VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

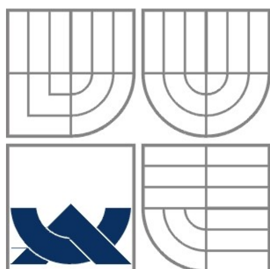
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

NÁVRH A IMPLEMENTACE JADER REAL-TIME OPERAČNÍCH SYSTÉMŮ BĚŽÍCÍCH NA HC08

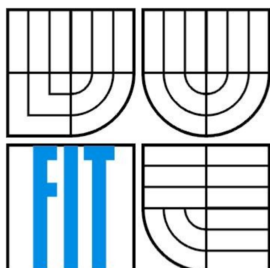
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. Jan Bednář
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

NÁVRH A IMPLEMENTACE JADER REAL-TIME OPERAČNÍCH SYSTÉMŮ BĚŽÍCÍCH NA HC08

DESIGN AND IMPLEMENTATION OF REAL-TIME OPERATING SYSTEM KERNELS RUNING ON HC08

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. Jan Bednář
AUTHOR

VEDOUCÍ PRÁCE — Ph.D. Ing. Josef Strnadel
SUPERVISOR

BRNO 2008

Abstrakt

Hlavním významem celé práce je testování jader real-time os na platformě HC08. Pro porovnání jsou použity jádra vyzývací smyčky, mechanizmu RM a EDF, a volně dostupných systémů FreeRTOS a QP. V práci jsou popsány postupy tvorby, získavání a realizování testovacích prostředí, zhodnocení na základě testu provedených na platformě HC08 a poznatky z programování pro jednotlivé typy jader real-time OS.

Klíčová slova

HC08, real-time, jádro, plánovač, RM, DM, EDF, LFF, TCB, operační systém.

Abstract

The project is aimed at testing the kernels of real-time OS within the HC08 platform. The RM, EDF and polled loop mechanisms are being compared as well as freely available FreeRSTOS and QP systems. The project also incorporates descriptions of techniques used in the development, obtaining and the implementation of test environments. The evaluation is based on the tests made within the HC08 platform and the knowledge gained from the programming for every individual type of real-time OS.

Keywords

HC08, real-time, kernel, scheduler, RM, DM, EDF, LFF, TCB, operating system.

Citace

Bednář Jan: Návrh a implementace jader real-time operačních systémů běžících na HC08. Brno, 2008, semestrální projekt, FIT VUT v Brně.

Návrh a implementace jader real-time operačních systémů běžících na HC08

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Josefa Strnadela Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Jméno Příjmení
Datum

Poděkování

Chtěl bych poděkovat všem co mi pomohli se získáváním zdrojů informací a obecně přispěli ke vzniku této práce. A Josefu Strnadelovi hlavně za pomoc při získávání použitého mikrokontoroléru.

© Jan Bednář, 2008.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah.....1
1 Úvod.....3
2 Popis platformy mikrokontroleru HC08.....4
2.1 Paměť.....6
2.2 Analogově digitální převodník.....7
2.3 Přerušovací podsystém.....8
3 Real-time OS.....11
4 Druhy úloh.....12
4.1 Synchronní úlohy.....12
4.2 Asynchronní úlohy.....12
4.3 Periodické úlohy.....12
4.4 Aperiodické úlohy.....12
5 Dělení plánovacích mechanizmů.....13
5.1 Pseudo jádra.....13
5.1.1 Vyzývací smyčka.....13
5.1.2 Cyklické provádění.....13
5.1.3 Stavově řízený kód.....14
5.1.4 Spolupracující úlohy.....14
5.2 Využití přerušovacího podsystému.....14
5.3 Systémy pracující v popředí a pozadí.....14
5.4 Model TCB.....15
5.5 Plánovací mechanizmy.....15
5.5.1 RM.....15
5.5.2 ID/DM.....15
5.5.3 EDF.....15
5.5.4 LLF.....15
5.5.5 Hybridní plánovací postupy.....16
5.5.6 Závislé úlohy na pořadí.....17
5.5.7 Závislé úlohy na sdílených prostředcích.....17
6 Využití stávajících systémů.....18
6.1 Quantum Platform.....18
6.2 FreeRTOS.....19
6.3 Nový systém.....20
7 Realizace.....21

7.1 Příprava........21
7.1.1 Vývojové prostředí........21
7.1.2 Vhodný typ microcontroléru........21
7.1.3 QP........22
7.1.4 Upravení FreeRTOS........22
7.1.5 Změna mikrokontroléru........23
7.1.6 Ovládání a řízení zavlažování kolem rodinného domku........27
7.2 Testovací program........28
7.2.1 Vyzývací smyčka........28
7.2.2 RM........29
7.2.3 EDF........30
7.2.4 FreeRTOS........31
7.2.5 QP........32
7.3 Zhodnocení programování........33
8 Testy........34
8.1 Perioda........35
8.2 Náročné výpočty........36
8.3 Nedělitelný program........37
8.4 Zahlcení plánovače........38
8.5 Náhodné události........39
8.6 Shrnutí výsledků........40
........41
Literatura........43

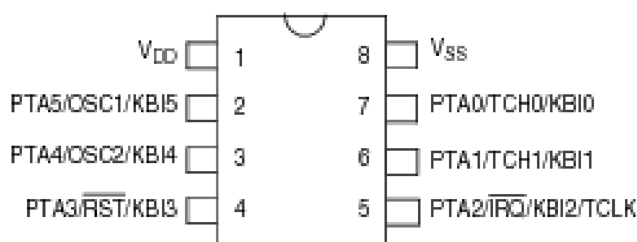
1 Úvod

Operační systémy jsou převážně doménou velkých serverů a osobních počítačů. Přesto díky zvyšujícímu se výkonu vestavěných systémů se dostávají operační systémy i do této oblasti. Umožňují tím pohodlnější prostředí pro programátory, tím že je odstiňují od nutnosti správy HW a nutnosti se starat o bezkoliznost s ostatními aplikacemi nutnými pro obsluhu dalších částí systému. Tady ale naráží na nové problémy. Jedním z hlavních problémů je víceúlohovost takového systému. Kde u serveru si lidé při přetížení prostě prohlédnou stránku o nedostupnosti služby, nebo při používání nějakého programu na svém osobním počítači chvíli vyčkají než se dokončí zpracování. U řídících vestavěných systémů může prodleva v obsluze nějaké aplikace znamenat oběti na životech. Příkladem budiž automatické zavírání dveří. Kdyby na popud snímače včas nezareagoval systém že dveře nejdou zavřít protože v nich je člověk, rozmačkaly by ho. A právě obsluhu plánovacích mechanizmů jednotlivých úloh tak aby nedocházelo k tomu, že některá úloha nebude zpracována včas, zajišťuje jádro real-time operačních systémů. Dalším neméně důležitým faktorem je náročnost takového systému na zdroje a platformu na které běží. Samozřejmě že je možné vytvořit a existují velké komplexní systémy které zvládají naplánovat i velice náročné aplikace ve velkém množství. Ale tyto systémy mají vysoké nároky na HW a i když se technologie stále zlepšuje, nikdy nebude dost dobrá na to aby dokázala všechno co je potřeba. Navíc jsme tlačeni trhem a pokud vyrobíme dané zařízení levněji a lépe, tak ho prodáme a vyděláme, jinak proděláme a zkrachujeme. Proto vyvstává otázka jak dostat výhody real-time operačních systémů do nejmenších a nejlevnějších řadičů. Aby byl zachován komfort rychlého a pohodlného psaní programů a zároveň spolehlivost a malé nároky na HW, které jsou nutné pro nasazení v těchto platformách. Tato práce se bude zabývat implementací jader real-time operačních systémů na platformě HC08. Cílem je získat srovnání různých jader, jejich možné použití, schopnosti plánování a řízení. Zjištění případných možných úprav a přizpůsobení pro danou platformu aby se dosáhlo co nejlepšího využití prostředků při zachování bezkolizního provozu.

Mikrokontroler HC08 poskytuje pro real-time operační systémy poměrně malý výkon. Největší problém představuje paměť. K dispozici je 128-1024B paměti a kupříkladu systémy OSEK nebo windows CE takovýto blok paměti vyhrazují pro uložení informací o jediné běžící úloze. Dalším problémem bude případná podpora obsluhy nějakého HW. Spousta programových částí využívá pro svůj běh a řízení přerušení mikrokontroleru. Tyto ale musí systém u některých řešení plánovacích mechanizmů odstínit aby nedocházelo k situaci, kdy přerušení zabrání jádru vykonávat obsluhu a řídit ostatní úlohy. S výkonem samotného procesoru může nastat problém až u některých náročnějších plánovacích mechanizmů, kde by docházelo k prohledávání stavového prostoru možného naplánování úloh. A co se týká paměti programu, tak 64KB poskytuje dostatečný prostor jak pro jádro, tak obslužné úlohy.

2 Popis platformy mikrokontroleru HC08

Mikrokontrolery řady HC08[4] jsou 8-bitové mikropočítače firmy Motorola (nyní Freescale Semiconductor). Jsou to jedny z nejrozšířenějších mikrokontrolerů díky nízké ceně a univerzální výbavě. Mikrokontrolery jsou založené na procesoru Motorola 6800. Jde o CISCový procesor na bázi architektury von Neumana. Od jeho nejmenšího zástupce NITRON se 128B paměti RAM a 4KB paměti FLASH taktovaného na 8 MHz není předpoklad, čekat příliš velký výkon. Přesto v některých aplikacích může být velice výhodný.

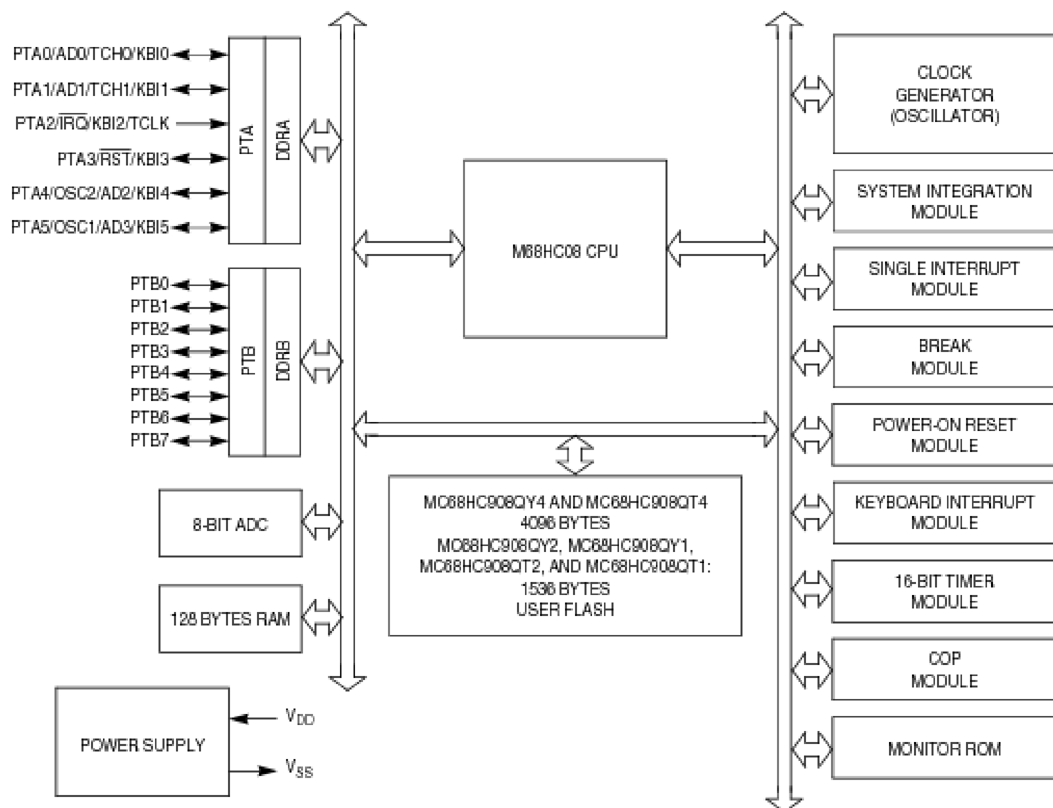


Obrázek 1. Pouzdro mikrokontroleru [4]

Kupříkladu u zabezpečení místností v budovách. Můžeme volit z několika možných řešení. V každém případě je ale na jedné straně množství senzorů v každé z chráněných místností objektu, které je potřeba současně vyhodnocovat, protože tak dávají informaci o stavu bezpečnosti dané místnosti. Jednou z možností je centrální počítač který bude napojen na všechny senzory a vše bude sledovat a vyhodnocovat. Toto řešení ale představuje bezpečnostní riziko v podobě jednoho řídícího bodu do kterého musí vést množství signalizačních vodičů a navíc musí být v cílovém řídícím bodu možnost jejich připojení. Pokud případný útočník přeruší tyto signalizační vodiče a nahradí vhodným zakončením získá přístup po celém objektu. Možné vylepšení je v dalších investicích, kdy nahradíme fyzické vodiče radiovým přenosem a ke každému snímači dodáme radiový vysílač a přijímač. Další a řekl bych že daleko pohodlnější a bezpečnější variantou je připojení všech senzorů v dané místnosti do jednoho řídícího bodu v každé místnosti, kde již na obsluhu pár signalizačních vodičů postačuje i výše zmíněný NITRON. Typů snímačů v místnosti nebude mnoho obvykle dva nebo tři. Jeden u vchodu pro identifikaci oprávněných vstupů a další uvnitř, hlídající pohyb nebo změnu teploty pokud je místnost uzavřena. Signály od snímačů stejného typu může obsluhovat jen jiná instance od stejné úlohy, takže v paměti flash stačí jen jeden kód pro každý snímač a protože informace ze snímačů není nutné snímat neustále, může v době nevyhodnocování snímačů běžet úloha zpracovávající informace ze senzorů. Každá místnost je chráněna samostatně a nehrozí přerušení signalizačních vodičů a díky nízké ceně mikrokontroleru není toto řešení ani příliš nákladné. Co je ale potřeba, je dobře naplánovat

spouštění jednotlivých úloh, aby nedošlo k zahlcení mikrokontroleru nebo ztrátě informace a tím porušení bezpečnosti. Jedním z možných řešení je právě implementace real-time operačních systémů. Systém ale musí být velice nenáročný na zdroje a přesto umožnit alespoň plánování jednotlivých úloh.

Mikrokontroler je vybaven generátorem hodin, čítačem/časovačem, systémem přerušení, analogově digitálním převodníkem, podporou pro připojení periferií. Což jsou právě komponenty potřebné pro nasazení ve většině řídících systémů. Řízení probíhá na základě vnějších signálů zpracovávaných přerušením v případě další číslicové součásti, čítači/časovači v případě nutnosti zpracování až po nějakém signalizovaném kvantu nebo na základě vypršení časového limitu. Pro připojení analogových snímačů využijeme analogově digitální převodník. A skoro každý systém bude potřebovat zpracovávat ovládání z venčí pomocí samostatného klávesnicového systému.



Obrázek 2. Blokové schéma mikrokontroleru [4]

Na obrázku blokového schéma 2 je zobrazena vybavenější varianta MC68HC908QT4, rozdíly jsou ale pouze ve velikosti FLASH paměti, což je uvedeno již v blokovém schématu a dále v přidání jednoho 8-bitového portu PTB, čímž získáme 16 pinové pouzdro. Mikrokontroler je napájen 5V kdy zvládá pracovat na již zmíněné frekvenci 8 MHz, při provozu na pouhých 3V dosáhneme jen poloviční frekvence 4 MHz.

2.1 Paměť

Architektura von Neumana znamená že do paměťového prostoru namapujeme jak paměť programu, tak paměť dat a ve výsledku tedy i pracovní a řídící registry. Z toho vyplývá, že velikosti paměti programu 64 KB nebude nikdy dosaženo. Na začátku jsou registry, pak je operační paměť a pak paměť programu. Na konci jsou některé funkční registry a přerušovací vektory jak ukazuje obrázek 3.

Adresa	Obsah
\$0000 ↓ \$003F	I/O REGISTERS 64 BYTES
\$0040 ↓ \$007F	RESERVED[1] 64 BYTES
\$0080 ↓ \$00FF	RAM 128 BYTES
\$0100 ↓ \$27FF	UNIMPLEMENTED[1] 9984 BYTES
\$2800 ↓ \$2DFF	AUXILIARY ROM 1536 BYTES
\$2E00 ↓ \$EDFF	UNIMPLEMENTED[1] 49152 BYTES
\$EE00 ↓ \$FDFF	FLASH MEMORY MC68HC908QT4 AND MC68HC908QY4 4096 BYTES
\$FE00	BREAK STATUS REGISTER (BSR)
\$FE01	RESET STATUS REGISTER (SRSR)
\$FE02	BREAK AUXILIARY REGISTER (BRKAR)
\$FE03	BREAK FLAG CONTROL REGISTER (BFCR)
\$FE04	INTERRUPT STATUS REGISTER 1 (INT1)
\$FE05	INTERRUPT STATUS REGISTER 2 (INT2)
\$FE06	INTERRUPT STATUS REGISTER 3 (INT3)
\$FE07	RESERVED FOR FLASH TEST CONTROL REGISTER (FLTCR)
\$FE08	FLASH CONTROL REGISTER (FLCR)
\$FE09	BREAK ADDRESS HIGH REGISTER (BRKH)
\$FE0A	BREAK ADDRESS LOW REGISTER (BRKL)
\$FE0B	BREAK STATUS AND CONTROL REGISTER (BRKSCR)
\$FE0C	LVISR
\$FE0D ↓ \$FE0F	RESERVED FOR FLASH TEST 3 BYTES
\$FE10 ↓ \$FFAF	MONITOR ROM 416 BYTES
\$FFB0 ↓ \$FFBD	FLASH 14 BYTES
\$FFBE	FLASH BLOCK PROTECT REGISTER (FLBPR)
\$FFBF	RESERVED FLASH
\$FFC0	INTERNAL OSCILLATOR TRIM VALUE
\$FFC1	RESERVED FLASH
\$FFC2 ↓ \$FFCF	FLASH 14 BYTES
\$FFD0 ↓ \$FFFF	USER VECTORS 48 BYTES

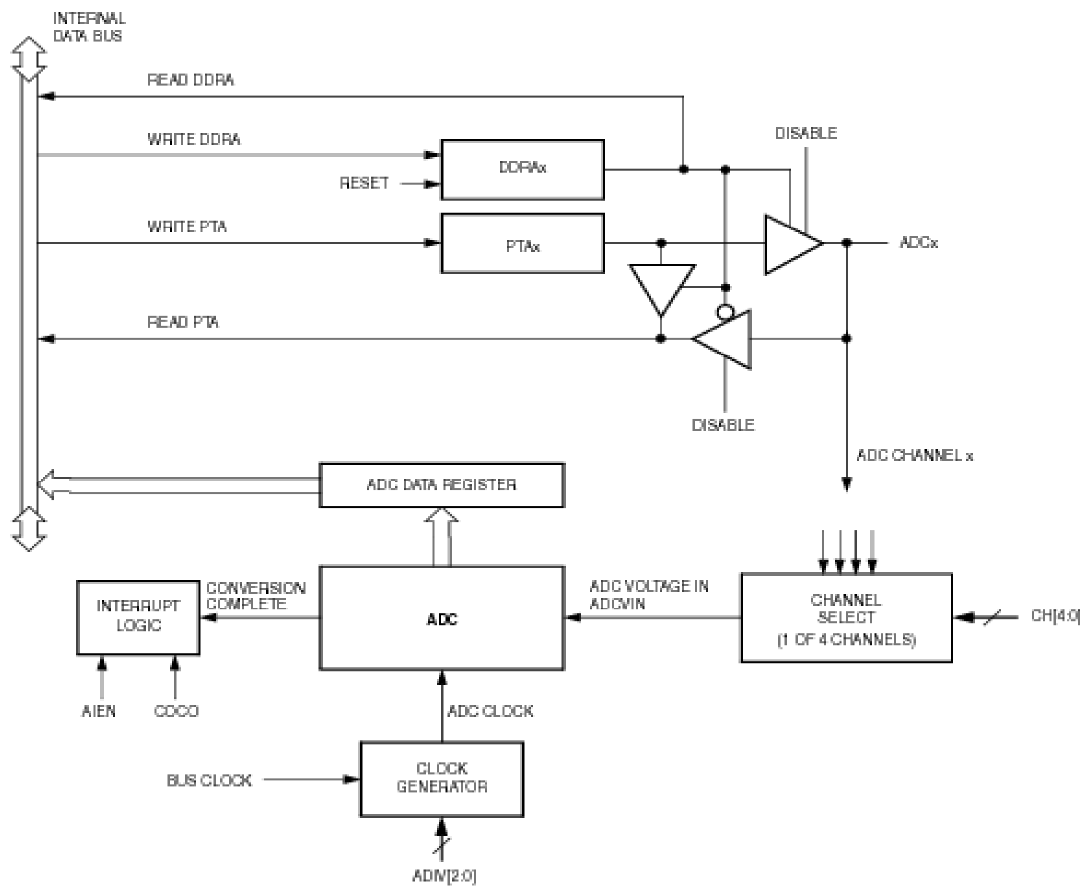
Obrázek 3. Mapa paměti mikrokontroleru [4]

Hlavní problém představuje paměť RAM. Systém bude muset zajistit aby její omezená velikost postačovala pro vlastní operační systém i pro všechny spuštěné úlohy. Ideální stav zajišťující oddělenou paměť pro všechny běžící úlohy je zřejmě nereálný, režie kolem takovéhoto systému by byla příliš náročná a navíc by bylo třeba zajistit možnost programování za běhu aplikace, aby se dal

obsah paměti odsouvat do programové části flash z důvodu virtuální maximální velikosti paměti. Proto budou muset být samotné úlohy navrženy tak aby každá měla od počátku přidělenou část paměti, kterou bude využívat a bude se spoléhat na korektnost zápisů, protože neexistuje HW ochrana paměti. Zároveň bude třeba zajistit konzistenci dat na zásobníku. Při běhu aplikací si tyto ukládají data na zásobník a v případě přepnutí kontextu může dojít k situaci, že na zásobníku zůstanou data z předchozí úlohy a ta budou omylem načtena v právě spuštěné aplikaci místo dat právě určených na zpracování. A to i v případě že by všechny úlohy napsány jako subrutiny, že nebudou předávat data z jednoho běhu do druhého přes zásobník. Protože v rámci některých plánovacích mechanizmů může dojít k přepnutí kontextu v průběhu vykonávání dané úlohy, možným řešením je neukládat data na zásobník, ale toho se v programu docílí obtížně. Proto zbývají další dvě možnosti. První snazší je vytvořit v paměti pro každou úlohu vlastní zásobník v jí přidělené části paměti. Tím se ale radikálně zvětší nároky na paměť a je potřeba formálně dokázat že přidělené části paměti budou dostatečné. Druhou výpočetně náročnější možností je odkládat při přepínání úloh část zásobníku využitou danou aplikací do paměti vyhrazené pro tyto účely v části pro jádro.

2.2 Analogově digitální převodník

Slouží k připojení zařízení která nemají digitální výstup. Nebude pravděpodobně sloužit k měření, protože 256 možných hodnot je pro složité programy a měření nedostatečný počet. Pro měření by bylo vhodné připojit externí A/D převodník. Ale poslouží výborně právě pro příjem signálu z analogových snímačů a senzorů.



Obrázek 4. Analogově digitální převodník [4]

Jak je z obrázku patrno, analogově digitální převodník používá systém přerušení pro signalizaci úspěšného dokončení převodu. Pokud budeme chtít použít plánovací mechanizmus, tak se budeme muset vyrovnat právě s tímto faktem. Při nějakém libovolném plánu bude docházet k narušování toho plánu právě přerušením. Možných řešení je více. Buď použít plánovací mechanizmus, který se zvládne vyrovnat s aperiodickými úlohami (podrobněji problém dále v textu v samostatné kapitole věnované plánovacím mechanizmům), nebo zahrnout výskyt přerušení jako jednu z plánovaných úloh případně nahradit přerušení. Na přerušovací rutinu pro ADC umístit pouze kód nastavující flag a uchovávající hodnotu v paměti a obsluhu úlohy spustit až na základě nastaveného flagu v době kdy má podle plánu nárok běžet. Což ale předpokládá že budeme schopni veškerá data vtěsnat do paměti, případně ADC jednotku využívá jen jedna běžící úloha.

2.3 Přerušovací podsystém

Je tvořen poměrně jednoduchou logikou, kdy na předem daném přerušovacím vektoru je uložena adresa obslužné rutiny, která je vyvolána v okamžiku kdy nastane přerušení. Přerušení představuje problém zejména z hlediska narušení časového plánu úloh. Dále také může nastat problém, kdy vyvolání obslužné rutiny zničí data v paměti. Při vyvolání přerušení dojde k uložení stavu registrů na

zásobník a pokud dochází paměť může zásobník přepsat data jiné běžící aplikaci. Většinu přerušení lze zakázat vymaskováním. Přehled veškerých přerušení ukazuje obrázek 5.

Vector Priority	Vector	Address	Vector
Lowest ↑	IF15	$FFDE	ADC conversion complete vector (high)
		$FFDF	ADC conversion complete vector (low)
	IF14	$FFE0	Keyboard vector (high)
		$FFE1	Keyboard vector (low)
	IF13 ↓ IF6	—	Not used
	IF5	$FFF2	TIM overflow vector (high)
		$FFF3	TIM overflow vector (low)
	IF4	$FFF4	TIM Channel 1 vector (high)
		$FFF5	TIM Channel 1 vector (low)
	IF3	$FFF6	TIM Channel 0 vector (high)
		$FFF7	TIM Channel 0 vector (low)
	IF2	—	Not used
	IF1	$FFFA	$\overline{IRQ}$ vector (high)
		$FFFB	$\overline{IRQ}$ vector (low)
	—	$FFFC	SWI vector (high)
		$FFFD	SWI vector (low)
	—	$FFFE	Reset vector (high)
↓ Highest		$FFFF	Reset vector (low)

Obrázek 5. Tabulka přerušovacích vektorů [4]

Nevyužité přerušovací vektory slouží pro zařízení, která nejsou v tomto čipu NITRON implementována. Zjistit vhodnou velikost paměti nacházející se mezi blokem uživatelských dat a vrcholem zásobníku, kterou bude nutno udržovat prázdnou, bude závislé na běžících úlohách. Přesto si myslím, že technika, kdy umístíme na každý vektor přerušení obslužnou rutinu s následujícím kódem, viz. obrázek 6.

```
LDA 80
ORA #1
STA 80
RTI
```

Obrázek 6. Jednoduchá obsluha přerušení

Při nutnosti plánování úloh za běhu systému je žádoucí, aby jádro a plánovací mechanizmus běžel co nejméně a dával prostor funkčním úlohám, které vykonávají úlohy o které zákazník stojí a nezabíral procesor jen pro sebe. Při použití zástupné subrutiny viz. obrázek 5, dojde k výpadku na 4 takty procesoru což je nepatrné a vždy je možné navrhnout plán který s tímto bude počítat. Pokud by došlo k velkému množství přerušení. Tak velkému že by došlo k výpadku celého systému. Došlo k

němu už v okamžiku, když nastalo takové množství požadavků. A použitý plánovací mechanizmus je špatný, případně je takové množství úloh nezpracovatelné na dané platformě.

Další možností je spouštět plánovací mechanizmus přímo při každém vyvolání přerušení, aby došlo k znovu přepracování a přepočítání plánu úloh, tak aby byl dodržen časový plán. Tento přístup má výhodu v ušetření příznakového bajtu v paměti a schopnosti plně přepracovat právě v té době kdy nastane požadavek a zvýhodnit tak některé úlohy s vyšší prioritou. Ale přepočet plánu je poměrně náročná činnost a bude tím podstatně větší část času procesoru zabraná jádrem na tyto režijní přepočty. V první variantě dochází k přepočtu plánu až na konci jedné periody, znatelně to sníží objem času spotřebovaného na přepočet plánu, znamená to ale problém v okamžiku, kdy existují úlohy u kterých může dojít k překročení doby výpočtu pro danou úlohu, dříve než je čas periody.

3 Real-time OS

Každý operační systém je podle mnohých odborníků real-time, protože všechno pracuje v reálném čase. Dělení vychází z úrovně jistoty a důvěry v daný operační systém že neselže a zaručí včasné obsloužení všech zpracovávaných úloh. Real-time operační systémy většinou označujeme takzvané hard real-time operační systémy. Tyto vycházejí z předpokladu, že je potřeba mít operační systémy schopné pracovat v náročných aplikacích. Řízení jaderných elektráren, řízení odpalování balistických raket, řízení brzd v autě a spoustě dalších. Jde o takzvané hard real-time operační systémy narozdíl od soft real-time operačních systémů, kde při jejich selhání nedochází k vážným ztrátám ani ohrožení životů.

Dále ještě rozlišujeme takzvané firm real-time operační systémy. Je to kombinace obou předchozích. Firm označujeme systémy u nichž havárie některé úlohy jednou za delší čas vede k degradaci výkonnosti systému, ale následky nejsou katastrofální. Ty nastanou až pokud procento selhání úloh přesáhne mez přijatelnou pro uživatele.

Zatímco na takové vojenské základně se o řízení stará spousta výkonných serverových stanic, v osobním autě bude použito malých vestavěných zařízení. Pole působnosti je velice široké. A zatímco velké systémy sáhnou po produktech renomovaných firem jako operační systém WindowsCE, automobilový průmysl má svůj systém OSEK, na spoustu zařízení se vhodný operační systém hledá. Zatím dochází k úpravám stávajících systémů jako RTlinux a podobně. Tyto úpravy ale nedosahují příliš dobrých výsledků. Pořád obsahují neduhy svých vzorů, protože jsou "jen" doplněny o plánovací mechanizmy.

4 Druhy úloh

Plánovací mechanizmy dělíme zaprvé podle času výskytu požadavků na obsluhu úloh které jsou v systému na synchronní a asynchronní. A dále podle periodicity jejich vyvolávání.

4.1 Synchronní úlohy

Synchronní úlohy jsou takové, u kterých známe přesně místo v programu po kterém dojde k znovuvyvolání dané úlohy během výpočtu. Nemusí jít nutně o periodické úlohy. Synchronní události nastávají kupříkladu v závislosti na jiných právě prováděných akcích. Pokud právě běžící úloha otevře kohout přívodu vody do kotle, tak je jisté že po ní přijde úloha měřící množství vody v kotli, aby mohla přívod včas uzavřít. Nejčastěji jsou reprezentovány jako samostatný podprogram.

4.2 Asynchronní úlohy

Asynchronní úlohy jsou naopak neočekávatelné. A mohou je tvořit i periodické úlohy. Zpravidla jsou vyvolány nějakou vnější událostí a jsou implementovány jako obsluha přerušení. Pokud použijeme opět příklad s napouštěním vody do kotle, tak pokud nebude existovat úloha, která by soustavně sledovala hladinu vody v kotli, ale do kotle umístíme snímač a úlohu zavírající kotel vyvoláme až na základě vnějšího signálu od snímače, bude se jednat o asynchronní úlohu. Nemá přímou návaznost na prováděnou úlohu, ale reaguje na vnější událost.

4.3 Periodické úlohy

Dalším dělením úloh dostáváme úlohy periodické, jde o úlohy, které budou volány neustále opakovaně po předem známém časovém kvantu. Může se jednat třeba právě o úlohy jádra systému. Toto bude pravidelně voláno po uplynutí časového kvanta, aby naplánovalo úlohy do dalšího časového kvanta, aktualizace hodin a podobně.

4.4 Aperiodické úlohy

Jde o úlohy které jsou vyvolávány zcela náhodně ve vztahu k plynoucímu času. Jde převážně o přerušení a podobně. Pokud je aperiodická úloha volána velice zřídka, označujeme ji jako sporadickou úlohu.

5 Dělení plánovacích mechanizmů

Protože plánovací mechanizmy jsou velice závislé na možnostech a použité platformě, existuje i množství technik plánování, které se ani neoznačují jako real-time operační systémy. Jsou hojně využívány právě na méně výkonných platformách jako je HC08, protože jsou přehledné a nenáročné. Označujeme je jako pseudojádra RTOS.

5.1 Pseudo jádra

Pseudojádra představují variantu pro mnoho systémů. Zajišťují víceúlohovost systému za minimální cenu. Úlohy jsou neparametrizované subrutiny volané podle stanoveného pevného plánu. Plánování se provádí v podstatě při programování a následně se nemění.

5.1.1 Vyzývací smyčka

Tento postup je založený na nekonečné smyčce a stavovém registru. Ve smyčce je neustále v cyklu testováno, zda nějaký stavový registr je nastaven a informuje tím, že se má vykonat daná obslužná rutina. Pokud se narazí na nastavený registr, dojde k zavolání subrutiny a vynulování stavového registru. Po ukončení zavolané subrutiny se pokračuje v prohledávání dalších stavových registrů, zda nebyl zatím další z nich nastaven. Jde o jednoduchý dobře analyzovatelný přístup. Avšak náročný na CPU a náchylný k ignorování nastalých událostí v důsledku vykonávání dlouhého kódu jiné úlohy.

Mírně upravenou variantou je synchronizovaná vyzývací smyčka, používá se v případech, kdy je reakce na nějakou událost podmíněna ověřením, zda opravdu došlo k události a nejedná se jen o zákmity na signalizačním vodiči. Rozšíří testování na dva testy následující po sobě s vloženým vhodným pozastavením, aby se mohlo ustálit napětí na signalizačním vodiči.

5.1.2 Cyklické provádění

Jde opět o spuštění hlavní smyčky, která nyní ale volá postupně části všech úloh. Pro správnou funkci je třeba rozdělit jednotlivé úlohy na menší celky trvající stejně dlouhou dobu. Následně je spouštět střídavě a tím je zajištěn současný běh všech aplikací. Zvýhodňování a prioritu lze řešit pomocí tabulky reprezentující pořadí volaných úloh. A následným přesouváním a upravováním položek v této tabulce lze zvýhodňovat vybrané úlohy. Hlavní problém tohoto přístupu je v nutnosti rozčlenit úlohy na malé stejně dlouhé části. Toto bývá ve spoustě případů nerealizovatelné.

5.1.3 Stavově řízený kód

Jde o případ, kdy je celá aplikace spuštěna jako jedna velká úloha a její jednotlivé části odpovídají jednotlivým původním úlohám. Přesto, že se jedná ve výsledku o složitý kód, protože se jedná o stavový automat, můžeme ho optimalizovat a formálně verifikovat, což jsou největší výhody tohoto přístupu. Navíc můžeme využít přítomnosti stejného stavu registrů a prostředí a není nutné složitě předávat parametry.

5.1.4 Spolupracující úlohy

Jde o techniku, kdy je plánování čistě na programátorovi jednotlivých úloh. A úlohy se musí dobrovolně vzdávat řízení ve prospěch jiných úloh a předávat ho nějaké centrální plánovací úloze, která se postará o zavolání další úlohy podle plánu.

5.2 Využití přerušovacího podsystému

Přerušení zde slouží k volání jednotlivých úloh a to buď HW přístupem, kdy je úloha obslužnou rutinou na nějakou vnější událost nebo událost v komponentě systému která používá k informování o své činnosti přerušovací systém. V případě HC08 může jít třeba o ADC. Nebo jde o úlohy volané pomocí přerušení od časovače, kdy tím zajišťujeme periodické volání vybrané úlohy. Podle mého ale přerušení v tomto druhu úloh představuje velké riziko při přerušení kódu, který tím povede na selhání systému. Varianta, kdy běží úloha která posouvá kotoučem pily v libovolném směru je přerušena úlohou snažící se aktualizovat údaje na displeji, může vést k ublížení na zdraví, kdy se pila nezastaví dokud se řízení nepředá zpět první úloze.

5.3 Systémy pracující v popředí a pozadí

Využití procesoru lze zvýšit použitím systému pracujících v popředí a pozadí. Tento princip využívá faktu, že existuje nějaká složitá úloha s malou prioritou. Tato úloha je vykonávána neustále v hlavní smyčce, takzvaně na pozadí a při výskytu události vedoucí na spuštění jiné úlohy je této předán procesor a běží pak na popředí. Cílem je využít nečinnosti CPU během čekání na potřebu zavolání další úlohy nějakou užitečnou činností. Tyto principy lze již s úspěchem rozšířit na plné real-time operační systémy. Kde jádro je ustaveno jako proces s nejvyšší prioritou a v průběhu své činnosti je hojně využíváno možnosti zakázat většinu přerušení a provádět tak kritickou sekci kódu. Samozřejmě je tento přístup zatížen možností ztráty informace o nastalém přerušení v případě jeho úplného zakázání.

5.4 Model TCB

Tento postup je využívaný i v systémech běžně nasazovaných na osobní počítače a servery. Jde o přístup, kdy je každé úloze přiřazen blok TCB, který obsahuje důležité informace o úloze. Hlavně obsahuje údaje o jejím stavu, zda právě běží, nebo čeká na spuštění, nebo je uspaná a podobně. Informace o její prioritě, velikosti periody a dalších věcech potřebných pro plánování. Takovýto přístup je používán zejména díky jeho možnosti přidávat a odebírat úlohy a možnostem pokročilé správy.

5.5 Plánovací mechanizmy

5.5.1 RM

Mechanizmus je založen na principu přidělování procesoru úlohám v závislosti na frekvenci jejich volání. Čím častěji je úloha volána, tím pravděpodobněji dostane čas CPU. Jde o statický plánovací mechanizmus pro periodické úlohy. Plán je sestaven předem.

5.5.2 ID/DM

Tyto mechanizmy opět slouží pro plánování statických periodických úloh. Jeho principem je přiřazení nejnižší priority úloze s nejkratším časovým limitem. Cílem je dosáhnout možnost plánovat i úlohy u kterých není velikost periody rovna časovému limitu, který je nutno dodržet. Časovým limitem se rozumí doba od vystavení požadavku na spuštění úlohy po okamžik, kdy svou nečinností úloha způsobí chybu systému.

5.5.3 EDF

Algoritmus EDF přiřazuje prioritu úlohám dynamicky. Vzhledem k tomu je možné plánovat pomocí mechanizmu EDF i aperiodické úlohy. Jeho princip přiřazování procesoru je v závislosti na zbývajícím čase do uplynutí časové kritické meze dané úlohy od aktuálního časového okamžiku.

5.5.4 LLF

Tento algoritmus vychází z myšlenky přidělovat procesor na základě nejmenší doby volnosti. Což odpovídá úloze, která může být nejdéle zbavena procesoru s tím, že pořád bude dodržena její kritická časová mez. Algoritmus je vhodný i pro aperiodické úlohy.

5.5.5 Hybridní plánovací postupy

V případě, že je potřeba naplánovat periodická a aperiodické úlohy ve vzájemné kombinaci, používáme buď některý z výše uvedených přístupů, kdy ho vhodně rozšíříme o práci v popředí a pozadí. Takto upravené mechanizmy jsou schopné plánovat množiny úloh s hard periodickými úlohami a soft aperiodickými úlohami. Periodické úlohy jsou plánovány v popředí a aperiodické v pozadí. Nutnou podmínkou je aby aperiodické úlohy byly soft, tedy neměli kritickou mez vedoucí k selhání systému.

Další možností je rozšířit stávající úlohy o server. Server představuje periodickou úlohu, která nevykonává žádnou užitečnou činnost. Jeho jediná funkce je v alokaci procesoru pro aperiodické úlohy. Ty se vykonávají v čase běhu serveru.

5.5.5.1 Vyzývací server

Tento server představuje variantu, kdy na počátku svého spuštění otestuje, zda existuje požadavek na spuštění aperiodické úlohy. Pokud ano, je tato úloha vykonána v době běhu serveru. Pokud ale žádný požadavek neexistuje, server se ukončí a jeho čas je věnován pro plánování periodických úloh. Hlavní výhodou je jednoduchost. Nevýhodou pak dlouhé odstavení aperiodických úloh, kdy v nejhorším případě úloha musí čekat celou periodu spouštění serveru. Server má obvykle nejvyšší prioritu.

5.5.5.2 Odkládací server

Jde o mírné vylepšení předchozí varianty, kdy není server spouštěn hned jak je mu přidělen čas CPU, ale vyčkává se spuštěním až do okamžiku, kdy nastal aperiodický požadavek. Ve skutečnosti pak, pokud server má nejvyšší prioritu, dochází k okamžitému spouštění aperiodických úloh na úkor periodických. Toto vede na snížení využitelnosti procesoru.

5.5.5.3 Sporadický server

Jde o mechanizmus, kdy aperiodické úlohy mají díky prioritě serveru také nejvyšší prioritu, protože server si ponechává čas na CPU dokud nevyvstane aperiodický požadavek, ale svou roli zde hraje i kapacita serveru. Aperiodický požadavek tuto kapacitu snižuje a plná kapacita serveru se obnoví až s počátkem nové periody. Tím je zajištěno že aperiodické úlohy nepůsobí na úkor periodických úloh a nesnižuje se tím využití CPU.

5.5.5.4 Slack steeling

Jde o postup kdy je periodickým úlohám kraden čas CPU v závislosti na zbývajícím času volnosti do vypršení jejich kritických mezí. A to tak, aby tyto meze byly nadále splnitelné a přitom došlo k co nejrychlejšímu obsloužení aperiodických úloh.

5.5.5.5 Další možnosti

Další možností je použít pro plánování mechanizmus, který zvládá plánovat jak periodické tak aperiodické úlohy. Kupříkladu výše zmíněný mechanizmus EDF.

Poté existuje potřeba plánování i aperiodických hard úloh. K čemuž nám právě poslouží tyto plánovací mechanizmy zvládající plánovat oboje. Případně je potřeba dodat dodatečnou logiku, která zajistí plánovatelnost úloh v závislosti na jejich kritických časových mezích.

5.5.6 Závislé úlohy na pořadí

Plánování závislých úloh znamená vyrovnání se s faktem, že úloha B může být spuštěna až po úloze A. Řada úloh má potřebu na sebe navazovat. Zpracování dat bývá postupné. A jen čistým plánováním bez braní v úvahu tohoto faktu můžeme nutit úlohu běžet dříve než na připravená data od jiné úlohy.

K plánování použijeme klasické plánovací mechanizmy, RM, DM, EDF …. A potřebu, aby úloha mohla být spuštěna až bude dokončen její předchůdce a nemohla být přerušena svým následníkem zahrneme do těchto mechanizmů vhodnou úpravou velikosti priorit, periody a kritického času na dokončení úlohy.

5.5.7 Závislé úlohy na sdílených prostředcích

Dalším problémem jsou situace, kdy dvě a více úloh potřebují přistupovat k jednomu sdílenému prostředku. Většinou se bude jednat o paměťový prostor společně zpracovávaných dat. Situace se musí řešit vstupem úlohy do kritické sekce. Je potřeba zajistit koherenci dat a není možné připustit, aby dvě úlohy měnily současně stejná data. A právě vstup do kritické sekce, kde se zdroje zamknou pomocí semaforů nebo jiné techniky, případně je kritickou sekcí celý výpočet úlohy až do dokončení práce se sdíleným prostředkem, kde je kritickou sekcí zabrán procesor. Takovéto případy mohou vést k uváznutí.

5.5.7.1 Protokol dědění priorit

Tato technika spočívá v tom, že úloha která je právě v kritické sekci, jakoby zdědí prioritu úlohy která by ji jinak přerušila a pokračuje v provádění kritické sekce dokud ji sama neopustí. To vede na poměrně rychlé dokončení prací v kritické sekci. Nezabraňuje to však uváznutí.

5.5.7.2 Protokol stropování priorit

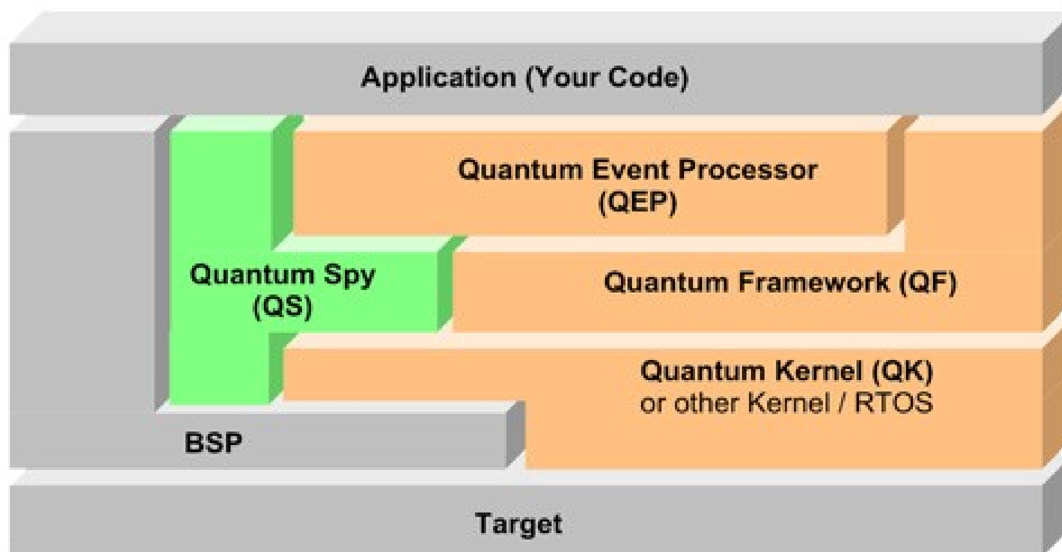
Technika je založená na stanovení stropní priority určené ze všech úloh které mají během výpočtu potřebu někdy zasáhnout do sdíleného prostředku. A úloha zdědí prioritu pouze pokud je priorita prostředku vyšší než stropové priority všech ostatních právě přidělených prostředků. Tento protokol již nevede na uváznutí.

6 Využití stávajících systémů

Existují komerční systémy. Ale nepodařilo se mi najít žádný kompatibilní s platformou HC08. Jediný který sem objevil je systém QP, realizovaný jedním člověkem jako open source. V dokumentaci je uvedeno, že by měl být použitelný i pro platformu HC08, ale nebyl pro ni testován. Jinak zbývá buď možnost vytvoření nového systému ať už přímo nebo pomocí vhodného nástroje. Nebo se pokusit modifikovat nějaký jiný stávající systém. Nadějně v tomto směru vypadá projekt freeRTOS.

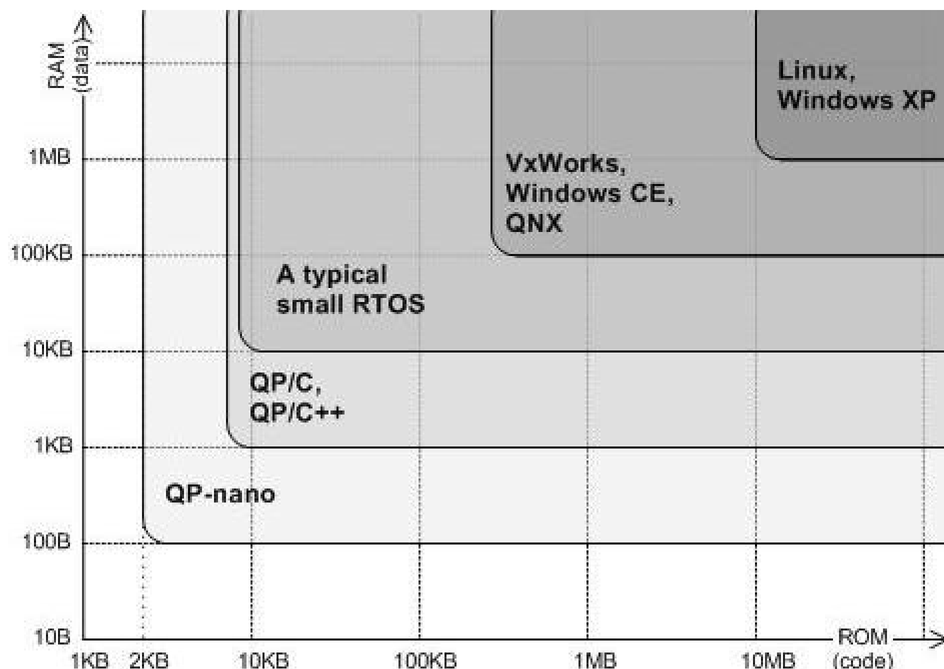
6.1 Quantum Platform

Jde o systém [4] založený na stavovém stroji na bázi UML. Programovaný v C a C++, je členěn na několik na sebe navazujících vrstev, které poskytují postupně lepší programové prostředky pro psaní aplikačních úloh. Jednotlivé funkční části jsou odděleny do bloků pro správu, Quantum-Spy, jádro samotného systému Quantum-Kernel, prostředí zajišťující asynchronní požadavky aplikací a požadavky na přímý přístup Quantum-Framework a nakonec konečný stavový automat starající se podporu vyšších úloh a spravující řízení na úrovni UML. Schéma návaznosti ukazuje obrázek 7.



Obrázek 7. Návaznost jednotlivých bloků systému QP [2]

Ve standardní verzi se všemi prostředky je ale stále velký. Pro použití na platformě HC08 však existuje jeho verze nano. Obsahující jen samotné plánovací jádro starající se o správný běh aplikačních úloh. Srovnání náročnosti představuje následující obrázek.



Obrázek 8. Srovnání náročnosti real-time operačních systémů [2]

QP-nano náročností spadá do oblasti pod 100B paměti RAM a 2KB programového kódu, což umožňuje jeho nasazení na platformě HC08.

6.2 FreeRTOS

Jde o systém [4] pracující na mnoha platformách, existuje i jeho verze pro HC12, 16-bitový mikrokontoroler podobného typu od stejné firmy. Bohužel pro HC08 by bylo třeba jeho kód upravit a předělat. Spíše než na tento typ, bych považoval za vhodnějšího kandidáta přetvoření kódu pro Zilog Z80 nebo Cygnal 8051, což jsou taktéž 8-bitové mikrokontrolery. Sice mají odlišnou instrukční sadu a periferie, ale přepsání jejich kódu představuje hlavně překlad. Přizpůsobení kódu z HC12 znamená i přepracování návrhu na architekturu s poloviční bitovou šířkou.

Pro plánování využívá buď preemptivní plánování stylem round robin. Nebo kooperativní přístup, kdy každá úloha se vzdává v čas řízení pomocí volání jádra taskYIELD(). Systém zkonzumuje v současné verzi bez úprav 4KB paměti programu a poměrně hodně paměti RAM, ale toto je dáno hlavně tím že údaje jsou pro 32 bitovou verzi.

Položka	**Spotřeba paměti**
Plánovač	236 B (Při použití menších datových typů spotřeba klesne).
Každá nová fronta	76 B + odkládací oblast.
Každá nová úloha	64 B (4 znaky pro jméno) + velikost zásobníku.

Tabulka 1. Náročnost systému na paměť.

6.3 Nový systém

Řešením je také implementace zcela nového systému. Můžeme využít nástroje na plánování a již existující normy POSIX nebo jiné a implementovat nové jádro pracující na zadaném mikrokontroleru. Při dobrém plánu implementace pseudojader pro synchronní cyklické úlohy je přímé napsaní kódu hlavní smyčky zřejmě stejně nejvýhodnější varianta.

7 Realizace

7.1 Příprava

7.1.1 Vývojové prostředí

Protože na mikrokontroléry řady HC08 je vyvinuto prostředí CodeWarrior od firmy Metrowerks, poskytované i v několika formách zdarma, je téměř nemožné vyhnout se použití tohoto nástroje. Proto i já ho využiji. Starší odkazy vedoucí na stránky společnosti Metrowerks jsou nyní již přesměrovány na stránky současného výrobce pro CodeWarrior, a to společnost FreeScale. Odtud tedy získáme verzi prostředí CodeWarrior pro mikrokontroléry. Pokud budeme získávat prostředí z hlavní anglické stránky FreeScale, bude nutné se zaregistrovat, vyplnit několik formulářů a po potvrzení volby na stažení, bude vaše žádost do několika dní prověřena administrátorem a o rozhodnutí budete informováni na email uvedený při registraci. Já jsem ale spěchal, tak jsem použil čínskou jazykovou variantu, místo anglické verze stránek společnosti FreeScale. Z politických důvodů je na ní v době psaní tohoto textu dostupná verze CodeWarrioru 6.0 k přímému stažení. Stačí pouze zorientovat se v čínském písmu, k čemuž výborně poslouží jména stránek kam vedou jednotlivé odkazy, které zůstávají v angličtině. Bohužel nelze použít model z anglické verze stránek, neboť čínská jazyková mutace má jinou mapu stránek. Bohužel tato verze CodeWarrioru vyžaduje trochu silnější PC a v laboratořích na fakultě informatiky je k dispozici starší verze CodeWarriora 3.1, proto by bylo vhodné použít právě tuto verzi kvůli kompatilibitě. Akorát původní stránky společnosti Metrowerks již neexistují a vedou na nynější stránky spolčnosti FreeScale a již je podporována pouze nová verze. Takže verze 3.1 je dostupná pouze na školních počítačích případně ke stažení na starých torrentech. Pokud se ji podaří získat je ještě potřeba opatřit si licenci. Ikdyž je tato pro omezené použití dostupná zdarma, tak z již výše popsaných důvodů, nejsou dostupné původní webové stránky společnosti Metrowerks a na stránkách společnosti FreeScale, již není dostupná. Takže je třeba ji najít na internetu, kde by mohla být ještě uložena na soukromých webových stránkách.

7.1.2 Vhodný typ microcontroléru

Jako typ microcontroléru použitého na testování by bylo vhodné zvolit typ HC08LJ12, protože je ve škole dostupný ve vývojovém kitu Janus. Janus je vybaven pro přímé propojení s PC a umožňuje naprogramování i sledování z vývojového prostředí CodeWarrior. Stejně jako připojení všech komponent použitelných na odsimulování později zvolené sady úloh na prověření jednotlivých možností různých jader RTOS. Tento typ je vybaven pro universální použití, má 512b paměti, proto

nebude zřejmě možné spouštět moc úloh a budou muset respektovat vzájemné využívání paměti. Což na jednu stranu umožní napsat program čistě pomocí C a nezatěžovat se managmentem paměti, protože překladač najde pro proměnné místo sám. Na druhou stranu se připravíme o možnost efektivněji využívat paměť a spouštět náročnější úlohy.

7.1.3 QP

Systém QP je jak už bylo zmíněno výše rozdělen na několik vrstev a v rámci použití na HC08 je doporučován samotný základ systému, a to QP nano. Kód je napsán pomocí C a C++, do objektové podoby. To umožňuje do velké míry nasazení na nejrůznějších platformách a přizpůsobení je jen otázkou předefinování názvů použitých řídících registrů v souboru k tomu určeném. Jinak je QP plně připraven na použití na platformě HC08. A nadále stačí pouze upravit nebo nahradit aplikaci, dodávanou spolu s QP nano jako ukázku, za aplikaci určenou na dané použití, kde pro testování jsem vymyslel smyšlenou aplikaci řízení zavlažování v okolí rodinného domku. Bohužel systém se mi nepodařilo zprovoznit na starší verzi CodeWarrioru 3.1. Zpětná kompatibilita je výrobcem zajištěna, dopředná tu ale selhává. QP nano byl vytvořen v novější verzi CodeWarrioru a má také trochu rozdílné požadavky, zajištěné licencí pro novější verze a využívá makra a systémové proměnné, dostupné až v této novější verzi. Z toho důvodu jsem musel přejít na verzi CodeWarrioru 6.0, tato verze již náročností vyžaduje novější PC a OS z řady windows NT, změnily se tedy i možnosti připojení periferií a programátoru z důvodů nepřímého přístupu k portům u systémů na bázi windows NT. Lepší situace ale bude u získávání licence, která již je ke stažení volně dostupná po registraci na FreeScale.

7.1.4 Upravení FreeRTOS

V rámci práce jsem zvolil existující sytém FreeRTOS. FreeRTOS není přímo portován na HC08, je napsán v programovacím jazyce C a přizpůsoben k přenesení na různé platformy. Většina kódu je neměnná, pouze je potřeba vhodně upravit soubory zajišťující nastavení a ovládání timeru a realizaci složitějších operací. Tento kód je v oddělených souborech pro lepší přehlednost. Problém je hlavně v používání timeru. Jádro je potřeba v pravidelných intervalech spouštět a zajistit mu tak možnost řízení běhu úloh. K tomu právě slouží timer a jeho přerušení.

V souborech určených pro nakonfigurování timeru je nastavena vhodná doba periody po jejímž uplynutí je přerušeno vykonávání úloh a řízení je předáno jádru, kde dojde k přepnutí kontextu na úlohu podle plánu. Při některých aplikacích je ale potřeba využít timer pro jednu z úloh a není proto možné poskytnout ho jádru. Tuto situaci můžeme vyřešit použitím simulátoru timeru, jde o to, že použijeme další soubor určený pro správu timeru, kde popíšeme vztah mezi hardwarovým timerem a časováním potřebným pro jednotlivé úlohy. HW timer pak spouští přerušení které nesměřuje k jádru, ale do funkce zajištující onu simulaci, kde inkrementuje vnitřní čítač simulátoru timeru a vyhodnotí,

pro kterou z úloh má nastat přerušení a to následně vyvolá softwarově. Je nutno si uvědomit, že pak i jádro je jednou z úloh obsluhovanou tímto simulátorem. Toto řešení má výhodu v tom že pomocí jednoho timeru zvládneme spouštět více úloh. Nevýhoda spočívá v častějším spouštění kódu simulátoru timeru, čímž zbytečně více zatěžujeme procesor.

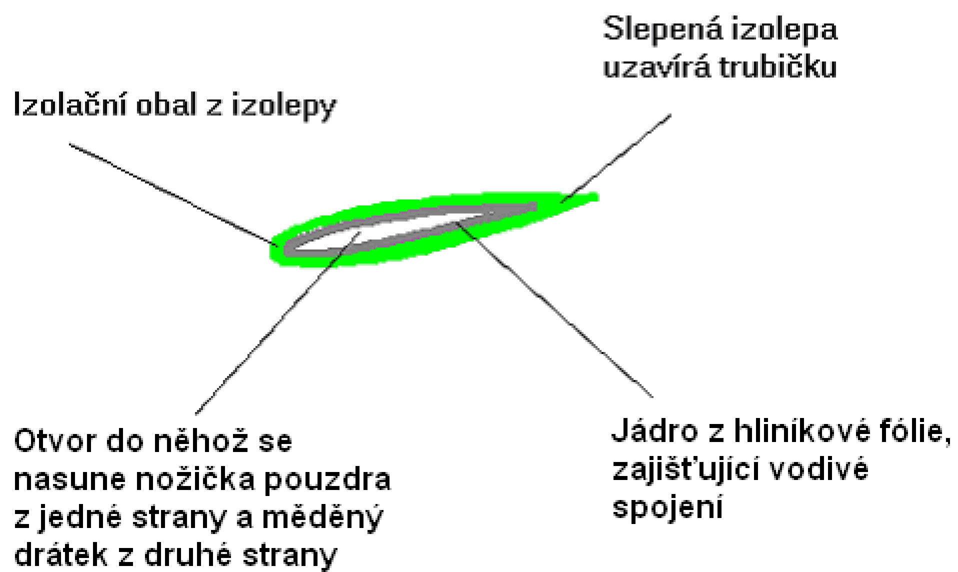
7.1.5 Změna mikrokontroléru

Další problém nastal v otázce paměti. Protože systém FreeRTOS byl původně navržen pro 32 a 16 bitové procesory. Přestože byl kód upraven na HC08, tedy 8 bitový procesor, zůstaly celkem vysoké nároky na paměť. Proto bylo třeba změnit vybraný typ HC08. Z důvodů potřeby až 4 KB jsem zvolil typ HCS08GB60. Protože přecházím na HCS, tak nastal malý problém s přepsáním QP dále z HC08LJ12 na HCS08GB60. Vetší problém nastal ale ohledně možnosti otestování. Od společnosti FreeScale by objednán a získán vzorek k otestování, bohužel již není možné využít kitu Janus ze školních laboratoří, protože mikrokontroléry řady HCS používají jiný programátor, navíc zvolený typ je vyráběn pouze v pouzdře LQFP. Takže nastal problém ohledně připojení vývodů. Jednou z možností je využití testovací patice, bohužel pro tento standard ještě není v ČR žádná dostupná a bylo by třeba ji vytvořit. Další možností je připájení vývodů, což by bylo zřejmě nejvhodnější, ale z důvodů mojí neschopnosti, těžko proveditelné. Kontaktní nožičky u pouzdra jsou s průměrem 0.2 mm velice náchylné na ulomení, zvlášť při zahřátí na teploty potřebné na vytvoření vodivého spoje, bez nežádoucích kapacitních a jiných nežádoucích vlastností.

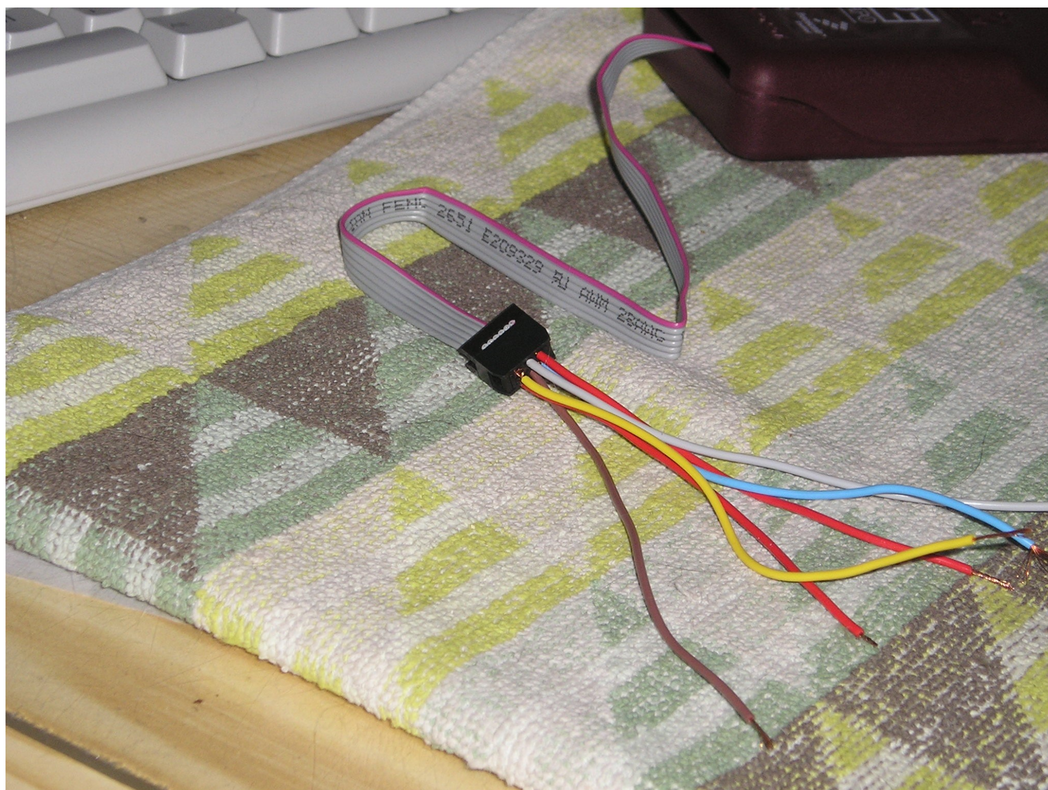
Obrázek 8. Kontaktní trubička.

Moje varianta připojení tedy spočívá v připojení pomocí kontaktních trubiček. Kontaktní trubička je vyrobena z hliníkové fólie, izolační pásky a kousku měděného drátku, jako je na obrázku 8. Hliníková fólie je přilepena na izolační pásku, čímž získám jednostranně vodivý materiál. Přehnutím vodivou stranou k sobě a utěsněním zbylých konců izolační pásky vznikne trubička s vodivou částí uvnitř a izolovaná zvenčí. Z ilustračního obrázku 9, je princip patrnější. Na jedné straně ji stabilně opatřím měděným drátkem aby ji bylo možné zapojit do nepájivého pole a na druhém konci vzniklou trubičku navléknu na jeden vývod pouzdra mikrokontroléru. Tím propojím mikrokontrolér s nepájivým polem a mohu ho dále připojit k programátoru, zdroji napájení a případným dalším periferiím.

Další věcí co bude potřeba, bude programátor, bohužel není k dispozici žádná přípojka, bylo nutné připojit programátor pomocí samostatných vodičů do nepájivého pole. Situaci ukazuje obrázek 10 a 11 s celkovým zapojením. Nahoře je programátor dále přes USB připojený k PC a dva vodiče od napájení obvodu. Dole je pak nepájivé políčko, v němž jsou připojeny pomocí drátků kontakty z programátoru s kontaktními trubičkami vedoucími dále k samotnému mikrokontroléru umístěnému v pravém dolním rohu nepájivého pole.

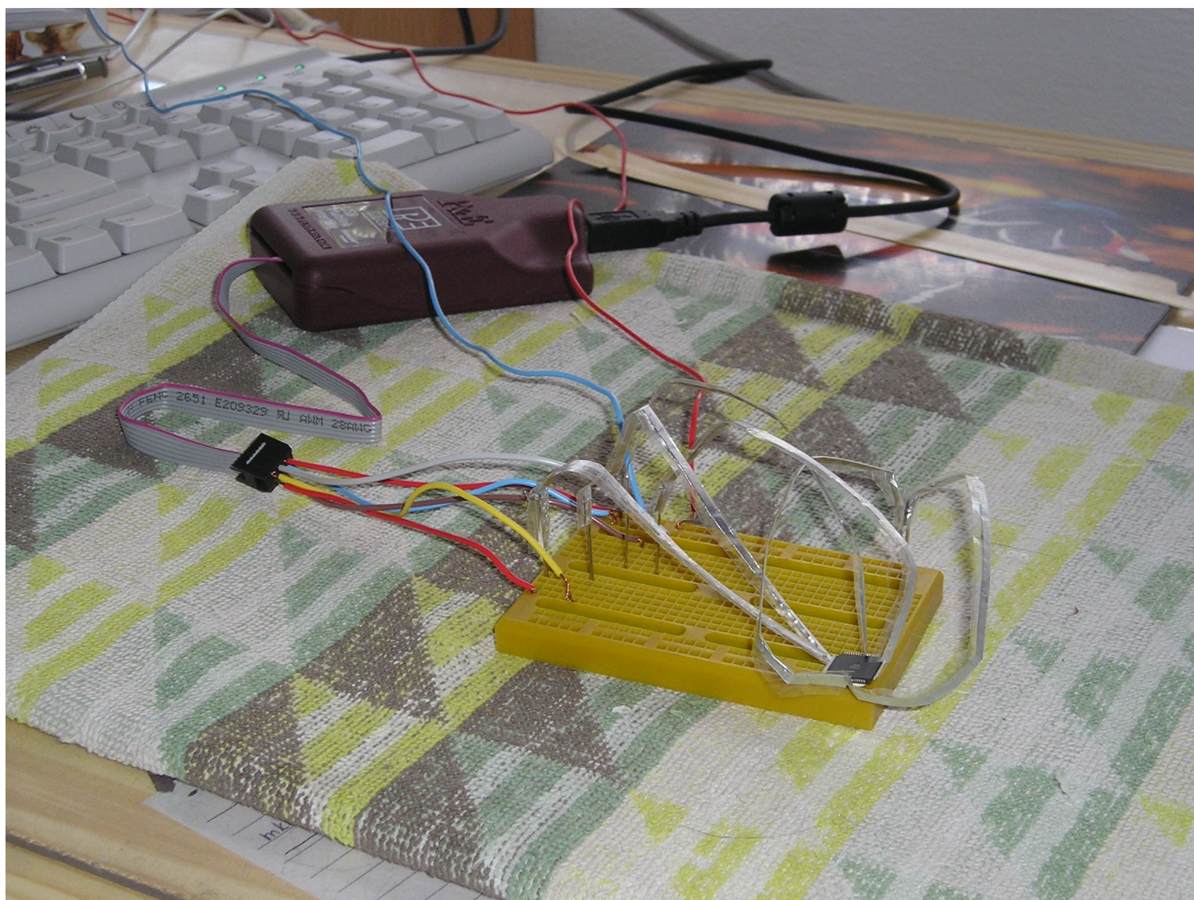


Obrázek 9. Propojovací trubička



Obrázek 10. Napojení programátoru

Takže další věc co bude potřeba bude nepájivé políčko, BDS programátor, zdroj stabilizovaného napětí pro napájení obvodu. Výsledné zapojení ukazuje následující obrázek.

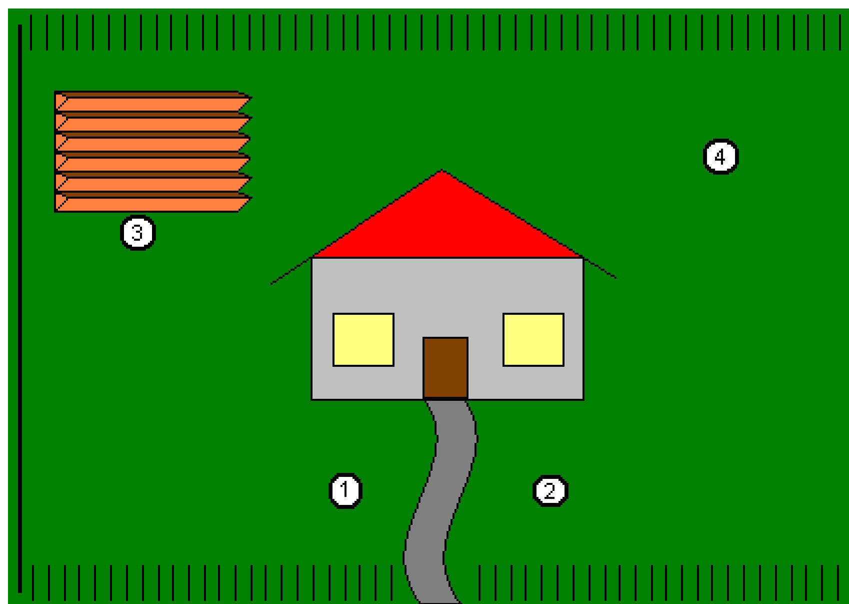


Obrázek 11. Celkové zapojení.

Spojení není ideální, takže je potřeba dobře zasunout vývody do trubiček a překontrolovat zda je spoj vodivý. Protože čip mikrokontroléru obsahuje všechny potřebné části, je na rušení náchylná pouze cesta BGDN, kde probíhá sériový přenos. Rychlost je sice možná poměrně nízká, přesto je dobré tomuto spoji věnovat více pozornosti. Stávalo se mi, že spojení nebylo ustanoveno, když spoj nebyl opravdu těsný a bylo třeba připojit trubičky i na okolní piny, aby zajistili a přidrželi spojení, ikdyž jinak nebyly využity. Ostatní napojení jsou poměrně bezproblémová, stabilní hodnoty napětí jsou odolné proti malým kapacitám přechodů, které mohou vzniknout nedokonalým obepnutím pinu pomocí trubičky. Nevhodné je toto zapojení v případě využití periferií jako je A/D převodník a podobně, proto jsem v testech žádné z těchto zařízení nevyužil. Pro otestování chování jader v kritických úlohách, kde by mohlo docházet k nežádoucím přerušením v době A/D převodu nebo přenosu po sériové lince, by bylo nutné vyrobit plošný spoj s potřebnými součástmi tak, aby byly dokonale spojené s čipem mikrokontroléru. Tato problematika ale zahrnuje spousty dalších problémů, které jsou nutné vyřešit v samotných jádrech a přístupech úloh k těmto periferiím a žádné z testovaných jader není ničím takovým vybaveno. Režie nutná k ošetření takovýchto extrémních případů je příliš vysoká na možnosti 8 bitových mikrokontrolérů a proto se spoléhá na korektní chování jednotlivých úloh tak, aby k těmto situacím nedocházelo.

7.1.6 Ovládání a řízení zavlažování kolem rodinného domku.

O zavlažování okolí fiktivního rodinného domku se stará čtveřice vodních rozstřikovačů, a zavlažovací systém zahrádky. Každý rozstřikovač má vlastní načasování ve kterém spouští vodu, aby bylo možno obsloužit každý úsek zahrady odpovídajícím množstvím vody v závislosti na intenzitě vypařování v různých koutech zahrady, ať už z důvodů že na jižní stranu svítí více sluníčka nebo že mezi domy vane siný vítr a vysušuje ten krásný anglický trávník. Zároveň je možné vnějším signálem spustit zavlažování na všech místech, pro případy zvláště horkých dní. Řídící systém je také propojen se vstupní brankou a vchodovými dveřmi, na jejichž otevření a zavření reaguje zastavením dvou rozstřikovačů umístěných v blízkosti cestičky mezi brankou a vchodovými dveřmi. Aby při příchodu nebo odchodu z domu nehrozila sprška vody. Situaci lépe vykreslí obrázek 10.



Obrázek 10. Plánek domku, bílá kolečka označují jednotlivé zavlažovače.

Reakce spočívá ve vypnutí dvou rozstřikovačů „1“ a „2“ v okamžiku, kdy někdo vstoupí brankou nebo otevře dveře domu, a opět se uvedou do provozu v okamžiku, kdy někdo opět otevře branku nebo dveře. Systém není dokonalý a na reálné použití by bylo třeba více než jen spínače reagující na otevření či zavření dveří a branky. Protože v okamžiku, kdy by člověk odcházel, zatím co by jiný přicházel, tak by systém špatně zareagoval a vyhodnotil to opětovným spuštěním rozstřikovačů. V testovacím prostředí je tato složka přítomna z důvodu vnesení asynchronních požadavků na spuštění úlohy. Aby bylo možné otestovat chování i v téhle oblasti.

Časované rozstřikovače představují naopak synchronní úlohy s pevně danou periodou a dobou trvání. Jejich plánování by mělo být bezproblémové. Proto byl přidán násilný stav zapnutí a vypnutí, pomocí ovladače. Je to způsob jakým se pokusím zmást algoritmy plánování, nečekanou událostí. A sledovat úspěšnost reakce jádra a plánovacího mechanizmu na tuto situaci.

7.2 Testovací program

7.2.1 Vyzývací smyčka

Zcela tou nejzákladnější metodou a testovaným případem je vyzývací smyčka. Pro každou z úloh existuje jednoduchá funkce a ty funkce jsou následně v cyklu volány jak přijdou na řadu procesoru. Plánování není žádné, nebo je ponecháno na programátorovi. Já zvolil funkce *ObsluhaOdstrikovace1 – 4* pro každý ze 4 zavlažovacích rozstřikovačů. Tyto funkce nejprve otestují zda uplynul čas jejich periody, která je nastavena na hodnoty pevně v jejich programu. Zda jsou splněny další závislosti potřebné pro spuštění a následně upraví hodnotu na příslušném pinu, kde v rámci testování je pouze dioda pro signalizaci. V reálném nasazení by signál vedl k ovládacímu ventilu pouštějícímu vodu do jednotlivých rozstřikovačů. Dále existuje zpožďovací smyčka, která v reálném obvodu zajistí čas potřebný na otevření či uzavření ventilu, aby nedošlo k situaci, že ventil nebude zcela otevřen a vlivem náhodné události přijde stav nutící uzavření ventilu změnou logické úrovně na příslušném vodiči, což by mohlo vést k poškození ventilu nebo navazujícího vodovodního systému. Periody spouštění jednotlivých rozstřikovačů bude nejspíše nutné během testování měnit.

Další funkcí je *StavChodnicku*, tato funkce představuje úlohu která využívá neustále se opakující dotazování na stav senzorů na brance a na vchodových dveřích. Na jejich základě vypíná nebo povoluje dva rozstřikovače v okolí chodníčku vedoucího k domu. Současný stav, zda na chodníčku někdo je nebo není, indikuje stavem logické úrovně na jednom z vývodů mikrokontroléru, tuto hodnotu zároveň využívá jako paměť k uložení stavu. Jako indikaci je možné připojit buď diodu nebo přes výkonový tranzistor nějaké silnější signalizační zařízení.

Funkce *ObsluhaZavlazovani* reaguje na vnější signál na jednom pinu mikrokontorléru a pokud je nastaven, tak přepne všechny rozstřikovače do stejného stavu. V tomto případě vypnutého pro možnou práci na zahradě. Tento stav signalizuje na dalším pinu, kde je vhodné umístit indikační diodu. Po přepnutí spínače opět do nulového stavu, přestane svítit indikační dioda a systém nechá volný přístup časovaným spouštěním zavlažování.

Samotné časování je zařízeno funkcí *Timer*, tato funkce reaguje na stav přetečení čítače a na jeho základě zvýší počítadlo tiků úloh. Toto počítadlo je pak teprve používáno ke spouštění jednotlivých úloh programu podle jejich period. Zavedení tohoto počítadla je nutné z důvodu principu vyzývací smyčky. Ta není přizpůsobena na vyrovnání se s asynchronní událostí, kterou přerušení od timeru určitě je. Přestože při testovaní dojde k vyzkoušení reakcí i při použití asynchronních signálů, jako je reakce na otevření branky. Tak tyto signály jsou z hlediska programu vyzývací smyčky synchronní. Nejsou signalizovány přerušením, ale na jejich nastavení se přijde až v okamžiku zavolání obslužné úlohy. Čímž je zachován systém časování vyzývací smyčky a ta není v průběhu výpočtu narušena přerušením, které by vedlo k situaci, že v průběhu spouštění rozstřikovače bude tento stav narušen a dojde k možnému poškození zařízení, i v případě, že tato situace není časově

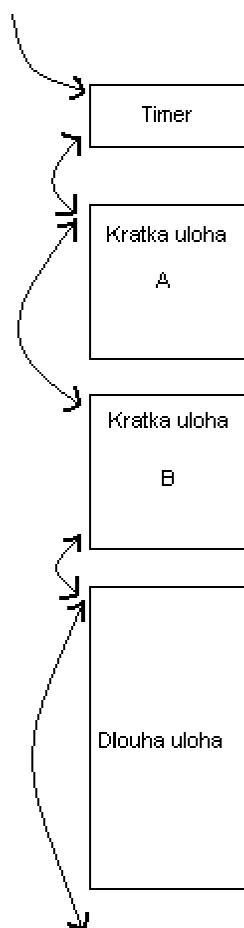
kritická, pouze přerušení nastalo v nevhodnou dobu. Naopak způsobí, že pokud snížím hodnoty na kritickou mez, tak nedojde ke spuštění některého z rozstřikovačů, případně při vstupu někoho brankou či dveřmi, nedojde k vypnutí rozstřikovačů, protože systém nestihne zareagovat.

7.2.2 RM

Je zde použit plánovací mechanizmus RM. Proto přibyla funkce plánovače. Úlohy již nejsou plánovány podle vůle programátora, ale o jejich naplánování se postará plánovač. Přibyla ale nutnost zadávat periody pro jednotlivé úlohy. A paměťová náročnost stoupla o právě zmíněný plán. Ten představuje pole indexů jednotlivých úloh.

Hlavní smyčka již neprovádí úlohy pořád dokola, ale reaguje na základě přerušení od časovače, toto opatření nemá při její softwarové realizaci přílišné využití, protože není možné mikrokontrolér uspat v době nečinnosti, čímž by se dosáhlo znatelné úspory energie. V menšině případů poběží mikrokontrolér na plné využití. Ve většině případů poběží určitou dobu na prázdno, protože i kdyby byly úlohy naplánované pro mikrokontrolér tak, že maximálně využijí jeho výpočetní sílu, tak se málo kdy stane, že nastane situace kdy poběží všechny úlohy nejdelší možnou programovou cestou. Proto hlavně při provozu na baterie by bylo vhodné využít této vlastnosti a doplnit kód programu jádra tak, aby po skončení provádění všech naplánovaných úloh povolil přerušení od časovače, nastavit vektor přerušení na konec smyčky jednoho plánovacího cyklu a zapnout úsporný režim provozu. Bez tohoto doprogramování slouží toto čekání pouze k synchronizaci zpracování s hodinovým kmitočtem.

Přestože jedna z úloh je timer a je i součástí plánu, je nutno ji volat před každou úlohou. Ty jsou shodné s úlohami použitými ve vyzývací smyčce a pro jejich správnou funkčnost je třeba aby byly takto rozděleny na co nejmenší celky, zaručující, že každý jeden skončí maximálně v jednom tiku. Složitější úlohy jsou doplněny o aktualizaci timeru v průběhu zpracování, aby nedošlo ke ztrátě informace že vznikl tik. Z důvodů plánování by nebylo nutné ještě ponechávat test na periodu na počátku úlohy, ověřující, zda se má spustit ovládání rozstřikovače, nebo je to jen další volání z důvodů cyklu. Ale plán by pak musel být dostatečně dlouhý a to jako nejmenší společný násobek všech period. A při kombinaci úloh s periodami 1,3,1000 by znamenalo plán o 3000 položek a nestačila by přidělená paměť. Proto jsem zůstal u způsobu, kdy je plán volán v cyklu a je využit spíše než jako pořadí zpracovávání jednotlivých úloh, jako prioritní systém. V plánu je pořadí priorit jednotlivých úloh na zpracování a během jednoho cyklu jsou všechny úlohy spuštěny. A rozhodnutí, zda je čas na provedení dané úlohy je ponechána na každé jednotlivě.



Obrázek 11. Schéma procházení pořadí úloh.

Jak ukazuje obrázek, dojde k tomu, že plán seřadí úlohy podle priorit RM a ty jádro začne volat ve smyčce v pořadí daném plánem. Každá z úloh na svém počátku otestuje, zda je požadavek na její provedení a pokud ano, tak se vykoná a pokud ne, tak předá řízení zpět do jádra a tím další úloze v pořadí. Na obrázku 11 je znázorněna situace, kdy dojde k postupnému vykonání úlohy Timer a krátké úlohy B. Ostatní úlohy jsou přeskočeny. Plán není během provádění úloh nijak měněn. Testy na to, zda je požadavek na vykonání dané úlohy, zůstaly zachovány z použití při plánování pomocí vyzývací smyčky, což je dobré i z hlediska testování, protože tím dochází k provádění shodného kódu úloh a měření bude přesnější.

Oproti vyzývací smyčce přibyla funkce plánovače. Jeho jedinou úlohou je na počátku po resetu mikrokontroléru setřídit plán úloh podle zadaných period a určit vhodné priority.

7.2.3 EDF

Při použití algoritmu EDF již není dobré používat testy na počátcích každé z úloh, zda je čas na její provedení, protože algoritmus je pojatý rozdílně. Umožňuje plánování asynchronních úloh a úloha může být zpracována mimo pořadí a zmíněný test by způsobil její nesplnění. Vedle pole plánu přibylo pole přerušení. Jsou v něm indikovány případy, kdy vyvstane žádost o vykonání nějaké úlohy a tento

příznak je odstraněn až v okamžiku, kdy je tento požadavek obsloužen. Starost o indikování přerušení je obsloužena právě funkcí plánovače. Ten je volán při každém volání jádra a ve smyčce vrací odkaz na právě naplánovanou úlohu. Výhoda tohoto přístupu je v možnosti reagovat na asynchronní požadavky na úkor větší zátěže procesoru jádrem. Požadavky na vykonání úlohy jsou synchronizovány s časem a dořešeny zpětně porovnáním s globální hodnotou LONG_TIME. Jejich nastavení se provádí při každém předání řízení jádru v nejbližší nové časové jednotce. Přesto to neovlivňuje podávání úloh na zpracování. Zůstává tedy výhoda, z metod řízení vyzývací smyčkou i RM, v možnosti zpracování více úloh v jedné časové jednotce. Čekání a nic nedělání procesoru nastane, až když není pro současný stav času požadavek na žádnou z úloh a čekací doby, kdy procesor nic nedělá jsou delší. A při použití přepínání do spánku z důvodů úspory energie tak nedochází k tak častému přepínání mezi spánkem a během procesoru. Ale přesto zůstává potřeba aktualizace času při běhu delších úloh a rozumné umísťování volání funkce *Timer* pro aktualizaci času.

Úlohy jsou obslouženy tedy jen v případě, že na ně vyvstal požadavek a na rozdíl od předešlých algoritmů plánování, tedy není třeba testování při vstupu do úlohy, zda nastal vhodný čas na její provedení.

7.2.4 FreeRTOS

U prostředí FreeRTOS existuje příjemné API pro tvorbu uživatelských úloh. Ve své podstatě si vystačíme s jednou z funkcí na zaregistrování úlohy. Já jsem využil funkci *xTaskCreate* a funkci na spuštění jádra a začátek plánování úloh *vTaskStartScheduler*. Pomocí těchto funkcí se zaregistrují úlohy ve frontě na zpracování a spuštěním plánovače je jádro začne podle nastavených hodnot předávat procesoru. Tyto hodnoty jsou definovány v souboru *FreeRTOSConfig.h* a možností co nastavit a jak, je spousta. Od základních, jako použitý plánovací mechanizmus, velikost využité paměti jádrem, možnosti přidělení velikosti zásobníku jednotlivým úlohám, až k možnostem ovlivňujícím počet úrovní priority, kterou je možné přiřadit úlohám. Zvláštní kolonku představuje možnost zavedení či odstranění jednotlivých funkcí jádra poskytovaných API FreeRTOS, je možné povolit či zakázat některé funkce a tím docílit snížení nároků systému na velikost programu v úlohách, kde je to třeba. Neboť funkce mají podmíněný překlad právě podle hodnot nastavených v tomto souboru. Každopádně využití tohoto je hodně omezeno, systém FreeRTOS má hlavně vysoké nároky na paměť a v čipech majících dostatek paměti na data už velikost samotného programu nebývá kritická. Kompletní seznam všech nastavení je samozřejmě k nalezení v manuálových stránkách k FreeRTOS.

Já jsem zvolil základní nastavení, které by mělo být optimální pro většinu úloh a nasazení, ve funkci *xTaskCreate* je jako hlavní parametr potřeba předat ukazatel na funkci úlohy kterou chceme zaregistrovat. Tyto funkce musejí být vytvořeny jako nekonečné smyčky. Jejich přepínání si obslouží jádro samo. Na naplánování periodického spouštění rozstřikovačů jsem využil další funkce API, které umožňují uspání úlohy na stanovený čas, pokud není její činnosti třeba. V jedné z obměn systému

použitých při testování byla i varianta, kde jsem opět použil funkci *Timer* a nechal úlohy běžet nepřetržitě bez pozastavování a v každé úloze ovládající jeden rozstřikovač testoval, zda doběhl čas odpočítávaný úlohou Timer a má se spustit zavlažování. Tato metoda ale byla z mnou neodhalených příčin neskutečně pomalá a v rámci testů by FreeRTOS dopadla velice spatně, ikdyž se domnívám, že je to způsobeno nastavením délky periody po kterou má úloha přístup k jádru než dojde k přepnutí kontextu, tak se mi toto tvrzení nepodařilo prokázat. Podobné, ale mírnější zhoršení stavu přišlo v situaci, kdy jsem nahradil všechny 4 úlohy ovládající rozstřikovače jednou velkou úlohou, která se větvila podle aktuálního času.

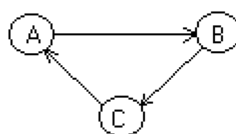
Nejspíše by bylo možné vytvořit i lépe sestavenou úlohu pro FreeRTOS, jeho API poskytuje spousty možností, jak ovládat a přepínat jednotlivé úlohy ve frontě na zpracování, možnosti předávání parametrů a spousty dalších možností.

7.2.5 QP

Tento systém je postaven na dosti odlišném principu. V první řadě je nutno říci, že vzhledem k tomu, že využívám kvůli nárokům na hardware pouze jádro jinak rozsáhlého systému QuantumPlatform, tak jeho možnosti co do uživatelského rozhraní, jsou omezené.

Systém QP je založen na principu objektového přístupu kombinovaného se zasíláním zpráv. Jádro je napsané nezávisle na platformě a proto v uživatelsky napsaném programu je jako první třeba zavolat funkci inicializující daný hardware. A pak funkce inicializující jednotlivé úlohy. A nakonec spuštění samotného plánování a běhu systému. Ještě před spuštěním je třeba vytvořit speciální pole, které obsahuje počáteční funkce a definice jednotlivých úloh. Vlastně dochází k dvojímu iniciování úlohy pro jádro.

Samotné úlohy jsou pak tvořeny většinou jako stavové automaty. A předávání hodnot, informací a přepínání je tvořeno přes volání jádra a využívání jeho stavů. Nedochází k přímému volání funkcí. Přepínání jednotlivých funkcí je tvořeno voláním funkce jádra, která má jako jeden parametr ukazatel na funkci kam se má přepnout řízení. Tento způsob zajišťuje přepínání řízení v jednotlivých stavech pomyslného stavového automatu. Každá funkce představuje jeden stav. Není možné v tomto případě totiž volat standardní volání funkcí, protože by došlo k přeplnění zásobníku. Pokud z jedné funkce/stavu dojde k přejití do jiné funkce/stavu, tak při standardním volání není volající funkci možné odstranit ze zásobníku.



Obrázek 12. Graf jednoduchého stavového automatu

Tato varianta přes volání jádra, ale zajistí přepnutí řízení bez ukládání návratových hodnot na vrchol zásobníku a zajistí vypořádání s proměnnými. Protože i jednoduchá úloha představovaná pomyslným automatem, jako na obrázku 12., kde může jít o přepínání třech barev na semaforu, by rychle způsobila přetečení zásobníku.

Pro případ úlohy o zavlažování zahrady, ale tento způsob nepoužiji. Není třeba zavádět stavový automat a je možné využít klasického přístupu, použitého již pro FreeRTOS. Jednotlivé úlohy jsou v nekonečné smyčce, takže žádná z funkcí nikdy neskončí a pro periodické časování je použito funkce, která předá řízení jádru s tím, že má navrátit řízení do dané funkce za udaný časový interval.

7.3 Zhodnocení programování

První zhodnocení je trochu subjektivní. Chtěl bych tu shrnout moje zkušenosti z programování jednotlivých typů jader, protože i to je pro jejich použití v praxi podstatné. Přesto, že systémy jako QP a FreeRTOS mají jádro a tudíž základ hotov a je třeba dodat již pouze samotné úlohy co se budou provádět, tak zvládnutí a pochopení jejich použití zabere více času než realizace jinými způsoby. Navíc se v těchto systémech obtížně hledají chyby. Zvlášť co se týká QP, tak jeho poněkud zvláštní přístup oproti jiným je složitý na pochopení. Uvažování o všech vazbách mezi jednotlivými úlohami a jejich stavu je zvlášť u velkých projektů značně náročné. Programátor musí spoléhat na pravdivost uvedených informací a věřit ve správné plánování úloh. Nebýt možnosti pozastavování výpočtu pomocí prostředí CodeWarrioru, tak netuším, jak by se dalo ověřit, že úlohy splňují limity a nedochází ke kolapsu sytému.

Přesto základní programování ve FreeRTOS bych považoval za dobré, pokud nedojde k problémům, tak jeho použití je nepoměrně přehlednější než u konkurence. Přesto pro použití na takovýchto jednoduchých platformách jako je HC08, bych raději využil jednodušší vyzývací smyčku. Program je tak velice přehledný a programátor na první pohled vidí co se kde děje. I bez nutnosti trasování programu, lze snadno určit zda se výstup chová korektně. Jako největší nevýhodu bych uvedl, že takovýto jednoduchý program není v podstatě plánován a tím optimalizován. Je vždy potřeba v časovém plánu smyčky počítat se situací, kdy daná úloha využije na maximum čas, který je jí přidělen, a tento časový úsek musí být dostatečně velký na to, aby pokryl všechny potřeby úlohy, ale tato situace málo kdy nastane a proto mikrokontrolér převětšinu času neprovádí užitečnou práci. Dochází k plýtvání strojového času. A vyvážit toto je také úkol, který stojí na programátorovi a stěžuje mu tak práci. Je třeba určit a spočíst vhodné způsoby spouštění, aby tato ztráta byla co nejmenší. Tato situace s využíváním procesoru se postupně zlepšuje a tak mi přijde jako nejlepší řešení využití jednoduchého EDF, kdy je toto ještě stále možné naimplementovat přehledně a jednoduše od základů, a zbavit se i nutnosti přemýšlet o pořadí spouštění a dělení jednotlivých úloh.

8 Testy

Teď, když už je jasné že i malý 8 bitový mikrokontrolér zvládne využívat plánovacích mechanizmů a i složitějších jader realtime OS, je vhodné zjistit, jak si jednotlivá jádra povedou při extrémním zatížení. Cílem všeho je otestovat schopnosti zpracování jednotlivými jádry a to tak, že jednotlivým

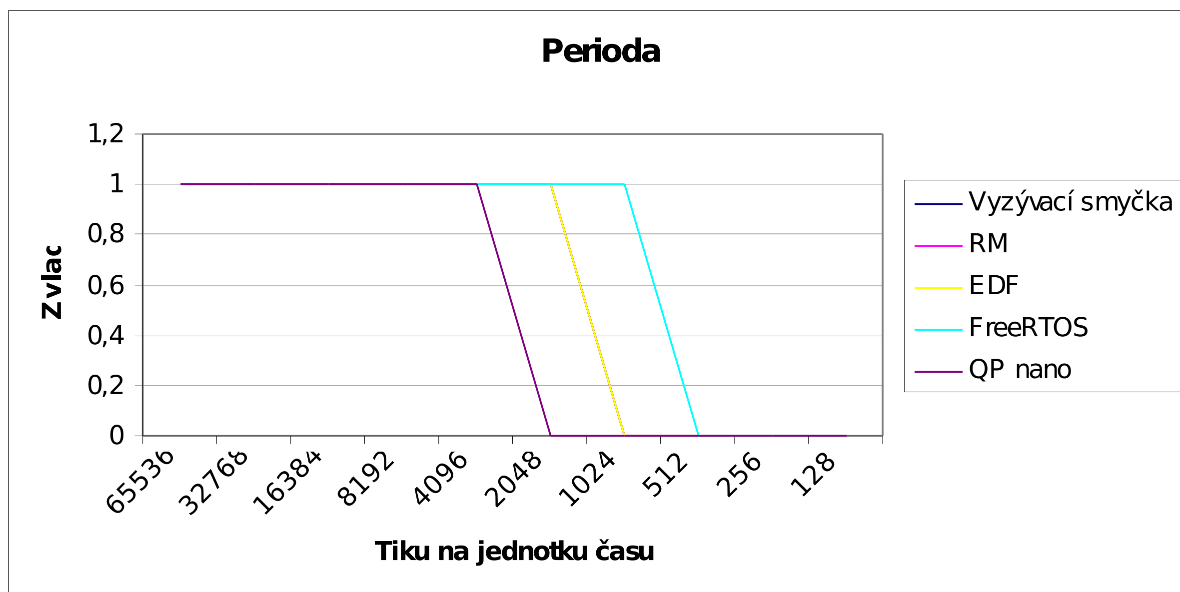
jádrům vždy upravím kód úloh, nebo periodu časovače. A to tak, aby kód nepříslušející k plánování a který rozlišuje jednotlivá jádra zůstal stejný.

8.1 Perioda

Prvním testovaným parametrem bude plánovací schopnost u všech jader. Postupně budu snižovat velikost periody jednoto tiku úloh a současně budu zjišťovat které z jader při tomto nastavení způsobí chyby v plánování tak, že překročí hodnoty povolené periodou (deadline). Tím budu ověřovat, že systém zvládne naplánovat úlohy, ikdyž ty budou náročnější. Tím, že já snížím periodu tiku, začnou nastávat požadavky na spouštění úloh a události čím dál častěji. Podle předpokladu by se s tímto postupem měly nejlépe vyrovnat jednodušší plánovací jádra, protože mají skoro nulovou režii na přepínání mezi úlohami. Oproti tomu složitější plánovače ve složitějších jádrech by měly lépe využít čas procesoru, a tak získat pro prováděné úlohy více času.

Tiku na jednotku času	Vyzývací smyčka	RM	EDF	FreeRTOS	QP nano
65536	ano	ano	Ano	ano	ano
32768	ano	ano	Ano	ano	ano
16384	ano	ano	Ano	ano	ano
8192	ano	ano	Ano	ano	ano
4096	ano	ano	Ano	ano	ano
2048	ano	ano	Ano	ano	ne
1024	ne	ne	Ne	ano	ne
512	ne	ne	Ne	ne	ne
256	ne	ne	Ne	ne	ne
128	ne	ne	Ne	ne	ne

Tabulka 2. Hodnoty pro různou periodu



Obrázek 13. Úspěšnost na změnu periody

Na tomto testu se zajímavě umístil FreeRTOS, ale je možné, že mnou pořízené výsledky jsou ovlivněny rozdílným způsobem nastavování velikosti periody pro mnou sepsaná jádra a pro jádra FreeRTOS a QP, kde jde jen o nastavení parametrů a jejich skutečná realizace je skryta v kódu těchto jader.

8.2 Náročné výpočty

Při dalším testu je zvětšena náročnost výpočtu. Protože se jedná jen o fiktivní zavlažovací systém, tak zvětším velikost zpoždění výpočtu, nutnou pro otevření ventilu a ustálení stavu. Přidávám počet opakování cyklu u jednotlivých úloh, obsluhujících ventily rozstřikovače. Zde by se naopak měla projevit schopnost složitějších plánovacích mechanizmů, zařadit do výpočtu méně náročné úlohy s kritickým časem. Naopak je ale možné, že dobré naplánování úloh u jednodušších jader povede k lepším výsledkům. Budu zjišťovat, které z jader při tomto nastavení způsobí chyby v plánování tak, že překročí hodnoty povolené periodou (deadline).

Počet opakování	Vyzývací smyčka	RM	EDF	FreeRTOS	QP nano
1	Ano	ano	ano	ano	Ano
2	Ano	ano	ano	ano	Ano
3	Ano	ano	ano	ano	Ano
4	Ano	ne	ano	ano	Ne
5	Ano	ne	ano	ano	Ne
6	Ne	ne	ano	ne	Ne
7	Ne	ne	ne	ne	Ne
8	Ne	ne	ne	ne	Ne
9	Ne	ne	ne	ne	Ne
10	Ne	ne	ne	ne	ne

Tabulka 3. Náročné výpočty

Obrázek 14. Náročné výpočty

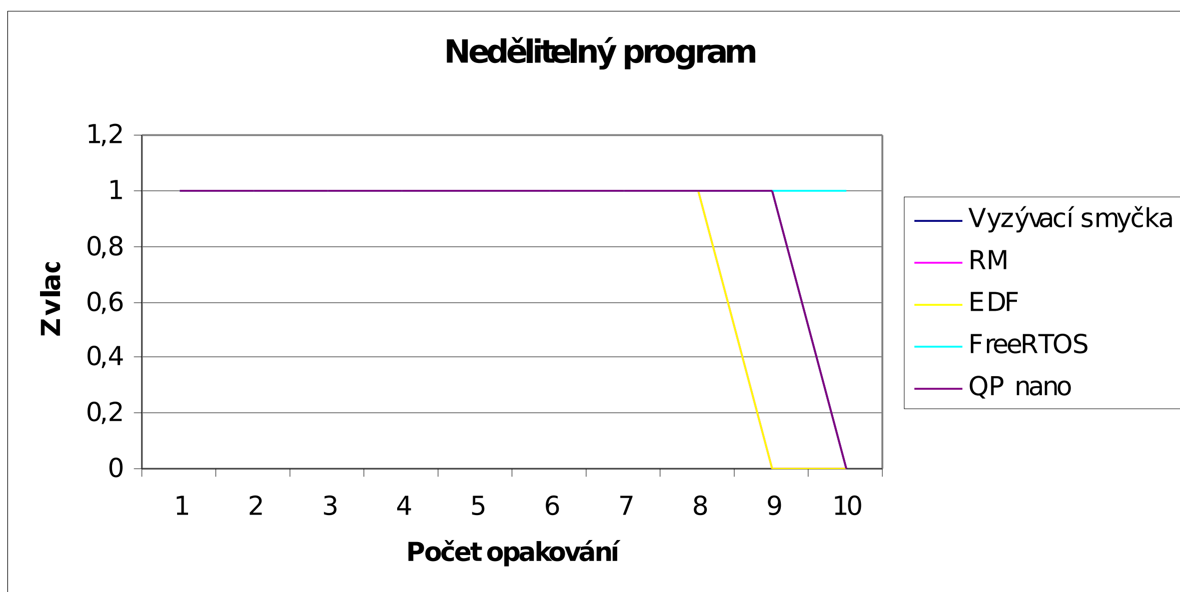
Zde se dost projevila záměrně špatná pořadí ve smyčkách zpracování u vyzývací smyčky a RM mechanizmu, kdy jsem záměrně neřadil úlohy tak, jak by optimálně měly být umístěny, pro pravdivější výsledky neovlivněné mým vlastním naplánováním.

8.3 Nedělitelný program

Při dalším testu použiji pouze jednu velmi náročnou úlohu, na rozdíl od předchozího testu, kde byly sice postupně také více náročné úlohy, ale všechny. Testem se snažím zjistit chování v prostředí, kde existuje jen jedna náročná úloha, těžce rozdělitelná. Málokdy se v reálném použití stane, že by běželo více úloh s přibližně stejnou náročností. Většinou existuje jedna úloha, běžící po většinu času procesoru, a jen v okamžicích jsou spouštěny vedlejší úlohy obsluhující třeba klávesnici. Opět budu postupně zvyšovat náročnost na výpočet této jedné složitější úlohy, do okamžiku, kdy dojde k výpadku plánování a nedodržení časových mezí.

Počet opakování	Vyzývací smyčka	RM	EDF	FreeRTOS	QP nano
1	Ano	ano	ano	ano	ano
2	Ano	ano	ano	ano	ano
3	Ano	ano	ano	ano	ano
4	Ano	ano	ano	ano	ano
5	Ano	ano	ano	ano	ano
6	Ano	ano	ano	ano	ano
7	Ano	ano	ano	ano	ano
8	Ano	ano	ano	ano	ano
9	Ano	ne	ne	ano	ano
10	Ne	ne	ne	ano	ne

Tabulka 4. Nedělitelný program



Obrázek 15. Nedělitelný program

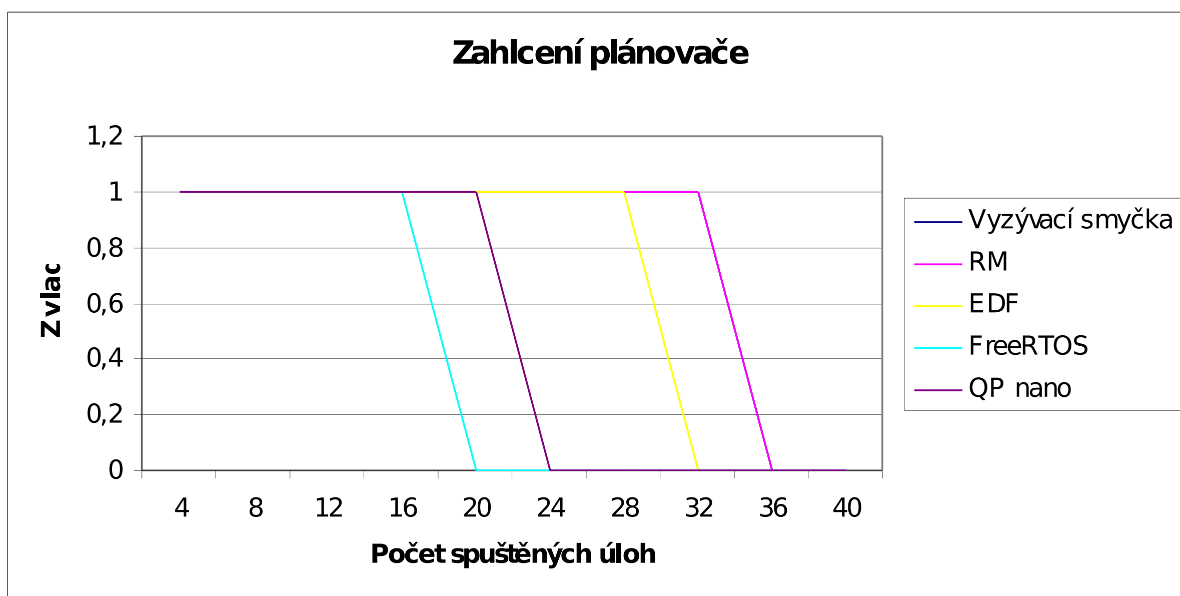
Zde se projevila schopnost lepších plánovacích mechanizmů vypořádat se s dělením programu, mnou naprogramovaná jádra nedosahovala ani z části jejich kvalit, tak jsem přistoupil k ruční optimalizaci kódu provádění úloh ve vyzývací smyčce. Tím jsem získal srovnatelné výsledky. Přece jen každý programátor bude podvědomě nějakým způsobem svůj program optimalizovat.

8.4 Zahlcení plánovače

Při dalším testu bude jádro obsluhovat postupně čím dál větší množství malých úloh. Budu přidávat do zpracování další rozstřikovače do okamžiku, než dojde k situaci, že se nedostane na řadu jeden z těchto rozstřikovačů. Jde o opak předchozí krajní meze, s tím, že zde je celé zpracování dobře rozdělitelné na spousty malých úloh a ověří hlavně, jak se projeví rozdílná náročnost jednotlivých způsobů plánování. Předpokládám, že nejlépe v testu obstojí jednodušší plánovací mechanizmy, protože mají nulovou režii na zpracování plánu a přepínání úloh je jen otázkou přímého volání podprogramu.

Počet spuštěných úloh	Vyzývací smyčka	RM	EDF	FreeRTOS	QP nano
4	Ano	ano	ano	ano	ano
8	Ano	ano	ano	ano	ano
12	Ano	ano	ano	ano	ano
16	Ano	ano	ano	ano	ano
20	Ano	ano	ano	ne	ano
24	Ano	ano	ano	ne	ne
28	Ano	ano	ano	ne	ne
32	Ano	ano	ne	ne	ne
36	Ne	ne	ne	ne	ne
40	Ne	ne	ne	ne	ne

Tabulka 5. Zahlcení plánovače



Obrázek 16. Zahlcení plánovače

Zde se ukázala síla jednoduchých technik. Nároky plánovacích mechanizmů u FreeRTOS a QP jsou hodně výpočetně náročné úlohy v porovnání s užitečným kódem a jejich časté spouštění rychle vedlo na vyčerpání paměti a kolaps celého systému. Zatímco jádra prakticky bez plánování dosahovala shodných hodnot. Trochu pokulhává EDF se svým plánovačem, ale ten není přímo závislý na počtu úloh a tak problémy nenastávají tak brzy.

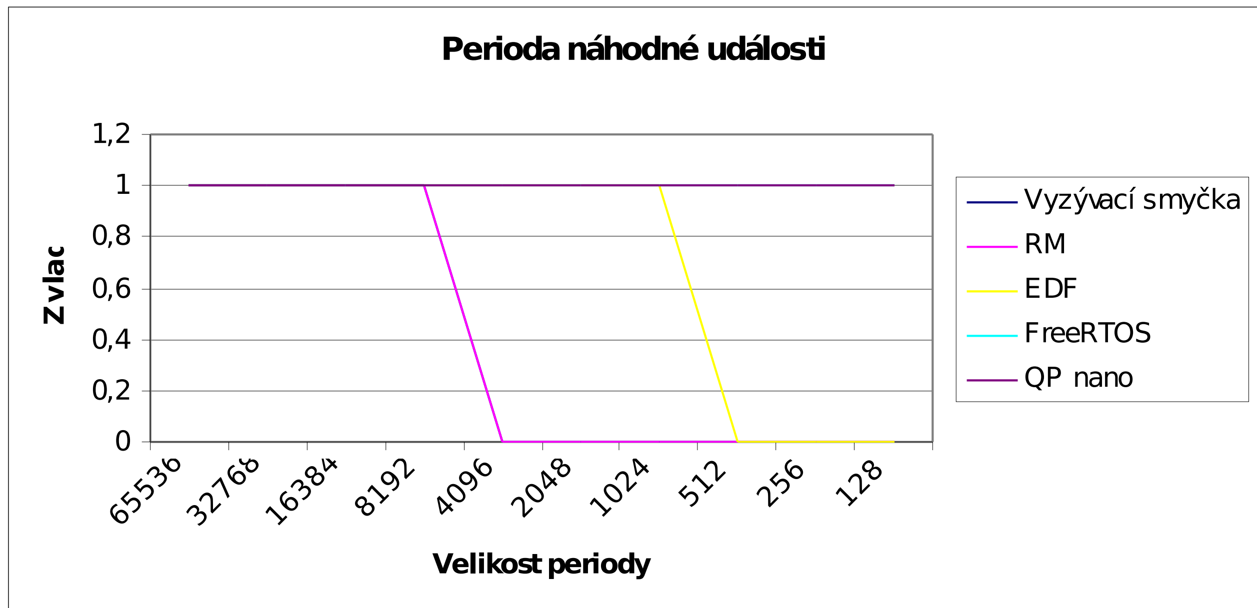
8.5 Náhodné události

Při dalším testu budou zvýhodněny složitější plánovací mechanizmy. Budu se snažit pomocí signálů na vstupech hlídajících branku a dveře domu, vytvořit postupně větší a větší množství náhodných událostí a sledovat do jaké míry je bude schopno jádro zpracovat a obsloužit. Aby testování v této oblasti bylo směrodatné, budu provádět u všech jader přerušení na základě počtu projitých tiků procesoru. Pomocí debuggeru, vždy když procesor provede stanovený počet tiků, zastavím běh programu a nastavím bit vstupního pinu na úroveň log 1 tak, abych vyvolal požadavek na obsluhu takto vzniklé události. A při testování budu postupně snižovat počet tiků než zastavím výpočet a nastavím parametry na požadavek na obsluhu náhodné události. Program vyzývací smyčky ani RM plánovacího mechanizmu nebudu upravovat.

Perioda náhodné události	Vyzývací smyčka	RM	EDF	FreeRTOS	QP nano
65536	Ano	ano	ano	ano	ano
32768	Ano	ano	ano	ano	ano
16384	Ano	ano	ano	ano	ano
8192	Ano	ano	ano	ano	ano
4096	Ne	ne	ano	ano	ano
2048	Ne	ne	ano	ano	ano

1024	Ne	ne	ano	ano	ano
512	Ne	ne	ne	ano	ano
256	Ne	ne	ne	ano	ano
128	Ne	ne	ne	ano	ano

Tabulka 6. Náhodné události



Obrázek 17. Náhodné události

Zde se opět projevily nedostatky jednoduchých mechanizmů. Nejsou stavěny na častá přerušení a i poměrně málo frekventované náhodné události, pokud přijdou v nepravý okamžik výpočtu, vedou k nedodržení časové meze na zpracování. Trochu to vylepšuje mechanizmus EDF, kdy dochází k plánování každou periodu hodin. Jednoznačně nejlépe vyšly mechanizmy QP, ale mám podezření, že je to způsobeno hlavně nízkou náročností úlohy na zachycení přerušení.

8.6 Shrnutí výsledků

Bohužel, jak se ukázalo, tak pokročilejší plánovací mechanizmy mají na takto omezené platformně slabé využití. Nároky na výpočty plánovače a nutnost využívání, na tyto poměry velkého množství paměti, z nich nedělají kandidáty na nasazení na těchto zařízeních. Jsou konstruovány jako univerzální jádra, což jim dává určitě lepší možnosti na prosazení na trhu, než jednoduchým jednoúčelovým plánovačům. Toto si ale vybírá daň v té podobě, kdy nasazení a následný výkon není ideální. Je zřejmé, že velkou míru na testech měl použitý řídící program, který je velice jednoduchý, a tak nevyužívá možností, které poskytují složitá jádra. Ale na 8 bitových mikrokontrolérech ani nikdo nečeká, že poběží několik stovek složitých úloh, kde by se režie vzniklá spouštěním jádra a plánováním vyplatila. Jako nejuniverzálnější, a co do nasazení podle mých testů nejvhodnější, je jednoduchá implementace EDF algoritmu.

Závěr

Z poznatků zjištěných během práce vyplívá, že jádra real-time operačních systémů lze s úspěchem používat i na tak malé a omezené platformě, jako je HC08. Hlavním kritériem je paměť a její velikost přímo určuje možnosti toho, co lze naimplementovat. Vždy to bude ale pouze plánovač a není možné očekávat přítomnost ovladačů periferií a podobné věci známé z PC. Plánovací mechanizmy jsou použitelné všechny. Ale vzhledem k využitelnosti složitých plánovacích technik využívajících přerušení prováděné úlohy a s tím spojené náročné operace na managment správy paměti, jsou vhodnější jednodušší plánovací techniky. S tím je spojen i rozpor ohledně toho, zda je vhodné použít jádra real-time OS na takto omezené platformě. Z hodnot získaných při testech a i s ohledem na potíže potřebné pro realizaci, je dle mého názoru vhodnější použít cílený program a čas, který by zabralo nastavení a implementace plánovače a dalšího kódu navíc, věnovat tento čas raději optimalizaci cíleného kódu a tím dosažení ještě lepší efektivity. I když dnes není cena HW a nutnost optimalizace kódu tak rozsáhlá, tak pořád platí, že čím levnější výrobek vyrobím, tím snáze ho prodám.

Při realizaci diplomové práce jsem musel vyřešit množství problémů, které se přímo netýkaly jader real-time OS a ve výsledném stavu představují většinu práce, kterou jsem na diplomové práci udělal. Při pokračování v této problematice by rozhodně stálo za zvážení, porovnání s výkonnější platformou a třeba i možnost napsání komplexního jádra real-time OS přímo pro HC08, vybaveného správou paměti a složitějším, ale kvalitnějším plánovačem a ověřit tak důkladněji možnosti HC08. Na mých testech se jednoznačně projevily problémy při realizaci, kdy je těžko ověřitelné, do jaké míry měl vliv mého připojení mikrokontroléru k PC a sledování provádění kódu na výsledky měření. Přenos po mých trubičkách jistě nebyl ideální a sériová komunikace tím mohla hodně utrpět a zkreslit tak hodnoty.

Přesto je přínosné ověřovat možnosti nasazení RTOS na HC08. Doba jde kupředu a programátoři si rychle zvykají na snadné programování pomocí API dodávaného s jednotlivými volně dostupnými jádry a je dobré tedy sledovat, zda jejich nasazení povede ke kýženému zisku. Mnou použitá jádra jsou hodně univerzální, nejsou přímo určena pro HC08, a tak je velice pravděpodobné, že pokud bude tento trend v používání výkonnějších programovacích prostředí i na takto omezených platformách pokračovat, tak jádra určená přímo pro HC08 dokáží dosáhnout zas o něco lepších výsledků, než jejich univerzální kolegové.

Literatura

[1] Strnadel, J., *Real-time operační systémy – studijní opora*. Brno, 2006.

[2] Quantum LeaPs. *Quantum Platform* [online]. [cit. 2008-12-04]. Dostupné na URL: <http://www.quantum-leaps.com/products/index.htm>

[3] FreeRTOS.org. *FreeRTOS* [online]. [cit. 2008-28-03]. Dostupné na URL: <http://www.freertos.org/>

[4] FreeScale. *Manuálové stránky k HC08* [online]. [cit. 2008-19-04]. Dostupné na URL: <http://www.freescale.com>