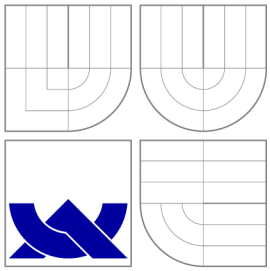
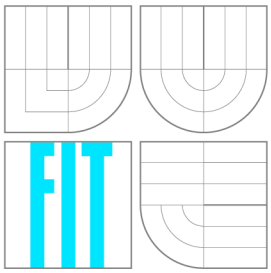


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

NÁVRH A IMPLEMENTACE JÁDRA VÍCEÚLOHOVÉHO OPERAČNÍHO SYSTÉMU BĚŽÍCÍHO NA PLATFORMĚ HC08

DESIGN AND IMPLEMENTATION OF A MULTITASK OPERATING SYSTEM KERNEL RUNNING ON HC08

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — ROMAN DAMBORSKÝ
AUTHOR

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

BRNO 2007

Zadání

Návrh a implementace jádra víceúlohového operačního systému běžícího na platformě HC08

1. Seznamte se s architekturou a programovým modelem mikrokontrolérů řady HC08.
2. Seznamte se s vývojovými nástroji pro tuto řadu mikrokontrolérů a se základy jejich programování v asembleru a jazyce C.
3. Pro zadaný typ mikrokontroléru řady HC08 navrhněte rozhraní jádra víceúlohového operačního systému.
4. Jádro z bodu 3 implementujte v asembleru nebo jazyce C.
5. Navrhněte a implementujte několik jednoduchých procesů.
6. Procesy z předchozího bodu využijte k ověření funkčnosti systému přímo v mikrokontroléru.

Kategorie: Vestavěné systémy

Licenční smlouva

Licenční smlouva je uložena v archivu Fakulty informačních technologií Vysokého učení technického v Brně.

Abstrakt

Software pro vestavěné systémy je často navrhován tak, aby plnil jediný úkol. Pokud je ale požadováno provádění více úloh současně, bývá řešení jednoúčelové, bez možnosti použití základu programu pro jinou aplikaci. Navrhnul jsem proto rozhraní, které umožní nezávisle na povaze jednotlivých úloh jejich současné zpracování. Jádro je implementováno s ohledem na znovupoužitelnost. Při návrhu rozebírám jednotlivé přístupy k řešení. Pro implementaci jsem použil plánovač úloh založený na algoritmu Round–Robin. Víceúlohovosti je dosaženo pravidelným přepínáním jednotlivých úloh, s využitím přerušovacího podsystému. Jako cílovou architekturu jsem zvolil mikrokontroléry Motorola řady HC08.

Klíčová slova

jádro, víceúlohovost, víceúlohový operační systém, HC08, přepínání procesů, preempce, periodické úlohy

Abstract

Software for embedded systems is usually designed for performing one particular task. If there is need to serve more tasks at once, solution is used to be dedicated without potential reusability for another application. That is why I've designed an interface which allows simultaneous execution of single tasks independently of their character. Kernel is implemented in consideration of reusability. I analyse individual approaches to solution. I used Round–Robin algorithm for implementing tasks management. Multitasking is achieved by periodical switching of single tasks. Interrupt subsystem is being used for this. As a target architecture, Motorola HC08 microcontrollers were chosen.

Keywords

kernel, multitasking, multitask operating system, HC08, processes switching, preemption, periodical tasks

Citace

Roman Damborský: Návrh a implementace jádra víceúlohového operačního systému běžícího na platformě HC08, bakalářská práce, Brno, FIT VUT v Brně, 2007

Návrh a implementace jádra víceúlohového operačního systému běžícího na platformě HC08

Prohlášení

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

.....
Roman Damborský
14. května 2007

Poděkování

Rád bych poděkoval vedoucímu mé bakalářské práce Ing. Josefu Strnadelovi, Ph.D., za podnětné připomínky a odborné rady. Děkuji také Ing. Richardu Růžičkovi, Ph.D., za poskytnutí vývojového kitu k praktickému ověření funkčnosti implementovaného jádra.

Obsah

Kapitola 1

Úvod

Když v polovině 20. století vznikaly první prototypy analogových počítacích strojů, stěží si mohli jejich návrháři a konstruktéři představit, že by vytvořený systém mohl být použitelný pro jiný, než jimi analyzovaný úkol. S nástupem digitálních technologií se objevily první programovatelné počítače. Neumožňovaly sice jednoduše zcela změnit typ úlohy, pro kterou byly navrženy, ale byly již vybaveny mechanismy pro modifikaci některých parametrů. Tím bylo dosaženo jisté variability úloh a lze říci že i znovupoužitelnosti. Možnost kompletní obměny celého algoritmu stroje byla dalším logickým krokem ve vývoji.

Jakmile byly známy způsoby použití počítače k provádění více různých úloh, došlo na otázku, zda by nebylo možné zajistit vykonávání více úloh *současně*. Výpočetní čas počítače byl totiž velmi drahý a periferie pomalé. Většinu času tedy procesor počítače čekal na dokončení periferních operací. Víceúlohovost přinesla možnost využít čas, kdy procesor čeká na dokončení periferní operace, k provádění další úlohy.

V 80. letech 20. století došlo k masovému rozšíření počítačů. To bylo umožněno jednak díky technologii výroby hardware, ale stejně důležitou roli sehrála i „duše“ počítače, tedy jeho programové vybavení. Ne každý totiž rozuměl tomu, jak konkrétní počítač funguje a jak komunikovat s připojenými zařízeními. Tady přichází do hry *operační systém* (dále OS), který má za úkol zpříjemnit uživateli práci s počítačem a jeho periferiemi. Složitost a propracovanost jednotlivých operačních systémů se může lišit. V některých případech se vyžaduje pouze základní funkcionalita zahrnutá v tzv. *jádru*, jindy poskytuje OS funkce pro práci s celou řadou periferií, grafické uživatelské rozhraní atd. Příkladem jednodušších OS mohou být např. OS/360 nebo CP/M. Mezi složitější OS pak lze zařadit Linux, Mac OS, Microsoft Windows.

S technologickým vývojem v oblasti elektroniky a počítačových systémů se OS začínají objevovat i mimo osobní počítače. Stále častěji je lze najít jako součást ve výrobních procesech, automobilovém průmyslu, zabezpečovací a komunikační technice apod. Většinou se zde vyskytují ve formě *vestavěných zařízení*.

V této práci navrhnu jádro operačního systému, který bude umožňovat současný běh více úloh. Cílovou platformou bude rodina mikrokontrolérů HC08, jejíž charakteristické vlastnosti popisuje kapitola 3. O různých přístupech k řešení problematiky víceúlohovosti pojednává kapitola 4. Z popsaných principů popíši v kapitole 5 konkrétní řešení pro vybraný typ mikrokontroléru. Pro ověření funkčnosti jádra jsem navrhnul několik jednoduchých testovacích úloh, popsaných v kapitole 6. Zhodnocení výsledků práce a návrhy k dalšímu rozšiřování uvádím v kapitole 7.

Kapitola 2

Cíle a motivace

Cílem práce je navrhnout a implementovat jádro víceúlohového operačního systému pro vestavěné zařízení.

Náš OS bude zastoupen pouze na té nejnižší úrovni, formou jádra. Bude plnit pouze základní funkce, nezbytné k zajištění víceúlohovosti. Jako cílová platforma je uvažována rodina mikrokontrolérů Motorola HC08 [2].

Specifikujme si základní požadavky na naše jádro a práci s ním:

- Vkládání úloh musí být jednoduché a uživatelsky přívětivé
- Přepínání jednotlivých úloh bude zajišťováno automaticky
- Bude přítomný mechanismus pro indikaci chyby při inicializaci jádra
- Paměťový prostor jednotlivých úloh bude chráněn jádrem
- Paměťová náročnost jádra bude minimální
- Přepínání úloh bude časově co nejefektivnější
- Pokusíme se dosáhnout přenositelnosti mezi jednotlivými variantami mikrokontrolérů

Splníme–li uvedené požadavky, získáme systém, s jehož pomocí bude možno zajistit provádění více úloh zdánlivě souběžně. Vlastní úlohy budou na jádru nezávislé, tzn. bude možné jejich použití i bez jádra (Nebereme zde ale v potaz případná sdílená data mezi úlohami).

Po implementaci a úspěšném ověření funkčnosti se budeme lépe orientovat v základní problematice funkce jádra operačního systému. Rozšíříme si také znalosti týkající se zvoleného mikrokontroléru a jemu příbuzných typů.

Kapitola 3

Popis architektury HC08

3.1 Vestavěné systémy

Počítačové systémy nás v současné době obklopují ze všech stran. Typickým představitelem je osobní počítač (PC). Hlavním rysem PC je jejich univerzálnost, tedy možnost využít výpočetní výkon k několika různým úkonům, od zpracování video záznamu, přes vedení databáze klientů, až po kancelářské práce a zábavu.

Vestavěným systémem rozumíme takový systém, který je součástí jiného zařízení. Data zpracovávaná vestavěným systémem jsou uživateli skryta. Narozdíl od PC plní vestavěné systémy specifické úkoly, které se nemění. Jedná se např. o dekódovací obvody v televizoru, regulaci teploty ve vytápěcím zařízení apod.

Vestavěné systémy prošly za dobu své existence významným vývojem, který v současné fázi umožňuje jejich nasazení téměř v každé oblasti lidské činnosti. Zpočátku byly realizovány pomocí pevné logiky, používané pro řízení jednodušších aplikací. Později byly vyvinuty mikroprocesory, které dovolily modifikovat řídící program podle aktuální potřeby. Slabým místem mikroprocesorů bylo použití externího paměťového podsystému. S nástupem mikrokontrolérů byla externí paměť přesunuta na stejný čip jako mikroprocesor. Došlo tak k další miniaturizaci řídících systémů.

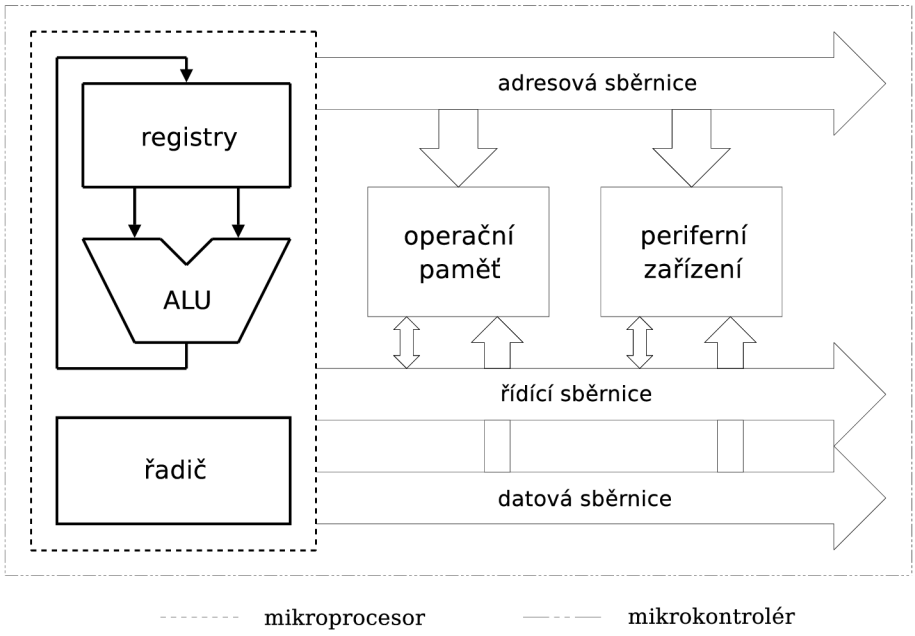
3.1.1 Mikroprocesor

Mikroprocesor je základní procesorovou jednotkou (dále CPU). Obsahuje aritmeticko logickou jednotku (ALU), dekodér instrukcí, registry a řadič. Blokové schéma mikroprocesoru je uvedeno na obrázku 3.1 (ohraničeno čárkovaně). Řadič řídí chod celého mikroprocesoru. Je tvořen registrem instrukcí, dekodérem instrukcí a řídícím obvodem. Tato část je nazývána *řídící*. Pomocí sběrnice je propojena s *paměťovou* částí tvořenou ALU a souborem registrů.

Pro vytvoření počítače je nutno přidat vnější operační paměť a potřebné periferní jednotky, připojené pomocí adresové, řídící a datové sběrnice. Lze tak vytvořit jednodeskový počítač.

3.1.2 Mikrokontrolér

Mikrokontroléry neboli *mikropočítače* získávají v posledních několika letech značnou pozornost. Díky jejich miniaturním rozměrům je možné je zahrnout do většiny zařízení, ve kterých je vyžadována složitější logika a komunikace s uživatelem. Blokové schéma mikrokontroléru je znázorněno na obrázku 3.1.



Obrázek 3.1: Struktura mikroprocesoru a mikrokontroléru

Narozdíl od mikroprocesorů obsahují na jednom čipu navíc operační paměť a periferní jednotky. Jádro zůstává u jednotlivých členů rodiny stejné, liší se velikost paměti a jednotlivé periferie. Důležitou roli hraje šířka datové sběrnice, podle které dělíme mikrokontroléry na osmi, šestnácti a dvaatřiceti bitové. Mezi nejznámější výrobce patří Atmel, Motorola (nově Freescale), Microchip (PIC), Siemens a Toshiba.

3.1.3 CISC, RISC

Každá rodina, případně každý typ mikroprocesorů, se od ostatních liší svou instrukční sadou. Mikroprocesory pak podle vlastností instrukční sady dělíme na dvě skupiny:

CISC

Z anglického „Complex Instruction Set Computer“ – počítač s komplexní instrukční sadou. Tyto procesory se jednoduše programují a efektivně využívají paměť. Díky velkému počtu složitých instrukcí a adresovacích módů bylo dosaženo zmenšení programů a nižšího počtu přístupů do hlavní paměti.

Tento přístup ale nebyl vždy nejefektivnějším. V případě příliš složitých instrukcí bylo jejich dekódování a provedení příliš náročné na hardwarovou stavbu mikroprocesoru. Vyplatilo se tedy použít větší množství jednodušších instrukcí, které byly ve výsledku vykonány rychleji a s menšími nároky na strukturu mikroprocesoru.

Mezi představitele této kategorie lze zařadit Intel 80x86, VOX a Motorolu 68K.

RISC

Z anglického „Reduced Instruction Set Computer“ – počítač s redukovanou instrukční sadou. Tato koncepce se netýká pouhého redukování rozsahu instrukčního soboru. Popisuje

také, jak by měl být počítač realizován a jak by měl fungovat. Poznatky využité při navrhování této architektury vycházejí z reálného programování mikroprocesorů.

Provádění jednodušších, hardwarově lépe navržených instrukcí je rychlejší a programy jsou efektivnější, než v případě použití složitých instrukcí. Vypuštěním složitých instrukcí došlo ke zjednodušení přístupu k paměti a bylo tak umožňeno její další vylepšování.

Nevýhodou architektury je častější čtení instrukcí z programové paměti.

3.2 Platforma HC08

Pro implementaci jádra OS jsem jako cílový mikrokontrolér zvolil typ 68HC908LJ12 firmy Motorola. Hlavním důvodem bylo využití programovacího kitu LJ12EVB z laboratoří fakulty. Rodina osmibitových mikrokontrolérů HC08 je vylepšenou verzí řady HC05, založené na mikroprocesoru HC6800. Tímto máme zajištěnu kompatibilitu objektového kódu z řady HC05 do HC08.

Popis označení MC68HC908LJ12:

- **MC** – výrobce (Motorola)
- **68HC908** – rodina mikroprocesorů a použitá technologie (908 – FLASH verze)
- **LJ** – řada procesorů, označení použitých periferií (viz kapitola 3.7)
- **12** – přibližná velikost hlavní paměti v kB

Flash paměť je pro mikrokontroléry obecně velkým přínosem. Umožňuje nám naprogramovat mikrokontrolér bez použití speciálního programátoru. Navíc může s touto pamětí operovat i uživatelský program. Lze tak tedy emulovat pamět EEPROM, zapisovat do FLASH nejrůznější provozní informace, či implementovat přepisování programu za běhu.

Rodiny mikrokontrolérů HC08 jsou postaveny na jádře CPU08. Mezi typické vlastnosti tohoto jádra patří minimální počet registrů (akumulátor, indexovací registr, ukazatel na zásobník, programový čítač a stavový registr). Počet registrů je vyvážen rychlým a flexibilním přístupem do paměti (16 adresovacích módů). Lze pracovat s proměnnými na přímých adresách, indexovat, pracovat s proměnnými na zásobníku, nebo tyto možnosti kombinovat. Tyto techniky jsou navíc podpořeny jednotným adresovým prostorem pro paměť dat, programu i periferie. Instrukční sada je tak jednodušší a přehlednější. Z těchto poznatků vyplývá, že HC08 je zástupcem von Neumannovy architektury počítačů[1].

Další vlastnosti jádra CPU08:

- jednocyklový přístup do paměti
- až dva přístupy do paměti v jedné instrukci
- taktování CPU a sběrnice 0 – 8 MHz
- jednoduché instrukce s dobou vykonávání 1T – 3T, složitější 7T (T – strojový cyklus)

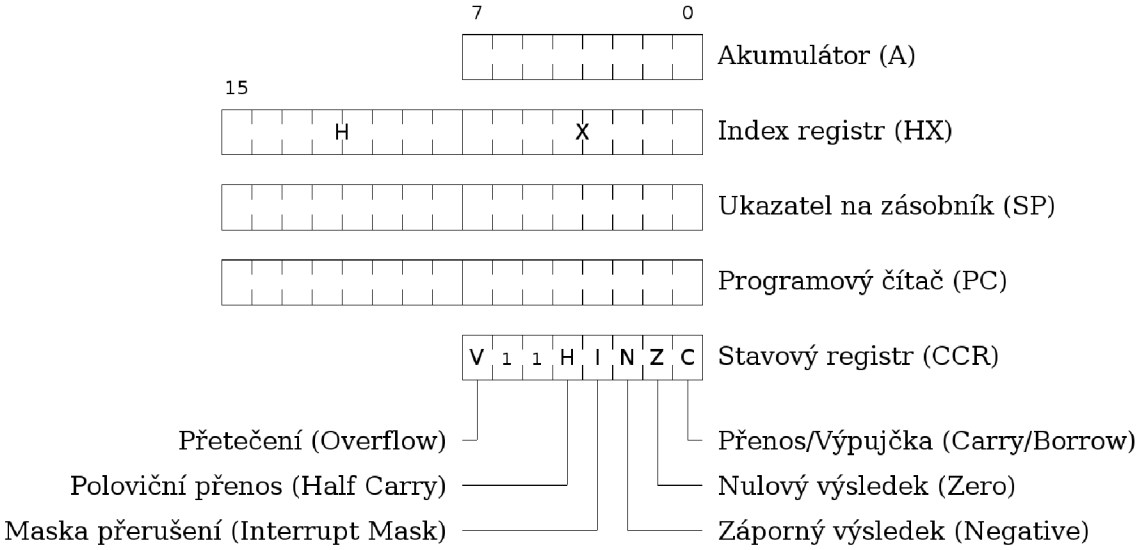
[1] Jinou konstrukci a přístup zajišťuje Harvardská architektura, kde je oddělen paměťový prostor pro program a data. Výhodou je větší rozsah adresového prostoru, nevýhodou jsou pak složitější a časově náročnější instrukce.

- hardwarové násobení a dělení
- podpora programovacího jazyka C
- podpora režimu se sníženou spotřebou
- ochrany systému pomocí Watchdogu, kontroly nízkého napětí, detekce neplatné adresy nebo operandu instrukce

3.3 Programovací model

Programovací model je soubor registrů, které musí programátor znát, aby mohl pro mikrokontrolér vytvořit program. Kromě registrů mikroprocesoru potřebuje znát instrukční sadu, případně specifika vyššího programovacího jazyka.

Registry mikroprocesoru jsou znázorněny na obrázku 3.2. Jsou umístěny v mikroprocesoru a nejsou tedy součástí pamětového prostoru.



Obrázek 3.2: Programovací model HC08

Popis jednotlivých registrů:

- **akumulátor**[2] – **A** pro obecné použití. CPU jej využívá pro ukládání operandů a výsledků aritmeticko–logických operací.
- **index registr** – **HX** umožňující adresování 64kB adresového prostoru. Skládá se ze dvou osmibitových registrů *H* a *X*. V indexovém adresovacím módu používá CPU obsah HX registru k výpočtu absolutní adresy dané relativním operandem. Může být také využit jako dočasné úložiště dat.

[2] Jiným českým termínem je *střadač*.

- **ukazatel na zásobník – SP** ukazuje na první volnou pozici zásobníku. Po resetu mikrokontroléru je nastaven vrchol zásobníku na adresu `0x00ff`. Zásobník lze umístit kdekoliv do oblasti paměti RAM. Při vložení dat do zásobníku se SP snižuje, při vyjmutí údaje se opět zvyšuje. Zásobník se také využívá pro předávání parametrů podprogramům. S ukazatelem na zásobník se pak pracuje obdobně jako s HX registrem.

- **programový čítač – PC** obsahuje adresu příští instrukce nebo operandu. Po resetu mikrokontroléru je naplněn resetovacím vektorem, který je uložen na adrese `0xfffe` a `0xffff`. Hodnotou vektoru je adresa první instrukce programu. Při postupném vykonávání programu se PC zvyšuje, pokud dojde ke skoku do jiné části programu nebo je vyvolána obslužná rutina pro přerušení, je obsah PC nastaven přímo, podle kontextu.

- **stavový registr – CCR** obsahuje stav globální masky přerušení a další bity (*příznaky*), indikující výsledek právě provedené instrukce. Bity 6 a 5 jsou nevyužity a permanentně nastaveny na logickou 1.

  Význam ostaních bitů je následující:

  - **V** – *příznak přetečení* je nastaven, pokud dojde k přetečení výsledku v doplňkovém kódu.
  - **H** – *příznak polovičního přenosu* je nastaven při přenosu mezi třetím a čtvrtým bitem akumulátoru během operací `ADD` a `ADC`. Tento příznak je důležitý pro aritmetické operace v BCD kódu.
  - **I** – *maska přerušení* globálně ovlivňuje povolení přerušení. Je–li nastavena, žádné žádosti o přerušení se nevyhoví. Při výskytu přerušení je automaticky nastavena po uložení CPU registrů na zásobník, ale dříve než je načten vektor přerušení. Více o přerušení viz kapitola 3.6.
  - **N** – *příznak záporného výsledku* je nastaven, pokud při aritmetické, logické nebo přesunové operaci je výsledek záporný.
  - **Z** – *příznak nulového výsledku* je nastaven, je–li výsledek aritmetické, logické nebo přesunové operace nulový.
  - **C** – *přenos / výpůjčka* - je nastavena, pokud při operaci součtu dojde k přenosu ze sedmého bitu nebo při operaci rozdílu k výpůjčce. Příznak je také ovlivňován některými logickými operacemi a instrukcemi pro úpravu operandů (např. rotace, posuvy, nepodmíněné skoky).

Je nutné podotknout, že HC08 patří do skupiny mikroprocesorů typu *big–endian*, což je označení principu ukládání vícebytových dat do paměti. Big–endian definuje uložení významnějšího (vyššího) bytu dat na nižší adresu, než na jakou je uložen méně významný (nižší) byte[3].

[3] Např. uložíme–li na adresu `0x0150` hodnotu `0x1234`, bude na adrese `0x0150` hodnota `0x12` a na adrese `0x0151` hodnota `0x34`.

3.4 Paměťový model

Mikroprocesory CPU08 mohou adresovat 64kB adresový prostor. Organizace paměti je znázorněna na obrázku 3.3.

$0000	V/V registry 98 bytů
$0060	RAM 512 bytů
$0260	neimplementováno 48 544 bytů
$c000	Flash paměť 12 288 bytů
$f000	neimplementováno 3 072 bytů
$fc00	monitor ROM 1 512 bytů
$fe00	SIM stavový registr pro zastavení
$fe01	SIM stavový registr pro reset
$fe02	rezervováno
$fe03	SIM řídící registr pro zastavení
$fe04	1. stavový registr přerušení
$fe05	2. stavový registr přerušení
$fe06	3. stavový registr přerušení
$fe07	rezervováno
$fe08	řídící registr Flash
$fe09	registr blokové ochrany Flash
$fe0a	rezervováno
$fe0b	rezervováno
$fe0c	registr adresy zastavení (vyšší byte)
$fe0d	registr adresy zastavení (nižší byte)
$fe0e	stavový a řídící registr zastavení
$fe0f	stavový registr LVI
$fe10	monitor ROM 2 448 bytů
$ffd0	Flash vektory 48 bytů

Obrázek 3.3: Mapa paměti 68HC908LJ12

Přístup do neimplementované oblasti paměťového prostoru může způsobit reset, pokud je tato kontrola povolena. Pokud přistupujeme na rezervované paměťové místo, může dojít k nepředvídatelnému chování mikrokontroléru.

Paměťový prostor od adresy `0x0060` do `0x025f` včetně, je vyhrazen paměti RAM. V *nulté stránce*[4] je prvních 160 bytů RAM paměti. Je-li zásobník po resetu přesunut mimo nultou stránku, vznikne ideální místo pro uložení globálních proměnných.

[4]Prvních 255 bytů paměti – pro tuto oblast je optimalizován přímý adresovací mód. Přístup do této části paměti trvá nejkratší dobu a je proto vhodné uložit sem data, která nejčastěji používáme.

3.5 Instrukční sada

Zvolený mikrokontrolér (viz kapitola 3.2) obsahuje instrukční sadu typu CISC. Instrukce lze rozdělit do několika základních skupin. Pro každou skupinu budou uvedeny pouze některé instrukce. Kompletní popis instrukcí je nad rámec této práce. Vyčerpávající informace o instrukčním souboru lze najít na internetu nebo v literatuře [1].

1. **Instrukce pro přesuny dat** přesouvají data mezi pamětí/registry a registry HC08. Patří sem např.:
   - instrukce `LDA`, sloužící k zapsání hodnoty do akumulátoru nebo `STX` k uložení hodnoty z registru X do paměti.
   - instrukce pro práci se zásobníkem. Konkrétně např. `PSHA`, která ukládá na zásobník obsah akumulátoru, nebo `PULH`, která naplní registr H bytem z vrcholu zásobníku. **Při práci se zásobníkem je nutno brát na vědomí, že SP ukazuje vždy na další volné (zásobníkem nevyužité) paměťové místo.**
   - instrukce pro přenos dat mezi registry – `TAP` resp. `TAX`, zajišťující přesun bytu mezi akumulátorem a CCR resp. akumulátorem a X registrem.
2. **Aritmetické instrukce** provádějící základní operace s daty. Jsou to např. instrukce:
   - `ADD` resp. `SUB`, přičítající k resp. odečítající od akumulátoru hodnotu danou operandem. Výsledek ukládají zpět do akumulátoru.
   - `INCA` resp. `DECA` zvyšující resp. snižující hodnotu v akumulátoru.
   - `CMP` porovnávající obsah akumulátoru s hodnotou určenou operandem.
   - `MUL` resp. `DIV` provádějící součin resp. celočíselné dělení.
3. **Logické instrukce** umožňují programátorovi pracovat s konkrétními bity daných bytů. Z pohledu těchto instrukcí je byte reprezentován polem 8 bitů, přičemž nejvíce významný bit (nejlevější) je na indexu 7 a nejméně významný bit (nejpravější) na indexu 0. Mezi zástupce z této skupiny řadíme např. instrukce:
   - `AND` resp. `ORA` provádějící logický součin resp. součet bytu uloženého v akumulátoru a bytu určeného operandem. Výsledek ukládá zpět do akumulátoru. Instrukce je vhodná, chceme–li vynulovat resp. nastavit několik bitů v bytu a ostatní bity ponechat beze změny.
   - `BCLR` resp. `BSET` nulující resp. nastavující bit určený operandem bytu určeného operandem.
   - `ASLA` resp. `ROLA` provádějící aritmetický posuv resp. rotaci bitů v bytu doleva.
4. **Řídící instrukce** ovlivňující běh hlavního programu a umožňující rozčlenění programu na logické celky a jejich vzájemné provázání a znovupoužití. Do této kategorie spadají např. instrukce:
   - `BRA` provádějící nepodmíněný skok (provede se vždy).
   - `BLT` resp. `BGE` provádějící skok na návěští dané operandem, pokud byl výsledek poslední operace měnící CCR příznaky „menší než“ resp. „větší nebo rovno“.
   - `BRCLR` resp. `BRSET` provádějící skok na návěští určené operandem, pokud je určitý bit daného bytu vynulován resp. nastaven na 1.

3.6 Systém přerušení

Pro správné pochopení pojmu přerušení je vhodné nejdříve uvést význam podprogramu, z něhož lze při popisu přerušení vycházet.

3.6.1 Podprogram

Mezi charakteristické znaky efektivního programování lze s jistotou zařadit vhodnost dekompozice složitého problému na jednotlivé podproblémy. Správná úvaha a rozvržení algoritmu může ušetřit spoustu času, zdrojů i výpočetní kapacity. V oblasti vestavěných systémů je tento aspekt také velmi vítaný a proto obsahuje instrukční sada některé speciální instrukce, umožňující „rozčlenění“ kódu do logických celků, tzv. *podprogramů*.

Podprogram je kód, začínající unikátním návěštím (které lze chápat jako název podprogramu) a končící instrukcí `RTS` (návrat z podprogramu). Pro volání podprogramu se používají instrukce `BSR` nebo `JSR`[5]. Instrukce nejdříve uloží obsah PC na zásobník a poté provede skok na návěští dané operandem. Pro správnou funkci je vyžadováno, aby podprogram končil instrukcí `RTS`, která zajistí, že ze zásobníku se obnoví obsah PC registru tak, jak byl nastaven před skokem do podprogramu, a program tak může pokračovat tam, kde byl přerušen.

3.6.2 Obsluha přerušení

Narozdíl od podprogramu, jehož volání máme pevně definováno a jsou dány jednoznačné podmínky jeho vykonání, k přerušení může dojít asynchronně vůči toku dat hlavního programu.

Obsluha přerušení je odezvou na externí údálost během vykonávání hlavního programu. Po rozpoznání zdroje přerušení je žádost o přerušení za určitých podmínek přijata a řízení je předáno připravené obslužné rutině. Po jejím dokončení je řízení předáno zpět hlavnímu programu.

Definice jednotlivých obslužných rutin jsou umožněny pomocí tzv. *vektorů přerušení*. Jedná se o tabulku na konci adresového prostoru, která obsahuje adresy obslužných rutin. Tabulka má pevně určenou strukturu, aby bylo možné jednoznačně přiřadit k danému typu přerušení odpovídající obslužnou rutinu. V našem případě začíná tabulka na adrese `0xffd0`.

V tabulce vektorů přerušení je rovněž definován vektor pro reset mikrokontroléru, který určuje adresu první instrukce programu. Reset může být vyvolán několika způsoby, přičemž všechny mají společný vektor přerušení:

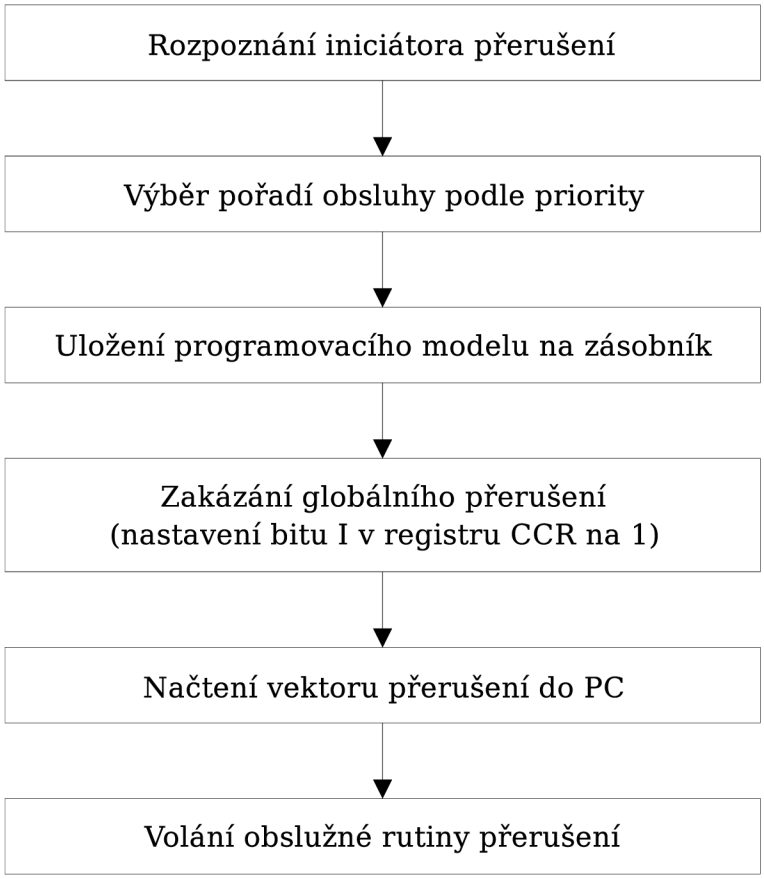
1. **externě** – při zapnutí mikrokontroléru, přivedením nízké úrovně na vnější vývod RESET.

2. **interně** – narušením správného běhu programu, poklesem napájecího napětí, detekcí neplatného operačního znaku a adresy.

Zdroje přerušení můžeme rozdělit na vnější a vnitřní. Vnějším přerušením je pouze signál IRQ. Vnitřní hardwarová přerušení jsou vyvolána jednotlivými periferiemi a jsou maskovatelná nastavením lokálních masek na vysokou úroveň. Všechny zdroje přerušení jsou pak globálně maskovatelné pomocí bitu I registru CCR. Je–li tento bit nastaven na log. 1, žádné žádosti o přerušní nebude vyhověno.

[5] Liší se v použitém adresování - instrukce BSR využívá relativní adresovací mód – více viz [1].

Proces detekce a provedení žádosti je uveden na obrázku 3.4.



Obrázek 3.4: Postup při volání obslužné rutiny přerušení

Nejdříve jsou modulem SIM (System Integration Module) určeny zdroje přerušení a je proveden výběr jednoho z nich, v případě že přišlo žádostí více. Poté je uložen programovací model na zásobník[6] a je nastavena globální maska přerušení, čímž se další žádosti o přerušení neuplatní. Do PC registru se uloží vektor přerušení pro přijatou žádost a nakonec se spustí obslužná rutina přerušení, jejíž adresa je v PC registru nastavena. Rutina končí instrukcí `RTI`, která obnoví programovací model, čímž nastaví hodnotu PC registru na adresu, na které došlo k přerušení. Program tak může pokračovat.

[6]Je třeba poznamenat, že při ukládání programovacího modelu není automaticky uložen registr H. V případě že se v obslužné rutině s tímto registrem pracuje, je nutné jej na začátku rutiny uložit a na konci opět obnovit „manuálně“.

3.7 Periferie

Na čipu mikrokontroléru je zahrnuto několik periferií, jejichž výčet a stručný popis následuje. Tato sestava periferií je pevně dána a je charakterizována písmeny LJ v označení mikrokontroléru. Detailnější popis lze najít v literatuře [1] nebo [3].

TIM – čítače/časovače – systém čítačů/časovačů TIM1 a TIM2. Umožňují použití jednotek záchytu hrany a výstupního komparátoru (Input capture a Output compare), s možností nastavit typ hrany (nástupná/sestupná/obojí) a akci při shodě (nastavení/vynulování/změna určitého bitu). Dále je možné použít časovače ke generování pulsů s proměnnou střídou (Pulsně šířková modulace – PWM). Tato technika se využívá např. k ovládání servomotorů, regulaci napětí atd. Při použití v podobě čítače lze nastavit horní limit čítání, směr čítání, událost při dosažení horního limitu apod.

CGM – generátor hodin – řídí chod mikrokontroléru, synchronizuje jednotlivé jeho části a řídí zpracování programu. CGM generuje střídavý periodický signál, jehož přesnost a frekvenci lze získat několika způsoby:

1. vestavěnými obvody krystalového oscilátoru, kdy stačí připojit pouze krystal o vhodné rezonanční frekvenci (jednotky až desítky MHz, podle typu mikrokontroléru).
2. vestavěnými obvody krystalového oscilátoru s fázovým závěsem, kdy stačí připojit levný krystal s nižší frekvencí. Fázový závěs pak vytvoří potřebný hodinový signál o frekvenci řádu MHz.
3. externí zdroj hodinového signálu.
4. vnitřní oscilátor, který nevyžaduje vnější součástky, ale nedosahuje takové přesnosti jako krystalové oscilátory.

COP – watchdog – systém pro hlídání správného běhu programu. Během vykonávání programu je nutné ve vhodných okamžicích volat speciální funkce, které informují COP obvod, že je vše v pořádku. Jakmile není COP takto informován, dojde k resetu mikrokontroléru a restartu programu.

RTC – obvod reálného času – slouží k podpoře hodin reálného času a kalendáře. Informuje o aktuálním času a datumu, umožňuje volat přerušení každou novou sekundu, minutu, hodinu nebo den, lze jej využít pro funkci budíku apod.

IRSCI – IR sériové synchronní rozhraní – synchronní sériová komunikace s podporou IR kodéru/dekodéru. Při komunikaci se spolu s daty přenáší i hodinový signál.

SPI – sériové asynchronní rozhraní – rozhraní pro asynchronní sériovou komunikaci. Hodinový signál se nepřenáší, příjmač si jej generuje sám. Je nutné zajistit dostatečně přesné generování a synchronizaci s generátorem vysílače.

ADC – AD převodník – šestikanálový 10bitový aproximační převodník z analogového signálu na číslicovou hodnotu. Umožňuje jednorázovou nebo kontinuální konverzi.

LCD – LCD displej – periferie usnadňující ovládání LCD displeje s podporou až 104 segmentů.

KBI – klávesnice – 8 vstupů umožňující vyvolat přerušení, nastavitelná úúdalost vyvolávající přerušení (hranou, úrovní).

Kapitola 4

Návrh jádra

4.1 Činnost operačního systému

Operační systém (dále OS) je skupina počítačových programů, které spravují softwarové a hardwarové zdroje počítače a zajišťují jejich vzájemnou komunikaci[1].

Mezi základní role operačního systému patří:

1. Správa prostředků - paměti, procesoru, periferií.

   Dovoluje efektivně a bezpečně sdílet dostupné prostředky mezi více procesy.

2. Vytváření prostředí pro uživatele a jejich aplikačí programy.

   Umožňuje reagovat na podněty od uživatele.

Cílem OS je zjednodušit použití počítače a současně maximálně využít všech jeho zdrojů. Operační systém samotný spotřebovává některé zdroje počítače (paměť, čas procesoru), proto je důležité aby byl OS efektivně navržen. Pro zvýšení komfortu uživatele poskytují OS uživatelské rozhraní. Může se jednat o prostou konzoli s ovládáním textovými příkazy (terminál) popř. grafické uživatelské rozhraní.

V případě vestavěných systémů se často operační systém ztotožňuje s aplikačním programem, a plní tak jedinou, předem danou úlohu.

Dále v textu je operačním systémem myšlen systém, jehož součástí je *jádro* a základní systémové knihovny a utility.

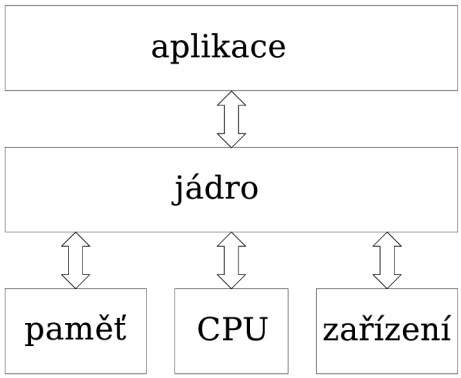
4.2 Jádro

Jádro[2] je srdcem operačního systému. Tvoří jeho nejnižší a nejzákladnější vrstvu. Jádro je prvním zaváděným programem, který běží po celou dobu běhu počítačového systému a zajišťuje základní správu prostředků pro vyšší vrstvy OS a uživatelské aplikace.

[1] Přesná definice OS neexistuje. Některé zdroje jej ztotožňují pouze s jádrem, jiné pod pojmem operační systém rozumí i přítomnost množství aplikačních programů. O to co je a co není součástí operačního systému se stále vedou spory.

[2] angl. kernel

Na obrázku 4.1 je znázorněno využití jádra jako vrstvy mezi hardwarem a softwarem počítače.



Obrázek 4.1: Pozice jádra v počítačovém systému.

Obecné jádro má na starost:

- plánování úloh/procesů a jejich vzájemnou synchronizaci a komunikaci
- správu paměti
- správu souborů

Jádra můžeme rozdělit do několika skupin, podle složitosti implementace a rozsahu poskytovnaných služeb. Jedná se o jádra:

monolitická – poskytují rozsáhlé rozhraní a velké množství služeb pro vyšší vrstvy OS. Obsahují moduly pro správu paměti, plánování, meziprocesovou komunikaci, souborové systémy, podporu síťové komunikace. Všechny služby běží současně s jádrem a jsou vzájemně provázány za účelem vyšší efektivity. Nevýhodou těchto jader je obtížnost údržby a komplikované odstraňování chyb. V případě selhání jednoho z modulů je také velká pravděpodobnost pádu celého systému.

monolitická s modulární strukturou – jsou vylepšenou koncepcí monolitických jader. Umožňují zavádění resp. odstraňování jednotlivých modulů za běhu. Tato modularita ale není dána přímo architekturou jádra. Prakticky se jedná pouze o jiný způsob práce s binární podobou jádra.

mikrojádra – definují velmi úzkou abstrakci nad hardwarem. Používají pouze sadu primitivních systémových volání, pomocí nichž spravují paměť, obstarávají víceúlohovost a meziprocesovou komunikaci. Ostatní služby jsou implementovány mimo jádro, v uživatelských programech, tzv. serverech (více viz [6]). Výhodou mikrojader je vyšší flexibilita (dynamické spouštění a zastavování služeb, implementace dalších modulů na úrovni aplikačního programu), zabezpečení (selhání služby nemusí nutně vést k selhání celého systému). Nevýhodou je vyšší režie, způsobená především přepínáním kontextu (podrobnosti v kapitole 5.6), které zpomaluje systém více než volání přímé systémové služby.

nanojádra – představují nejužší provázání s hardwarem. Zpracovávají pouze přerušení a komunikaci s jednotkou správy paměti[3]. Jsou velmi podobná mikrojádrům, liší se ale v závislostech mezi jádrem a jeho moduly při formování operačního systému. U nanojádra nemusí být nutně moduly přerušení a jednotky správy paměti přímo jeho součástí. Díky této vysoké modularitě lze lehce změnit operační systém pouhou záměnou softwaru a to i za běhu.

hybridní jádra – mikrojádra rozšířena o kód, který by mohl být implementován mimo jádro (v serveru), ale za účelem nižší režie je s mikrojádrem těsněji provázán a běží v jeho režimu.

exojádra – poskytují minimální rozhraní zaměřené převážně na bezpečné sdílení prostředků, nikoli na tvorbu abstrakcí. Jádra pouze zajišťují, že žádaný prostředek je volný a aplikační program jej může využít.

V našem případě budeme od jádra vyžadovat pouze základní funkcionalitu. Zaměříme se na přepínání procesů, tj. dosažení víceúlohovosti. Budeme vyžadovat pohodlné vkládání procesů do plánovače, základní bezpečnostní ochrany a jednoduchý mechanismus komunikace mezi procesy. Jádro nebude obsahovat správu síťové komunikace ani správu paměti.

Z výše uvedeného vyplývá, že se bude jednat o návrh a implementaci **mikrojádra**. Naši specifikaci by bylo možné zahrnout i pod nanojádro, hlavně pokud jde o modularitu a možnost změny modulů za běhu. Bylo by totiž možné (díky možnostem vybraného mikrokontroléru) v určitém okamžiku přepnout obsluhu systému na jiné jádro. Naskytla by se tak možnost použít např. jiný plánovač (viz 4.4), jiný přístup paměťové ochrany (viz 4.8) apod. Nicméně pro vyšší názornost nebudeme dále tuto možnost uvažovat a použijeme pouze jediné jádro.

4.3 Víceúlohovost

Mezi jeden z charakteristických rysů operačního systému patří zajištění vykonávání více úloh současně. Nejedná se však o jejich paralelní provádění[4]. Víceúlohovost je metoda, díky které je umožněno více úlohám (procesům) sdílet zdroje systému, jako např. čas mikroprocesoru. V případě jednoprocesorového počítače může být v daný okamžik věnován strojový čas pouze jedné úloze. Rozhodnutí o tom, která úloha dostane přidělen procesor, je zajištěno *plánovačem*. Přepíná–li plánovač jednotlivé úlohy dostatečně často, vzniká iluze paralelního provádění úloh.

4.4 Plánovač

Plánovač má za úkol rozhodnout, které z požadovaných úloh bude přidělen mikroprocesor a zdroje systému. Nelze však vždy jednoznačně říct, která z úloh by to měla být. Objevuje se zde mnoho různých problémů (např. nelze dopředu určit jak dlouho bude úloha trvat a zda by nebylo lepší nejdříve spustit proces, který je časově méně náročný. Dále mohou např.

[3] angl. MMU – Memory Management Unit

[4] Uvažujeme pouze jeden mikroprocesor.

vznikat závislosti mezi jednotlivými procesy, tj. jeden proces potřebuje ke svému úspěšnému provedení data od jiného procesu apod.). V současné době existuje řada algoritmů a strategií, které tyto problémy více či méně úspěšně řeší (míra úspěšnosti a použitelnosti závisí na okolnostech).

Přehled některých plánovacích strategií a jejich základní princip:

1. **FIFO**[5] – princip analogický frontě. První příchozí požadavek je obsloužen jako první, další příchozí požadavek čeká, dokud není první vyřízen.

2. **Shortest–First**[6] – čekající proces s nejkratší dobu dosavadního provádění získává čas procesoru. Metoda je jednoduchá a má velkou propustnost (množství dokončených procesů za určitou dobu). Nevýhodou je její náchylnost k *vyhladovění*[7] procesu, který je v systému dlouhou dobu (je časově náročnější na dokončení), z důvodu neustálého přibývání nových, krátce trvajících procesů.

3. **Round–robin** – jeden z jednodušších algoritmů přidělující postupně každému z procesů pevně dané množství času (tzv. *časové kvantum*) bez ohledu na priority procesů. Tento přístup je jednoduchý na implementaci a eliminuje problém vyhladovění.

4. **Priority queue**[8] – procesy jsou obsluhovány (jsou jim přidělovány požadované zdroje a čas procesoru) podle *priority*. Čím vyšší prioritu proces má, tím dříve se dostane na řadu. Prioritu je možné dynamicky přepočítávat a pořadí procesů tak měnit podle aktuální potřeby. Metoda je náchylná na vyhladovění.

5. **Lottery**[9] – každému procesu je přiřazeno několik náhodných čísel (lístků loterie), podle nichž je pak volán. Čím více má proces lístků, tím je pravděpodobnost jeho vykonávání vyšší.

Strategie plánovače lze rozdělit rovněž z hlediska odevzdání procesoru samotným procesem:

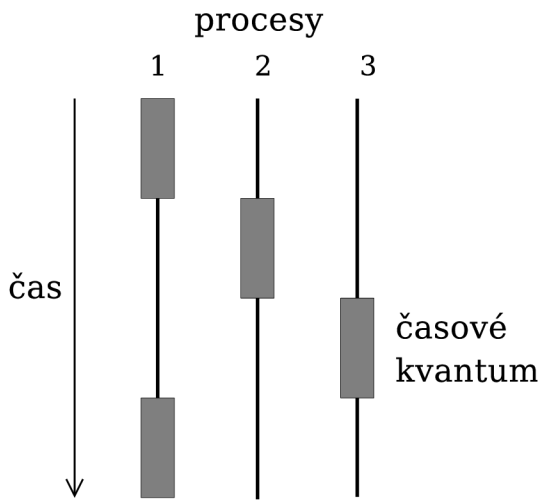
- Nepreemptivní – procesor může být procesu odebrán, prokud je zablokován při vstupně–výstupní operaci nebo pokud se proces procesoru sám vzdá.

- Preemptivní – procesu může být procesor odebrán, vyprší–li časové kvantum procesu přidělené, nebo pokud přijde požadavek na obsloužení procesu s vyšší prioritou.

Pro tuto práci byla zvolena strategie **Round–Robin**. Hlavním důvodem je její jednoduchost a nenáročnost na zdroje pro vlastní exekutivu plánovače. Jedná se o strategii preemptivní, protože procesy budou přepínány po vypršení časového kvanta.

[5] First In First Out –První dovnitř, první ven
[6] Doslovný překlad je „nejkratší–první“.
[7] Stav, ve kterém proces nemá přístup k potřebným prostředkům, z důvodu jejich blokování jiným procesem.
[8] Prioritní řada
[9] Loterie

Na obrázku 4.2 je znázorněno přepínání procesů při použití strategie Round–Robin.



Obrázek 4.2: Princip algoritmu Round–Robin.

4.5 Zajištění víceúlohovosti

Víceúlohovost v počítačovém systému s jedním mikroprocesorem lze zajistit několika způsoby. Je na uživateli, aby zvolil nejvhodnější variantu vzhledem k potřebám svého projektu, složitosti a rozsahu implementace.

Základní přístupy k řešení problému „současného“ běhu více úloh jsou následující:

1. **cyklicke zpracování**[10] – tento přístup nevyužívá přerušení ani složité algoritmy. Jedná se o několik metodik založených na nekonečných smyčkách (`for(;;) {...}`). Uvnitř smyčky jsou postupně uvedeny úlohy v pořadí, ve kterém se mají provádět. Existují i varianty, které na základě různých stavů úloh (tzv. *flagů*) mohou měnit priority těchto úloh a tím upravovat pořadí jejich spouštění. Mezi tyto metodiky patří např. vyzývací smyčka[11], cyklické provádění nebo stavově řízený kód. Výhodou tohoto přístupu je snadná analýza systému. Nevýhodou je určité zpoždění mezi přijatou žádostí o zpracování události a jejím vlastním vyřízením. Více se lze dočíst v [4].

2. **využití přerušovacího podsystému** – umožňuje realizaci plánovače pomocí hardwarového nebo softwarového přerušení. Přerušení určující okamžik přepnutí je voláno např. časovačem, či jiným zdrojem hodinového signálu. Úloha která je momentálně prováděna se přeruší, dojde k přepnutí kontextu (viz 5.6) a podle plánovače je určena úloha následující. Výhodou těchto systémů je rychlá odezva na příchozí požadavek a přehlednost kódu. Nevýhodou pak plýtvání časem procesoru v nekonečné smyčce, která tvoří hlavní tělo systému, nebo komplikace při poskytování složitějších služeb (správa souborů, sítí apod.).

[10] Označováno také jako pseudojádro.
[11] angl. polling

3. **systémy pracující v popředí/pozadí**[12] – jsou založeny na využití přerušení. V systému jsou úlohy rozděleny do dvou skupin. Tzv. *popředí systému* je tvořeno úlohami s vysokou prioritou, které se provádějí při přerušení. *Pozadí systému* je tvořeno úlohami s nižší prioritou, které by neměly provádět časově náročné operace, ale spíše by např. mohly pracovat s pomalými zařízeními nebo provádět testování částí systému. V momentě, kdy dojde k přerušení, vyvolá se některá z úloh s vysokou prioritou, která přeruší provádění úloh na pozadí do doby, než je zpracována. Tyto systémy mají krátké doby odezvy na události, jsou však náchylné k chybám způsobeným změnami časů provádění kódu či selhání hardware. Také je nutné předem znát počet úloh v popředí pro efektivnější implementaci.

4. **TCB model**[13] – je vhodný použít v systémech, kde se předpokládá dynamické přidávání a odebírání úloh. Každé úloze je přidělena datová struktura (tzv. *blok řízení úlohy*), obsahující identifikaci úlohy, její stav, registry, prioritu a další informace. Jádro (běžící jako úloha s nejvyšší prioritou) si pak uchovává seznamy připravených a blokovaných úloh. Každé přerušení a systémový požadavek od běžících úloh vyvolá úlohu typu jádro. Jádro zkontroluje zda je nějaká úloha v seznamu připravených úloh. Pokud ano, přesune se TCB aktuálně běžící úlohy na konec seznamu připravených úloh a ze seznamu úloh vezme první připravenou úlohu, kterou spustí. Výhodou tohoto systému je velká flexibilita a dynamičnost správy úloh. Nevýhodou pak je velká režie spojená se správou a uchováváním informací o úlohách.

Každá z uvedených technik přináší výhody i nevýhody. Podrobnější analýze a návrhu řešení pro naše jádro se věnují následující kapitoly.

4.6 Možnosti architektury HC08

U vestavěných systémů je kladen velký důraz na správné využití zdrojů poskytovaných systémem. Před vlastním návrhem jádra operačního systému je nutné důkladně prostudovat cílovou platformu, především z pohledu softwarového. Máme nyní potřebné infomace k určení strategie pro vytvoření jádra tak, aby správa prostředků mikrokontroléru byla efektivní a dostatečně účinná.

Před návrhem jádra se blíže zaměříme na následujícími témata.

4.6.1 Využití časovače

Vzhledem k tomu, že předmětem návrhu a implementace je víceúlohový operační systém, bude nutné zajistit pseudoparalelní běh více úloh. Pro přepínání úloh zvolíme mechanismus využívající přerušovacího podsystému (bod 2, kapitola 4.5).

Časovač umístěný na čipu 68HC908LJ12 se skládá ze dvou modulů, TIM1 a TIM2, z nichž každý má dva kanály.

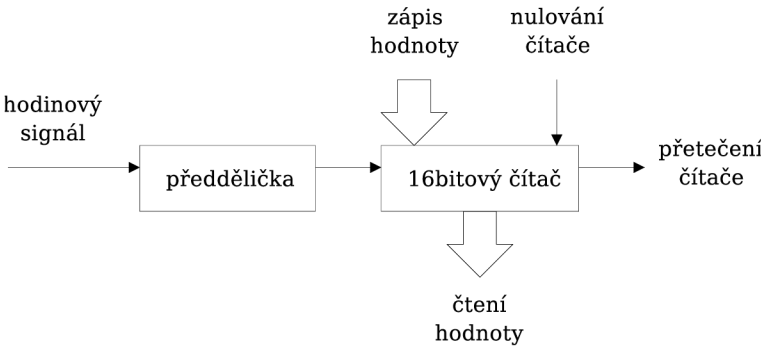
[12] angl. foreground/background systems – FBS
[13] Task Control Block

Modul časovače lze využít jako:

- jednotku záchytu hrany – v případě že od systému očekáváme reakci na změnu určitého číslicového signálu
- výstupní komparátor – v případě, že chceme generovat impulzy s proměnnou střídou a periodou. Jakmile dosáhne čítač časovače přednastavené hodnoty, může dojít na pinu mikrokontroléru k nastavení, vynulování nebo změně napěťové úrovně.
- prostý čítač – kdy můžeme reagovat na dosažení určité hodnoty čítače

Od našeho jádra vyžadujeme, aby v určitých intervalech umožnilo přepnutí prováděné úlohy. Využijeme tedy časovač jako čítač (viz obr 4.3).

Jelikož jsou k dispozici dva časovače, vybereme pro účely jádra pouze jeden z nich. Protože by mohla nastat situace, kdy jedna z úloh využívá druhý časovač, vybereme pro jádro modul s vyšší prioritou obsluhy přerušení. To proto, abychom upřednostnili vykonání operace jádra před vykonáním požadavku úlohy. Využívat tedy budeme časovač s označením **TIM2** a pro uživatelské úlohy ponecháme k dispozici **TIM1**.



Obrázek 4.3: Časovač s předděličkou.

Pro správnou funkci časovač nastavíme vhodně předděličku, abychom se řádově dostali do oblasti přepínání v řádech Hz (pro větší názornost). Poté bude nutné nakonfigurovat správně vlastní časovač. Půjde především o nastavení modula, tj. hodnoty po jejímž dosažení bude čítač časovače vynulován. Hodnotu bude časovač zvyšovat počítáním impulzů z předděličky, až dokud se nedostane na hodnotu danou modulem. V tento okamžik také dojde k vygenerování přerušení a vyvolání obslužné rutiny.

4.6.2 Zásobník

Pro ukládání kontextu úloh budeme využívat zásobník. Proto bude nezbytně nutné znát jeho správnou funkci.

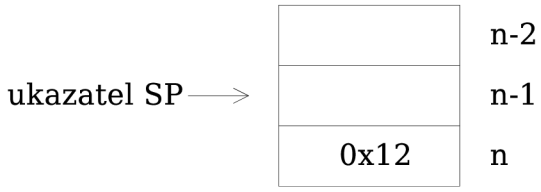
Zásobník je datovou strukturou označovanou jako *LIFO*[14]. Znamená to, že hodnota naposled uložená na zásobník bude při odebrání ze zásobníku vrácena jako první.

[14] Last In First Out

Zásobník se používá převážně pro:

- ukládání resp. obnovování návratových adres při odskoku do resp. návratu z podprogramu
- ukládání a obnovování programovacího modelu při přerušení
- předávání proměnných mezi podprogramy

Adresace zásobníku je umožněna prostřednictvím SP registru, který ukazuje na další volné paměťové místo, kam se bude případně ukládat další hodnota. Vztah mezi obsahem zásobníku a registrem SP je znázorněn na obrázku 4.4.



Obrázek 4.4: Vztah mezi ukazatelem na zásobník a pamětí zásobníku.

Pokud je zásobník prázdný, obsahuje SP adresu místa, kam lze uložit první byte. Po vložení hodnoty `0x12` na zásobník (např. pomocí instrukce `PSHA`), ukazuje SP na adresu o 1 nižší než je adresa vložené hodnoty. Při výběru bytu ze zásobníku (např. instrukcí `PULA`) se SP o 1 zvýší a ukazuje opět na první volné místo zásobníku.

Zásobník u rodiny HC08 je dynamicky realokovatelný. To znamená, že jeho umístění můžeme ovlivnit dynamicky, za běhu programu, což také v jádru uplatníme.

Zásobník je umístěn v paměti RAM. U námi zvoleného typu mikrokontroléru je to v oblasti paměti od `0x0060` do `0x025f`. Není mu ale vyhrazena celá tato oblast. Ihned po resetu mikrokontroléru je zásobník umístěn na adresu `0x00ff`. Pro dosažení vyšší efektivity kódu se doporučuje umístit do oblasti od `0x0060` do `0x00ff` globální proměnné, z důvodu rychlejšího přístupu k nim. Od adresy `0x0100` do `0x025f` pak zůstává pro zásobník a případné další proměnné 352 bytů.

Při skoku do podprogramu prostřednictvím instrukcí `JSR` nebo `BSR`, se na zásobník ukládají 2 byty, obsahující adresu instrukce, která bude provedena po návratu z podprogramu. Ze zásobníku budou tato data opět odebrána použitím instrukce `RTS`. Při vyvolání přerušení se na zásobník ukládá 5 bytů programovacího modelu. Konkrétně jde o registry PC, X, A a CCR (stavový registr). Ze zásobníku je programovací model odebrán při vykonání instrukce `RTI`. Registr H není automaticky ukládán z důvodu zachování kompatibility s mikroprocesory M6805[15].

Registr SP je možné použít i jako index registr pro indexové adresování. Více informací lze najít např. v [1].

Při práci se zásobníkem je třeba dávat pozor, aby nedošlo k přepsání jiných dat.

[15] V případě použití registru H v obsluze přerušení je nutné jej na zásobník uložit resp. z něj obnovit pomocí instrukce `PSHH` resp. `PULH`.

4.7 Inicializace úloh jádra

U obecného operačního systému platí, že jeho jádro je prvním zaváděným programem. V našem případě bude toto pravidlo dodrženo. Před vlastním spuštěním jádra (tj. před předáním řízení jádru) ale musíme specifikovat, které úlohy bude jádro spravovat. Je proto nutné navrhnout mechanismus, kterým tyto úlohy jednoduše, efektivně a jednoznačně určíme a zařadíme do plánovače.

Je třeba si uvědomit, že musíme jádru sdělit na které adrese začíná vlastní kód dané úlohy. Tato informace se ale po prvním spuštění stává zbytečnou. Jádro se totiž v následujícím spuštění úlohy nebude vracet na její první instrukci. Bude pokračovat ve vykonávání kódu úlohy od adresy, na které byl v předchozím běhu proces přerušen. Potřebujeme tedy navíc u každé úlohy ukládat informaci o adrese, od které má pokračovat při dalším spuštění.

Nabízí se dvě možnosti, jak adresu počátku úlohy a místa dalšího vstupu uchovávat:

1. pamatovat si obě adresy
2. adresu počátku přepisovat adresou dalšího vstupu do procesu

První možnost je jistě čistější a přehlednější. Vyplácí se v případě že součástí operačního systému bude např. logovací mechanismus, monitorování běhu úloh, správce procesů apod. Pokud ale bude adresa počátku procesu využita pouze jednou, je její další přítomnost zbytečná. V systémech s omezeným množstvím paměti je pak vítáno tuto informaci z paměti uvolnit, nebo ji využít pro jiné účely, což naznačuje druhá možnost. Po prvním volání procesu se adresa začátku procesu stává zbytečnou. Dojde–li k odebrání procesoru dané úloze, uložíme na adresu první instrukce informaci o místě návratu do přerušené úlohy.

Při vkládání úlohy do systému je nutné úloze zaručit, že budou dostupné prostředky, nezbytné pro její běh. Vzhledem k rozsahu implementace jádra se v našem případě bude jednat pouze o zajištění paměti pro úlohu. Bohužel tento údaj nelze určit bez předběžné znalosti dané úlohy. Musíme tedy použít alternativní přístup, tj. určit množství paměti pro každý proces.

K tomuto problému můžeme přistupovat automatizovaně – určili bychom konstantní velikost vyhrazené paměti pro každý proces. Uživatel by se tak nemusel starat o to, kolik paměti jeho proces potřebuje ke svému běhu. Tato možnost je pohodlná, ale nepříliš efektivní v případě, že používáme různě paměťově náročné úlohy. Pak by se tato hodnota musela stanovovat s ohledem na paměťově nejvíce náročnou úlohu, což by velo k výraznému snížení maximálního počtu procesů.

Řešením tedy v našem případě bude explicitní určení paměti pro každý proces jednotlivě. Při vkládání úlohy bude muset uživatel uvést název této úlohy a její paměťovou náročnost. Určení potřebné paměti nebude triviální. Uživatel bude muset znát princip úlohy, případně provést simulaci pro zjištění minimální paměťové zátěže. Přínosem ale bude dosažení vyšší efektivity využití paměti mikrokontroléru.

U obou možností přidělení paměti procesu může dojít k situaci, kdy procesu paměť dojde. Pokusí se pak zapsat svá data mimo jemu vyhrazený prostor. Tuto situaci rozebírá do podrobna sekce 4.8.

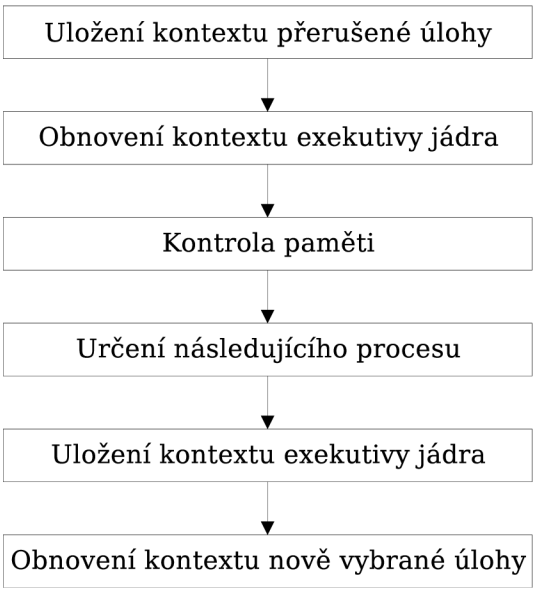
4.8 Běh jádra

Jakmile máme všechny úlohy vložené, můžeme dokončit inicializaci jádra a předat mu řízení systému.

Se spouštěním jádra je spojena ještě jedna povinnost. Je totiž nutné spustit některou z úloh, aby byla zajištěna univerzálnost přepínání úloh a jádro nemuselo obsahovat řešení pro speciální případ, kdy by nedocházelo k počátečnímu přepnutí kontextu. Toto ošetření by znamenalo snížení výkonnosti jádra. Pokud bychom nespustili některou z úloh před zavedením jádra, došlo by k tomu, že při prvním volání jádra by nebyla známá přerušená úloha. Za prvotní úlohu bychom mohli považovat hlavní vstupní bod programu, nicméně by bylo nezbytné přidělit hlavnímu programu část paměti. Ta by ale byla využita pouze jednorázově. Proto zvolíme prvotní spuštění některé z úloh. Bude tak zachována univerzálnost jádra a minimalizovány prostředky nutné pro inicializaci a spouštění jádra.

Jádro máme tedy spuštěné a běží nám i počáteční úloha. Nyní je čas procesoru plně věnován této úloze. Po uplynutí nastaveného časového kvanta dojde k přerušení od časovače. Systémem SIM je vybrána a zavolána obslužná rutina pro přerušení při přetečení časovače.

Princip přepínání úloh je znázorněn na obrázku 4.5.



Obrázek 4.5: Princip přepínání úloh (emulace víceúlohovosti).

V obslužné rutině provedeme přepnutí kontextu běžící úlohy a módu OS. Tímto se dostaneme z režimu uživatelského (běh jednotlivých úloh) do režimu jádra (správa úloh, plánovač apod.). V tomto místě provedeme kontrolu paměti, kterou přerušený proces využíval. V případě že došlo k přepsání dat jiného procesu, zastavíme činnost jádra a upozorníme na tuto skutečnost uživatele. Pokud bychom tuto kontrolu neprovedli, mohla by nastat situace, kdy by se po skončení obslužné rutiny nevrátilo vykonávání procesu do správného místa. Dále rozhodneme o úloze, které bude v příštím okamžiku přidělen výpočetní čas. Zde přebírá exekutivu plánovač, v našem případě založený na metodě Round–robin (viz 4.4).

Zbývá nám přepnout se zpět do uživatelského režimu, tj. obnovit kontext nově vybrané úlohy. Díky informaci o adrese na které bylo vykonávání úlohy přerušeno je zajištěno, že úloha bude pokračovat na správném místě.

Čas procesoru tedy opět využívá uživatelská úloha, dokud nedojde k dalšímu přerušení od časovače.

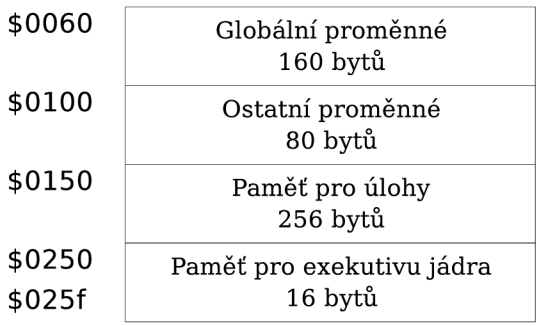
4.9 Omezení daná architekturou

V oblasti vestavěných systémů je návrh jádra komplikovanější než v ostatních počítačových systémech. Jsme nuceni přizpůsobit naše požadavky podle dostupných kapacit paměti. Musíme vzít také v úvahu frekvenci mikroprocesoru. Dále bychom měli dbát na bezpečnou práci s periferiemi mikrokontroléru, aby nedocházelo k neočekávanému a nebezpečnému chování systému.

Pro naše účely poskytuje mikrokontrolér 68HC908LJ12 dostatek zdrojů. Nejdůležitějším je z našeho hlediska velikost paměti RAM.

Pro vlastní použití máme k dispozici 512 bytů RAM paměti, od adresy `0x0060` do `0x025f` včetně. Do tohoto prostoru ale spadá i místo pro zásobník. Musíme tedy zvážit umístění jednotlivých typů proměnných a zásobníku.

Paměť pro naše jádro uspořádáme podle obrázku 4.6.



Obrázek 4.6: Organizace paměti RAM pro efektivní využití jádrem.

Do oblasti globálních proměnných je vhodné ukládat data, která jsou využívána nejčastěji. Jedná se o proměnné jádra, o data sdílená více procesy apod.

Mezi ostatní proměnné lze zařadit globální proměnné jednotlivých úloh.

Paměti pro procesy je k dispozici nejvíce, aby bylo umožněno spravovat co největší množství procesů. Tuto paměť má pod správou jádro.

Poslední část je věnována samotnému jádru, které si do ní ukládá data během výpočtu paměťové ochrany a plánovače.

Kapitola 5

Implementace

5.1 Vývojové prostředky

Vývoj aplikací pro vestavěné systémy je značně usnadněn existencí vývojových prostředí.

Protože se u jednotlivých výrobců mikrokontrolérů jednotlivé rodiny od sebe odlišují, ať už po stránce hardwarové nebo softwarové, lze narazit na množství vývojových prostředí, stavěných specifickým skupinám mikrokontrolérů na míru. Aplikace pro čipy od výrobce Atmel tak můžeme vyvýjet v programu AVR Studio, Microchip doporučuje na svých stránkách vývojové prostředí MPLAB atd. I Motorola má pro rodinu 8bitových kontrolérů své typické prostředí, usnadňující a urychlující vývoj aplikací. Je jím *Freescale CodeWarrior*, v současné době ve verzi 5.1. V tomto prostředí byla vyvýjena i programová část této práce.

CodeWarrior poskytuje uživateli množství nástrojů pro větší komfort, z nichž uvedu pouze základní z nich, které napomohly při vývoji zadaného jádra:

- editor zdrojového kódu se zvýrazněním syntaxe
- simulátor pro ladění aplikace bez nutnosti přítomnosti cílového hardwaru
- obvodový emulátor, umožňující sledování stavu mikrokontroléru za běhu
- možnost práce s projekty
- propracovaná nápověda

CodeWarrior nabízí možnost vývoje aplikace v assembleru nebo vyšším programovacím jazyce – C a C++. Pro implementaci jádra jsem zvolil jazyk C. Kód je pak přehlednější a čitelnější, s možností jednodušších úprav a opravy chyb. Nevýhodou je snížení efektivity, protože píšeme–li aplikaci přímo v assembleru, máme přehled nad každým bytem a každou provedenou instrukcí. I přesto je mezipřeklad z jazyka C do assembleru dostatečně efektivní.

Více informací o vývojovém prostředí lze najít na www stránkách [2].

Pro snazší orientaci jsem celý zdrojový kód rozdělil do dvou specifických celků, a sice:

1. **modul jádra** – obsahuje proměnné a funkce, nezbytné pro správné fungování jádra. Tvoří jej soubory `kernel.h` a `kernel.c`.
2. **uživatelský modul** – definující uživatelské funkce a volající funkce jádra. Jedná se o soubor `main.c`.

5.2 Důležité konstanty

V souboru `kernel.h` jsou uvedeny veškeré konstanty a proměnné, které jádro ke svému běhu potřebuje. Jejich úplný výpis uvádět nebudu, zmíním se jen o těch, které jsou pro uživatele klíčové.

Při návrhu a implementaci jsem se snažil o zajištění přenositelnosti mezi jednotlivými typy mikrokontrolérů rodiny HC08. Pro každý typ mikrokontroléru je proto zásadní nastavení následujících konstant:

```
#define MAX_ULOH 5
#define ZASOBNIK_DOLNI_MEZ 0x0150
#define ZASOBNIK_HORNI_MEZ 0x024f
#define ZASOBNIK_OBSLUHA #$025f
#define BUS_CLOCK 2457600
```

Konstanta `MAX_ULOH` udává maximální počet úloh, které lze vložit do plánovače. Při určování této hodnoty je nutné přihlédnout k velikosti volné paměti mikrokontroléru.

Hodnoty `ZASOBNIK_DOLNI_MEZ` a `ZASOBNIK_HORNI_MEZ` ohraničují paměť, kterou bude plánovač přidělovat jednotlivým úlohám. Velikost této paměti by měla být volena s ohledem na další části paměti RAM, popsané v kapitole 4.9. Také je vhodné uvažovat paměťovou náročnost jednotlivých úloh.

Protože vlastní jádro potřebuje pro své výkonné funkce také určitou část paměti, je vhodné tento blok specifikovat a umístit mimo ostatní logické úseky. Já jsem zvolil umístění na konec RAM paměti. Pozice posledního bytu této paměti je dána hodnotou `ZASOBNIK_OBSLUHA`. Velikost této paměti by měla být pro implementované jádro minimálně 2 byty.

Poslední konstanta `BUS_CLOCK` udává frekvenci sběrnice mikrokontroléru. Tato hodnota je využita pro převod hodnoty časového kvanta na modulo čítače. Může se lišit podle použité metody generování hodinového signálu pro mikrokontrolér (viz kapitola 3.7). V našem případě jsme využili vestavěný obvod krystalového oscilátoru s fázovým závěsem. Frekvence sběrnice se získá dělením frekvence jádra mikrokontroléru (v obvodu CGM a v obvodu SIM je dělena celkem 4).

5.3 Inicializace jádra

Rutina inicializace jádra musí být první volanou funkcí jádra. Ovlivňuje nastavení plánovače a přepínače úloh. Uživatel při jejím volání zadává hodnotu časového kvanta.

```
void init(unsigned int casove_kvantum_ms) {
```

V první fázi inicializace je třeba nastavit časovač, zajišťující volání rutiny pro přepínání úloh. Z důvodů uvedených výše (kapitola 4.6.1) jsem zvolil pro tento úkol časovač TIM2.

U časovače je zapotřebí povolit přerušení po přetečení čítače, aby po uplynutí časového kvanta došlo k zavolání přepínače úloh. Dále jsem nastavil předděličku 1:64, abychom se dostali do oblasti frekvencí, které umožňují názorné otestování jádra. Z frekvence sběrnice f [Hz] a z časového kvanta k [ms] je odvozena hodnota čítače, po jejímž dosažení dojde k přerušení od časovače a zavolání přepínání úloh.

Vzorec pro výpočet modula je následující:

$$modulo = \frac{f}{64 \cdot \left(\frac{1000}{k}\right)}$$

Rozsah hodnot k je povolen od 1ms do 1000ms. Dolní hranice poskytuje pro naše účely dostatečnou efektivitu, horní hranice slouží spíše k účelům demonstrativním. Pokud je zadána hodnota mimo tento rozsah, je signalizována chyba[1].

```
unsigned int modulo;

if ((casove_kvantum_ms < 1) || (casove_kvantum_ms > 1000)) {
  chyba();
} // if

T2SC = 0x76;

modulo = BUS_CLOCK / 64 / (1000 / casove_kvantum_ms);

T2MODH = modulo / 256;
T2MODL = modulo % 256;
```

K nastavení časovače jsem použil následující registry:

- **T2SC** – Stavový a kontrolní registr pro nastavení předděličky a povolení přerušení
- **T2MODH** a **T2MODL** – k nastavení modula časovače na základě časového kvanta

Dalším krokem při inicializaci je odstranění kontrolního mechanismu COP (viz kapitola 3.7 nebo literatura [3]).

```
CONFIG1 = 0x01;
```

Pokud bychom nechali tento systém aktivní, bylo by nutné zahrnout volání jeho instrukcí do přepínače procesů nebo do uživatelských úloh. Obě dvě varianty jsem zavrhnul. V případě umístění do přepínače procesů by nebylo možné použít vyšší hodnoty pro časové kvantum. V případě umístění v uživatelských úlohách bychom zase znepříjemnili práci uživateli, protože by byl nucen při implementaci zahrnovat volání COP instrukcí do svého kódu.

Mechanismus COP jsem tedy nahradil kontrolou integrity paměti v přepínači procesů, která spolu s pravidelným voláním jádra watchdog dostatečně nahrazuje. Více podrobností je uvedeno v kapitole 5.6.

Posledním krokem v inicializaci jádra je nastavení údajů nezbytných pro přidávání úloh. Jedná se především o nastavení proměnných[2] `__zasobnik_volny` (udávající velikost doposud volné paměti pro úlohy) a `__zasobnik_vrchol` (specifikující umístění zásobníku pro úlohy). Tyto dvě proměnné jsou odvozeny z konstant, specifických pro zvolenou platformu. Při použití jiného typu mikrokontroléru je může uživatel přizpůsobit danému typu, podle kapitoly 4.9.

[1] Chybou budeme dále v textu označovat volání rutiny `chyba()`. Chyba jádra je indikována blikáním červené LED diody na vývojové desce.

[2] Veškeré proměnné jádra používají jako prefix dvojí podtržítko.

```
  __pocet_uloh = 0;
  __uloha_spustena = 0;
  __zasobnik_volny = ZASOBNIK_HORNI_MEZ + 1 - ZASOBNIK_DOLNI_MEZ;
  __zasobnik_vrchol = ZASOBNIK_HORNI_MEZ;

} // init
```

Konfigurace jádra je dokončena, můžeme přistoupit ke vkládání uživatelských úloh.

5.4 Vkládání jednotlivých úloh

Po inicializaci jádra lze vkládat do plánovače jednotlivé úlohy. Uživatel musí uvést jméno vkládané úlohy a počet bytů paměti, které chce úloze rezervovat k využití.

```
int pridejUlohu(void (*uloha)(void),
  unsigned char velikost_zasobniku) {
```

Na začátku rutiny pro přidání úlohy provedeme 2 testy. Prvním zajistíme, že se do plánovače vloží jen platná úloha, která má své umístění v programové paměti. Druhý test kontroluje, jestli je ještě v plánovači místo pro novou úlohu. Vzhledem k tomu, že není uvažována správa paměti, je maximální počet úloh omezený.

```
if (uloha == NULL) {
  return FALSE;
} // if

if (__pocet_uloh == MAX_ULOH) {
  return FALSE;
} // if
```

Každá jednotlivá úloha v plánovači je identifikována datovým typem `__uloha`. Obsahuje dvě složky – `zasobnik_ukazatel` a `zasobnik_vrchol`. Záměrně zde není uvedena adresa na počátek úlohy, resp. adresa místa kde byla úloha přerušena (jde o tentýž identifikátor, jak jsem zmínil v kapitole 4.7). Využil jsem totiž principu ukládání programovacího modelu na zásobník v případě volání obslužné rutiny přerušení.

Po spuštění jádra se volá první vložená úloha (viz důvody uvedené v kapitole 4.8). Uplyne–li doba daná časovým kvantem, volá se funkce pro přepínání úloh. Přitom se na zásobník ukládá mimo jiné i ukazatel na adresu další instrukce přerušené úlohy. To znamená, že další vložená úloha se nebude volat přímo, ale tak, jako by již byla dříve přerušena. Na konci přerušení se obnoví její kontext (programovací model) a vykonávání programu se přesune k této vybrané úloze.

Je tedy nutné přizpůsobit uvedené situaci obsah zásobníku každé vkládané úlohy. Před prvním spuštěním úlohy jí na zásobník připravíme data tak, aby se po ukončení přerušení běh programu dostal na tuto úlohu.

Při zajišťování paměti pro úlohu tedy musíme připočítat paměť nezbytnou pro uložení programovacího modelu. V našem případě se jedná o 6 bytů (PC, X, A, CCR, H[3]). Také je třeba uvažovat režii vzniklou při přepínání kontextu (popsáno dále), kterou tvoří 1 byte. Celkem je tedy nutné k paměti každé úlohy rezervovat navíc 7 bytů.

[3]Díky CodeWarrioru se registr H ukládá automaticky, takže programovací model je spravován kompletně. Instrukce pracující s programovacím modelem ale používáme pořád stejně, tedy bereme v úvahu 5 bytů.

```
velikost_zasobniku += 7;
```

Známe celkovou velikost paměti, nezbytnou pro správný chod úlohy. Pokud je dostatek volné paměti, můžeme přejít k inicializaci zásobníku úlohy a jejímu vložení do plánovače.

Seznam úloh plánovače je reprezentován polem `__seznam_uloh` konstantní velikosti, tvořeného datovým typem `__uloha`. Do tohoto pole jsou úlohy vkládány a z tohoto pole je pak plánovač postupně vybírá.

Vložením úlohy do plánovače rozumíme nastavení složek `zasobnik_ukazatel` a `zasobnik_vrchol`. První složka odpovída adrese, od níž bude uložen programovací model. Druhá proměnná ukazuje na poslední byte paměti přidělené dané úloze. Tato proměnná se využívá pro test integrity paměti při přepínání úloh. Pro plánovač samotný význam nemá.

```
if (__zasobnik_volny - velikost_zasobniku < 0) {
  return FALSE;
} // if

__seznam_uloh[__pocet_uloh].zasobnik_ukazatel =
  __zasobnik_vrchol - 5;
__seznam_uloh[__pocet_uloh].zasobnik_vrchol = __zasobnik_vrchol;
```

V případě vkládání první úlohy si pro pozdější spuštění jádra uložíme informace o této úloze – její počáteční adresu a ukazatel na vrchol jejího zásobníku.

```
if (__pocet_uloh == 0) {
  __pocatecni_uloha_adresa = uloha;
  __pocatecni_uloha_sp = __zasobnik_vrchol;
} // if
```

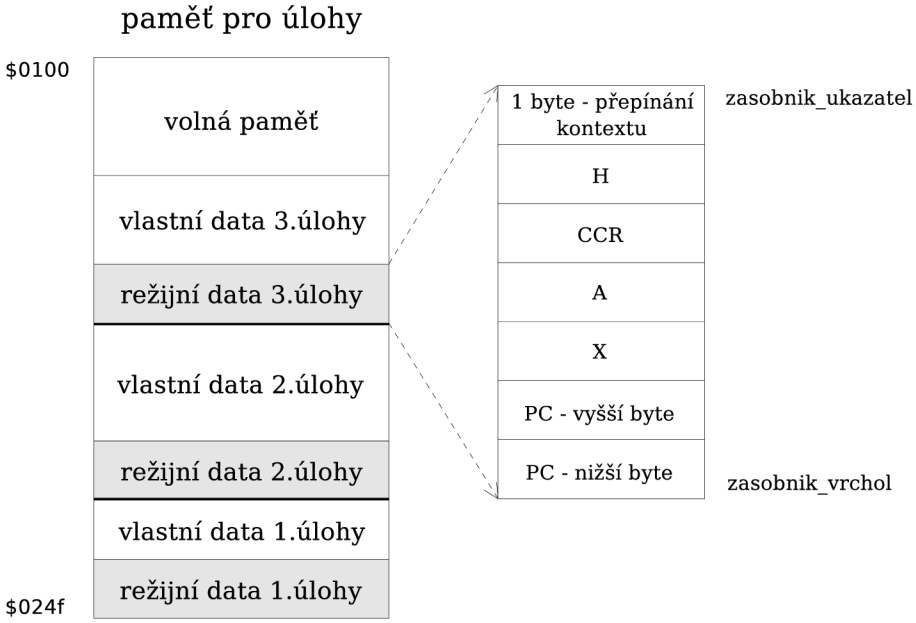
Po vložení úlohy do plánovače musíme připravit zásobník úlohy, jak jsem uvedl výše. Protože přistupujeme přímo k registrům programovacího modelu, vložíme do kódu krátký úsek assembleru, který nám tyto registry zpřístupní.

```
asm {
  ldhx __zasobnik_vrchol
  lda uloha:1
  sta ,x
  decx
  lda uloha:0
  sta ,x
  decx
  clra
  sta ,x
  decx
  sta ,x
  decx
  lda #$60
  sta ,x
  lda #$00
  decx
  sta ,x
} // asm
```

Ukládání registrů probíhá od posledního bytu paměti přidělené úloze v pořadí, které vyhovuje instrukci `RTI` :

- Registr **PC** nastavíme na adresu počátku úlohy (Při prvním vstupu do úlohy vyžadujeme provádění od začátku).
- **X** bude stejně jako A nulový.
- Stavový registr **CCR** nastavíme na výchozí hodnoty, uvedené v katalogovém listu [1], s vyjímkou povolení globálního přerušení. V případě, že by nebylo povoleno, by se po skoku do této úlohy zablokovalo volání jádra a již by nedošlo k dalšímu přepnutí úloh.
- Registr **H** nastavíme nulový.

Organizace paměti pro úlohy, včetně umístění režijních informací je na obrázku 5.1.



Obrázek 5.1: Organizace paměti úloh a uložení režijních dat.

Nakonec zvýšíme počet úloh v plánovači a upravíme globální proměnné jádra, udávající dostupnou volnou paměť a ukazatel na další byte paměti, který bude moct využít další vkládaná úloha. Dostupnou volnou paměť snížíme o celkovou velikost paměti přidělené úloze. Ukazatel přesuneme na novou pozici, od které se bude přidělovat paměť další úloze.

```
  __pocet_uloh++;
  __zasobnik_volny -= velikost_zasobniku;
  __zasobnik_vrchol -= velikost_zasobniku;

} // pridejUlohu
```

5.5 Spuštění jádra

Po inicializaci jádra a zavedení úloh nezbývá než jádro spustit a tím mu předat kontrolu nad systémem, prostřednictvím patřičné funkce.

```
void start() {
```

Samotnému spuštění předchází inspekce stavu plánovače. Případy které mohou nastat při volání rutiny pro spuštění jádra jsou následující:

Byly vloženy alespoň 2 úlohy. Pro tento stav bylo jádro navrženo a jen za této situace bude spuštěno.

```
if (__pocet_uloh > 1) {
```

Běh jádra je založen na přerušení od časovače. Proto je prvním nutným krokem při spouštění jádra povolení globálního přerušení.

```
EnableInterrupts;
```

Protože každá úloha má vymezený svůj paměťový prostor, musíme i prvotní spouštěné úloze zpřístupnit její paměť. To provedeme nastavením ukazatele na zásobník, podle údajů který jsme si při přidávání uložili. Bez tohoto opatření by první spuštění počáteční úlohy znamenalo, že by si úloha ukládala data mimo jí vyhrazený prostor. Tím by mohla přepsat data dalších procesů.

```
__pocatecni_uloha_sp++;

asm {
  ldhx __pocatecni_uloha_sp
  txs
} // asm
```

Následně je spuštěn čítač časovače, abychom zajistili přepínání úloh. Od tohoto momentu časovač počítá čas strávený v procesu až po dosažení časového kvanta. Spuštěním první úlohy uvedeme celý systém do chodu.

```
T2SC_TSTOP = 0;

__pocatecni_uloha_adresa();
```

V plánovači je jen jedna úloha. Přidání jediné úlohy proběhlo v pořádku, podle kapitoly 5.4. Nemá ale smysl spouštět jádro. Proto pouze zavoláme přímo vloženou úlohu, ve které program setrvá po celou dobu běhu systému. Ke spuštění využijeme adresu, kterou jsme si při přidávání úlohy uložili do proměnné `__pocatecni_uloha_adresa`.

```
} else if (__pocet_uloh == 1) {

  __pocatecni_uloha_adresa();
```

Nebyla zadána žádná úloha. Tento stav může nastat, pokud uživatel nevloží žádnou úlohu. Může se stát, že pro úlohu není dostatek paměti a uživatel netestuje výsledek přidávání úlohy. V tomto případě dojde k zacyklení programu v nekonečné smyčce. Mohli bychom také indikovat chybu, nicméně neznáme přesný důvod, proč v plánovači není žádná úloha.

```
  } else {

    for(;;);

  } // if

} // start
```

5.6 Přepínání úloh

Po vypršení času přidělenému úloze (uplynutí časového kvanta) je voláno přerušení od časovače. Jedná se o funkci `prepniUlohu()`. Tato rutina je implementována jako přerušení, čímž překladači CodeWarrioru sdělujeme, že má adresu této funkce uložit do tabulky vektorů přerušení a že má nadále s touto funkcí jednat jako s obslužnou funkcí přerušení (automaticky se postará o ukládání a obnovení registru H na zásobník – viz kapitola 3.6).

```
interrupt 10 void prepniUlohu() {
```

Při vstupu do obslužné rutiny je mikrokontrolérem automaticky vypnuto globální přerušení i přerušení od časovače. Máme tak zajištěno že z funkce přepínaní se nedostaneme zavoláním dalšího přerušením od jiné periferie. Z přerušení tedy můžeme vystoupit pouze zavoláním jiné rutiny, nebo použitím instrukce `RTI`.

V obslužné rutině musíme nejdříve zajistit přepnutí uživatelského režimu na režim jádra, aby nedošlo k přílišnému využívání paměti procesu jádrem. Minimálnímu režijnímu zatížení se ale stejně nevyhneme (viz níže).

Nejdříve musíme uložit aktuální ukazatel na zásobník do proměnné `__obsluha_temp`. Tento úsek kódu by bylo možné na jiném typu mikrokontroléru optimalizovat. Např. na 68HC908GP32 by bylo možné jej nahradit jedinou instrukcí `sthx __obsluha_temp`. Tato část také zabírá jediný režijní byte paměti každé úlohy. Získaný ukazatel později uložíme do údajů přerušené úlohy.

```
asm {
  tsx
  stx __obsluha_temp:1;
  pshh
  pula
  sta __obsluha_temp:0;
```

Následuje nastavení nového ukazatele na zásobník pro režim jádra. Zajistíme tak vlastní paměťový prostor pro exekutivu jádra.

```
  ldhx ZASOBNIK_OBSLUHA
  txs
} // asm
```

Nyní musíme určit další prováděnou úlohu. Jelikož je náš plánovač založen na metodě Round–Robin (viz 4.4), bude se jednat o další úlohu v řadě, případně o první úlohu (pokud jsme přerušili poslední úlohu ze seznamu).

```
__uloha_dalsi = __uloha_spustena + 1;

if (__uloha_dalsi >= __pocet_uloh) {
  __uloha_dalsi = 0;
} // if
```

Protože jádro nepoužívá ochranný mechanismus COP, ošetřil jsem kritické situace jinak. V přepínači procesů je obsažen **mechanismus kontroly integrity paměti**. Po určení nové úlohy se provede test, který zjistí zda nedošlo k nechtěnému přepsání paměti mezi procesy.

Uvažujme případ, kdy máme dva procesy. Proces 1 má přidělenu paměť 2 byty. Pokud např. proces 1 ve svém běhu použije namísto 2 bytů, které mu byly přiděleny, byty 3, dostane se mimo svůj vyhrazený prostor. Z obrázku 5.1 vyplývá, že 1 byte zapsal do paměti druhého procesu. Tím byla porušena integrita dat a signalizujeme chybu. Podobně může svou paměť vyčerpat i proces 2 v případě, že v systému po přidělení druhého procesu již nezbyla žádná volná paměť. Mohlo by tedy dojít k přepsání paměti obecných proměnných, protože proces 2 by se dostal pod spodní hranici paměti procesů.

V případě, že se podobná situace vyskytne, volá jádro funkci `chyba()`, a díky vypnutému přerušení v této funkci setrvá po zbytek programu.

```
__obsluha_temp--;

if (__uloha_dalsi != 0) {
  if (__seznam_uloh[__uloha_dalsi].zasobnik_vrchol >=
    __obsluha_temp) {
    chyba();
  } // if
} // if

if (__uloha_dalsi == 0) {
  if (__obsluha_temp < ZASOBNIK_DOLNI_MEZ) {
    chyba();
  } // if
} // if

__obsluha_temp++;
```

Pro spuštění nově vybrané úlohy musíme obnovit kontext této úlohy a přepnout se z režimu jádra do uživatelského režimu. Uložíme tedy ukazatel na zásobník současné úlohy do seznamu úloh plánovače a ze stejného seznamu načteme ukazatel na zásobník nově vybrané úlohy. Přepnutí zpět do uživatelského režimu pak obnáší už jen nastavení ukazatele na zásobník na právě získanou adresu.

```
__seznam_uloh[__uloha_spustena].zasobnik_ukazatel =
  __obsluha_temp;
__obsluha_temp = __seznam_uloh[__uloha_dalsi].zasobnik_ukazatel;
__uloha_spustena = __uloha_dalsi;
```

```
    asm {
      ldhx __obsluha_temp
      txs
    } // asm
```

Na konci obslužné rutiny přerušení je nutné znovu povolit přerušení od časovače. Tím opět spustíme odpočet pro další přepnutí úloh. Při opuštění obslužné rutiny je automaticky obnoven registr H a programovací model. Díky tomu, že jsme nastavili SP registr na zásobník nové úlohy, bude se systém chovat, jako by se vracel z podprogramu, který nově vybraná úloha volala. Nová úloha bude spuštěna z místa jejího posledního přerušení.

```
    T2SC_TOF = 0;

  } // prepniUlohu
```

5.7 Zařazení úloh pod správu jádrem a spuštění jádra

Abychom mohli využít služeb jádra, je nutné vložit definované úlohy (uvedeny v následující kapitole) do plánovače. To provedeme ve vstupním bodu programu, tj. ve funkci `main()`.

Nejprve je nutné provést inicializaci jádra. Při té předáme funkci `init()` jako parametr údaj, specifikující velikost časového kvanta. Konkrétně se jedná o hodnotu 1ms. Po uplynutí 1ms tedy bude docházet k přepínání úloh.

```
  void main() {

    init(1);
```

Nyní máme nakonfigurovaný plánovač a můžeme do něj začít přidávat postupně úlohy v pořadí, v jakém chceme aby se vykonávaly. První vložená úloha také bude spuštěna jako první se zavedením jádra. Pokud dojde k chybě při přidávání některé z úloh (viz 5.4), zavolá se funkce `chyba()`.

Pro vložení úlohy voláme funkci `pridejUlohu()`, jejímž prvním parametrem je ukazatel na funkci, která reprezentuje danou úlohu. Druhým parametrem je velikost paměti v bytech, kterou proces pro svůj běh vyžaduje.

```
    if (pridejUlohu(blikani_cervena, 10) == FALSE) { chyba(); }
    if (pridejUlohu(blikani_zelena, 10) == FALSE) { chyba(); }
    if (pridejUlohu(piezo, 5) == FALSE) { chyba(); }
    if (pridejUlohu(lcd_pocitadlo, 20) == FALSE) { chyba(); }
```

Pokud nedošlo během přidávání úloh k chybě, můžeme spustit vlastní jádro. Odstartujeme tak činnost našeho systému, kterou zahájí první vložená úloha (`blikani_cervena`).

```
    start();

  } // main
```

Kapitola 6

Testovací úlohy

Pro otestování činnosti jádra jsem implementoval několik úloh, jejichž popis uvádím.

Pro komunikaci mezi úlohami 1 a 4 je využito sdílené paměti. Proměnné, které mají být sdíleny více procesy, jsem umístil do nulté stránky paměti, z důvodu zvýšení efektivity. Ve zdrojovém kódu je tato oblast vyhrazena pomocí direktiv překladače :

```
#pragma DATA_SEG MY_ZEROPAGE

  unsigned int pocet_bliknuti = 0;

  // ... další sdílené proměnné

#pragma DATA_SEG DEFAULT

  // ... proměnné využívané např. více funkcemi jednoho procesu
```

Proměnné uvnitř funkcí jednotlivých úloh se ukládají do paměti, kterou si úloha při vkládání do plánovače rezervuje.

6.1 Úloha 1

Jedná se o blikání červené LED diody.

LED dioda je připojena na port B a její stav je upravován pomocí 5. bitu tohoto portu. Funkce pro tento proces je pojmenována `blikani_cervena()`.

Po inicializaci portu B pro zajištění ovládání této LED diody, následuje v těle funkce nekonečná smyčka s vlastním výkonným kódem.

```
DDRB_DDRB5 = 1;
PTB_PTB5 = 1;

for(;;) {
```

Před změnou stavu diody je nutné vyčkat určitý okamžik, aby byla změna viditelná. Zpoždění dosáhneme zavedením následujícího cyklu:

```
for (i = 0; i < 20000; i++);
```

Přesná doba zpoždění není pro naše účely podstatná. Po dokončení tohoto prázdného cyklu můžeme změnit stav LED diody. Při nulové hodnotě LED dioda svítí (je připojena přes pull–up rezistor), při hodnotě log. 1 je zhasnutá.

```
PTB_PTB5 = ~PTB_PTB5;
```

Pokud jsme diodu právě rozsvítili, zvýšíme počet bliknutí. Tento údaj prostřednictvím sdílené paměti použijeme v úloze 4.

```
if (PTB_PTB5 == 0) {
  pocet_bliknuti++;
  if (pocet_bliknuti == 100) {
    pocet_bliknuti = 0;
  } // if
} // if

} // for
```

V kódu je vložena podmínka pro omezení počitadla bliknutí. Maximální hodnotu jsem omezil na 99, tzn. na 2 segmenty LCD displeje. Přidání dalších řádů je možné, stačí patřičně dodefinovat a upravit funkce 4. úlohy.

6.2 Úloha 2

Tento proces zajišťuje blikání zelené LED diody.

LED dioda je připojena na port B. Stav diody je ovládán bitem 4 tohoto portu. Funkce ve zdrojovém kódu je označena jako `blikani_zelena()`.

Pro běh úlohy je nutná inicializace portu B, po níž následuje nekonečná smyčka, tvořící tělo procesu.

```
DDRB_DDRB4 = 1;
PTB_PTB4 = 1;

for(;;) {
```

Do těla smyčky pak zařadíme zpoždění, aby bylo zřejmé přepnutí stavu LED diody. Počitadlo cyklu je zde nižší, frekvence blikání bude tedy vyšší, než u úlohy 1. Po dokončení zpožďovacího cyklu změníme stav LED diody.

```
  for (i = 0; i < 4000; i++);
  PTB_PTB4 = ~PTB_PTB4;

} // for
```

6.3 Úloha 3

Úloha generující brum z piezoelektrického měniče.

Měnič je připojen na port A a jeho stav je kontrolován 1. bitem tohoto portu. Funkce tohoto procesu je pojmenována `piezo()`.

Nastavením příslušného pinu portu A inicializujeme měnič.

```
DDRA_DDRA1 = 1;
PTA_PTA1 = 1;

for (;;) {
```

V nekonečné smyčce pak můžeme měnit jeho stav – přepínat log. úrovně – a dosáhnout tak generování tónu určitého kmitočtu. Frekvence zde není důležitá, protože důsledkem přepínání procesů vznikne zpoždění, které tuto frekvenci posune.

```
  for (i = 0; i < 1500; i++);
  PTA_PTA1 = ~PTA_PTA1;

} // for
```

6.4 Úloha 4

Úloha zobrazuje počet bliknutí červené LED diody z úlohy 1 na LCD displeji.

Pro svou činnost úloha využívá LCD displeje umístěného na vývojovém kitu. Tento proces tvoří více funkcí, jejichž význam uvádím.

Hlavní funkcí úlohy je `lcd_pocitadlo()`. V ní smažeme obsah LCD displeje a provedeme jeho konfiguraci. Jedná se především o nastavení vlastností zobrazování. Vyčerpávající popis konfiguračních registrů je uveden v literatuře [1].

```
smazatLCD();

CONFIG2 |= 4;
LCDCR = 0xa5;
LCDCLK = 0x53;

for(;;) {
```

Po inicializaci následuje tělo vlastní funkce pro zobrazování údaje na displeji. Z počtu bliknutí získáme hodnoty jednotlivých číselných řádů a zavoláme funkci pro zobrazení číslic.

```
  desitky = pocet_bliknuti / 10;
  jednotky = pocet_bliknuti % 10;

  zobrazNaLCD(desitky, jednotky);

} // for
```

Funkce `smazatLCD()` vynuluje obsahy jednotlivých datových registrů LCD displeje. Tím zajistíme, že před zobrazením údaje nebude displej obsahovat jiné znaky.

Pro zobrazení číselného údaje, informujícího o počtu bliknutí, potřebujeme 2 cifry. Předáváme je jako parametr funkci `zobrazNaLCD()`. Ta nejprve smaže displej, zavoláním `smazatLCD()`, a následně rozsvítí patřičné segmenty jednotlivých číslic displeje.

```
void zobrazNaLCD(unsigned int desitky, unsigned int jednotky) {

  smazatLCD();

  LDAT9 |= lcd_cislice_2[2 * desitky];
  LDAT10 |= lcd_cislice_2[2 * desitky + 1];

  LDAT7 |= lcd_cislice_3[2 * jednotky];
  LDAT8 |= lcd_cislice_3[2 * jednotky + 1];

} // zobrazNaLCD
```

Pro zobrazení řádu desítek použijeme druhou číslici prvního řádku LCD. Nastavení jejích segmentů provedeme přes registry LDAT9 a LDAT10. Jednotky zobrazíme na třetí pozici LCD. Konfigurace segmentů je dostupná prostřednictvím registrů LDAT7 a LDAT8.

Funkce `zobrazNaLCD()` také využívá dvou specifických polí konstant. Jedná se o převodní tabulky pro jednotlivé číslice LCD displeje, které používáme. Obsahují konfiguraci datových registrů displeje pro každou číslici. Jedná se o pole s konstantním obsahem, proto jsou umístěny ve FLASH paměti.

Kapitola 7

Závěr

7.1 Zhodnocení funkčnosti

Navržené a implementované jádro pracuje podle předpokladů. Při testování jsem nenarazil na žádný problém, který by vedl k nesprávné funkci jádra.

Vlastní přepnutí úloh je rychlé, při frekvenci sběrnice 2,4576MHz trvá pouhých 75μs. Uvažujeme–li 4 testovací úlohy a časové kvantum 1ms, spotřebuje přepínání procesů 7% času procesoru z celkových 4.3ms, než se dostane opět na první úlohu. Tato hodnota není zanedbatelná, bylo by proto vhodné přepínání úloh zefektivnit. Např. by bylo možné odstranit paměťovou kontrolu. Je nutné brát ale v úvahu, že pro hodnoty časového kvanta menší než 1ms bude doba přepínání tvořit větší procentuální podíl. Řešením by bylo zvýšení frekvence jádra mikrokontroléru, např. použitím externího oscilátoru. Pro hodnoty časového kvanta vyšší než 1ms by se naopak procentuální podíl zmenšoval, ovšem za tu cenu, že procesy by se nemusely přepínat dostatečně rychle a bylo by tak patrné jejich postupné provádění.

Jádro operačního systému bylo vyvinuto na mikrokontroléru 68HC908LJ12. Přenositelnost na jiné typy mikrokontrolérů z rodiny HC08 je možná po nastavení základních konstant, uvedených v kapitole 5.2. Prakticky ale byla ověřena funkčnost pouze na mikrokontroléru 68HC908LJ12.

Pro správnou činnost jádra je vyžadována přítomnost časovače, umožňujícího generovat přerušení. Žádné další požadavky na vestavěné periferie systém nemá.

Tabulka 7.1 uvádí několik typů mikrokontrolérů, pro které by bylo možné navržené jádro použít. Ve sloupci **Počet úloh** jsou uvedeny tři typy úloh. Hodnoty uvedené v tabulce udávají, kolik procesů podobných dané úloze by bylo možné pro mikrokontrolér použít. Při výpočtech vycházíme z následujících hodnot:

- **A** – odpovídá úloze `blikani_cervena()`, jejíž náročnost na paměť FLASH je 64 bytů, v paměti RAM pak potřebuje 11 bytů (10 + 1 sdílená paměť).
- **B** – představuje úlohu `piezo()`, s paměťovou náročností 32 bytů FLASH, 5 bytů RAM.
- **C** – zastupuje úlohu `lcd_pocitadlo()`, která v paměti FLASH zabírá 194 bytů a v paměti RAM 20 bytů.

K požadavkům na paměť RAM jednotlivých úloh je nutné ještě přičíst režijní údaje, tj. 7 bytů pro každou úlohu v plánovači. Jádro samotné zabírá v paměti FLASH prostor 496 bytů, v RAM potom $15 + 4 \cdot n$ bytů, kde n je maximální počet úloh.

Mikrokontrolér	**FLASH** [B]	**RAM** [B]	**Počet V/V**	**Frekvence** [MHz]		**Počet úloh**		
				jádra	**sběrnice**	**A**	**B**	**C**
68HC908QT1A	1 500	128	6	32	8	5	7	3
MC908QT4ACDWE	4 000	128	6	32	8	5	7	3
MC908JB8FBE	8 000	256	37	32	3	11	15	7
68HC908JB16	16 000	384	21	12	6	17	23	11
68HC908LJ12	12 000	512	32	32	8	23	31	16
68HC908GP32	32 000	512	31	32	8	23	31	26
68HC908BD48	48 000	1 024	32	32	6	48	63	32
68HC908AP64	62 000	2 048	30	32	8	96	127	65

Tabulka 7.1: Porovnání některých typů mikrokontrolérů

Vzhledem k tomu, že samotné jádro má minimální náročnost na paměť FLASH, záleží u jednotlivých mikrokontrolérů při použití uvedených testovacích úloh převážně na velikosti RAM paměti.

Při návrhu jsem nenarazil na vážnější komplikace, které by neumožnily implementaci výsledného jádra. Při návrh správy úloh jsem postupoval systematicky, od logického principu. Při implementaci bylo ovšem nutné přizpůsobit návrh zvolené architektuře a jejím možnostem.

Realizací a ověřením tohoto jádra jsme získali systém, který je možné využívat pro nejrůznější úkoly. Cílovou oblastí využití jsou systémy, ve kterých je potřeba současné obsluhy více zařízení resp. zpracování většího množství nezávislých dat. Jako příklad uvádím hromadný sběr dat, vyhodnocování stavů senzorů, přizpůsobování formátu vstupních dat pro více výstupů apod.

Analýzou zadaného problému jsem si osobně rozšířil znalosti práce operačního systému a jeho jádra. Seznámil jsem se se základními principy sloužícími k dosažení víceúlohovosti, i s problémy které se objevují při jejich řešení. Prakticky jsem si vybrané přístupy ověřil a získal tak představu o způsobu jejich implementace. Také jsem získal zkušenosti s programováním vybraného mikrokontroléru a detailně jsem se seznámil zejména s využitím časovače a přerušovacího podsystému. Práci považuji osobně za velmi přínosnou.

Dále úvádím soupis možných vylepšení, která by vedla k větší výkonnosti a efektivnosti použití jádra.

7.2 Náměty pro další postup

Zahrnutí níže uvedených funkcionalit by usnadnilo a zpříjemnilo práci s rozhraním jádra, zvýšilo jeho efektivitu a rozšířilo jeho schopnosti.

Komunikace mezi procesy. Současné řešení pomocí globálních proměnných dostačuje potřebám jádra. Bylo by ale možné tuto část vylepšit, např. zavedením ochrany přístupu pomocí semaforů. Semafory by se ovládaly příslušnými funkcemi, kdy jedna funkce by obsadila požadovaný zdroj a druhá by čekala na jeho uvolnění.

Implementace správy paměti. Uživatel by nebyl nucen zadávat množství potřebné paměti při vkládání úlohy do plánovače. Také by bylo možné odstranit kontrolní mechanismus integrity paměti v přepínači procesů. Správa paměti by přinesla efektivnější využití RAM prostoru, ovšem za cenu zvýšení režijních nákladů. Mohla by být řešena např. přidělováním bloků pevné velikosti nebo alokací na haldě. Paměť by byla přidělována prostřednictvím funkcí typu alloc/free.

Dynamický počet úloh. Pokud by byla v systému obsažena správa paměti, bylo by možné počet procesů určovat dynamicky, bez omezení jejich maximálního počtu. Seznam úloh by bylo možné řešit např. lineárním seznamem.

Zavedení správy souborů. Bylo by možné využít rozhraní SCI ke komunikaci s externí pamětí, například kartami typu SD/MMC, k ukládání a čtení souborů. Mohli bychom tak např. monitorovat běh systému, shromažďovaná data průběžně úkládat, zaznamenávat konfiguraci před ukončením běhu systému nebo průběžně zálohovat prováděné výpočty. Realizace by probíhala s využitím přerušení, kdy v případě žádosti o zápis, by se přerušila probíhající úloha, zakázalo by se přerušení, provedlo by se zapsání na paměťové médium, přerušení by se opět povolilo a přerušená úloha by pokračovala. Tento mechanismus by zajistil, že zápis by nebylo možné přerušit. Analogicky by probíhalo čtení.

Vylepšení plánovače. Použitý algoritmus Round–Robin představuje jednoduchý plánovač procesů. Bylo by užitečné zavést např. priority, jejich dynamické určování při běhu, upřednostňování procesů s vyšší prioritou apod. Toto vylepšení by mírně zpomalilo přepínání úloh, vedlo by ale k efektivnějšímu plánování jednotlivých procesů. Priorita by byla reprezentována číslem, přičemž nejvyšší číslo by znamenalo nejvyšší prioritu.

Implementace jiného plánovače. Namísto použitého plánovacího algoritmu by bylo možné zavést např. stavy úloh (blokovaná, připravená, běžící). Implementovány by byly jako jednotlivé seznamy, ze kterých by plánovač vybíral resp. do kterých by odkládal úlohy ke spuštění resp. zastavení. Získali bychom větší dynamičnost, ovšem za cenu zvýšení požadavků jádra na paměť RAM.

Využití LCD k vizualizaci. Umožnili bychom uživateli sledovat stav jádra, případně jednotlivých procesů, prostřednictvím LCD displeje. Měl by tak možnost vidět, který proces se bude vykonávat, který skončil s chybou, číslo iterace procesu apod. Cenou za tento komfort by bylo obsazení LCD displeje na vývojové desce pro potřeby jádra.

Lepší signalizace chyb. Chyby jádra, stejně tak jako chyby uživatelské, by bylo možné zobrazovat na LCD displeji, vysílat přes SCI na terminál, ukládat do logovacích souborů apod. Pro zobrazení nebo odeslání chyby by existovala speciální funkce, kterou by mohlo jádro nebo proces volat.

Literatura

[1] Motorola: Katalogové listy 68HC908LJ12. 2002, [Online; navštíveno 10. 5. 2007].
URL http://www.freescale.com/files/microcontrollers/doc/data_sheet/MC68HC908LJ12.pdf

[2] Motorola: Freescale polovodiče. 2007, [Online; navštíveno 10. 5. 2007].
URL http://www.freescale.com/

[3] SCHWARZ, J.; RŮŽIČKA, R.; STRNADEL, J.: Studijní opora k předmětu Mikroprocesorové a vestavěné systémy. 2006.

[4] STRNADEL, J.: Studijní opora k předmětu Realtime operační systémy. 2006, [Online; navštíveno 10. 5. 2007].
URL http://www.fit.vutbr.cz/~strnadel/download/ros_studijni_opora.pdf

[5] Wikipedia: Kernel. 2007, [Online; navštíveno 10. 5. 2007].
URL http://en.wikipedia.org/wiki/Kernel_(computer_science)

[6] Wikipedia: Microkernel. 2007, [Online; navštíveno 10. 5. 2007].
URL http://en.wikipedia.org/wiki/Microkernel

[7] Wikipedia: Operační systém. 2007, [Online; navštíveno 10. 5. 2007].
URL http://en.wikipedia.org/wiki/Operating_System