakázání přerušení. Některé RTOS využívají pouze zamknutí plánovače a nepodporují zakázání přerušení. Přesto díky těmto registrům může programátor využít i tohoto typu.

PRIMASK registr zakáže všechna přerušení kromě NMI nemaskovatelného přerušení (angl. NonMaskalbe Interupt) a Hard fault chyby.

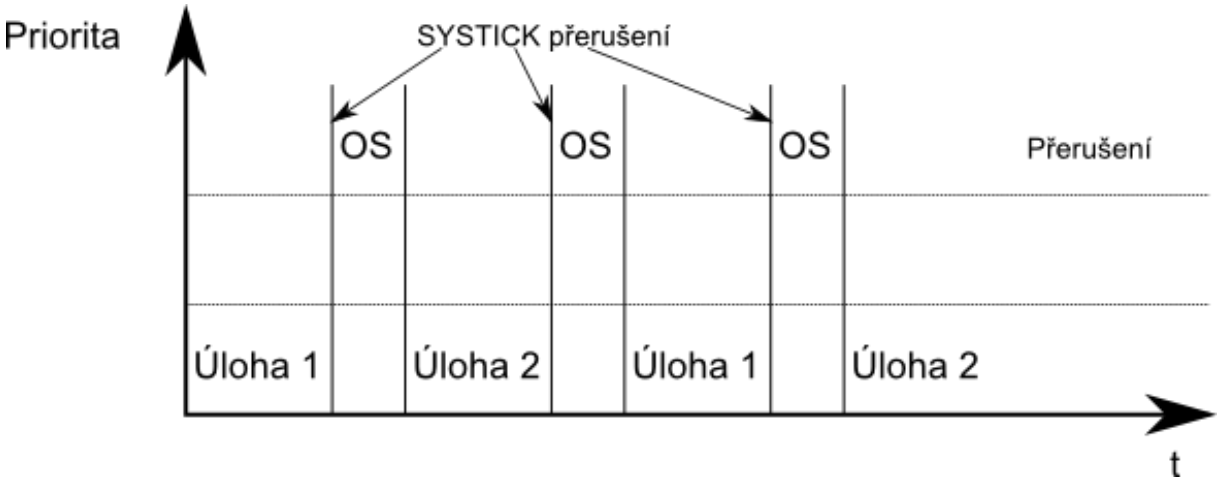
FAULTMASK zakáže všechna přerušení kromě NMI.

BASEPRI je až 8 bitový registr. Zamaskuje všechny přerušení se stejnou nebo nižší hodnotou priority, než je hodnota v něm zapsaná.

Tyto tři registry umožňují programátorovi velmi rozsáhlé nastavení povolování a zakazování přerušení. I v případě, že RTOS podporuje zakázání přerušení, může být vhodné využití těchto registrů pro jemnější nastavení.

2.1.4 SYSTICK

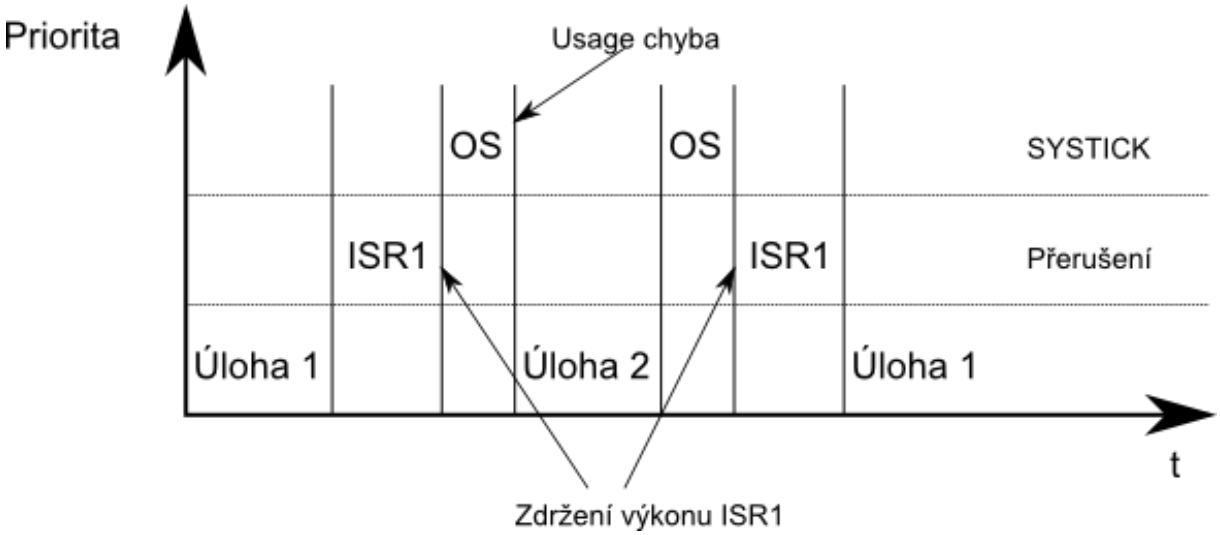
SYSTICK je 24 bitový dekrementující čítač. Je součástí NVIC kontroléru. Používá se pro vyvolání SYSTICK přerušení. Na výběr jsou dva zdroje hodinových signálů, vnitřní hodiny procesoru, nebo vnější zdroj hodin. Čítač se automaticky obnovuje při dosažení nuly. SYSTICK slouží při generování přerušení pro běh RTOS obslužné rutiny. Zde RTOS provádí správu úloh. Především pro round-robin plánovač, kdy v tomto přerušení dochází k přepnutí běžící úlohy. Takovou činnost ilustruje obrázek č. 16. Další činností vykonávanou RTOS v tomto přerušení je například správa paměti [3].



Obr. č. 16 SYSTICK přerušení

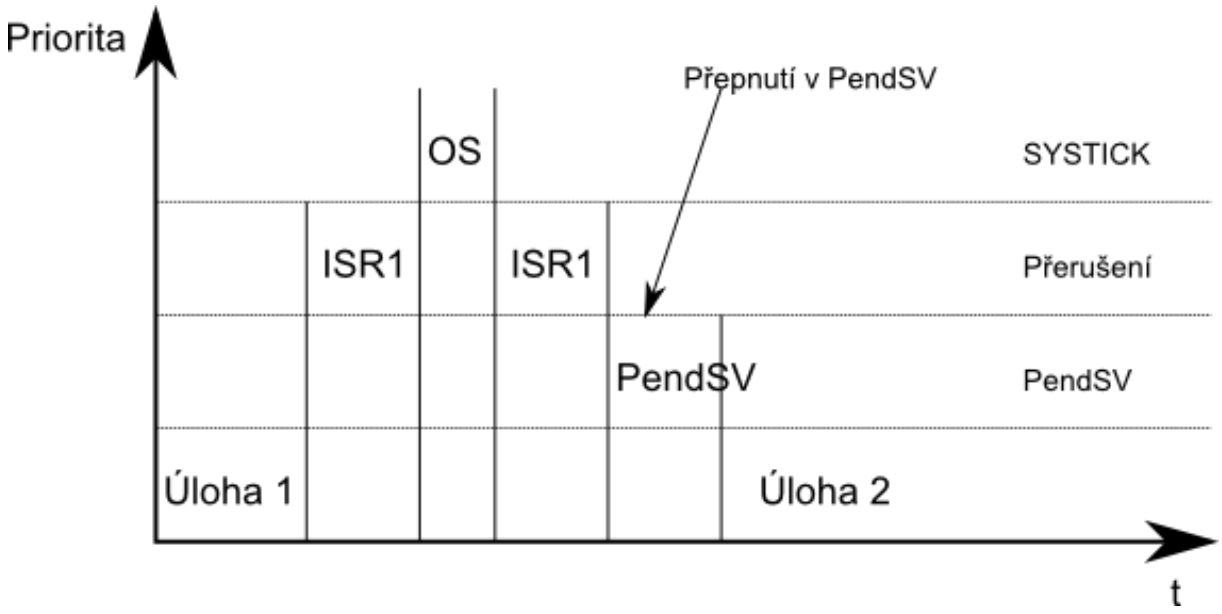
2.1.5 PendSV

Vzhledem k různým prioritám přerušení může v případě volání přerušení SYSTICK dojít ke zpoždění vykonání jiné rutiny přerušení s nižší prioritou. V průběhu SYSTICK přerušení dochází k přepnutí kontextu (úloh). Procesor se tak může dostat zpět do Thread módu, i když je stále aktivní jiné přerušení. Tento stav, jak lze vypozorovat z obrázku č. 17, může vyvolat Usage chybu a oddálit vykonání přerušení. To je v RTOS nepřijatelné.



Obr. č. 17 Vyvolání Usage chyby a zpoždění vykonávání ISR

Některé RTOS to mohou řešit kontrolou před začátkem změny úloh, jestli se nevykonává ISR. Bohužel se tím může výrazně prodloužit doba pro přepnutí úloh. PendSV řeší možnou chybu a zpoždění ve vykonání tím, že pozdrží vykonání přepnutí. PendSV má nastavenou nejnižší prioritu ze všech povolených přerušení. Pokud je operačním systémem rozpoznáno probíhající přerušení (již probíhající rutina je vystřídána SYSTICK přerušením), je přepnutí pozdrženo do PendSV. PendSV je voláno plánovačem, pokud je na začátku procesu přepnutí obsahu detekována probíhající ISR rutina. Následující obrázek č. 18 ukazuje použití PendSV při přepínání obsahu. Probíhající úloha je přerušena přerušením s nějakou prioritou. V jeho průběhu přijde SYSTICK. OS rozpozná probíhající přerušení (běží IRQ handler). Vykoná jen práci, která nezpůsobí přepnutí obsahu. Přepnutí kontextu se provede v PendSV. Po skončení SYSTICK přerušení se dokončí vykonávání předchozího přerušení. K přepnutí úloh dojde až po jeho skončení v PendSV [3].



Obr. č. 18 Využití PendSV při výskytu přerušení pro přepnutí kontextu

2.1.6 MPU

Důležitou vlastností RTOS je jejich spolehlivost a bezpečnost. MPU (angl. Memory Protection Unit) umožňuje při běhu RTOS ochranu dat, požívaných jádrem od úloh programu. Dále nabízí oddělení paměťových oblastí pro jednotlivé úlohy. Tím lze zabránit smazání dat jedné úlohy úlohou jinou. Chrání také před přetečením zásobníkové paměti. Pokud jsou práva pro přístup k datům porušena, je vyvolána chybová výjimka. Je také možné určit některé oblasti paměti pouze ke čtení a tím zabránit smazání dat v nich obsažených [3].

2.1.7 Bit-Band

Bit-Band slouží pro práci s jednotlivými bity. Mikroprocesor běžně pracuje buď s byty, celými slovy nebo půlslovy. Občas je však zapotřebí změnit v jednom slovu pouze jeden bit. Dříve muselo dojít k načtení celého slova, jeho zamaskování maskou podle zvoleného bitu, provedení zápisu a následnému uložení celého slova zpět. Tento postup je zdlouhavý a nespolehlivý. Pokud se v průběhu změny bitu tímto způsobem vyskytlo přerušení, mohlo dojít k znehodnocení této operace a výslednému chybnému zapsání bitu.

Bit-Band s skládá ze dvou oblastí paměti. Jednou je oblast, ve které se nacházejí slova, kekterým lze přistupovat po jednotlivých bitech. Druhou je alias oblast, kde se nacházejí slova, jejichž LSB je namapován právě vždy na jeden bit z předchozí oblasti. Tímto způsobem lze změnou celého slova v alias oblasti změnit jeden bit v mapované oblasti. Tato operace je navíc nepřerušitelná a rychlejší než předchozí způsob. Nemůže tedy dojít k chybnému zapsání bitu. Většinou jsou v části Bit-Band regionu paměti namapovány registry periferií mikrokontroléru. Zbytek je volně použitelný uživatelem [3].

Při použití RTOS může díky Bit-Bandu přistupovat k registrům periferií více úloh, aniž by hrozilo chybné zapsání.

3 Měřené RTOS

3.1 RTOS pro STM32

Dnes jsou na trhu desítky RTOS pro různé architektury [4], například VxWorks [5], SafeRTOS [6], eCos [7], FreeRTOS [8], CoOs [9] a desítky dalších. Orientovat se v takovémto množství je obtížné. Některé RTOS je možné zakoupit, jiné jsou určeny jen pro účely firem, které je vlastní a další jsou distribuovány volně.

Při výběru vhodných RTOS pro účely této práce bylo primárním kritériem to, aby podporovaly mikrokontroléry STM32 potažmo jádra ARM Cortex M-0, M-3 a M-4. Druhým nejdůležitějším kritériem byla dostupnost autorovi. Zde padlo rozhodnutí soustředit se na volně šířené RTOS, zde se ukázaly jako nejvhodnější ChibiOS [10], TNKernel [11], FreeRTOS a CoOS. Díky tomuto požadavku se tato práce komerčně dostupnými RTOS, jako jsou VxWorks nebo SafeRTOS, nezabývá. Jelikož byl pro testování vybrán mikrokontrolér s jádrem M-4, bylo nutné, aby jej testované RTOS podporovaly. Posledním požadavkem byl dostatek materiálů a snadnost naučení se s daným RTOS pracovat. Tyto požadavky nejlépe splnily CooCox CoOS a FreeRTOS. ChibiOS a TNKernel nebyly vybrány právě z důvodu malé podpory pro programátory, kteří s nimi začínají a jejich obtížnému rozběhnutí nadostupném hardwaru.

3.2 CooCox CoOS

CoOS od CooCoxu je RTOS určený speciálně pro mikrokontroléry s jádrem Cortex-M. V současné době jsou oficiálně podporována jádra M0 a M3. CoOS je schopen celkem bez problémů pracovat i na jádrech M4, ta však nejsou zatím oficiálně podporována a RTOS tak pravděpodobně nevyužije všechny možnosti těchto jader. RTOS od Coocoxu podporuje překladače ICCARM, ARMCC a GCC.

CoOS obsahuje kromě tří základních objektů úlohy, semaforu a fronty zpráv také dva další objekty. Těmi jsou MailBox a registr příznaků (angl. Flags). Mailbox slouží pro poslání jedné zprávy. Jedná se tedy vlastně o základní prvek fronty zpráv. Příznaky jsou použity pro synchronizování úlohy s více událostmi najednou. Tedy je-li nutné splnění více podmínek, než je úloha spuštěna. Dále obsahuje funkce pro práci s pamětí a softwarovým čítačem.

CoOS podporuje preemptivní plánování mezi úlohami s různou prioritou a round-robin u úloh se stejnou prioritou. Prioritní inverzi řeší formou prioritní dědičnosti.

CoOs je distribuován formou C a H souborů, které se přidávají do projektu. Ty jsou rozděleny na zdrojové (angl. source) soubory, které se přidávají podle toho, které části RTOS jsou použity, a portovací soubory, které se vybírají dle použitého překladače. Nastavování probíhá pomocí OsConfig.h. RTOS je škálovatelný, je tedy možné používat jen ty části, které jsou v aplikaci potřeba [9].

Tento operační systém obsahuje jen nejnutnější funkce. Je jednoduchý na zprovoznění. Tyto vlastnosti z něj dělají vhodného kandidáta pro začátečníky a pro studijní účely. Právě tato jednoduchost však může být překážkou při vývoji náročnějších a specializovaných aplikací. Například CoOS neumožňuje v případě plné fronty zpráv, aby se úloha při pokusu o zápis dostala do blokovaného stavu a počkala na uvolnění místa ve frontě.

3.3 FreeRTOS

FreeRTOS patří k nejrozšířenějším RTOS systémům. Podporuje 34 různých architektur. Pro tuto práci je podstatné, že podporuje Cortex-M mikrokontroléry. A to řady M0, M3 a M4. Podporuje také množství překladačů a vývojových prostředků např. IAR, Atollic TrueStudio, GCC, Keil, Rowley CrossWorks.

Opět, jako v předchozím případě, kromě základních objektů obsahuje několik navíc. Jedná se o příznaky zde nazvané Event Groups a Co-rutiny. Co-rutiny jsou hodně podobné úlohám. Mohou pracovat společně s úlohami, nebo je i úplně nahradit. Hlavním rozdílem Co-rutiny a úlohy je to, že zatímco každá úloha disponuje svým vlastním zásobníkem, Co-rutiny sdílejí jeden společný zásobník. Tím dojde k ušetření místa v paměti. Takto ušetřené místo je ale vykoupeno tím, že je omezeno, kdy a jak lze Co-rutiny použít.

Stejně jako CoOS podporuje preemptivní i round-robin plánování.

Přidání tohoto RTOS do aplikace probíhá podobně jako v předchozím případě. Opět jsou zde zdrojové a portovací soubory. Portovací soubory se vybírají nejen podle použitého překladače, ale i podle zvolené architektury mikroprocesoru. Nastavování požadovaných částí a vlastností se provádí v hlavičkovém souboru FreeRTOSConfig.h. FreeRTOS je škálovatelný a nabízí tak možnost vypuštění nepotřebných objektů. Tím jsou sníženy paměťové nároky jádra RTOS [8].

4 Měření výkonnosti RTOS

Při výběru vhodného RTOS pro vyvíjenou aplikaci je potřeba vzít v úvahu vlastnosti, jaké jsou pro daný případ žádoucí. V této práci je na jednotlivé vlastnosti RTOS dán stejný důraz. To by však ve skutečnosti neplatilo. Někdy je důležité rychlé posílání zpráv mezi

úlohami, jindy je požadováno minimální využití paměti atd. Dále by záleželo na objektech a službách, jež by RTOS nabízel.

Tato kapitola se zaobírá jen měřitelnými vlastnostmi RTOS. Tyto hodnoty byly měřeny jen na základních objektech, které jsou pro všechny RTOS společné. Těmi jsou úlohy, semafory (mutexy) a fronty zpráv.

Je možné rozdělit toto měření na dvě základní oblasti. Časovou a na využití paměti. Časové vlastnosti lze následně členit na Latenci přerušení a na Latenci využití služeb RTOS.

4.1 Služby RTOS

Vykonání každé služby RTOS zabere nějaký počet hodinových cyklů procesoru. Je tedy důležité, především v časově náročných aplikací, aby se provádění těchto funkcí bylo rychlé a časově stálé.

V této práci budou zmíněny dva možné způsoby změření latence služeb RTOS. Buď softwarově pomocí čítače/časovače obsaženého v procesoru, nebo pomocí změny hodnot na výstupním pinu měřených osciloskopem.

V prvním případě je použit volně běžící čítač. Před volání funkce systému je jeho hodnota čítacího registru uložena a stejně tak po jejím skončení. Výsledný čas je následně spočten z rozdílu těchto hodnot a z frekvence, na které čítač běží. Uložení registru do proměnné trvá nějakou dobu, která by se mohla projevit jako chyba. Při odečtení hodnot se tato chyba vyruší.

Pro případ měření latence osciloskopem jsou použity GPIO piny, na které je připojen osciloskop. Úroveň na GPIO pinu je nastavena na určitou hodnotu. Před vykonáním funkce RTOS je tato úroveň změněna a po jejím skončení je vrácena do původní hodnoty. Pomocí osciloskopu je pak změřena doba, po jakou je na GPIO pinu odlišná hodnota. Stejně jako v předchozím případě, i zde se případná chyba ztratí při výpočtu.

Měření bylo provedeno na těchto službách systému:

- Vytvoření úlohy
- Vytvoření semaforu
- Vytvoření fronty zpráv
- Přepnutí úloh (stejná priorita)
- Přepnutí úloh (různé priority)
- Prohození semaforu
- Doba přenosu zprávy (mezi dvěma úlohami)

4.2 Latence přerušení

Jednou z nejdůležitějších vlastností RTOS je rychlost, s jakou je schopen zpracovat příchozí přerušení. Jak již bylo uvedeno v předchozím textu (kapitola 1.3), doba strávená v obslužné rutině přerušení musí být co nejkratší. O latenci přerušení také do značné míry rozhoduje typ mikrokontroléru, na jakém je RTOS aplikován (kapitola 2.1.3).

V této práci bude za latenci přerušení považován čas od výskytu přerušení až po začátek vykonávání obslužné úlohy, která vykonává požadovanou činnost.

Měření této doby bude probíhat stejným způsobem, jako měření Latence služeb RTOS.

4.3 Využití paměti

Embedded systémy většinou disponují jen omezeným množstvím paměti, kterou může program využít. Je tedy důležité, aby samotné jádro RTOS mělo co nejmenší požadavky na zabírané místo. Objekty RTOS, jež jsou vytvořeny aplikací, by pak také měly zabírat co nejméně místa.

Pro určení, kolik místa v paměti zabere jádro RTOS, je použita informace, kterou poskytuje překladač při překladu. Pro účely této práce byl použit překladač ARMCC a prostředí Keil poskytované samotným ARMem. Informace jsou následně uloženy v .map souboru. To zahrnuje jak velikost ROM potřebné pro uložení programu, tak také velikost nutné RAM pro jeho běh.

Nejdříve byly zjištěny paměťové nároky aplikace bez připojeného RTOS. Poté byly do aplikace přidány RTOS a opět se zjistily hodnoty. Od těch se následně odečetla hodnota získaná v prvním případě.

4.4 Rhealstone

Rhealstone je způsob srovnávání RTOS systémů. Byl navržen na konci 80. let minulého století Rabindrou P. Kar [12]. Vznikl z požadavku pro ucelené a opakovatelné porovnávání RTOS systémů. Systémem se zde myslí jak samotné jádro RTOS, tak hardwarová platforma na které běží.

Rhealstone je určen pro RTOS systémy, kde dochází k multitaskingu. To znamená, že v aplikaci běží dvě a více úloh s různými prioritami. Nehodí se pro programy, ve kterých běží pouze úlohy s jednou prioritou.

Rhealstone staví na tom, že každá RTOS aplikace je jiná. V jedné může docházet k častému posílání zpráv, jiná je založena na přerušeních atd.

Rhealstone je prezentován jako jedno číslo. Toto číslo se dostane sečtením šesti změřených hodnot. Ty reprezentují aktivity, jež patří k nejdůležitějším z hlediska výkonnosti

RTOS systémů. Každá změřená hodnota je nejdříve reprezentována v časové oblasti. Následně se převede na sekundy a spočte se její převrácená hodnota. Tím se získá číslo, kolikrát se daná činnost provede v jedné sekundě. Takto získaná hodnota je následně použita ve finálním výpočtu, přičemž r_1 až r_6 reprezentují jednotlivé aktivity. Platí, že čím vyšší výsledné číslo, tím lépe vykonává RTOS dané operace.

$$r_1 + r_2 + r_3 + r_4 + r_5 + r_6 = r\ [Rhealstony/sekundu]$$

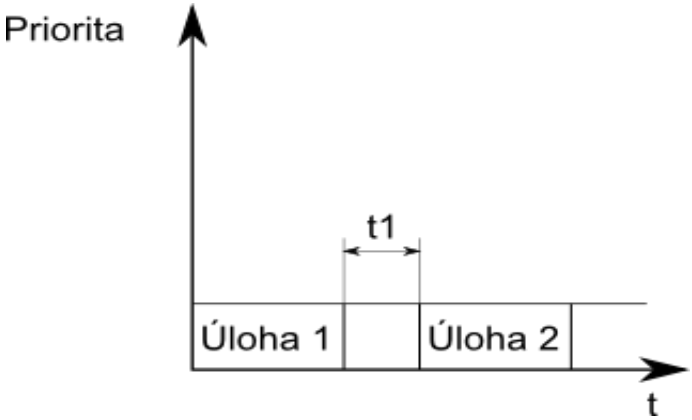
Jak již bylo napsáno každá aplikace využívající RTOS je jiná. To je odraženo i ve výpočtu finální hodnoty. Dosáhne se toho tím, že každému číslu ve výpočtu se přiřadí váha. Velikost vah se volí podle zastoupení jednotlivých činností v aplikaci. Záleží na tom, jak již bylo zmíněno výše, zda například v aplikaci dochází k častému posílání zpráv nebo k velkému využití přerušení. V některých případech je možné dokonce vypustit některé části, pokud nejsou v programu vůbec používány. Vzorec s váhováním je následující.

$$n_1 * r_1 + n_2 * r_2 + n_3 * r_3 + n_4 * r_4 + n_5 * r_5 + n_6 * r_6 = r\ [Rhealstony/sekundu]$$

4.4.1 Přepnutí úloh (stejná priorita)

Jedná se o čas potřebný pro přepnutí mezi dvěma aktivními úlohami stejné priority. Tyto úlohy jsou vzájemně nezávislé, nesdílejí vzájemné zdroje ani jinak na sobě nezávisí. Dochází k němu například při round-robin přepínání. Tato hodnota představuje informaci o tom, jak efektivně RTOS pracuje při ukládání a obnovování kontextu jednotlivých úloh [12].

Na obrázku č.19 tuto hodnotu představuje t1.

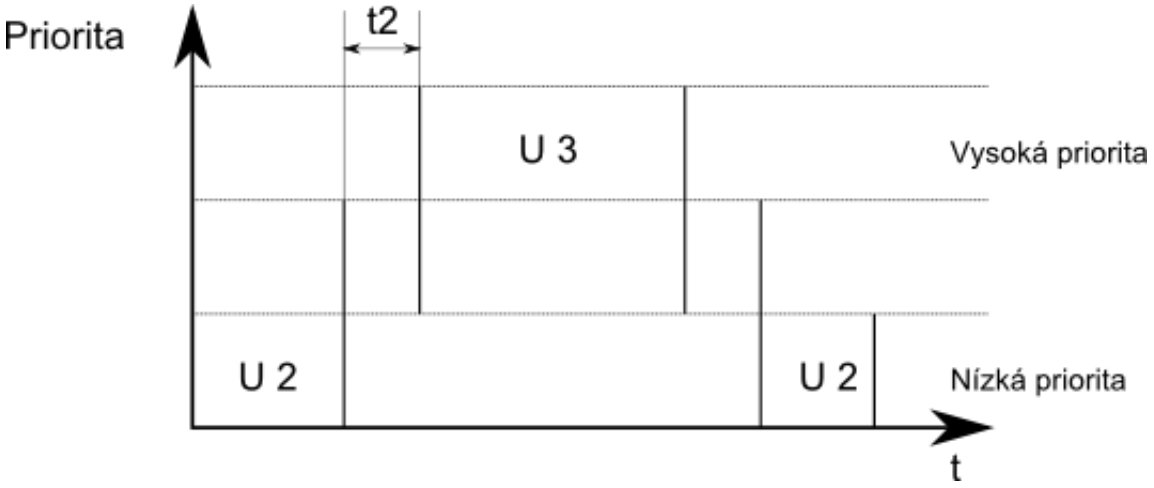


Obr. č. 19 Zpoždění při přepnutí úloh se stejnou prioritou

4.4.2 Přepnutí úloh (různá priorita)

Tato hodnota představuje čas potřebný k tomu, aby úloha s vyšší prioritou získala kontrolu nad procesorem od úlohy s prioritou nižší. K tomuto přepnutí většinou dochází při přechodu úlohy s vyšší prioritou z blokovaného stavu v reakci na nějakou vnější událost (například při uvolnění semaforu, na němž tato úloha byla blokována). I když se tato hodnota může zdát na první pohled podobná té předchozí, tento případ zabere více času. RTOS totiž musí nejdříve rozeznat akci, která odblokovala úlohu a porovnat priority běžící a právě odblokované úlohy [12].

Na následujícím obrázku č. 20 je toto zpoždění reprezentováno jako t2.



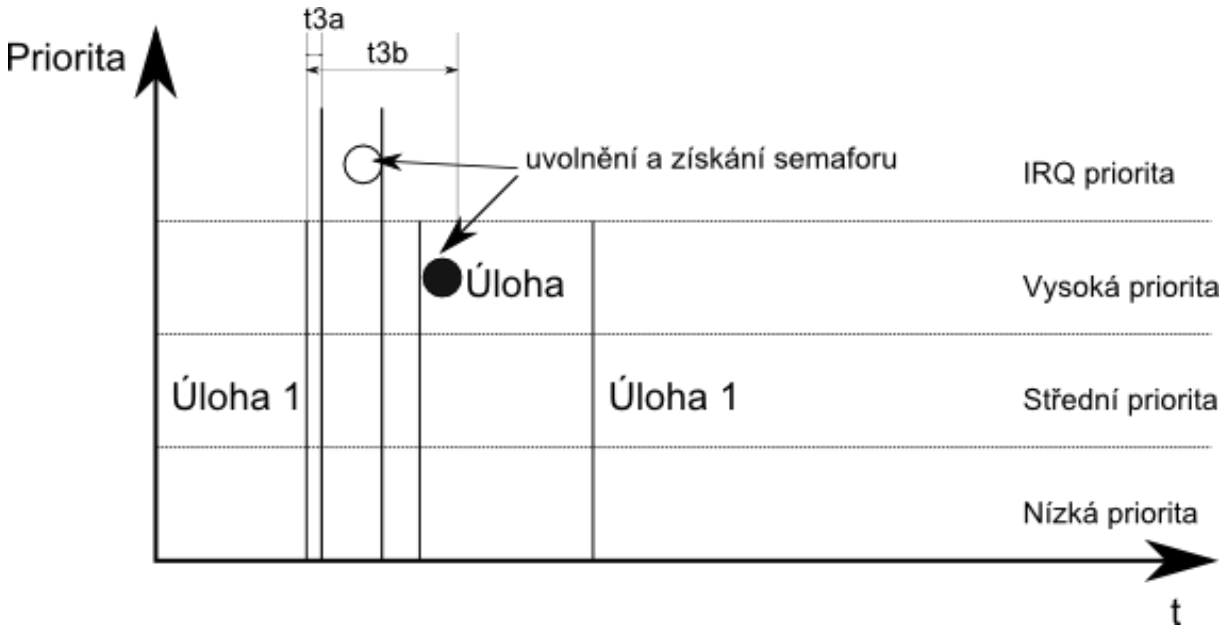
Obr. č. 20 Zpoždění při přepnutí úloh s různou prioritou

4.4.3 Latence přerušení

Latence přerušení je v Rhealstonu definována jako čas od výskytu požadavku na přerušení do začátku vykonávání jeho první instrukce. Během této doby mikroprocesor uloží současný obsah registrů do zásobníku a načte první instrukci rutiny přerušení [12].

V RTOS se však běžně používá přerušení spolu s úlohou vysoké priority. Bylo by tedy možné považovat za latenci přerušení také dobu začínající výskytem požadavku a končící začátkem vykonávání dané obslužné úlohy.

Původní definice, která je čistě hardwarově závislá, je na obrázku č. 21 zobrazena jako t3a, t3b pak představuje latenci při využití obslužné úlohy podle kapitoly 1.3.

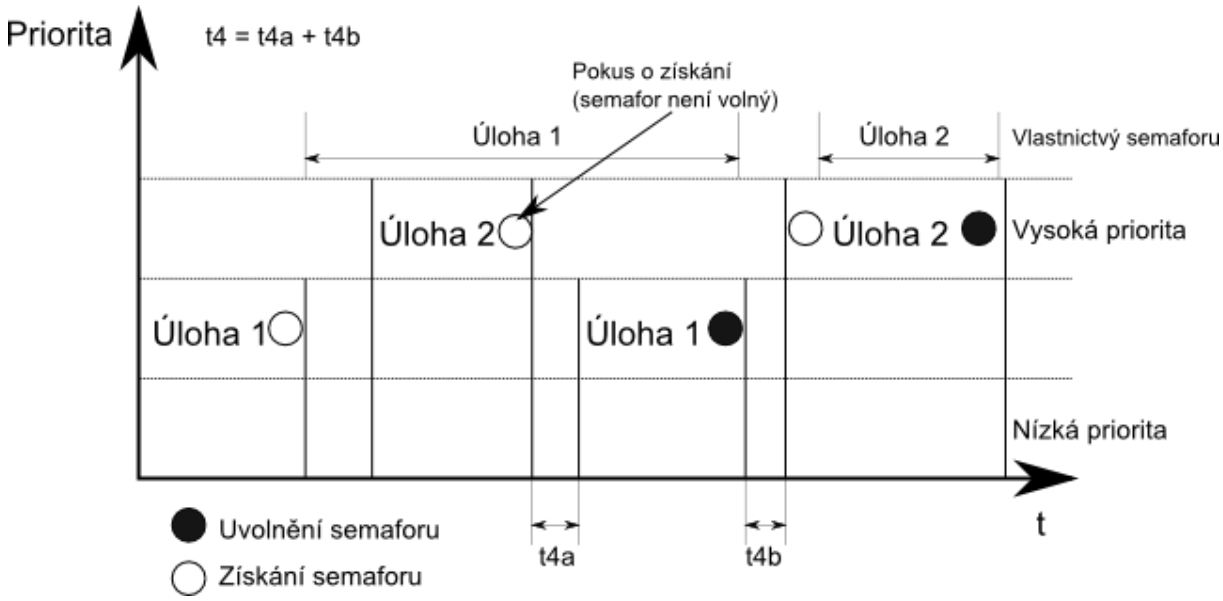


Obr. č. 21 Latence přerušení

4.4.4 Prohození semaforu (semaphore shuffle)

Tímto názvem je myšlen čas, který je potřebný pro uvolnění semaforu jednou úlohou a začátkem vykonávání jiné úlohy, jež předtím čekala na semafor v zablokovaném stavu. Toto měření je asociováno s jevem vzájemné exkluze (využití mutexů). Ta se používá při sdílení společných zdrojů více úlohami a zajišťuje to, že přístup ke sdílenému zdroji má vždy pouze jedna úloha [12].

Jak je vidět na následujícím obrázku č. 22, úloha 1 získá semafor. V průběhu jejího vykonávání jí vystřídá úloha 2. Ta se také pokusí získat semafor. Semafor, ale není dostupný, a úloha 2 přejde do blokovaného stavu a nechá vykonávat úlohu 1. Úloha 1 po nějaké době uvolní semafor a úloha 2 se dostane z blokovaného stavu a získá jej. Po skončení požadovaných operací je následně také uvolní pro další zpracování. Součet dob t4a a t4b představuje dobu nutnou pro vyřešení vlastnictví semaforu.



Obr. č. 22 Zpoždění při prohození semaforů

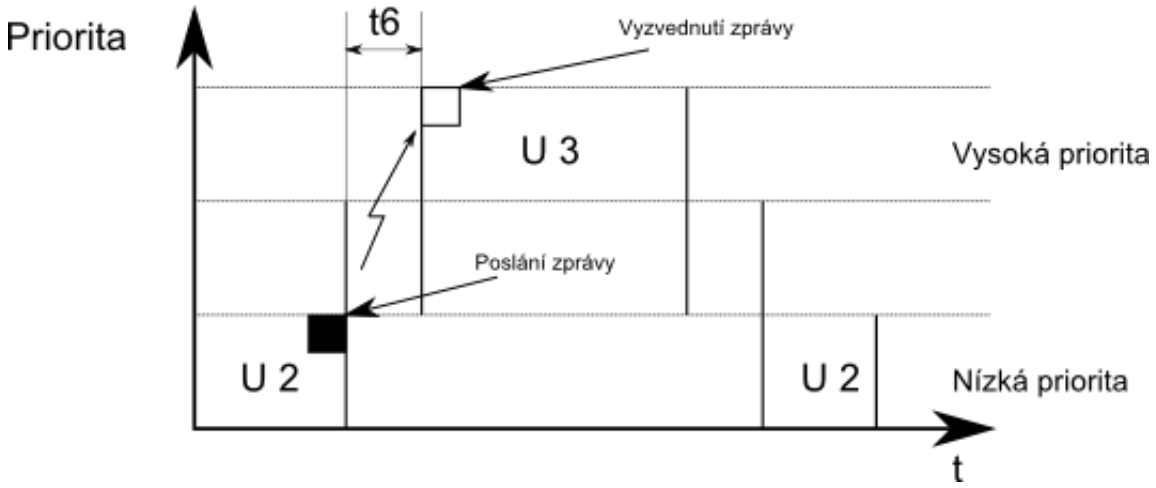
4.4.5 Vyřešení prioritní inverze (Deadlocku)

V původním návrhu nese tato hodnota název vyřešení deadlocku. Avšak tak, jak je definovaná, se ve skutečnosti jedná o vyřešení prioritní inverze. Tento problém, jak již bylo napsáno výše, je řešen většinou prioritní dědičností. V programu musí běžet alespoň tři úlohy s různými prioritami. Úlohy s nejnižší a nejvyšší prioritou sdílejí pomocí mutexu jeden společný zdroj. Dobou vyřešení prioritní inverze se myslí čas, který zabere aplikace prioritní dědičnosti na úlohu s nejnižší prioritou a získání přístupu ke zdroji (mutexu) úlohou s nejvyšší prioritou. Předpokládá se, že úloha s nižší prioritou předtím měla přístup ke sdílenému zdroji (vlastnila mutex) [12]. Tento jev je zobrazen na obrázku č. 8.

4.4.6 Doba předání zprávy

Cílem této hodnoty je poskytnutí informace o tom, jaké je zpoždění při poslání dat z jedné úlohy do druhé. K tomu slouží v RTOS většinou fronty zpráv (dále např. mailbox atd.). Jedná se o čas při poslání zprávy o nenulové velikosti. Měření probíhá tak, že úloha odesílající zprávu je po jejím odeslání okamžitě vystřídána úlohou s vyšší prioritou, jež je cílem zprávy. Podmínkou pro použití tohoto mechanismu je nutnost zabránit tomu, aby při posílání více zpráv nedocházelo k přepsání ještě nevyzvednutých. Došlo by totiž k jejich ztrátě [12].

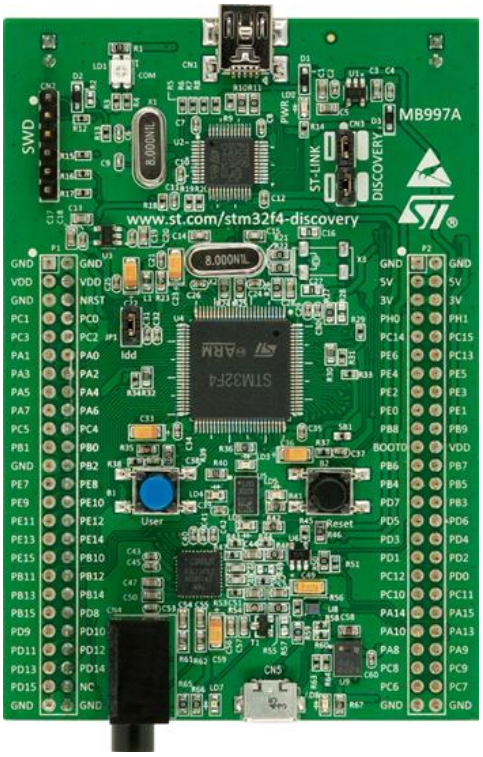
Na obrázku č. 23 je doba přenosu zprávy zobrazena jako t6.



Obr. č. 23 Zpoždění při posílání zprávy

5 Testované aplikace

Aplikace je realizována na testovací desce STM32F4-Discovery [13] od firmy STMicroelectronics. Tato deska obsahuje mikrokontrolér STM32F407VG, který běží na frekvenci 168 MHz, debuggovací rozhraní ST-LINK/V2, MEMS digitální akcelerometr, MEMS digitální mikrofon, USB mikro konektor, čtyři uživatelské LED diody a dvě tlačítka. Jedno tlačítko slouží pro resetování desky a druhé je programovatelné uživatelem. Deska je zobrazena na obrázku č. 24.



Obr. č. 24 STM32F4-Discovery testovací deska

Měření bylo provedeno ve dvou částech. Nejdříve byly napsány krátké programy pro

změření jednotlivých částí Rhealstone, pro každou hodnotu jeden. Jako druhá část byla realizována aplikace pro měření akcelerace. V ní se změřila většina hodnot Rhealstonu v „reálném“ provozu. Navíc byly v aplikaci změřeny doby nutné pro vytvoření základních objektů RTOS.

K měření doby byl použit jeden z dostupných čítačů. Čítač byl nastaven tak, aby běžel na maximální frekvenci 168 MHz. Bylo tak dosaženo maximálního rozlišení 5,9 ns.

5.1 Měření Rhealstone

Měření rhealstone bylo rozděleno do pěti programů, každý pro zjištění jedné hodnoty. Každý program je složen z několika úloh pro měření.

Ve funkci main jsou inicializovány potřebné periferie (GPIO, čítač, UART a NVIC) a RTOS, jsou vytvořeny úlohy UARTTask a TaskC (neplatí pro interupt.c) a nakonec je spuštěn RTOS.

Ukázka inicializace systému, vytvoření úloh a spuštění RTOS pro CoOS (u FreeRTOS obdobné, jen bez inicializace) je níže.

```
CoInitOS ();                          // Inicializace CooCox CoOS

//vytvoreni ulohy
CoCreateTask (TaskC, 0, prio,&TaskC_stk[STACK_SIZE_TASK-1], STACK_SIZE_TASK);
//vytvoreni ulohy
CoCreateTask (UARTTask, 0, prio + 4,&UARTTask_stk[STACK_SIZE_TASKC-1], STACK_SIZE_TASKC);

CoStartOS ();                     // Start multitasku
```

UARTTask je úlohou, nacházející se ve všech těchto programech (kromě interupt.c), pro vypsání výsledků přes UART do PC. Má ze všech úloh nejnižší prioritu, čeká tedy na dokončení měření. Až proběhnou všechny úkony spojené s měření, ujme se procesoru a vypíše výsledky měření.

TaskC je úloha, jež je také použita ve všech programech. Je to úloha, v níž probíhá samotné spočtení času a jsou v ní vytvořeny všechny měřené úlohy a případně další objekty dle potřeby měření (semafor, fronta zpráv). V této úloze vždy nejdříve proběhne změření, jak dlouho trvá průběh měřených úloh bez dané činnosti. Poté jsou vytvořeny dané úlohy provádějící požadovanou činnost a tato úloha sníží svoji prioritu, aby se mohly vykonat. Až skončí jejich vykonávání, opět se ujme procesoru a dokončí výpočet (neplatí pro interupt.c). Ukázka pro **switch.c** je níže. Jen s malými úpravami platí i pro ostatní.

```
void TaskC (void* pdata)
{
  for (;;) {
       //mereni bez prepinani
       pom1 = TIM_GetCounter(TIM2);           //zacatek mereni
       for (cnt1 = 0; cnt1 < MAXLOOPS; cnt1++)
             ;
       for (cnt2 = 0; cnt2 < MAXLOOPS; cnt2++)
             ;
       pom2 = TIM_GetCounter(TIM2);           //konec mereni

       //spocteni ticku kolik zabralo 1. mereni
       if (pom2 < pom1)
             cpom = pom2 + (0xFFFFFFFF - pom1 + 1);
       else
             cpom = pom2 - pom1;

       //vztvoreni dvou stejnzch uloh pro mereni

       TaskID2 = CoCreateTask (TaskA, 0, prio + 1, &TaskA_stk[STACK_SIZE_TASK-1],
       STACK_SIZE_TASK);   //vytvoreni ulohy
       TaskID3 = CoCreateTask (TaskB, 0, prio + 1, &TaskB_stk[STACK_SIZE_TASK-1],
       STACK_SIZE_TASK);   //vytvoreni ulohy

       pom1 = TIM_GetCounter(TIM2);           //zacatek mereni
       //nastaveni nizsi priority aby mohli bezet merene ulohy
       CoSetPriority (TaskID, prio + 2);
       //yeild
       CoYield();
       pom2 = TIM_GetCounter(TIM2);           //konec mereni

       //vypocet kolik ticku je rozdil mezi prvnim a druhym merenim
       if (pom2 < pom1)
             celkTick = (pom2 + (0xFFFFFFFF - pom1 + 1)) - cpom;
       else
             celkTick = (pom2 - pom1) - cpom;

       //vypocet casu nutneho pro prepnuti
       celkCas = ((float)celkTick / 168000000) / ((float)MAXLOOPS * 2);

       GPIO_SetBits(GPIOD, GPIO_Pin_14);
       CoExitTask ( );
  }
}
```

Switch.c slouží pro zjištění doby nutné k přepnutí úloh se stejnou prioritou (obrázek č. 19). Je tvořen třemi úlohami. TaskA a TaskB jsou prázdné úlohy se stejnou prioritou, jež se po určený počet iterací střídají v běhu. Vždy když jedna získá procesor, okamžitě jej předává druhé. V TaskC je spočítána doba průběhu prázdných úloh bez přepínání a následně s přepínáním. Z rozdílů těchto časů je spočtena doba přepnutí. Zdrojový kód je na přiloženém CD k dispozici ve složkách Rhealstone\CoOS\Switch a Rhealstone\FreeRTOS\Switch.

Preemption.c je použit pro změření času při přepnutí úloh s rozdílnou prioritou (obrázek č. 20). Je opět tvořen třemi úlohami. TaskA je úloha s nižší prioritou, ve které probíhá zpožďovací smyčka, jež musí být delší než je doba blokace úlohy TaskB. TaskB je úloha s vyšší prioritou, která střídá TaskA, probere se vždy za určitý čas, přebere kontrolu nad procesorem a okamžitě se opět zablokuje. Tyto úlohy běží konečný počet iterací. V TaskC jsou spočtena doba přepínání úplně stejným způsobem jako v switch.c. Ve výsledném čase je také započteno přepnutí zpět do TaskA a musí se od času odečíst (zjištěno v switch.c). Zdrojový kód je na přiloženém CD k dispozici ve složkách Rhealstone\CoOS\Preemption a Rhealstone\FreeRTOS\Preemption.

Shuffle.c je program pro změření, jak dlouho trvá vyřešení prohození semaforu (obrázek č. 22). Měření je složeno ze tří úloh. TaskA a TaskB jsou shodné úlohy se stejnou prioritou. Úloha se pokusí získat semafor. Pokud se jí to povede, okamžitě se vzdá procesoru. Pokud se jí ho získat nepodaří, zablokuje se při čekání na semafor. Nyní se pokusí získat Semafor stejným způsobem druhá úloha. Následně má-li úloha semafor a běží, tak semafor uvolní a opět se vzdá procesoru ve prospěch druhé úlohy. To se opakuje daný počet iterací. V Task C je spočtena doba trvání průběhu úloh bez a následně se získáváním a uvolňováním semaforu. Z rozdílu je opět zjištěna doba trvání prohazování semaforu. Zdrojový kód je na přiloženém CD k dispozici ve složkách Rhealstone\CoOS\Shuffle a Rhealstone\FreeRTOS\Shuffle.

V **message.c** se měří doba přenosu zprávy přes frontu zpráv (obrázek č. 23). Zpráva je typu integer a ve frontě je místo jen na jednu zprávu. V TaskA je zpráva odeslána. TaskB je v blokovaném stavu a čeká na zprávu. Vyskytne-li se ve frontě zpráva, přijme ji, pokusí se vyčíst další a zablokuje se. Následně TaskA opět vyšle zprávu. TaskA má nižší prioritu než TaskB. TaskC je shodná s předchozími případy. Ve výsledné době je započteno přepnutí zpět do úlohy TaskA a je nutné s ním počítat. Zdrojový kód je na přiloženém CD k dispozici ve složkách Rhealstone\CoOS\Message a Rhealstone\FreeRTOS\Message.

Interupt.c je prográmek pro změření latence přerušení v RTOS a mírně se liší od předchozích. Je složen ze dvou úloh, semaforu a obsluhy přerušení. Ve funkci main je opět

provedena nutná inicializace periferií, jsou vytvořeny obě úlohy (TaskC a ISRTask) a prázdný semafor, nakonec je spuštěn RTOS. Program neměří latenci původně definovanou v návrhu Rhealstone (na obrázku č. 21 čas t3a). Měří ji od výskytu přerušení až po získání semaforu obslužnou úlohou ISRTask a začátek jejího vykonávání (kapitola 1.3 a na obrázku č. 21 čas t3b). Výpis přes výsledku měření přes UART, probíhá v obslužné úloze. Pro generaci požadavku na přerušení slouží vnější přerušení generované softwarově v úloze TaskC na jednom z pinů. Zdrojový kód je na přiloženém CD k dispozici ve složkách Rhealstone\CoOS\Interupt a Rhealstone\FreeRTOS\Interupt. Níže je kód inicializace v CoOS pro tento program včetně vytvoření semaforu (obdobné pro FreeRTOS).

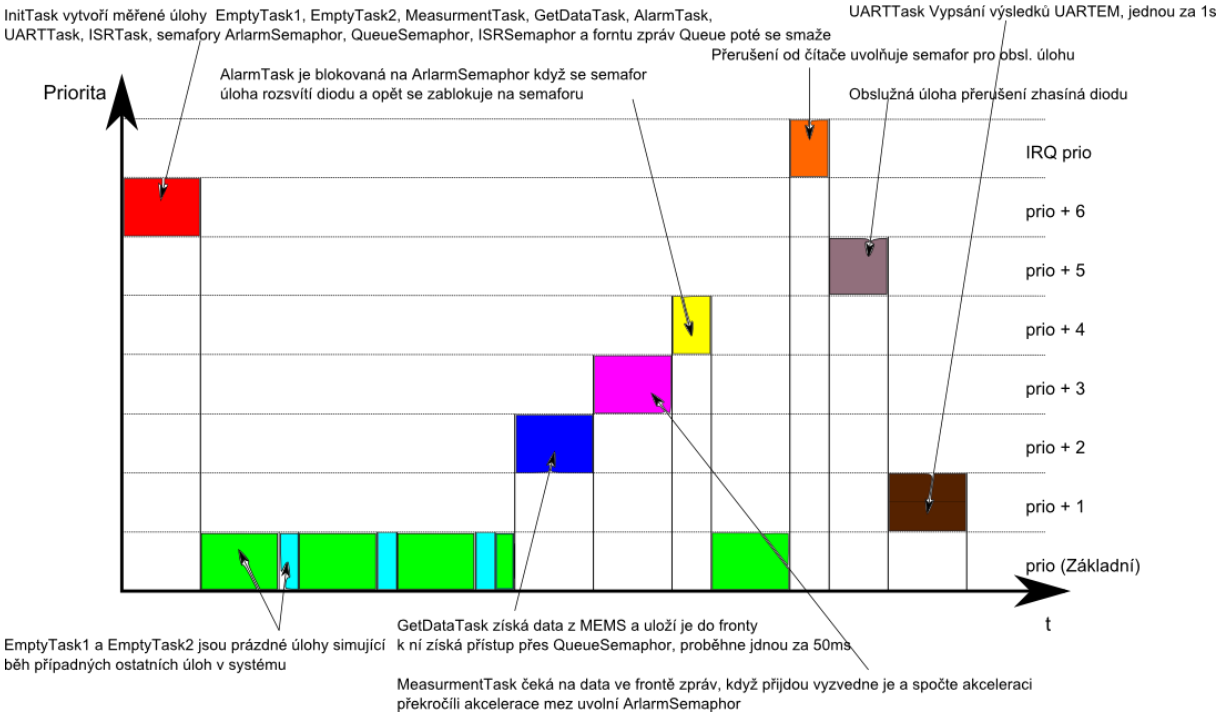
```
CoInitOS ();                        // Inicializace CooCox CoOS

Sem = CoCreateSem (0, 1, EVENT_SORT_TYPE_FIFO);      //synchronizacni semafor
TaskID   =   CoCreateTask   (ISRTask,   0,   prio,&ISRTask_stk[STACK_SIZE_TASK-1],
STACK_SIZE_TASK);   //vytvoreni ulohy
TaskID   =   CoCreateTask   (TaskC,   0,   prio   +   1,&TaskC_stk[STACK_SIZE_TASK-1],
STACK_SIZE_TASK);   //vytvoreni ulohy
TIM_Cmd(TIM2, ENABLE);     //povoleni citace

CoStartOS ();                  // Start multitasku
```

5.2 Testovací aplikace

Aplikace pomocí MEMS senzoru kontroluje polohu testovacího kitu. V případě vychýlení nebo náhlého pohybu dojde k zhasnutí blikající diody. Program se skládá z osmi úloh. Přibližný průběh aplikace je vidět na obrázku č. 25. Zdrojový kód programu se nachází ve složce Test\CoOS nebo Test\FreeRTOS na přiloženém CD.



Obr. č. 25 Diagram testovací aplikace

Program využívá MEMS akcelerometr pro měření změny polohy desky, LED diody pro indikaci běhu programu a změny polohy. Z periferií mikrokontroléru jsou použity GPIO piny, NVIC kontrolér přerušení, čítač pro měření časové latence, SPI pro komunikaci s MEMS akcelerometrem, UART pro zobrazování výsledků měření na PC. V main funkci jsou inicializovány potřebné periferie, RTOS a je vytvořena úloha InitTask, nakonec je spuštěn RTOS. Podobně jako v předchozích ukázkách.

V první úloze InitTask jsou vytvořeny ostatní úlohy a objekty používané programem. Mezi ně patří tři semafory a fronta zpráv. Úloha má vyšší prioritu než úlohy, které vytváří. Kdyby měla nižší prioritu, spustily by se tyto úlohy a již by se nevytvořily ostatní objekty. Je zde měřena doba nutná k vytvoření úlohy, semaforu a fronty zpráv. Tato úloha se po svém průběhu ukončí a smaže.

EmptyTask1 a EmptyTask2 jsou úlohy s nejnižší prioritou v celé aplikaci. Tato priorita je brána jako základní a dále je na ní odkazováno. Neprobíhá zde žádná činnost, kromě měření. Úlohy se střídají po 10ms. Simulují případné další úlohy s nižší prioritou v systému a slouží pro změření latence při přepínání se stejnou prioritou.

UARTTask se spustí jednou za 1s a pouze vypíše výsledky z měření na obrazovku počítače, jinak se měření neúčastní. Má prioritu o 1 vyšší než předchozí.

GetDataTask slouží k vyzvednutí dat z MEMS akcelerometru. GetDataTask běží na prioritě o 2 vyšší než je základní. Úloha se spustí jednou za 50ms a při každém svém

průběhu zkontroluje, zda jsou v MEMS k dispozici nová data o akceleraci. Pokud ano, tak tato data vyzvedne. Získáním semaforu QueueSemaphore dostane úloha přístup k frontě zpráv. Vyzvednutá data uloží do fronty Queue a poté odevzdá semafor zpět pro další použití. V této úloze probíhá měření doby přepnutí mezi úlohami s různou prioritou, latence získání a uvolnění semaforu. Začíná zde také měření doby přenosu zprávy mezi dvěma úlohami.

V MeasurmentTask probíhá výpočet akcelerace ze získaných hodnot a porovnání s danou mezí. Úloha má prioritu o 3 vyšší než základní a o 1 vyšší než GetDataTask. MeasurmentTask čeká na příchozí data ve frontě zpráv Queue. Po jejich přijmutí z nich spočte akceleraci pro každou osu a porovná, zda nedošlo k překročení meze. Překročí-li akcelerace v jakékoliv ose danou mez, je uvolněn semafor AlarmSemaphore. V této úloze je dokončeno měření přenosu zprávy, dále je zde započato měření při prohození semaforů.

AlarmTask má prioritu o 4 vyšší než je základní. Její funkce je prostá. Pouze rozsvěcuje diodu. Úloha je zablokována na semaforu AlarmSemaphore. Je-li semafor v MeasurmentTask uvolněn, spustí se AlarmTask díky svojí vysoké prioritě. Nastaví bit na rozsvícení diody a díky tomu, že běží v nekonečné smyčce, se znovu pokusí získat semafor. Na něm se zablokuje a čeká na jeho opětovné uvolnění. Mezi tím běží úlohy popsané v předchozím textu. Je zde dokončeno měření doby, jakou zabere prohození semaforů.

ISRTask je obslužná úloha s vysokou prioritou pro přerušení od čítače. Její priorita je základní priorita + 5. Úloha čeká na uvolnění semaforu ISRSemaphore. Ten je uvolňován v obslužné rutině přerušení. V úloze je pouze zhasínána dioda a měřena latence přerušení. Následuje kód obslužné rutiny a úlohy ISRTask pro CoOS.

```
//obsluha preruseni
void TIM3_IRQHandler(void)
{
	/*dava CoOS spolu s CoExitISR() vedet ze se budou pouzivat jeho API
	v preruseni*/
	CoEnterISR ( );
	TIM_ClearITPendingBit(TIM3, TIM_IT_Update);   //schozeni pending bitu

	chyba = isr_PostSem (ISRSemaphore);     //uvolneni semaforu pro obsl. ulohu
	if (chyba != E_OK) //kontrola jestli se uvolneni podarilo
	{
		if (chyba == E_SEV_REQ_FULL)
		{
			isr = 1;
		}
	}
	CoExitISR ( );      //konec preruseni
}
```

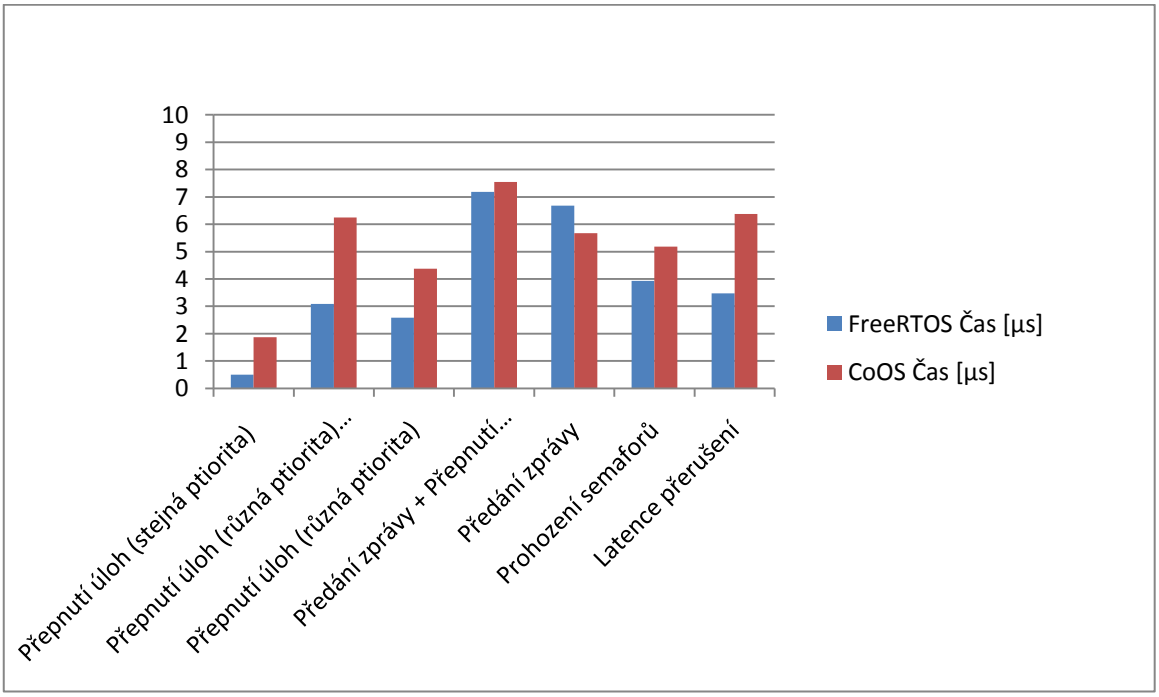
```
//obsluzna uloha
void ISRTask (void* pdata)
{
	for (;;)
	{
		CoPendSem (ISRSemaphore, 0);	//ceka na semafor nekonecne dlouho
		vysledek[7] = (TIM_GetCounter(TIM3) * 6); //mereni latence preruseni
		GPIO_ResetBits(GPIOD, GPIO_Pin_13); //zhasne diodu
	}
}
```

6 Výsledky měření

6.1 Rhealstone

Název	FreeRTOS Čas [µs]	CoOS Čas [µs]
Přepnutí úloh (stejná priorita)	0,499	1,868
Přepnutí úloh (různá priorita) + Přepnutí úloh (stejná priorita)	3,087	6,242
Přepnutí úloh (různá priorita)	2,588	4,374
Předání zprávy + Přepnutí úloh (stejná priorita)	7,180	7,542
Předání zprávy	6,681	5,674
Prohození semaforů	3,927	5,183
Latence přerušení	3,468	6,372

Tab. č. 1 Výsledky měření Rhealstone pro FreeRTOS a CoOS



Obr. č. 26 Graf výsledků měření Rhealstone pro FreeRTOS a CoOS

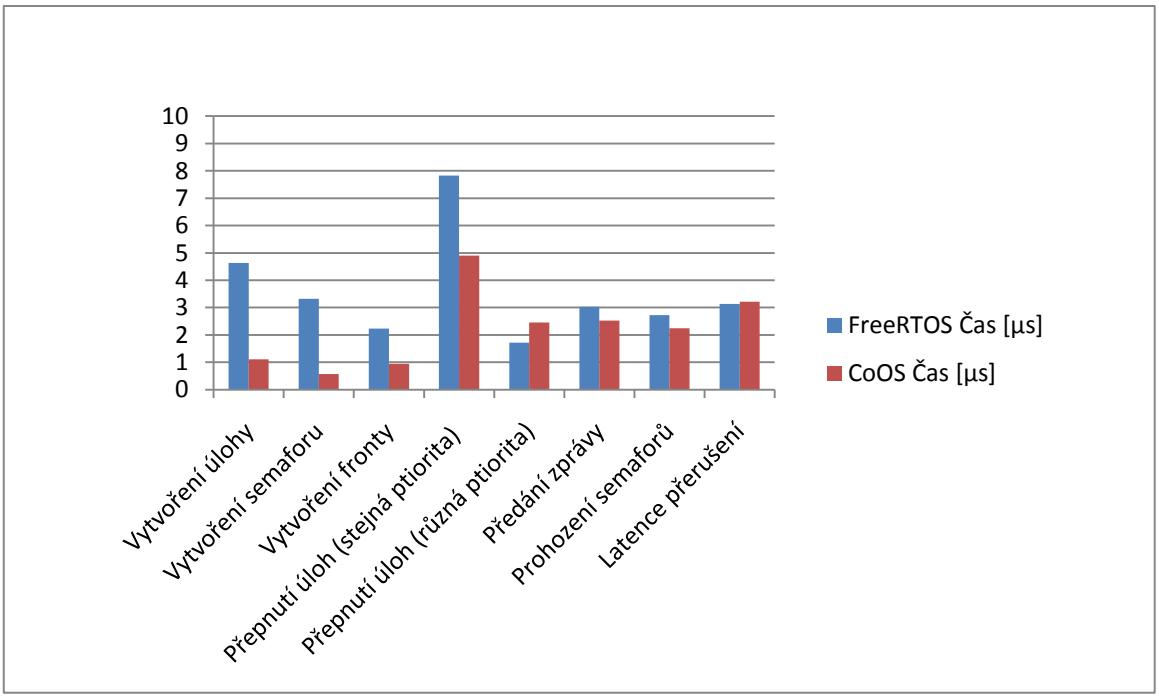
Název	FreeRTOS	CoOS
Výsledné číslo Rhealstone [Rhealstony/s]	3082436	1290073

Tab. č. 2 Hodnota Rhealstone při měření každé hodnoty zvlášť

6.2 Testovací aplikace

Název	FreeRTOS Čas [µs]	FreeRTOS ticks	CoOS Čas [µs]	CoOS ticks
Vytvoření úlohy	4,632	772	1,104	184
Vytvoření semaforu	3,324	554	0,576	96
Vytvoření fronty	2,232	372	0,948	158
Přepnutí úloh (stejná priorita)	7,824	1304	4,896	816
Přepnutí úloh (různá priorita)	1,716	286	2,460	410
Předání zprávy	3,024	504	2,520	420
Prohození semaforů	2,724	454	2,238	373
Latence přerušení	3,138	523	3,210	535

Tab. č. 3 Výsledky měření v testovací aplikaci pro FreeRTOS a CoOS



Obr. č. 27 Graf výsledků měření v testovací aplikaci pro FreeRTOS a CoOS

Název	FreeRTOS	CoOS
Výsledné číslo Rhealstone [Rhealstony/s]	1727032	1765931

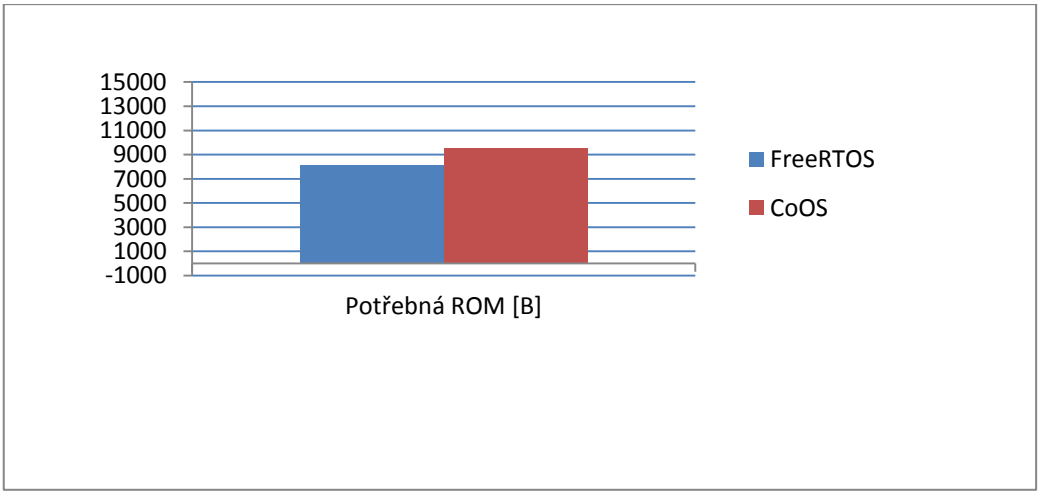
Tab. č. 4 Hodnota Rhealstone při měření v testovací aplikaci

6.3 Paměťové nároky RTOS

První byla zjištěna velikost programu pro projekt bez přítomnosti RTOS. Projekt obsahoval prázdný soubor main.c, dále nutné soubory startup_stm32f40xx.s, core_cm4.h, core_cm4_simd.h, core_cmFunc.h, core_cmInstr.h, stm32f4xx.h a konfigurační soubor stm32f4xx_conf.h. Následně byl do projektu přidán RTOS a znovu se zjistilo místo potřebné programem. Samotná velikost paměti potřebné pro RTOS se zjistila odečtením první naměřené hodnoty od hodnoty při připojeném RTOS.

	Bez RTOS	CoOS	CoOS bez CMSIS	FreeRTOS	FreeRTOS bez CMSIS
Potřebná ROM [B]	1320	10828	9508	9448	8128
Potřebná RAM [B]	1656	2840	1184	78848	77192

Tab. č. 5 Paměťové nároky RTOS



Obr. č. 28 Potřebná ROM pamět pro RTOS

Závěr

Cílem této diplomové práce bylo porovnat vybrané RTOS, jež jsou dostupné pro mikrokontroléry STM32.

Pro porovnání byly vybrány CoOS, FreeRTOS a testování bylo realizováno na testovací desce STM32F4-Discovery s mikrokontrolérem STM32F407VG, běžícím na frekvenci 168 MHz. Jedná se o volně dostupné RTOS s dobrou podporou. Porovnány byly z hlediska naměřených hodnot latence služeb, velikosti zabrané paměti a z uživatelského hlediska.

Výsledky měření zpoždění služeb systému jsou zobrazeny v kapitole 6, v tabulkách č. 1 až č. 4 a na grafech v obrázcích č. 26 a č. 27. Rozdíly časů při měření rhealstone ve zvláštních programech a při měření v testovací aplikaci jsou značné. Rozdíly vznikly především z důvodu rozdílného způsobu měření.

Hodnoty v tabulce č. 1 byly všechny, kromě latence přerušení, získány měřením určitého počtu iterací programu, který prováděl měřenou činnost. Z této celkové doby byla spočtena doba dané činnosti. Toto měření je zatíženo dobou nutnou pro přepnutí z měřící úlohy do měřených úloh a zpět. Vzhledem k tomu, že se jedná o probíhající program, dochází také k probouzení plánovače RTOS, jež se stará o průběh aplikace. V některých případech (přepnutí úloh s různou prioritou a doba poslání zprávy) je ve výsledné době také přítomno přepnutí zpět do úlohy s nižší prioritou. Latence přerušení byla změřena jednou pomocí softwarového přerušení jako absolutní hodnota. Z výsledků těchto měření vychází, kromě doby poslání zprávy, lépe FreeRTOS.

V případě testovací aplikace se vždy jednalo o změření absolutní doby, kterou jedna činnost zabere. Měření započalo vždy na začátku činnosti a skončilo při jejím ukončení. Toto měření tak podává lepší informaci o tom, jak dlouho vykonání měřené činnosti trvá. Výsledky jsou zobrazeny v tabulce č. 3. I zde jsou některá měření zatížena chybou. Ta vznikla z požadavku na měření ve funkční aplikaci, a tak muselo dojít k několika kompromisům. Týká se to hlavně měření prioritní inverze, které by bylo v průběhu aplikace jen těžko měřitelné. Ke druhému kompromisu došlo při měření doby prohození semaforů, kde je měřena jen doba t4b z obrázku č. 22. Z tohoto měření vychází lépe CoOS ve většině případů, kromě přepnutí úloh s různou prioritou a latence přerušení. Je nutné taky vyzdvihnout rozdíl dob přepnutí se stejnou prioritou mezi tímto měřením a měřením předchozím. V samostatném měření této činnosti, jak je popsáno v kapitole 5.1, se měří jen doba trvání samotného přepnutí. V testovací aplikaci je změřena doba od konce časového úseku jedné úlohy až po začátek vykonávání úlohy druhé podle definice round-robin. V měřené době je tak

zohledněno také přerušení od SYSTICKu a arbitráž RTOS systému. V testovací aplikaci byla také změřena doba nutná pro vytvoření základních objektů RTOS. Zde se ukázal lepší CoOS, jehož doby pro vytvoření úlohy, semaforu a fronty zpráv byly několikanásobně nižší než u FreeRTOS. To je z části způsobeno tím, že CoOS alokuje místo pro řídící bloky staticky.

Z výsledků těchto dvou měření se dá soudit, že zatímco CoOS má lépe odladěné samotné činnosti, FreeRTOS je lepší v běžné arbitráži.

Jak je psáno v kapitole 3.2 a 3.3, oba RTOS jsou modulární. Porovnání paměti bylo provedeno při přítomnosti všech dostupných služeb a objektů. Z výsledků zobrazených v tabulce č.5 a na obrázku č. 28 vyplynulo, že FreeRTOS má nižší nároky na paměť ROM než CoOS. Velikost potřebné RAM je ještě více závislá na nastavení RTOS a je uvedena jen pro informaci. V RAM paměti jsou alokovány potřebné systémové a uživatelské haldy. Její nutné množství se odvíjí od nastavení v konfiguračních souborech jednotlivých RTOS.

Z hlediska programátora není porovnání úplně jednoznačné. Oba systémy mají své silné a slabé stránky. CoOS nabízí jen nejnutnější minimum funkcí při práci s jádrem a objekty. Tento nedostatek se projeví znatelně především při práci s funkcemi RTOS v přerušení. CoOS umožňuje v ISR provést jen uvolnění semaforu, odeslání zprávy a nastavení příznaku, jiné činnosti v přerušení RTOS nepodporuje. Na druhou stranu, právě díky nízkému počtu funkcí, lze do programování s CoOS velmi rychle a snadno proniknout. To z tohoto systému činí vhodný nástroj pro začátečníky v programování pod RTOS. U FreeRTOS je situace odlišná. Tento operační systém disponuje značným množstvím funkcí pro práci s objekty a jádrem. To platí i pro funkce podporující použití v ISR. FreeRTOS umožňuje nejen to samé jako CoOS, ale může také v přerušení přijímat zprávy, získávat semafory a další činnosti. Při programování pod FreeRTOS je od programátora vyžadována větší dávka pozornosti než u CoOS. Zatímco CoOS má některé činnosti zabudovány již ve svých funkcích, ve FreeRTOS se o to musí postarat programátor sám. Například CoOS při vytvoření semaforu umožňuje rozhodnout se, zda-li jej bude možné získat nebo bude nedostupný, ve FreeRTOS se vždy vytvoří připravený pro získání. Jestliže jej programátor zapomene po vytvoření znedostupnit, může tak vyvolat potíže v programu, pokud je semafor používán k synchronizaci.

Celkově lze oba RTOS zhodnotit následovně. CoOS je vhodný pro programátory, kteří s RTOS začínají a pro užití v malých jednoduchých aplikacích, které často využívají služeb systému. FreeRTOS je vhodnější pro uživatele, kteří již mají nějakou zkušenost při programování pod RTOS a pro větší složitější aplikace, jež využijí množství dostupných funkcí.

Seznam tabulek a obrázků:

Tab. č. 5 Paměťové nároky RTOS

Použitá literatura

[1] LI, Qing a YAO, Carolyn. *Real-Time Concepts for Embedded Systems.* San Francisco: CMP Books, 2003. ISBN 1-57820-124-1

[2] *Architecture of Device I/O Drivers* [online]. [Cit. 12. 3. 2014]. Dostupné z: http://www.kalinskyassociates.com/Wpaper4.html

[3] YIU, Joseph. *The Definitive Guide to the ARM Cortex-M3.* Burlington: Newnes, 2010. ISBN 978-0-12-382090-7

[4] *RTOS list* [online]. [Cit. 12. 3. 2014]. Dostupné z: http://www.embeddedcraft.org/listrtos.html

[5] *VxWorks* [online]. [Cit. 12. 3. 2014]. Dostupné z: http://www.windriver.com/

[6] *SafeRTOS* [online]. [Cit. 12. 3. 2014]. Dostupné z: http://www.highintegritysystems.com/safertos/

[7] *eCos* [online]. [Cit. 12. 3. 2014]. Dostupné z: http://www.ecos.sourceware.org/

[8] *FreeRTOS* [online]. [Cit. 12. 3. 2014]. Dostupné z: http://www. freertos.org

[9] *Coocox CoOS* [online]. [Cit. 12. 3. 2014]. Dostupné z: http://www.coocox.org/CoOS.htm

[10] *ChibiOS/RT* [online]. [Cit. 12. 3. 2014]. Dostupné z: http://www.chibios.org/dokuwiki/doku.php

[11] *TNKernel* [online]. [Cit. 12. 3. 2014]. Dostupné z: http://www.tnkernel.com/

[12] *RHEALSTONE: A REAL-TIME BENCHMARKING PROPOSAL* [online]. [Cit. 12. 3. 2014]. Dostupné z: http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/DDJ/1989/8902/8902a/8902a.htm

[13] *STM32F4DISCOVERY* [online]. [Cit. 12. 3. 2014]. Dostupné z: http://www.st.com/web/catalog/tools/FM116/SC959/SS1532/LN1848/PF252419