VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

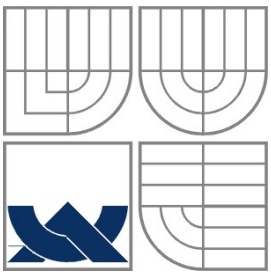
# IMPLEMENTACE REAL-TIME OPERAČNÍHO SYSTÉMU μC/OS-II NA PLATFORMĚ FREESCALE MC9S08JM60

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

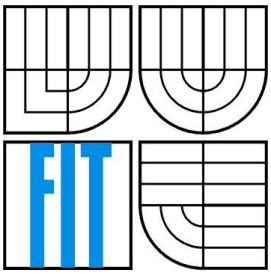
AUTOR PRÁCE
AUTHOR

Bc. LUKÁŠ VÁVRA

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

# IMPLEMENTACE REAL-TIME OPERAČNÍHO SYSTÉMU μC/OS-II NA PLATFORMĚ FREESCALE MC9S08JM60

IMPLEMENTATION OF μC/OS-II REAL-TIME OPERATING SYSTEM ON FREESCALE MC9S08JM60 PLATFORM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. LUKÁŠ VÁVRA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JOSEF STRNADEL, Ph.D.

BRNO 2013

# Abstrakt

Pojem real-time operační systém pro vestavené aplikace má dnes velký význam. Tato práce popisuje ovládání robotické ruky pomocí systému μC/OS-II na platformě MC9S08JM60. Robotická ruka využívá pro pohyb tři serva HS-311, která jsou řízena pomocí PWM. Šířku pulzu PWM nastavujeme dle načtených dat z akcelerometru, který je umístěn na DEMOJM kitu.



# Abstract

The term real-time operating system for embedded aplications has great importance today. This thesis descbires robitc arm control by μC/OS-II system on MC9S08JM60 platform. Robotic arm uses three servomotors HS-311 for motion, which are controlled by PWM. PWM pulse width is set according to the loaded data from the accelerometer, which is placed on DEMOJM board.



# Klíčová slova

μC/OS-II, vestavená aplikace, robotický manipulátor, PWM, servo, HS-311, MC9S08JM60, DEMOJM, akcelerometr.



# Keywords

μC/OS-II, embedded aplication, robotic arm, PWM, servo, HS-311, MC9S08JM60, DEMOJM, accelerometer.



# Citace

Lukáš Vávra: Implementace real-time operačního systému μC/OS-II na platformně Freescale MC9S08JM60, diplomová práce, Brno, FIT VUT v Brně, 2013

# Implementace real-time operačního systému μC/OS-II na platformně Freescale MC9S08JM60

## Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Josefa Strnadela, Ph.D.
Další informace jsem čerpal z webových stránek www.freescale.com, www.micrium.com.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Lukáš Vávra
30. července 2013

## Poděkování

Rád bych zde poděkoval lidem z mého okolí dodávajícím mi motivaci k dokončení této práce a to především, kamarádům, kolegům z práce a rodičům. Dále vedoucímu práce Ing. Josefu Strnadelovi, Ph.D a vedení fakutly za umožnění dokončení studia. Velký dík patří také korektorovi mé práce.

# Obsah

# 1 Úvod

V dnešní době se návrh vestavěných aplikací posouvá stále dále. Doby, kdy programátor tvořil zdrojové kódy pouze v assembleru jsou prakticky již minulostí. Nejčastěji se používá jazyk C. Již v roce 1992 vytvořil J. J. Labrosse první verzi real-time operačního systému μC/OS pro své vlastní použití. Od Té doby se RT systémy pro vestavěné aplikace vyvinuly do stále více komplexnějších verzí. Nyní je již verze systému μC/OS-III.

V této diplomové práci je vytvořena sada testovacích aplikací systému μC/OS-II na platformě MC9S08JM60, které slouží k demonstraci funkčnosti systému, zároveň nás více zavedou do problematiky tvorby aplikací a ukáží na případné limitace systému a na možné programátorské chyby.

Ve druhé kapitole je podrobné zaměření na čip MC9S08JM60, jeho základní charakteristiky, integrované periferie, režimy běhu čipu, systém přerušení, pamětový prostor a samotnou architekturu jádra mikrokontroleru – procesoru HCS08.

V následující kapitole se seznámíme se základními pojmy real-time operačních systémů nutných k základnímu pochopení problematiky návrhu takovýchto aplikací. Dále se seznámíme s různými typy jader real-time operačních systémů. Pronikneme do problematiky plánování úloh výčtem známých plánovacích mechanismů. Povíme si něco o prostředcích sloužících ke komunikaci a synchronizaci jednotlivých úloh v těchto systémech jako jsou semafory, poštovní schránky, globální proměnné, atd.

Čtvrtá kapitola je zaměřena na real-time operační systém společnosti Micrium μC/OS-II. Zmíněny jsou základní vlastnosti a výhody tohoto systému. Dále je podrobněji popsáno jádro systému, a to především implementace kritické sekce, obecné seznámení se s úlohou v systému a jejími datovými strukturami (zásobníkem a řídícím blokem úlohy), plánovacím mechanismem jádra, generováním vnitřního periodického přerušení v systému a způsobu zpracování přerušení.

Pátá kapitola obsahuje uživatelskou příručku k jádru systému μC/OS-II, která by měla být vést běžného uživatele při zprovoznění systému na platformě MC9S08JM60. Je zde popsána instalace všech potřebných ovladačů a SW, následně postup při portaci i s vysvětlenou funkcí systémových procedur a i samotný test funkčnosti portu.

V šesté kapitole jsou zmíněny testovací aplikace různých nejčastěji používaných systémových funkcí a řešení případných problémů při jejich použití. Jedná se o funkce používané v souvislosti se správou úloh a uspáváním úloh. Zmíněny jsou i základní synchronizační a komunikační prostředky.

Sedmá kapitola je věnována RT aplikaci, pomocí které je ovládána robotická ruka. Je zde popsán způsob ovládání ruky, návrh aplikace, popis implementace celé aplikace i zapojení a uvedení do provozu.

# 2 Čip MC9S08JM60

V této kapitole se podrobněji seznámíme s mikrokontrolery řady HCS08, přesněji typem MC9S08JM60 (dále zkráceně JM60). Zmíníme základní vlastnosti tohoto čipu a podrobněji se zaměříme na ty části, které jsou podstatné pro implementaci uC/OS-II.

Obrázky a tabulky v této kapitole jsou převzaty z dokumentace k čipu JM60 [1].

## 2.1 Základní charakteristika řady JM60

Mikrokontroler JM60 je rozšířením nabídky firmy Freescale v řadě 8bitových mikrokontrolerů. Je určen pro průmyslové řízení a uživatelské aplikace, jako jsou například periferie pro osobní počítače, diagnostická zařízení, lékařská technika nebo snímače čárových kódů.

Technické parametry čipu:

- pouzdro QFN 48pinů, QFP 64 pinů, LQFP 64 nebo 44 pinů
- flash paměť o velikosti 60 KB – read/write/program
- RAM paměť o velikosti 4 KB, 256 B USB RAM
- frekvence jádra čipu S08 48 MHz
- maximální rychlost sběrnice 24 MHz (při použití USB modulu 48 MHz)
- napájecí napětí v rozmezí 2,7 V až 5,5 V při teplotě -40 °C až 85 °C, díky rozsahu napájecího napětí není nutno použít stabilizátor. Většina obvodů pracuje buď na napětí 3,3 V nebo 5 V
- podpora až 32 přerušení/resetovacích zdrojů, výhodné pro real-time aplikace
- wait a dva stop módy pro šetření energie
- USB 2.0 full-speed modul (12 Mb/s) se zabudovaným regulátorem 3,3 V (v případě napojení na vyšší napájecí napětí)
- analogový komparátor (ACMP)
- 12kanálový 12bitový ADC převodník
- dvě sériová komunikační rozhraní SCI
- dvě sériové periferní rozhraní SPI
- dva časovače 6kanálový a 2kanálový TPM 16bitový časovač
- KBI modul umožňující až 8 vnějších přerušení
- až 51 I/O pinů, jeden pouze vstupní a jeden pouze výstupní
- COP watchdog modul
- detekce nízkého napájecího napětí
- blokace přístupu do flash paměti

## 2.2 Různé verze čipu JM60

Modelová řada JM60 se vyrábí v různých pouzdrech s různým počtem pinů. Díky tomu jsou vlastnosti periferií odlišné. Pro úplnost je v tabulce uvedena i verze čipu MC9S08JM32, která se liší pouze ve velikosti paměti RAM a flash pro dané pouzdro.

| modul | čip | | | | | |
|---|---|---|---|---|---|---|
| | MC9S08JM60 | | | MC9S08JM32 | | |
| pouzdro | 64pin | 48pin | 44pin | 64pin | 48pin | 44pin |
| flash | 60912 | | | 32768 | | |
| RAM | 4096 | | | 2048 | | |
| USB RAM | 256 | | | 256 | | |
| ACMP | ano | | | ano | | |
| ADC | 12kanálový | 8kanálový | 8kanálový | 12kanálový | 8kanálový | 8kanálový |
| IIC | ano | | | ano | | |
| IRQ | ano | | | ano | | |
| KBI | 8 | 7 | 7 | 8 | 7 | 7 |
| SCI1 | ano | | | ano | | |
| SCI2 | ano | | | ano | | |
| SPI1 | ano | | | ano | | |
| SPI2 | ano | | | ano | | |
| TPM1 | 6kanálový | 4kanálový | 4kanálový | 6kanálový | 4kanálový | 4kanálový |
| TPM2 | 2kanálový | | | 2kanálový | | |
| USB | ano | | | ano | | |
| I/O piny | 51 | 37 | 33 | 51 | 37 | 33 |
| Typ pouzdra | QFP / LQFP | QFN | LQFP | QFP / LQFP | QFN | LQFP |

tabulka 2.1 periferie JM60 a JM32

## 2.3 Režimy běhu čipu

Mikrokontroler řady JM60 nabízí uživateli mimo základní *run mode* a *active background mode* ještě tři režimy se sníženým příkonem, a to *wait*, *stop2* a *stop3* režimy.

### 2.3.1 Run mode

Režim běhu MCU, čip je automaticky nastaven na tento režim po resetu. Běh programu začíná na adrese 0xFFFE:0xFFFF

### 2.3.2 Active bacground mode

Režim pro analýzu běhu MCU. Využívá background debug controller, který je součástí čipu. Režim může být aktivován pomocí pěti různých přístupů:

### 2.3.3 Režim WAIT

Režim je vyvolán instrukcí WAIT, po jejím zavolání je CPU zastaven, systémové hodiny nadále běží a dojde k povolení přerušení v CCR registru. Pokud přijde přerušení, CPU ukončí wait režim, uloží kontext přerušené úlohy a vyvolá příslušné přerušení.

### 2.3.4 STOP režimy

Režim je vyvolán instrukcí STOP, CPU hodiny a hodiny sběrnice jsou zastaveny. Výběr stop režimu je proveden dle následují tabulky:

| STOPE | ENBDM | LVDE | LVDSE | PPDC | STOP režim |
|---|---|---|---|---|---|
| 0 | x | x | | x | Stop režim vypnut |
| 1 | 1 | x | | x | Stop3 režim s BDM |
| 1 | 0 | Oba bity 0 | | x | Stop3 režim s napěťovou regulací |
| 1 | 0 | Jeden bit 0 | | 0 | Stop3 |
| 1 | 0 | Jeden bit 0 | | 1 | Stop2 |

Tabulka 2.2 STOP režimy

#### STOP3

Režim lze opustit buď pomocí resetu nebo přerušením z následujících zdrojů: RTC přerušení, USB resume přerušení, LVD, ADC, IRQ, KBI, SCI nebo ACMP.

V případě probuzení čipu pomocí resetu načítá čip novou adresu na reset vektoru, pokud je probuzen jiným modulem, dojde ke zpracování přerušení daného modulu.

#### STOP2

Většina periferií na čipu je vypnuta až na pamět RAM. Všechny I/O piny jsou zachovány díky latch registrům. Stop2 režim nabízí širší možnosti nastavení, které zde nebudou zmíněny. Podrobné vysvětlení se nachází v dokumentaci k čipu [1].

V následující tabulce jsou zobrazeny možné varianty pro oba dva režimy, jak Stop2 tak Stop3

| Periferie | Režim | |
|---|---|---|
| | Stop2 | Stop3 |
| CPU | off | pohotovostní |

| | | |
|---|---|---|
| RAM | pohotovostní | pohotovostní |
| Flash | off | pohotovostní |
| Registry portů | off | pohotovostní |
| ADC | off | volitelně zapnut |
| ACMP | off | volitelně zapnut |
| MCG | off | volitelně zapnut |
| IIC | off | pohotovostní |
| RTC | volitelně zapnut | volitelně zapnut |
| SCI | off | pohotovostní |
| SPI | off | pohotovostní |
| TPM | off | pohotovostní |
| Regulátor napětí | off | pohotovostní |
| XOSC | off | volitelně zapnut |
| I/O piny | Latch registr | Latch registr |
| USB tansceiver | off | volitelně zapnut |
| USB regulátor | off | pohotovostní |
| USB RAM | pohotovostní | pohotovostní |

Tabulka 2.3 Periferie a Stop režimy

## 2.4 Paměťový prostor

Paměť čipu JM60 se skládá z paměti RAM, flash paměti programu pro neměnná data a I/O řídících a stavových registrů. Tyto registry se dělí do tří skupin:

### direct-page registry

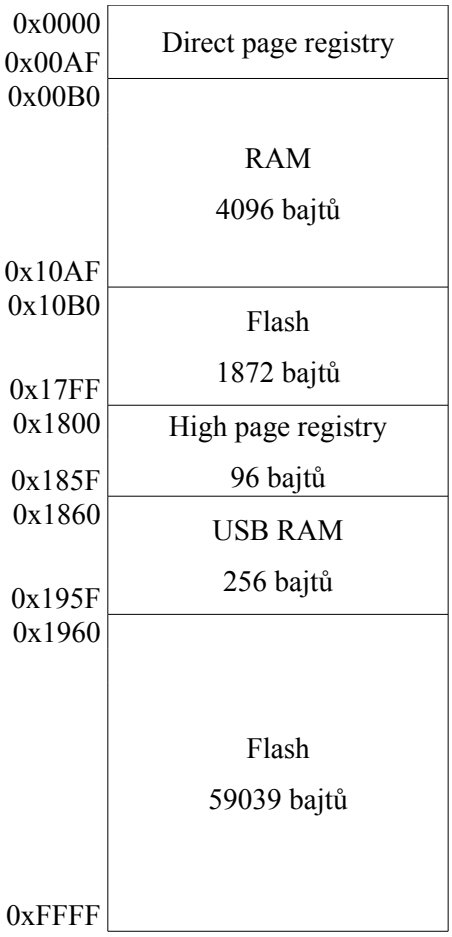
Adresy v rozsahu 0x0000 až 0x00AF. Přístupné velmi rychle díky přímé adresaci. Lze použít bitové operace. Na těchto adresách se nacházejí řídící a stavové registry periferií čipu

### high-page registry

Adresy v rozsahu 0x1800 až 0x185F. Obsahuje registry, jejich použití je velmi málo frekventované, Většinou se nastaví jednou a vícekrát se již nepoužívají.

### Nonvolatile registry

Jsou umístěny v paměti flash, běžně se v aplikacích nepoužívají.




Obrázek 2.1 mapa paměti

## 2.5 Přerušení a reset

### 2.5.1 Přerušení

Využívá tabulku vektorů přerušení s pevně danými prioritami jednotlivých žádostí. Pokud se objeví žádost o přerušení a čip ji detekuje jako legální (přerušení dané periferie je povoleno či softwarové vyvolané instrukcí SWI) dojde k zpracování přerušení následujícím způsobem:

- uložení CPU registrů na zásobník
- nastavení I bitu v CCR pro maskování případných dalších přerušení
- načtení adresy nejprioritnější rutiny přerušení
- naplnění fronty instrukcí prvními třemi bajty od načtené adresy z vektoru přerušení

Rutina přerušení končí návratovou instrukcí RTI. Tato instrukce se postará o načtení dat ze zásobníku a obnovení kontextu CPU, který byl přerušen.

Díky kompatibilitě s čipem M68HC08 se neukládá na zásobník registr H. Je nutné jej uložit na zásobník hned po začátku rutiny přerušení a následně těsně před instrukcí RTI jen obnovit.

## Tabulka priorit přerušení

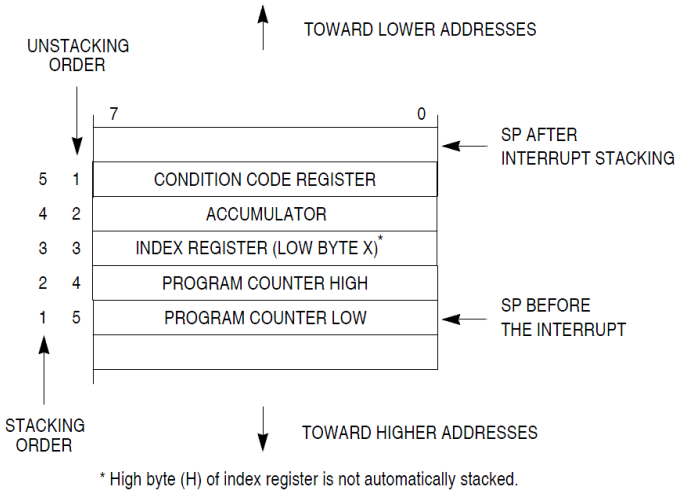
Jak již bylo výše zmíněno, čip využívá tabulky přerušení s pevně danými prioritami jednotlivých periferií. Programátorovi také umožňuje využít dvou vektorů pro vlastní přerušovací rutiny. Jednotlivé vektory přerušení jsou seřazeny v následující tabulce od nejnižší po nejvyšší prioritu.

| Vector Number | Address (High/Low) | Vector Name | Module | Source | Enable | Description |
|---|---|---|---|---|---|---|
| 31 to 30 | 0xFFC0:FFC1<br>0xFFC2:FFC3 | Unused vector space (available for user program) | | | | |
| 29 | 0xFFC4:FFC5 | Vrtc | System control | RTIF | RTIE | RTC real-time interrupt |
| 28 | 0xFFC6:FFC7 | Viic | IIC | IICIF | IICIE | IIC |
| 27 | 0xFFC8:FFC9 | Vacmp | ACMP | ACF | ACIE | ACMP |
| 26 | 0xFFCA:FFCB | Vadc | ADC | COCO | AIEN | ADC |
| 25 | 0xFFCC:FFCD | Vkeyboard | KBI | KBF | KBIE | Keyboard pins |
| 24 | 0xFFCE:FFCF | Vsci2tx | SCI2 | TDRE<br>TC | TIE<br>TCIE | SCI2 transmit |
| 23 | 0xFFD0:FFD1 | Vsci2rx | SCI2 | IDLE<br>RDRF | ILIE<br>RIE | SCI2 receive |
| 22 | 0xFFD2:FFD3 | Vsci2err | SCI2 | OR<br>NF<br>FE<br>PF | ORIE<br>NFIE<br>FEIE<br>PFIE | SCI2 error |
| 21 | 0xFFD4:FFD5 | Vsci1tx | SCI1 | TDRE<br>TC | TIE<br>TCIE | SCI1 transmit |
| 20 | 0xFFD6:FFD7 | Vsci1rx | SCI1 | IDLE<br>RDRF | ILIE<br>RIE | SCI1 receive |
| 19 | 0xFFD8:FFD9 | Vsci1err | SCI1 | OR<br>NF<br>FE<br>PF | ORIE<br>NFIE<br>FEIE<br>PFIE | SCI1 error |
| 18 | 0xFFDA:FFDB | Vtpm2ovf | TPM2 | TOF | TOIE | TPM2 overflow |
| 17 | 0xFFDC:FFDD | Vtpm2ch1 | TPM2 | CH1F | CH1IE | TPM2 channel 1 |
| 16 | 0xFFDE:FFDF | Vtpm2ch0 | TPM2 | CH0F | CH0IE | TPM2 channel 0 |
| 15 | 0xFFE0:FFE1 | Vtpm1ovf | TPM1 | TOF | TOIE | TPM1 overflow |
| 14 | 0xFFE2:FFE3 | Vtpm1ch5 | TPM1 | CH5F | CH5IE | TPM1 channel 5 |
| 13 | 0xFFE4:FFE5 | Vtpm1ch4 | TPM1 | CH4F | CH4IE | TPM1 channel 4 |
| 12 | 0xFFE6:FFE7 | Vtpm1ch3 | TPM1 | CH3F | CH3IE | TPM1 channel 3 |
| 11 | 0xFFE8:FFE9 | Vtpm1ch2 | TPM1 | CH2F | CH2IE | TPM1 channel 2 |
| 10 | 0xFFEA:FFEB | Vtpm1ch1 | TPM1 | CH1F | CH1IE | TPM1 channel 1 |
| 9 | 0xFFEC:FFED | Vtpm1ch0 | TPM1 | CH0F | CH0IE | TPM1 channel 0 |
| 8 | 0xFFEE:FFEF | Reserved | — | — | — | — |
| 7 | 0xFFF0:FFF1 | Vusb | USB | STALLF<br>RESUMEF<br>SLEEPF<br>TOKDNEF<br>SOFTOKF<br>ERRORF<br>USBRSTF | STALL<br>RESUME<br>SLEEP<br>TOKDNE<br>SOFTOK<br>ERROR<br>USBRST | USB Status |
| 6 | 0xFFF2:FFF3 | Vspi2 | SPI2 | SPRF<br>MODF<br>SPTEF<br>SPMF | SPIE<br>SPIE<br>SPTIE<br>SPMIE | SPI2 |
| 5 | 0xFFF4:FFF5 | Vspi1 | SPI1 | SPRF<br>MODF<br>SPTEF<br>SPMF | SPIE<br>SPIE<br>SPTIE<br>SPMIE | SPI1 |
| 4 | 0xFFF6:FFF7 | Vlol | MCG | LOLS | LOLIE | MCG loss of lock |
| 3 | 0xFFF8:FFF9 | Vlvd | System control | LVWF | LVWIE | Low-voltage detect |
| 2 | 0xFFFA:FFFB | Virq | IRQ | IRQF | IRQIE | IRQ pin |
| 1 | 0xFFFC:FFFD | Vswi | Core | SWI Instruction | — | Software interrupt |
| 0 | 0xFFFE:FFFF | Vreset | System control | COP<br>LVD<br>$\overline{\text{RESET}}$ pin<br>Illegal opcode<br>LOC<br>POR<br>BDFR | COPE<br>LVDRE<br>—<br>ILOP<br>CME<br>POR<br>BDFR | Watchdog timer<br>Low-voltage detect<br>External pin<br>Illegal opcode<br>Loss of clock<br>Power-on-reset<br>BDM-forced reset |

Obrázek 2.2 Tabulka priorit přerušení

### Zásobník a přerušení

Na obrázku 2.3 je zobrazen průběh automatické práce se zásobníkem při vyvolání přerušení. Před přerušením ukazuje SP na první volnou položku zásobníku. Při vyvolání přerušení dojde k postupnému ukládání dat na obrázku zespoda nahoru od nižšího bajtu PC registru až po CCR registr. Zásobník automaticky ukazuje na první volnou položku. Na ní je ovšem nutné uložit H registr a následně posunout (dekrementovat) SP registr o jednu adresu, aby opět ukazoval na první prázdnou položku zásobníku. Před ukončením rutiny přerušení je potřeba zvýšit adresu zásobníku, vyčíst H registr a následně je možné zavolat instrukci RTI.




Obrázek 2.3 Práce se zásobníkem při přerušení

## 2.5.2 Reset čipu

Reset čipu nám nastaví všechny registry do výchozího nastavení. Jakmile provede inicializaci těchto registrů načte první instrukci z reset vektoru (adresa 0xFFFE:0xFFFF). Zde musí být vložena instrukce skoku JMP na návěští, které nám určuje start programu. Periferie na čipu jsou po resetu vypnuty, maskovatelná přerušení jsou zakázána v kontrolním registru, aby bylo možné nastavit požadované vlastnosti. Ukazatel zásobníku je po resetu nastaven na adresu 0x00FF.

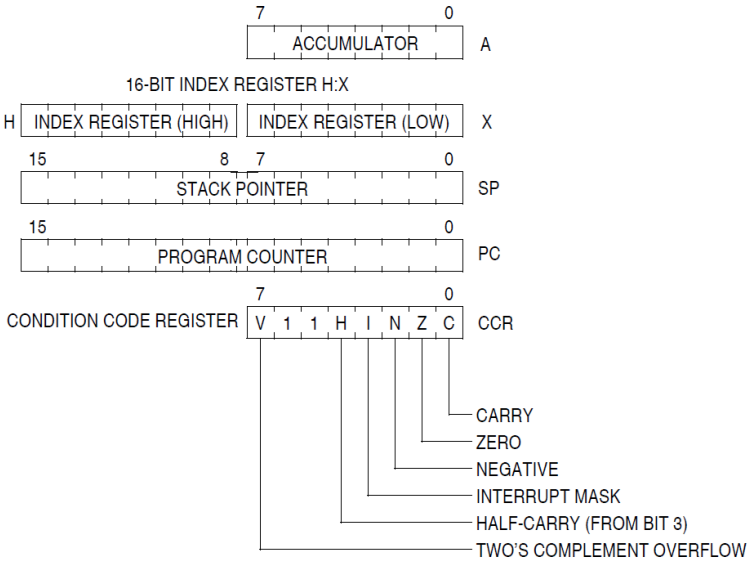
Zdroje resetu:

- vnější RESET pin
- přivedení napájecího napětí (POR)
- nízká úroveň napájecího napětí (LVD)
- modul watchdog (COP)
- chybný kód instrukce (ILOP)
- reset vynucený při ladění programu
- ztráta závěsu generátoru hodin (LOC)

# 2.6 CPU

Procesor HCS08 je navržen tak, aby byl plně kompatibilní s M68HC08. Má jistá vylepšení v podobě instrukcí a adresovacích módů pro zlepšení účinnosti překladačů jazyka C a také nový background modul pro ladění aplikací.

## 2.6.1 Programovací model




Obrázek 2.4 CPU registry

Procesor má pět svých vlastních registrů a klasickou akumulátorovou architekturu. Tyto registry nejsou součástí paměťového modulu, nýbrž samotného CPU.

### Akumulátor (A)

Jedná se o univerzální 8bitový registr. Používá se jako vstupní a i výstupní operand pro ALU jednotku pro aritmetické i logické operace. Lze využít veškeré adresovací režimy, jak pro načtení hodnoty do akumulátoru, tak i pro její uložení. Reset čipu neovlivňuje akumulátor.

### Index registr (H:X)

Index registr je rozdělen na dva samostatné registry 8bitové H a X. Používá se jako 16bitový ukazatel adresy. Vzhledem k zachování kompatibility s předchozí řadou některé instrukce používají jen jeho nižších 8 bitů. Reset čipu automaticky nuluje H část, X nijak neovlivní.

### Stack pointer (SP)

16bitový registr zásobníku LIFO (last-in-first-out). Ukazuje na první volnou položku zásobníku, který roste směrem od vyšší adresy k nižší. Samotný zásobník může být kdekoliv v paměti RAM. Jeho

velikost je omezena pouze velikostí samotné RAM paměti. Automaticky se do něj ukládají návratové adresy při volání podprogramů, návratové adresy a CPU registry během přerušovací rutiny a lokální proměnné.

Hodnota SP registru je automaticky nastavena po resetu na 0x00FF skrz kompatibilitu s předchozí rodinou mikrokontrolerů. U programů pro HCS08 rodinu je dobré změnit hodnotu SP registru na nejvyšší možnou adresu v RAM paměti v závislosti na daném typu čipu.

## Program counter (PC)

16bitový registr obsahují adresu následující instrukce. Během běhu programu je inkrementován. Pokud dojde k přerušení, skokům jak statickému, tak podmíněnému, či návratu z rutin, je jeho hodnota přepsána.

Po resetu je hodnota PC registru automaticky načtena z adresy 0xFFFE:0xFFFF. Jedná se o adresu do tabulky vektorů přerušení pro reset. Zde je uložena adresa první vykonané instrukce po resetu.

## Condition Code Registr (CCR)

Tento registr obsahuje bit maskující globálně přerušení a dále jednotlivé flagy indikující stav výsledku právě proběhlé instrukce.

| bity | popis |
|---|---|
| 7<br>V | Instrukce skoku používají tento flag<br>0 nedošlo k přetečení<br>1 přetečení |
| 6 a 5 | Vždy nastaveny na log 1 |
| 4<br>H | Detekuje přetečení mezi nibly v akumulátoru<br>0 bez carry<br>1 carry |
| 3<br>I | Povoluje globálně veškerá maskovatelná přerušení.<br>0 přerušení povoleno<br>1 přerušení zakázáno |
| 2<br>N | Detekce záporného výsledku operace<br>0 kladný výsledek<br>1 záporný výsledek |
| 1<br>Z | Flag detekující nulový výsledek<br>0 nenulový výsledek operace<br>1 nulový výsledek operace |
| 0<br>C | carry přetečení<br>0 bez carry<br>1 carry |

Tabulka 2.4 popis registru CCR

# 3 Real-time operační systémy (RTOS)

V této části se seznámíme historií rozvoje RTOS a základními pojmy nutnými pro orientaci v dané problematice.

## 3.1 Historie a vývoj RTOS

Počátky real-time operačních systémů se dle [3] datují do 70. let minulého století, kdy se jednalo především o systémy, které byly vytvořeny v domácím prostředí malou skupinou zájemců.

Velký rozvoj přichází v následující dekádě, kdy díky velkému technickému rozvoji mikroprocesorů a jejich použití ve vestavěných systémech, které bylo nastartováno v předešlém desetiletí, docházelo ke stupňování nároků kladených na tyto systémy, a to především dodržení časových mezí pro dané úlohy a zavedení podpory víceúlohového zpracování (multitaskingu). Na začátku 80. let bylo založeno hned několik společností, které se zaměřily na vývoj RTOS, jako Wind Rivers a jejich RTOS systém VxWorks či firma Ready Systems a jejich RTOS VRTX. Velké firmy dokázaly svoje produkty udržet na trhu i s dostatečnou uživatelskou podporou.

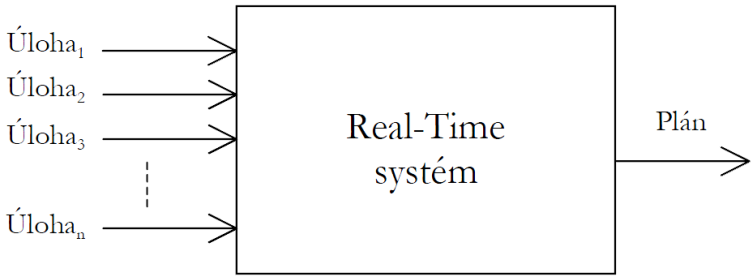
V 90. letech došlo k masivnímu nárůstu počtu RTOS díky malým společnostem, které se specializovaly na úzké oblasti použití svých systémů. Díky takto velkému nárůstu počtu RTOS se snižovala především jejich cena. Velké společnosti začaly ztrácet zájem o své produkty, což vedlo k opuštění 8bitových i 16bitových operačních systémů, a začaly se zaměřovat na 32bitové.

S příchodem nového tisíciletí se situace v oblasti používání RTOS rapidně změnila. RT operační systémy se používají masivně ve vestavěných zařízeních. Vývojáři těchto zařízení začínají hledat levnější alternativy, jako je třeba real-time varianta systému Linux, které jim poskytnou stejnou funkčnost za mnohem menší náklady. Dochází k ústupu velkých firem, které v této oblasti již dlouho působí. Ty se snaží zabránit úpadku zlepšením podpory, nabídkou vývojových kitů a především snížením cen licencí.

## 3.2 Základní Terminologie RTOS

### 3.2.1 Model real-time oparačního systému

Real-time systém spadá to skupiny reaktivních systémů, od kterých se očekává včasná a přesná reakce na podněty, které přicházejí z jeho okolí. Tyto systémy se používají především v oblastech, kde je nutné dodržet časové omezení všech úloh. Na obrázku 3.1 je zobrazen logický model real-time systému, který je založený na plánování úloh reagujících na vstupy systému.




Obrázek 3.1 logický model RTOS

RT systém je systém, který musí v explicitně stanovených mezích splňovat omezení kladená na dobu své odezvy na vstupní podněty nebo riskovat vážné důsledky plynoucí z jejich nesplnění [2].

### 3.2.2 Doba odezvy systému a včasnost

RT systémy nejsou schopny reagovat okamžitě, dochází k jistému zpoždění v čase. Doba od výskytu vstupních podnětů na vstupech systému do provedení požadované odezvy, a to včetně výskytu daných hodnot na všech požadovaných výstupech, se nazývá *doba odezvy systému* [2].

Důležitým pojmem je také *včasnost* odezvy systému. Každá úloha může mít různě stanovenou maximální dobu odezvy. Doba reakce na podněty se může lišit dle aktuálního stavu systému. Systém lze považovat za RT tehdy, pokud všechny doby reakce nepřekročí stanovenou maximální dobu odezvy dané úlohy (*deadline*).

### 3.2.3 Selhání systému

Abychom se mohli bavit o selhání systému, musíme znát všechna časová omezení pro všechny možné úlohy zavedené v systému, pak lze následně definovat selhání. K selhání systému dojde právě tehdy, když systém nedodrží některé z návrhových omezení stanovených ve formální specifikaci systému [2].

### 3.2.4 Dělení real-time systémů

Nejčastěji používané dělení RT systémů v literatuře je rozdělení do tří skupin na systém *soft*, *hard* a *firm*. Velikost maximální doby odezvy systému nijak neovlivňuje zařazení systému. Hard úloha může mít dobu odezvy na příklad minutu, kdežto soft úloha třeba několik milisekund.

#### Soft

Soft RT je systém, jehož výkon je v důsledku nedodržení časových omezení degradovám, ale jehož funkce není tímto nedodržením dotčena. Nedodržení mezí odezev nepovede ke katastrofálnímu výsledku [2].

*Příklad: Nápojový automat. Ten se snaží vydat nápoj co nejrychleji, nicméně časové zdržení má za následek maximálně špatnou náladu kupujícího a krátké zdržení.*

### Hard

Hard RT je systém, u něhož nedodržení jediného časového omezení vede k selhání systému jako celku a má katastrofální následky [2].

*Příklad: Systém aktivace ABS ve vozidle. Zde může mít jakékoliv zdržení systému za následek případnou nehodu či až smrt cestujících*

### Firm

Firm RT je systém, u něhož nedodržení několika časových odezev nevede k selhání systému jako celku, avšak větší počet nedodržení odezev může k tomuto selhání vést a může mít katastrofální následky [2].

*Příklad: Teplotní čidlo. Pokud dojde k opožděné reakci systému náhodně, systém se nepřehřeje. Nicméně pokud dojde k opožděné reakci vícekrát za sebou, hrozí zničení systému přehřátím.*

## 3.2.5 Typy událostí v systému

Události v systému lze dělit dle dvou kritérií, a to na synchronní a asychronní, dále na periodické, aperiodické, případně sporadické.

- Synchronní … výskyt události v systému je předvídatelný
- Asynchronní … výskyt této uládosti nelze v systému předvídat, většinou se jedná o události vyvolané vnějším zdrojem systému.
- Periodické … událost je volána s danou časovou periodou
- Aperiodické … události které se nevyskytují v periodických intervalech
- Sporadické … jedná se o události, které se vyskytují velmi zřídka. Jejich četnost je nekolikanásobně nižší než u událostí aperiodických

## 3.2.6 Determinismus

Systém je deterministický nebo-li „*předvídatelný*“, pokud pro libovolný stav systému a libovolnou množinu vstupních podnětů lze jednoznačně určit následující stav systému a množinu výstupních odezev [2].

Determinismus u RT systému lze ještě dále specifikovat a na událostní a temporální (časový) determinizmus.

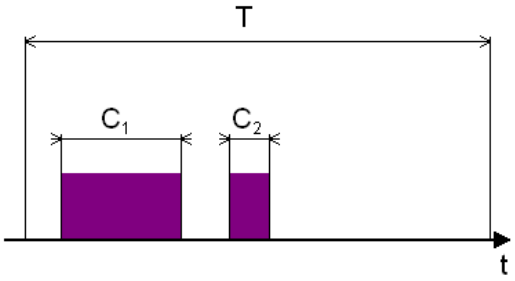
- Událostní … následující stav systému a jeho odezva jsou známy pro libovolnou množinu vstupních podnětů vyvolávajících událost. Událostně deterministický systém obecně deterministický není [2].

- Temporální … v případě že máme deterministický systém a známe doby odezev pro všechny možné množiny úloh, jedná se o temporální (časový) determinismus. Jedná se o významnou vlastnost RT systémů

### 3.2.7 Činitel zatížení CPU

Velmi důležitý pojem při návrhu RT systémů. Vyjadřuje podíl doby, kdy systém v rámci své periody vykonává užitečnou práci ku době periody systému. Vysoká hodnota tohoto činitele je nežádoucí, protože procesor nemusí mít již dostatek volných prostředků pro případnou změnu v systému, což může vést k přetížení a selhání systému. Nízká hodnota tohoto činitele nicméně neznamená kvalitní systém, a to z důvodu, že může být pro jeho realizaci použit zbytečně nákladný hardware.

$$U = \frac{(C_1)+(C_2)}{T}$$




Obrázek 3.2 Činitel zatížení CPU

### 3.2.8 Multitasking

V dnešní době zastaralé operační systémy nepodporovaly souběžné spuštění více úloh současně, což bylo velmi nevýhodné. Obyčejně nemáme k dispozici velké množství výpočetních jednotek, tudíž je multitasking řešen velmi rychlým přepínáním kontextu jednotlivých úloh na procesoru. Čas vyhrazený pro běh jedné úlohy je označován ve většině publikací jako *časové kvantum*.

Podle způsobu přidělování a odebírání těchto kvant lze rozdělit multitasking na dva základní typy preemtivní a nepreemptivní.

#### Preemptivní multitasking

Při použití preemptivního multitaskingu se o přidělování a odebírání procesoru stará samotný operační systém. Ten v pravidelné periodě přeruší svůj aktuální běh, zjistí aktuální stav možných úloh žádajících o přidělení procesoru a jejich priorit, a buď dojde k přerušení aktuální úlohy úlohou s větší prioritou, nebo nechá nadále běžet tutéž úlohu.

Výhodou takto navržených systémů je jejich schopnost vypořádat se zacyklenou úlohou, pokud nemá nejvyšší prioritu v systému. Nevýhodou je složitější implementace samotného jádra operačního systému.

Většina real-time operačních systémů jsou právě preemptivní systémy.

### Nepreemptivní multitasking

Při nepreemptivním multitaskingu uvolňují procesor samotné úlohy. Programátor musí vytvořit aplikaci, aby byla schopna předávat v pravidelném intervalu procesor zpět operačnímu systému. Ten následně vybere buď tu samou úlohu, nebo předá procesor jiné důležitější úloze.

Výhodou takovýchto systémů je velká jednoduchost jejich návrhu a implementace. Jako velká nevýhoda se jeví nutnost korektního naprogramování aplikace. Pokud by byla chyba v předávání řízení, dojde k zahlcení procesoru jednou úlohou a systém není schopen se s tímto vypořádat.

Některé real-time operační systémy jsou nepreemptivní. Lze je použít v případech, kdy je počet obsluhovaných úloh nízký a jednotlivé obsluhy úloh jsou velmi krátké.

## 3.3 Typy jader RTOS

Hlavním prvkem každého operačního systému je samotné jádro. U real-time operačních systémů patří mezi základní funkce jádra především plánování úloh pomocí plánovače s využitím speciálních plánovacích algoritmů, schopnost víceúlohového zpracování (multitaskingu) a zajištění komunikace mezi jednotlivými úlohami. Dále je také důležitá velikost jádra pro jeho samotnou implementaci na daný mikrokontroler.

### 3.3.1 Pseudojádro

Jedná se o typ jádra, který nevyužívá systém přerušení dané architektury, ale následně popsané způsoby pro zajištění schopnosti multitaskingu. Tato jádra poskytují několik výhod a pokud dovolují návrhová omezení aplikace použití pseudojádra, je doporučováno jeho použití.

- ***vyzývací smyčka***

  umožňuje velmi rychlé zpracování události. Jedná se o cyklus, kde je testován příznak dané události, pokud má systém událost zpracovat, dojde k jejímu zpracování a příznak je následně vynulován. Využívá se v případech, že obsluha události je velmi rychlá, tím pádem nemůže dojít k překrytí obsluhy s novým požadavkem na obsloužení.

  Používá se také její upravená varianta nazývaná ***synchronizační vyzývací smyčka***, kde je před samotným zpracováním úlohy vložena časová prodleva, a to z důvodu potřeby ustálení události.

Vyzývací smyčky jsou snadno implementovatelné, nicméně velmi zatěžují procesor, což není vhodné pro složitější RT systémy

```
for (;;) {
        if(udalost_1){
                zpozdeni(time);     //synchronizační vyzývací smyčka
                beh(udalost_1);
                udalost_1 = 0;
        }
        ...
}
```

- ***cyklické provádění***

  jedná se o smyčku, ve které jsou prováděny jednotlivé úlohy za sebou. Problémem může být různorodá délka trvání jednotlivých úloh a jejich synchronizace. Naopak pro stejně dlouhé doby trvání úloh a dobrou synchronizaci je toto pseudojádro vhodné.

```
for (;;) {
        beh(udalost_1);
        beh(udalost_2);
        ...
        beh(udalost_n);
}
```

- ***stavově řízený kód***

  ve své podstatě jde o implementaci stavového automatu aplikace. Výhodou je možnost dělit události na přibližně stejně trvající části a také to, že stavové automaty mají formální matematickou definici a lze její pomocí docílit dodržení maximálních dob odezev. Nicméně ne všechny úlohy jsou řešitelné pomocí stavových automatů.

```
for (;;) {
        switch(stav){
                case 1:
                        beh(udalost_1); break;
                ...
                default;
        }
}
```

- ***spolupracující úlohy***

  systém založený na rozdělení jednotlivých úloh na přibližně stejně dlouhé části zpracovávané stavově řízeným kódem. Po částech úloh dochází k vyvolání plánovače, který vybere následující část a případně uloží potřebná data, pokud dojde k přepnutí mezi jednotlivými úlohami. Vyvolání plánovače má zcela ve své režii programátor.

### 3.3.2 Jádro využívající přerušení

Další možností jak implementovat v jádře plánovač úloh je využití systému přerušení dané platformy. V tomto případě je pak možné přepínat mezi úlohami pouze díky vyvolání přerušení, a to způsobeného jak HW nebo SW. Při příchodu přerušení dojde k vyhodnocení jeho priority a případně ke spuštění úlohy s vyšší prioritou, než je priorita úlohy aktuálně běžící.

Při použití této metody je nutné se důkladně seznámit s HW přerušení, programovou obsluhou přerušení i způsobem uložení dat při přerušení na dané platformě. Při přepínání úloh dochází k uložení kontextu právě přerušené úlohy. Objem ukládaných dat se liší v závislosti na platformě a zpravidla se ukládá na zásobník. Samotný HW ukládá pouze data samotného procesoru jako je programový čítač, stavové slovo, akumulátor a případně další. U různých výrobců v zájmu zachování kompatibility svých produktů kolikrát nedojde ke kompletnímu uložení dat nutných k přepnutí kontextu a programátor musí následně uložit data sám, případně uložit i datové struktury, o které nechce přijít.

Dalším problémem může být přerušení samotných rutin přerušení a případných kritických sekcí v programu. Zde je nutné přerušení vypnout, aby nedošlo k přerušení v době zpracování kritické části programu či přerušení. Obecně je doporučeno volit jak obslužné rutiny přerušení, tak i části kritických sekcí co nejkratší, aby nemohlo dojít k případnému přerušení jejich běhu jiným přerušením.

Výhodou takovýchto systémů je přehlednost zdrojových kódů aplikace a krátká doba odezvy systému. Nicméně tento systém má i své nevýhody v podobě plýtvání výpočetního výkonu při čekání v hlavní smyčce.

### 3.3.3 Systémy pracující v pozadí/popředí

Jedná se o vylepšený model jádra využívajícího přerušení. Vylepšení spočívá ve využití času stráveného čekáním pro tzv. *úlohy běžící v pozadí.* Jedná se o úlohy s velmi nízkou prioritou vykonávající užitečné funkce pro systém jako je například ovládání displeje. Tyto úlohy mají minimální kritičnost oproti úlohám, které jsou zpracovány pomocí přerušení. Ty jsou označovány jako *úlohy běžící v popředí*.

Tyto systémy mají specifickou inicializaci:

- zakázání přerušení
- nastavení vektorů přerušení pro úlohy v popředí a inicializaci zásobníku
- další nastavení systému jako jsou periferie na čipu, inicializace datových struktur atd.
- povolení přerušení

### 3.3.4 Task control block model (TCB)

V dnešní době velmi populární model, který má své využití u real-time systémů, kde dochází za běhu ke změně počtu úloh. Každá úloha zavedená do systému má svou datovou strukturu zvanou *task control block*, jenž obsahuje údaje nutné k identifikaci úlohy, její kontext, prioritu, atd.

Samotné jádro si uchovává tyto TCB bloky většinou ve formě seznamu, dále má speciální seznamy, ve kterých jsou např. úlohy připravené k běhu a ze kterých vybírá plánovač systému či zablokované úlohy.

| ID |
|---|
| priorita |
| stav |
| Registr 1<br>...<br>Registr n |
| PC |
| Ukazatel na další TCB |

Tabulka 3.1 Struktura task control bloku [2]

## 3.4 Základní plánovací mechanismy

Každý víceúlohový systém potřebuje ke své správné činnosti plánovač, který využívá různé strategie výběru následně běžící úlohy na procesoru. U systému reálného času dochází k problému sdílení výpočetního času počítače mezi více úlohami při zajištění jejich vzájemné nezávislosti [4].

Mezi nejznámější strategie plánová patří následující:

- postupné plánování - *FCFS*
- s cyklickou obsluhou - *Round-Robin*
- plánování podle délky běhu procesu - *SJF*
- prioritní plánování
- víceúrovňové systémy plánování využívající fronty

### 3.4.1 Postupné plánování *First Come First Served (FCFS)*

Jedná se o nejjednodušší plánovací mechanismus, který využívá jednu frontu příchozích požadavků FIFO. Prvně příchozí úloha do fronty je zpracována bez ohledu na ostatní později příchozí úlohy. Jakmile aktuálně běžící úloha dokončí svůj běh na procesoru, je z fronty vybrána následují úloha, která do ní byla zařazena hned po předchozí běžící. Tento systém není preemptivní – úloha se sama nevzdává procesu, k uvolnění procesoru dojde až po doběhnutí běžící úlohy.

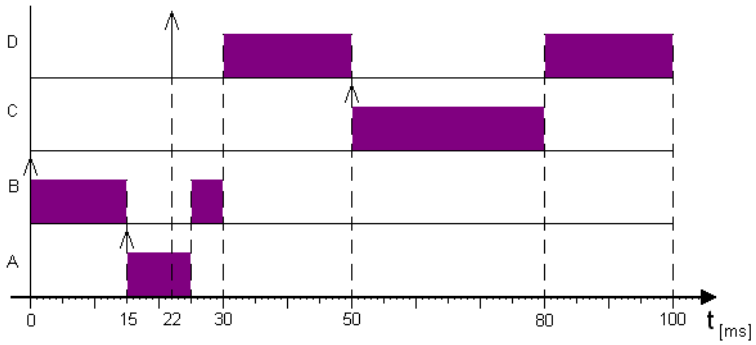
Jako příklad uveďme čtyři úlohy A, B, C a D a jejich doby zpracování 10, 20, 40 a 30 ms. V případě, že se vyskytnou úlohy v systému ve výše zmíněném pořadí z rozestupem 1 ms, bude průměrná doba čekání 26,25 ms a doby čekání jednotlivých úloh (0, 9, 28 a 68 ms). Pokud by do systému vstoupily úlohy v pořadí D, C, B a A se stejný rozestupem 1ms, bude průměrná doba čekání mnohem větší, a to 46 ms (doba čekání D – 0, C – 29, B – 68 a A – 87 ms).

Doba čekání jednotlivých úloh zde není minimalizována, její rozptyl je dán pořadím vstupních úloh systému a jejich dobou vykonávání. S dobou čekání je úzce spjatá i doba odezvy systému – tento mechanizmus není příliš vhodný pro RT aplikace. Nicméně se dá využít v systémech, kde známe předem počet úloh, jejich dobu trvání a jejich doby odezev. Velkou výhodou je velmi jednoduchá implementace tohoto systému.

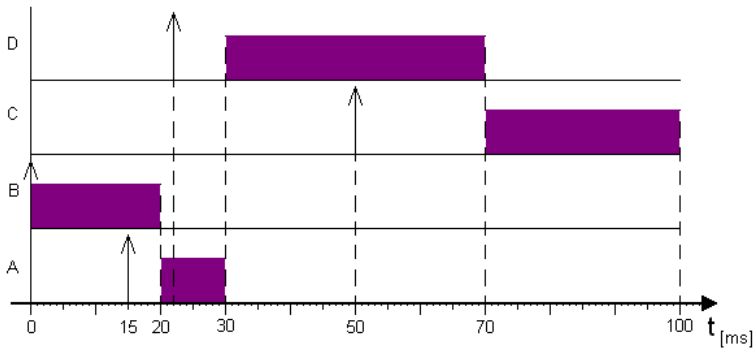
### 3.4.2 Plánování podle délky procesu *Shorted Job First (SJF)*

Plánovač při zavedení nové úlohy provede aktualizaci fronty – zařadí danou úlohu do fronty dle její doby zpracování. Úlohy ve frontě jsou řazeny od nejkratších po nejdelší dobu provádění. Procesor dostane přidělnou úlohu s nejkratší dobou provádění ve frontě.

Tento systém může být jak preemtivní, tak i nepreemptivní. Na následujících obrázcích 3.3 a 3.4 je zobrazeno plánování úloh A,B,C a D se stejnou dobou výpočtu jako v předchozím příkladu preemptivním a nepreemptivním způsobem. Doby zavedení úloh do systému jsou následující: A – 15, B – 0, C – 50 a D – 22 ms.




Obrázek 3.3 preemptivní plánování podle délky úlohy




Obrázek 3.4 nepreemptivní plánování podle délky úlohy

Z obrázku 3.3 je patrné, že preemptivní plánování zkracuje dobu čekání na zahájení úlohy. Nicméně v případě úlohy D znatelně prodlužuje její dobu odezvy v systému, což může vést k porušení maximální doby odezvy.

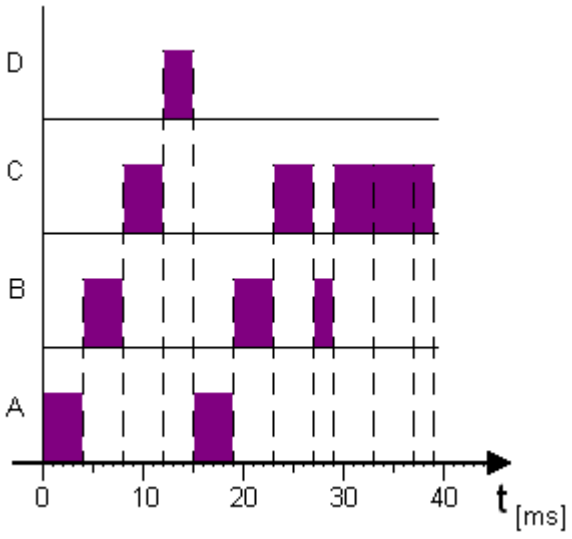
### 3.4.3 Plánování s cyklickou obsluhou *Round-Robin*

Jedná se o preemptivní systém podobný systému FCFS. Systém přiděluje procesor jednotlivým úlohám ve frontě po přesně specifikovaný čas nazývaný *časové kvantum*. To je definováno přímo v operačním systému, většinou v intervalu 1 – 100 ms. Po uplynutí tohoto času dojde k přepnutí kontextu na další úlohu ve frontě. Pokud aktuálně běžící úloha ukončí svůj běh dříve, nečeká se na doběhnutí časového kvanta, ale plánovač ihned po doběhnutí úlohy spustí další úlohu ve frontě. Pokud úloha potřebuje více času na své dokončení, než je stanovené kvantum, plánovač úlohu přeruší a spustí následující úlohu ve frontě. V dalším cyklu dostane přerušená úloha opět časové kvantum pro svůj běh.

Problémem této metody je stanovení velikosti časového kvanta pro systém. Obecně by se zdálo, že snižováním hodnoty časového kvanta dosáhneme lepších výsledků, avšak v reálu tomu tak není. S přepnutím kontextu procesoru je většinou spjato velké množství uložení dat pro pokračování výpočtu v místě přerušení.

Doba přepnutí kontextu se pohybuje v řádu desítek až stovek mikrosekund a závisí na použitém operačním systému a použitém hardware [4].

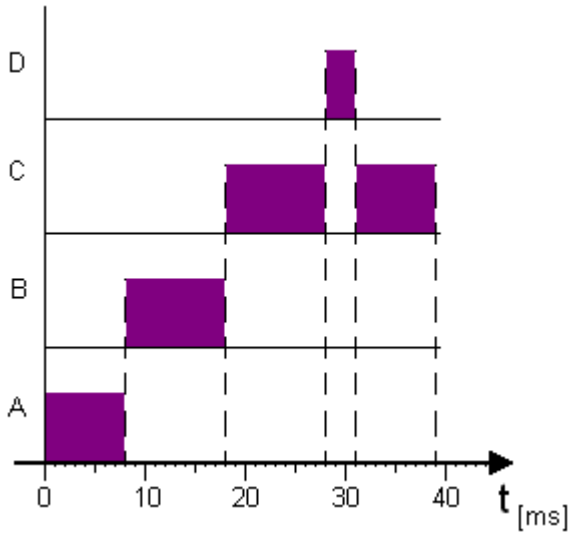
U real-time operačních systémů se plánování round-robin používá pro plánování úloh se stejnou prioritou ve víceúrovňové strategii. Je výhodnější než plánování first come first served, protože právě díky zavedení preemprivnosti dokáže rovnoměrněji obsluhovat čekající úlohy se stejnou prioritou a díky jejich přepínání lépe zajišťuje dodržení maximálních dob odezev.




Obrázek 3.5 plánování Round-Robin, kvantum = 4 ms

Na obrázku 3.5 je zobrazeno plánování round-robin s časovým kvantem 4 ms pro množinu úloh s následujícími délkami trvání jejich běhu: A – 8, B – 10, C – 18 a D – 3 ms.

V tomto případě dochází k častému přepínání kontextu úloh. Je to způsobeno malým časovým kvantem pro dané úlohy. Obecně se doporučuje volit kvantum jako průměr průměrných dob trvání všech úloh v systému, případně pokud jsou v systému úlohy buď několikanásobně kratší či delší a jejich počet je velmi malý, tyto úlohy vynechat při výpočtu. Na obrázku 3.6 je zobrazeno plánování úloh s časovým kvantem 10 ms (horní zaokrouhlení průměru dob trvání všech úloh). Zde je přepínání kontextu méně časté, tudíž dochází ke snížení režie s tím spojené.




Obrázek 3.6 plánování Round-Robin, kvantum = 10 ms

### 3.4.4 Prioritní plánování úloh

Až doposud využívaly všechny plánovače časových informací o délce daných úloh zaváděných do systému. Ovšem reálné systémy mají různě důležité úlohy. Proto je nutné zavést priority úloh do jejich zpracování. Obecně vzato se dá považovat i systém SJF za systém s prioritou, kde nejvyšší priorita je přiřazena úloze s nejkratším časem vykonávání.

Priorita v systému se dá chápat jako interní či externí. Externí prioritu nastavuje úlohám samotný programátor. Pro výpočet interní priority se využívají měřitelné či počitatelné veličiny. Dle [4] to jsou například: Časový limit, paměťové nároky, počet otevřených souborů, případně poměr operací procesoru a I/O operací.

Dále lze prioritu chápat jako statickou, a to v případě, kdy se po celou dobu zavedení úlohy v systému nemění, nebo dynamickou. Dynamické změny priority využívají různých strategií a algoritmů. Dle [4] může docházet k dynamickým změnám priorit v následujících případech: Při enormním zatížení systému mohou být úlohy s nižší prioritou odstaveny, dlouho neobsluhovaným úlohám je postupem času priorita zvyšována (technika zvaná *aging*).

### 3.4.5 Plánování pomocí více front

Jedná se o plánovač sestávající se z více úrovní (front). Každá úroveň má svoji vlastní frontu, která je plánována pomocí některých výše uvedených systémů. Fronta představuje úlohy na stejné prioritní hladině. Jednotlivé fronty mají také své priority, aby nedocházelo k záměně front při plánovacím algoritmu. Pokud je fronta nejvyšší úrovně prázdná, přejde plánovač o úroveň níže a vybere úlohu nižší priority.

Jednotlivé fronty lze rozšířit o vnitřní priority úloh, jež se v nich nacházejí. Jedná se jen o další zjemnění prioritního systému, například pro systémy, kde může být velké množství úloh v jedné frontě a my potřebujeme i mezi tyto úlohy zavést priority.

Další možností jak prioritně určovat jaké úlohy budou na procesoru spouštěny, je procentuální dělení doby běhu jednotlivých front na procesoru.

Obecně je systém plánování pomocí více front preemptivní, nicméně záleží na jeho konkrétní implementaci. Každá fronta může být plánována různým plánovacím algoritmem.

## 3.5 Komunikační a synchronizační prostředky

Ve většině víceúlohových real-time aplikací se setkáváme s úlohami, které jsou na sobě nějak závislé, potřebují spolu komunikovat, případně kooperovat na výpočtu.

V následující podkapitole se zmíním o prostředcích sloužících právě ke komunikaci a synchronizaci, jsou například globální proměnné, datové vyrovnávající paměti a poštovní schránky.

### 3.5.1 Globální proměnné

Jednou z nejjednodušších a z hlediska rychlosti komunikace nejefektivnější možností je použití globálních proměnných. Ačkoliv tento styl programování není plně v souladu se zásadami návrhu kvalitního SW díla, je přesto často používán zejména tam, kde je hlavním cílem maximalizovat výkon výsledné aplikace. V oblasti synchronizace je možno globální proměnné využít např. k zamykání přístupu do kritické sekce programu [2].

Programátor si ale musí být vědom možnosti, že může dojít k přepsání údajů v globálních proměnných, a to v případě, že dojde k přerušení běhu úlohy úlohou s vyšší prioritou a právě tato úloha taktéž pracuje s danými globálními proměnnými.

### 3.5.2 Datové vyrovnávací paměti

Jedná se o klasický buffer pro data na straně producenta. Ve většině případů aplikací nedochází ke stejným rychlostem zápisu dat na straně producenta a čtení na straně konzumenta. Úloha produkující

data je ukládá do bufferu, úloha zpracovávající data je postupně svou rychlostí čte a zpracovává. Pokud dojde k naplnění bufferu, musí produkující úloha čekat na jeho uvolnění vyčtením dat úlohy pro zpracování dat. Důležité je dobře volit velikost bufferu v dané aplikaci.

Často se využívá i mechanismus *zdvojení bufferů,* a to pro úlohy, kde je potřeba předávat časově souvislá data, jako jsou například jednotlivé snímky v grafických kartách, či data naměřená čidly systému pro jeden určený okamžik.

Odstranění komplikací způsobených řízením zdvojených bufferů nám přináší kruhový buffer. Jeho velkou výhodou je umožnění souběžného čtení ze začátku a zápisu na konec bufferu v aplikacích, kde se vyskytují více než dva producenti či konzumenti dat. Jako ochrana k nechtěnému přepisu dat uložených v této struktuře je používán semafor o velikosti daného bufferu

### 3.5.3 poštovní schránky *mailboxes*

Většina dnešních real-time jader implementuje poštovní schránky jako předem definované paměťové místo, které může využívat pro svou synchronizaci či komunikaci více úloh. Obecně lze předávat do schránek libovolná data, případně ukazatel na datovou strukturu. Při operací zápisu a následně čtení je pak nutné, aby souhlasily datové typy předávaných dat. Velkou výhodou tohoto systému je, že čekající úloha nemusí aktivně testovat obsah schránky (polling) a tudíž nezatěžuje procesor aktivním čekáním. Operační systém úlohu uspí, dokud se nevyskytnou ve schránce data. Schránka má implementovanou frontu čekatelů pro tyto účely.

Různé operační systémy podporují různé varianty schránek, jako například schránka s časovým limitem pro čekání na data, čtení ze schránky, i když je schránka prázdná s chybovým návratovým kódem atd.

### 3.5.4 Semafory

Semafor je nejčastěji používaný systém pro synchronizaci dvou a více úloh. Jedná ze o klasický zámek, který je implementovaný jako *binární* či *čítací*.

Binární semafor nabývá hodnot *true* a *false* značících, zdali je úloha v kritické sekci či nikoliv. Pro vstup do kritické sekce slouží operace zamčení semaforu, po provedení kritického kódu je volána operace odemčení semaforu. Operace zamčení a odemčení jsou atomické, což zaručuje jádro operačního systému.

Čítací jsou semafory, které mohou nabývat více než dvou hodnot. Tento typ semaforu má ještě dvě další atomické operace, a to jeho inkrementaci a dekrementaci pro obě dvě možnosti použití.

### 3.5.5 Fronty zpráv

Jsou implementovány jako pole poštovních schránek pro zasílání více zpráv. V závislosti na implementaci v operačním systému může být tato fronta typu FIFO (data jsou načtena v pořadí jejich zápisu), anebo i LIFO (data jsou čtena v opačném pořadí, posledně zapsaná jsou čtena jako první).

## 3.6 Dnešní Real-time operační systémy

V dnešní době není prakticky možné přesně určit počet real-time operačních systémů. Po roce 2000 došlo k velmi velkému rozvoji v této oblasti, kdy si mnohé společnosti sami vytvářely vlastní systémy, protože v té době komerční řešení bylo velmi nákladné a nevýhodné. Tyto jejich systémy jsou v některých případech zveřejněné podléhající komerční licenci, nicméně zcela určitě se najdou i společnosti, které mají vlastní real-time operační systém, který je tajný.

### 3.6.1 Známé real-time operační systémy

Přehled nejznámějších výrobců real-time operačních systémů působících v této oblasti již mnoho let včetně jejich produktů:

- Micrium – uC/OS ... verze 2, nejnovější verze 3
- Autostar – OSEK
- QNX – QNX
- WindRiver – VwWorks
- Microsoft – Windows CE
- LinuxWorks – LunxOS
- Freescale – MQX

### 3.6.2 Volně dostupné real-time systémy (GPL licence)

Přehled GPL real-time operačních systémů:

- uOS
- cocoOS … přechod na BSD licenci, starší verze pravděpodobně zůstanou GPL
  http://www.cocoos.net/
- RTLinux … http://www.rtlinuxfree.com/
- FreeOSEK ... http://opensek.sourceforge.net/
- eCos … http://ecos.sourceware.org/
- FreeRTOS … http://www.freertos.org/

# 4 µC/OS-II

Jedná se o druhou verzi systému, který byl vytvořen v roce 1992 Jeanem J. Labrossem. Od té doby se sytém rozšířil do všech druhů aplikací, jako jsou například lékařské přístroje, síťové adaptéry, automatizační technika, průmyslové roboty a další. Systém jako takový také našel uplatnění při výuce na mnoha univerzitách po celém světě.

µC/OS-II je zpětně kompatibilní systém s jeho první verzí, navíc je přidána podpora řízení paměti, mazání úloh systému (pokud se jedná o dymamické úlohy), podpora rozšíření task control blocku, kompletní správa zásobníku a mnohé další. Dalším vylepšením oproti předchozí verzi je mnohem snadnější portace pro různé platformy od různých výrobců. Samotný zdrojový kód je rozděnely do několika souborů pro jednodušší orientaci.

V následující části se seznámíme se základními vlastnostmi systému µC/OC-II, činností a strukturou jeho jádra, potřebnými datovými strukturami pro běh operačního systému, prostředky pro komunikaci a synchronizaci úloh a dalšími důležitými pojmy.

Jedná se o velmi stručný přehled. Pro důkladné pochopení daných vlastností, mechanismů datových struktur a jiných pojmů tohoto systému slouží kniha autora [5], která je zároveň i uživatelskou příručkou tohoto systému.

## 4.1 Základní charakteristika systému

### Přenositelnost

většina zdrojových kódů systému je psána v jazyku ANSI C, jen velmi specifické části jsou pak psány v jazyku symbolických adres (assembleru) dané platformy, pro kterou se vytvářejí aplikace (jedná se o portaci systému na danou platformu). Tato část kódu v assembleru je minimalizována, aby se dalo velmi jednoduše přecházet mezi platformami. Stejně jako předchozí verze je µC/OS-II portovatelný na velké množství mikrokontrolerů od různých výrobců (podporované architektury jsou na stránkách firmy Micrium). Překladač také podporuje vkládání assemblerových instrukcí pro například povolování přerušení či jiné specifické nastavení architektury. Systém lze implementovat na většině 8bitových, 16bitových, 32bitových ale i 64bitových architekturách.

### Schopnost uchovat systém v paměti čipu

µC/OS-II je navržen jako systém pro vestavěné aplikace. Pokud jsou k dispozici vývojářské nástroje, je možné systém nahrát do paměti ROM na dané architektuře a následně jej distribuovat s produktem.

### Škálovatelnost

Systém je navržen tak, že umožňuje programátorovi určit, které části systému chce využívat ve své aplikaci. Tato vlastnost je velkou výhodou, protože umožňuje ušetřit paměť na čipu, případně může vývojář přejít na levnější platformu, která má menší paměti RAM i ROM.

### Preemptivnost

mC/OS-II obsahuje plně preemptivní jádro. To zaručuje přerušení nížeprioritní úlohy v systému, pokud se vyskytne úloha s vyšší prioritou než je aktuálně běžící.

### Víceúlohovost

Jádro sytému poskytuje možnost vytvoření až 64 úloh v systému. Osm z nich je rezervováno systému, tudíž má uživatel k dispozici 56 zbylých úloh. Každá úloha v systému má specifickou prioritu. Nelze mít dvě úlohy se stejnou prioritou – μC/OS-II nepodporuje Round-Robin plánování.

### Determinističnost

Doba výpočtu všech funkcí systému je deterministická. Vždy víme, jak dlouho bude dané volání funkce trvat. Existuje jedna jediná výjimka v systému, kdy záleží doba běhu funkce na počtu úloh zavedených do systému.

### Zásobníky úloh

Každá úloha zavedená do systému má svůj vlastní zásobník. μC/OS-II umožňuje specifikovat velikost zásobníku pro každou úlohu zvlášť. Tato funkce umožňuje další snížení paměťových požadavků na platformu. Systém disponuje utilitou pro zjištění velikosti zásobníku pro jednotlivé úlohy pro její přesné nastavení.

## 4.2 Architektura jádra systému

Samotné jádro operačního systému se skládá ze 14 souborů. Každý soubor obsahuje specifickou část systému. Systém byl takto rozdělen autorem, aby byla snažší orientace ve zdrojových kódech a jednodušší portace systému na jednotlivé platformy různých výrobců.

### 4.2.1 Kritické sekce

Obecně potřebujeme zajistit, aby běh kritické sekce nebyl přerušen kteroukoliv jinou úlohou systému. Toho dosáhne μC/OS-II globálním zakázáním přerušení. Nicméně čas, po který je v systému vypnuto

přerušení, závisí velmi na zvolené platformě. Každá architektura má svou vlastní instrukci pro zapnutí a vypnutí přerušení. Pro skrytí používání této instrukce jsou definována dvě makra sloužící ke vstupu (zakázání přerušení) a výstupu (opětovné povolení přerušení) z kritické sekce v programu. Jedná se to tato makra: `OS_ENTER_CRITICAL()`, `OS_EXIT_CRITICAL()`. Nastavení těchto maker je platformě specifické.

### 4.2.2 Úlohy systému

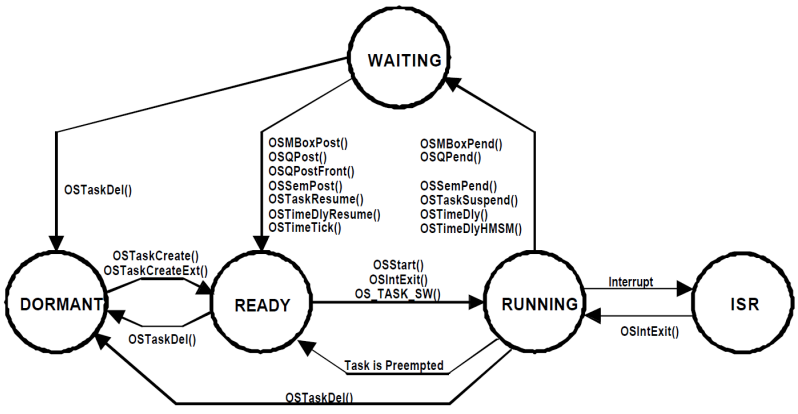
Úloha systému je klasickou funkcí v jazyce C. V jejím těle je nekonečná smyčka například pomocí cyklu *for* či *while* a návratový typ úlohy je vždy *void*. Typická úloha systému vypadá dle [5] takto:

```
void YourTask (void *pdata)
{
      for (;;) { (2)
            /* kod ulohy */
            OSMboxPend(); OSQPend(); OSSemPend();
            OSTaskDel(OS_PRIO_SELF);
            OSTaskSuspend(OS_PRIO_SELF);
            OSTimeDly(); OSTimeDlyHMSM();
            /* kod ulohy */
      }
}
```

Datový ukazatel na data předávaný do úlohy slouží jako univerzální ukazatel do paměti. Lze přes něj předávat libovolné datové struktury jednotlivým úlohám. Vždy je lepší mít jednu jedinou úlohu s jedním kódem a předanými daty specifikovat, např. S kterým portem pracuje, než mít v systému zavedeny úlohy pro všechny porty zvlášť.

Systém má dvě vlastní úlohy, jedná se o *idle* úlohu běžící při nečinnosti systému a statistickou *stat* úlohu, kterou lze v konfiguračním souboru povolit či zakázat.

Stavy úlohy




Obrázek 4.1 stavový automat úlohy v systému µC/OS-II [5]

Na obrázku 4.1 je zobrazen stavový automat úloh v systému μC/OS-II. Úloha je do systému zavedena voláním `OSTaskCreate()`, případně `OSTaskCreateExt()`. Takto vytvořená úloha přejde do stavu připravena pro běh – *READY*. Vytvoření úloh v systému je možné před spuštěním víceúlohového plánování či dynamicky až za jeho běhu. Pokud má nově vytvořená úloha větší prioritu, je jí ihned operačním systémem předán procesor.

Samotné víceúlohové zpracování je spuštěno příkazem `OSStart()`. Ten vyvolá plánovač, který vybere úlohu s nejvyšší prioritou, která je v *READY* stavu. Běžící úloha přejde do stavu *RUNNING*. V tomto stavu může být v systému pouze jedna úloha.

Běžící úloha může přejít do stavu *WAITING* při čekání na příchozí událost. Jakmile úloha přejde do stavu *WAITING,* je spuštěna další úloha s nejvyšší prioritou. Čekající úloha přejde do stavu ready, jakmile přijde požadovaná událost.

Běžící úloha může být kdykoliv přerušena, pokud není globálně zakázáno přerušení. Pokud je v systému jiná úloha s vyšší prioritou, dojde k přerušení běhu a spuštění této úlohy, jinak je procesor zpět přidělen přerušené úloze.

### Idle úloha

Jedná se o úlohu s nejnižší prioritou v systému μC/OS-II. Tato úloha je zavedena do sytému vždy a její činností je inkrementace 32bitového čítače. Tento čítač je použit následně ve statistické úloze ke spočítání využití procesoru danou aplikací. Samotná implementace této základní úlohy je řešena pomocí kritické sekce, protože inkrement 32bitové hodnoty na 8bitových či 16bitových procesorech není otázkou jedné instrukce.

### Statistická úloha

Úloha shromažďující údaje o běhu aplikace, jako je využití procesoru v procentech. Statistická úloha se povoluje nastavením příslušné proměnné při konfiguraci systému (soubor *os_cfg.h*). Pokud chce programátor použít statistickou úlohu, je nutné, aby byla zavedena do systému jejím voláním z jediné úlohy zavedené v sytému před vytvořením dalších úloh aplikace. Jedná se o úlohu s druhou nejnižší prioritou v systému hned po `IdleTask` úloze

## 4.2.3 Task Control Block (TCB)

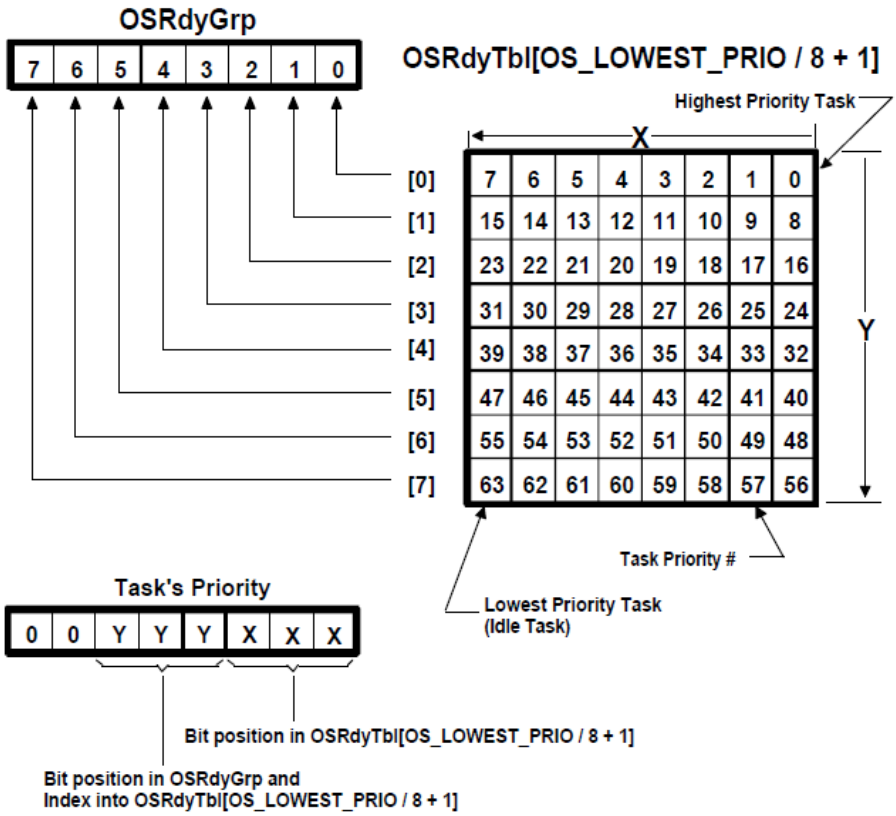
Jedná se o datovou strukturu, která je přiřazena úloze při jejím vzniku v systému. Tato struktura slouží v systému k zachování kontextu přerušené úlohy. Jakmile tato úloha získá zpět procesor je pomocí bloku řízení úlohy zajištěno pokračování úlohy v místě, kde byl její výpočet přerušen. Všechny tyto bloky jsou uloženy v paměti RAM na čipu.

Implementačně nejdůležitější je hned první položka TCB, a to je ukazatel na vrchol zásobníku dané úlohy, dále se v bloku nachází priorita úlohy, velikost TCB či ukazatel na poslední validní adresu zásobníku úlohy. Detailní popis bloku řízení úlohy je v [5].

### 4.2.4 Ready List

Jedná se o datovou strukturu, která má pro každou úlohu uloženy dvě proměnné, a to `OSRdyGrp` a `OSRdyTbl[]`. 64 prioritních úrovní je rozděleno do skupin po osmi úlohách. Pokud je kterákoliv úloha z dané osmice připravena k běhu, je nastaven příslušný bit pro danou osmici v `OSRdyGrp`, také je nastaven příslušný bit v `OSRdyTbl[]`. Velikost `OSRdyTbl[]`a tím i celého Ready Listu může programátor ovlivnit. Tím lze ušetřit další část paměti RAM na čipu.

Na obrázku 4.2 je znázorněn princip výběru úlohy s nejvyšší prioritou. Nejnižší tři bity z priority úlohy jsou použity k výpočtu pozice v proměnné `OSRdyTbl[]`. Nejvyšší tři bity z priority úlohy jsou použity k výpočtu indexu k `OSRdyTbl[]`.




Obrázek 4.2 systém výběru úlohy s nejvyšší prioritou [5]

### 4.2.5 Plánování úloh

µC/OS-II plánovač vždy vybírá úlohu s nejvyšší prioritou. Pokud je plánování vypnuto či v případě obsluhování rutiny přerušení (ISR), nedochází k plánování. V opačném případě najde plánovač úlohu s nejvyšší prioritou a se stavem *READY*. Pokud je to právě běžící úloha, tak se plánovač ukončí, aby nedocházelo ke zbytečnému přepnutí kontextu, které také trvá jistý čas.

Samotný kód plánovače je v systému µC/OS-II implementován jako kritická sekce, která nemůže být přerušena za žádných okolností.

Samotné přepnutí kontextu se skládá z uložení registrů procesoru na zásobník přerušované úlohy a obnovení těchto registrů ze zásobníku úlohy s vyšší prioritou.

### 4.2.6 Přerušení v µC/OS-II

µC/OS-II vyžaduje napsání rutiny přerušení v assembleru díky různorodosti cílových platforem. Některé specifické překladače podporují vkládání asemblerových kódů přímo do zdrojových souborů psaných v ANSI C.

Rutina přerušení začíná uložením registrů procesoru na zásobník. O to se většinou postará samotný čip, nicméně jak bylo zmíněno v části 2.5.1, u čipu JM60 nedochází k uložení všech registrů. Následně je inkrementována proměnná pro počet zanoření přerušení. Nyní je prostor pro obsluhu daného přerušení. Po obsluze je zpětně dekrementována proměnná pro počet zanoření, obnoveny registry procesoru (pokud jsme museli dodatečně ukládat nějaké registry, tak je ve správném pořadí načteme) a nakonec je volána instrukce pro návrat z přerušení pro obnovu registrů, které byly automaticky uloženy na dané architektuře.

### 4.2.7 Vnitřní čas systému µC/OS-II

Jako každý operační systém potřebuje i tento vnitřní generování přerušení ve specifikovaném časovém intervalu Tento *clock tick* bývá nastaven v rozmezí 10 až 100 ms. Samotné nastavení se provádí v konfiguračním souboru operačního systému. Čím vyšší bude frekvence generování přerušení, tím více se bude projevovat režie nutná s jeho zpracováním. To může mít důsledek v prodlužování doby odezvy aplikace.

Samotná obsluha přerušení časovače je stejná jako byla popsána výše. Místo uživatelského kódu pro obsluhu je zde ale volána funkce jádra systému, a to `OSTimeTick()`. Tato funkce také slouží k měření času běhu systému. Ten je přístupný uživateli pomocí proměnné `OSTime`.

## 4.3 Implementační nároky systému μC/OS-II

Díky škálovatelnosti a možnosti nahrát jádro systému do paměti ROM je možné minimalizovat nároky na paměť RAM na čipu nahráním minimalizovaného jádra systému právě do paměti ROM (na čipu paměť flash – 60 KB). Čip JM60 disponuje pamětí RAM o velikosti 4 KB, což může být pro složitější aplikace, které využívají větší počet úloh a mnoho funkcí jádra systému, nedostačující, pokud by se měly uložit právě do paměti RAM. Nemluvě o tom, že jednotlivé úlohy v systému také vyžadují část paměti.

Každá úloha zavedená do systému má svůj vlastní řídící blok, svůj vlastní zásobník a také lokální proměnné. Struktura řídícího bloku je co se velikosti týče omezena shora, nicméně velikost zásobníku omezena není, tudíž je důležité být při volbě jeho rozsahu velmi opatrní. Lokální proměnné také nejsou nijak omezeny, záleží jen a jen na programátorovi, kolik paměti potřebuje využít.

Dále také systém zabere jeden časovač na čipu pro generování systémového tiku. Tomu se lze vyhnout v případě, že bychom potřebovali pro běh aplikace oba dva časovače, přivedením externího zdroje periodického přerušení na příslušný pin na čipu.

# 5 Uživatelská příručka

Tato uživatelská příručka slouží k popisu nezbytných kroků a činností nutných k zprovoznění jádra real-time operačního systému µC/OS-II na platformě JM60. Popisuje postupy instalace software a ovladačů, vytvoření projektu v prostředí CodeWarrior Development studio for Microkontrollers 6.1 (dále jen CodeWarrior), portaci systému na daný čip i tvorbu samotné uživatelské aplikace. Příručka je psána chronologicky, místy jsou zmíněny tipy pro práci, které usnadňují danou činnost.

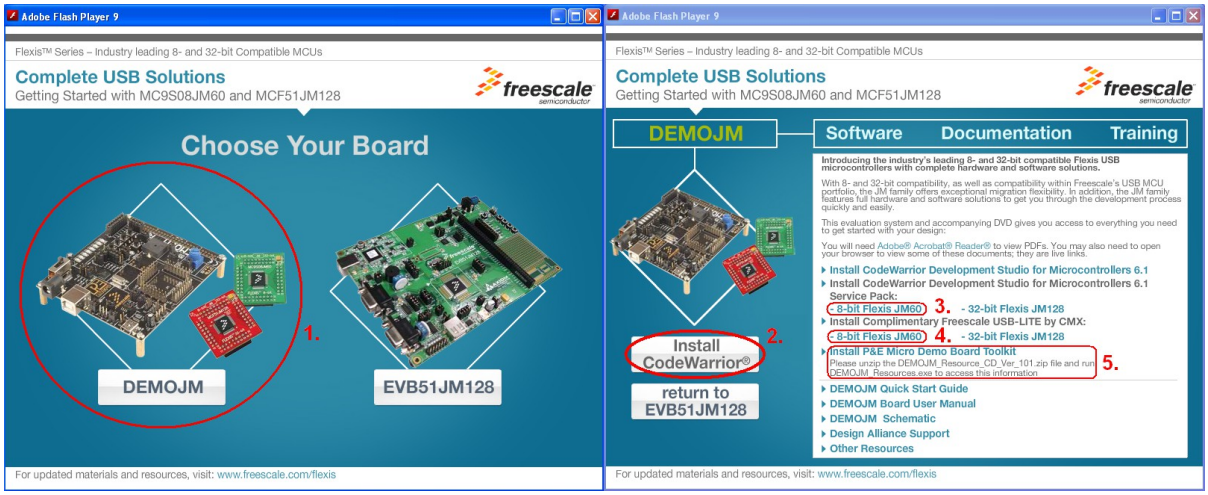
Před samotnou instalací potřebného software a ovladačů si stáhněte zdrojové soubory real-time operačního systému µC/OS-II, ty jsou dostupné po bezplatné registraci na stránkách www.micrium.com. V sekci *download* vyberte dle produktu právě *µC/OS-II(Kernel),* aktuální nejnovětší verze zdrojových souborů je 2.90. Zde **doporučuji stáhnout** i **uživatelskou příručku** *µC/OS-II: The Real-Time Kernel, 2nd Edition* [5]. Nedoporučuji stahovat zdrojové soubory či uživatelskou příručku z jiných webových stránek. Zdrojové soubory mohou být pozměněny a uživatelská příručka nemusí obsahovat všechny kapitoly.

## 5.1 Instalace software a ovladačů

Jako výchozí platforma je zvolen operační systém Microsoft Windows XP professional (32bitová verze) s instalovaným service pack 3 plně aktualizovaným. Další operační systémy (Windows Vista, Windows 7 či Windows 8) zatím nebyly odzkoušeny, případně zde budou doplněny pokyny ke zprovoznění software a ovladačů na těchto systémech.

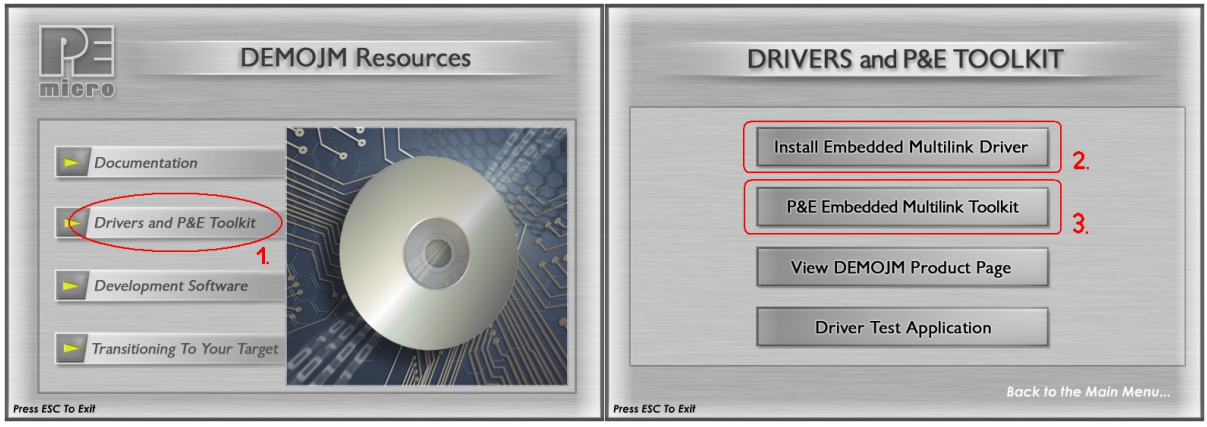
Pro instalaci vývojového prostředí a všech potřebných ovladačů použijeme instalační DVD přiložené u DEMOJM KITu. Po vložení DVD do mechaniky nám buď automaticky či po manuálním otevření naskočí okno výběru vývojové desky. Zde vybereme desku „DEMOJM“.

Jako první nainstalujeme vývojové prostředí CodeWarrior. Jedná se o typickou instalaci software, kde uživatel potvrzuje jednotlivé kroky. Za upozornění stojí volba složky, do které se rozbalí instalační data pro CodeWarrior. Zde je lepší zvolit případně složku uživateli dobře dostupnou (např. plocha), aby mohl zbytečná data po instalaci odstranit (výchozí složka je Temp). Doporučuji neměnit žádná nastavení jednotlivých součástí, které budou instalovány. Po instalaci CodeWarrioru je nezbytně nutné nainstalovat servise pack *8-bit Flexis JM60*. Zde stačí pouze potvrzovat jednotlivé kroky instalace. Stejný postup je i u instalace *USB-LITE stack*. Na následujícím obrázku 5.1 je zobrazen výše popsaný postup instalace softwaru a ovladačů.




Obrázek 5.1 postup instalace softwaru a ovladačů

Při instalaci *P&E Demo Board Toolkitu*(obrázek 5.2) doporučuji uložit archív do vlastní složky, kde jej následně rozbalte (archív obsahuje rovnou instalační sobory). Po spuštění *DEMOJM_Resources.exe* vyberte položku *Drivers and P&E Toolkit*. Nainstalujte *Embedded Multilink Driver* a následně bez nutného restartování operačního systému *P&E Embedded Miltilink Toolkit*. Jedná se opět o typickou instalaci ovladačů, navíc si můžete vybrat pro jaké uživatele chcete ovladače instalovat.



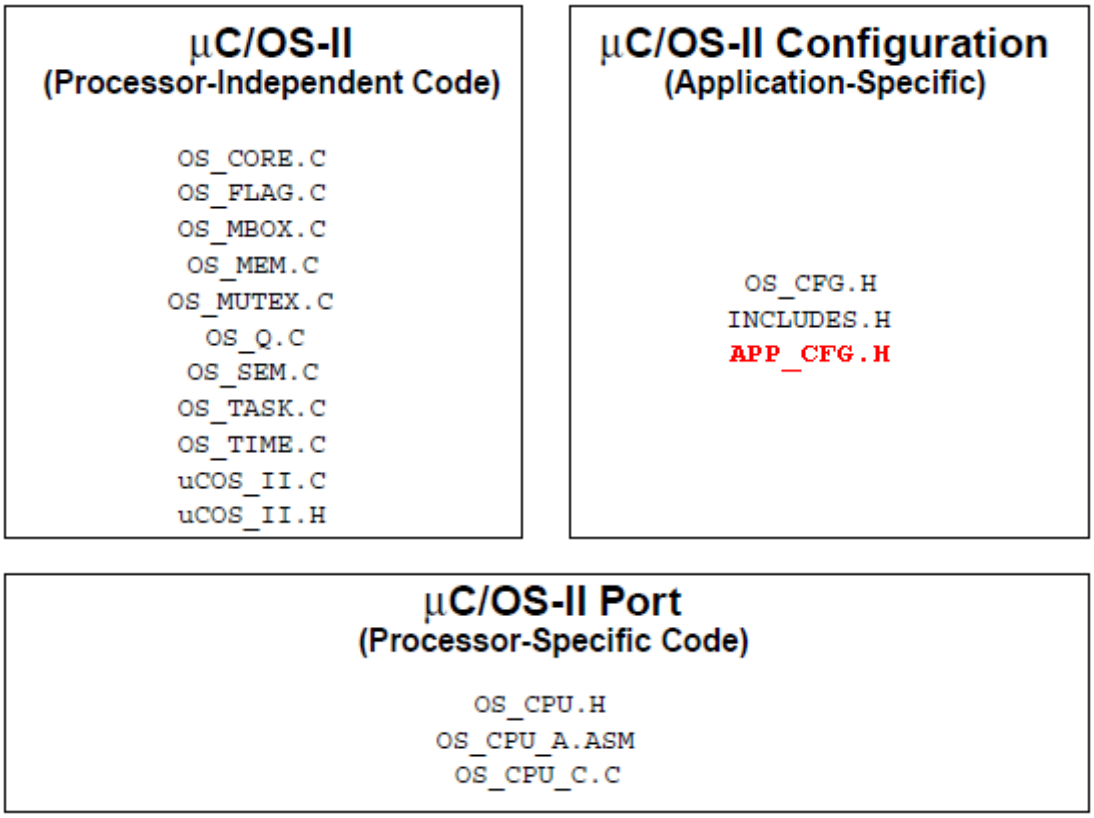

Obrázek 5.2 instalace ovladačů DEMOJM

Nyní již máme potřebný software a ovladače nainstalovány či staženy, pokročíme k vytvoření nového projektu v prostředí CodeWarrior.

## 5.2 Portace systému µCOS-II

Portace jako pojem označuje přizpůsobení real-time operačního systému µC/OS-II na různé mikrokontrolery či mikroprocesory. Většina kódů pro různé architektury je psána v jazyce C, nicméně je vždy nutné specifické části psát přímo v assembleru pro konkrétní platformu. Samotná portace je většinou jednoduchá, nicméně pokud máme velmi rozlišnou architekturu samotného

procesoru, může být i časově velmi nákladná. Zdrojové soubory můžeme rozdělit dle obrázku 5.3 [5] do tří hlavních skupin, a to soubory jádra systému µC/OS-II, konfigurační soubory aplikace a soubory portace. Červeně je doplněn soubor *app_cfg.h*, který se objevuje v novějších verzích systému (2.8 a vyšší) a v publikaci [5] ještě není uveden.




Obrázek 5.3 Souborová struktura systému µC/OS-II

Portace jako taková má pět podmínek, které je nutné splnit, jinak nelze na dané platformě zprovoznit systém µC/OS-II. Náš mikrokontroler JM60 všechny tyto podmínky splňuje. Jsou to:

- pro procesor je dostupný kompilátor jazyka C, který dokáže generovat reentratní kód
- procesor podporuje přerušení a dokáže jej generovat v přesném intervalu, typicky mezi (0,1 až 0,01 s)
- přerušení je možné povolovat a zakazovat prostřednictvím jazyka C
- procesor má vlastní HW zásobník, který je schopen pojmout větší objem dat (u velkých aplikací řádově kilobajty – KB)
- procesor má instrukční sadu vybavenou instrukcemi pro uložení a načtení ukazatele vrcholu zásobníku, dalších procesorových registrů buď na zásobník, nebo do paměti.

Samotný proces portace je velmi přímočarý po pochopení specifických hardwarových vlastností procesoru a kompilátoru jazyka C, který je použit. V závislosti na použitém procesoru se může portace skládat z vytvoření či změny 50 až 300 řádků kódu a může zabrat od pár hodin až po celý

týden. Nejjednodušším způsobem, jak portaci vytvořit, je modifikovat již existující port pro procesor, který je velmi podobný našemu. Následná tabulka zobrazuje nezbytné nutné úpravy zdrojového kódu, pro úspěšnou portaci. Je převzata z publikace [5], kde autor doplnil ještě sloupec *Complexity*, který určuje náročnost: 1 znamená jednoduchou, 2 průměrnou složitost a 3 více náročnou část.

| *Name* | *Type* | *File* | *C or Assembly?* | *Complexity* |
|---|---|---|---|---|
| BOOLEAN | Data Type | OS_CPU.H | C | 1 |
| INT8U | Data Type | OS_CPU.H | C | 1 |
| INT8S | Data Type | OS_CPU.H | C | 1 |
| INT16U | Data Type | OS_CPU.H | C | 1 |
| INT16S | Data Type | OS_CPU.H | C | 1 |
| INT32U | Data Type | OS_CPU.H | C | 1 |
| INT32S | Data Type | OS_CPU.H | C | 1 |
| FP32 | Data Type | OS_CPU.H | C | 1 |
| FP64 | Data Type | OS_CPU.H | C | 1 |
| OS_STK | Data Type | OS_CPU.H | C | 2 |
| OS_CPU_SR | Data Type | OS_CPU.H | C | 2 |
| OS_CRITICAL_METHOD | #define | OS_CPU.H | C | 3 |
| OS_STK_GROWTH | #define | OS_CPU.H | C | 1 |
| OS_ENTER_CRITICAL() | Macro | OS_CPU.H | C | 3 |
| OS_EXIT_CRITICAL() | Macro | OS_CPU.H | C | 3 |
| OSStartHighRdy() | Function | OS_CPU_A.ASM | Assembly | 2 |
| OSCtxSw() | Function | OS_CPU_A.ASM | Assembly | 3 |
| OSIntCtxSw() | Function | OS_CPU_A.ASM | Assembly | 3 |
| OSTickISR() | Function | OS_CPU_A.ASM | Assembly | 3 |
| OSTaskStkInit() | Function | OS_CPU_C.C | C | 3 |
| OSInitHookBegin() | Function | OS_CPU_C.C | C | 1 |
| OSInitHookEnd() | Function | OS_CPU_C.C | C | 1 |
| OSTaskCreateHook() | Function | OS_CPU_C.C | C | 1 |
| OSTaskDelHook() | Function | OS_CPU_C.C | C | 1 |
| OSTaskSwHook() | Function | OS_CPU_C.C | C | 1 |
| OSTaskStatHook() | Function | OS_CPU_C.C | C | 1 |
| OSTCBInitHook() | Function | OS_CPU_C.C | C | 1 |
| OSTimeTickHook() | Function | OS_CPU_C.C | C | 1 |
| OSTaskIdleHook() | Function | OS_CPU_C.C | C | 1 |

Tabulka 5.1 Souhrn portace systému µC/OS-II

Nejednodušším způsobem jak portaci vytvořit bez nutnosti psaní celých souborů od začátku, je vyjít z již vytvořeného portu. Na stránkách www.micrium.com jsou již funkční porty pro různé architektury. Při jejich prohledání nalezneme projekt pro mikrokontroler MC9S08QE128 se stejným výpočetním jádrem HCS08. Podrobný popis samotného mikroprocesoru HCS08 najdeme v [6]. Po stažení rozbalíme samorozbalovací archiv *Micrium-Freescale-uCOS-II-Probe-MC9S08.exe*. Doporučuji vytvořit složku projektu např.: *Project_JM60*. Do této složky překopírujeme obsah složky

*Micrium\Software\EvalBoards\Freescale\MC9S08QE128\DEMOQE\Paged\Codewarrior**OS-Probe***. Přejmenujeme soubor *OS-Probe.mcp* na *Project_JM60.mcp* a následně nakopírujeme do složky *Sources* v *Project_JM60* celou složku *uCOS-II* ze složky *Micrium\Software*. Dále nakopírujeme do složky *Sources* složku *BSP*, z umístění:

*Micrium\Software\EvalBoards\Freescale\MC9S08QE128\DEMOQE\Paged\Codewarrior*

a smažeme soubor *probe_com_cfg.h*. Nyní otevřeme soubor *Project_JM60.mcp.*

### 5.2.1 Úpravy nastavení projektu CodeWarrioru

Po otevření souboru *Project_JM60.mcp* je nutné změnit různá nastavení vývojového prostředí. Není důležité dodržet následují pořadí kroků, nicméně jej doporučuji:

- ze stromové struktury projektu odstranit ze složky *Sources* tyto přebytečné složky: *uC/CPU*, *uC/LIB* a *uC/Probe*. Dále odstranit ze složky *Sources/Application* soubor *probe_com_cfg.h* a ze složky *Sources/BSP* soubory accelerometer.*. To provedeme pravým kliknutím na jednotlivé složky či soubor výběrem volby „Remove“.
- V menu CodeWarriroru v sekci *Project* změníme cílovou platformu pomocí *Change MCU/Connection* na *HCS08→HCS08JM family→MC9S08JM60* a způsob připojení zvolíme jako *P&E Multilink/Cyclone Pro.* Doporučuji nechat vytvořit zálohu předešlého projektu.
- Pravým kliknutím na soubor *ansibfm.lib* ve složce *Libs* a výběrem *Open in Windows Explorer* otevřeme složku všech **.lib* souborů. Najdeme soubor *ansiis.lib* a přetáhneme ho myší do projektu pod původní *ansibfm.lib*. *Ten* z projektu odstraníme.
- Pomocí ALT+F7 otevřeme nastavení *Standard Settings* kde v levé části vybereme *Access Paths*. Zde potřebujeme odstranit přebytečné cesty. Všechny cesty začínající řetězcem „*{Project}.“*. odstraníme. Nyní přidáme cesty ke zdrojovým souborům systému μC/OS-II. Voblou Add... přídáme *Sources\uCOS-II\Source* , *Sources\BSP* a *Sources\uCOS-II\Ports\MC9S08\Paged\Codewarrior*.
- V nastavení *Assembler for HC08* a *Compiler for HC08* změníme příkazy *Command Line Arguments*: na „-Cs08 -Ms“ a potvrdíme změny (budeme upozorněni na nutnost opětovné kompilace projektu).

V následující části projdeme jednotlivé soubory portace a konfigurace převzaté z již existujícího projektu systému μC/OS-II pro platformu MC9S08QE128 a jejich nutné změny. Po dokončení potřebných změn bude funkční portace, jádro systému a konfigurační soubory budou připraveny pro tvorbu aplikace. Následně může uživatel začít tvořit vlastní aplikaci. Předem doporučuji v souborech nepotřebné části kódu **zakomentovávat**. Případným smazáním si můžeme přidat práci při následné tvorbě aplikace v podobě nutnosti dopisování smazaných částí kódu.

### 5.2.2 Úpravy souborů portace

#### os_cpu.h *v projektu Sources\Application\uC/OS-II\Ports\MC9S08\Paged\Codewarrior*

Tento soubor obsahuje specifické definice konstant, maker a definice datových typů pro daný procesor. Různé procesory mají různé délky slov, proto je na začátku souboru seznam definic typů zajišťující přenositelnost systému. Dále je zde definován výběr kritické sekce. Uživatel má na výběr jednu ze tří možných, nicméně v praxi se většinou pro mikroprocesor HCS08 používá metoda 3 (`OS_CRITICAL_METHOD 3`). Ostatní dvě mají implementační nevýhody a pro platformu JM60 jsou nevyhovující. Jsou zde definována makra pro vstup do kritické sekce (zakázání přerušení) a výstup z kritické sekce (povolení přerušení), jejich implementace je v souboru *os_cpu_a.s*. Použití Kritické sekce je následující:

```
{ ...
      OS_ENTER_CRITICAL();
      /* kód kritické sekce, který nesmí být přerušen */
      OS_EXIT_CRITICAL();
...}
```

Dále je zde definován směr růstu zásobníku `OS_STK_GROWTH 1` a způsob vyvolání softwarového přerušení `OS_TASK_SW()` pomocí instrukce assembleru `SWI` (dle kapitoly 2.5.1).

Soubor nepotřebujeme pro naši platformu editovat.

#### os_cpu_c.c *v projektu Sources\Application\uC/OS-II\Ports\MC9S08\Paged\Codewarrior*

V tomto soubru musíme definovat deset funkcí v jazyce C. Jedinou vyžadovanou funkcí je pouze `OSTaskStkInit()`. Tato funkce simuluje práci se zásobníkem při vytvoření nové úlohy v systému. Je potřeba na zásobníku vytvořit prostor pro všechny potřebné hodnoty. Prvně simulujeme volání funkce s parametrem, následně ukládání registrů, které je konáno při vyvolání přerušení, a jako poslední je případně uložení ostatních registrů mikroprocesoru, které nejsou automaticky ukládány. Ostatní funkce je nutno deklarovat, nicméně nemusí obsahovat žádný zdrojový kód.

Nutnou úpravou je změna kódu funkce `OSTaskStkInit(...)` na:

```
{
  INT16U *wstk;
  INT8U  *bstk;
  opt     = opt;
  wstk    = (INT16U *)ptos;
  *--wstk = (INT16U)p_arg;
  *--wstk = (INT16U)(task);
   ...
```

```
   ...
  bstk    = (INT8U *)wstk;
  *--bstk = (INT8U)0x22;
  *--bstk = (INT8U)0xAA;
  *--bstk = (INT8U)0x00;
  *--bstk = (INT8U)0x11;
  return ((OS_STK *)bstk);
}
```

Zakomentujte všechny hook funkce, v případě, že je budete v budoucnu využívat je zde odkomentujte a implementujeme. Jde o:

```
App_TaskCreateHook()                App_TaskIdleHook()
App_TaskDelHook()                   App_TCBInitHook()
App_TaskStatHook()                  App_TimeTickHook()
App_TaskSwHook()
```

## os_cpu_a.s *v projektu Sources\Application\uC/OS-II\Ports\MC9S08\Paged\Codewarrior*

V tabulce 5.1 uveden pod názvem *os_cpu_a.asm*. Vzhledem k tomu, že náš mikrokontroler JM60 nepoužívá posuvné okénko pro adresaci, smažeme na začátku souboru dva řádky kódu, a to:

```
NON_BANKED:       section
PPAGE:            equ    $0078
```

Samotný systém vyžaduje implementaci čtyř funkcí pomocí assembleru, které pracují se zásobníkem čipu. Jedná se o následující:

```
OSStartHighRdy              [I.]
OSCtxSw                     [II.]
OSTickISR                   [III.]
OSIntCtxSw                  [IV.]
```

### I. OSStartHighRdy

Tato funkce je volána z `OSStart()` za účelem výběru úlohy s nejvyšší prioritou připravenou k běhu v systému. Pseudokód pro `OSStartHighRdy()`:

```
void OSStartHighRdy (void){
    Zavolat uživatelem definovaný OSTaskSwHook();                  [I.1]
    OSRunning = TRUE;                                              [I.2]
    Načíst ukazatel zásobníku z úlohy pro návrat:                  [I.3]
       Stact pointer = OSTCBHighRdy->OSTCBStkPtr;
    Obnovit všechny automaticky neukládané registry procesoru;     [I.4]
    Zavolat instrukci návratu z přerušení (obnovení automaticky    [I.5]
    ukládaných registrů);
}
```

Pro mikrokontroler JM60 je v assembleru implementována následovně:

```
OSStartHighRdy:
      jsr   OSTaskSwHook                [I.1]
      lda   #$01                        [I.2]
      sta   OSRunning
      ldhx  OSTCBHighRdy                [I.3]
      ldhx  0, x
      txs
      pulh                              [I.4]
      rti                               [I.5]
```

## II. OSCtxSw

Tato funkce je spouštěna z makra `OS_TASK_SW` sloužícího k výměně kontextů jednotlivých úloh v systému. Vektor přerušení vyvolaný instrukcí `SWI` (softwarové přerušení) musí ukazovat na tuto funkci! Pseudokód pro `OSCtxSw`:

```
void OSCtxSw (void){
     Uložit všechy automaticky neukládané registry procesoru;      [II.1]
     Uložit ukazatel zásobníku aktuální úlohy na aktuální OS_TCB: [II.2]
        OSTBCCur->OSTCBStkPtr = stack pointer;
     Zavolat uživatelem definovaný OSTaskSwHook();                 [II.3]
     OSPrioCur = OSPrioHighRdy;                                    [II.4]
     OSTCBCur = OSTCBHighRdy;                                      [II.5]
     Načíst ukazatel zásobníku z úlohy pro návrat:                 [II.6]
        Stact pointer = OSTCBHighRdy->OSTCBStkPtr;
     Obnovit všechny automaticky neukládané registry procesoru;    [II.7]
     Zavolat instrukci návratu z přerušení (obnovení automaticky   [II.8]
     ukládaných registrů);
}
```

Pro mikrokontroler JM60 je v assembleru implementována následovně:

```
OSCtxSw:                                        ...
      pshh                    [II.1]    jsr   OSTakSwHook         [II.3]
      tsx                     [II.2]    lda   OSPrioHighRdy       [II.4]
      pshx                              sta   OSPrioCur
      pshh                              ldhx  OSTCBHighRdy        [II.5]
      ldhx  OSTCBCur                    sthx  OSTCBCur
      pula                              ldhx  0, x                [II.6]
      sta   0, x                        txs
      pula                              pulh                      [II.7]
      sta   1, x                        rti                       [II.8]
       ...
```

## III. OSTickISR

V projektu pro QE128 je takto funkce nazvána `Tmr_TickISR`. Systém µC/OS-II vyžaduje generování přerušení v intervalu mezi 10 a 100 Hz. Uživatel musí povolit přerušení od časovače po nastartování systému pomocí funkce `OSStart()`. Tato funkce nicméně nikdy neskončí. Proto je nutné toto povolení udělat v první úloze zavedené do sytému. Pseudokód pro `Tmr_TickISR`:

```
void TMR_TickISR (void){
     Uložit všechy automaticky neukládané registry procesoru;    [III.1]
     Inkrementace OSIntNesting;                                  [III.2]
     if (OSIntNesting == 1)                                      [III.3]
        OSTCBCur->OSTCBStkPtr = stack pointer;
     Znovunastavení časovače pomoci Tmr_TickISR_Handler()        [III.4]
     Zavolani OSIntExit()                                        [III.5]
```

```
        Obnovit všechy automaticky neukládané registry procesoru;   [III.6]
        Zavolat instrukci návratu z přerušení (obnovení automaticky [III.7]
        ukládaných registrů);
    }
```

Pro mikrokontroler JM60 je v assemleru implementována následovně:

```
Tmr_TickISR:                                    ...
      pshh                      [III.1]         ldhx  OSTCBCur
      lda   OSIntNesting        [III.2]         pula
      add   #1                                  sta   0, x
      sta   OSIntNesting                        pula
      cmpa  #$01                [III.3]         sta   1, x
      bne   Tmr_TickISR1                  Tmr_TickISR1:
      tsx                                       jsr  Tmr_TickISR_Handler [III.4]
      pshx                                      jsr  OSIntExit           [III.5]
      pshh                                      pulh                     [III.6]
       ...
                                                rti                      [III.7]
```

## IV. OSIntCtxSw

Tato funkce je volána z `OSIntExit()`, aby zajistila přepnutí kontextu z rutiny přerušení. Před voláním této funkce máme zajištěno uložení **všech** registrů mikrokontroleru (včetně registru H, který není automaticky ukládán). Následují pseudokód je pro verze systému µC/OS-II vyšší jak 2.51 včetně:

```
    void OSIntCtxSw (void){
        Zavolat uživatelem definovaný OSTaskSwHook();                [IV.1]
        OSPrioCur = OSPrioHighRdy;                                   [IV.2]
        OSTCBCur = OSTCBHighRdy;                                     [IV.3]
        Načíst ukazatel zásobníku z úlohy pro návrat:                [IV.4]
           Stact pointer = OSTCBHighRdy->OSTCBStkPtr;
        Obnovit všechny automaticky neukládané registry procesoru;
        [IV.5]
        Zavolat instrukci návratu z přerušení (obnovení automaticky  [IV.6]
        ukládaných registrů);
    }
```

Pro mikrokontroler JM60 je v assembleru implementována následovně:

```
OSIntCtxSw:                                     ...
      jsr   OSTakSwHook         [IV.1]          sthx  OSTCBCur
      lda   OSPrioHighRdy       [IV.2]          ldhx  0, x                [IV.4]
      sta   OSPrioCur                           txs
      ldhx  OSTCBHighRdy        [IV.3]          pulh                      [IV.5]
       ...                                      rti                       [IV.6]
```

Dále jsou zde ještě definovány ještě dvě funkce implementující použití samotné kritické sekce. Jedná se o funkce zakazující přerušení a obnovu původního stavového registru mikroprocesoru:

```
OS_CPU_SR_Save
OS_CPU_SR_Restore
```

V těchto funkcích je nutno změnit instrukce návratu z podprogramu. Čip QE128 má instrukci RTC, zatímco JM60 používá instrukci RTS. Nic jiného neměníme

### 5.2.3 Úpravy souborů konfigurace

#### includes.h *v projektu Sources\Application*

Tento soubor ve skutečnosti není součástí samotného portu, nicméně je zde zmíněný díky jeho funkci. Hlavní výhodou tohoto souboru je jeho použití jako hlavního hlavičkového souboru pro celou uživatelskou aplikaci. Při tvorbě aplikace bude v celém zdrojovém souboru vložen právě a jen *includes.h (*`#include <includes.h>`*)*. Tento soubor umožňuje každému zdrojovému souboru jádra **.c*, aby byl zapsán bez obav jakýkoliv hlavičkový soubor, který je skutečně potřeba. Jedinou nevýhodou tohoto souboru je, že může obsahovat různé hlavičkové soubory, které nejsou potřeba pro aktuálně kompilovaný **.c* soubor projektu. Soubor je většinou v projektech umístěn v konfigurační části systému viz obrázek 5.3.

Úprava souboru pro zprovoznění jádra systému µC/OS-II:

- úprava jména čipu → `#include <MC9S08JM60.h>`
- ponechání:
  - `#include <ucos_ii.h>`
  - `#include <bsp.h>`
- vše ostatní zakomentovat (pro samotné jádro není potřeba)

#### app_cfg.h *v projektu Sources\Application*

Dle názvu je patrné, že se jedná o konfigurační soubor aplikace. Nyní zatím netušíme, jakou aplikaci budeme vytvářet. Pro úspěšný překlad projektu nám stačí v tomto souboru ponechat pouze tyto řádky kódu:

```
#ifndef  APP_CFG_H
  #define  APP_CFG_H
  #define  OS_TASK_TMR_PRIO  63 //nejnižší priorita
#endif
```

Zbytek kódu zakomentujte, při tvorbě aplikace bude větší část potřeba pro konfiguraci parametrů jednotlivých úloh v systému µC/OS-II.

### os_cfg.h *v projektu Sources\Application*

Jedná se o soubor konfigurace samotného systému, ve kterém jako uživatel určujeme, které části samotného jádra systému budeme používat. Je velmi důležitý co se týče optimalizace velikosti systému µC/OS-II pro nahrávání do flash pamětí na mikrokontrolerech. Můžeme zde povolit moduly zasílání zpráv, signalizace událostí pomocí vlajek, semafory, fronty zpráv, funkce pracující s časem a další.

Dle nároků tvořené aplikace zde můžeme nepotřebné moduly po jejím vytvoření zakázat a zmenšit tak velikost zdrojových kódů systému nahrávaných na čip.

Úpravy *os_cfg.h* :

```
#define OS_TASK_STAT_EN    0       //zakázání statistické úlohy
#define OS_FLAG_EN         0       //zakázání vlajek
#define OS_MBOX_EN         0       //zakázání schránek pro zprávy
#define OS_MEM_EN          0       //zakázání správy paměti
#define OS_MUTEX_EN        0       //zakázání mutexů
#define OS_Q_EN            0       //zakázání front zpráv
#define OS_SEM_EN          0       //zakázání semaforů
#define OS_TMR_EN          0       //zakázání timer management
```

## 5.2.4 Úprava souboru aplikace

### bsp.h *v projektu Sources\BSP*

Hlavičkový soubor Board Support Package (BSP) pro soubor *bsp.c*. Jsou zde definovány prototypy jednotlivých funkcí obsažených, dále je zde specifikován kanál časovače, který bude použit pro generování pravidelného přerušení v systému µC/OS-II. Zde je čistě na uživateli, který kanál kterého časovače pro tento účel zvolí. Pokud nebude zmíněno jinak, bude nadále používán kanál 0 časovače 1. Nutnou, ač ne zcela postačující změnou tohoto souboru (na základě aplikace mohou být zapotřebí další změny) je přepsání typů proměnných podle *os_cpu.h*.

### bsp.c *v projektu Sources\BSP*

Soubor obsahuje mnohé funkce pro ovládání periferií, jako jsou LED diody a případné další komponenty. Tyto funkce nejsou předmětem portace a každý programátor si je může upravit dle vlastních potřeb, případně funkce umístit do jiného souboru.

Nicméně se zde nacházejí i pro portaci nezbytné součásti, jako konfigurace čipu a jeho periferií. Především jde o výchozí nastavení čipu po resetu (vypnutí watchdog modulu a případná další konfigurace dle aplikace), nastavení výchozího stavu časovače generujícího pravidelné přerušení

pro systém a jeho spuštění (funkce `Tmr_TickInit()`) a funkce ošetřující přerušení od zvoleného kanálu časovače `Tmr_TickISR_Handler()`.

Pro různé aplikace může být zvoleno různé nastavení nejen časovače generujícího přerušení systému, stejně tak i odlišné chování ostatních funkcí. Proto zde není podrobněji komentován zdrojový kód jednotlivých funkci.

Upravené soubory *bsp.h* a *bsp.c* se nacházejí v archivech projektů na přiloženém CD.

### vectors.c *v projektu Sources\Application*

Soubor sloužící pro konfiguraci obsluh vektorů přerušení. Pro prvotní překlad a test portu jsou zapotřebí pouze prototypy pro funkce `Startup`, `TmrTickISR` a `OSCtxSw`. Tabulka vektorů přerušení je zde pro čip QE128, nutno změnit umístění obslužných rutin dle tabulky 2.2 na straně 10.

Upravený soubor *vectors.c* se nachází v archivech projektů na přiloženém CD.

### app.c *v projektu Sources\Application*

Zde je to velmi jednoduché. Zatím netvoříme aplikaci, tudíž nám ve zdrojovém souboru aplikace stačí pro testování úspěšného překladu portu pro platformu JM60 ponechat následující kód:

```
#include <includes.h>
void main (void)
{
   BSP_Init();        //výchozí nastavení čipu a periferií
   OSInit();          //inicializace systému
   OSStart();         //start systému, fukce ze které se
                        nikdy nevrátíme
}
```

Původní kód aplikace doporučuji ponechat, bude se velmi hodit při tvorbě nové aplikace jako ukázkový příklad.

## 5.2.5 Úprava souboru projektu

### Project.prm *v projektu Project Settings\Linker Files*

Soubor definující rozsahy a typy pamětí na čipu, podle tohoto souboru jsou ukládána veškerá data do paměti čipu JM60. Dále je zde definována velikost zásobníku.

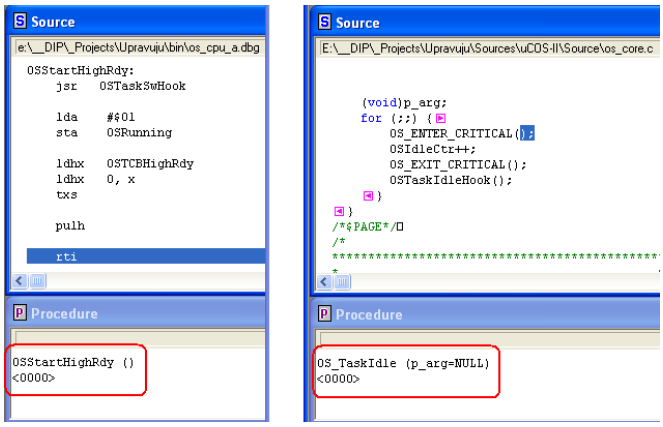
Nutné změny:

- řádek 16 přepsat `FAR_RAM` na `DEFAULT_RAM`
- řádek 29 smazat `DEFAULT_RAM`
- řádek 35 zakomentovat `VECTOR 0 _Startup`

## 5.3 Test portu

### 5.3.1 `OSTaskStkInit()` a `OSStartHighRdy()`

Po provedení výše popsaných kroků lze port systému bez chyb přeložit. Pro jeho testování využijeme debugeru v prostředí CodeWarrior. Postupně zkontrolujeme správnou funkčnost implementovaných procedur v souboru *os_cpu_a.s*. Prvně je zapotřebí změnit *MCU Connection* na *Full Chip Simulation* a **vypnout** časovač generující tik systému µC/OS-II (v *bsp.c*). Po spuštění debugeru (klávesa F5) můžeme program krokovat pomocí kláves F10 či F11. Po odkrokování do procedury `OSStartHighRdy` a následné instrukci `rti` bychom měli být ve funkci `OS_TaskIdle()`, viz následující obrázek. Pokud bychom časovač generující přerušení pro systémové hodiny nevypnuli, došlo by k vyvolání `Tmr_TickISR` a až následně `OS_TaskIdle()`.




Obrázek 5.4 Debuger test `OSStartHighRdy()` a `OSTaskStkInit()`

### 5.3.2 `OSCtxSw()`, `OSIntCtxSw()` a `OSTickISR()`

Pro test těchto procedur vytvoříme úlohu `TestTask()`, která bude blikat LED na kitu DEMOJM. Úloha vždy jen přepne diody a uspí se. Systém tedy bude přepínat mezi touto úlohou a `OS_TaskIdle()` úlohou pomocí `OSCtxSw`. Následně je samotná idle úloha přerušena tikem systému `Tmr_TickISR`. Po skončení `Tmr_TickISR` opět systém spustí úlohu `TestTask()`.

V souboru *app.c* je zapotřebí definovat velikost zásobníku pro tuto úlohu, naprogramovat úlohu a vytvořit ji v systému. Dále v *bsp.c* **zapnout** časovač generující přerušení pro `Tmr_TickISR`.

V následujícím kódu je do systému zavedena testovací úloha blikající LED (vytvoření úlohy před samotným startem systému). Implementace funkcí pracujících s diodami zde není uvedena, tento zdrojový kód je přiložen na CD ve složce tohoto testovacího projektu (soubor *bsp.c*).

Kód souboru *app.c*:

```
#include <includes.h>

OS_STK TestTaskStk[50];
static  void  TestTask(void *p_arg);

void main (void)
{
  BSP_Init();
  OSInit();
   OSTaskCreate(TestTask, (void *)0,
                &TestTaskStk[49], 0);
  OSStart();
}
```

```
void TestTask (void *pdata) {
  pdata = pdata;
  INT8U LEDState = FALSE;
  while(1){
    if (LEDState) {
      LED_On(0);
      LEDState = FALSE;
    } else {
      LED_Off(0);
      LEDState = TRUE;
    }
    OSTimeDly(4);
  }
}
```

Pro testování přímo na vývojovém kitu je nutné přepnout *MCU Connection* na *P&E Multilink/Cyclone Pro.* Po nahrání programu a spuštění budou LED blikat. Pokud zastavíme program, uvidíme s největší pravděpodobností jako aktuální úlohu `OS_TaskIdle`. Samotné přepnutí stavu diod (úloha `TestTask`) je velmi krátké. Pro její zachycení v debugeru můžeme vložit breakpoint na úlohu `TestTask()`.

Pokud se diody rozblikají, znamená to správnou funkci testovaných procedur. Nyní můžeme tvořit vlastní testovací aplikace jádra a jeho funkcí případně již cílovou aplikaci.

# 6 Testovací aplikace jádra

V této kapitole budou uvedeny jednoduché testovací aplikace pro různé funkce jádra, které se nejčastěji využívají a které můžeme využít i ve vlastní aplikaci a případné možné komplikace při použití těchto funkcí. Hlavní snahou je uživateli co nejvěrohodněji vizuálně znázornit správný běh programu (využití LED diod jako signalizace běhu úloh na DEMOJM desce). Všechny zde popsané ukázkové příklady použití sytémových funkcí jsou na přiloženém CD ve složce *Source* pod názvem zde v nadpise uvedeném. Jednotlivé programy budou reprezentovat vždy jednu skupinu z následujících:

- Task management
- Time management
- Synchronization and communication
- ISR test

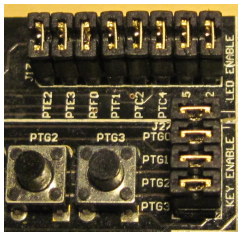
Podrobnou definici systémových funkcí v těchto testovacích aplikacích použitých, jejich případné varianty využití, návratové hodnoty a datové typy argumentů jsou uvedeny v knize [5] v kapitole 16. Zde se nachází pouze popis funkcionality jednotlivých ukázkových aplikací.

Některé testovací aplikace využívají ovládání pomocí tlačítek, které nám deska DEMOJM nabízí. Pro správnou funkci tlačítek je nutné mít správně zapojené jumpery na desce DEMOJM a zároveň propojit např. pomocí drátků port `PTA` (možno libovolný pin portu) a portem `PTG` konkrétněji jeho piny `PTG0`, `PTG1` a `PTG2`. Jednotlivé piny jsou na DEMOJM vyvedeny a očíslovány na konektoru `MCU PORT`. Podrobný popis vývodů na konektoru `MCU PORT` a jejich funkcí je na následujícím obrázku 6.1 převzatého z [7].

| | | | |
|---|---|---|---|
| VDD | 1 | 2 | IRQ/TPMCLK |
| VSS | 3 | 4 | RESET |
| PTE0/TxD1 | 5 | 6 | BKGD/MS |
| PTE1/RxD1 | 7 | 8 | VUSB33 |
| PTG0/KBIP0 | 9 | 10 | PTB0/MISO2/ADP0 |
| PTG1/KBIP1 | 11 | 12 | PTB1/MOSI2/ADP1 |
| PTE2/TPM1CH0 | 13 | 14 | PTB2/SPSCK2/ADP2 |
| PTE3/TPM1CH1 | 15 | 16 | PTB3/SS2/ADP3 |
| PTE5/MOSI1 | 17 | 18 | PTB4/KBIP4/ADP4 |
| PTE4/MISO1 | 19 | 20 | PTB5/KBIP5/ADP5 |
| PTE6/SPSCK1 | 21 | 22 | PTB6/ADP6 |
| PTE7/SS1 | 23 | 24 | PTB7/ADP7 |
| PTF0/TPM1CH2 | 25 | 26 | PTC0/SCL |
| PTF1/TPM1CH3 | 27 | 28 | PTC1/SDA |
| PTF2/TPM1CH4 | 29 | 30 | PTG2/KBIP6 |
| PTF3/TPM1CH5 | 31 | 32 | PTG3/KBIP7_J1 |
| VREFH | 33 | 34 | PTF4/TPM2CH0 |
| VREFL | 35 | 36 | PTF5/TPM2CH1 |
| PTD0/ADP8/ACMP+ | 37 | 38 | PTC5/RxD2 |
| PTD1/ADP9/ACMP- | 39 | 40 | PTC3/TxD2 |
| PTD2/KBIP2/ACMPO | 41 | 42 | PTG4/XTAL |
| PTD3/KBIP3/ADP10 | 43 | 44 | PTG5/EXTAL |
| PTD4/ADP11 | 45 | 46 | PTA0 |
| PTD5 | 47 | 48 | PTA1 |
| PTD6 | 49 | 50 | PTA2 |
| PTD7 | 51 | 52 | PTA3 |
| PTC2 | 53 | 54 | PTA4 |
| PTC4 | 55 | 56 | PTA5 |
| PTC6 | 57 | 58 | PTF6 |
| NC | 59 | 60 | PTF7 |

Obrázek 6.1 funkce jednotlivých pinů na MCU PORT na desce DEMOJM

Dále používají pro detekci běhu jednotlivých úloh uživateli osm LED diod. V každé aplikaci (pokud je to nutné) je softwarově řešena logika ovládání tlačítek a v samotných úlohách i diod tak, aby LED diody vždy jednoznačně detekovali běh aktuálně běžící úlohy. Samotná tlačítka a LED diody mají na desce DEMOJM povolovací jumpery. Je nutné, aby byly propojeny, viz následující obrázek 6.2.




Obrázek 6.2 Jumpery tlačítek a LED diod

Pokud nebude zmíněno jiné nastavení u dalších testovacích úloh, platí výše popsané nastavení jumperů a propojení pinů portu PTA a PTG.

## 6.1 Task management

Správou úloh v systému µC/OS-II se rozumí funkce sloužící k vytvoření úlohy, jejímu pozastavení, znovurozběhnutí, ale i smazání úlohy ze systému, případně změny priority vybrané úlohy. Pro testování funkčnosti jsou vytvořeny tři jednoduché aplikace pracující s těmito funkcemi. Jedná se o tyto testovací aplikace *TM-PrioChange*, *TM-Susp-Resum* a *TM-Creat-Delet*.

### 6.1.1 TM-Creat-Delet

Tato jednoduchá aplikace demonstruje uživateli běh tří primitivních úloh blikajících LED na desce DEMOJM. Jednotlivé úlohy jsou nazvány `LeftTask()`, `MiddleTask()` a `RightTask()` podle diod, které ovládají (`LeftTask` diody `PTE2` a `PTE3`, `MiddleTask` diody `PTF1` a `PTC2` a `RightTask` diody `PTF5` a `PTD2` ).

Uživatel může ovládat tyto úlohy (mazat je ze systému či znovu zavádět) pomocí tlačítek na DEMOJM desce, a to pomocí `PTG0` úlohu `LeftTask`, `PTG1` ulohu `MiddleTask` a `PTG2` úlohu `RightTask`.

Po spuštění této ukázkové úlohy začnou blikat stejnou rychlostí dvojce diod jednotlivých úloh, uživatel může pomocí stisku daného tlačítka pro danou úlohu ze systému vymazat či ji znovu zavést (podle aktuálního stavu úlohy). Tlačítka jsou ovládána pomocí úlohy s nejvyšší prioritou, a to úlohy `ButtonTask()`. Po stisku libovolného tlačítka je toto tlačítko softwarově zablokováno po dobu jedné sekundy, aby při stálém stisku nedocházelo k rychlému mazání a znovuzavádění úlohy do systému (zbytečná režie v systému), především pak z důvodu možnosti vizuální kontroly běhu dané

úlohy (při stálém stisku či velmi rychlém po sobě následujícím zmáčknutí tlačítka nelze lidským okem rozpoznat, jestli úloha běží, či nikoliv). Hodnoty proměnných určujících jak dobu uspání jednotlivých úloh (čas po kterém úloha změní stav diod), tak blokaci jednotlivých tlačítek lze libovolně měnit.

### 6.1.2 TM-PrioChange

Tato aplikace testuje funkčnost změny priority úlohy za běhu systému. V aplikaci jsou zavedeny dvě úlohy `LeftTask()` a `MiddleTask()`, úloze `MiddleTask()` je stiskem tlačítka `PTG1` změněna priorita na vyšší než má `LeftTask()`. Při dalším stisku je opět priorita vrácena na výchozí. Simulace běhu úloh je opět pomocí LED, kde každá úloha ovládá pár různých diod. Tělo těchto dvou úloh je prakticky identické s rozdíly v době běhu úlohy a době uspání. Zde je uvedena `MiddleTask()`:

```
static void MiddleTask(void *pdata){
  INT16U  i;
  (void)pdata;
  while (1){
      LED_Off(0);                       [a]
      for (i = 0; i < 22000; i++){      [b]
        LED_On(4);
         ...
         ...
        LED_On(5);
      }
      LED_Off(0);                                [c]
      OSTimeDly(50);
      }
    }
```

Úloha po probuzení zhasne všechny diody `[a]`, následně ve smyčce simulující čas běhu úlohy zapisuje hodnoty pro rozsvícení diod indikujících její běh `[b]` a před samotným uspáním opět zhasíná všechny diody na desce `[c]`. Takto detekujeme vždy korektně běh jen jedné úlohy pomocí LED, a to i v případě, že dojde ke změně priorit pomocí stisku tlačítka `PTG1` v době, kdy úlohy běží.

V systému je opět úloha `ButtonTask()`, která po stisku tlačítka `PTG1` změní prioritu úlohy `MiddleTask()` a následně tlačítko softwarově blokuje po dobu jedné sekundy, aby bylo na první pohled zřejmé, že LED signalizující běh jednotlivých úloh změnily frekvence svícení.

### 6.1.3 TM-Susp-Resum

Tato aplikace rozšiřuje předešlou testovací aplikaci TM-Creat-Delet. Je zde testováno mazání úlohy a její znovuzavádění v kombinaci s uspáním a opětovném probuzení úlohy. Opět je zde použita úloha `ButtonTask()` implementující logiku běhu demonstrační úlohy, načítání dat z tlačítek a jejich softwarovou blokaci po jejich stisku.

V systému je zavedena demonstrační úloha `KRTask()`, která pomocí LED ilustruje svůj běh. Dioda je postupně rozsvěcována zleva doprava a zpět, přičemž vždy svítí právě jen jedna. Ve výchozí

poloze (levá dioda PTE2 svítí) je definována krátká pauza. Rychlost pohybu jedné svítící diody lze libovolně v této úloze měnit v obou směrech, stejně tak jako velikost pauzy po doběhnutí zpět do výchozí polohy.

Pomocí tlačítek PTG0 a PTG2 lze tuto úlohu ovládat. Tlačítko PTG0 slouží k již v předešlé aplikaci testovanému mazání a znovuzavedení úlohy do systému, zatím co tlačítko PTG2 danou úlohu uspí, případně probere ze spánku (pouze v případě, že je úloha zavedená v systému). Hlavním rozdílem mezi uspáním/probuzením úlohy v systému a smazáním/znovuzavedením je v běhu samotné úlohy. Pokud je úloha uspána a zpět probuzena, pokračuje přesně tam, kde předtím skončil její běh. Při znovuzavedení úlohy běží úloha vždy od jejího začátku.

## 6.2 Time management

Systém μC/OS-II využívá pro uspání úlohy dvě možné funkce, a to OSTimeDlyHMSM() a OSTimeDly(). Každá tato funkce pracuje rozdílným způsobem. Testovací aplikace těchto funkcí *TM-DlyHMSM* obsahuje jako výchozí funkci OSTimeDly(). Kód pro správný běh OSTimeDlyHMSM() je zakomentován.

OSTimeDly() má jako jediný argument číslo typu INT16U, může tedy nabývat maximálně hodnoty 65 535. Tato funkce uspí úlohu na daný počet **tiků systému**. Pokud máme v souboru os_cfg.h definovaný počet tiků systému OS_TICKS_PER_SEC například na hodnotu 100 a zavoláme tuto funkci s argumentem 50, bude úloha uspána na půl sekundy.

OSTimeDlyHMSM() nabízí uživateli možnost definovat mnohem delší uspání úlohy **nezávisle na počtu systémových tiků za sekundu** v podobě zadání hodin, minut, sekund a milisekund. Maximální hodnota je u hodin 255, u minut a sekund 59 a milisekund 999. Výsledná hodnota je **zaokrouhlena k nejbližší vyšší hodnotě systémového tiku**. Jako příklad poslouží uspání úlohy pomocí této funkce na 33ms při systémovém tiku 100 Hz. Systémový tik je generovaný každých 10 ms, tudíž uspání této úlohy nebude trvat 33 ms, ale 40 ms.

Pro testování těchto dvou funkcí zajišťujících uspávání úloh je vytvořena testovací aplikace TM-DlyHMSM.

### 6.2.1 TM-DlyHMSM

Na této testovací aplikaci je názorně ukázán implementační a výkonnostní rozdíl obou funkcí sloužících k uspávání úlohy. Ve zdrojovém kódu je ve výchozí variantě použita funkce OSTimeDly(). Použití OSTimeDlyHMSM() je zakomentováno.

V systému běží pro uživatele LED simulované tři úlohy pojmenované jako v TM-Creat-Delet `LeftTask()`, `MiddleTask()` a `RightTask()`. Každá z těchto úloh má totožný kód, a to po půl sekundě změnit hodnotu LED této úloze náležejícím (diody blikají).

Při testování obou variant funkcí sloužících k uspání úlohy byla časovačem měřena doba běhu uspávací funkce. Dále byla postupně snižována velikost zásobníku těchto úloh tak, aby si úlohy vzájemně zásobníky nepřepisovaly, což by vedlo k pádu aplikace. Zásobník každé úlohy je na začátku inicializován pomocí funkce `OSTaskStkInit()`, tudíž je v každé úloze reálně o 10 B menší než je definován v aplikaci.

Porovnání naměřených časových hodnot a velikostí zásobníků při použití těchto funkcí je v následující tabulce 6.1.

| | `OSTimeDly()` | `OSTimeDlyHMSM()` | rozdíl |
|---|---|---|---|
| Čas běhu [µs] | 92 | 870 | 778 |
| Velikost zásobníku [B] | 23 | 89 | 66 |

Tabulka 6.1 Porovnání `OSTimeDly()` a `OSTimeDlyHMSM()`

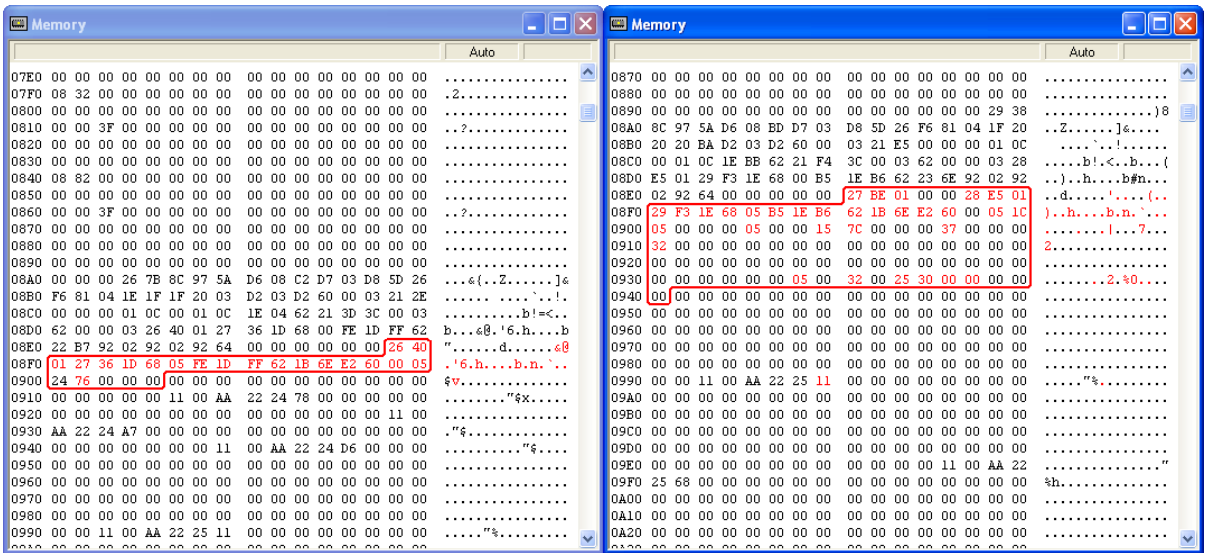
Hlavní příčinou takto velkého rozdílu je interní implementace těchto funkcí v samotném jádru systému µC/OS-II. `OSTimeDlyHMSM()` používá 32bitové proměnné pro přepočítání výsledné hodnoty uspání, kdežto `OSTimeDly()` používá pouze 16bitové proměnné. Samotný mikrokontroler je 8bitový a tak výpočet a práce s 32bitovými proměnnými zabere mnohonásobně více času. `OSTimeDly()` je přibližně 9,5krát rychlejší než `OSTimeDlyHMSM()`. Rozdíl 778 µs je při rychlosti jádra mikrokontroleru 48 MHz a průměrné době zpracování jedné instrukce propastný. Za 778 µs by mikrokontroler zpracoval při průměrné době zpracování jedné instrukce činící 3,9 taktu přibližně 9 500 instrukcí.

Tyto údaje mohou velmi ovlivnit odezvu systému, případně jej zpomalit do nepřípustných mezí, pokud by se jednalo o hard real-time aplikaci. Dále bychom neměli opomíjet i několikanásobně větší nároky na velikost zásobníku jednotlivých úloh. Pokud bychom měli cenově dostupnější mikrokontroler s menší pamětí RAM, nemusela by se nám všechna data to této paměti vejít. Je důležité si uvědomit, že na zásobník jsou ukládány veškeré lokální proměnné, návratové adresy při volání různých pomocných funkcí a také samotný kontext úlohy, pokud je přerušena obsluhou přerušení případně tikem systémového časovače.

Z programátorského hlediska je zde ale i velké nebezpečí pádu systému zapříčiněného právě nedostačující velikostí zásobníku při použití `OSTimeDlyHMSM()`. Tato chyba se velmi těžko hledá v případě, kdy programátor používá od začátku tvorby aplikace pro uspání úlohy funkci `OSTimeDly()` a najednou potřebuje delší interval uspání, než mu právě tato funkce umožňuje. Není jinou volbou než použít právě `OSTimeDlyHMSM()`. Systém µC/OS-II nemá žádnou kontrolu

rozsahu zásobníku při ukládání dat. Realizace je prakticky nemožná díky hardwarovému provedení mikrokontrolerů (není žádná možnost před zápisem na aktuální ukazatel vrcholu zásobníku pomocí hardware např. generovat přerušení, kde by následně mohla systémová úloha zkontrolovat platnou adresu pro zápis).

Sám jsem se s touto chybou a následným pádem systému setkal při vývoji aplikací a jedinou možností, jak tuto chybu hledat, je mít kvalitní real-time debuger, ve kterém lze v paměti RAM najít zásobníky jednotlivých úloh, postupně krokovat program a sledovat, jestli nedošlo k přepsání zásobníku jiné úlohy. Na následujícím obrázku 6.3 jsou ilustrativně zobrazeny zásobníky úloh z debugeru po provedení obou uspávacích funkcí. Červeně jsou hodnoty, které jsou na zásobník zapsány právě v průběhu obou funkcí pro uspání úlohy. Velikost zásobníku je znázorněna červeným orámováním paměti. Na první pohled je patrný rozdíl ve velikosti obou zásobníků. Vpravo je `OSTimeDly()` vlevo `OSTimeDlyHMSM()`.




Obrázek 6.3 Zásobníky po vykonání `OSTimeDly()` a `OSTimeDlyHMSM()`

# 6.3 Synchronization and communication

V mnoha případech potřebujeme, aby úlohy mezi sebou komunikovaly, případně předávaly data či statusy svého běhu. Lze to řešit pomocí globálních proměnných s použitím kritické sekce, kdy běh úlohy nemůže být přerušen žádnou jinou úlohou ani obsluhou přerušení, nicméně samotné úlohy pak musí použít polling na daný zdroj dat případně jinak softwarově ošetřovat různé možné stavy. Nehledě na to, že úlohy běží v systému se při neúspěšném dotazu na zdroj zase uspí a po čase periodicky opět zkouší, zda-li se jim povede přistoupit.

Systém µC/OS-II nabízí programátorovi širokou škálu komunikačních a synchronizačních prvků jako jsou semafory, mutexy, mailboxy, fronty zpráv a vlajky událostí. Ve většině případů lze

použít tyto prostředky jako blokující i neblokující. Neblokujících prostředků komunikace a synchronizace se dá využít především u rutin přerušení, případně tam, kde je nutné vždy vykonat nějakou další periodickou činnost. Přerušení může být generováno náhodně z vnějšího zdroje, tudíž nemusí být daná podmínka splněna pro odblokování, nicméně obsloužit daný vektor přerušení je potřeba. Blokujících prostředků lze využít v případech, kdy nějaká úloha čeká na dokončení celého či částečného výpočtu jiné úlohy. V momentě, kdy je výpočet dokončen, odblokuje tato úloha daný zdroj a zablokovaná úloha provede svůj kód (pokud má vyšší prioritu než úloha provádějící dlouhý výpočet).

Veškeré tyto synchronizační a komunikační prostředky lze v systému vytvářet, ale i mazat. Pokud se uživatel rozhodne jakýkoliv takový prostředek smazat je velmi důležité, aby předtím smazal i všechny úlohy, které na s tímto prostředkem pracují. Samotné **povolení** těchto **synchronizačních a komunikačních prostředků** je prostřednictvím souboru ***os_cfg.h***.

V následujících ukázkových aplikacích bude popsán semafor blokující a neblokující a blokující zasílání zpráv.

### 6.3.1 SC-OSSemAccept a SC-OSSemPend

Tyto dvě aplikace využívají jako synchronizačního prostředku semafor. SC-OSSemAccept jako neblokující a SC-OSSemPend i jako blokující.

Semafor je v systému μC/OS-II implementován jako ukazatel na datový typ `OS_EVENT`, může nabývat hodnot 0 až 65 535. Pokud chceme použít semafor jako signalizaci nějaké události či synchronizaci s jinou úlohou, inicializuje se při vytvoření pomocí `OSSemCreate()` na 0. V momentě, kdy daná úloha má semafor nastaven a přistupuje k němu jako blokující tak jeho hodnotu dekrementuje a provede příslušný kód úlohy. V případě, že přistupuje jako neblokující, periodicky po době svého uspání testuje stav semaforu a podle jeho stavu provede jednu z větví zdrojového kódu.

V obou případech použití semaforu se aplikace skládá ze dvou úloh, kde `LeftTask()` má vyšší prioritu, svůj běh detekuje rozsvícenými LED `PTE2` a `PTE3` a nastavuje semafor detekující událost po třetím běhu tého úlohy. Úloha `MiddleTask()` má nižší prioritu, běh detekuje pomocí LED `PTF1` a `PTC2`.

Rozdíl mezi během těchto aplikací není okem patrný a však v případě, kdy by měla `MiddleTask()` vyšší prioritu a semafor by byl použit jako neblokující, tak by přerušovala `MiddleTask()` periodicky běh méně prioritní úlohy `LeftTask()` a vykonávala část kódu, který nečeká na nastavený semafor.

Vždy záleží na situaci, v jaké chce uživatel semafor použít, pokud je použit jako **blokující**, tak se již **nepoužívá** žádná funkce pro **uspání úlohy**. Systém μC/OS-II nám díky preemtivnímu

plánování nabízí širokou škálu různých využití těchto synchronizačních a komunikačních prostředků. Nicméně skrývá i jistá zákoutí při případném ladění a vyhledávání chyb v běhu systému, pokud je nepoužijeme korektně.

### 6.3.2 SC-Messages

Dalším často používaným způsobem komunikace a synchronizace současně je použití mailboxu, tedy poštovní schránky.

Poštovní schránka je opět implementována jako ukazatel na datový typ `OS_EVENT`, **navíc** má ale **vůdči semaforu** definovaý **buffer** pro zprávy. Zde je na uživateli, jaký datový typ potřebuje předávat prostřednictvím zaslání zprávy. Může to být základní datový typ, ale i struktury programátorem vytvořené, případně pole vybraného datového typu.

V ukázkovém příkladu je jako buffer použit datový typ `INT8U`, který nám stačí pro komunikaci a současné předání dat o pozici LED, kterou chceme rozsvítit. V úloze `LeftTask()` simulujeme pomocí dvou for cyklů postupné zasílání hodnot 1 až 8 a následně 7 až 1 vždy po 100 ms. Po odeslání těchto hodnot je úloha na kratší čas uspána. Úloha `MiddleTask()` čeká na data s využitím blokující funkce pro příjem zpráv, pokud zprávu přijme bez bych, zhasne všechny diody a rozsvítí diodu na právě přijaté pozici.

Takto je simulován běh jedné diody zleva doprava a zpět. Opět lze v aplikaci měnit libovolně doby uspání a tak zrychlovat či zpomalovat rychlost běhu diody v obou směrech i pauzu ve výchozí poloze.

## 6.4 ISR test

Systém µC/OS-II má pro každou platformu své limity, při kterých jej lze ještě kvalitně používat. Jedním z následujících limitů je i schopnost vykonávat kód jednotlivých úloh, při zvyšující se frekvenci přerušení ať již generovaného interním zdrojem či externím.

Architektura systému sama o sobě vyžaduje generování přerušení pro systémový tick, aby byl systém schopen přepnout na případnou úlohu s vyšší prioritou. V tomto testovacím příkladě je systémový tick generován frekvencí 100Hz. Úlohy v systému běží bez jakýchkoliv problémů, cílem této aplikace je najít limit pro maximální frekvenci přerušení, kdy ještě systém vykonává kód jednotlivých úloh.

Pro testování je zvolen časový interval jedné sekundy, který je měřen pomocí obvodu RTC, který po jedné sekundě generuje přerušení. Obsluha tohoto přerušení je zacyklení programu pomocí nekonečného *for* cyklu. Pomocí debugeru do této obsluhy vložíme breakpoint, tudíž aplikace poběží právě jen a jen jednu vteřinu. Velmi důležité je si připomenout tabulku na obrázku 2.2, kde máme

seřazeny vektory přerušení od nejnižší priority (nahoře) po nejvyšší (dole). Z tabulky je patrné, že obsluha RTC obvodu bude mít nejnižší prioritu, následně bude časovač 2 a jeho kanál 0, který je použit pro generování systémového ticku. Vyšší prioritu má časovač 1 a jeho přetečení, který v testovací aplikaci použijeme pro testování při jaké frekvenci přerušení ještě běží samotné úlohy. Nejvyšší prioritu má instrukce `SWI`, která je použita pro přepínání kontextu.

V samotné aplikaci je potřeba provést několik zásadních změn a to:

- *vectors.c*
  - na vektor obsluhy přetečení časovače 1 nastavit rutinu pro obsluhu `Timer1OF` a přidat její prototyp
- *os_cpu_a.s*
  - vytvořit rutinu obsluhy `Timer1OF`, zde se použije rutina `Tmr_TickISR`, upraví se návěští a zavolá se správná obluha `Timer1OF_Handler`
- *bsp.c*
  - definice handleru `Timer1OF_Handler`. Je to tělo obsluhy přerušení, v souboru *os_cpu_a.s* je pouze nutný kód pro správnou práci se zásobníkem při přerušení a systémem μC/OS-II
  - definice globální proměnné, která je inicializována na hodnotu 0. Obsluha přerušení od časovače 1 pouze smaže flag přetečení a tuto proměnnou inkrementuje (víme kolikrát časovač v intervalu jedné sekundy přetekl).
- *app.c*
  - definice globálních proměnných a inicializace na hodnotu 0 pro úlohy běžící v systému. Tyto proměnné slouží pro počítání zahájení běhu úlohy a jejího dokončení.
  - úprava jednotlivých úloh pro práci s globálními proměnnými zaznamenávajícími jejich běh

Při běhu aplikace bez přerušení generovaného časovačem 1 byly úlohy v intervalu jedné sekundy měřeném RTC obvodem spuštěny:

- LeftTask() spuštěno / dokončeno 7 / 7
- MiddleTask() spuštěno / dokončeno 4 / 4
- Right Task() spuštěno / dokončeno 3 / 3

V následující tabulce jsou shrnuta data při různých frekvencích přerušení od časovače 1, na prvním řádku jsou uvedeny hodnoty bez přerušení. Každá úloha má uveden počet startů a dokončení úlohy. Časovač 1 počet uskutečněných ISR a čas odpovídající tomuto počtu (měl by se blížit hodnotě jedné vteřiny, v obsluze RTC nastavené na generování přerušení po 1 sekundě je breakpoint).

| Frekvence přerušení [Hz] | LeftTask | MiddleTask | RightTask | Časovač 1 | |
|---|---|---|---|---|---|
| | start / end | start / end | start / end | Počet ISR | Čas [ms] |
| - | 7 / 7 | 4 / 4 | 3 / 3 | - | - |
| 10 | 7 / 7 | 4 / 4 | 3 / 3 | 9 | 900 |
| 100 | 7 / 7 | 4 / 4 | 3 / 3 | 97 | 970 |
| 1000 | 7 / 7 | 4 / 3 | 3 / 2 | 978 | 978 |
| 1428 | 7 / 7 | 4 / 3 | 3 / 2 | 1394 | 976 |
| 3952 | 7 / 7 | 4 / 3 | 3 / 2 | 3857 | 976 |
| **4000** | **1 / 1** | **1 / 1** | **1 / 0** | **617** | **154** |

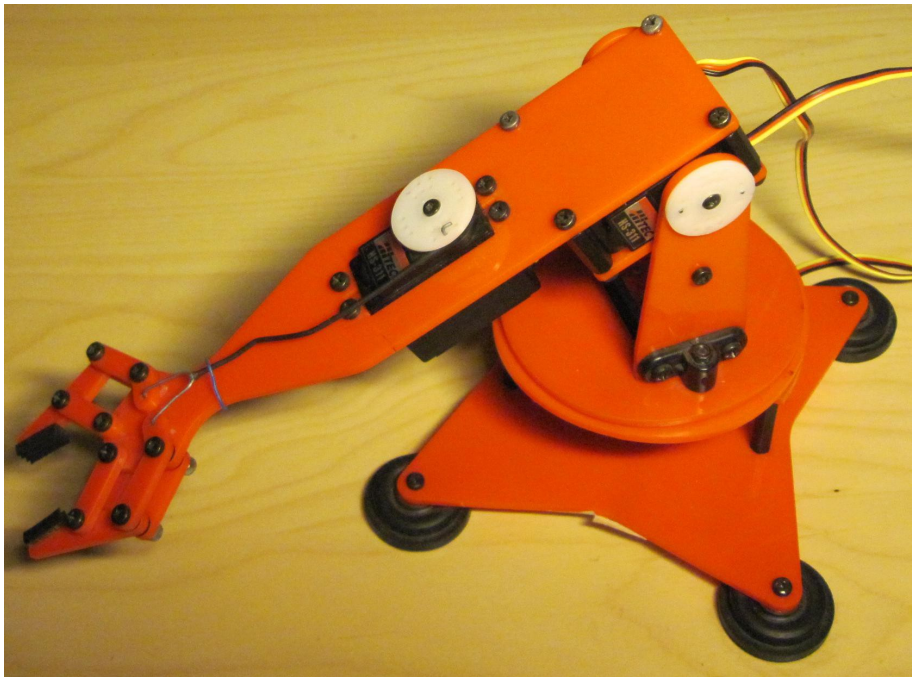
Tabulka 6.2 Sumarizce naměřených a zjištěných hodnot pro maximální možnou frekvenci ISR

Z tabulky je patrné, že hraničící hodnota je přibližně 3950Hz, čemuž odpovídá přerušení každých 253µs. Nicméně je za potřebí vzít v úvahu, že obslužná rutina prakticky nevykonávala žádný kód. Pokud by byla tato rutina výpočetně náročnější nebo by bylo v systému více periodických přerušení může být tato hodnota mnohem nižší. Při tomto testu byly v systému pouze přerušení od RTC obvodu s frekvencí 1Hz, od systémového časovače a přepínání kontextu s frekvencí 100Hz a testující přerušení od časovače 1 s proměnnou frekvencí podle nastavené modulo hodnoty a předděličky frekvence.

Již při generování přerušení každou 1ms neběžely úlohy v systému tak, jako bez přerušení od časovače 1. Osobně bych jako maximální frekvenci pro přerušení v systému použil hodnotu 3900Hz což jest přerušení každých 256µs za předpokladu velmi krátké ISR.

# 7 RoboArm RT aplikace

Jako hlavní RT aplikaci využívající různé periferie na desce DEMOJM byla vybrána aplikace ovládající robotický manipulátor RoboArm, obrázek 7.1.



Obrázek 7.1 RoboArm

## 7.1 Ovládání robotické ruky

Pohyb jednotlivých částí robotické ruky zajišťují servomotorky firmy Hitec HS-311 (obrázek 7.2), jedná se o standardní servomotorky levnější cenové kategorie velmi hojně používané v RC modelech jako jsou větší lodě a letadla, případně různá další automobilová manipulační technika.


Obrázek 7.2 Hitec HS-311

Servo může být dle [8] napájeno napětím v rozsahu 4,8 až 6,0 V při spotřebě 180mA bez zatížení při pohybu. Rozsah otáčení je 180°, ten je však využit plně pouze při otáčení ramene kolem svislé osy. Pro sklápění ramene a pohyb čelistí stačí cca poloviční úhel.

Servo je ovládáno pomoci pulzně šířkové modulace (PWM), kde úhel natočení závisí na šířce kladného pulzu. Perioda generovaného signálu musí být mezi 15 až 25ms. Neutrální poloze 0° by měla dle [8] odpovídat šířka pulsu 1,5ms, nicméně při zprovoznění robotické ruky bylo zjištěno, že tomu tak není. Výchozí hodnoty byly voleny na základě empirického testování, stejně tak jako krajní hodnoty. Samotný pohyb není nijak přesný, servo je spíše určeno pro rychlou změnu polohy. Ač jsme schopni softwarově generovat velmi malé změny šířky pulzu PWM, tak servo se pohybuje spíše trhavým pohybem nežli plynule.

Napájecí napětí je přivedeno dvěma vodiči, červený je pro napájecí napětí Ucc a černý je zemnící vodič GND. Žlutý vodič pak slouží k přívodu PWM signálu pro otáčení servomotorku.

## 7.2 Návrh aplikace

Mikrokontroler JM60 má časovače schopné generovat PWM signál na několika kanálech, čehož využijeme pro ovládání servomotorků. Na desce DEMOJM je umístěn 3osý analogový akcelerometr, který nám poslouží jako zdroj dat, podle kterých budeme pohybovat robotickou rukou kolem vlastní osy a sklápět či zvedat rameno. Pro převod analogových dat na digitální využijeme na čipu AD převodníku. Čelisti budou ovládány pomocí tlačítek připojených na port `PTG`.

V systému vytvoříme úlohy pro načítání dat z akcelerometru `AccelTask()`, načítání dat z tlačítek `ButtonTask()`, úlohu pro změnu šířky PWM pulzů `PWMTask()` a úlohu detekující běh systému `SystemRunInfo()`.

## 7.3 Implementace

Samotná implementace této RT aplikace vyžaduje definování konstant programu, speciální datové struktury pro ukládání načtených dat z akcelerometru, pomocných funkcí pro výpočty v jednotlivých úlohách a samozřejmě samotné úlohy.

### 7.3.1 Konstany, datové typy, globální proměnné

Pro ukládání dat jednotlivých os načtených z akcelerometru je vytvořen uživatelský datový typ `accelData` skládající se ze struktur `axisData` (pro každou osu zvlášť), která obsanuje dvě položky `reading` a `result`, a ukazatele na předchozí a následující prvek `accelData`. Tento uživatelský datový typ je pak použit v globálním kruhovém bufferu `AccelDataBUF` o velikosti

SAMPLES obsahující vzájemně provázané struktury accelData. Dalšími globálními proměnnými jsou vyfiltrované hodnoty dat z akcelerometru pro obě osy Ax a Ay a ukazatel na typ OS_EVENT použitý jako semafor.

Další velmi důležité konstanty jsou velikosti jednotlivých zásobníků úloh, priority úloh v systému a horní a dolní mez šířky PWM pulzu pro jednotlivé kanály(otáčení ruky, pohyb ramene nahoru a dolů a sevření a otevření čelistí).

### 7.3.2 Pomocné funkce

Aby byl kód programu přehlednější, jsou použity různé pomocné fuknce, které jsou volány ze samotných úloh. Zde je dobré připomenout, že každé volání funkce v úloze zabere na zásobníku 2B (uložení návratové adresy z pomocné funkce).

#### CreateTasks

Pro zavádění úloh do systému μC/OS-II je použit způsob, kdy při startu systému vyvolaného funkcí OSStart(), je zavedena pouze jedna úloha s nejvyšší prioritou InicialTask(), která prvotně inicializuje všechny periferie potřebné pro běh aplikace a následně zavolá právě CreateTasks(), která vytvoří všechny zbylé úlohy.

#### accelData_Init

Funkce inicializující kruhový buffer AccelDataBUF, jejími parametry jsou ukazatel na globální proměnnou bufferu a jeho velikost. Po inicializaci jsou všechny prvky správně provázány i v případě, že buffer obsahuje pouze jednu položku.

#### FiltrAccelData

Filtrace dat v kruhovém bufferu. Data jsou filtrována pomocí IIR filtru. Parametry funkce jsou ukazatel na globální proměnnou bufferu a jeho velikost.

#### GetPWMChange

Pomocná funkce normalizující vyfiltrovaná data předána argumentem. Vrací hodnotu typu INT8U, která je v intervalu <0, 15>. Na základě vrácené hodnoty je pak zvoleno snížení nebo zvýšení šířky pulzu PWM signálu.

### BSP_Init

Funkce volána inicializační úlohou v systému. Inicializuje veškeré periferie, kanály PWM časovače pro ovládání robotické ruky, tlačítka pro ovládání čelistí a časovač generující systémový tik. Zdrojový kód je v souboru *bsp.c*.

### Accelerometer_Init

Inicializace AD převodníku pro převádění dat z akcelerometru. Nastavena 12bitová konverze a vstupní frekvence `bus_clock/2`. Funkce je volána z inicializační úlohy. Zdrojový kód v souboru *accelerometer.c*.

### Accelerometer_Rd

Samotný převod načtené hodnoty z akcelerometru pomocí AD převodníku. Funkce vrací načtenou hodnotu. Jejím argumentem je kanál ze kterého bude načítat data (osa X nebo Y). Zdrojový kód v souboru *accelerometer.c*.

## 7.3.3 Řídící úlohy

### InicialTask

Úvodní úloha zavedená do systému, inicializuje veškeré periferie (`BSP_Init` a `Accelerometer_Init`) a kruhový buffer (`accelData_Init`), následně vytvoří všechny zbylé úlohy v systému (`CreateTasks`) a sama sebe ze systému smaže. Dále běží pouze řídící úlohy.

### SystemRunInfo

Jednoduchá úloha detekující běh systému blikáním LED číslo 8 (`PTD2`). Tato úloha má nejnižší prioritu v aplikaci. Neguje hodnotu výsputního pinu po 500ms.

### ButtonTask

Úloha s nejvyšší prioritou v aplikaci (po smazání `InicialTask`). Periodicky načítá hodnoty z portu PTG (tlačítka) po 50ms. Pokud je stisknuté tlačítko ovládající čelisti (PTG1 nebo PTG2) zvýší či sníží šířku pulsu na kanálu 0 časovače 1. Stisk tlačítka a změna šíře generovaného PWM pulzu je také uživateli deteková rozsvícením LED (PTG1 LED 7, PTG2 LED 6). Pokud není žádné tlačítko stisknuto zhasne obě LED.

### AccelTask

Načítá data z akcelerometru po 10ms, vždy do následujícího prvku kruhového bufferu. Pokud je načteno více vzorků provede filtraci dat a do globálních proměnných zapíše izolovaně v kritické sekci aktuální hodnoty. Při svém čtvrtém běhu (po 40ms) nastaví semafor a odblokuje běh úlohy pro aktualizaci šířky půlzu PWM signálu.

### PWMTask

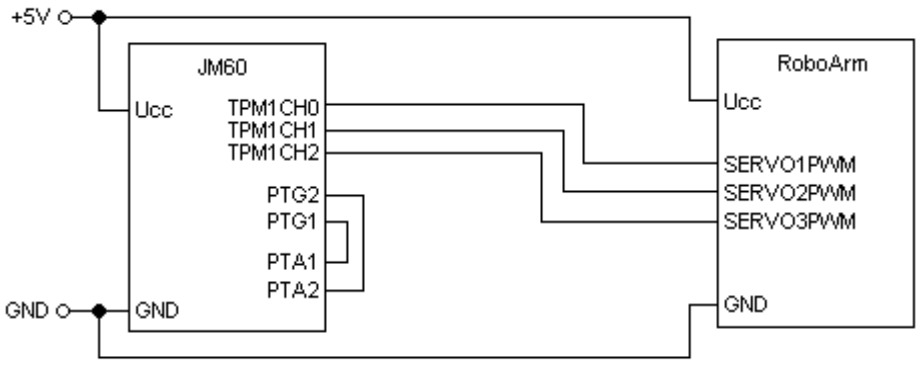
Úloha měnící šířku PWM pulzu na kanálech 1 a 2 časovače 1 (pohyb ramene a otáčení kolem svislé osy). Úloha čeká na nastavený semafor od accelTask, jakmile je nastavený, přepočítá pomocí GetPWMChange vyfiltrované hodnoty do normalizovaného intervalu a podle této hodnoty pomocí switch provede případnou změnu šířky pulzů na obou kanálech. Pokud bychom již narazili na meze definované konstantami, tak nastaví jako šířku pulzu právě tuto mez.

## 7.4 Realné zapojení RoboArm

Samotný mikrokontroler při testování funkčnosti a konstant pro omezení pohybu jednotlivých servomotorků zvládl dodávat dostatečnou energii jednomu servomotorku. Nicméně při zapojení všech tří serv to již nedokázal. Z těchto důvodů bylo nutné pořídit externí zdroj napájení.

Jako dostačující zdroj byl vybrán stabilizovaný zdroj stejnosměrného napětí MINWA 5V/1500mA. Na desce DEMOJM je nutné změnit zdroj napájecího napětí pomocí jumperu na `REG_VDD` a na konektoru `MCU_PORT` propojit port `PTA` s portem `PTG` piny `PTG1` a `PTG2`.

Samotné kanály 0, 1 a 2 časovače 1 jsou vyvedeny taktéž na konektor `MCU_PORT` piny dle obrázku 6.1. Kanál 0 ovládá čelisti a jeho šíře pulzu je měněna pomocí tlačítek PTG1 a PTG2. Kanál 1 ovládá pohyb ramene nahoru a dolů. Kanál 2 otáčí robotickou rukou kolem svislé osy. Na následujícím obrázku je zjednodušené schéma zapojení (vnitřní propojení mikrokontroleru a periferií na desce DEMOJM není známo).




Obrázek 7.3 Zjednodušené schéma propojení JM60 a RoboArm

# 8 Závěr

Cílem této diplomové práce byla implementace real-time operačního systému μC/OS-II na platformě freescale JM60, vytvoření reálné řídící RT aplikace a dalších testovacích aplikací samotného systému μC/OS-II a jeho případných limitů.

Uživatelská příručka by měla být nápomocna běžnému uživateli při zprovoznění systému μC/OS-II na této platformě, případně na jiných čipech se stejným CPU HCS08.

Testovací aplikace ukázaly možné problémy při tvorbě aplikací, na které si musí dávat programátor velký pozor. Zároveň ale i ukázaly komplexnost celého systému μC/OS-II, který nabízí programátorovi široké spektrum použití.

Snažil jsem se co nejvíce využít možné periferie jak na čipu JM60, tak i hardware na desce DEMOJM, proto jsem zvolil jako řídící RT aplikaci ovládání robotické ruky pomocí akcelerometru a tlačítek. Výsledná hlavní řídící aplikace robotické ruky je plně funkční, lze ji ještě případně rozšířit, nicméně vzhledem k tomu, že servomotorky nelze jemně ovládat, považoval bych další rozšiřování spíše jako praxi s prací s tvorbou apliakcí v systému μC/OS-II.

# Literatura


[1] Freescale Semiconductor: *MC9S08JM60 Series Data Sheet* [online]. Rev.3 1/2009. [cit. 18.12.2011]. Dostupný z WWW http://www.freescale.com/files/microcontrollers/doc/data_sheet/MC9S08JM60.pdf.

[2] Strnadel, J.: *Real-time operaèní systémy (ROS) - Studijní opora* [online]. FIT VUT v Brně, 2006-10-10 [cit. 2009-12-04].

[3] Ganssle, J.: *RTOS dissatisfaction* [online]. 2006-5-22 [cit. 26.12.2011]. Dostupný z WWW http://www.ganssle.com/rants/rtosdissatisfaction.htm.

[4] Srovnal, V.: *Operační systémy pro řízení v reálném čase*. VŠB TU, Ostrava, 1. vydání, ISBN 80-248-0503-0.

[5] Labrosse, J. J.: *MicroC/OS-II: The Real-Time Kernel, Second Edition*. CMP Books, 2002, ISBN 1-57820-103-9.

[6] Freescale Semiconductor: *HCS08 familyReference Manual* [online]. Rev.2 5/2007. [cit. 5.4.2013]. Dostupný z WWW http://www.freescale.com/files/microcontrollers/doc/ref_manual/HCS08RMV1.pdf

[7] Freescale Semiconductor: *DEMOJM User Manual* [online]. Rev.1.02 9/2008. [cit. 20.7.2013]. Dostupný z WWW http://www.freescale.com/files/microcontrollers/doc/user_guide/DEMOJMUM.pdf

[8] Autor neznámý: HS-311 Standard [online]. Dostupný z WWW http://www.letmodel.cz/technical-data/serva/hs-311.pdf

# Seznam příloh

**příloha 1** CD se všemi aplikacemi v práci popsanými (složka Source), včetně dostupných PDF (složka Doc) a videa s ukázkou ovládání RoboArm (složka Video)