Jednoduchý realtime operační systém pro mikropočítač HCS08

Simple realtime operating system for HCS08 microcontroller

Richard Červinka

Bakalářská práce
2010

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2009/2010

ZADÁNÍ BAKALÁŘSKÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Richard ČERVINKA**
Osobní číslo:	**A06140**
Studijní program:	**B 3902 Inženýrská informatika**
Studijní obor:	**Informační a řídicí technologie**
Téma práce:	**Jednoduchý realtime operační systém pro mikropočítač HCS08**

Zásady pro vypracování:

1. **Seznamte se s mikropočítačem HCS08 a vývojovým kitem M68EVB908GB60, který se používá pro výuku programování mikropočítačů na naší fakultě.**
2. **Navrhněte strukturu operačního systému pro tento mikropočítač s využitím struktury již existujícího systému RTMON, využívaného na naší fakultě.**
3. **Realizujte navržený operační systém.**
4. **Vytvořte ukázkovou aplikaci s využitím vytvořeného operačního systému.**
5. **Zpracujte uživatelskou dokumentaci k vytvořenému systému.**

Rozsah bakalářské práce:

Rozsah příloh:

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

Seznam odborné literatury:

1. **MC9S08GB/GT Data Sheet [online]. Freescale Semiconductor, 2004 [cit. 2009-01-26]. Dostupný z WWW: http://www.freescale.com/files/microcontrollers/doc/data_sheet/MC9S08GB60.pdf.**
2. **HCS08 Family Reference Manual [online]. Freescale Semiconductor, 2007 [cit. 2009-01-26]. Dostupný z WWW: http://www.freescale.com/files/microcontrollers/doc/ref_manual/HCS08RMV1.pdf.**
3. **CPU08 Central Processor Unit Reference manual [online]. Freescale Semiconductor, 2006 [cit. 2009-01-26]. Dostupný z WWW: http://www.freescale.com/files/microcontrollers/doc/ref_manual/CPU08RM.pdf.**
4. **VÁŇA, V. Začínáme s mikrokontroléry Motorola HC08 Nitron. Praha : BEN -- technická literatura, 2003. 96 s. ISBN 80-7300-124-1.**
5. **MANN, B. C pro mikrokontroléry. Praha : BEN - technická literatura, 2004. 280 s. ISBN 80-7300-077-6.**
6. **SROVNAL, V. Operační systémy pro řízení v reálném čase, VŠB Technická universita Ostrava 2003, ISBN 80-248-0503-0.**
7. **SOLOMON, D. A. Windows NT pro administrátory a vývojáře. Praha : Computer Press, 2000. 492 s. ISBN 80-7226-147-9.**

Vedoucí bakalářské práce: **Ing. Jan Dolinay**
Ústav automatizace a řídicí techniky

Datum zadání bakalářské práce: **5. března 2010**

Termín odevzdání bakalářské práce: **1. června 2010**

Ve Zlíně dne 5. března 2010

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.

prof. Ing. Vladimír Vašek, CSc.
děkan

doc. Ing. Ivan Zelinka, Ph.D.
ředitel ústavu

ABSTRAKT

Cílem práce je vytvořit operační systém reálného času pro mikropočítač řady HCS08. Systém musí podporovat standardní operace s procesy a komunikaci mezi nimi. Teoretická část práce se věnuje popisu mikropočítačů a operačních systémů obecně. Praktická část se zaměřuje na popis klíčových částí vytvořeného systému a vysvětluje principy programování pod operačním systémem. V závěru praktické části je přehled služeb vytvořeného systému.

Klíčová slova: mikropočítač, HCS08, operační systém, reálný čas

ABSTRACT

The aim of the project is to create real-time operating system for microcontroller HCS08. The system must support standard operations with processes and communication between them. The theoretical part of the work deals with the explanation of microcomputers and operating systems in general. The practical part focuses on the description of the key parts of the system created and explains the principles of programming under the operating system. In conclusion of the practical part is an overview of the services created by the system.

Keywords: microcontroller, HCS08, operating system, realtime

"Společným nedostatkem všech děl je jejich přílišná délka."

[VAUVENARGUES, Réflexions, 628 (1746)]

Chtěl bych poděkovat svému vedoucímu bakalářské práce Ing. Janu Dolinayovi za cenné rady a připomínky a za trpělivé a důsledné vedení práce.

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.

Ve Zlíně …….…….

podpis diplomanta

OBSAH

ÚVOD

Operační systémy jsou důležitým programovým vybavením počítačů a přinášejí nový rozměr v jejich používání a programování. Zavádějí řád při psaní různorodých aplikací takovým způsobem, že programy správně napsané pro stejný operační systém mohou bezpečně běžet společně na jednom počítačovém systému, aniž by docházelo k jejich vzájemnému nežádoucímu ovlivňování. Na druhou stranu operační systémy poskytují jednoduché a efektivní nástroje pro komunikaci mezi jednotlivými aplikacemi, které mohou být vyvíjeny na sobě zcela nezávisle. Jedinou podmínkou pro bezproblémový běh je správné využití aplikačního rozhraní, které operační systémy poskytují.

Počítačové systémy jsou nesmírně variabilní zařízení. Různorodost, jakou nacházíme v architekturách počítačů a v jejich využití, se odráží i v návrzích operačních systémů. Je rozdíl vytvářet operační systém pro počítače architektury PC a pro mikropočítač, který je součástí nějakého výrobního procesu. Cílem bakalářské práce je vytvořit operační systém reálného času určený pro mikropočítače řady HCS08. Protože může být výsledný systém použit při výuce programování mikropočítačů, měl by být, v rámci možností, co nejobsáhlejší a odolný proti nestandardnímu použití. Zároveň by měl mít co nejmenší nároky na samotný mikropočítač.

Teoretická část bakalářské práce je rozdělena na dvě kapitoly. První popisuje nejdůležitější části mikropočítačů a věnuje se především konkrétnímu typu MC9S08GB60. Je zde také stručný úvod do programování mikropočítačů se zaměřením na jazyk symbolických adres (Assembler). Druhá část se věnuje operačním systémům, jejich obecným vlastnostem a cílům. Protože mnohé zde popsané vlastnosti jsou součástí vytvořeného systému, nachází se jejich obsáhlejší popis v praktické části.

Praktická část je, stejně jako teoretická, rozdělena na dvě kapitoly. První se věnuje popisu klíčových součástí vytvořeného systému, včetně způsobu jejich implementace. Ve druhé kapitole jsou vysvětleny základy psaní programů běžících pod operačním systémem. Kapitola končí přehledem všech služeb systému a jejich stručným popisem.

I. TEORETICKÁ ČÁST

1 MIKROPOČÍTAČE

Význam pojmu mikropočítač se měnil s vývojem počítačů a technologií jejich výroby. Jeden z největších milníků vývoje počítačů byl vynález integrovaného obvodu. Rázem bylo možné vměstnat velké elektrické obvody do velmi malých a levných součástek. Zároveň se, díky stále se zmenšující technologii výroby, zvyšuje výkon logických obvodů a snižují jejich energetické nároky a cena. Rychlý vývoj velmi brzy umožnil integrovat všechny důležité součásti tvořící počítač do jednoho obvodu a tím vznikl jednočipový mikropočítač. Díky svému výkonu, univerzálnosti a nízké ceně, našli mikropočítače velmi rychle uplatnění v různých oborech lidské činnosti a staly se velmi rozšířenými.

Mikropočítač se skládá z těchto základních částí:

- mikroprocesor
- paměť
- sběrnice
- porty

Mikroprocesor je hlavní výpočetní jednotka celého mikropočítače, často označován jako CPU (Central Procesing Unit). Úkolem mikroprocesoru je zpracovávat instrukce, které jsou mu sekvenčně dodávány z operační paměti. Obsahuje aritmeticko-logickou jednotku ALU (Aritmetic Logic Unit), zpracovávající veškeré aritmetické a logické operace. Další důležitou součástí mikroprocesoru je řadič, který zajišťuje činnost mikroprocesoru v závislosti na programu. Dalšími bloky, které patří k mikroprocesoru, jsou například vyrovnávací paměť a pracovní registry pro ukládání mezivýsledků.

Paměť slouží k uchovávání programu a dat. Většina dnešních mikropočítačů vychází z von Neumannovy architektury. Počítač navržený podle této architektury má společnou paměť pro program i data. Pomalá komunikace s pamětí je nejslabším článkem celého konceptu. Existuje velké množství typů pamětí, které se odlišují rychlostí čtení a zápisu, přepisovatelností a závislostí na napájení. Von Neumannova architektura chápe paměť jako celek, který může být ovšem složen z různých typů pamětí.

Sběrnici je možné chápat jako sadu vodičů zajišťující komunikaci mezi jednotlivými součástmi mikropočítače. Sběrnice se dělí na adresovou, datovou a řídící část. Konstrukce sběrnice ovlivňuje množství zařízení, které je k ní možné připojit.

Porty jsou elektrické obvody zajišťující spojení mikropočítače s okolím. K portům můžeme připojit různá periferní zařízení a pomocí mikropočítače je ovládat, popřípadě zpracovávat jejich výstupy.

1.1 Mikropočítač řady HCS08 – MC9S08GB60

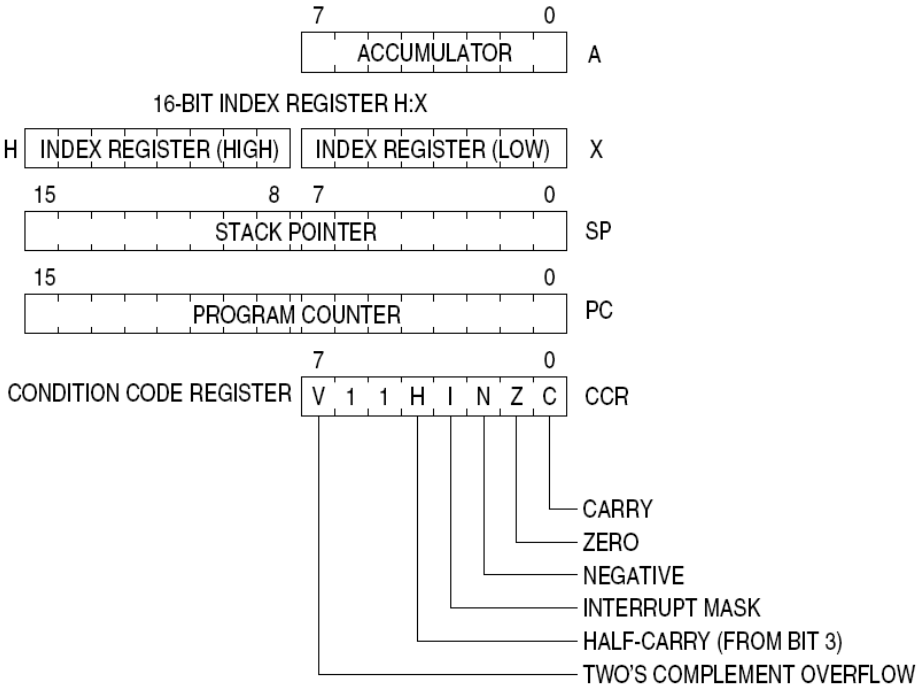
1.1.1 Základní informace

MC9S08GB60 je osmibitový jednočipový mikropočítač vyráběný firmou Freescale Semiconductor. Řada HCS08 je rozšířením řady HC08. Zde jsou některé vybrané vlastnosti mikropočítače.

- Mikroprocesor pracuje na maximální frekvenci 40 MHz.
- 20 MHz frekvence sběrnice
- Napájení v rozmezí 1,8 až 3,6 V
- Mikropočítač obsahuje 60 kB FLASH paměti a 4 kB paměti RAM
- Instrukční sada řady HC08 je rozšířena o BGND instrukce
- 16 bitové adresování.
- 3 kanálový a 5 kanálový, 16 bitový modul časovače
- 8 kanálový, 10 bitový AD převodník
- Dva SCI moduly pro sériovou komunikaci
- Sériový SPI modul
- KBI modul přerušení klávesnice
- Podpora režimu se sníženou spotřebou

1.1.2 Registry mikroprocesoru

Registr je velmi rychlá paměť, do které se ukládají například operandy aritmetických a logických operací, které vykonává procesor. Po provedení výpočtu jsou výsledky uloženy opět do registrů.



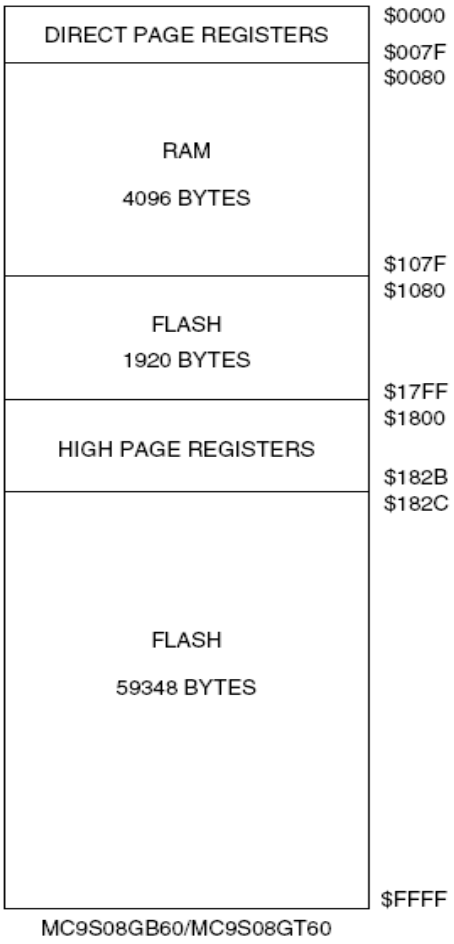
Obrázek 1 – registry [3]

Do registru A se nejčastěji ukládají operandy. Například při aritmetickém součtu se první operand uloží do registru A, následně je k němu přičten druhý operand a výsledek zůstává v registru A. Registr H:X je indexový registr. Do indexového registru se ukládají adresy, pomocí kterých se můžeme odkazovat na místa v paměti. Indexový registr je také rozdělený na dvě samostatné části – registr H a registr X. Registr X můžeme používat téměř stejným způsobem jako registr A. Práce s registrem H je kvůli zpětné kompatibilitě mikropočítače značně omezena. V registru SP (Stack Pointer) je adresa vrcholu zásobníku. SP ukazuje na místo v paměti následující bezprostředně za posledním prvkem zásobníku. V registru PC (Program Counter) je vždy uložena adresa následující instrukce. Například instrukce skoků fungují tak, že do registru PC zkopírují cílovou adresu. Příznakový registr (CCR) informuje nastavením svých bitů o událostech, jako je přetečení, záporná hodnota nebo práce s nulou. Jak s registry pracovat je popsáno v instrukční sadě.

1.1.3 Mapa paměti

Paměť je podle von Neumannovy architektury společná pro program i data. Znamená to, že program spolu s daty spadají do stejného adresovatelného prostoru, ale kvůli různým

požadavkům a vlastnostem pamětí je program ukládán do FLASH paměti a data do paměti RAM. Rozdělení paměťového prostoru je znázorněno na obrázku.



Obrázek 2 - Paměťová mapa [3]

1.1.4 Přerušení

Přerušení umožňuje mikropočítači reagovat na vnější události. Takovou událostí může být například přetečení časovače nebo přerušení generované při stisku tlačítka klávesnice. Přerušení je vyvoláno i při poklesu napájecího napětí pod kritickou hodnotu. Pokud dojde k vyvolání přerušení, je dokončena právě vykonávaná instrukce a poté je program přerušen. Mikropočítač uloží na zásobník stav registrů (kromě registru H, který se neukládá kvůli zpětné kompatibilitě se staršími typy mikropočítačů) a z vektoru přerušení vybere adresu patřičné obsluhy přerušení. V obsluze přerušení nemůže být vyvoláno jiné přerušení, dokud se neprovede návrat z obsluhy instrukcí RTI. Přerušení můžeme povolit také vynulováním

příznakového bitu „I.“ Po návratu se ze zásobníku obnoví stav registrů a program pokračuje dále.

1.1.5 Porty

Porty jsou řízeny registry umístěnými na začátku paměti RAM, která je na obrázku 2 označena jako Direct Page Registers. Jednotlivé piny portů odpovídají bitům řídícího registru. Porty jsou označeny písmeny A až G. V souboru MC9S08GB60.inc jsou definovány konstanty pro práci s registry. Každý port můžeme řídit pomocí třech registrů.

- PTxD – datový registr, jeho bity reprezentují aktuální stav portu
- PTxDD – řídící registr, slouží ke konfiguraci pinů portu mezi vstupním a výstupním režimem
- PTxPE – řídí připojení „pull-up“ rezistorů

Na místě „x“ v názvech registrů je vždy písmeno označující konkrétní port (například PTAD). Pull-up rezistory nacházejí využití při vstupní konfiguraci portu. Při výstupním nastavení je odpojujeme. Pokud například připojíme na pin portu A tlačítko, které není sepnuto, není stav pinu nijak definován. Pokud ovšem připojíme pull-up rezistory, je přes ně přivedeno napětí, které indikuje stav rozepnutého tlačítka. Pokud tlačítko sepneme, bude přes něj toto napětí uzemněno do GND, tím vznikne na portu nulové napětí, a to indikuje sepnuté tlačítko. Při práci s porty tedy logická jednička znamená neaktivní a logická nula aktivní pin. To samozřejmě platí i pro výstupní režim pinů portu.

1.2 Programování mikropočítače

Teoreticky můžeme k programování mikropočítačů použít jakýkoliv programovací jazyk. Jediné, co skutečně potřebujeme, je správný kompilátor a textový editor pro zápis zdrojového kódu. Kompilátor převádí zápis v programovacím jazyce do strojového kódu mikropočítače. Strojový kód se skládá z elementárních instrukcí, které jsou reprezentovány binárními hodnotami a odpovídají instrukční sadě procesoru mikropočítače. Samotný zkompilovaný program nám není nijak užitečný, dokud ho nenahrajeme do paměti mikropočítače. K tomu slouží zařízení nazývané „programátor.“ Programátor uloží program do FLASH paměti, popřípadě do nějakého druhu ROM (Read Only Memory) paměti určené k uchovávání programů.

1.2.1 CodeWarrior

CodeWarrior je integrované vývojové prostředí (IDE) od společnosti Freescale Semiconductor. Vývojové prostředí je software, jehož součástí je sada nástrojů usnadňující programování počítačů. CodeWarrior obsahuje především editor pro psaní zdrojového kódu, kompilátor, debugger včetně simulátoru a rozhraní pro uložení programu do mikropočítače. K psaní programů můžeme využít jazyky Assemler, C nebo C++. Simulátor je velmi užitečný nástroj, umožňující ladit program, aniž bychom ho museli předem nahrát do mikropočítače.

1.2.2 Programování v jazyku symbolických adres (Assembleru)

Jazyk symbolických adres je též znám jako Assembler. Ve skutečnosti pojem Assembler označuje pouze překladač a samotný jazyk se nazývá jazykem symbolických adres. V praxi se však často používá termín Assembler v obou významech. Pro zjednodušení budeme pojem Assembler používat ve smyslu programovacího jazyka. Assembler je nízkoúrovňový programovací jazyk, který je založen na symbolické reprezentaci strojových instrukcí procesoru. Usnadňuje tedy programování tím, že umožňuje přiřadit adresám instrukcí a jejich parametrům zástupná jména. Protože je Assembler přímo spojen s konkrétním procesorem a jeho instrukční sadou, jsou programy v něm napsané jen velmi obtížně přenositelné na jiné platformy.

Zápis zdrojového kódu se provádí po řádcích. Na jednom řádku může být jen jeden příkaz. Obecná struktura programového řádku je následující:

návěští instrukce operand komentář

Zdrojový kód se zapisuje do souboru s příponou „asm.“ Vytváříme-li program ve vývojovém prostředí CodeWarrior, musí být tento soubor, stejně jako všechny připojené soubory, součástí projektu.

Direktivy se podobají instrukcím, nejsou však zpracovávány procesorem, ale překladačem. Příkladem je direktiva INCLUDE pro vkládání hlavičkových souborů s příponou „inc.“ Pokud do nějaké části kódu vložíme pomocí direktivy INCLUDE hlavičkový soubor, překladač na toto místo vloží celý obsah tohoto souboru.

2 OPERAČNÍ SYSTÉMY

Operační systém, přestože je tento pojem velmi často používán, je velmi těžké nějakým způsobem definovat. Operačních systémů existuje nepřeberné množství, ovšem všechny mají něco společného, a to snahu řešit několik základních problémů, kvůli kterým vlastně vznikají. Často se můžeme setkat s definicí operačního systému jako základního programového vybavení počítače, software, který řídí hardware a podobně. Jakkoliv zní tyto definice přesvědčivě, dají se aplikovat jen na některé systémy nebo jejich části. Vyhneme se definici operačního systému a raději si popíšeme jeho vlastnosti a cíle. Operační systém pracuje na nejnižší hardwarové úrovni a pomocí aplikačního rozhraní (API) vytváří abstraktní vrstvu, prostřednictvím které s počítačem pracují ostatní aplikace. Pomocí tohoto rozhraní umožňuje uživateli spouštět programy a zajistit jim bezpečné vstupní a výstupní operace. Systém spravuje zdroje, které přiděluje, popřípadě odebírá, ostatním programům. Jedním z nejdůležitějších úkolů je umožnit bezpečný běh více programů na jednom počítači. Některé systémy jsou natolik obsáhlé, že je těžké určit hranici mezi systémem a aplikacemi, které jsou společně s ním dodávány.

Operační systém reálného času (RTOS – Real Time Operating System) má o něco vyšší nároky než běžné systémy. Mnohem více zde, kromě správného vykonání úlohy, záleží také na době jejího provedení. RTOS dále musí například minimalizovat prodlevu reakce na události a musí mít plánovač navržený tak, aby byl systém deterministický.

Operační systém by měl poskytovat co nejkomplexnější služby s co nejnižším dopadem na výkon. S tímto problémovým požadavkem se setkáváme velmi často a nejobvyklejším řešením je hledání kompromisů.

2.1 Architektura, jádro

Základní částí operačního systému je jádro (kernel), které obsahuje nejdůležitější součásti potřebné pro chod systému a je spuštěno po celou dobu jeho běhu. Hlavním úkolem kernelu je přidělování paměti a času procesoru uživatelským programům. Součástí kernelu jsou také další komponenty, jejichž rozsah je u každého systému značně rozdílný. Příkladem je grafické uživatelské rozhraní (GUI) u složitějších systémů, které je buď součástí jádra, nebo je řešeno jako nástavba a běží na vyšších vrstvách. Podle rozsahu a skladby jádra rozdělujeme architekturu operačních systémů na tyto základní typy.

- Monolitický operační systém – všechny součásti jsou obsaženy v kernelu, mají neomezený přístup k hardware a přímo spolu komunikují. Tato návrh je velmi efektivní. Nevýhodou je vzájemná závislost jednotlivých komponent a problémy vznikající s rostoucí velikostí jádra.
- Mikrokernel – jádro obsahuje jen nejnutnější součásti pro vykonání ostatních služeb. Cílem je zachovat co nejmenší rozměry jádra. Ostatní služby jsou realizovány v uživatelském prostoru. Nevýhodou je vyšší režie potřebná pro častější přepínání kontextu, díky čemuž může být mikrokernel pomalejší než monolitický operační systém.
- Hybridní jádro – ne vždy je výhodné držet se vymezení konkrétní architektury, a proto si hybridní návrh vybírá konkrétní vlastnosti z monolitického operačního systému i z mikrokernel architektury.

Toto jsou tři základní přístupy k návrhu operačního systému. Existují další specifické typy architektury, například vrstvený, distribuovaný nebo síťový operační systém.

2.2 Proces

Nejprve si přesněji definujme pojem „program.“ Program je postup operací, které popisují nějakou úlohu - algoritmus. Program zapisujeme pomocí programovacího jazyka a následně kompilátorem převádíme do strojového kódu. Programem bychom tedy mohli nazvat blok zdrojového kódu, přesněji instrukcí, které spolu vzájemně souvisejí. Program můžeme také rozdělit do několika logických celků, které plní samostatné úlohy a mohou být na sobě zcela nezávislé. Tyto celky nazýváme „podprogramy“. Pokud program přeložíme a zkopírujeme do paměti mikropočítače, mikroprocesor začne provádět instrukce programu v takovém pořadí, v jakém jsou uloženy v paměti. V tomto okamžiku můžeme tento program označit termínem „proces.“ Proces je tedy běžící program, jakási entita, které je přidělen procesor.

2.2.1 Plánování

Jestliže nemá systém žádný prostředek k přepínání mezi procesy za jejich běhu, mluvíme o nepreemptivním plánování. Aktivní proces blokuje procesor tak dlouho, dokud ho sám

neuvolní. Tento způsob plánování je velmi snadno implementovatelný, ovšem naprosto nevhodný pro realtime operační systémy.

Operační systémy s preemptivním plánováním umožňují přerušit běžící proces, a tím dovolují pružněji reagovat na události. Systém už nemusí čekat na uvolnění procesoru právě aktivním procesem, ale může ho odebrat a přidělit jinému procesu.

Podle úrovně rozdělujeme plánování procesů na:

- Krátkodobé – plánování v rámci přidělování procesu. Rozhoduje se, kterému procesu bude v následujícím okamžiku přidělen procesor.
- Střednědobé – provádí výběr procesu, který bude přesunut do virtuální paměti, a tím ušetří paměť a systémové zdroje. Tento druh plánování je možný jen na počítačových systémech umožňujících vytvoření virtuální paměti.
- Dlouhodobé – plánuje se spouštění úloh tak, aby co nejefektivněji využily výkon procesoru.

Návrh krátkodobého plánování je klíčový pro systémy reálného času. Algoritmus plánování má zásadní vliv na dobu provádění úloh, odezvu na události a také na rychlost samotného plánování. Algoritmy plánování jsou tvořeny na základě různých kritérií a přinášejí rozdílné výsledky. Různé strategie berou v potaz například prioritu procesů, dobu jejich zpracování, nebo čas potřebný k dokončení procesu.

V procesu se za běhu systému odehrává veškerá uživatelská činnost. Často je potřeba, aby na mikropočítači běželo ve stejném okamžiku více procesů. Bylo by výhodné, kdyby se mohly zpracovávat paralelně (multitasking), jenže mikropočítač může současně provádět jen tolik procesů, kolik má procesorů. Přestože není možné na jednom procesoru zajistit skutečný paralelismus, pomocí rychlého přepínání mezi procesy můžeme vytvořit takzvaný „pseudoparalelimus.“ Pokud budeme v pravidelných časových intervalech přepínat procesor mezi jednotlivými procesy, budou se navenek tvářit, jako by se vykonávaly paralelně. Programy nejsou (a ani by být neměly) na toto přepínání připravené. Operační systém proto musí skrýt skutečnost, že procesy neběží souvisle, a postarat se o veškerou logiku jejich přepínání.

2.2.2 Komunikace mezi procesy

Protože jsou procesy samostatně pracující jednotky, musí komunikaci mezi nimi zajistit operační systém. Využívají se dva základní způsoby komunikace, a to zasíláním zpráv a prostřednictvím sdílené paměti.

Do sdílené paměti mají přístup všechny procesy. Úkolem operačního systému je do této paměti zajistit výlučný přístup.

Zasílání zpráv nabízí pružnější a efektivnější způsob komunikace, jeho implementace je ovšem oproti sdílené paměti podstatně složitější. Samotná zpráva může být realizována mnoha způsoby. Výhodou je, že tento typ komunikace je možné provozovat i po síti.

2.3 Multitasking

Multitaskingem označujeme schopnost operačního systému zpracovávat několik úloh současně. Jestliže není počítač schopen zajistit skutečný multitasking, to znamená, že nemá hardwarové prostředky k paralelnímu zpracovávání velkého množství úloh, vytváříme zdánlivý multitasking, který je řešen programově. Protože skutečný multitasking v celém rozsahu je zatím nerealizovatelný, mluvíme o zdánlivém multitaskingu jednoduše jako o multitaskingu.

K realizaci multitaskingu lze přistupovat mnoha způsoby. Obvykle se vychází ze schopnosti nebo neschopnosti operačního systému provádět preemptivní plánování procesů. U preemptivního multitaskingu se starost o realizaci multitaskingu přenechává operačnímu systému. Každému procesu je přiděleno časové kvantum, po jeho vypršení je proces systémem přerušen a na základě určitých kritérií je vybrán další proces. Celý cyklus se neustále opakuje. Pokud systém nepodporuje preemptivní plánování, přenechává se veškeré řízení procesům a vzniká tak jednoúlohový operační sytém. V tomto případě se jedná o kooperativní multitasking.

2.4 Vlákna

Vlákno (thread) je obdobou procesu, vzniká však v rámci konkrétního procesu. V operačních systémech nepodporujících vytváření více vláken běží každý proces právě v jednom vlákně. Všechna vlákna v rámci jednoho procesu sdílejí společný paměťový prostor. Vlákna nám umožňují paralelně zpracovávat složité a časově náročné úlohy.

Procesor se přiděluje vláknům. Vytvoříme-li v procesu dvě vlákna, rozdělí se proces na dvě části, které se budou vykonávat paralelně, respektive pseudoparalelně. Podle přístupu ke správě vláken rozlišujeme dva druhy vláken.

- Vlákna na uživatelské úrovni (ULT) – správa vláken je přímo součástí procesu a operační systém o těchto vláknech neví. Systém tedy proces zpracovává standardně jako jednovláknový.
- Vlákna na úrovni jádra (KLT) – o správu vláken se stará jádro operačního systému. Tento přístup umožňuje zpracovávat proces na více procesorech. Nevýhodou je vyšší režie operačního systému.
- Kombinace ULT a KLT – přináší kombinaci výhod obou přístupů. Je možné vybrat, jaká vlákna budou na uživatelské a jaká na systémové úrovni.

2.5 Uváznutí (deadlock)

Pokud operační systém zajišťuje výlučný přístup a postupné přidělování prostředků, může dojít k uváznutí procesů. Uváznutí je nežádoucí stav, kdy dva nebo více procesů vzájemně čekají na nějakou událost. Uváznutí nastane například v situaci, kdy jeden proces pracuje se sdílenou pamětí, druhému byl přidělen port a první proces čeká na uvolnění portu, zatímco druhý na uvolnění sdílené paměti. Během uváznutí dojde k zablokování obou procesů. Detekce uváznutí není jednoduchá a jeho předvídání prakticky nemožné. Systém může uváznutí ignorovat a spoléhat se na zásah uživatele (uživatelského procesu). Předcházení uváznutí spočívá v nesplnění alespoň jedné z následujících podmínek nutných pro vznik uváznutí.

- Výlučný přístup – se stejným systémovým prostředkem může pracovat současně jen jeden proces
- Proces může žádat o přidělení prostředků i v momentě, kdy mu už nějaké prostředky přiděleny byly
- Proces musí uvolnit přidělený prostředek, systém k tomu nemá oprávnění
- Cyklické čekání procesu na uvolnění prostředků

II. PRAKTICKÁ ČÁST

3 POPIS SYSTÉMU

Vytvořený operační systém je naprogramovaný v jazyku symbolických adres (assembleru). K programování bylo použito vývojové prostředí CodeWarrior od společnosti Freescale Semiconductor. Součástí projektu je pět souborů:

- kernel.asm
- kernel.inc
- params.inc
- ckernel.asm
- kernel.h

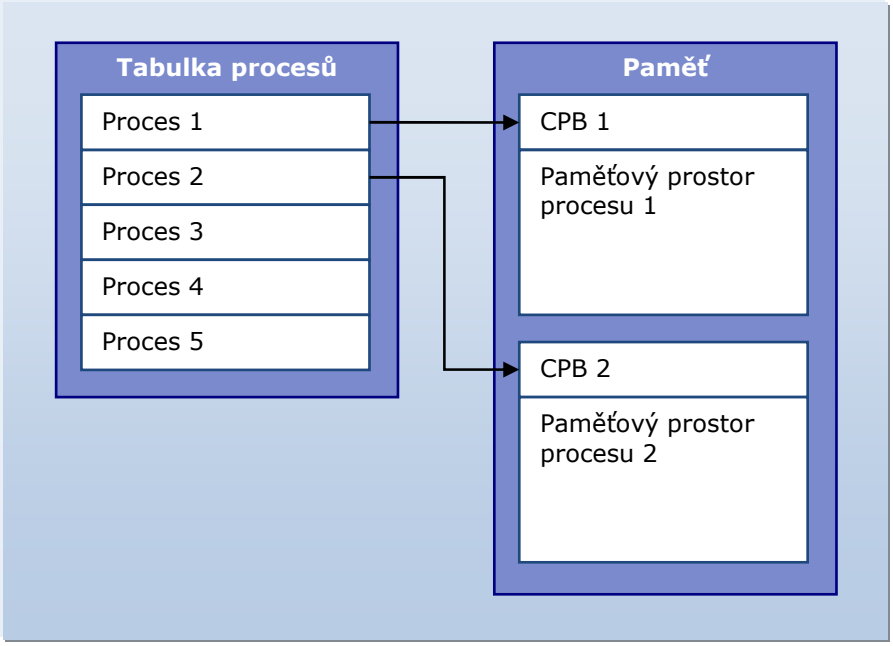
Kernel.asm je hlavní soubor projektu a jsou v něm umístěny definice všech služeb systému. Hlavičkový soubor kernel.inc zpřístupňuje služby a konstanty definované v kernel.asm a pomocí direktivy include se vkládá do .asm souboru, ve kterém chceme systémové služby používat. Součástí kernel.inc jsou i definice maker zjednodušujících volání služeb. Hlavičkový soubor params.inc je takzvaný „parametrický soubor," který slouží k nastavení různých částí operačního systému během překladu. Soubory ckernel.asm a kernel.inc se vkládají do projektu jazyka C nebo C++.

Nyní se podívejme na nejdůležitější části, ze kterých je systém složen.

3.1 Proces

Operační systém je založen na preemptivním plánování procesů a podporuje tedy preemptivní multitasking. U operačního systému s preemptivním plánováním může být proces přerušen v jakémkoliv místě svého běhu. Systém si musí uložit dostatečné množství informací o stavu procesu, aby mohl pokračovat v jeho vykonávání tak, jako by k žádnému přerušení nedošlo. Uložení všech potřebných informací se nazývá „uložení kontextu." Pod pojmem „kontext" se rozumí stav procesoru, registrů, případně jiných dalších zařízení. Aby mohl operační systém procesy spravovat, musí si udržovat informace o jejich stavu. K tomu slouží datová struktura „Process Control Blocks" (PCB), označovaná také jako „hlavička procesu." PCB v podstatě reprezentuje samotný proces a určuje veškeré jeho vlastnosti. PCB je umístěno v paměti RAM před blokem paměti, který je procesu přidělen. Protože se PCB nalézají v různých částech paměti, vede si systém tabulku, která obsahuje odkazy na jednotlivé PCB. Tato tabulka má konstantní velikost, proto může existovat jen

omezený počet procesů. Počet záznamů tabulky je odvozen od velikosti paměti RAM a minimální potřebné paměti, kterou každý proces obsadí. Velikost tabulky je možné explicitně změnit v parametrickém souboru, čímž ušetříme paměť, kterou by jinak zabírala tabulka, a taky snížíme maximální počet vytvořených procesů.

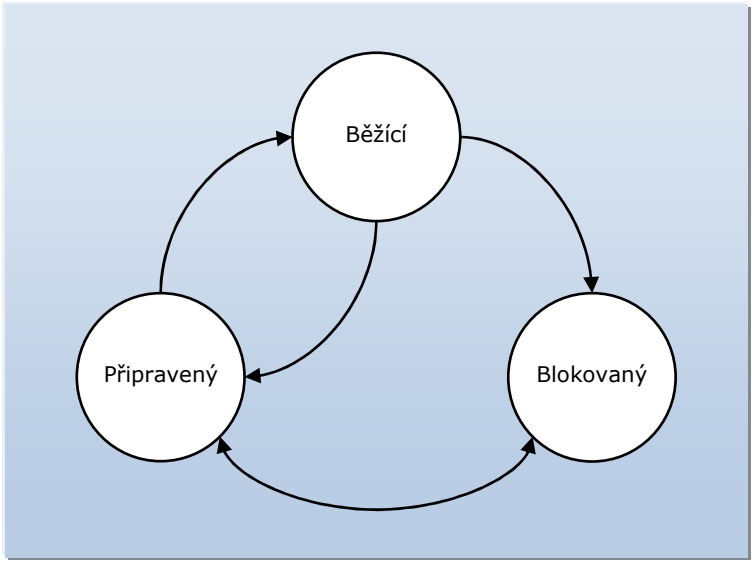


Obrázek 3 - Spojení tabulky procesů a PCB

PCB obsahuje, mimo jiné, následující hodnoty:

- ID procesu (1B)
- Začátek programu (2B)
- Adresa dna zásobníku (2B)
- Kopie registru SP (2B)
- Priorita (1B)
- Aktuální priorita (1B)
- Příznakový bajt (1B)
- Ukazatel na následující proces (2B)
- Číslo přidělené při vstupu do kritické sekce (1B)
- Počet zpráv čekajících ve frontě (1B)
- Ukazatele na vrchol a dno fronty zpráv (4B)

Při vytváření procesu nejdřív systém pomocí správce paměti najde dostatečně velkou oblast paměti, na jejím začátku vytvoří hlavičku procesu a odkaz na ni uloží do tabulky procesů. Takto vytvořenému procesu je přiděleno jedinečné identifikační číslo (ID), které odpovídá umístění záznamu v tabulce procesů. Tento způsob přidělování ID usnadňuje vyhledávání procesu v tabulce a zaručuje jedinečnost hodnoty. Implicitně je proces po svém vytvoření ve stavu „připravený." Během své existence proces prochází několika fázemi, které dohromady tvoří „životní cyklus procesu."



Obrázek 4 - Životní cyklus procesu

Proces, kterému je přidělen procesor, je ve stavu běžící. Jakmile uplyne doba, kterou může v tomto stavu strávit (uplynutí časového kvanta), dojde ke změně kontextu a přesune se do stavu „připravený," ve kterém čeká na přidělení procesoru. O tom, jakému procesu bude přidělen procesor, rozhoduje plánovač. Cílem procesů je strávit co nejvíce času ve stavu „běžící." Může nastat situace, kdy proces čeká na dokončení nějaké operace, a proto, aby zbytečně netrávil čas na procesoru, se uvede do stavu „blokovaný." Blokovaný proces nadále existuje, zabírá místo v paměti, jen je vyjmut z fronty připravených procesů. Proces může být do stavu „blokovaný" přiveden i jiným procesem, musí k tomu mít ale jeho povolení. Existují ještě další dva stavy – odložený blokovaný a odložený připravený, které se objevují v systémech podporujících střednědobé plánování. V obou stavech je proces přesunut do virtuální paměti, aby uvolnil místo v hlavní operační paměti jinému procesu.

Protože mikropočítač žádnou virtuální paměť nemá, nejsou tyto stavy využity.
Proces nemůže ke svému ukončení použít běžné instrukce pro návrat z podprogramu. Pokud bychom se například pokusili proces ukončit instrukcí JSR, procesor by tuto instrukci zpracoval standardním způsobem, to znamená, že by provedl skok na adresu uloženou na vrcholu zásobníku. Ale jaké místo v paměti tato adresa reprezentuje? To ve skutečnosti nemůžeme s určitostí říct. Může to být jakákoliv hodnota, kterou proces nebo systém uložil na zásobník. Může to být adresa dalšího procesu nebo dokonce adresa přesahující adresovatelný prostor. Taková nepředvídatelná událost by měla pro systém kritické následky. Při ukončení procesu se systém musí postarat o několik věcí. Proces například zabírá operační paměť, kterou je potřeba uvolnit, taky je nutné odstranit záznam z tabulky a vyjmout proces z fronty. Systém proto poskytuje nástroje pro bezpečné ukončení procesu.

K ukončení procesu jsou definovány dvě služby. První služba ukončí proces tak, že proces přestane existovat a nadále nezabírá paměť. K jeho opětovnému spuštění musí být znovu vytvořen. Rušení a vytváření procesu je ale poměrně časově náročná operace. Mohli bychom například potřebovat v určitých intervalech spustit krátký proces, který se rychle vykoná a skončí. Neustálé vytváření a rušení procesu by bylo neefektivní, proto je definována druhá verze služby pro ukončení, která proces pouze odloží a restartuje do stavu, ve kterém byl při svém vytvoření. Odloženému procesu je stále přidělena paměť a může být spuštěn pouze jiným procesem.

Typickou vlastností procesoru je, že musí neustále vykonávat instrukce. Abychom mohli splnit tento požadavek i v případě, že není spuštěn žádný proces, má systém k dispozici takzvaný „základní proces.“ Systém do základního procesu vstoupí pouze v případě, že není spuštěn žádný proces. Základní proces (BASIC_PROC) je systémový a jeho úkolem je vytížit procesor v době, kdy žádný proces nečeká ve frontě na své vykonání. V případě základního procesu se nejedná o klasický proces v pravém slova smyslu. Nemá žádnou hlavičku a není mu přidělena žádná paměť. Základní proces je prakticky jednoduchá nekonečná smyčka a není možné s ním provádět žádné standardní operace jako s ostatními procesy. V případě, že je vyvoláno přerušení časovače, obsluha přerušení zkontroluje obsazenost fronty procesů. Pokud je tato fronta prázdná, obsluha přeskočí služby starající se o přehození kontextu a přeplánování fronty, a provede skok přímo do základního procesu.

3.2 Dynamická správa paměti

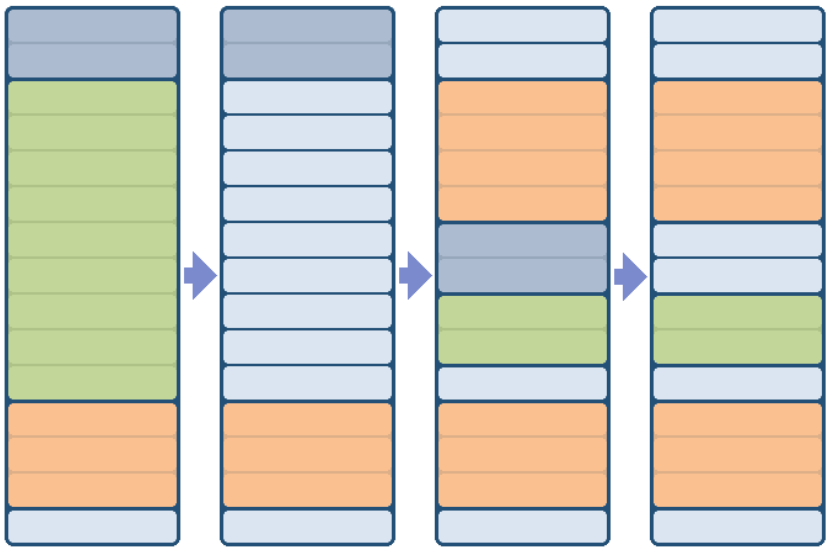
Paměťový prostor využívaný procesy je rozdělen na tři části – blok pro statické proměnné, sdílenou paměť a paměťový prostor pro procesy. První část je společná pro všechny procesy a je vyhrazena pro programové proměnné, jejichž umístění je pevně dáno při překladu programu. Začátek této oblasti je nastaven na začátek paměti RAM, tedy na adresu 80, tuto hodnotu můžeme ale změnit. Proměnné jsou v paměti řazeny za sebou v pořadí, v jakém jsou v programu deklarovány (například pomocí instrukce DS.B). Deklarujeme-li například proměnnou „clock," překladač jí přiřadí adresu poslední volné paměťové buňky, v našem případě hodnotu $80. Od tohoto okamžiku výraz „clock" představuje zástupné jméno hexadecimálního čísla $80. Nyní, kdykoliv v programu použijeme výraz „clock," překladač ho nahradí touto hodnotou. Protože pracujeme s přímými adresami, jsou operace s proměnnými poměrně rychlé. Nevýhodou je, že zabírají místo v paměti po celou dobu běhu programu. Pokud potřebujeme krátkodobě uložit nějakou hodnotu, třeba jako parametr pro podprogram, je výhodnější ji umístit do zásobníkové paměti. Na zásobník se ukládají také návratové adresy při volání podprogramu a stav procesoru při vyvolání přerušení. Je tedy zřejmé, že každý proces musí mít vlastní paměťový prostor vyhrazený pro zásobník. Jestliže bude v další části textu zmíněno „přidělování paměti," bude to myšleno ve smyslu přidělování zásobníkové paměti.

Paměť se procesům přiděluje dynamicky při jejich vytváření. K tomu slouží správce paměti. Při požadavku na vytvoření procesu správce paměti najde dostatečně velký blok paměti a služba pro vytvoření procesu na začátku tohoto bloku vytvoří hlavičku procesu. Jestliže má správce paměti vyhledat blok neobsazené paměti, bylo by nerozumné procházet celou paměť a testovat jednotlivé bajty. Systém místo toho rozdělí paměť na malé části o velikosti několika bajtů a pomocí bitové tabulky si udržuje informace o jejich obsazenosti. Každý bit v této tabulce reprezentuje 8 bajtů paměti. Pokud budeme uvažovat 2048 bajtů adresovatelné paměti, bude mít tabulka velikost 32 bajtů.

Správce paměti nejprve sekvenčně prochází tabulku, testuje jednotlivé bity a hledá potřebný neobsazený blok. Jestliže ho najde, uloží adresu jeho začátku a aktualizuje záznam v tabulce. Pro aktualizaci tabulky musí opět sekvenčně procházet bajty a pomocí masky nastavovat jednotlivé bity. Samotný správce paměti je poměrně složitý podprogram.

Díky nutnosti dvakrát procházet tabulku a provádět bitové operace s jednotlivými bajty, přičemž si neustále udržovat dostatečné množství informací, je vytvoření procesu časově docela náročná operace. Nabízely se i jednodušší způsoby správy paměti, neposkytovaly ale takovou pružnost a univerzálnost. Současný správce paměti není vázaný jen na procesy a může být použit prakticky v jakémkoliv jiném kontextu. Pokud je proces zrušen, musí správce paměti aktualizovat záznam v tabulce, aby mohla být paměť přidělena jinému procesu.

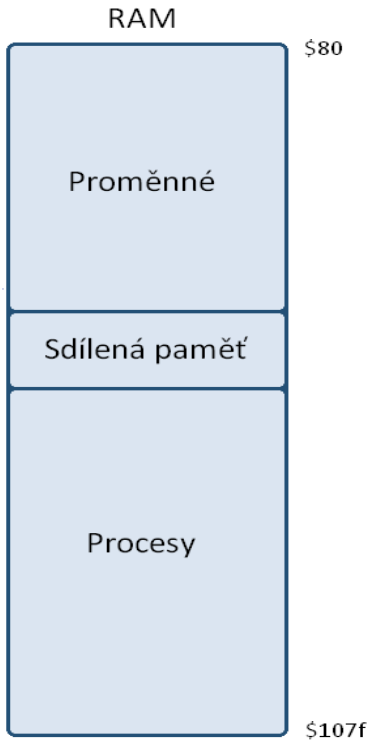
Dynamické přidělování paměti přináší kromě nesporných výhod i jisté nevýhody. O složitosti algoritmu už byla řeč. Dynamická správa paměti přináší další nepříjemný vedlejší efekt a tím je „fragmentace.“ Fragmentace je natolik velký problém, že může způsobit vážný nedostatek paměti, přestože je procesy fyzicky obsazena pouze její malá část. Fragmentace vzniká při opakovaném obsazování a uvolňování paměti o různé velikosti. Pro příklad uvažujme část souvislé volné paměti o velikosti 64 bajtů. Pokud obsadíme 56 bajtů z tohoto bloku, zůstane 8 bajtů, které jsou prakticky nevyužitelné. Při vytváření a rušení malých procesů tak mohou vzniknout jakési „díry,“ které svou velikostí neodpovídají požadavku na vytvoření nového procesu. Princip fragmentace je zřetelný z obrázku. Světle modré oblasti představují volnou paměť a vybarvené paměť obsazenou jednotlivými procesy.



Obrázek 5 - Fragmentace paměti

Systém neobsahuje žádný nástroj k odstraňování fragmentace. To by vyžadovalo složitou správu paměti s využitím logického adresového prostoru, popřípadě neefektivní časově náročné operace. Fragmentaci je však možné předcházet. Obecně by se měly procesy rušit jen v nejnutnějších případech, například k nutnému uvolnění paměti. Zrušení procesu lze nahradit jeho odložením. Dále platí, že pokud vytváříme procesy, které alokují stejně velké množství paměti, k fragmentaci nedochází.

Rozdělení paměti RAM na tři části je znázorněno na obrázku.



Obrázek 6 - Rozdělení paměti RAM

3.3 Plánovač

Systém je založený na preemptivním plánování, a proto může být běžící proces přerušen prakticky kdykoliv během svého vykonávání. Úkolem plánovače je v takovém okamžiku rozhodnout, kterému dalšímu procesu bude přidělen procesor. Toto rozhodování se uskutečňuje na základě priority. Priorita je jednobajtová proměnná přidělená každému

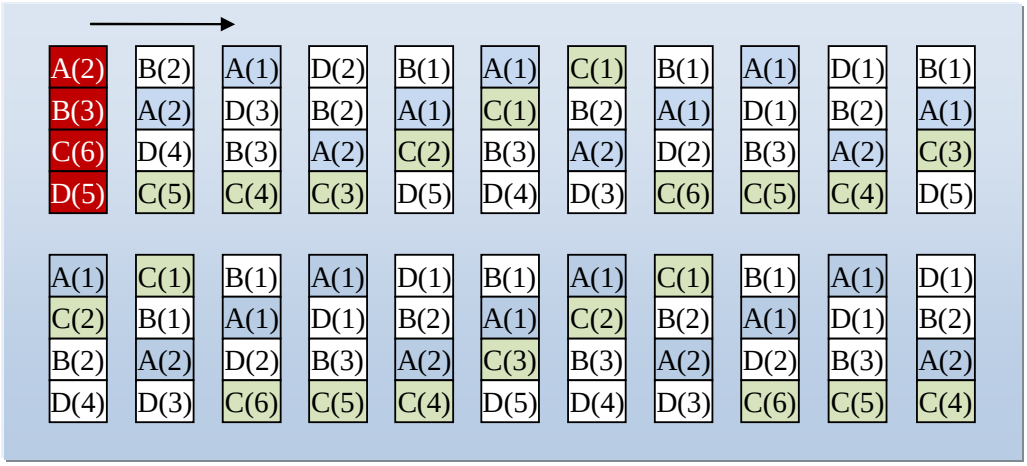
procesu a může nabývat hodnoty 1 až 255. Nižší hodnoty reprezentují vyšší prioritu. Platí, že proces s vyšší prioritou dostává přednost před procesem s nižší prioritou.

Připravené procesy jsou řazeny do fronty, kde čekají na přidělení procesoru. Vykonává se vždy ten proces, který je ve frontě umístěn jako první. Fronta je ve skutečnosti jednosměrný lineární seznam a je realizovaná pomocí ukazatelů na následující proces, které jsou umístěny v PCB. Implementace fronty umožňuje realizaci plánovače.

Pokud je vykonávající se proces přerušen v důsledku vyvolání hlavního systémového přerušení časovače, musí se plánovač postarat o přeplánování fronty. Přerušený proces je ze začátku fronty vyjmut a zařazen na místo před první proces s nižší prioritou. Protože se po tomto přeplánování může proces nacházet opět na začátku fronty a v takových situacích by pracoval plánovač neefektivně, proces přeplánování se v takovémto případě přeskočí. Z principu činnosti plánovače vyplývá, že pokud mají všechny procesy stejnou prioritu, budou vždy zařazeny na konec fronty.

Prioritní plánování přináší problém s takzvaným „stárnutím procesu.“ Uvažujme dva procesy, první (P1) s prioritou 10 a druhý (P2) s prioritou 15. Zamyslíme-li se nad principem plánování, kdy je proces zařazen vždy před první proces s nižší prioritou, zjistíme, že proces P2 se dostane na procesor až po ukončení nebo odložení procesu P1 (nižší hodnota znamená vyšší prioritu). Pokud P1 poběží v nekonečné smyčce, procesu P2 nebude procesor přidělen nikdy. V takových případech říkáme, že dochází ke stárnutí procesu. V předchozích příkladech je problém jasně viditelný a mohli bychom se postarat o jeho odstranění, i když za cenu jistých ústupků. Snadno však vzniknou situace, kdy je odhalení tohoto nežádoucího jevu poměrně obtížné. Procesy mohou například vytvářet jiné procesy jako reakci na nějakou událost a nepředvídatelně tak zablokovat jiný proces.

Aby se zabránilo stárnutí procesu, systém v každém svém cyklu zvyšuje prioritu všem procesům čekajícím ve frontě (jen po nastavitelnou mez). To sice zabere nějaký čas, je ale zaručeno, že každému procesu bude po nějaké době přidělen procesor. Postup při vyvolání přerušení časovače je zjednodušeně následující. Proces, který je ve frontě jako první, se z této fronty vyjme, resetuje se jeho priorita na původní hodnotu, následně se zvýší priorita všem ostatním procesům a poté je proces zařazen před první proces s nižší prioritou. Ve výsledku bude procesu s vyšší prioritou přidělen procesor častěji než procesu s nižší prioritou.



Obrázek 7 - Příklad činnosti plánovače

Tento druh plánování je jistě v mnoha případech výhodný, ale je pravděpodobné, že se díky němu systém stane nedeterministický. Determinismus je ovšem důležitou podmínkou operačního systému reálného času. Funkce plánovače postupně zvyšující prioritu procesů je proto aktivní pouze v případě, že je v parametrickém souboru params.inc definováno DEFINE_RPRIORITIES. V opačném případě zůstává priorita během plánování nezměněna.

3.4 Systémový čas

Velmi často, a to při programování počítačů obecně, potřebujeme pracovat se „skutečným časem.“ Můžeme například chtít každou sekundu získat hodnoty z A/D převodníku, zjistit dobu vykonávání nějaké úlohy nebo jen chceme, aby v pravidelných intervalech blikala LED dioda indikující činnost zařízení. K určení času můžeme využít skutečnost, že i samotný mikroprocesor zpracovává instrukce určitou rychlostí, která je závislá na jeho pracovní frekvenci a vytvořit čekací smyčky. Čekací smyčka je jednoduchý cyklus, který se vykoná mnohokrát po sobě jen proto, aby strávil potřebný čas na procesoru. Pokud víme, kolik cyklů procesoru spotřebují instrukce, z nichž je čekací smyčka tvořena, a známe i pracovní frekvenci procesoru, můžeme celkem přesně vypočítat, kolik času program ve smyčce stráví. Toto řešení není příliš šťastné a takto napsané programy jsou velmi neefektivní a rovněž velmi obtížně přenositelné mezi různými mikropočítači. Při použití čekacích smyček plýtváme výkonem mikropočítače, který by mezitím mohl vykonávat jiný program, a v případě změny pracovní frekvence hardwaru musíme provést změny ve

zdrojovém kódu. Využitelnost v operačních systémech s preemptivním plánováním, kde je čekací smyčka součástí procesu, je výrazně nižší. Použití časovače a jím generovaného přerušení je rozhodně lepší volba. Ovšem časovač je využívaný operačním systémem a zpracování jeho přerušení je základním stavebním kamenem celého systému. Aby byl zajištěn bezpečný a stabilní chod systému, neměly by procesy obsluhovat žádné přerušení, včetně přerušení generovaného časovačem.

Aby mohly procesy pracovat s „reálným časem," poskytuje jim operační systém prostředek ke zjištění „aktuálního času" prostřednictvím systémových hodin. Systémové hodiny jsou dvoubajtová proměnná, jejíž hodnota je při každém přerušení časovače zvýšena o jedničku. Systémové hodiny reprezentují „systémový čas." Ke zjištění přesného „reálného" času nám stačí znát systémový čas, a to, s jakou periodou je generováno přerušení časovače. Systémové hodiny jsou omezeny na dva bajty, což je celkem 65536 cyklů. Po překročení této hodnoty jsou nastaveny na nulu a celý průběh se opakuje.

Podívejme se nyní na využití systémového času ke spouštění procesů v časových intervalech. Nabízí se dva přístupy. Buď můžeme přenechat zodpovědnost na systému nebo na uživateli, respektive na procesech samotných. Pokud by měl tento úkol na starosti operační systém, získali bychom velmi efektivní a jednoduchý nástroj. Ovšem operační systém by měl spotřebovávat co nejméně výkonu procesoru a rozhodování o tom, který proces a kdy má být spuštěn, zabere čas strávený v obsluze přerušení časovače. Navíc samotné spouštění může být kromě uplynutí potřebného času podmíněno i jiným způsobem. Systém proto ponechává implementaci veškeré logiky na uživateli. K tomu ovšem musí poskytovat nástroje, které to umožní. Jedním z nich jsou systémové hodiny a druhým služba WAIT_CYCLE.

Služba WAIT_CYCLE pozdrží proces zadaný počet jednotek systémového času. Čekání zbytečně neblokuje procesor, ale uvolní ho jiným procesům. Protože je mikropočítač osmibitový a systémový čas je uložen jako dvoubajtová hodnota, je z důvodu snazší implementace vstupní parametr pouze osmibitové číslo. To omezuje službu jen na čekání po dobu 255 cyklů. Tento nedostatek je možné odstranit několikanásobným čekáním. Vstupní parametr služby se ukládá do registru H:X, je ale využit jen nižší bajt. To přináší možnost rozšíření služby na šestnáctibitový vstupní parametr při zachování zpětné kompatibility.

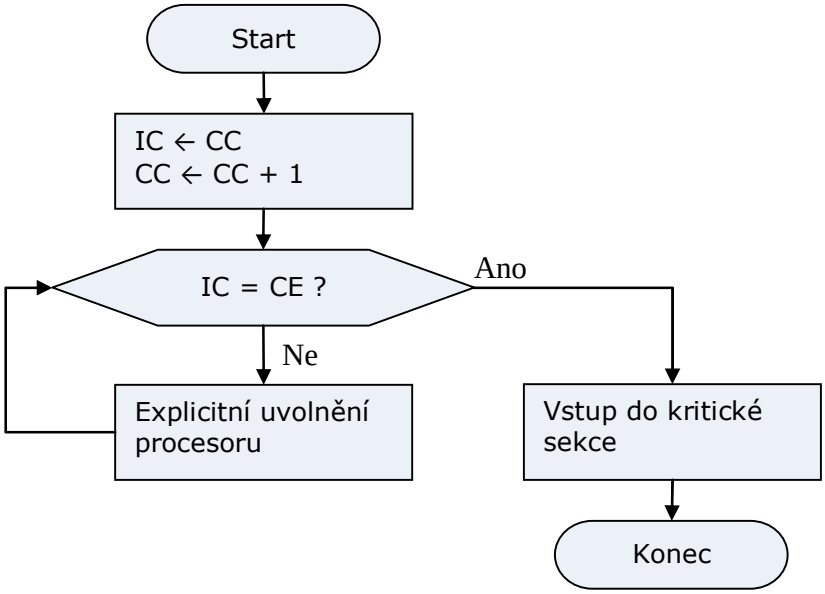
3.5 Komunikace mezi procesy

Komunikace mezi procesy je další důležitou službou, kterou poskytuje operační systém. Existují dva způsoby komunikace, a to prostřednictvím sdílené paměti nebo zasíláním zpráv. Oba způsoby mají své výhody a nevýhody a přinášejí různé výsledky. V systému jsou proto implementovány obě možnosti.

3.5.1 Sdílená paměť

Protože do sdílené paměti mají formálně přístup všechny procesy, může být využita pro jejich vzájemnou komunikaci. Nevýhodou tohoto řešení je, že neexistuje záruka, že si procesy nebudou data vzájemně modifikovat. Jako sdílená paměť může být použita v podstatě jakákoliv část paměti. Úkolem operačního systému je do této paměti zajistit především bezpečný přístup. Zde se opět projevují problémy generované preemptivním plánováním. Do sdílené paměti nesmí mít přístup zároveň proces, který data zapisuje a proces, který se snaží data číst. Zapisující proces by mohl totiž přepsat část dat ještě před dokončením jejich čtení a ty by se staly nekonzistentní. Systém proto nesmí umožnit pracovat se sdílenou pamětí více procesům najednou, jinak řečeno, musí zajistit výlučný přístup.

Sdílená paměť se nazývá kritická oblast, nejmenší část kódu pracující s kritickou oblastí se označuje pojmem kritická sekce. Aby byl zajištěn výlučný přístup do kritické oblasti, musí být splněna podmínka, že do kritické sekce může vstoupit pouze jeden proces a ostatní procesy musí čekat na její uvolnění. Jakmile proces vstoupí do kritické sekce, systém do ní znemožní přístup ostatním procesům. Při jejím opuštění je přístup opět povolen. Nyní je otázka, jakému dalšímu procesu bude umožněno do kritické sekce vstoupit. Mohli bychom to ponechat náhodě a do kritické sekce nechat vstoupit proces, který se dostane jako první na procesor. Systém nabízí lepší řešení. Algoritmus pro přidělování přístupu zaručuje, že do kritické sekce budou mít procesy přístup v takovém pořadí, v jakém o něj systém zažádaly. Toho je dosaženo přidělováním osmibitové hodnoty každému procesu (IC), který se pokouší do kritické sekce vstoupit a udržováním si informace o tom, které hodnotě je do kritické sekce vstup povolen (CE). Princip algoritmu je znázorněn na vývojovém diagramu.



Obrázek 8 - Algoritmus vstupu do kritické sekce

Jediná společná kritická oblast by byla pro uživatele poměrně omezující. Mohli bychom chtít například synchronizovat dva páry procesů, které pracují s vlastními společnými daty. V tomto případě by procesy na sebe musely zbytečně čekat a práce se sdílenou pamětí by byla neefektivní. Systém proto podporuje několikanásobné kritické oblasti. Jejich počet a velikost lze změnit v parametrickém souboru params.inc. Každá z těchto kritických oblastí je identifikována číslem (0 až 255) a přístup do nich je vzájemně zcela nezávislý. Stále ovšem platí, že s každou kritickou oblastí může pracovat jen jeden proces a zároveň každý proces může být jen v jedné kritické oblasti.

K získání přístupu do kritické oblasti jsou určeny služby GET_CRITICAL_SECTION a WAIT_FOR_CRITICAL_SECTION. Obě služby přebírají jako parametr číslo kritické oblasti a při úspěšném vstupu vrátí adresu jejího začátku. Rozdíl mezi službami je v tom, že WAIT_FOR_CRITICAL_SECTION čeká, dokud není přístup povolen, zatímco GET_CRITICAL_SECTION vrací při nemožnosti vstupu do kritické oblasti chybové hlášení. Jakmile proces opustí kritickou sekci, musí ji uvolnit dalšímu procesu pomocí služby RELEASE_CRITICAL_SECTION. Služby pro získání přístupu a uvolnění musí být volány vždy v páru. Procesy by měly v kritické sekci setrvávat nejkratší možnou dobu.

3.5.2 Zasílání zpráv

Sdílená paměť nabízí užitečný způsob komunikace, ale ne vždy se jeví jako ideální volba. Alternativou ke sdílené paměti je zasílání zpráv. Zpráva je zasílána vždy jen jednomu procesu (výjimkou jsou systémové zprávy) a pouze tento má k ní přístup. Ovšem tento způsob komunikace není omezen jen na vzájemné zasílání zpráv mezi procesy, ale využívá ho i operační systém (i když v poněkud jiné podobě). Proces, který zprávu zasílá, se nazývá „odesilatel“ a proces přijímající zprávu se nazývá „adresát.“ Odesílatel může zaslat zprávu libovolnému procesu, ale neexistuje žádný způsob, jak donutit adresáta na tuto zprávu reagovat. Vyzvednutí zprávy je zcela dobrovolné a vyžaduje jisté změny v návrhu programu. Procesy je možné synchronizovat tak, že odesilatel čeká na potvrzení přijetí zprávy od příjemce, samozřejmě opět formou zaslané zprávy.

Protože odesilatelé nemají záruku, že příjemce bude zpracovávat přijaté zprávy okamžitě po jejich doručení, jsou odeslané zprávy ukládány do FIFO zásobníku, odkud je adresát může vyzvednout. Zásobník má ovšem omezenou kapacitu, a pokud je zcela zaplněn, další zasílání není možné až do doby, dokud se příjemce nepostará o vyzvednutí zpráv a tím i o uvolnění místa v zásobníku. Zásobníku můžeme říkat „fronta zpráv.“ Zprávy ve frontě zůstávají do té doby, dokud nejsou vyzvednuty.

Zpráva je složena ze tří bajtů. Do nejvyššího bajtu je vždy vloženo ID odesilatele a do zbylých dvou je uložena samotná zpráva. Zpráva může představovat prakticky cokoliv. Můžeme klidně procesu zaslat zprávu, ve které bude uložena informace, kde ve sdílené paměti má proces hledat výsledek výpočtu, který provedl odesilatel zprávy. Fronta má implicitně místo pro čtyři zprávy. Jestliže zabírá každá zpráva tři bajty, musí mít každý proces pro frontu vyhrazeno dvanáct bajtů.

Systém nabízí ještě jednu variantu ukládání zpráv. Často nemusí být využit potenciál zásobníku, do kterého se zprávy ukládají, pokud je zasílání zpráv méně častá událost, popřípadě není tato služba systému využita vůbec. Jak už bylo řečeno, fronta zpráv ve standardním nastavení obsadí 12 bajtů paměti každého procesu. Navíc implementace FIFO fronty není, na rozdíl od LIFO zásobníku, zcela triviální. Nejenže vyzvedávání a zasílání zpráv trvá více času, ale i samotný zdrojový kód zabírá cenné místo ve FLASH paměti mikropočítače. Proto systém nabízí volbu mezi variantou fronty pro více zpráv a variantou, kdy mají procesy vyhrazený prostor pouze pro jednu zprávu. Druhá varianta vyžaduje

podstatně jednodušší implementaci a tedy i kratší a rychlejší zdrojový kód. Rozhodnutí, která varianta bude použita, se provádí v době překladu na základě systémového parametrického soubor „params.inc.“ Pokud je v tomto souboru definováno „DEFINE_MSG_QUEUE,“ bude v systému použita varianta s klasickou frontou.

Jak už bylo naznačeno na začátku, i operační systém používá zprávy jako prostředek ke komunikaci s procesy. Například pokud je generováno KBI přerušení, systém zpracuje obsluhu přerušením a procesy o tomto přerušení informuje právě zasláním zprávy. Bylo by ovšem neefektivní zasílat zprávu každému procesu zvlášť, nehledě na to, že by měl systém v obsluze přerušení trávit co nejméně času. Systém v těchto případech nepoužívá klasické zprávy, ale takzvané „systémové zprávy.“ Systém má vlastní frontu zpráv (systémová fronta zpráv), do které mají přístup všechny procesy, ale zprávy z ní můžou jen číst. Po přečtení zůstává zpráva ve frontě. Aby byly operace se systémovými zprávami co nejrychlejší, má fronta kapacitu jen pro dvě zprávy. Místo pro jednu zprávu by poskytovalo jistě výkonnější řešení, ovšem kapacita fronty se zdála docela omezující. Práce se dvěma zprávami může být taky poměrně rychlá, zprávy totiž v tomto případě stačí ukládat střídavě mezi dvěma místy. Systémová zpráva nepotřebuje nést informace o odesilateli, protože tím je vždycky operační systém, proto si vystačí pouze se dvěma bajty. Ve vyšším bajtu je uložena zpráva a v nižším její parametr. Parametr se odvíjí od typu zprávy. V případě KBI přerušení je to hodnota registru PTAD. Ke každé zprávě je ještě uložen stav systémových hodin v době odeslání zprávy.

3.6 Přidělování portů

Porty mají podobné vlastnosti jako sdílená paměť. Mají k nim přístup taktéž všechny procesy a vyžadují tedy výlučný přístup. Přidělování portů je řešeno jednodušeji než práce se sdílenou pamětí. Jestliže proces požádá o přidělení portu, systém mu poskytne pouze informaci, jestli byl procesu port přidělen nebo jestli je port obsazen jiným procesem. Po ukončení práce s portem musí být opět uvolněn, aby mohl být použit jiným procesem.

Systém si vede pro každý port bitový příznak, který indikuje, jestli je přidělen některému z procesů. Přidělování a uvolňování portů obnáší v podstatě pouze nastavení patřičného příznakového bitu. Aby mohl systém uvolnit správný port, má každý proces příznakový bajt, ve kterém si udržuje informace o tom, který port mu byl přidělen.

Stejně jako u sdílené paměti i zde platí, že uživatel může tento mechanismus ignorovat a pracovat s porty přímo. Systém nemá nástroje, jak tomu zabránit. Využívání systémových služeb však poskytuje mnohem větší bezpečnost. Protože se jedná o bezpečnostní rozšíření systému, jsou služby pro přidělování portů aktivní pouze v případě, že je v parametrickém souboru params.inc definováno DEFINE_EXT_PORT_ACCES. V opačném případě jsou služby kvůli zachování kompatibility stále přístupné, ale jsou zjednodušeny na pouhé vracení hodnoty RTN_OK.

3.7 Předcházení uváznutí

Aby se systém vyhnul možnému uváznutí procesů, nedovoluje procesu vstoupit do kritické sekce, popřípadě přidělit port, v případě, že už mu jeden z těchto prostředků vyžadující výlučný přístup byl přidělen. Tímto není splněna jedna z podmínek nutných k uváznutí procesů.

4 PROGRAMOVÁNÍ V OPERAČNÍM SYSTÉMU

4.1 Makra

Volání služeb systému je klasické volání podprogramu. Běžný postup je takový, že nejdřív do registrů, popřípadě na zásobník, uložíme parametry a pomocí instrukce JSR se provede skok do podprogramu. Všechny služby systému je možné volat tímto běžným způsobem. Systém nabízí ještě jedno, elegantnější a bezpečnější řešení. Pro každou službu je v souboru „kernel.inc“ definováno makro, které zjednoduší volání všech služeb na jednořádkový zápis, který svou strukturou připomíná volání funkcí ve vyšších programovacích jazycích. Výsledkem je přehlednější a čitelnější zdrojový kód s menší pravděpodobností vzniku chyby. Syntaxe při použití maker je obecně následující:

název_služby parametr1, parametr2, ... , parametr N

Za názvem služby je seznam parametrů oddělených čárkami. Některé služby přirozeně nepotřebují žádný parametr, proto stačí k jejich zavolání napsat samotný název. Samotné makro je při překladu programu nahrazeno kódem, který odpovídá standardnímu volání podprogramu. Zde je příklad makra definovaného pro službu „SEND_MESSAGE“

```
SEND_MESSAGE:     MACRO
      lda   \1
      ldhx  \2
      jsr   RTOS_SEND_MESSAGE
                  ENDM
```

Jednoduše řečeno, makra nepřinášejí žádný rozdíl ve funkčnosti ani výkonnosti programu, ale nabízejí snazší použití a přehlednější kód méně náchylný na vznik chyb. V dalším popisu jsou používána jen makra, klasický zápis lze snadno přečíst z definice maker.

4.2 Start a inicializace

Předtím, než budeme moct systém spustit, musíme provést jeho inicializaci. Inicializace nastaví všechny proměnné využívané systémem na počáteční hodnoty a nakonec inicializaci potvrdí nastavením interního příznakového bitu. Inicializace musí být vždy první služba, která je zavolána. Pokud bychom volali nějakou sužbu před inicializací

systému, mohli bychom očekávat zcela nepředvídatelné výsledky. Inicializace se provádí službou INIT, která má dva parametry. První parametr není v současné verzi využit, ale nabízí možnosti pro případné rozšíření inicializace při zachování kompatibility s již napsanými programy. Druhý parametr je adresa podprogramu, který se bude volat při každém generovaném KBI přerušení. Standardně systém během tohoto přerušení zasílá systémovou zprávu a zodpovědnost za její zpracování ponechává na procesech. Někdy je potřeba na toto přerušení reagovat okamžitě, ale jednoduché zaslání zprávy může způsobit značně velkou odezvu. Volání uživatelského podprogramu umožňuje na přerušení reagovat okamžitě. Podprogram by měl být co nejkratší, protože zvyšuje dobu zpracování přerušení, což je nežádoucí. Podprogram bude volán jen v případě, že hodnota druhého parametru služby INIT je nenulová a v parametrickém souboru je definováno DEFINE_KBI_PROC.

Po inicializaci, ještě před startem systému, je jedinou službou která může být volána, vytvoření procesu. Volání ostatních služeb není doporučeno (nikoliv však zakázáno, služby neobsahují kontrolu toho, jestli systém běží) a také by nemělo žádné rozumné opodstatnění.

Systém je spouštěn službou START nebo START_EX. START aktivuje časovač a začne provádět první proces čekající ve frontě. Pokud není spuštěn žádný proces, skočí do základního procesu. START_EX je rozšíření služby START. Jako parametr přebírá adresu podprogramu, který se vykoná ještě před samotným spuštěním systému. Do tohoto podprogramu můžeme umístit vytvoření procesů a inicializaci proměnných a lépe tak strukturovat program.

Ukončení činnosti systému se provádí službou STOPS. Program bude dále pokračovat za voláním služby pro spuštění. Před opětovným spuštěním je systém nutné opět inicializovat.

4.3 Vytvoření procesu

První příležitost k vytvoření procesu je bezprostředně po inicializaci. Za běhu systému může být proces vytvořen jen jiným procesem. Situace je komplikovanější, jestliže spustíme systém, přestože jsme žádný proces nevytvořili. V takovém případě systém skočí do základního procesu a jediná příležitost k vytvoření je při generování KBI, popřípadě jiného přerušení. Nejlepší příležitost pro vytvoření procesu je tedy ještě před zavoláním

služby START. Elegantní řešení nabízí služba START_EX, kdy vytvoření procesů umístíme do podprogramu, jehož adresu předáme jako parametr.

Procesy se vytvářejí pomocí služby CREATE_PROCESS, která požaduje čtyři parametry. První parametr je binární a určuje režim procesu. Pokud je jeho hodnota nulová, vytvoří se proces standardním způsobem, takže bude zařazen do fronty a bude mít veřejné práva (kapitola 4.5). Veřejná práva umožňují ostatním procesům provádět s tímto procesem operace, jako je pozastavení a zrušení. První parametr může mít nastaveny následující bity:

PROC_BLOCKED
PROC_RUNNING
PROC_PRIVATE

Neměli bychom nastavovat zároveň bity PROC_BLOCKED a PROC_RUNNING, přestože PROC_BLOCKED má větší váhu než PROC_RUNNING. Ostatní bity systém ignoruje.

Druhý parametr je priorita procesu. Třetí parametr určuje velikost paměti, kterou má systém procesu přidělit. Zde si je potřeba uvědomit několik skutečností. Tato hodnota nepředstavuje přímou velikost v bajtech, ale počet segmentů, se kterými pracuje správce paměti. Z konstanty SEG_SIZE můžeme zjistit, kolik bajtů obsahuje jeden segment. K námi zadané hodnotě je navíc přičtena paměť pro hlavičku procesu. Hlavička má velikost PROC_HEAD_SIZE_SEG segmentů.

V posledním parametru předáváme systému adresu začátku programu. Na tomto místě se začne proces po svém spuštění vykonávat. Následující příklad předvádí inicializaci a spuštění systému, včetně vytvoření procesu. Výpis není kompletní, obsahuje jen nejdůležitější části.

```
;*******************************************
; Soubor main.inc
; Priklad spusteni systemu a vytvoreni procesu
;*******************************************

;Include derivative-specific definitions

            INCLUDE 'derivative.inc'
            INCLUDE 'kernel.inc'

; code section
```

```
MyCode:     SECTION
      NOP
      INIT #0
      START_EX #startInit
mainLoop:
      feed_watchdog
      BRA    mainLoop

;*******************************************
;inicializacni podprogram
startInit:
      ;vytvoreni procesu
      CREATE_PROCESS #PROC_RUNNING,#1,#5,#testovaciProces
      rts

;*******************************************
;proces
testovaciProces:
      feed_watchdog
      bra   testovaciProces
```

V příkladu je vidět typické použití maker. Program nejdřív provede inicializaci a služba START_EX ještě před spuštěním systému zavolá krátký podprogram *startInit*, který vytvoří jeden proces. Nově vytvořený proces má prioritu 1, systém mu přidělí 5 segmentů paměti a kód procesu se nachází na adrese *#testovaciProces.* Následně se provede spuštění systému a proces se začne vykonávat. Nyní se řízení předává operačnímu systému. Na instrukci následující za voláním START_EX se dostaneme jen v případě ukončení systému službou STOPS. V příkladě by pak program vstoupil do nekonečné smyčky. V hlavním podprogramu *main* skutečně nepotřebujeme dělat více, než je v příkladě.

Uvedený příklad má ještě jednu zajímavou vlastnost. Vytvořme si stejným způsobem ještě jeden proces, který bude naprosto stejný jako *testovaciProces,* jen bude mít nižší prioritu (například 15). Pojmenujme ho třeba *testovaciProces2.* Jak již bylo popsáno v kapitole 3.3, můžeme vypnout funkci plánovače, která se stará o postupné zvyšování priority všem procesům čekajícím ve frontě. Jestliže tedy tuto možnost vypneme (v parametrickém souboru odstraníme řádek DEFINE_RPRIORITIES) a program přeložíme a spustíme, stane se to, že druhému procesu nebude nikdy přidělen procesor. To je způsobeno tím, že při vyvolání přerušení časovače je vykonávající se proces vyjmut z fronty a zařazen před

proces s nižší prioritou. Protože ale oba procesy běží v nekonečné smyčce, bude proces s nižší prioritou čekat na přidělení procesoru po celou dobu běhu systému.

Je tedy zřejmé, že nastavení plánovače má mnohem rozsáhlejší důsledky a vyžaduje změnu přístupu při programování uživatelských aplikací.

4.4 Ukončení a odložení procesu

Každý proces musí jednou skončit. K ukončení může dojít buď procesem samotným, jiným procesem nebo nějakou externí událostí (ukončení činnosti systému). Ke standardnímu ukončení procesu jsou určeny služby DESTROY_PROCESS a DESTROY_THIS_PROCESS. Druhá jmenovaná služba má definovaný ještě jeden název - RETURN. Můžeme používat obě verze, přinášejí naprosto stejné výsledky. Služba DESTROY_PROCESS ukončí libovolný proces, který má veřejná přístupová práva (kapitola 4.5). Parametr této služby je ID procesu. Pokud se toto ID shoduje s volajícím procesem, automaticky se použije služba DESTROY_THIS_PROCESS, která slouží k ukončení volajícího procesu. Ukončenému procesu jsou odebrány všechny systémové zdroje a proces nadále přestane existovat. Pokud chceme proces opět spustit, musíme ho vytvořit službou CREATE_PROCESS.

Časté rušení a vytváření procesů je velmi neefektivní. Systém nabízí alternativu ke službě RETRUN, službu RETURN_ABORT. RETURN_ABORT proces neukončí, pouze ho nastaví na počáteční hodnoty a odloží.

Služby pro odložení procesu se používají stejným způsobem jako služby pro ukončení. DELAY_PROCESS odloží proces, jehož ID je shodné s předaným parametrem a služba DELAY_THIS_PROCESS odloží volající proces. Odloženému procesu zůstávají všechny přidělené systémové zdroje, jen se neúčastní plánování a není mu přidělen procesor. Odložený proces může být spuštěn pouze jiným procesem.

4.5 Přístupová práva

Systém obsahuje několik služeb umožňujících pracovat běžícímu procesu s ostatními procesy. Příkladem takové služby je odložení procesu. Službu umožňující spravovat jiné procesy poznáme podle toho, že přebírá v jednom z parametrů ID procesu. Aby nemohl proces měnit stav jiného procesu bez jeho souhlasu, nabízí systém ochranu v podobě

veřejných a privátních (neveřejných) přístupových práv. Privátní přístupová práva se přidělují procesu při jeho vytváření nastavením bitu PROC_PRIVATE v parametru „režim.“ Jestliže je tento bit nulový, proces má veřejná přístupová práva. Privátní přístupová práva znemožní ostatním procesům provádět s tímto procesem některé vybrané operace. Například služba pro odložení procesu vyžaduje veřejná práva odkládaného procesu. Při spuštění procesu ale už povolení vyžadováno není. Proces může být spuštěn jen jiným procesem, proto by taková ochrana neměla žádný význam a znemožňovala by spuštění „privátních“ procesů.

4.6 Ukázkové programy

Součástí projektu je i šest ukázkových programů, které na jednoduchých příkladech postupně předvádějí práci s nejdůležitějšími částmi systému. Ukázkové programy se nesnaží předvést všechny služby systému na komplikovaných příkladech řešících komplexní problémy. Naopak jsou koncipovány tak, aby v co nejjednodušší formě naučily uživatele především inicializovat systém, vytvářet a spouštět procesy, zajistit komunikaci mezi procesy a pracovat s časem. Příkladem může být program „demo6,“ který je zaměřen na komunikaci mezi procesy prostřednictvím zpráv. Tento ukázkový program pro svou správnou činnost vyžaduje vývojový kit s připojeným displejem. Činnost programu „demo6“ je rozdělena do dvou procesů. Proces „testovaciProces2“ (P2) získává v pravidelných časových intervalech hodnotu z AD převodníku, kterou následně porovnává s předchozí uloženou hodnotou. Jestliže dojde ke změně této hodnoty, nejprve odešle zprávu pozastavenému procesu „testovaciProces1“ (P1), do které vloží aktuální hodnotu AD převodníku a následně tento pozastavený proces spustí. Jakmile je procesu P1 přidělen procesor, vyjme tuto zprávu z fronty zpráv a aktualizuje obsah displeje. Nakonec se proces P1 sám pozastaví. Kompletní zdrojový kód se nachází v souboru demo6.asm.

5 PŘEHLED SLUŽEB

V následujícím přehledu nejsou zahrnuty služby využívané pouze systémem (například plánovač). V přehledu je vždy název služby se seznamem parametrů psaných kurzívou. Zápis odpovídá volání služby pomocí maker. Následuje vysvětlení jednotlivých parametrů a krátký popis služby. Součástí názvu parametru je vždy číslo udávající počet bitů parametru.

INIT *param8, KBIproc16*

param8	nevyužito, parametr upřesňující inicializaci
KBIproc16	adresa podprogramu volaného při KBI přerušení, pouze pokud je v params.inc definováno DEFINE_KBI_PROC

Inicializace systémových proměnných na počáteční hodnoty. První služba, která musí být volána.

START

Start systému. Spustí se časovač a začne se provádět první proces. Pokud před voláním služby neproběhla inicializace, vrátí v registru A hodnotu ERROR_FAIL.

START_EX *initProc16*

initProc16	adresa podprogramu, který je zavolán těsně před spuštěním systému

Rozšíření služby start o volání podprogramu, do kterého se umisťuje především počáteční vytvoření procesů.

CREATE_PROCESS *rezim8, priorita8, velikostPameti8, program16*

rezim8	režim procesu, může být nastaven na binární hodnoty PROC_BLOCKED, PROC_RUNNING a PROC_PRIVATE
priorita8	priorita procesu v rozmezí 1 až 255

velikostPameti8 velikost požadované paměti v segmentech, velikost segmentu je MEM_SEG_SIZE bajtů

program16 adresa začátku programu

Vytvoření nového procesu. K požadované paměti je přičtena paměť pro hlavičku o velikosti PROC_HEAD_SIZE_SEG segmentů. V registru X vrací ID vytvořeného procesu a v registru A potvrzení o zdaru RTN_OK, popřípadě chybové hlášení.

ERROR_FULL_PROC_TABLE - plná tabulka procesů
ERROR_BAD_ALLOC – chyba správce paměti

START_PROCESS *id8*

id8 ID procesu

Spuštění vytvořeného a pozastaveného procesu. Návratová je v registru A.

DELAY_PROCESS *id8*

id8 ID procesu

Odložení spuštěného procesu, opět spuštěn musí být jiným procesem. Proces by měl uvolnit všechny přidělené systémové prostředky, aby je neblokoval. Je potřeba, aby měl volající proces povolení, odkládaného procesu (odkládaný proces musí mít veřejná přístupová práva), jinak systém vrátí ERROR_PRIVATE_ACCES. Návratová hodnota je v registru A.

DELAY_THIS_PROCESS

Odložení právě běžícího procesu. Po odložení se začne vykonávat další proces čekající ve frontě. Funkce služby je stejná jako DELAY_PROCESS. Jestliže DELAY_PROCESS odkládá právě běžící proces, automaticky použije DELAY_THIS_PROCESS.

DESTROY_PROCESS *id8*

id8 ID procesu

Zrušení procesu. Nutná veřejná práva rušeného procesu. Proces uvolní systémové prostředky a přestane existovat. Před ukončením musejí být uvolněny všechny zdroje obsazené procesem, jinak systém vrátí chybové hlášení ERROR_ALLOC_RESOURCES.

DESTROY_THIS_PROCESS (RETURN)

Zrušení právě vykonávaného procesu. Ostatní vlastnosti jsou stejné jako u DESTROY_PROCESS. Název RETURN je ekvivalentem k DESTROY_PROCESS a používá se pro ukončení programů.

RETURN_ABORT

Ukončení procesu, nedojde ovšem ke zrušení procesu, ale jen k jeho restartování na počáteční stav a odložení.

BREAK_PROC

Explicitní vyvolání změny kontextu (stejný efekt jako při přerušení časovače).

SET_PROC_PRIORITY *id8, prior8*

id8 ID procesu

prior8 nová priorita procesu

Nastavení nové priority procesu. Proces musí mít veřejný přístup, jinak vrací ERROR_PRIVATE_ACCES.

GET_CRITICAL_SECTION

Pokus o vstup do kritické sekce. Při neúspěchu vrátí v registru A hodnotu ERROR_FAIL. Při vstupu d kritické sekce vrátí v registru H:X adresu začátku kritické oblasti.

WAIT_FOR_CRITICAL_SECTION

Pomoci BREAK_PROC provádí aktivní čekání na kritickou sekci. Při úspěchu vrátí adresu začátku kritické oblasti.

RELEASE_CRITICAL_SECTION

Uvolnění kritické sekce. Volá se v páru společně s GET_CRITICAL_SECTION nebo WAIT_FOR_CRITICAL_SECTION. V případě, že se kritickou sekci pokouší uvolnit proces, který se v ní nenachází, vrátí v registru A hodnotu ERROR_FAIL.

SEND_MESSAGE *id8, msg16*

id8 ID procesu, kterému se zpráva zašle

msg16 zpráva

Zaslání zprávy procesu. Jestliže je fronta zpráv plná, v registru A vrátí hodnotu ERROR_FULL_MSG_QUEUE.

GET_MESSAGE

Vybere zprávu z fronty, zprávu z fronty odstraní. Zpráva má 3 bajty. V registru A vrací ID odesilatele, v registru H:X zprávu. Jestliže se žádná zpráva ve frontě nenachází, v registru A vrátí hodnotu ERROR_FAIL a oba bajty registru H:X jsou nulové.

PEEK_MESSAGE

Přečte zprávu uloženou ve frontě, ale neodstraní ji. Jinak se chová stejně jako GET_MESSAGE.

WAIT_FOR_MESSAGE

Čekání na zprávu ve frontě. Nespotřebovává zbytečně čas procesoru. Pokud je zpráva ve frontě, vrátí řízení zpět procesu. Zprávu z fronty nevyjme, ani ji nepřečte.

MESSAGE

Vrací nulovou hodnotu, jestliže není ve frontě žádná zpráva a nenulovou, pokud je.

WAIT_CYCLE *pocetCyklu16*

pocetCyklu16 počet čekacích cyklů, vyšší bajt parametru se ignoruje (možné rozšíření)

Aktivní čekací smyčka. Procesu vrátí řízení až po uběhnutí zadaného počtu systémových cyklů. Smyčka nespotřebovává čas procesoru, ale uvolňuje ho ostatním procesům.

GET_SYS_MESSAGE *param8*

param8 poslední nebo předchozí zpráva

Přečte systémovou zprávu. Pokud je parametr kladný, přečte poslední uloženou zprávu, pokud záporný, přečte předchozí zprávu.

GET_SYS_MESSAGE_EX *param8*

param8 poslední nebo předchozí zpráva

Stejné jako GET_SYS_MESSAGE, navíc uloží na zásobník systémový čas, kdy byla zpráva uložena.

GET_PROC_PRIORITY

V registru A vrátí prioritu běžícího procesu.

GET_PROC_ID

V registru A vrátí ID běžícího procesu.

GET_SYSTEM_TIME

V registru H:X vrátí systémový čas.

RTOS_GET_PROC_STATE

V registru A vrátí bitovou proměnnou indikující stav procesu. Stavy procesu jsou:

PROC_BLOCKED - blokovaný proces
PROC_RUNNING - běžící proces
PROC_MESSAGE - zpráva čekající ve frontě
PROC_PRIVATE - privátní nebo veřejná přístupová práva
PROC_EXCLUSIVE - výlučný přístup
PROC_CSECTION - proces je v kritické sekci
PROC_HAVE_PORT - procesu je přidělen port

SET_PROC_STATE *state8*

state8 - bitový parametr, nastavení stavu procesu, může být: PROC_PRIVATE

Nastaví stav procesu. Jsou nastaveny všechny bity podle masky mPROC_SET_STATE. Nepodporované bity jsou ignorovány.

GET_PORT *port16*

Port16 - identifikátor portu, hodnoty PORT_A až PORT_G

Pokusí se přidělit procesu port. Pokud je port volný, přidělí ho a vrátí RTN_OK, jinak ERROR_FAIL. Pokud není v parametrickém souboru definováno DEFINE_EXT_PORT_ACCES, vrátí vždy RTN_OK.

RTOS_RELEASE_PORT

Uvolní přidělený port.

STOPS

Ukončení činnosti systému. Pro opětovné spuštění je nutné opět systém inicializovat. Řízení se vrátí za místo volání služby START.

5.1 Funkce jazyka C, C++

Služby operačního systému je možné volat i v projektech jazyka C nebo C++. Všechna makra volání služeb mají své protějšky v souboru kernel.h. Názvy funkcí se shodují s názvy maker s tím rozdílem, že jsou psány malými písmeny, jen první znak každého slova názvu je psán velkým písmenem a vždy začínají slovem „Rtos.“ Makru CREATE_PROCESS tedy odpovídá funkce RtosCreateProcess(). Součástí projektu jazyka C musí být také soubor ckernel.asm, který obsahuje samotné definice funkcí, jejichž prototypy jsou v souboru kernel.h. Tyto funkce tvoří rozhraní mezi funkcemi jazyka C a samotnými službami operačního systému.

5.2 Uživatelská dokumentace

Kompletní přehled služeb, včetně funkcí jazyka C, se nachází v přiložené uživatelské dokumentaci (soubor dokumentace.pdf). Dokumentace obsahuje také stručný popis programování v prostředí operačního systému včetně jednoduchých příkladů.

SHRNUTÍ

Proces je běžící program a jako takový je základním prvkem celého systému. Prakticky je reprezentován datovou strukturou označovanou PCB. Systém je postavený na preemptivním plánování. Procesy jsou přerušovány v pravidelných intervalech časovačem. Při každém takovémto přerušení musí systém uložit kontext přerušeného procesu a vybrat další proces, kterému bude přidělen procesor. K tomu slouží plánovač, který vybírá procesy na základě jejich priority. Proces s vyšší prioritou má přednost před procesem s nižší prioritou. Aby nedocházelo ke stárnutí procesů, je možné aktivovat funkci plánovače, která při každém přerušení časovače zvýší jejich prioritu. Dynamické přidělování paměti umožňuje alokovat procesu různě velké bloky paměti za běhu systému. Aby si systém udržoval informace o tom, která paměť je volná a která obsazená, vede si bitovou tabulku, mapující úseky paměti o předem dané velikosti. Při požadavku na přidělení paměti prochází tuto tabulku a hledá souvislý volný blok. Procesy spolu mohou komunikovat pomocí sdílené paměti nebo prostřednictvím zasílání zpráv. Oba způsoby jsou užitečné a mají své výhody a nevýhody. Sdílená paměť umožňuje přistupovat ke stejným datům všem procesům. Úkolem operačního systému je zajistit k těmto datům výlučný přístup. Kritická oblast je část sdílené paměti, do které může mít přístup v jednom okamžiku jen jeden proces. Systém obsahuje několik kritických oblastí, jejichž počet je možné změnit v parametrickém souboru. Do každé kritické oblasti může mít přístup právě jeden proces. Procesy si mohou vzájemně zasílat zprávy. Každá zpráva má tři bajty a procesy si udržují frontu zpráv, do které jsou ukládány. Opět si je možné vybrat mezi jednodušší frontou pro jednu zprávu nebo frontou pro více zpráv. Systém má vlastní frontu zpráv, do které posílá systémové zprávy informující procesy o událostech, jako je přerušení. Některé vlastnosti systému je možné upravit, vypnout nebo omezit pomocí parametrického souboru params.inc, který je součástí projektu. Vypnutí některých funkcí sníží paměťové nároky systému.

ZÁVĚR

V rámci práce byl vytvořen jednoduchý operační systém pro mikropočítač s procesorem HCS08, který bude využíván jako pomůcka při výuce programování mikropočítačů. Operační systém je velmi komplexní software a jeho vytvoření vyžaduje vynaložení značného úsilí. Správný návrh datových struktur a základních principů je klíčový a má vliv na celou stavbu systému. Jednotlivé programové části na sebe navazují a chyba v jedné může způsobit pád celého systému. Vytvořený operační systém se snaží tyto části udržovat co nejvíce oddělené, aby je bylo možné snadno upravovat a popřípadě nahradit lepším řešením. Nejdůležitějšími součástmi systému jsou správa procesů, přidělování a odebírání zdrojů a komunikace mezi procesy. Systém podporuje dynamickou správu paměti, což umožňuje efektivně přerozdělovat různě velké bloky paměti za běhu systému. Preemptivní multitasking vytváří iluzi paralelního zpracovávání procesů, vyžaduje však náročnější režii systému. Algoritmus plánovače vybírá procesy podle jejich priority tak, že proces s vyšší prioritou má přednost před procesem s nižší prioritou. Komunikaci mezi procesy umožňuje sdílená paměť a zasílání zpráv. Systém má také nástroje pro předcházení uváznutí a souběhu procesů. Poskytuje také množství služeb rozšiřujících jeho možnosti například o efektivní čekací smyčky nebo bezpečnou práci s porty. S rostoucím množstvím funkcí se zvyšuje počet událostí majících vliv na správný chod systému. Velká část zdrojového kódu připadá právě na kontrolu a ošetřování nesprávných vstupních dat a odhalování chybových stavů.

Systém byl úspěšně otestován a odladěn na vývojovém kitu M68EVB08GB60. Součástí práce je také několik jednoduchých ukázkových programů, které demonstrují správné použití služeb systému při tvorbě uživatelské aplikace.

ZÁVĚR V ANGLIČTINĚ

A simple operating system for HCS08 microcontrollers was created. This system will be used in teaching microcontroller programming.

Operating system is very complex software and its creation requires great effort. Correct plan of data structures and basic principles is crucial and affects the whole building system. Individual program parts are in close interaction and error in one can bring down the whole system. Created operating system tries to keep these parts separate, as far as possible, so that they can be easily adjusted and replaced, where appropriate, with a better solution. The most important components of the system are task management, resource allocation and communication between processes. The system supports dynamic memory management, allowing efficient allocation of memory blocks of varying sizes on the fly. Preemptive multitasking creates the illusion of parallel processing of tasks, however, requires a more demanding system overhead. Scheduling algorithm selects the tasks according to their priorities so that the task with higher priority takes precedence over the task with lower priority. Inter-process communication is enabled using shared memory and messaging. The system also has the tools to prevent deadlock and overlapping of processes. It also provides a number of services expanding its options, for example, efficient wait or secure access to the ports. With increasing amount of functionality the number of events affecting the correct operation of the system is rapidly increasing. Much of the source code serves for controlling and treatment of incorrect input data and detection of error states.

The system was successfully tested on the learning kit M68EVB08GB60. The thesis also contains simple example program which illustrates correct usage of the system when creating user applications.

SEZNAM POUŽITÉ LITERATURY

[1] MC9S08GB/GT Data Sheet [online]. Freescale Semiconductor, 2004 [cit. 2009-01-26]. Dostupný z WWW: <http://www.freescale.com/files/microcontrollers/doc/data_sheet/MC9S08GB60.pdf>.

[2] HCS08 Family Reference Manual [online]. Freescale Semiconductor, 2007 [cit. 2009-01-26]. Dostupný z WWW: <http://www.freescale.com/files/microcontrollers/doc/ref_manual/HCS08RMV1.pdf>.

[3] CPU08 Central Processor Unit Reference manual [online]. Freescale Semiconductor, 2006 [cit. 2009-01-26]. Dostupný z WWW: <http://www.freescale.com/files/microcontrollers/doc/ref_manual/CPU08RM.pdf>.

[4] VÁŇA, V. Začínáme s mikrokontroléry Motorola HC08 Nitron. Praha: BEN -- technická literatura, 2003. 96 s. ISBN 80-7300-124-1.

[5] MANN, B. C pro mikrokontroléry. Praha: BEN - technická literatura, 2004. 280 s. ISBN 80-7300-077-6.

[6] SROVNAL, V. Operační systémy pro řízení v reálném čase, VŠB Technická universita Ostrava 2003, ISBN 80-248-0503-0.

[7] SOLOMON, S. A. Windows NT pro administrátory a vývojaře. Praha: Computer Press, 2000. 492 s. ISBN 80-7226-147-9.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

AD	Analogově Digitální
ALU	Aritmetic Logic Unit
CCR	Condition Code Register
CPU	Central Procesing Unit
FIFO	First In First Out
GUI	Graphical User Interface
ID	Identifikátor
IDE	Integrated Development Environment
KBI	Keyboard Interrupt Module
KLT	Kernell Layer Thread
LIFO	Last In First Out
PCB	Process Control Blocks
RAM	Random Acces Memory
ROM	Read Only Memory
RTOS	Real Time Operating System
ULT	User Layer Thread

SEZNAM OBRÁZKŮ