ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA ELEKTROTECHNICKÁ
KATEDRA ŘÍDICÍ TECHNIKY

Diplomová práce

ŘÍDÍCÍ SYSTÉM V REAL-TIME JAVA

Tomáš Procházka
2004

Zadávací formulář

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č.121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne ....
podpis

Anotace

Tato diplomová práce se zabývá popisem vlastností procesoru aJ-80 od firmy aJile Systems a desky JSTAMP, která tento procesor obsahuje. Jedná se o procesor s nativní exekucí bajtového kódu a implementací základních mechanismů jazyka Java přímo v hardwaru procesoru. Popis je proveden jak na úrovni technické (konstrukční) specifikace, tak z hlediska obecného chování procesoru s důrazem na jeho využití v real-time řízení.

V rámci diplomové práce vznikla ukázková aplikace demonstrující možnosti zařízení JSTAMP. Jedná se o laboratorní model motorku, který je připojen k desce JTAMP, jako k řídícímu systému. Chování řídícího systému je ovládáno a monitorováno z nadřazeného počítače PC, kde se také v grafické formě zobrazují průběhy důležitých veličin regulace.

Annotation

This thesis describes features of JSTAMP board and aJ-80 microprocessor from aJile Systems included in JSTAMP. AJ-80 executes Java byte code as it’s own native instruction set and implements Java language basic mechanisms directly into processor’s hardware. Description is made in two parts. The first part is technical specification and aJ-80’s internal circuits introduction. The second part explains processor’s general behavior with focus on it's utilization in real-time control processing.

Showcase application running in JSTAMP and demonstrating its features and limits has been created as a part of this thesis. Application provides control system that drives DC engine model. JSTAMP is connected to PC that hosts client applicaton (CA). The CA sets control system’s parameters and displays major control process variables in graphical form.

Obsah

Zadávací formulář i
Prohlášení ii
Anotace iii
Annotation iii
Obsah iv
Tabulky vii
Obrázky vii
1 Úvod 1
2 Jazyk Java 3
2.1 Historie 3
2.2 Bajtový kód 4
2.3 Verze jazyka Java 5
2.3.1 J2ME 5
2.3.1.1 Konfigurace 5
2.3.1.2 Profily 6
2.3.1.3 CLDC 7
2.3.1.4 MIDP 8
2.4 IDE pro Javu 9
2.5 Rozšíření jazyka Java o RT vlastnosti 9
2.5.1 Systémy reálného času 9
2.5.2 Specifikace RTSJ 10
2.5.2.1 Základní principy 10
2.5.2.2 Vlastnosti RTSJ 11
2.5.2.3 Příklady použití 13
2.6 Garbage Collector 13
2.6.1 GC pro RT systémy 14
3 Mikroprocesor aj-80 16
3.1 Historie 16
3.2 Základní charakteristika 16
3.2.1 Přehled vlastností 17
3.2.2 Tvorba aplikací a vývojové nástroje 18
3.3 Konstrukce 20
3.3.1 Jádro procesoru (JEM2) 21
3.3.1.1 Uživatelsky definované instrukce 21
3.3.2 Vícenásobný JVM 22
3.3.3 Vnitřní a vnější paměť 23
3.3.3.1 Vnitřní paměť 23
3.3.3.2 Vnější rozhraní sběrnice 23
3.3.4 Logika RESET 24
3.3.5 Systémové hodiny 25
3.3.6 Systém přerušení 25
3.4 Vnitřní periferní zařízení 27
3.4.1 Univerzální vstupně/výstupní obvod 27
3.4.1.1 Java třídy a rozhraní 28
3.4.2 Univerzální obvody čítače/časovače 29
3.4.2.1 Vstupně výstupní blok GPTC 31
3.4.2.2 Dělič hodinového taktu 31
3.4.2.3 Čítače/časovače 32
3.4.2.4 Generování pulzně šířkové modulace (PWM) 33
3.4.2.5 Obvod automatického vzorkování 35
3.4.2.6 Java třídy a rozhraní 35
3.4.3 Obvody UART 35
3.4.3.1 Java třídy a rozhraní 36

3.4.4 Univerzální obvod pro sériovou komunikaci 36
3.4.4.1 Java třídy a rozhraní 37
3.5 Správa paměti a vláken 37
3.5.1 Správce vláken 37
3.5.2 Ochrana paměti 38
3.5.3 Garbage Collector 38
3.6 Vývoj aplikací 39
3.6.1 Program JEMBuilder 40
3.6.1.1 Projekt 41
3.6.1.2 Jednotlivá JVM 41
3.6.2 Program Charade 42
3.7 Volitelné periferie 43
3.7.1 Sběrnice CAN 44
3.7.1.1 Základní obvodové řešení CAN 44
4 JSTAMP a vývojové stavebnice 46
4.1 JSTAMP 46
4.1.1 Společnost Systronixs 46
4.1.2 Konstrukce 47
4.2 Vývojové stavebnice 48
4.2.1 JSTAMP DEVELOPMENT BOARD 48
4.2.2 aJile aJ-100EVB 50
4.2.3 Aidonic JPC1001 51
5 Testování vlastností aJ-80 (JStamp) 52
5.1 Doba reakce na vnější událost 52
5.2 Komunikace přes sériový port 53
5.3 Generování PWM 55
6 Popis aplikace 56
6.1 Část v JSTAMP 56
6.1.1 Třída IntDriver 58
6.1.2 Třída PWMDriver 59
6.1.3 Třída RegulatorPID 59
6.1.4 Třída MeasureDataStorage 60
6.1.5 Třída SerialDriver 60
6.1.6 Třída PersistentStorage 60
6.1.7 Třída IdleThread 61
6.2 Klientský program v PC 61
6.2.1 Třída SerialDriver 62
6.2.2 Třída DataStorage 62
6.2.3 Třída Frame 62
6.2.4 Třída Okno 63
6.2.5 Třída PanelAnimGraph 64
7 Závěr 65
8 Použitá literatura 67
9 Přílohy 68
9.1 Příloha A – Slovník použitých zkratek 68
9.2 Příloha B – Instalace a nastavení programů 69
9.2.1 JBuilder 8 – Personal Edition 69
9.2.2 aJile Tools 69
9.2.3 Jem Builder 70
9.2.4 Charade 70
9.3 Příloha C – Struktura přiloženého CD 71
9.4 Příloha D – Příklady zdrojových kódů 71
9.4.1 SpecifikaceRTSJ 71
9.4.1.1 Výpis A 71

9.4.1.2 Výpis B 72
9.4.1.3 Výpis C 72
9.4.1.4 Výpis D 73
9.4.2 Testování vlastností aJ-80 74
9.4.2.1 Doba reakce na vnější událost 74
9.4.2.2 Komunikace přes sériový port 80
9.4.2.3 Generování PWM 82
9.5 Příloha E – Tabulka vývodů zařízení JSTAMP 85

Tabulky

Tabulka 3.1 - Význam jednotlivých přerušení procesoru aJ-80 26
Tabulka 3.2 – Nastavení registrů pro různé druhy spouštění přerušení 28
Tabulka 4.1 - Odlišnosti procesorů aJ-80 a aJ-100 46
Tabulka 4.2 - Velikost paměti zařízení JSTAMP 47
Tabulka 4.3 - Cenová kalkulace sady JSTAMP Dev. Kit 50
Tabulka 5.1 - Výsledky měření časové prodlevy 53

Obrázky

Obrázek 3.1 – Architektura procesoru aJ-80 20
Obrázek 3.2 - Jádro procesoru aj-80 (JEM2) 21
Obrázek 3.3 – Schéma MJM (vícenásobný JVM) 22
Obrázek 3.4 - Vnější rozhraní sběrnice - External Bus Interface (EBI) 23
Obrázek 3.5 – Příklad připojení externích paměťových zařízení k aJ-80 24
Obrázek 3.6 – Schéma zapojení logiky RESET 24
Obrázek 3.7 – Časové průběhy signálů CRSTn a WRSTn 25
Obrázek 3.8 – Schéma zapojení logiky PLL 25
Obrázek 3.9 - Přiřazení obslužných metod přerušením v programu JEMBuilder 27
Obrázek 3.10 - Schéma zapojení jednoho vstupu/výstupu GPIO 28
Obrázek 3.11 - Blokové schéma obvodů GPTC 29
Obrázek 3.12 - Vstupně/výstupní blok GPTC 30
Obrázek 3.13 - Dělič frekvence hodinového signálu 32
Obrázek 3.14 - Vnitřní logika čítače/časovače 33
Obrázek 3.15 - Časový diagram základních operací čítače/časovače 33
Obrázek 3.16 - Obvod generování pulzně šířkové modulace (PWM) 34
Obrázek 3.17 - Časový průběh generování signálu PWM 34
Obrázek 3.18 - Obvod automatického vzorkování 35
Obrázek 3.19 - Blokové schéma obvodu UART 36
Obrázek 3.20 - Sériové rozhraní SPI 37
Obrázek 3.21 - Připojení periferních zařízení přes rozhraní SPI 37
Obrázek 3.22 – Vývoj aplikací pro procesor aJ-80 40
Obrázek 3.23 - Program Charade 43
Obrázek 3.24 - Propojení sběrnice CAN s mikroprocesorem aJ-80 45
Obrázek 4.1 - Zařízení JSTAMP 47
Obrázek 4.2 - Blokové schéma desky JDB 48
Obrázek 4.3 - Popis vývojové desky JDB 50
Obrázek 4.4 - Blokové schéma vývojové desky aJ-100EVB 51
Obrázek 5.1 - Blokové schéma měření časové prodlevy 52
Obrázek 5.2 - Blokové schéma sériové komunikace 54
Obrázek 5.3 - Test generování PWM 55
Obrázek 6.1 - Schéma řídícího systému 56
Obrázek 6.2 - Blokové schéma aplikace v JSTAMP 57
Obrázek 6.3 - Popis vláken a synchronizace 57
Obrázek 6.4 - Měření otáček motorku 58
Obrázek 6.5 - Blokové schéma klientské aplikace 62
Obrázek 6.6 - Okno třídy Frame 63
Obrázek 6.7 - Třída Okno 64

1 Úvod

Zabudovaná (embedded) zařízení mají typicky omezený paměťový prostor a výkon procesoru. Proto byla donedávna programována pomocí nižších programovacích jazyků (jazyk C, Assembler atd.). Bylo to z důvodu snadné optimalizace a nízké paměťové náročnosti takto vznikajících programů. U nižších programovacích jazyků však chybí podpora objektového programování a také celkový programátorský komfort není nijak veliký.

Proto není divu, že se začalo uvažovat o použití některého z vyšších programovacích jazyků. Jedním z nich je jazyk Java. Ten byl dokonce původně vyvíjen pro použití v zabudovaných systémech. Dnes je jazyk Java znám jako programátorsky komfortní, bezpečný a platformně nezávislý jazyk, ale díky nedeterministickému chování a slabému výkonu většiny jeho implementací, zcela nevhodný pro aplikace typu real-time (RT). Na obecné úrovni sice jazyk Java RT požadavky nesplňuje, ale existuje jeho rozšíření, nazvané Real Time Specification for Java (RTSJ), jehož implementací se jazyk Java obohatí o vlastnosti RT.

Dalším logickým krokem ve vývoji zabudovaných systémů byla snaha vytvořit zařízení, které by mohlo být programováno pomocí jazyka Java s rozšířením RTSJ. Proto byl vyvinut mikroprocesor aJ-80 firmy aJile Systems, jehož instrukčním souborem je Java bajtový kód a zároveň implementuje základní principy jazyka Java na hardwarové úrovni.

Přidáním externích paměťových zařízení a dalších obvodů vzniklo zařízení JSTAMP, které může být použito samostatně v průmyslové automatizaci, v přístrojích určených pro domácí použití a dalších zabudovaných zařízeních.

Pro tuto diplomovou práci bylo objednáno a zakoupeno zařízení JSTAMP, za účelem seznámit se s technologií tohoto zařízení a otestovat jeho vlastnosti s důrazem na chování v RT režimu.

Jako praktickou ukázku možností zařízení JSTAMP byl stanoven úkol naprogramovat aplikaci řídícího systému, který řídí připojený laboratorní model motorku. Tato aplikace měla být navíc schopna komunikovat s nadřazeným počítačem PC, pomocí něhož se aplikace ovládá a na němž se vizuálně zobrazují průběhy důležitých veličin regulace.

V následující kapitole této práce se snažím přiblížit historii, vývoj a základní pojmy jazyka Java. Navíc se zde pokouším vysvětlit pojmy, které jsou v dalších částech práce použity. Ve třetí a čtvrté kapitole je uveden popis procesoru aJ-80 a zařízení JSTAMP. Pátá kapitola obsahuje popis testů vlastností zařízení JSTAMP. V šesté kapitole můžeme nalézt popis vzniklé demonstrační aplikace.

Nedílnou součástí této diplomové práce je přiložené CD, na kterém se kromě této práce nacházejí instalační soubory veškerých programů, které byly při realizaci aplikací použity a většina odkazovaných dokumentů

stažených ze sítě Internet. Popis struktury CD je možno nalézt v příloze C. Přílohy obsahují navíc jednoduchý slovník pojmů a zkratek, popis instalace a nastavení použitých programů a výpisy programů odkazovaných v textu práce.

V textu této práce často používám zkratky anglických názvů. Vždy se snažím uvést celý název v anglickém jazyce a jeho volný překlad. Zkratky používám proto, aby se čtenář mohl lépe orientovat ve schématech a doprovodné dokumentaci v anglickém jazyce, kde jsou tyto zkratky používány.

2 Jazyk Java

V této kapitole se pokusím přiblížit základy, historii, vývoj a základní pojmy programovacího jazyka Java.

2.1 Historie

Vývoj programovacích jazyků je řízen dvěma faktory – vývojem v metodice programování a vývojem operačních systémů. Programovací jazyk Java není výjimkou. Je založen na dědictví programovacích jazyků C a C++, navíc obsahuje další vlastnosti odpovídající požadavkům na vývoj moderních aplikací. Vývoj Javy reagoval na růst prostředí „on-line“ a také na rostoucí požadavky distribuovaných počítačových systémů.

Základy a koncepci jazyka definovali v roce 1991 pánové James Gosling, Patrick Naughton, Chris Warth, Ed Frank a Mike Sheridan, všichni zaměstnanci firmy Sun Microsystems. Jejich programovací jazyk byl původně označován jako „Oak“ a v roce 1995 byl přejmenován na „Java“.

Je poněkud překvapující, že původním hybným momentem pro vytvoření jazyka Java nebyl Internet, ale potřeba programovacího jazyka nezávislého na počítačové platformě, jenž by mohl být používán ke tvorbě programového vybavení na různých zákaznických elektronických zařízeních, jako jsou mikrovlnné trouby nebo dálkové ovladače. Jak patrně tušíte, díky této skutečnosti mohou aplikace napsané v jazyce Java běžet na počítačích s různými mikroprocesory a s různými operačními systémy. Většina ostatních programovacích jazyků byla totiž navržena tak, že bylo možné s jejich pomocí překládat aplikace jen pro jisté počítačové platformy. Jako příklad si uveďme programovací jazyk C++.

Pánové Gosling a spol. se rozhodli vytvořit přenositelný programovací jazyk nezávislý na počítačové platformě, který by umožňoval vytvářet aplikace běžící na počítačích s různými mikroprocesory a s různými operačními systémy. Tato snaha vedla k vytvoření programovacího jazyka Java.

Na vývoj jazyka Java měl také vliv jiný faktor, jehož význam rostl právě v době, kdy se pracovalo na detailech překladače Java. Je jím systém www (World Wide Web), který ovlivnil vývoj překladače zcela zásadním způsobem. Bez existence systému www by jazyk Java pravděpodobně zůstal pouze tajemným jazykem pro programování elektronických zařízení. Avšak v důsledku naléhavých potřeb rozvíjejícího se systému www se jazyk Java stal „průkopníkem“ v oblasti moderních programovacích jazyků, protože systém www rovněž vyžadoval maximální přenositelnost programů.

Většina programátorů si jistě ze začátků své kariéry pamatuje, že požadavek přenositelnosti programů existoval již dávno a je tak starý, jako programování samo. Často se tento problém dostával do pozadí zejména v případě, kdy bylo nutné řešit naléhavější problémy. Avšak s rozvojem sítě Internet vyvstal požadavek na přenositelnost programů znovu a postupně se stal kritickým měřítkem při posuzování vlastnosti programovacích systémů.

Konec konců, Internet sestává z ohromného množství počítačů s různými mikroprocesory a různými operačními systémy. To, co kdysi bylo sice znervózňujícím, ale ne příliš významným problémem, přerostlo v kritický aspekt při vývoji síťových aplikací.

V roce 1993 začínalo být autorům programovacího jazyka Java zřejmé, že problém přenositelnosti programů pro různá elektronická zařízení je velmi podobný problému přenositelnosti aplikací napsaných pro síť Internet. Proto se zaměřili na takové rozšíření jazyka, aby jej bylo možné použít i pro Internet.

Zatímco na začátku se tedy jednalo o vytvoření jazyka nezávislého na použitém mikroprocesoru, byl to Internet, který se zasloužil o úspěch jazyka Java v soutěži o přední místo na žebříčku popularity mezi ostatními programovacími jazyky.

2.2 Bajtový kód

Schopnost jazyka Java řešit problém v bezpečnosti a přenositelnosti spočívá v tom, že Java negeneruje spustitelný kód jako překladače ostatních programovacích jazyků. Místo toho generuje tzv. bajtový kód (byte-code). Bajtový kód je vysoce optimalizovaná množina instrukcí navržená tak, aby mohla být spuštěna prostřednictvím run-time systému, jenž se nazývá virtuální stroj Javy (Java Virtual Machine, JVM). To znamená, že ve své standardní formě je Java interpretem bajtového kódu. Tato skutečnost může být dosti překvapivá – vždyť překladače ostatních programovacích jazyků generují efektivní kód přímo spustitelný pod daným operačním systémem. Avšak skutečnost, že programy napsané v jazyce Java jsou realizovány pomocí virtuálního stroje, pomáhá řešit hlavní problémy spojené se zaváděním programů ze sítě Internet. Nyní si vysvětlíme proč.

Program přeložený jazykem Java do bajtového kódu může běžet na mnoha počítačových platformách, protože je nutné pro danou platformu implementovat pouze virtuální stroj. Jakmile pro konkrétní platformu existuje prostředí run-time jazyka Java, může být na této platformě spuštěn kterýkoliv program napsaný v jazyce Java. Ačkoliv se implementace virtuálního stroje Javy pro jednotlivé platformy mohou lišit, jsou schopny interpretovat tentýž bajtový kód. Kdyby Java, tak jako ostatní překladače, generovala spustitelný kód, musela by existovat její zvláštní verze pro každý typ procesoru připojený k Internetu.

Skutečnost, že je každý program napsaný v jazyce Java interpretovaný, zajišťuje také jeho bezpečnost. Spuštění každého programu je řízeno virtuálním strojem Javy a tak je program chráněn před generováním vedlejších efektů mimo systém. Bezpečnost je dále zvýšena jistými omezeními, které jsou implementovány uvnitř jazyka Java.

Obecně platí, že interpretovaný program běží podstatně pomaleji, než program přeložený do spustitelného kódu. V případě jazyka Java však tento rozdíl není tak výrazný.

Ačkoliv byl jazyk Java navržen jako interpretr, nic nebrání tomu, aby byl bajtový kód přímo překládán do nativního kódu konkrétního počítače. Spolu s virtuálním strojem poskytuje společnost Sun také překladač JIT (Just in Time), jenž překládá bajtový kód v reálném čase část

po části a/nebo podle požadavků. Je důležité pochopit, že celý bajtový kód nemůže být přeložen, protože Java provádí celou řadu kontrol, které mohou být realizovány pouze v reálném čase. Proto provádí překladač JIT kompilaci v průběhu vykonávání programu. Překlad „Just in Time“ umožňuje, aby programy v jazyce Java běžely ve srovnání s jinými interpretovanými programy, mnohem rychleji. I v případě, že se při běhu programu aplikuje dynamická kompilace bajtového kódu, zůstává zachována přenositelnost a bezpečnost programu. Je tomu tak proto, že překladač JIT je součástí run-time systému jazyka Java. Nezáleží na tom, zda je program interpretován tradičním způsobem, nebo zda je „urychlován“ překladačem JIT.

2.3 Verze jazyka Java

2.3.1 J2ME

J2ME je verze produktu Java společnosti Sun Microsystems, určená pro trh s elektronickými zařízeními, jakými jsou např. mobilní telefony, pagery, osobní digitální asistenti (PDA), přídavná zařízení a další malá zařízení. Od doby jeho vzniku se k vývoji J2ME připojilo více než 600 společností, např. Palm, Nokia, Motorola a RIM. Směr, jímž se J2ME vyvíjí, však nemá být uchováván v tajnosti za zavřenými dveřmi velkých korporací. Vývoj J2ME je zajištěn projektem zvaným Java Community Process (JCP).

J2ME poskytuje kompletní řešení nejmodernějších síťových aplikací pro malá zařízení. Výrobcům, poskytovatelům a vývojářům těchto zařízení také slibuje možnost vyvíjet nově aplikace pro své zákazníky. Tato možnost však neznamená, že by byly obětovány základní vlastnosti Javy, jejichž význam v současnosti stále roste, jmenovitě kompatibilita různých prostředí a zabezpečení.

Verze J2ME definuje skupinu knihoven a API, které lze spustit na každém virtuálním stroji; říká se jim konfigurace a profily.

2.3.1.1 Konfigurace

Mobilní telefony, pagery, organizátory a další malá zařízení se liší svojí formou, funkcemi a vlastnostmi. Často však používají podobné procesory a mají podobné množství paměti. Proto autoři J2ME vytvořili konfigurace. Konfiguracemi se definuje horizontální členění produktů založené na dostupném množství paměti a výkonu procesoru jednotlivých zařízení. Jakmile má tyto informace k dispozici, konfigurace zjistí následující údaje:

- podporované rysy programovacího jazyka Java
- podporované rysy virtuálního stroje Javy
- podporované základní javovské knihovny a API

V současnosti existují pro J2ME dvě standardní konfigurace: CLDC (Connected Limited Device Configuration) a CDC (Connected Device Configuration).

CDC

Konfigurace CDC je určena pro výkonná zařízení, která jsou občas připojena k síti. Patří sem přídavná zařízení, televize po Internetu, domácí spotřebiče a navigační systémy pro vozidla. CDC obsahuje plnou verzi virtuálního stroje Javy podobného tomu, který se dnes používá u J2SE. Rozdíl spočívá v paměti příslušného zařízení a v zobrazovací schopnosti.

CLDC

Druhý typ konfigurace, CLDC, je pro J2ME obvyklejší. Tato konfigurace udává mnohem menší požadavky na zabudovaná zařízení než CDC. Konfigurace CLDC byla poprvé vydána v říjnu 1999 se záměrem vytvořit „nejnižšího společného jmenovatele“ v prostředí Java pro zabudovaná zařízení, dvousměrné pagery, osobní digitální asistenty (PDA) a osobní organizéry.

2.3.1.2 Profily

J2ME umožňuje definovat javovská prostředí pro různé produkty stejné úrovně tím, že zavádí profily. Profil je vlastně sada programových rozhraní (API) tvořících nadstavbu konfigurace. Profil nabízí programu přístup k vlastnostem specifickým pro dané zařízení. Následuje několik příkladů profilů, které jsou v současnosti k dispozici pro J2ME.

MIDP

MIDP je navržen pro práci s CLDC a poskytuje sadu programových rozhraní použitelných v mobilních zařízeních, jako jsou mobilní telefony a obousměrné pagery. MIDP obsahuje třídy pro uživatelská rozhraní, trvalé ukládání a práci v síti. Dále obsahuje standardizované pracovní prostřední, které umožňuje „nahrávat“ do koncového zařízení nové aplikace. Malým aplikacím, které běží pod MIDP, se říká Midlety.

Profil PDA

Profil PDA je založen na CLDC a poskytuje API pro uživatelské rozhraní (mají být podskupinou AWT) a API pro ukládání dat v příručních zařízeních. V době vzniku této práce se na profilu PDA ještě pracovalo a nebyly dostupné žádné popisy jeho implementace.

Základní profil

Základní profil rozšiřuje programové rozhraní, které poskytuje CDC, ale nedodává žádné API pro uživatelské rozhraní. Jak naznačuje název „Základní“ (Foundation), tento profil má sloužit jako základní pro další profily, například pro Osobní profil a RMI profil.

Osobní profil

Osobní profil rozšiřuje možnosti Základního profilu o grafické rozhraní (GUI), na němž lze spustit applety pro Java Web. Jelikož se

PersonalJava mění v Osobní profil, bude zpětně kompatibilní s aplikacemi pro PersonalJava 1.1 a1.2.

RMI profil

Vzdálené volání metod (Remote Method Invocation - RMI) je mechanismus jazyka Java, umožňující volání metod vzdálených objektů v distribuovaných aplikacích. RMI profil rozšiřuje možnosti Základního profilu o RMI pro daná zařízení. Protože navazuje na Základní profil, je RMI profil určen pro CDC/Základní profil, a ne pro CLDC/MIDP. RMI profil je kompatibilní s J2SE RMI API 1.2x a vyššími.

2.3.1.3 CLDC

Podle specifikace mají zařízení, na nichž se uplatňuje CLDC, následující vlastnosti:

160 kB až 512 kB celkové paměti

Zařízení s CLDC by mělo mít minimálně 128 kB paměti ROM pro virtuální stroj Javy a pro CLDC knihovny a minimálně 32 kB paměti RAM využitelné pro práci virtuálního stroje, to vše nezávisle na jiných aplikacích.

16bitový nebo 32bitový procesor s minimální taktovací frekvencí 25 MHz

Tyto typy procesorů jsou v současnosti pro přenosná zařízení typická.

Připojitelnost k některému druhu sítě

V případě CLDC se často jedná o dvousměrné dvoudrátové připojení s omezenou šířkou pásma.

Nízká spotřeba energie

Zařízení s CLDC často pracují na energii z baterií. Proto mají velmi nízkou spotřebu energie.

Tomuto popisu vyhovují zařízení všech tvarů a velikostí. Okamžitě se nám vybaví mobilní telefony a pagery, ale stejně dobře bychom mohli Javu instalovat i na čtečky čárového kódu, video a audio vybavení, navigační systémy a další bezdrátová zařízení, která se ještě objeví. V podstatě lze očekávat, že společně se změnami na trhu těchto zařízení se budou měnit i specifikace pro CLDC.

S danými, výše uvedenými omezeními, poskytuje v současnosti CLDC zařízením tyto funkce:

- soubor rysů Javy a virtuálního stroje
- soubor základních knihoven Javy (java.lang a java.util)
- základní vstup/výstup (java.io)
- základní podporu sítí (javax.microedition.io)
- zabezpečení

CLDC nezasahuje do průběhu aplikací, do uživatelských rozhraní, ani do správy událostí, či do interakce mezi uživatelem a aplikací. Tyto rysy

totiž spadají do sféry profilů, jako je MIDP, které jsou uváděny na vrcholu CLDC a přispívají k jeho funkčnosti.

2.3.1.4 MIDP

Standard MIDP definuje mobilní informační zařízení jako zařízení s následujícími minimálními vlastnostmi:

Displej

Velikost obrazovky minimálně 96 x 54 pixelů s minimální hloubkou barev 1 bit.

Vstupní zařízení

Klávesnice pro jednu nebo dvě ruce, případně dotyková obrazovka.

Paměť

32 kB paměti RAM pro práci Javy; 128 kB paměti ROM pro komponenty MIDP a 8 kB stálé paměti pro dlouhodobé ukládání dat z aplikací.

Práce v síti

Přerušované obousměrné spojení, obvykle bezdrátové s omezenou šířkou pásma.

Protože je MIDP vrcholem CLDC, zaměřuje se na následující oblasti, které jsou v CLDC neřešené:

Správa průběhu aplikací

MIDP obsahuje balík `javax.microedition.midlet` s třídami a metodami pro zahájení, přerušení a vymazání aplikací z hostitelského prostředí.

Uživatelské rozhraní a události

MIDP také poskytuje balíky `javax.microedition.lcdui`, které zahrnují třídy a rozhraní k tvorbě komponent grafického uživatelského rozhraní v aplikacích.

Připojitelnost k síti

MIDP rozšiřuje rozhraní `ContentConnection` obecného připojovacího systému (Generic Connection Framework) tím, že poskytuje rozhraní `HttpConnection` a současně instalační sadu pro http protokol.

Ukládání dat v zařízení

MIDP také poskytuje balík `javax.microedition.rms`, který zavádí databázový záznamový systém. Ten umožňuje aplikacím ukládat data v zařízení.

2.4 IDE pro Javu

Nástroje IDE (Integrated Development Enviroment) jsou GUI aplikace, sdružující v sobě několik programů. Je to editor kódu, kompilátor, debugger, systém nápověd a další vývojářské nástroje. Nástroje IDE svou koncepcí „vše v jednom“ usnadňují vývojářskou práci.

Dále uvádím některé z klíčových vlastností prostřední IDE. Jsou to:

- sdružování do projektů – nastavíme cesty ke knihovnám, souborům nápovědy, verzi kompilátoru a další vlastnosti projektu
- automatické zálohování
- možnost vyvolání nápovědy přímo k vybranému objektu
- automatické formátování, vizuální zvýraznění syntaxe
- zobrazení chyb při kompilaci – kliknutím na chybu se přeneseme přímo do chybné pasáže kódu
- kódový asistent – nabízí například třídy z balíků nebo ukazuje typy parametrů metod, počet parametrů, atd.

Některé nejznámější IDE nástroje pro jazyk Java:

- JBuilder (Borland)
- NetBeans (Sun)
- Eclipse (IBM)
- WebLogic Workshop (BEA)
- JDeveloper (Oracle)
- Visual J++ (Microsoft)
- Intelli J (JetBrains)
- CodeGuide (OmniCore)

2.5 Rozšíření jazyka Java o RT vlastnosti

I přes své nesporné výhody, jakými jsou jednoduchý objektový model, silná typová kontrola a bezpečnost, integrovaná podpora více vláknových aplikací a platformní nezávislost, byla Java pro aplikace, které vyžadují vlastnosti systémů reálného času, zcela nevhodná. Především z důvodu svého nedeterministického chování a slabého výkonu většiny jejích implementací.

2.5.1 Systémy reálného času

Systém reálného času (Real Time – RT – systém) je systém, u něhož požadujeme, aby kromě výsledků, které z něho vzejdou, systém zaručoval, že výsledky získáme za určitou, přesně definovanou dobu. Přívlastek „real-time“ zde nutně neznamená rychlý, ale signalizuje především to, že časové charakteristiky reakcí systému jsou deterministické. To znamená, že měřítkem správné činnosti není průměrná doba reakcí, ale nejdelší doba reakce, která může nastat. Někdy se RT systémy dále dělí na hard (tvrdé) a

soft (měkké). Hard RT systém je takový, v němž je požadavek na dobu reakce absolutní. U soft RT systému se tolerují určité malé odchylky.

Kromě výše uvedeného časového determinismu jsou na RT systémy kladeny další požadavky. Jsou to především:

- preemptivní plánovač s velkým počtem (typicky 64 až 256) nastavitelných priorit vláken
- prediktivní mechanismus synchronizace vláken
- systém dědičné priority
- rychlé hodiny a časovače
- dostatečně známé chování systému

2.5.2 Specifikace RTSJ

Poté co byl rozpoznán potenciál Javy pro nasazení v RT systémech, začala skupina expertů na Javu a RT systémy vyvíjet požadavky na jazyk Java s vlastnostmi RT. Tato aktivita byla finančně podporována americkým úřadem National Institute of Standards and Technology (NIST). Datem začátku prací byl červen 1998. Jednalo se pouze o obecné požadavky a základní principy, které se mají stát základem specifikace Real-Time Specification for Java (RTSJ).

V období před vývojem RTSJ byly veškeré specifikace jazyka Java vyvíjeny firmou Sun Microsystems a jejími licencovanými partnery. Jako odpověď na sílící snahy nezávislých expertních skupin o možnost participace na vývoji nových specifikací, ustanovila firma Sun v prosinci roku 1998 souhrn pravidel pro vyvíjení nových specifikací s názvem Java Community Process. V rámci těchto pravidel může kdokoliv podat požadavek na novou specifikaci - Java Specification Request (JSR). Pokud je JSR přijat firmou Sun jako relevantní, jsou osloveny expertní skupiny, aby připravily návrhy na řešení problému. Mezi návrhy pak firma Sun vybere (podle svého soudu) nejlepší návrh a jeho autor (autoři) jsou odpovědní za sestavení vývojářského týmu, který je zodpovědný za vývoj a testování vznikající specifikace.

RTSJ byl přiřazen požadavek JSR s označením JSR-000001. V březnu 1999 byla založena skupina Real Time for Java Expert Group (RTJEG).

2.5.2.1 Základní principy

Prvním úkolem skupiny RTJEG bylo definovat základní principy, kterými se vznikající RTSJ měla řídit. RTSJ má splňovat následující :

- možnost včlenění do stávajících prostředí jazyka Java - RTSJ nemá obsahovat vlastnosti, které by omezily její použití ve stávajících Java prostředích
- zpětnou kompatibilitu, RTSJ nesmí omezovat existující, „čistě“ napsané non-RT Java programy při běhu na RTSJ platformě
- platformní nezávislost - důležité je, aby se RTSJ zaměřila na řešení složitosti dosažení platformní nezávislosti u RT programů

- ponecháni prostoru pro budoucí rozšiřování – RTSJ musí implementovat RT principy známé v současné době, ale zároveň musí ponechat dostatečný prostor pro další rozšiřování vlastností v budoucnu
- chování systému musí být předvídatelné – RTSJ má mít, jako jedno z nejdůležitějších, deterministické chování. V každém okamžiku provádění programu musí mít uživatel přesnou představu o tom, jakým způsobem se bude program v příštím kroku provádět/chovat
- žádné rozšiřování syntaxe jazyka – RTSJ nesmí přidávat nová klíčová slova do jazyka Java, ani nesmí obsahovat jiná syntaktická rozšíření

2.5.2.2 Vlastnosti RTSJ

Plánování a určování vlastností vlákna

RTSJ umožňuje programové přiřazení parametrů plánovače úloh vlastních pro daný RT systém, navíc poskytuje metody pro vytváření, správu a ukončování uživatelských Java RT vláken. Java RT vlákna jsou konstruována vytvořením instancí třídy javax.realtime.RealtimeThread, která rozšiřuje třídu java.lang.Thread.

Základním plánovacím mechanismem (pro plánování činnosti vláken) je v RTSJ prioritní, preemptivní plánování vláken – 28 různých hodnot priorit. V jednotlivých hladinách priorit se plánovač rozhoduje podle algoritmu FIFO – první vlákno, které bylo registrováno plánovačem jako připravené k běhu, je mezi vlákny se stejnou prioritou spuštěno jako první, pokud v současné chvíli není připraveno k běhu vlákno s vyšší prioritou.

Správa paměti

Automatická správa paměti (Garbage Collector – GC viz kapitola 2.6) je specifická vlastnost prostředí Java, proto se RTSJ snaží izolovat programátory od možnosti spravovat paměť co možná nejvíce. Při tvorbě RTSJ se ale ukázalo, že mezi stávajícími GC algoritmy neexistuje žádný „vše řešící“, který by obstál ve všech nárocích, které vyžadují RT systémy. Z toho důvodu zavedla RTSJ specifikaci paměťové alokace a rekultivace, která není ovládána žádným specifickým algoritmem GC. Naopak ponechává přesné ovládání vlastností GC, jakou jsou doba běhu, četnost preempce, atd., v rukou programu (programátora).

Specifikace RTSJ definuje nové typy oblastí paměti jako alternativu k běžné haldě (Heap), spravované GC. Jsou to oblasti paměti s názvy ImmortalMemory („nesmrtelná“ paměť) a ScopedMemory (paměť s omezenou dobou života).

Ani jedna z těchto oblastí paměti není spravována automatickou správou paměti (GC). Oblast ImmortalMemory je společná pro všechna vlákna aplikace. Objekty v této oblasti vytvořené mají stejnou délku života jako celá aplikace. Jsou dostupné po celou dobu běhu aplikace.

Oblast ScopedMemory má omezenou dobu života. Po vytvoření nového vlákna s přiřazením oblasti ScopedMemory nebo po přepnutí do vlákna se stejným přiřazením, jsou nově vznikající dynamické objekty alokovány do oblasti ScopedMemory. Po ukončení činnosti posledního z vláken, které pracovalo s oblastí ScopedMemory, jsou všechny objekty, které se v této oblasti paměti nacházejí uvolněny.

Synchronizace a sdílení prostředí

Skupina RTJEG rozhodla, že specifikace, která bude vyžadovat nejméně změn a zároveň zajistí bezpečnou synchronizaci v RT prostředí, je doporučení, aby implementace stávajícího klíčového slova jazyka Java „synchronized“ obsahovala jeden nebo více algoritmů, které by předcházely inverzi priorit mezi Java RT vlákny. Inverzí priorit rozumíme situaci, která nastane, požaduje-li vlákno s vyšší prioritou přístup ke zdrojům programu, které jsou v té chvíli v exkluzivním držení vlákna s nižší prioritou. V tomto případě dojde k přepnutí (preempci) do vlákna s vyšší prioritou, které však nemůže pokračovat v běhu kvůli zablokovanému zdroji. Jediným řešením této situace je umožnit vláknu, které programové zdroje drží, co nejrychleji dokončit činnost a umožnit tak i jiným vláknům pokračovat v činnosti.

Zpracování asynchronních událostí

Specifikace RTSJ zobecňuje pojem zpracovávání asynchronních událostí v prostředí Java. AsyncEvent je objekt, který je programově svázán s objektem AsyncEventHandler. Třída AsyncEventHandler implementuje rozhraní Runnable, překrytá metoda run() je spuštěna, pokud byla aktivována událost AsyncEvent. Provádění kódu ovladače (AsyncEventHandler) je na stejném principu jako provádění Java RT vlákna.

Asynchronní přenos řízení

Specifikace RTSJ popisuje chování metod, ve kterých je definována výjimka AsynchonousInterruptedException (AIE). Tato výjimka je vyvolána virtuálním strojem JVM, pokud byla zavolána metoda interrupt() příslušným vláknem. Třída Timed rozšiřuje třídu AsynchonousInterruptedException a pokud byla při vytváření zavolána s parametrem odpovídajícím času, bude AIE poslána příslušnému vláknu po vypršení definovaného času.

Ukončení vlákna

Přestože specifikace RTSJ nedefinuje mechanismus, kterým by se vlákno přímo explicitně ukončovalo, dává návod, jak toho efektivně dosáhnout. Je to pomocí obsluhy asynchronních událostí a asynchronního přenosu řízení. Externí událost je v JVM svázána s objektem AsyncEvent, který, pokud je vyvolán, spustí k němu přiřazený ovladač AsyncEventHandler. Ovladač ve svém kódu spustí metodu Thread.interrupt() pro cílové vlákno. To má za následek, že metoda run() daného vlákna je odblokována a dokončena.

Fyzický přístup k paměti

Přímý přístup k paměti není vyžadován pro dosažení RT vlastností prostředí Java, ale je to užitečná vlastnost, která rozšiřuje možnosti jazyka. Je to ovšem také určitá ztráta bezpečnosti, protože nesprávným použitím přímého přístupu do paměti se můžeme (i nevědomky) dopustit nepřijatelných operací, které mohou mít za následek havárii systému bez jakéhokoliv varování.

Pro fyzický přístup do paměti definuje RTSJ třídy, které dovolují práci na úrovni bajtů fyzické paměti. Také definuje třídy, pomocí nichž je možné vytvářet objekty Java přímo ve specifických oblastech adresového prostoru paměti.

Kompletní specifikaci RTSJ je možno najít v [14].

2.5.2.3 Příklady použití

V příloze D, kapitola Specifikace RTSJ, jsou uvedeny výpisy programů, které jsou příklady použití RTSJ v praxi.

Výpis A popisuje nastavení a využívání asynchronních událostí. V tomto konkrétním případě je událost svázána s hardwarovým přerušením. Událost je zpracována ovladačem (Handler). Ovladač si lze představit jako vlákno, které čeká na notifikaci. Notifikace je vyvolána asynchronní událostí, která je přiřazena k příslušnému HW přerušení.

Výpis B demonstruje použití oblasti paměti s názvem ImmortalMemory, jako možnou alternativu používání běžné oblasti paměti (heap) spravované GC. Ačkoliv je práce GC preemptivní (je možno ji přerušit), může být čas spotřebovaný na přepnutí z GC do vlákna, které je kritické a je nutné, aby provedlo neodkladné činnosti, příliš dlouhý a tak pro dodržení RT vlastností celé aplikace nepoužitelný. Oblasti paměti Scoped a Immortal nevyžadují ke své správě činnost GC. A tak jsou volně přístupná v kontextu vlákna NoHeapRealtimeThread, které pracuje s vyšší prioritou než GC a tak nemůže být automatickým správcem paměti ve své činnosti přerušeno. Výpis B také ilustruje použití parametrů pro dosažení periodického spouštění naplánovaných RT vláken.

Výpis C ukazuje použití asynchronního přenosu řízení z operace, u které může dojít k překročení času pro ni vymezeného.

Výpis D je příkladem využívání třídy RawMemoryAccess pro psaní kódu na úrovni fyzických zařízení v jazyce Java. V tomto konkrétním případě se jedná o rozhraní k programovatelnému čítači/časovači Intel 8253.

2.6 Garbage Collector

Garbage Collector – GC - (doslova „popelář“ nebo „vozidlo pro svoz odpadů“) byl vynalezen v roce 1958, kdy ho John McCarthy použil jako část implementace programovacího jazyka Lisp. GC je v současné době využíván v programovacích jazycích jako jsou Java, Pearl, Modula-3, Prolog, ML nebo Smalltalk.

V kontextu jazyka Java je Garbage Collector program, který je vykonáván v Java runtime prostředí. Automaticky detekuje a uvolňuje volnou paměť, která již nebude dále využívána uživatelskou aplikací. Oproti jiným programovacím jazykům (jako např. jazyk C), které nemají automatickou správu paměti a ve kterých existuje k příkazu vytvoření nového dynamického objektu (např. příkaz new) komplementární příkaz pro uvolnění daného dynamického objektu z paměti (např. příkaz delete), je v

jazyce Java pouze příkaz pro vytvoření nového dynamického objektu. O uvolnění objektů z paměti, které již uživatelský program nebude využívat, se postará automatická správa paměti – GC. Uživatel tedy nemusí sám explicitně odstraňovat dynamické objekty z paměti počítače. Při vytváření nových dynamických objektů by měl ale uživatel postupovat efektivně (viz [3]). V opačném případě hrozí vytváření nadbytečných objektů a tím komplikovaní činnosti GC. Nejjednodušší implementace GC prostě pozastaví provádění všech uživatelských vláken (pokud začne být nedostatek volné paměti), vyhledá všechny dynamické objekty, které jsou uživatelskou aplikací nadále nevyužívané a uvolní je z paměti. GC prověřuje všechny reference na dynamické objekty. Každý objekt, na který existuje reference, je označen. Po dokončení této „značkovací“ procedury, jsou odstraněny objekty, které nebyly v předchozí činnosti GC označeny. Tento jednoduchý druh GC se nazývá „mark and sweep“ („označ a zameť“).

Předchozí příklad implementace GC je sice jednoduchý a spolehlivý, ale v systémech RT nepoužitelný. Protože čas, který je potřeba k „úklidu“ paměti, je úměrný počtu objektů v tu dobu alokovaných. I pro běžné systémy (bez vlastností RT) se proto vyvíjejí nové implementace GC. To, jak kvalitním GC dané systémy disponují, je jedním z důležitých hledisek při hodnocení kvality systémů s exekucí Java bajtového kódu (JVM, JEM, atd.)

2.6.1 GC pro RT systémy

Garbage Collectory pro RT systémy by měly splňovat následující základní podmínky. Měly by :

- správně identifikovat a odstraňovat všechny nadále nepoužívané objekty aby se předešlo nedostatku volné paměti
- být preemptivní
- být inkrementální
- provádět defragmentaci dostupné paměti
- uvolňovat dostupnou paměť s podobnou četností, jako uživatelská aplikace vytváří a přestává používat dynamické objekty

Preemptivní GC je forma GC, která je většinou implementována jako systémové vlákno s určitou prioritou. Provádění tohoto vlákna je možno přerušit, pokud se v systému vyskytne vlákno s vyšší prioritou, připravené k běhu. Obecněji lze říci, že preemptivní GC je možno v jeho činnosti kdykoliv přerušit a po provedení neodkladných částí uživatelské aplikace mu opět navrátit řízení.

Inkrementální GC je takový GC, který po navrácení řízení, které mu bylo odebráno ve prospěch uživatelské aplikace, pokračuje ve své činnosti přesně tam, kde byl přerušen. GC, který po navrácení řízení začíná pracovat „od nuly“, je preemptivní, ale nikoliv inkrementální.

Defragmentace paměti znamená konsolidaci malých úseků paměti do jednoho. GC tedy v tomto případě přesouvá objekty v rámci paměti tak, aby jejich paměťová reprezentace tvořila co nejdelší souvislé úseky a tím se předcházelo neefektivnímu využití paměti při jejím rozdělení do množství malých nesouvislých částí.

Nejnovější trend v oblasti implementací Garbage Collectorů je tzv. generační GC. Využívá hypotézu, že u „mladých“ objektů (objekty, od jejichž alokace v paměti uplynulo méně času) je větší pravděpodobnost, že přestanou být používané, než je tomu u „starých“ objektů. Objekty jsou řazeny do tzv. generací. Nové objekty (právě vytvořené) jsou zařazeny do nejmladší generace. Postupem času , pokud „přežijí“, jsou objekty povyšovány do starších generací. Objekty zařazené ve starších generacích se prověřují (z hlediska jejich další použitelnosti) méně častěji než objekty v generacích mladších. Tím se dosahuje větší efektivity GC a šetří se tím potřebný strojový čas procesoru.

3 Mikroprocesor aj-80

3.1 Historie

Ve firmě Rockwell Collins (americká firma specializující se na vývoj leteckých systémů) existovala skupina Advanced Technology Center (ATC). Tato skupina dostala za úkol vyvinout vkládaný (embedded) systém, založený na průmyslových standardech. Později bylo zadání blíže specifikováno – výstupem projektu měl být procesor s nízkou spotřebou energie, na kterém bude možno spouštět aplikace napsané v některém z vyšších programovacích jazyků. Tímto jazykem byla zvolena Java. Tak vznikl první procesor JEM1 (Java direct-Execution Microprocessor), jehož instrukčním souborem byl Java bajtový kód. Ačkoliv bylo zadání projektu splněno, pokračovala skupina ATC v dalším vývoji a zdokonalování této právě vznikající technologie. Aby mohl být JEM použit v leteckých systémech, musel implementovat bezpečné vzájemné oddělení kritických aplikací. To znamená, že cokoli se děje s jednou aplikací, nesmí mít vliv na správných chod aplikací ostatních. Z běžného života známe případy, kdy na počítačích typu PC způsobí jedna aplikace havárii celého systému. Je jasné, že poslední věcí, kterou piloti potřebují, aby se jim při práci stala, je ekvivalent „modré obrazovky smrti“ (nám všem notoricky známé z prostředí MS Windows). Proto byl v dalším vývoji procesoru JEM kladen velký důraz na bezpečnost a stabilitu na něm provozovaných aplikací. Druhá generace JEM – JEM2 přišla s myšlenkou několika virtuálních strojů (VM) na jednom čipu, jejichž činnost se dělí do časových slotů (time slaced). Tedy v jednom okamžiku je aktivní právě jeden VM. Jednotlivé VM se v činnosti (v aktivním stavu) pravidelně periodicky střídají.

Prototyp procesoru JEM2 se stal základem komerčního produktu firmy aJile Systems – procesoru aJ-80.

3.2 Základní charakteristika

Procesor aJ-80 firmy aJile Systems je jednočipový mikrokontrolér. Přímo (nativně) provádí Java bajtový kód, obsluhuje plánovač Java RT vláken a provádí je. Dále obsahuje rozšíření Java bajtového kódu o instrukce „šité na míru“ pro činnost zabudovaných (embedded) systémů. Díky implementaci nativní exekuce bajtového kódu odpadá nutnost jakékoliv softwarové vrstvy, která je používána ve standardních implementacích JVM. Myslím tím softwarové interprety Java bajtového kódu a JIT (just in time) překladače. Tím se provádění programu stává efektivnější a snižují se také nároky na dostupnou paměť, která není spotřebována činností nutnou pro převod z bajtového kódu do nativního instrukčního souboru příslušného procesoru.

Navíc fakt, že procesor umožňuje správu a plánování vlákna (mechanismy jako wait, yield, notify, monitor enter/exit), eliminuje potřebu doplnění tradičního RTOS (RT operační systém). Implementace plánovače a

správce vláken je pomocí rozšířeného bajtového kódu, což má za následek nízké zatížení systému vlastním plánováním a správou vláken. Také to způsobuje velmi rychlé přepínání mezi vlákny, kdy jsou (podle výrobce) dosahovány časy pod 1 μs. V kapitole 5 jsou uvedeny zkušenosti s dosahovanou rychlostí při přepínání mezi vlákny v praxi.

Všechny jmenované vlastnosti, společně s podporou dvou „nezávislých“ JVM na jednom čipu, jsou dobrým základem pro provozování výkonných, bezpečných a robustních Java aplikací. Procesor aJ-80 je určen pro správu a řízení sítí „inteligentních“ senzorů (smart sensors) v průmyslové automatizaci, v přístrojích určených pro domácí použití i v dalších uživatelských aplikacích.

3.2.1 Přehled vlastností

JEM2-32bitový Java procesor
- nativní exekuce Java bajtového kódu
- rozšíření bajtového kódu o specializované instrukce pro práci s vstupně/výstupními zařízeními a pro správu a plánování vláken
- aritmetické operace s plovoucí řádovou čárkou – podle normy (IEEE-754)
- možnost doplnění uživatelem definovaných rozšíření bajtového kódu

Nativní podpora činnosti Java vláken
- podpora hard RT aplikací a více vláknových aplikací umístěná přímo v hardwaru procesoru
- atomické (nedělitelné) operace s vlákny – používají se standardní principy synchronizace v jazyce Java
- vlákna jsou plánována do front, jejichž podpora je přímo v hardwaru procesoru a jejichž chování je deterministické
- přímá podpora specifikace RTSJ
- přepnutí mezi vlákny za čas menší než 1 μs
- není potřeba žádné implementace RTOS

Více virtuálních strojů (JVM) na jednom čipu
- podpora dvou „nezávislých“ JVM
- oddělení činnosti obou JVM na úrovni času (time-slots) a na úrovni přístupu do paměti

Interní paměť typu RAM
- celková velikost 48kB, paměť typu RAM
- část 32kB je určena pro data
- zbytek 16kB je určen pro mikrokód

Správa externích pamětí
- datová sběrnice 8 bitů
- 6 krát bit pro ovládání výběru konkrétní připojené paměti (chip select)
- podpora pamětí typu ROM, Flash, SRAM

Dvakrát obvod pro sériovou komunikaci (UART)
- 128 bajtový zásobník typu FIFO pro vysílání i příjem
- podpora protokolu fyzické vrstvy standardu IrDA

Třikrát 16 bitový čítač/časovač
- standardní operace čítače/časovače
- přímá vazba na přerušovací podsystém
- přímé generování PWM (Pulse Width Modulation)
- automatické vzorkování vstupního signálu (viz kapitola 3.4.2.5)

Univerzální sériové rozhraní SPI (Seriál Peripheral Interface)
- operace master/slave
- programovatelná délka rámce

Univerzální vstupně/výstupní piny
- 22 vstupně výstupních pinů
- programovatelné na úrovni jednotlivých pinů (bitů)
- propojení s přerušovacím podsystémem

Řízení spotřeby energie
- zastavení činnosti procesoru pokud je bez požadavků na vykonání – pro uživatele transparentní
- jednotlivé periferie mohou být samostatně deaktivovány
- generátor hodinových pulzů procesoru může být pozastaven – reaktivace externí událostí přes v/v pin

Standardizované testovací rozhraní IEEE 1149.1(JTAG)
- testování na úrovni bajtového kódu
- přímý přístup k registrům procesoru

Nízké nároky na spotřebu energie
- spotřeba méně než 1mW/Mhz
- vyrobeno technologií CMOS 0,25 μm, napájení 3,3 V

3.2.2 Tvorba aplikací a vývojové nástroje

Softwarové JRE, jak jej známe z většiny aplikací prostředí Java, je v procesoru aJ-80 nahrazeno JRE na hardwarové úrovni. Je postaveno na standardu firmy Sun-Java 2 Micro Edition (J2ME), profil Connected Limited Device Configuration (CLDC). S procesorem jsou standardně dodávány programy pro vývoj aplikací. Je to program, pomocí něhož se staticky sestavují jednotlivé soubory typu class (linker) a který zajistí nezahrnutí nepotřebných (nepoužitých) tříd do výsledného sestaveného souboru (optimalizer). Dále je to program, který přímo komunikuje s procesorem a pomocí něhož můžeme přenášet aplikace z počítače – PC, na němž vyvíjíme, do paměti procesoru. Tento program také zajišťuje krokování běžícího programu na úrovni jednotlivých instrukcí bajtového kódu. Společně

s vývojovou stavebnicí (kapitola 4) tedy poskytuje tato dodávka kompletní prostředek pro vývoj aplikací pro procesor aJ-80.

Při použití některého z komerčních IDE nástrojů je možné vyvíjet samostatné RT Java aplikace napsané celé v jazyce Java, při využití veškerého programátorského komfortu, který moderní IDE nástroje poskytují. Vzniklé aplikace se mohou, z hlediska výkonnosti a paměťové náročnosti, přiblížit aplikacím napsaným v jazycích C nebo Assembler.

Na procesorech aJ-80 můžeme provozovat aplikace dvou typů. Aplikace s vlastnostmi RT a aplikace využívající dynamické Java prostředí. U RT aplikací předpokládáme, že *je* většinou používají statická data – typicky se jedná o obsluhu přerušení od vnější události. Program se inicializuje a potom, pokud je spuštěn, na základě statických vstupních dat, provede svoji úlohu. U aplikací těchto typů není většinou potřeba použití automatické správy paměti (GC). Dynamické aplikace využívají vytváření (a rušení) dynamických objektů. Procesor aJ-80 podporuje standard CLDC Mobile Information Device Profile (MIDP). Aplikace tohoto standardu, tzv. MIDlety, mohou být přímo spouštěny jako programy v aJ-80. Aplikace mohou být kombinovány a vykonávat se „současně“ – jsou odděleny pracovní oblasti paměti a Java Application Manager (JAM) přiděluje jednotlivým aplikacím aktivitu procesoru.

Druhou možností, jak mohou dynamické a RT aplikace koexistovat, je jejich spouštění v oddělených JVM.

Následuje přehled podpůrných komponent pro vývoj aplikací pro procesor aJ-80:

Aplikace pro sestavování aplikačních celků

- aplikace typu GUI – název JEM Builder
- vstupem jsou standardní soubory typu `class`
- staticky vyhodnocuje vztahy mezi jednotlivými soubory `class` a eliminuje nepoužité metody a atributy
- provádí optimalizaci na úrovni bajtového kódu
- nahrazuje volání metod rozšířeným bajtovým kódem, pokud pro metodu existuje
- přiřazuje přerušením obslužné rutiny
- konfiguruje jednotlivé JVM a definuje rozložení paměti

Prostředí Java Runtime

- založeno na standardu J2ME CLDC
- obsahuje knihovny pro práci v síti a pro perzistentní (stálé) uložení dat
- komponenty JAM a GC
- ovladače pro integrované periferie

Nástroje pro testování a ovládání aplikací

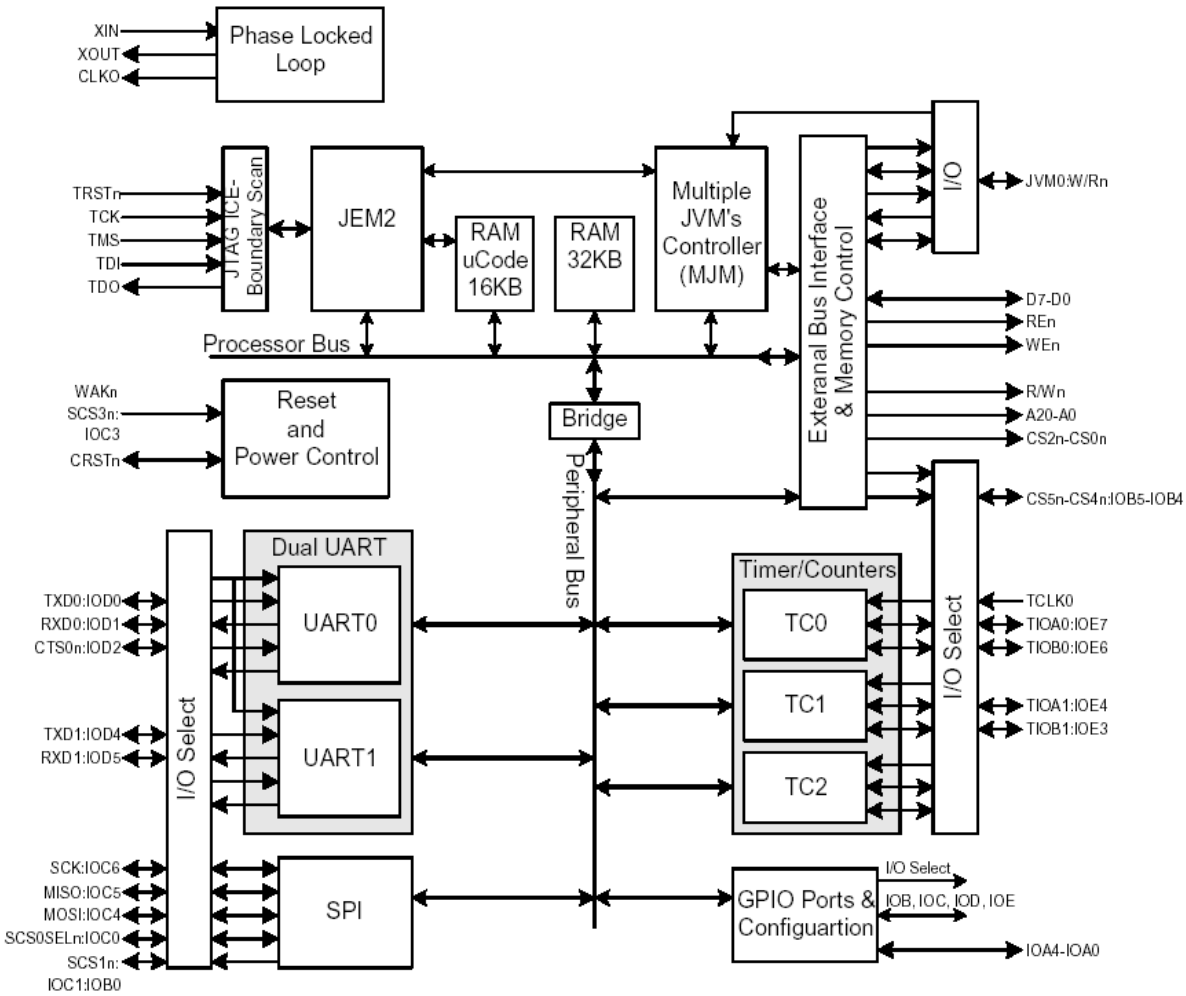
- aplikace typu GUI, název Charade
- komunikace s procesorem přes standardizované rozhraní IEEE1149 (JTAG)
- krokování aplikace na úrovni bajtového kódu

3.3 Konstrukce

Procesor aJ-80 je postaven na architektuře SOC (System-on-chip), kdy všechny potřebné obvody pro běh procesoru (kromě externí paměti), jsou umístěny přímo na čipu. Architektura procesoru je zobrazena na obrázku 3.1. Procesor používá dvě sběrnice – sběrnici procesorovou a sběrnici periferií.

Pro snížení zatížení sběrnice a s tím spojené spotřeby energie procesoru jsou k procesorové sběrnici připojena jen zařízení vyžadující velikou propustnost sběrnice. Tedy jádro procesoru (JEM2), vnitřní paměti (RAM) a externí rozhraní sběrnice (External Bus Interface). Sběrnice periferií umožňuje přístup k periferiím umístěným uvnitř procesoru (vnitřní periferie). K procesorové sběrnici je sběrnice periferií připojena pomocí mostu (bridge), Toto připojení též funguje jako oddělení těchto dvou sběrnic – v případě přetížení procesorové sběrnice.

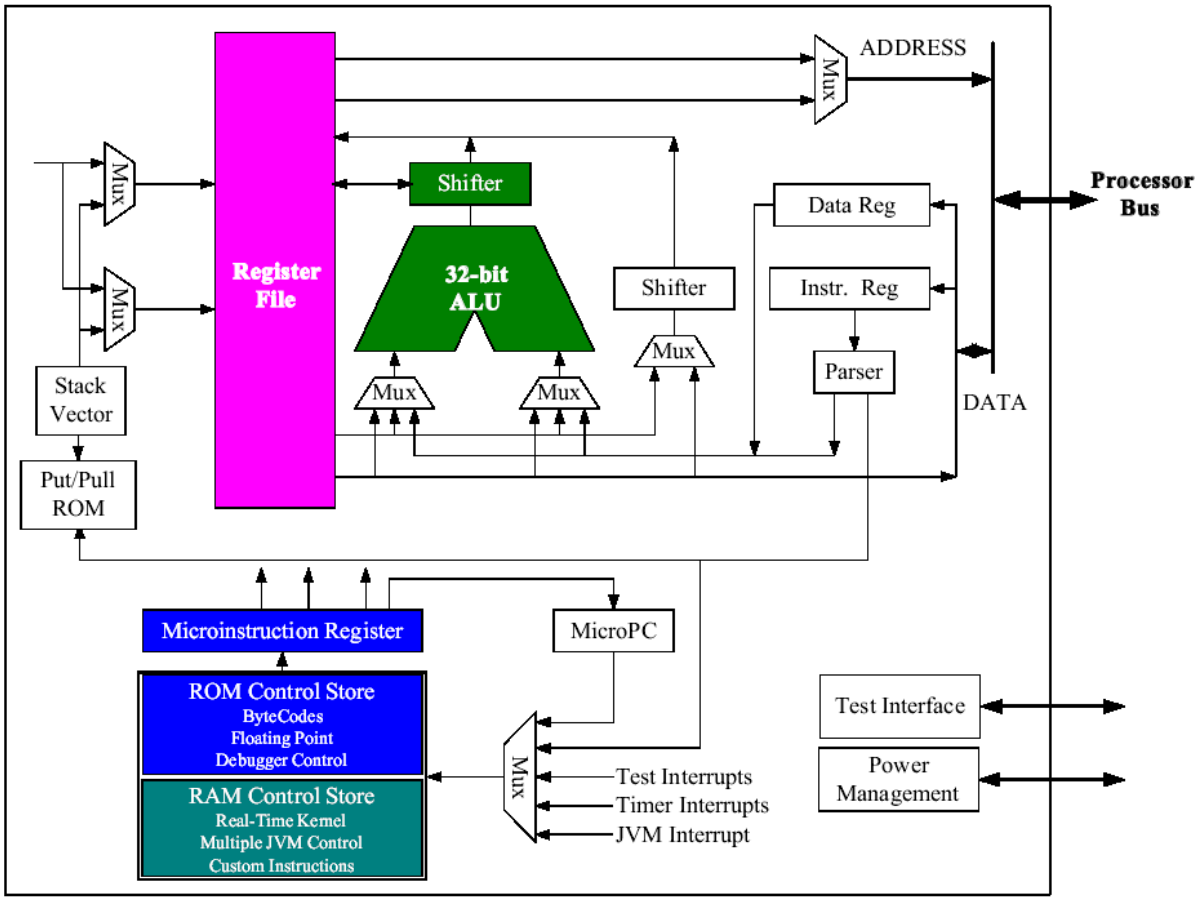
Externí rozhraní sběrnice (External Bus Interface) poskytuje navenek adresovou, datovou a řídící sběrnici, ke kterým je možné připojit většinu pamětí a periferních zařízení.



Obrázek 3.1 – Architektura procesoru aJ-80

3.3.1 Jádro procesoru (JEM2)

Vlastní výkonnou jednotkou, která provádí uživatelský program, je jádro procesoru (JEM2). Toto zařízení kompletně nahrazuje softwarový JVM známý z klasické architektury prostředí jazyka Java. Na obrázku 3.2 je zobrazeno blokové schéma jádra procesoru (JEM2).



Obrázek 3.2 - Jádro procesoru aj-80 (JEM2)

3.3.1.1 Uživatelsky definované instrukce

Procesor aJ-80 poskytuje možnost vytváření uživatelsky definovaných instrukcí. Ty jsou implementovány jako posloupnost instrukcí mikrokódu (nejmenší, nedělitelná – atomická – programová jednotka). Uživatelské instrukce mohou výrazně zvýšit výkon (snížit časovou náročnost) často používaných algoritmů.

Jako příklad použití uživatelsky definované instrukce uvedu výpočet druhé odmocniny. V knihovnách jazyka Java existuje metoda sqrt třídy java.lang.Math. Tato metoda může být implementována pomocí mikrokódu. Potom může být tato posloupnost mikrokódu provedena při zavolání příslušné uživatelské instrukce, které je mikrokód přiřazen. Při zpracovávání souboru typu class pomocí programu JEM Builder je volání

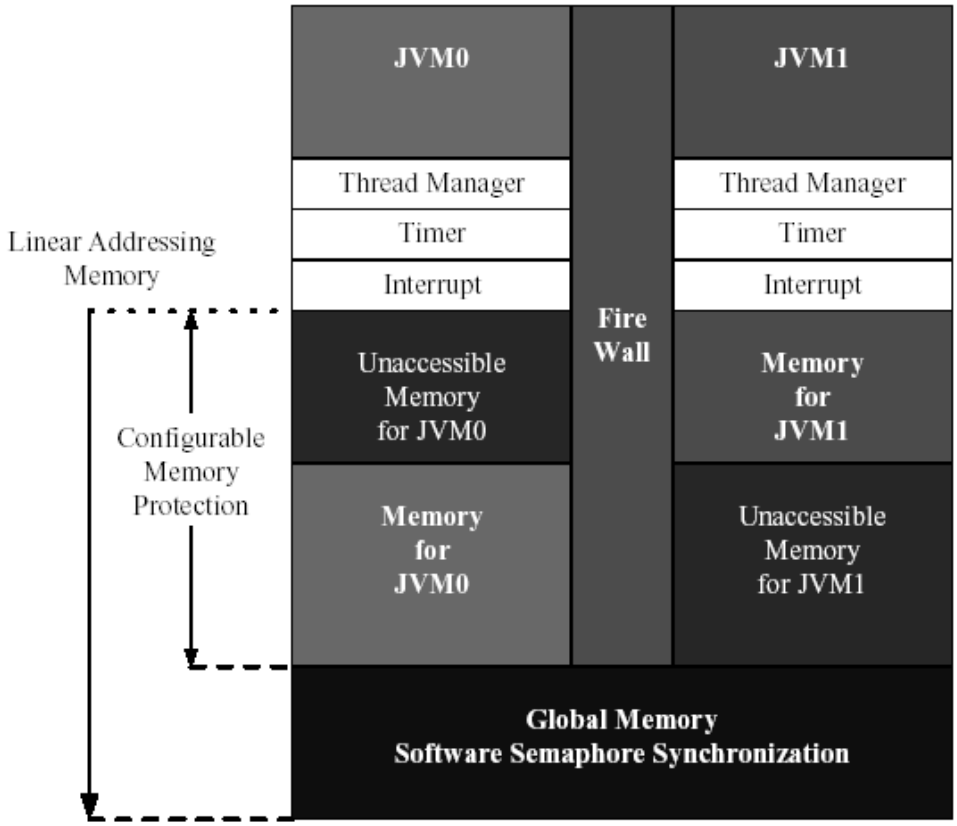
metody java.lang.Math.sqrt nahrazeno uživatelsky definovanou instrukcí. Výrobce uvádí pětinásobné zlepšení výkonu pro jednoduché algoritmy a až dvacetinásobné pro složité.

3.3.2 Vícenásobný JVM

Vícenásobný JVM (virtuální stroj Java) poskytuje možnost provádění dvou nezávislých Java aplikací. Provádění je naplánováno do časových slotů, jejichž velikost (doba časového intervalu) je předem známá, nastavitelná a samovolně neměnná.

Procesor aJ-80 ve skutečnosti neobsahuje dva hardwarové JVM, jak by se z předchozího mohlo zdát, ale pouze jeden. Dvojnásobnost si můžeme představit jako dvě instance JVM, každá má přidělenu vlastní pracovní oblast paměti a systémové zdroje, aby mohla provádět Java aplikace. Vlastní fyzický JVM je přepínán mezi kontexty obou instancí procesoru a to po přesně daných časových intervalech. Dobu potřebnou pro přepnutí mezi kontexty výrobce uvádí menší než 1 μs, při taktu systémových hodin 80 MHz.

Každá z instancí má vlastní plánovač vláken a správu paměti. Obě instance JVM ovládá Multiple JVM Manager (MJM). MJM ovládá přepínání mezi kontexty, také zajišťuje, aby nedocházelo ke „střetu zájmů“ mezi instancemi v případě přerušení a používání vnitřních periferií. Oddělení obou instancí není úplné – aby mohly mezi sebou komunikovat, je vyhrazena sdílená oblast paměti, kde mohou být umístěny sdílené objekty určené pro komunikaci.



Obrázek 3.3 – Schéma MJM (vícenásobný JVM)

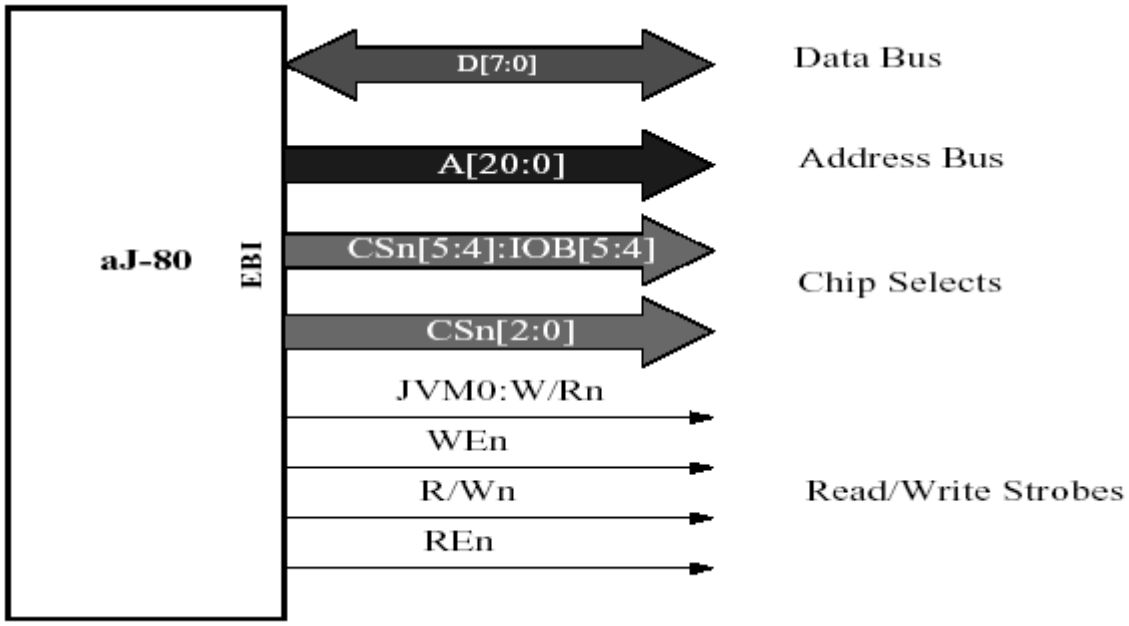
3.3.3 Vnitřní a vnější paměť

3.3.3.1 Vnitřní paměť

Procesor aJ-80 obsahuje 48kB vnitřní paměti typu RAM: Z toho 32kB je používáno jádrem procesoru (JEM2) jako pracovní zásobník. Zbývajících 16kB je použito pro práci s mikrokódem – implementace rozšířeného bajtového kódu a uživatelských instrukcí.

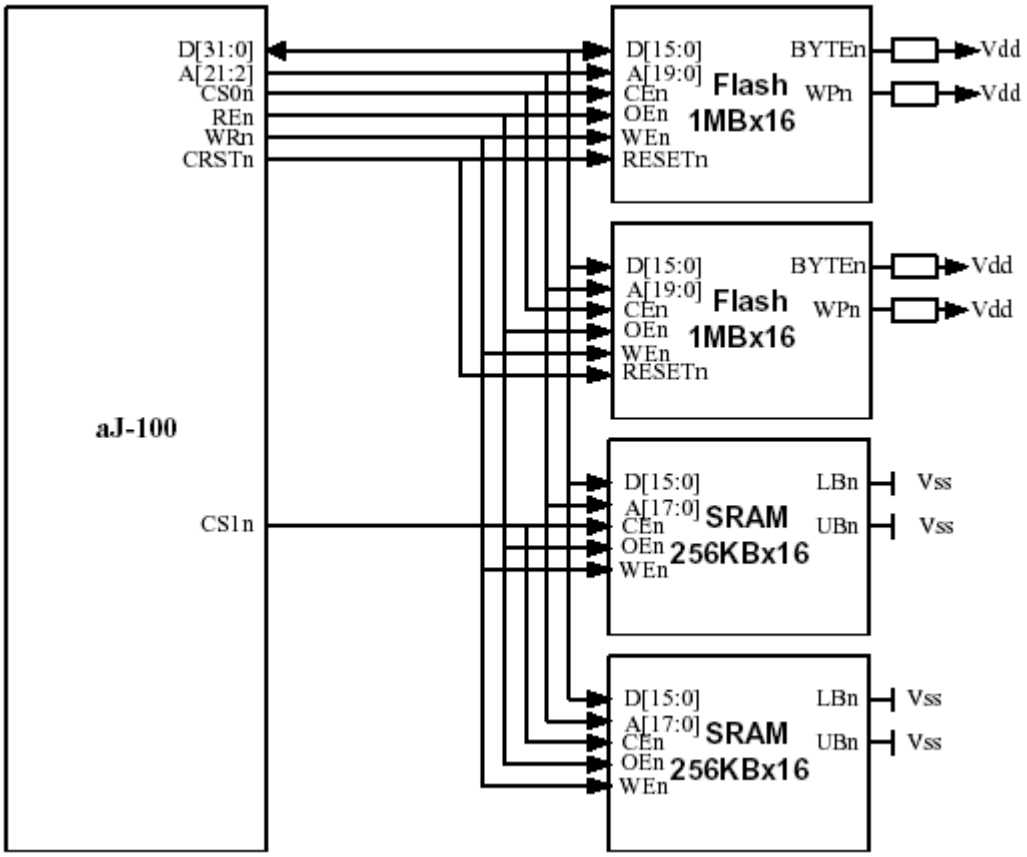
3.3.3.2 Vnější rozhraní sběrnice

Vnější rozhraní sběrnice External Bus Interface (EBI) generuje řídící signály pro vnější připojené paměti nebo periferní zařízení. EBI může přímo ovládat až 256MB externího paměťového prostoru. EBI je konfigurovatelné pro podporu 32, 16 a 8 bitových paměťových zařízení. Řídící signály dovolují přímé připojení externích pamětí nebo periferních zařízení s možností přímého přístupu „paměťovým“ způsobem. Na obrázku 3.4 je schéma EBI.



Obrázek 3.4 - Vnější rozhraní sběrnice - External Bus Interface (EBI)

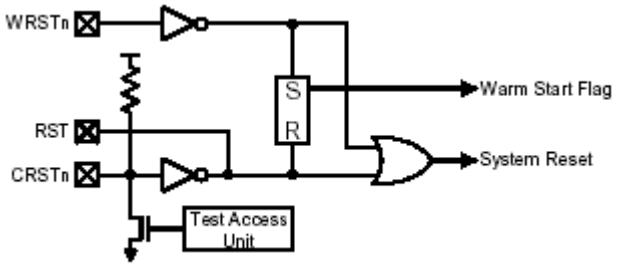
Praktická ukázka možností připojení externích paměťových zařízení je na obrázku 3.5. V tomto případě se jedná o připojení dvou pamětí typu SRAM a dvou pamětí typu FLASH.



Obrázek 3.5 – Příklad připojení externích paměťových zařízení k aJ-80

3.3.4 Logika RESET

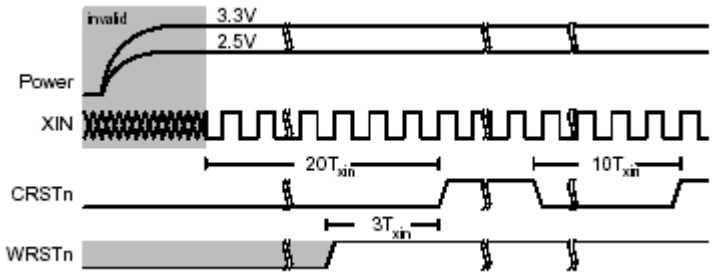
U procesoru aJ-80 můžeme využít jak tzv. „studený reset“ – cold start reset (CRSTn), tak „teplý reset“ – warm start reset (WRSTn). Schéma zapojení „resetovací“ logiky je na obrázku 3.6.



Obrázek 3.6 – Schéma zapojení logiky RESET

Signál CRSTn se používá při připojení napájecího napětí k procesoru. Obrázek 3.7 zobrazuje požadavky na změny signálů CRSTn a WRSTn v čase. Signál CRSTn musí zůstat aktivní do doby, než se napájecí napětí ustálí na správných hodnotách (2,5V pro jádro procesoru a 3,3V pro vnitřní periferie). Zdroj hodinového taktu má také určitou prodlevu, po připojení napájení – než se ustálí na přípustných hodnotách. Navíc nesmí být signál CRSTn deaktivován v průběhu dvaceti period hodinového signálu

XIN poté, co se ustálilo napájecí napětí a zdroj systémových hodin. Minimální časový interval, po který musí být signál CRSTn v aktivním stavu, pokud se nejedná o situaci po připojení napájecího napětí, je 10 period hodinového signálu. Abychom předešli vyvolání „teplého“ restartu zároveň se „studeným“, musí být signál WRSTn deaktivní v průběhu minimálně tří period hodinového signálu před okamžikem, kdy je signál CRSTn deaktivován.



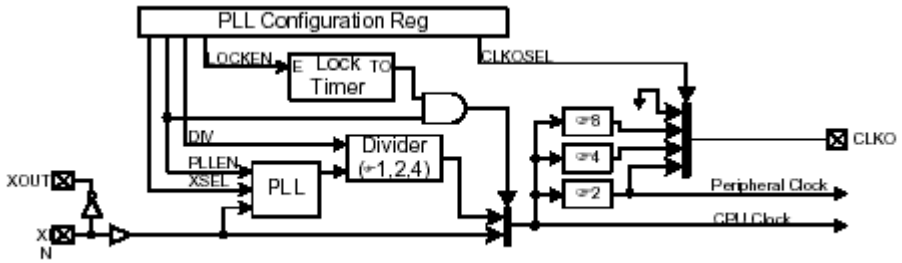
Obrázek 3.7 – Časové průběhy signálů CRSTn a WRSTn

3.3.5 Systémové hodiny

Procesor aJ-80 používá vnitřní fázový závěs – Phase Locked Loop (PLL) pro generování vysokofrekvenčního hodinového signálu pro CPU a vnitřní periferie, na základě reference z vnějšího krystalu nebo oscilátoru.

PLL vytváří z frekvence oscilátoru (10-20MHz) hodinový signál s kmitočtem do 80MHz. Zapojení logiky PLL je na obrázku 3.8.

Chování PLL se konfiguruje pomocí registru procesoru PLL Configuration Register. PLL může být zcela vypnuto, pokud pro méně náročné systémy (na výkon CPU) požadujeme nižší spotřebu energie.



Obrázek 3.8 – Schéma zapojení logiky PLL

3.3.6 Systém přerušení

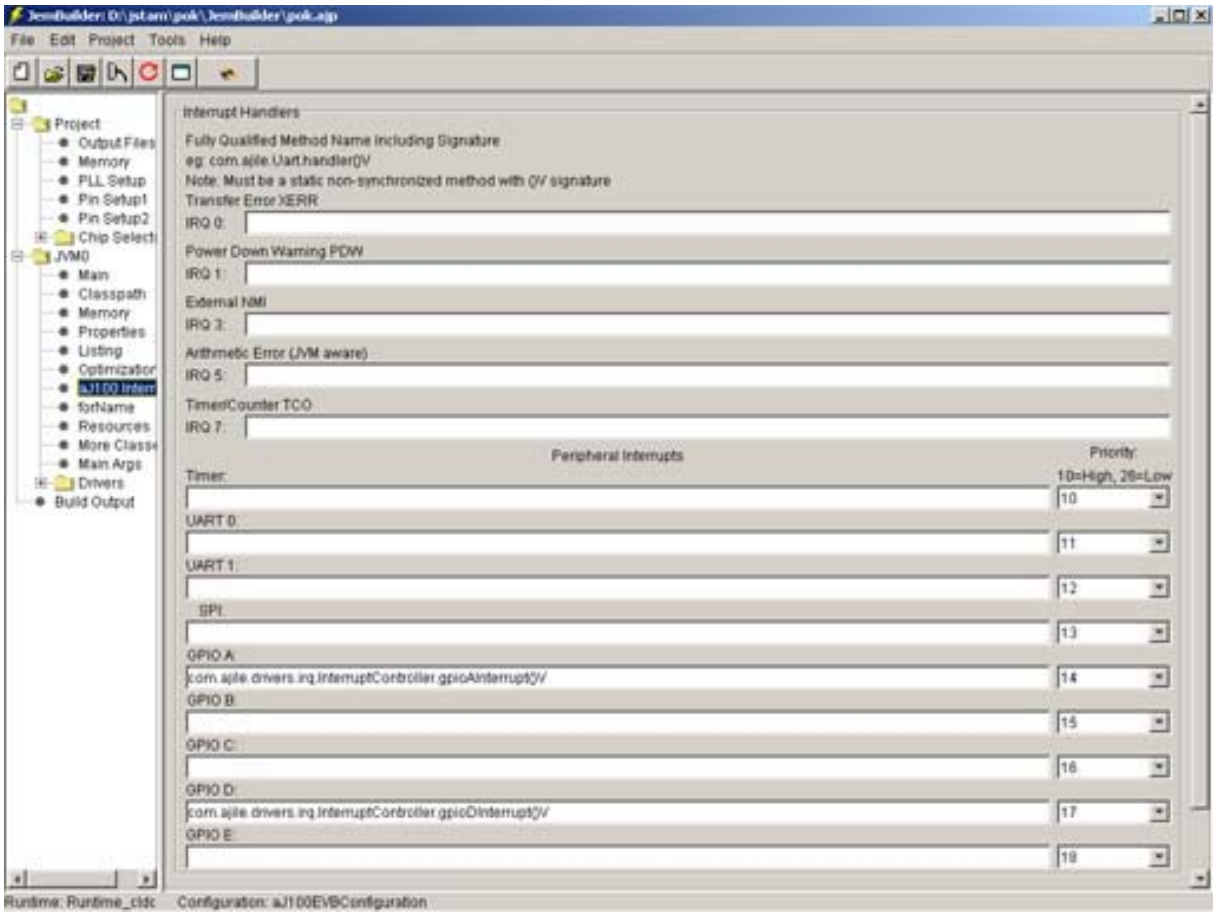
Procesor aJ-80 podporuje 26 maskovatelných, asynchronních přerušení, aktivovaných sestupnou hranou. Přerušení jsou seřazena podle priorit. Některé z těchto přerušení jsou použita vnitřní logikou jádra procesoru (JEM2). Ostatní jsou vyhrazena pro vnitřní periferní zařízení. Čtyři přerušení s nejvyšší prioritou jsou nemaskovatelná, z důvodu jejich velikého významu pro správný chod procesoru. Význam jednotlivých přerušení je shrnut v tabulce.

Číslo přerušení	Označení	Popis
0	Transfer Error (XERR)	Přerušení je generováno ochrannou logikou vnější paměti, pokud se procesor pokouší zapsat mimo vymezenou oblast paměti, dedikovanou pro JVM
1	Power Down Warning (PDW)	Přerušení je generováno vnější logikou jako upozornění, že dojde k přerušení napájení
2	JVM Switch Interrupt (JSI)	Přerušení je generováno vnitřním časovačem JSI jako signál, že má být přepnut kontext na další JVM
3	External NMI (ENMI)	Přerušení generováno externí logikou, pokud nastane událost specifická pro uživatelskou aplikaci
5	Arithmetic Error (OVR)	Přerušení je generováno vnitřními obvody, pokud při aritmetickém výpočtu dojde k chybě. Jako například aritmetické přetečení, dělení nulou, atd.
7	Timer/Counter Output (TCO)	Přerušení (specifické pro každý JVM) je generováno, pokud vnitřní čítač/časovač dočítal do nuly
8	Piano Roll Timer Output (PTO)	Přerušení (specifické pro každý JVM) je generováno, pokud vnitřní čítač typu „piano roll“ dočítal do nuly
10-25	Peripheral Interrupts	Přerušení vyhrazená pro vnitřní periferní zařízení

Tabulka 3.1 - Význam jednotlivých přerušení procesoru aJ-80

Při vyvolání přerušení je nejprve vyhledáno aktivní přerušení s nejvyšší prioritou (čím menší číslo přerušení, tím větší priorita). Přerušení s nejvyšší prioritou je obslouženo vnitřní obslužnou rutinou nebo je vyvolána statická metoda, přiřazená k danému přerušení.

Přiřazení obslužných metod se provádí v programu JEMBuilder, viz obrázek 3.9.



Obrázek 3.9 - Přiřazení obslužných metod přerušením v programu JEMBuilder

3.4 Vnitřní periferní zařízení

Procesor aJ-80 obsahuje některé periferní zařízení přímo uvnitř procesoru – vnitřní periferie. Jsou to čítače/časovače, universální vstupně/výstupní obvod, obvody UART, universální obvod pro sériovou komunikaci a můžeme mezi ně započítat i obvod testovacího rozhraní. Periferní zařízení komunikují vzájemně a s CPU přes sběrnici periférií.

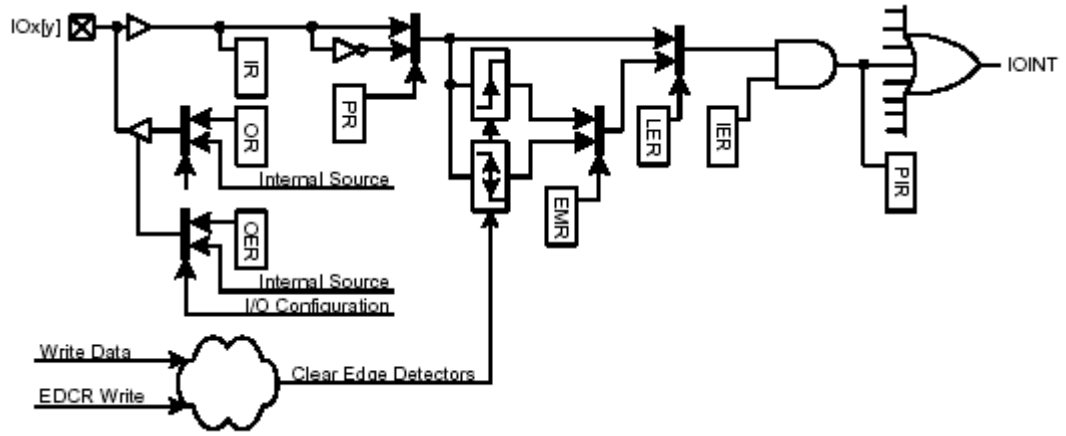
Komunikace s vnitřními periferiemi je z uživatelského programu možná v zásadě dvěma způsoby. Použitím Java tříd a rozhraní z výrobcem dodávaných knihoven nebo přímým zápisem do konfiguračních registrů (čtení ze stavových registrů) příslušných periferií. Na závěr každé podkapitoly uvádím jednotlivé Java třídy a rozhraní, vztahující se k příslušnému periferním zařízení. Popis konfiguračních (stavových) registrů je velmi přehledně popsán v literatuře [17].

3.4.1 Univerzální vstupně/výstupní obvod

Procesor aJ-80 obsahuje pět osmibitových vstupně/výstupních portů pro všeobecné použití (General Purpose I/0 port – GPIO).

Jeden z portů s označením A (GPIOA), má vyhrazeno osm fyzických pinů (pro každý bit jeden). Ostatní porty (B, C, D a E) sdílejí některé fyzické

piny s ostatními vnitřními zařízeními procesoru. Maximální proud, který může protékat piny portu A, je 24mA. U ostatních čtyř portů je to 8mA. Schéma zapojení je na obrázku 3.9, diagram je stejný pro každý jednotlivý pin každého z pěti portů.



Obrázek 3.10 - Schéma zapojení jednoho vstupu/výstupu GPIO

Logická úroveň přivedená na vstup IOx[y] se ukládá v registru IR. To, zda se jedná o vstupní nebo výstupní pin a jeho povolení, či zakázání, nastavíme pomocí registru OER. V případě, že se jedná o výstupní pin, nastavíme požadovanou hodnotu výstupu IOx[y] registrem OR.

Každý jednotlivý vstupní pin libovolného portu může generovat přerušení. Podmínky pro vyvolání (nevyvolání) přerušení jsou dány nastavením registrů PR (Priority Register), LER (Level/Edge Register) a EMR (Edge Mode Register). Těmito registry nastavíme, zda bude přerušení vyvoláno jakoukoliv změnou úrovně signálu, náběžnou hranou, sestupnou hranou nebo při úrovni signálu logická jedna, či nula. V následující tabulce je přehled nastavení jednotlivých registrů pro různé druhy spouštění přerušení.

Podmínka spuštění přerušení	PR [x]	LER [x]	EMR [x]
úroveň logická 0	0	0	-
úroveň logická 1	1	0	-
sestupná hrana	0	1	0
náběžná hrana	1	1	0
jakákoliv změna	-	1	1

Tabulka 3.2 – Nastavení registrů pro různé druhy spouštění přerušení

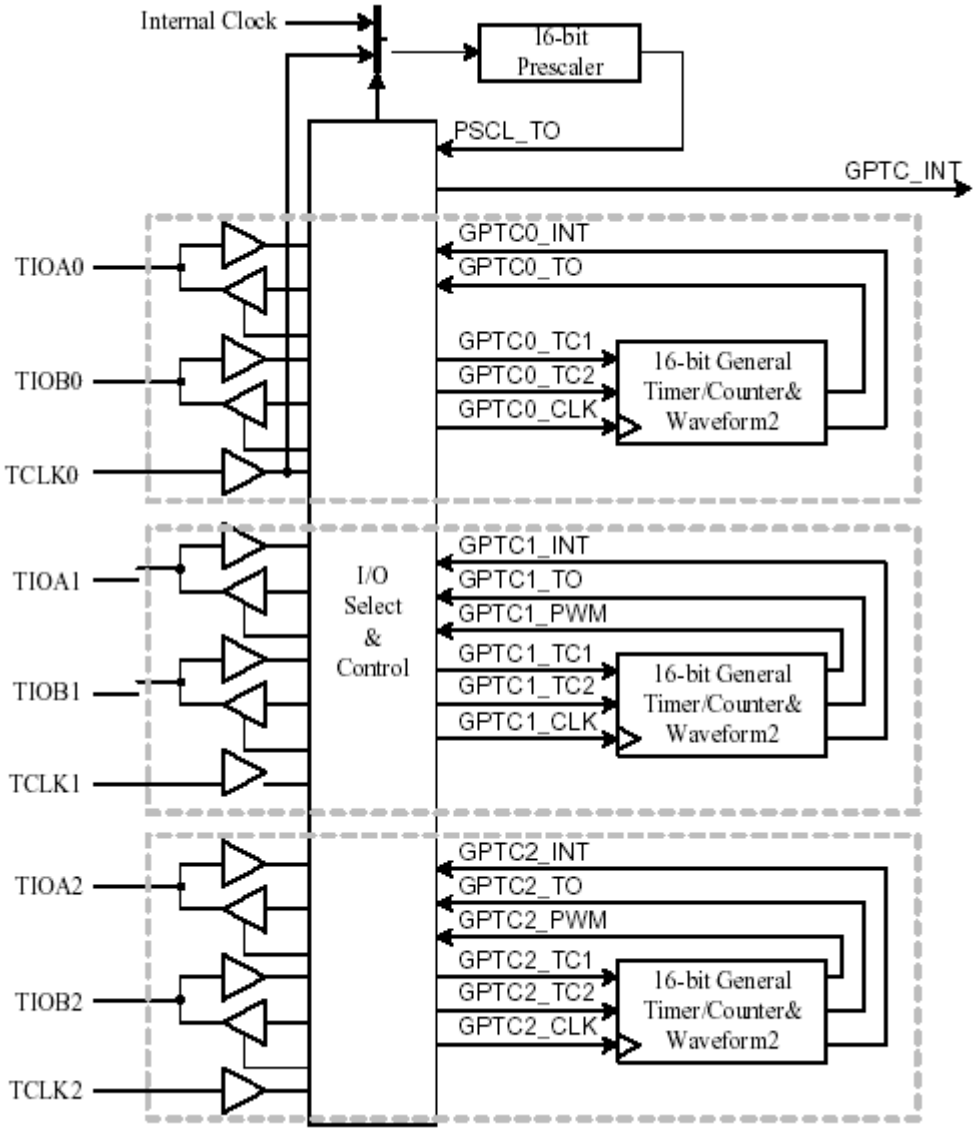
3.4.1.1 Java třídy a rozhraní

com.ajile.drivers.gpio.GpioField
com.ajile.drivers.gpio.GpioPin
com.ajile.drivers.gpio.IOPort

3.4.2 Univerzální obvody čítače/časovače

Procesor aJ-80 obsahuje funkční blok univerzálních čítačů/časovačů – General Purpose Timer/Counter (GPTC). Pro rychlý přehled následuje souhrn vlastností bloku GPTC:

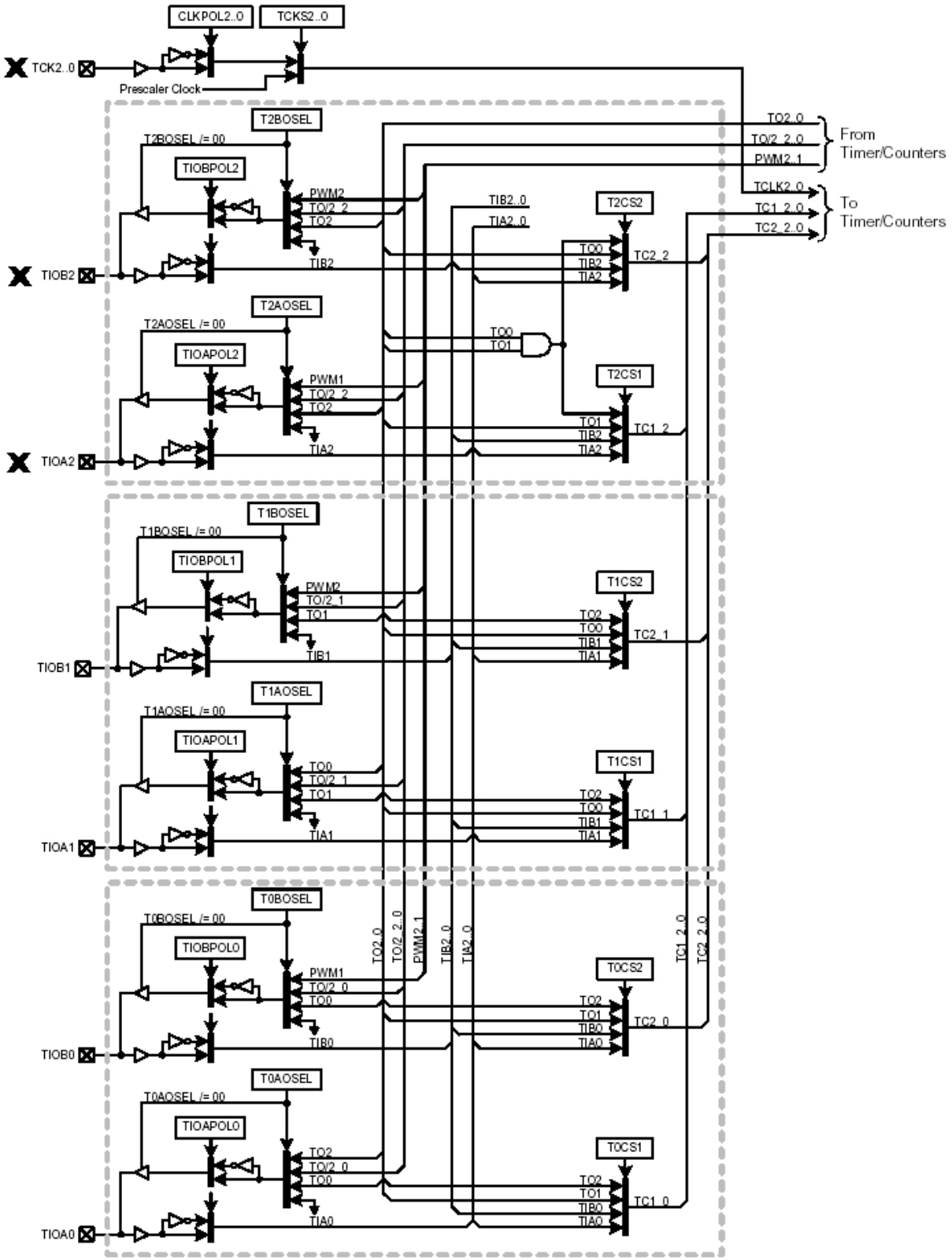
- obsahuje třikrát 16bitový čítač/časovač
- čítače/časovače je možné spojovat do kaskád (uvnitř GPTC)
- řízení hodinového taktu možné i z vnějšího zdroje
- 2x modul pro hardwarové vytváření pulzně šířkové modulace (PWM)
- 2x modul pro vzorkování vstupního signálu
- může generovat přerušení procesoru
- obsahuje 16bitový dělič hodinového signálu



Obrázek 3.11 - Blokové schéma obvodů GPTC

Na obrázku 3.11 je znázorněno blokové schéma funkčního bloku GPTC. Vodiče označené TIOA0-2, TIOB0-2 a TCLK0-2 jsou vstupy/výstupy

z/do bloku GPTC. Generování přerušení je naznačeno vodičem označeným GPTC_INT. Funkční blok GPTC obsahuje vlastní čítače/časovače, dělič hodinového signálu (Prescaler) a vstupně/výstupní blok, pomocí něhož definujeme fyzické propojení jednotlivých částí GPTC (I/O Select & Control).



Obrázek 3.12 - Vstupně/výstupní blok GPTC

3.4.2.1 Vstupně výstupní blok GPTC

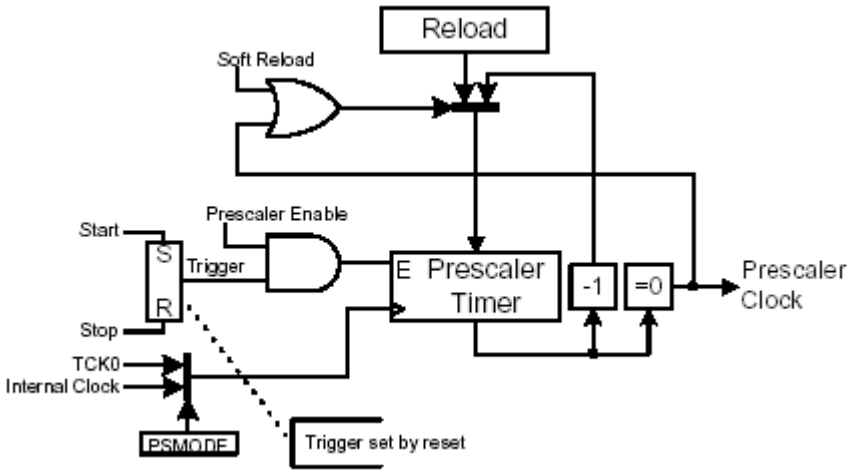
Schéma vstupně/výstupního bloku GPTC je na obrázku 3.12. Vodiče v levé části obrázku označené TIOA0-2, TIOB0-2 a TCLK0-2 reprezentují fyzické vývody (piny) procesoru. Obrázek 3.12 pochází z dokumentace k procesoru aJ-100 (což je následník aJ-80, více v kapitole 4). Procesor aJ-80 obsahuje vstupně/výstupní blok GPTC totožný se schématem na obrázku, jen některé vstupně/výstupní vodiče nejsou fyzicky spojeny s vývody na pouzdře procesoru. Jedná se o vodiče označené TCK1, TCK2, TIOA2 a TIOB2 (na obrázku označené křížkem). Vodiče TCK0, TIOA0-1, TIOB0-1 jsou připojeny na fyzické vývody pouzdra procesoru, ale kromě vodiče TCK0, který má dedikován samostatný pin, sdílejí fyzické vývody s některými z pinů portu GPIOE. Příslušné sdílené piny jsou na obrázku uvedeny s označením IOE3-7.

Jak je patrné z obrázku 3.12, vstupně/výstupní blok GPTC umožňuje množství různých variant nastavení vzájemného propojení funkčních částí a vstupně/výstupních vodičů. Pomocí registrů TIO[A,B]POL0-2 můžeme nastavit polaritu vstupních nebo výstupních signálů (přesněji to, zda jsou signály invertovány, či ne).

Nastavení registrů T[0-2][A,B]SEL určuje, zda jsou vodiče TIOA0-2, TIOB0-2 vstupními, či výstupními. V případě, že se jedná o výstupní vodiče, určují registry T[0-2][A,B]SEL navíc, co bude zdrojem signálu. Pro názornost, v případě registru T0BOSEL, může být jako zdroj vybrán signál pulzně šířkové modulace generované čítačem/časovačem č. 1 (PWM1), výstupní signály čítače/časovače č. 0 nebo je vodič TIOB nastaven jako vstupní. Soubor registrů T[0-2]CS[1,2] určuje, které signály budou přivedeny na vstup čítačů/časovačů. Registry TCKS0-2 určují zdroj hodinového signálu pro každý z čítačů/časovačů. Je to buď externí zdroj hodinového signálu (vodiče TCK0-2) nebo výstup z vnitřního děliče (Prescaler Clock).

3.4.2.2 Dělič hodinového taktu

Blok GPTC v sobě obsahuje dělič frekvence hodinového signálu. Dělič je 16bitový, může tedy dělit hodinový signál v rozsahu 1 až 65536. Schéma zapojení děliče je na obrázku 3.13. Hodnota dělení frekvence se nastavuje zápisem do registru GPTC_PRLR (GPTC Prescaler Reload Register). Konfigurační registr PSMODE určuje, zda bude vstupem pro dělič signál z vodiče TCK0 nebo signál vnitřních systémových hodin (Internal Clock).



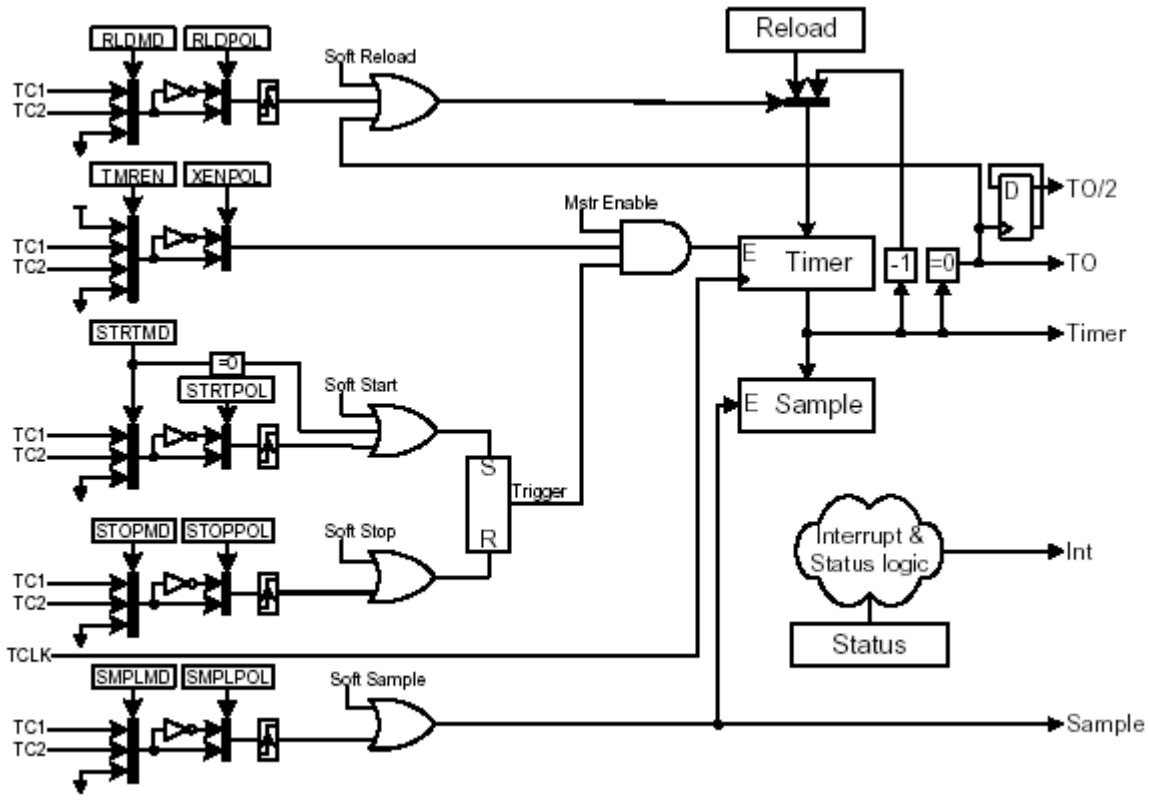
Obrázek 3.13 - Dělič frekvence hodinového signálu

3.4.2.3 Čítače/časovače

Procesor aJ-80, potažmo funkční blok GPTC, obsahuje tři čítače/časovače k uživatelskému využití. Schéma zapojení jednoho z nich (všechny tři jsou konstrukčně identické) je na obrázku 3.14. Vstupy čítače/časovače jsou vodiče označené TC1 a TC2. Pomocí konfiguračních registrů definujeme, které z akcí čítače/časovače budou vstupy TC1 a TC2 ovlivňovat. Akce jsou tyto:

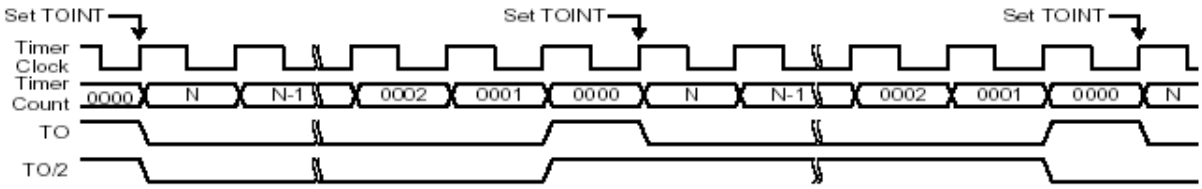
- znovunačtení přednastavené hodnoty z registru Reload
- povolení operace čítače/časovače (THREN)
- spuštění čítače/časovače (STRTMD)
- zastavení čítače (STOPMD)
- vzorkování (SMPLMD)

Tyto akce mohou být také vyvolány softwarově, zápisem do příslušných registrů. V obrázku tomuto faktu odpovídají vodiče označené Soft Reload, Soft Start, Soft Stop a Soft Sample.



Obrázek 3.14 - Vnitřní logika čítače/časovače

Základní operace čítače/časovače jsou zobrazeny na obrázku 3.15. Tento časový diagram zobrazuje, jak se chová čítač/časovač, pokud je na jeho vstup přiveden hodinový signál (Timer Clock). Čítač snižuje vnitřní hodnotu (Timer Count) o jeden krok v taktu hodinového signálu. Počáteční hodnota položky Timer Count je N. Po dosažení hodnoty 0 je generováno přerušení (TOINT) a je znovu načtena hodnota N do položky Timer Count. Význam výstupních signálů TO a TO/2 je zřejmý z obrázku.

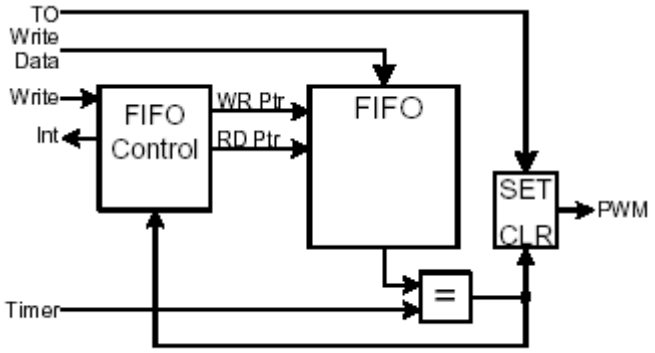


Obrázek 3.15 - Časový diagram základních operací čítače/časovače

3.4.2.4 Generování pulzně šířkové modulace (PWM)

Na obrázku 3.16 je schéma zapojení obvodu, který na základě nastavení v registrech procesoru samostatně generuje pulzně šířkovou modulaci (PWM). Tento obvod spolupracuje s čítačem/časovačem, Na obrázku vidíme vodič označený TO, což je výstup čítače/časovače. Procesor

aJ-80 obsahuje celkem dva tyto obvody pro generování PWM. Uživatelsky můžeme nastavit dobu periody a střídu generovaného signálu PWM.

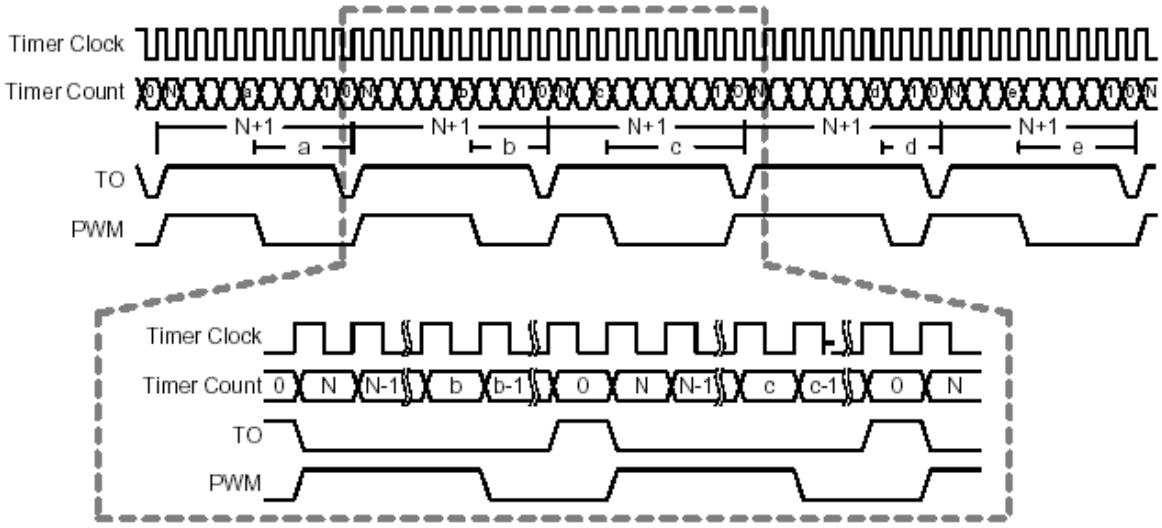


Obrázek 3.16 - Obvod generování pulzně šířkové modulace (PWM)

Na obrázku 3.17 je zobrazen časový průběh generovaného signálu PWM. Délku periody generovaného signálu nastavujeme zápisem do registru GPTCx_RLR (GPTC Reload Register, symbol x je číslo příslušného čítače/časovače svázaného s generátorem PWM). Na obrázku 3.17 odpovídá hodnotě registru GPTCx_RLR hodnota N. Perioda generovaného signálu je dlouhá přesně N+1 hodinových taktů. Střída signálu PWM se vypočte následovně:

$$Střída = \frac{(N+1) - FIFOSample}{(N+1)},$$

položka FIFOSample je aktuální výstupní hodnota zásobníku typu FIFO – viz obrázek 3.17. Zásobník FIFO má velikost 32 hodnot (hodnoty jsou 16bitové). Hodnoty zásobníku FIFO jsou na obrázku 3.17 označeny písmeny a, b, c, d, e.

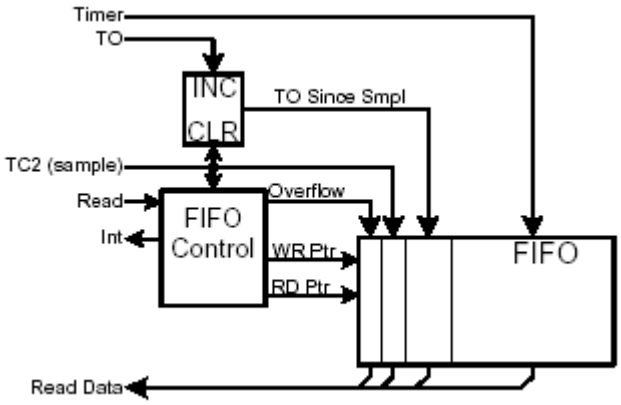


Obrázek 3.17 - Časový průběh generování signálu PWM

Pokud do zásobníku FIFO nedodáváme další hodnoty, zůstane po vyprázdnění zásobníku nastavena střída výstupního signálu PWM, podle poslední hodnoty získané ze zásobníku FIFO.

3.4.2.5 Obvod automatického vzorkování

Schéma zapojení obvodu, který na základě změn vstupního signálu zaznamenává počet hodinových pulzů, je na obrázku 3.18. Funkční blok GPTC obsahuje tyto obvody dva, každý z nich kooperuje s čítačem/časovačem, který je pro vzorkovač zdrojem hodinového signálu. Obvod vzorkování pracuje následovně. Sleduje vstupní signál TC2 a při jeho změně uloží do zásobníku FIFO počet hodinových pulzů signálu TO. Zásobník FIFO má velikost 32 hodnot a každá hodnota je 18bitová. Při zaplnění zásobníku FIFO je nově získaná hodnota uložena na poslední místo zásobníku. Tím je přepsána předchozí uložená hodnota.



Obrázek 3.18 - Obvod automatického vzorkování

3.4.2.6 Java třídy a rozhraní

com.ajile.drivers.gptc.TimerCounter
com.ajile.drivers.gptc.WaveformModule

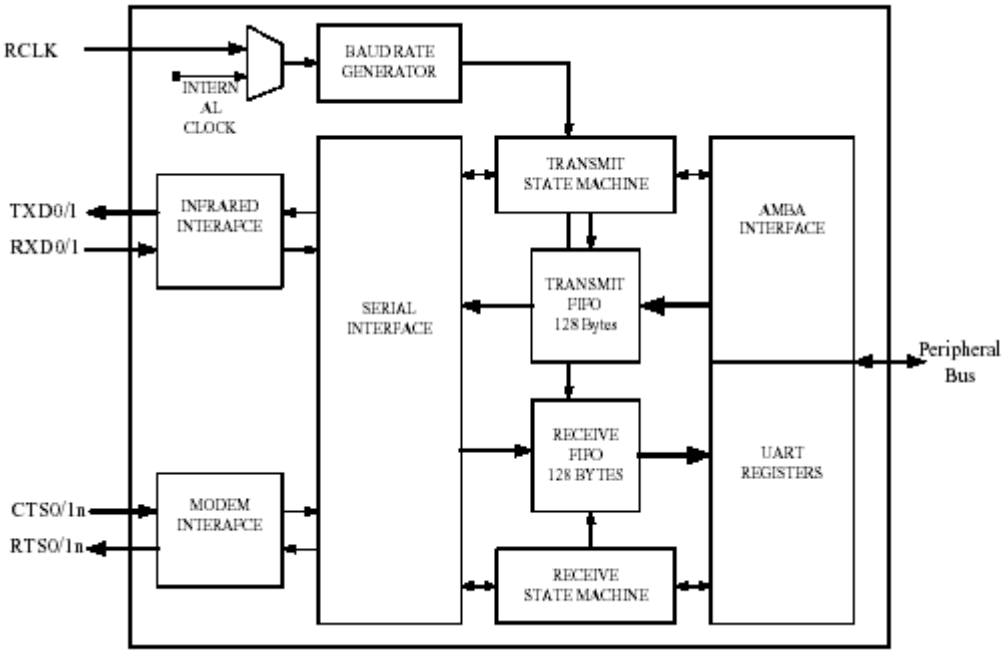
3.4.3 Obvody UART

Procesor aJ-80 obsahuje, jako vnitřní periferie, dva obvody typu UART (Universal Asynchronous Receiver/Transmiter) podle průmyslového standardu 16550 UART. Jejich základními vlastnostmi jsou :

- softwarová kompatibilita s obvodem NSC-NS16550A
- programovatelná délka slova, počet stop bitů, typ parity
- programovatelná rychlost přenosu (Baud Rate)
- možnost generování přerušení procesoru
- případná diagnostika na základě zpětné smyčky
- zásobník FIFO o délce 128 bajtů pro vysílání i příjem

- podpora protokolu fyzické vrstvy IrDA1.0 s rychlostí do 115,2 kb/s

Na obrázku 3.19 je blokové schéma obvodu UART. Obvod je ovládán (programován) přes sběrnici periferií (Peripheral Bus).



Obrázek 3.19 - Blokové schéma obvodu UART

3.4.3.1 Java třídy a rozhraní

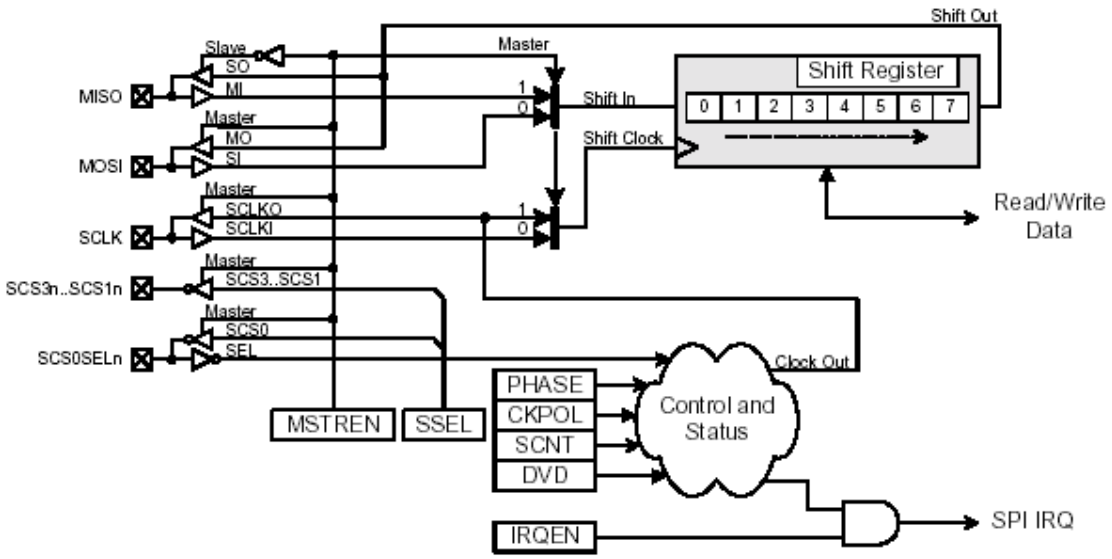
com.ajile.drivers.aJileCommDriver
com.ajile.drivers.SerialChip
com.ajile.drivers.SerialOutputStream
com.ajile.drivers.SerialInputStream

3.4.4 Univerzální obvod pro sériovou komunikaci

Mezi vnitřní periferie procesoru aJ-80 se také řadí univerzální obvod pro sériovou komunikaci s vnějšími periferiemi – Serial Peripheral Interface (SPI).

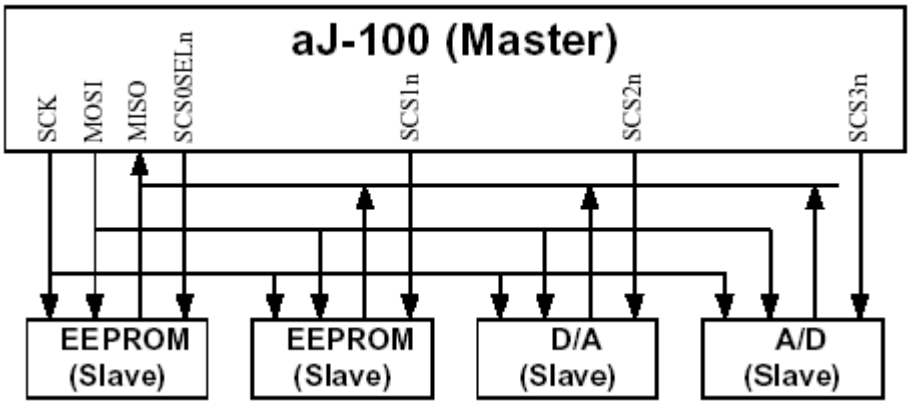
Přes obvod SPI můžeme k procesoru aJ-80 připojit množství periferních zařízení, jako například A/D a D/A převodníky, sériové paměti EEPROM, senzory, signálové procesory (DSP), dotykové obrazovky a další. Procesor aJ-80 podporuje současné připojení až čtyř podřízených (Slave) periferních zařízení přes rozhraní SPI.

Obvod SPI umožňuje programové nastavení délky slova, frekvence hodinového signálu a jeho polaritu. Na obrázku 3.20 je schéma zapojení obvodu SPI.



Obrázek 3.20 - Sériové rozhraní SPI

Příklad připojení čtyř periferních zařízení k procesoru aJ-80 přes rozhraní SPI je na obrázku 3.21.



Obrázek 3.21 - Připojení periferních zařízení přes rozhraní SPI

3.4.4.1 Java třídy a rozhraní

com.ajile.drivers.spi.SpiMaster

3.5 Správa paměti a vláken

3.5.1 Správce vláken

Procesor aJ-80 podporuje jak běžná vlákna známá ze specifikace jazyka Java, tak implementuje speciální druhy vláken. S běžnými Java vlákny se pracuje standardním způsobem, známým z prostředí Java.

Jednotlivá vlákna mají určitou prioritu a to v rozmezí 1 až 10 (čím větší číslo, tím vyšší priorita). Pokud v systému přejde do připraveného stavu vlákno s vyšší prioritou, než je právě vykonáváno, je aktivnímu vláknu pozastavena činnost a je spuštěno příslušné připravené vlákno s vyšší prioritou. Po přechodu právě aktivního vlákna do stavu spánku systém prohledá frontu čekajících vláken, vybere vlákno s nejvyšší prioritou a to spustí.

Rozšířením běžného vlákna jazyka Java (třídy java.lang.Thread) je periodicky spouštěné vlákno (třída com.ajile.jem.PeriodicThread).

Periodické vlákno je prováděno jako běžné vlákno, do doby než je zavolána metoda cycle(). Poté přejde periodické vlákno do stavu spánku. Vlákno setrvá v tomto stavu do té doby, než vyprší nastavený časový interval. Tento interval se nepočítá od okamžiku přechodu do stavu spánku (jako je tomu u standardního vlákna a volání metody wait(), ale od okamžiku, kdy bylo vlákno naposledy aktivováno. Tím je dosahováno pravidelného přesného periodického spouštění vlákna. Doba potřebná na provedení vlastního kódu vlákna tak neovlivní dobu periody spouštění periodického vlákna. Periodické vlákno má vyšší prioritu než běžné vlákno se stejnou číselnou hodnotou priority. Tzn. pokud nastavíme periodickému vláknu prioritu například na hodnotu 5, bude mít vlákno ve skutečnosti hodnotu priority v rozmezí 5 a 6.

3.5.2 Ochrana paměti

Ochrana paměti je v procesoru aJ-80 dána implementací principů programovacího jazyka Java. Jde o to, že uživatel (na obecné úrovni), nemá možnost zasahovat do operační paměti procesoru. V jazyce Java neexistuje práce s ukazateli, jak to známe například z jazyka C, což by bylo zdrojem potenciálních problémů.

Na úrovni obecného jazyka Java je práce s pamětí bezpečná a pokud potupujeme programátorsky korektně, tak je malá pravděpodobnost, že bychom způsobili nějaké problémy. Ovšem, jak již bylo řečeno, procesor aJ-80 implementuje rysy specifikace RTSJ. Je zde proto možnost přímého přístupu do paměti a práce s ní.

Je tedy vhodné oddělit aplikace, které pracují na nejnižší úrovni – práce s pamětí, registry a vnitřními/vnějšími periferiemi (na úrovni přímé komunikace s těmito zařízeními) a aplikace, které využívají pouze obecné principy jazyka Java a u kterých nehrozí havárie aplikace z důvodů provedení některé nepovolené operace. Pro tato oddělení se nabízí možnost využití dvou JVM, kterými procesor aJ-80 disponuje.

3.5.3 Garbage Collector

Automatická správa paměti (GC – více viz kapitola 2) je v procesoru aJ-80 implementována jako Java vlákno. Algoritmus GC je základní „mark and sweep“. Díky implementaci jako vlákno může být činnost GC preemptivně přerušena vláknem s vyšší prioritou.

Aby se halda (heap) spravovaná GC nedostala při přerušení práce GC do nekonzistentního stavu, je použita nativní Java synchronizace. Klíčové slovo „synchronized“ je aplikováno na objekt haldy (heap). Při použití

RT vlákna je dobré použít RT vlákno, které nepracuje s haldou (heap), aby nemohlo dojít k inverzi priorit (viz kapitola 2) mezi RT vláknem a vláknem GC.

Činnost GC lze zcela zakázat. Což ale znamená, že nám při programování vzniknou jistá omezení související s tím, že v Jazyce Java nemáme prostředek pro explicitní uvolnění dynamických objektů z paměti. Je tedy nutné sledovat, zda při vývoji nepoužíváme objekty, jejichž implementace využívá práce GC, jako jsou například objekty typu Linked List. Prakticky se zakázání činnosti GC provede v programu JEMBuilder. Typicky se „vypnutí“ GC používá v aplikacích využívajících vícenásobných JVM (MVM), kdy prvnímu JVM zakážeme používat GC, protože v něm běží časově kritický program. V druhém JVM provozujeme zbytek aplikace, tam je GC povolen, protože možné časové prodlevy, způsobené prací GC, neovlivní korektní chod aplikace.

3.6 Vývoj aplikací

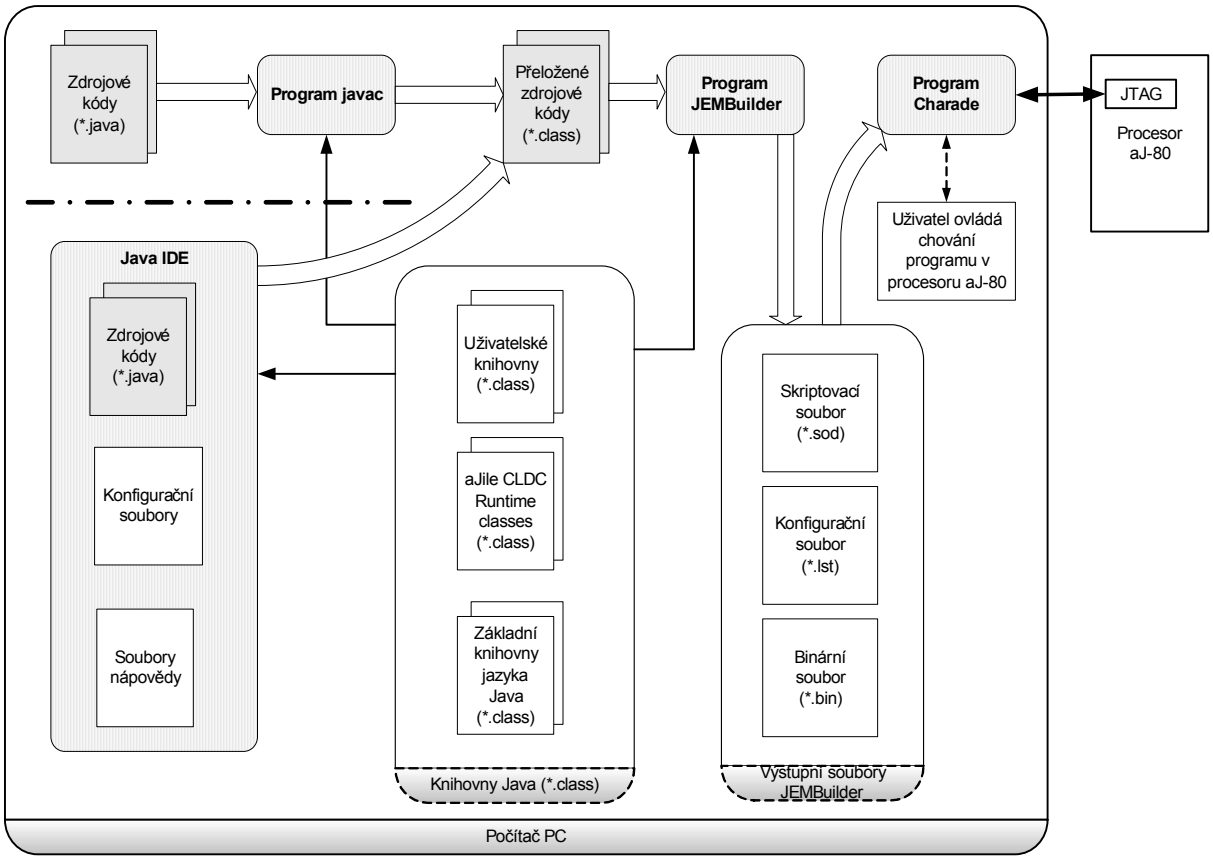
Kompletní řetězec akcí, které je nutné provést při vývoji aplikací pro procesor aJ-80, ilustruje obrázek 3.22. Postup při vývoji aplikací je následující. Nejprve postupujeme, jako bychom programovali klasickou Java aplikaci. Napíšeme zdrojové kódy a provedeme kompilaci. Máme možnost použít program javac (dodávaný s každým prostředím JRE) nebo využijeme výhody některého z Java IDE. Pro kompilaci vyžadují oba způsoby dodání knihoven (soubory obsahující bajtový kód, přípona .class nebo archívy .jar). A to jak základní (definiční) knihovny jazyka Java, tak uživatelské knihovny odkazované ze zdrojových kódů. Knihovny dodávané výrobcem procesoru aJ-80 (aJile Runtime CLDC) můžeme považovat za uživatelské knihovny odkazované ze zdrojových kódů.

Přeložené (zkompilované) kódy se spolu s uživatelskými knihovnami a knihovnami aJile Runtime CLDC předají programu JEMBuilder. Ten vyprodukuje skript pro program Charade a binární soubor, ve kterém se nachází obraz paměti procesoru.

Program Charade komunikuje s procesorem aJ-80 přes standardizované rozhraní JTAG (IEEE1149.1). K počítači, na kterém je spuštěn program Charade, je procesor aJ-80 fyzicky připojen přes paralelní port.

Pomocí programu Charade můžeme zkopírovat obraz paměti z binárního souboru (vytvořeného programem JEMBuilder) do paměti připojené k procesoru aJ-80. Také pomocí něho ovládáme chování programů spouštěných v procesoru. Jako například spouštění, pozastavení, ukončení programů, restart procesoru atd.

Navíc je pomocí programu Charade možné krokovat spouštěné programy na úrovni instrukcí bajtového kódu.



Obrázek 3.22 – Vývoj aplikací pro procesor aJ-80

3.6.1 Program JEMBuilder

Program JEMBuilder je GUI aplikace běžící v prostředí MS Windows. Popis instalace a nastavení je uveden v příloze B.

Vstupem programu jsou přeložené zdrojové kódy (.class), uživatelské knihovny a knihovny dodávané výrobcem (aJile CLDC Runtime classes). JEMBuilder staticky vyhodnotí vztahy mezi nimi a sestaví z nich celek (linkování). Do výsledku nezahrne třídy nebo atributy, na které neexistuje odkaz z uživatelského programu. Tím se snižuje potřeba volné paměti - nepoužité objekty se zbytečně nenahrávají do paměti procesoru.

Program JEMBuilder dále umožňuje nastavení konfigurace procesoru aJ-80 a k němu připojených vnějších pamětí. Výstupem činnosti programu JEMBuilder je několik souborů. Je to binární soubor (.bin) s obrazem paměti, jak je vytvořen ze zdrojových kódů, skriptovací soubor (.sod) pro program Charade, konfigurační soubor (.jcf), soubor s popisem nastavení (.lst) a binární soubory pro jednotlivá JVM (.jvm).

Následuje popis položek menu programu JEMBuilder. Tyto položky se dělí podle jejich příslušnosti do dvou skupin, položky vztahující se k celému projektu (celá aplikace) a položky příslušné jednotlivým JVM.

3.6.1.1 Projekt

Položky v této skupině se týkají globálně celé aplikace.

Výstupní soubory (Output Files)

Nastavujeme adresář, do kterého budou vytvořeny. Dále nastavujeme názvy jednotlivých výstupních souborů.

Paměť (Memory)

Nastavení počáteční adresy a velikosti fyzicky připojených pamětí typu RAM a ROM.

Nastavení PLL (PLL Setup)

Možno nastavit násobící nebo dělící koeficient PLL. Dále můžeme zcela zakázat činnost PLL.

Nastavení pinů (Pin Setup)

Nastavujeme, jaké zařízení bude používat příslušné sdílené fyzické vývody pouzdra procesoru.

Nastavení Chip Select (CS)

Pro každý z pěti CS lze nastavit velikost datové sběrnice a parametry časování jednotlivých připojených pamětí.

3.6.1.2 Jednotlivá JVM

Nastavení položek v této skupině je specifické pro každý ze dvou JVM.

Nastavení hlavní metody (Main)

Nastavujeme jméno (kompletní – s názvem balíku a třídy) hlavní (první spouštěné) metody. Standardně je nastaven název „main()“.

Cesta ke knihovnám (Class Path)

Nastavujeme cestu k přeloženým zdrojovým kódům (.class) a cestu k uživatelským knihovnám.

Nastavení paměti (Memory)

Nastavujeme počáteční adresu a velikost paměti pro JVM. Pro paměť typu RAM (data) a typu ROM (kód). Dále můžeme nastavit velikost haldy (heap), velikost zásobníku pro vlákno metody main, pro ostatní vlákna a pro Garbage Collector. V této položce můžeme také zakázat činnost GC.

Systémové vlastnosti (Properties)

Možno ručně nastavit systémové vlastnosti, jako například oddělovač řádků.

Výpisy (Listing)

Nastavení určující, které z informací mají být zapsány do výstupního souboru (.lst).

Nastavení optimalizátoru (Optimizations)

Možno nastavit chování optimalizátoru – zdali má vypustit nepoužité metody a atributy, převádět pole typu boolean na méně náročnou paměťovou reprezentaci, atd.

Nastavení přerušení (aJ-100 Interrupts)

Nastavení obslužných rutin jednotlivým přerušením, viz. obrázek 3.9.

Přidání zdrojů (Resources)

Možno zahrnout některé dodatečné zdroje, jako obrázky nebo soubory, do výsledného obrazu paměti. Zdroje se přidávají výběrem příslušného souboru, který je obsahuje.

Parametry hlavní metody (Main Args)

Dovoluje vložit parametry hlavní metody (main). Parametry zapíšeme do textového pole oddělené oddělovačem řádků.

3.6.2 Program Charade

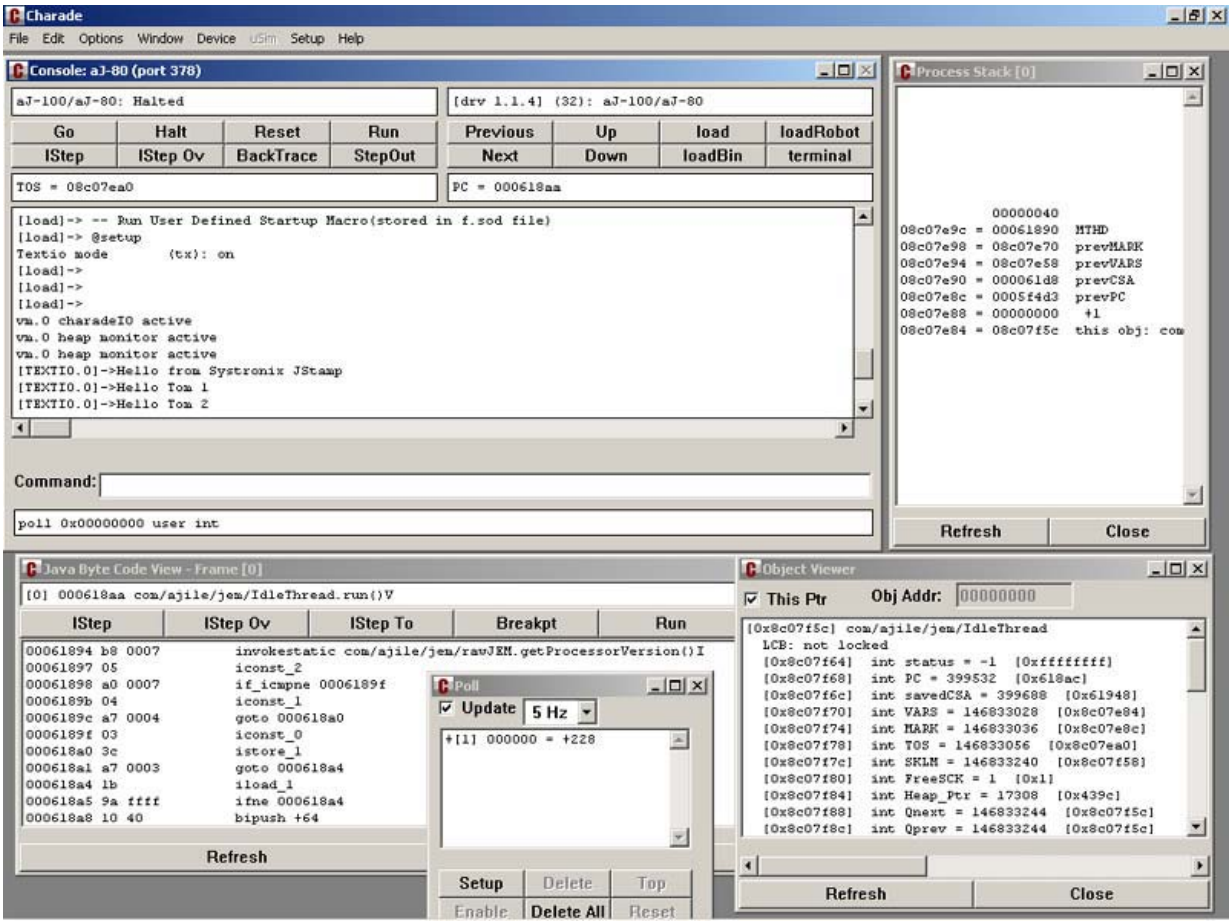
Program Charade je GUI aplikace běžící v prostředí MS Windows. Popis instalace a nastavení je uveden v příloze B. Program Charade umožňuje na základě výstupních souborů programu JEMBuilder transport uživatelských programů z počítače PC do paměti procesoru aJ-80.

Dále umožňuje ovládání vlastností programů spouštěných v procesoru aJ-80 a jejich krokování na úrovni bajtového kódu. Kompletní informace o programu Charade je možno nalézt v [22].

Základní vlastnosti aplikace Charade jsou:

- plná kontrola nad vykonáváním programu v procesoru aJ-80
- přímý přístup do vnitřních registrů a vnější paměti procesoru
- krokování (trasování) na úrovni bajtového kódu
- zobrazení obsahu procesního zásobníku procesoru
- podpora značek „breakpoint“
- uživatelsky definovaná makra a proměnné
- možnost spouštění skriptovacích souborů
- uživatelsky konfigurovatelné prostředí programu
- možnost průběžně zaznamenávat veškeré události do logovacího souboru

Příklad konfigurace oken programu Charade je na obrázku 3.23



Obrázek 3.23 - Program Charade

3.7 Volitelné periferie

K procesoru aJ-80 je možné připojit řadu různých periferních zařízení. Vlastní připojení realizujeme buď pomocí některé z vnitřních periferií procesoru (UART, SPI, GPIO) nebo připojením přes adresovou a datovou sběrnici. Oba způsoby lze kombinovat. Jako příklady připojení vnějších periferních zařízení uvedu připojení k síti standardu Ethernet a možnost komunikace prostřednictvím sběrnice CAN.

Připojení k síti Ethernet poskytuje přímo výrobce procesoru (společnost aJile) na svých vývojových stavebnicích (více v následující kapitole 4). Například stavebnice aJ-100EVB obsahuje obvod CS8900A firmy Cirrus Logic, což je jednočipový kontrolér standardu IEEE 802.3 Ethernet. Ten zajišťuje přijímání a vysílání paketů Ethernet a fyzickou vrstvu připojení k síti Ethernet.

Obvod CS8900A je k procesoru aJ-80 připojen přes adresovou a datovou sběrnici. Přerušení generované obvodem CS8900A je do procesoru přivedeno přes jeden z pinů portu GPIOE. Výrobce navíc poskytuje „ovladače“ k tomuto obvodu, což jsou Java třídy, díky nimž je programátor zproštěn povinnosti řešit komunikaci mezi obvodem CS8900A a procesorem aJ-80. Tato komunikace se pro něj stává transparentní a programátor se

může věnovat implementaci některé z vyšších vrstev komunikačního protokolu.

Popis možností připojení procesoru aJ-80 ke sběrnici CAN je uveden v následující kapitole.

3.7.1 Sběrnice CAN

Sběrnice CAN byla vyvinuta v 80. letech dvacátého století firmou Robert Bosch pro použití v automobilech. Podobně se pracovalo po celém světě na dalších automobilových sběrnicích, např. firma VW spolu s firmou Telefunken pracovala na sběrnici ABUS, od které však VW upustilo a převzalo již v té době silně se prosazující sběrnici CAN. I když byla sběrnice CAN navržena pro automobilový průmysl, začala se hned od začátku prosazovat pro své dobré vlastnosti i v průmyslových aplikacích - v komunikaci řídících systémů, čidel apod.

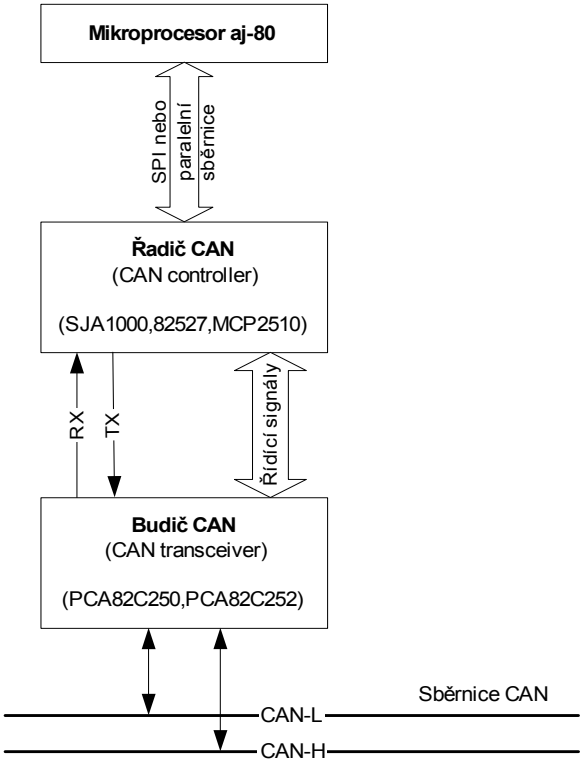
Vzniklo sdružení CiA (Can in Automation), které pro průmyslové aplikace zavedlo standardizaci vyšších vrstev komunikačního protokolu - CAL (CAN Application Layer), CANopen, DeviceNet, ale i např. zapojení konektorů. První nasazení sběrnice CAN v automobilech proběhlo v roce 1991 v Mercedesu Benz třídy S. První nasazení sběrnice CAN ve VW a ŠKODA AUTO bylo v roce 1996 v komfortním systému u vozů Passat a Octavia.

3.7.1.1 Základní obvodové řešení CAN

Pro realizaci řídící jednotky nebo jiného systému se sběrnicí CAN je zapotřebí několik obvodů:

- Mikroprocesor obsluhující události, dávající pokyn pro vysílání zpráv a zpracovávající přijatá data. V našem případě procesor aJ-80.
- Řadič CAN (CAN controller) realizující datovou linkovou vrstvu protokolu CAN (rámce, chybové zabezpečení, arbitráž, filtrování zpráv atd.).
- Budič CAN (CAN transceiver) realizující fyzickou vrstvu protokolu CAN, tj. převod signálu z řadiče CAN (TTL úroveň) do fyzické vrstvy CAN - napětí, diferenciální sběrnice.

Typické propojení sběrnice CAN s mikroprocesorem aJ-80 je na obrázku 3.24.



Obrázek 3.24 - Propojení sběrnice CAN s mikroprocesorem aJ-80

Budič sběrnice (CAN transceiver)

Asi nejvíce používané jsou budiče firmy Philips, existují však i další výrobci. Pozor je třeba dávat na současné použití budičů různých výrobců, nemusí vždy přesně vyhovovat specifikacím nebo mohou způsobovat špatnou odolnost vůči rušení.

- CAN High-speed - PCA82C250 nebo novější náhrada TJA1050, pro nákladní vozy a autobusy PCA82C251.
- CAN Low-speed - PCA82C252 nebo novější náhrada TJA1053 / TJA1054, případně budič Infineon TLE6252G.

Řadič sběrnice (CAN controller)

Řídící jednotky, které nepoužívají jednočipové mikropočítače se zabudovaným CAN řadičem, musí používat externí řadič zapojený k mikroprocesoru. Některé řadiče je možné připojit i malým počtem vodičů - sériovým rozhraním (např. rozhraním SPI) k jednočipovému mikropočítači, který nemá datovou a adresovou sběrnici. Zde jsou některé nejznámější řadiče:

- Intel 82C257 - má i sériové rozhraní
- Philips 82C200 nebo novější náhrada SJA1000
- Microchip MCP2510 - pouze sériové rozhraní
- Infineon (dříve Siemens) 81C90/91 - má i sériové rozhraní
- OKI MSM9225 - má i sériové rozhraní

4 JSTAMP a vývojové stavebnice

V předchozí kapitole byl popsán procesor aJ-80. Společnost aJile produkuje i nástupce tohoto procesoru – procesor aJ-100. Oba typy procesorů mají velmi podobnou konstrukci a z velké části stejné vnitřní obvody. Klíčové odlišnosti těchto procesorů jsou uvedeny v následující tabulce.

Procesor	aJ-80	aJ-100
šíře datové sběrnice	8 bitů	32 bitů
kmitočet systémových hodin	80 MHz	100 MHz
pouzdro procesoru (počet fyzických pinů)	100	176

Tabulka 4.1 - Odlišnosti procesorů aJ-80 a aJ-100

Kompletní informace o procesoru aJ-100 je možno nalézt na [16]. Pro vylepšení vývojářského komfortu a zvýšení prodejního úspěchu se procesory aJ-80(100) doplňují některými obvody, jako jsou paměti, řadič sítě Ethernet, různá napěťová přizpůsobení, řadiče klávesnice nebo displeje. Tyto komplety nazýváme vývojové stavebnice (Development Kit).

Výrobce procesorů aJ-80(100), společnost aJile, produkuje vývojovou stavebnici aJ-100EVB, která je postavena na procesoru aJ-100.

Společnost Systronixs poskytuje funkční zařízení JSTAMP, které obsahuje procesor aJ-80 a externí paměť. JSTAMP je základem vývojové stavebnice firmy Systronixs – JSTAMP Development Board.

Kompletní mikropočítač založený na procesoru aJ-100 vyrábí společnost Aidonic s názvem JPC1001.

4.1 JSTAMP

4.1.1 Společnost Systronixs

Výrobce zařízení JSTAMP, společnost Systronixs, sídlí ve městě Salt Lake City v USA. Má čtyřicetileté zkušenosti s vývojem kritických RT zabudovaných systémů. Provozuje kvalitně zpracované www stránky (viz [23]) s množstvím materiálů, vztahujících se k jejím výrobkům.

Tato společnost založila diskusní skupinu (odkaz je také možné nalézt na [23]), její zaměstnanci tam zodpovídají dotazy vznesené uživateli.

4.1.2 Konstrukce

Zařízení JSTAMP obsahuje kromě procesoru aJ-80 i další obvody, což dohromady tvoří samostatně fungující celek.

JSTAMP obsahuje procesor aJ-80, paměť (připojenou k aJ-80 přes EBI), krystal, obvody přizpůsobení a ochrany napájení a obvod zajišťující resetovací logiku. Zařízení JSTAMP existuje ve dvou provedeních, lišících se velikostí paměti typu FLASH.

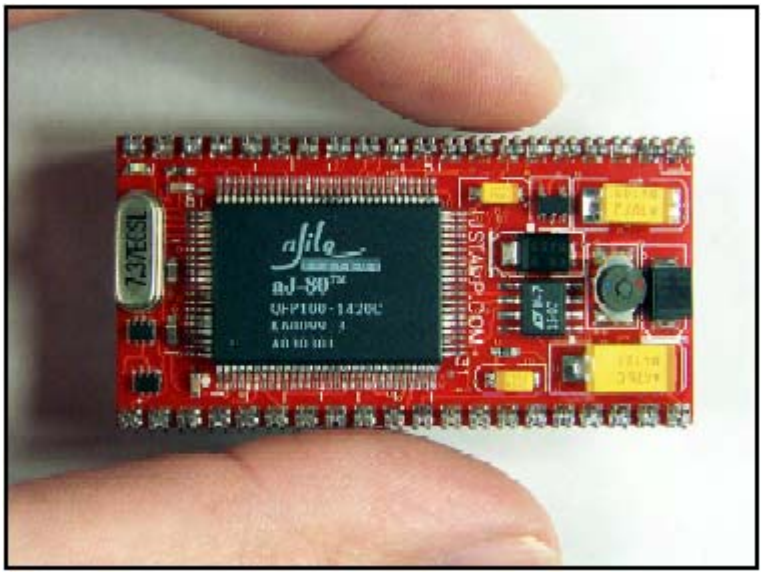
Následující tabulka popisuje velikosti jednotlivých typů pamětí u variant zařízení JSTAMP.

komplet/typ paměti	velikost SRAM (70ns)	velikost FLASH (90ns)
JSTAMP	512 kB	512 kB
JSTAMP +	512 kB	2MB

Tabulka 4.2 - Velikost paměti zařízení JSTAMP

Fyzické provedení zařízení JSTAMP je jako deska plošného spoje se čtyřiceti vývody (piny). Rozteč pinů je shodná se standardizovaným pouzdrem typu DIP40, liší se pouze vzdáleností obou řad pinů.

Vzhled zařízení ilustruje obrázek 4.1. Tento obrázek je dvojnásobně větší než realita.



Obrázek 4.1 - Zařízení JSTAMP

Popis vývodů JSTAMP spolu s povolenými napěťovými a proudovými úrovněmi je uveden v příloze E.

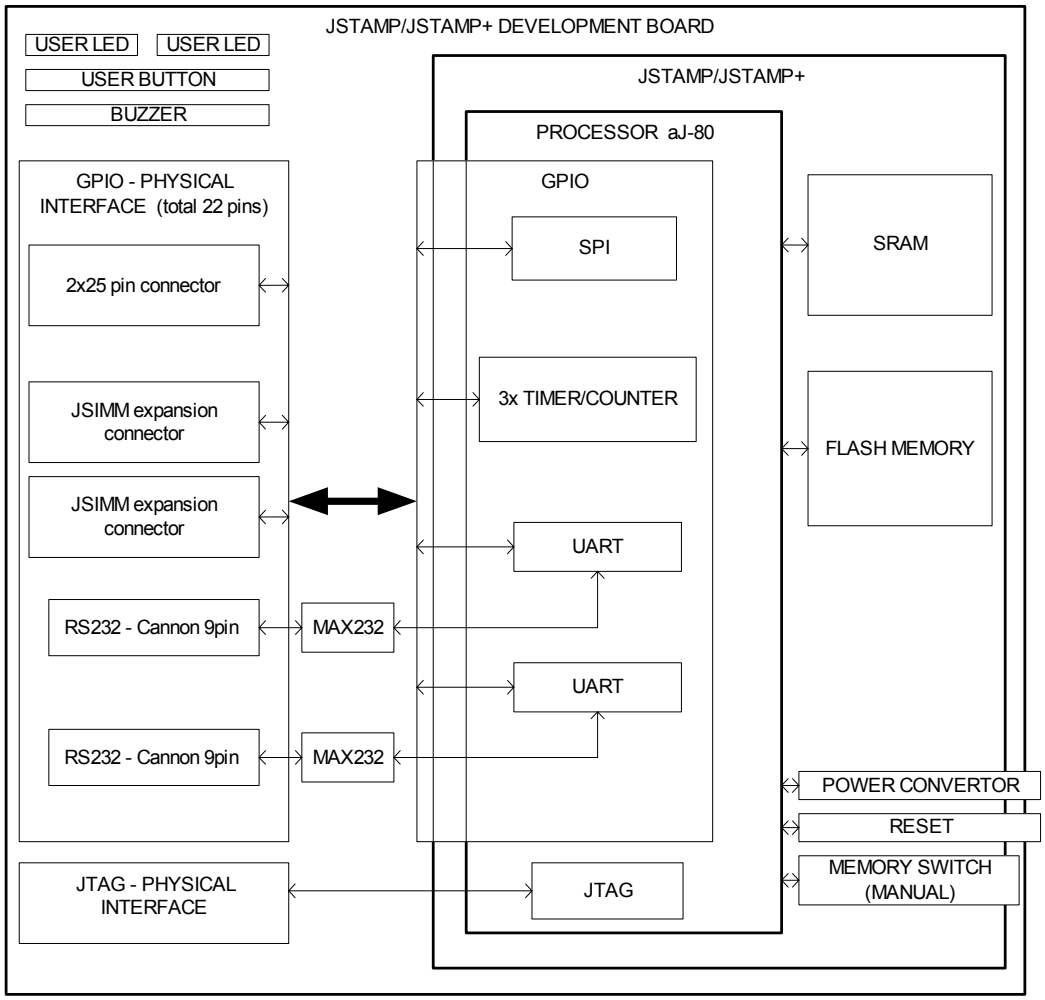
Napájení zařízení JSTAMP je možné stejnosměrným napětím v rozsahu 5 až 14V nebo napětím 3,3V s možnou odchylkou ±10%. Spotřeba energie se pohybuje od 45 do 300mW v závislosti na taktovacím kmitočtu procesoru a způsobu napájení (při „přesném“ napájení 3,3V je spotřeba nižší). Teplota pracovního prostředí má být v rozmezí 0 až 70°C.

4.2 Vývojové stavebnice

4.2.1 JSTAMP DEVELOPMENT BOARD

Vývojová deska společnosti Sytronixs (JSTAMP Development Board – JDB) rozšiřuje zařízení JSTAMP o některé dodatečné obvody a hlavně poskytuje množství fyzických konektorů pro připojení dalších zařízení.

Deska JDB obsahuje zařízení JSTAMP, konektory pro zajištění dostupnosti GPIO, 2x obvod MAX232 jako přizpůsobení napěťových úrovní pro standard RS232, 2x 9pin konektor Cannon, port pro připojení JTAG. Navíc jsou na desce JDB součástky, které je možno jednoduše (pomocí zkratovacích propojek) připojit ke zařízení JSTAMP. Je to reproduktor, tlačítko a 2x dioda LED. Blokové schéma desky JDB je na obrázku 4.2.



Obrázek 4.2 - Blokové schéma desky JDB

Fotografie desky JDB s popisem jednotlivých komponent je na obrázku 4.3. Význam jednotlivých popisků označených písmeny A – Q je následující:

A) soubor vývodů, kam jsou vyvedeny následující signály: SS napětí 5V, zem, signály RST a CRST

B) při propojení zkratovací propojky je napětí 5V přivedeno i do konektoru označeného písmenem O

C) 2x konektor typu Jsimm. Je to konektor pro připojení zařízení formátu SimmStick, což jsou standardizovaná zařízení, jako například řadič displeje, klávesnice, I/O porty, atd. Více viz [24].

D) konektory Cannon, vidlice 9pin standardu RS232

E) propojením zkratovací propojky je reproduktor připojen přes tranzistor k vývodu IOA1

F) konektory napájecího napětí v rozsahu 6 – 20V

G) zelená dioda LED indikující připojení napájecího napětí

H) zelená dioda LED připojená přes invertor k vývodu IOA0

I) propojením/nepropojením těchto vývodů určujeme, zda bude základní adresa 0 přiřazena paměti SRAM nebo FLASH. Při propojení je základní adresa přiřazena paměti SRAM

J) vývody tlačítka a dvou LED diod k uživatelskému využití

K) uživatelské diody (2x žlutá dioda LED), pro rozsvícení je nutné přivést na vývod 3 nebo 4 konektoru J logickou 0

L) uživatelské tlačítko, při stisknutí se na vývodu č. 1 konektoru J objeví signál logické 0

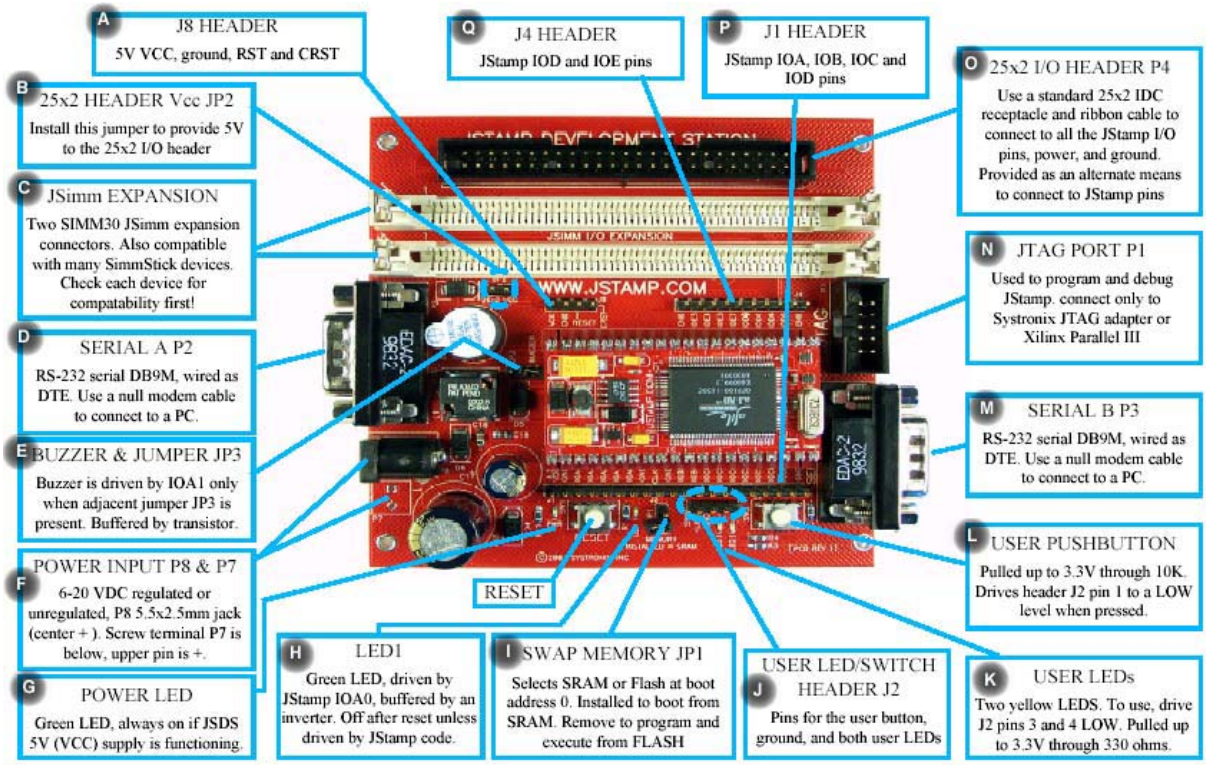
M) konektory Cannon, vidlice 9pin standardu RS232

N) konektor rozhraní JTAG

O) standardní patice 25 x 2, kam jsou vyvedeny všechny dostupné signály a vstupně/výstupní vývody

P) konektor obsahující vývody portů IOA, IOB, IOC a IOD

Q) konektor obsahující vývody portů IOD a IOE



Obrázek 4.3 - Popis vývojové desky JDB

Vývojová deska JDB se dodává samostatně nebo je možno zakoupit celou sadu (JSTAMP Development Kit) kompletního řešení vývoje aplikací pro procesor aJ-80. Sada obsahuje vývojovou desku JDB, JTAG adaptér pro připojení k paralelnímu portu PC, CD ROM s programy JEMBuilder a Charade a základní dokumentací, napájecí adaptér a prototypovou desku Jsimm. Cenová kalkulace sady JSTAMP Dev. Kit je uvedena v tabulce 4.3. Ceny jsou platné k 20. 12. 2003, aktuální ceník je možno nalézt na [23].

	Cena v USD
Dev. kits	
JSTAMP DEV KIT	299,00
JSTAMP+ DEV KIT	399,00
Poštovné	
FedEx International (except Canada, Mexico, & east Eurasia)	40,00
FedEx International (eastern Eurasia, Africa)	70,00
Global Priority or Air Mail (4 lbs max - NO INSURANCE - you accept all risk of loss or damage)	20,00

Tabulka 4.3 - Cenová kalkulace sady JSTAMP Dev. Kit

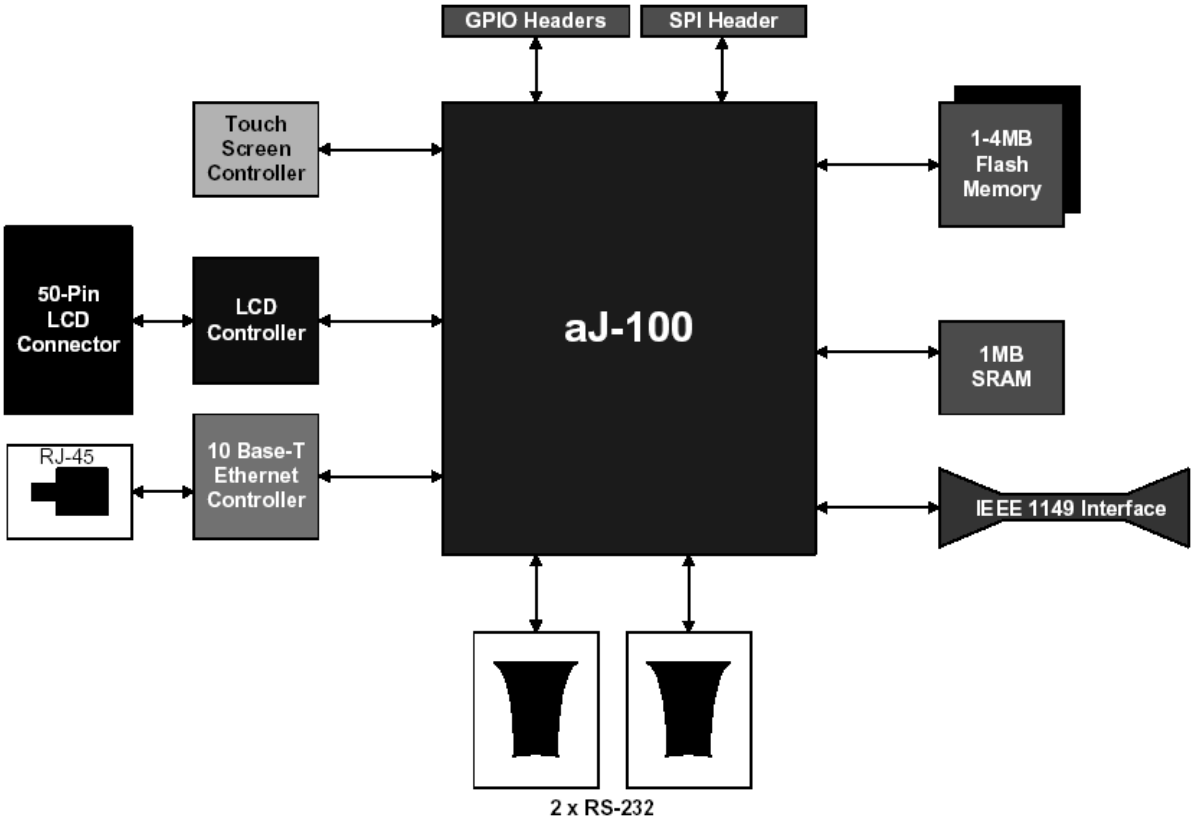
4.2.2 aJile aJ-100EVB

Výrobcem vývojové desky aJ-100EVB je společnost aJile – výrobce procesoru aJ-80.

Deska aJ-100EVB obsahuje:

- procesor aJ-100
- paměť typu SRAM, velikost 1 MB (aJ-100EVB) nebo typu SDRAM, velikost 16 MB (aJ-100EVBm)
- paměť typu FLASH, velikost 4 MB
- kontrolér pro připojení dotykové obrazovky
- kontrolér pro připojení LCD zobrazovače – obvod Epson S1D13706
- obvod pro připojení k síti Ethernet – Cirrus Logic CS8900

Blokové schéma vývojové stavebnice aJ-100EVB je na obrázku 4.4. Podrobnější informace o tomto produktu je možno nalézt na [24].



Obrázek 4.4 - Blokové schéma vývojové desky aJ-100EVB

4.2.3 Aidonic JPC1001

Výrobek společnosti Aidonic JPC1001 je kompletní mikropočítač založený na procesoru aJ-100.

JPC1001 kromě aJ-100 obsahuje také:

- paměť SRAM, velikost 1 MB
- paměť FLASH, velikost 4 MB
- kontrolér pro připojení k síti Ethernet
- dotykový LCD displej – barevný TFT, velikost 5,7´´, rozlišení 320x240, povrch displeje je vodo a prachotěsný

Kompletní informace je možno nalézt na [25].

5 Testování vlastností aJ-80 (JStamp)

V této kapitole se snažím srozumitelně popsat testy funkčnosti provedené s procesorem aJ-80, umístěným v zařízení JSTAMP. Ke každé z podkapitol se vztahuje zdrojový kód. Zdrojové kódy jsou uvedeny v příloze D (kapitola Testy vlastností aJ-80).

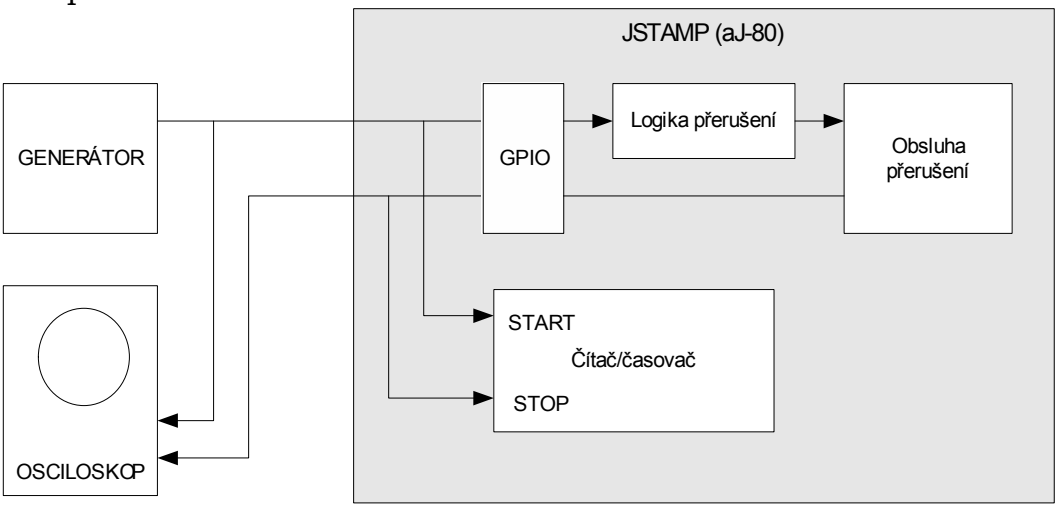
Na uvedených testech funkčnosti jsem si také osvojil programové konstrukce, které jsem použil při tvorbě výsledné aplikace, jejíž popis je uveden v následující kapitole.

Při programování testů byly použity oba dva způsoby programového ovládání vnitřních periferních obvodů procesoru. Tedy použití Java třídy (ovladače), vztahující se k dané periferii a přímý zápis do konfiguračních registrů příslušného zařízení.

5.1 Doba reakce na vnější událost

Účelem tohoto testu bylo zjistit časovou prodlevu mezi vznikem externí události, která vyvolá přerušení procesoru a jejím obsloužením příslušnou rutinou. Prakticky byla tato zkouška realizována přivedením signálu obdélníkového průběhu na jeden ze vstupních portů procesoru. Tento signál periodicky vyvolává přerušení. Tato přerušení jsou obsluhována přiřazenou metodou, která pouze přečte úroveň signálu na vstupu, tu invertuje a invertovanou ji pošle na výstupní pin portu.

Blokové schéma měření je na obrázku 5.1. Časová prodleva je měřena vnitřním čítačem/ časovačem, jehož přesnost není ovlivňována prováděním programu. Pro kontrolu, zda je čítač správně nakonfigurován a zda poskytuje korektní údaje, měříme časovou prodlevu navíc pomocí osciloskopu.



Obrázek 5.1 - Blokové schéma měření časové prodlevy

Obslužná rutina je příslušnému přerušení přiřazena pomocí programu JEMBuilder – viz obrázek 3.9. V našem případě je to metoda s názvem myInterruptEvent. Kromě měření doby reakce při běhu jednoho (hlavního) vlákna, bylo totožné měření provedeno při běhu dvou, deseti, sta a sto padesáti vláken současně. Výsledky měření jsou patrné z tabulky 5.1.

Počet vláken (kromě hlavního)	**0**	**2**	**10**	**100**	**150**
Číslo měření	Doba odezvy [μs]	Doba odezvy [μs]	Doba odezvy [μs]	Doba odezvy [μs]	Doba odezvy [μs]
1	7,23	10,74	10,80	11,56	12,31
2	7,53	10,47	10,58	11,13	11,82
3	7,12	10,63	10,85	12,15	11,93
4	7,00	10,58	10,96	11,77	12,98
5	6,78	10,42	10,91	12,99	11,94
6	6,72	10,58	10,72	11,88	12,33
7	6,84	10,48	10,75	12,16	12,57
Průměrná doba odezvy [μs]	7,03	10,56	10,80	11,95	12,27

Tabulka 5.1 - Výsledky měření časové prodlevy

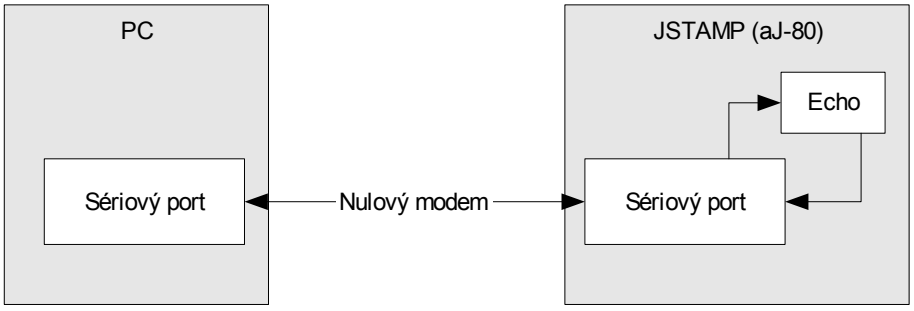
Program pro testování se spouští metodou main() třídy IntTestSharp100, číslice 100 napovídá, že se jedná o případ se současným během sta vláken.

Metoda nejprve spustí inicializační metodu setupIrq(), která nastaví příslušné piny portu GPIO a nastaví typ spouštění přerušení.

Dále jsou metodou main() spouštěny statické metody třídy TmrTestDemo. Metoda tmrSetup() nastaví počáteční konfiguraci čítače/časovače, metoda setupMeasure() nastaví počáteční podmínky pro měření. Poté jsou v metodě main() vytvořena a spuštěna vlákna. Následuje smyčka, ve které jsou v pětisekundových intervalech zobrazovány výsledky posledního měření.

5.2 Komunikace přes sériový port

Abych otestoval komunikaci systému (počítač PC) se systémem JSTAMP, použil jsem program, jehož jedna část (klient) běží v počítači PC a druhá část (server) je v paměti zařízení JSTAMP. Obě zařízení jsou fyzicky spojena nulovým modemem (Null Modem). Blokové schéma komunikace je na obrázku 5.2.



Obrázek 5.2 - Blokové schéma sériové komunikace

Klientský program vysílá ASCII znaky po sériové lince do zařízení JSTAMP. Program JSTAMP pouze přečte příchozí znak, vytiskne ho na obrazovku a odešle ho zpět do počítače PC.

Program v PC používá pro práci knihovny z balíku `javax.comm`. Což jsou knihovny pro práci se sériovým portem na obecné úrovni. Je v nich použita forma abstrakce tak, aby programátor nemusel řešit sériovou komunikaci na nejnižší úrovni a komunikaci s hardware, který sériovou komunikaci zajišťuje. Komunikace s hardware je navíc platformně závislá, takže pro každý typ hardware je dodávána run-time knihovna, jejíž kód je vyvolán metodami tříd z balíku `javax.comm`. Tyto run-time knihovny musejí být součástí distribuce programů využívajících balík `javax.comm`. V tomto ohledu ztrácí Java jednu ze svých předností, platformní nezávislost.

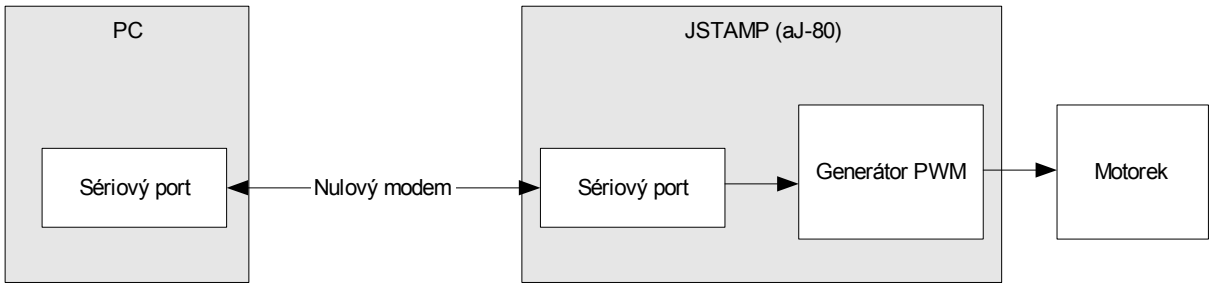
Program v zařízení JSTAMP používá pro komunikaci přes sériový port obdobné knihovny z balíku `javax.comm`. Funkci přídavných run-time knihoven suplují takzvané ovladače (Driver). Mají v podstatě stejnou funkci jako run-time knihovny, liší se jen způsobem přiřazení. Například na počítačích PC, v prostředí MS Windows, jsou run-time knihovny svázané s balíkem `javax.comm` ve formě souboru typu DLL (Dynamic Linked Library), jehož umístění v souborovém systému počítače je předáno jako parametr stroji JVM.

Přiřazení ovladačů se v systému procesoru aJ-80 provede pomocí programu JEMBuilder.

Program v JSTAMP pracuje následovně. Metoda `main` třídy SerialEcho nejprve vytvoří instanci sama sebe (třídy SerialEcho). Poté spustí metodu `initializeSerialPort`. Ta provede inicializaci sériového portu. Vytvoří instanci třídy SerialPort a pomocí metody SetSerialPortParams nastaví parametry přenosu, jako je rychlost přenosu, počet datových bitů, počet stopbitů a typ parity. Metoda `initializeSerialPort` dále pomocí metody `addEventListener` přiřadí příslušné události sériového portu (Serial Event) jejího posluchače. Tím je metoda `SerialEvent`, protože třída `SerialEcho` implementuje rozhraní `SerialPortEventListener` z balíku `javax.comm`. Systém tedy při vzniku události od sériového portu zavolá metodu SerialEvent. Tato metoda zjišťuje, zda jsou k dispozici nějaká přijatá data. Pokud ano, pak volá metodu echo, která přečte data ze vstupního proudu (Input Stream), vyčká půl sekundy, zapíše přijaté znaky do výstupního proudu (Output Stream) a vytiskne je na obrazovku.

5.3 Generování PWM

Účelem tohoto testu bylo zjistit, zda je vestavěný generátor modulace PWM vhodný pro řízení rychlosti otáčení připojeného školního zařízení obsahujícího stejnosměrný motorek. Aby bylo možno sledovat dosahované výsledky, je v tomto testu také měřen počet otáček motorku. Chování modulace se ovládá z nadřazeného systému přes sériové rozhraní. V této kapitole se zaměřím pouze na část generování PWM, protože ostatní části této aplikace jsou popsány v kapitole 6. Blokové schéma zapojení testu generování PWM je na obrázku 5.3.



Obrázek 5.3 - Test generování PWM

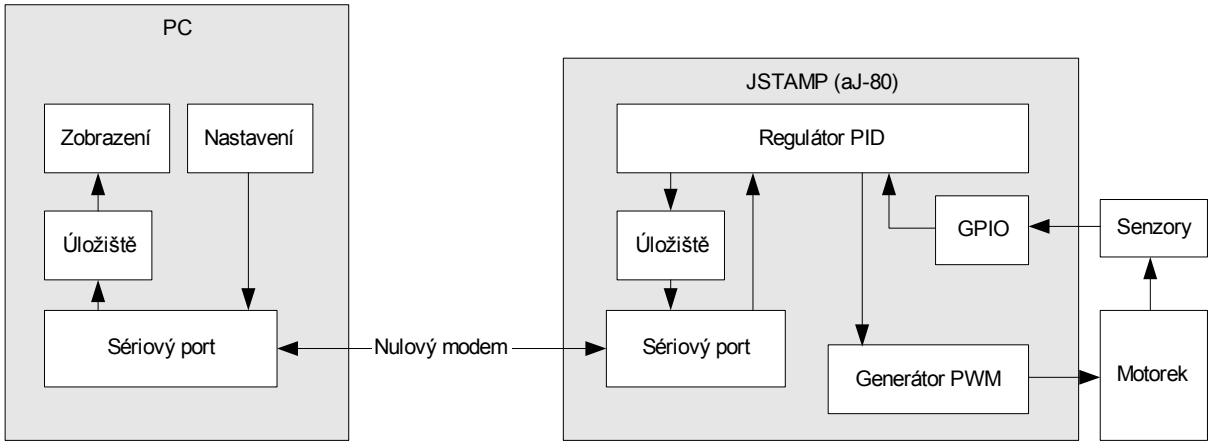
Metoda tmrSetup třídy PWMTestDemo3 nejprve pomocí zápisu do registrů bloku GPTC nastaví propojení jednotlivých částí GPTC a určí, že se bude jednat o generování PWM. Poté nastaví výchozí parametry pulsně šířkové modulace (PWM). Zápisem do registru GPTC_PRLR je nastaven dělící koeficient děliče hodinového signálu.

Registr GPTC1_PRLR udává, kolik pulsů hodinového signálu připadá na jednu periodu modulace PWM. V registru WF1_WPR je počet pulzů hodinového signálu, po kterém se mění logická hodnota signálu PWM v rámci jedné periody. Tento registr tedy určuje střídu signálu PWM.

Metoda SetDutyCycle třídy PWMTestDemo3, jejímž vstupním parametrem je požadovaná střída generovaného signálu PWM v procentech, přepočítává procentuální vyjádření střídy na konkrétní hodnotu, která bude zapsána do registru WF1_WPR, podle hodnoty v registru GPTC1_RLR. Takto nastavená střída generovaného signálu PWM se samovolně nemění, změnit ji můžeme opětovným zavoláním metody SetDutyCycle s odlišným parametrem.

6 Popis aplikace

V průběhu seznamování se s JSTAMP bylo upřesněno zadání demonstrační úlohy, která měla ukázat reálné možnosti procesoru aJ-80, potažmo zařízení JSTAMP. Úkolem bylo vytvořit jednoduchý řídící systém motorku, provozovaný v zařízení JSTAMP. Pomocí nadřazeného systému (v našem případě počítač PC) mělo být možné nastavovat parametry regulace a graficky zobrazovat průběhy jednotlivých významných signálů. Blokové schéma celého systému je na obrázku 6.1.



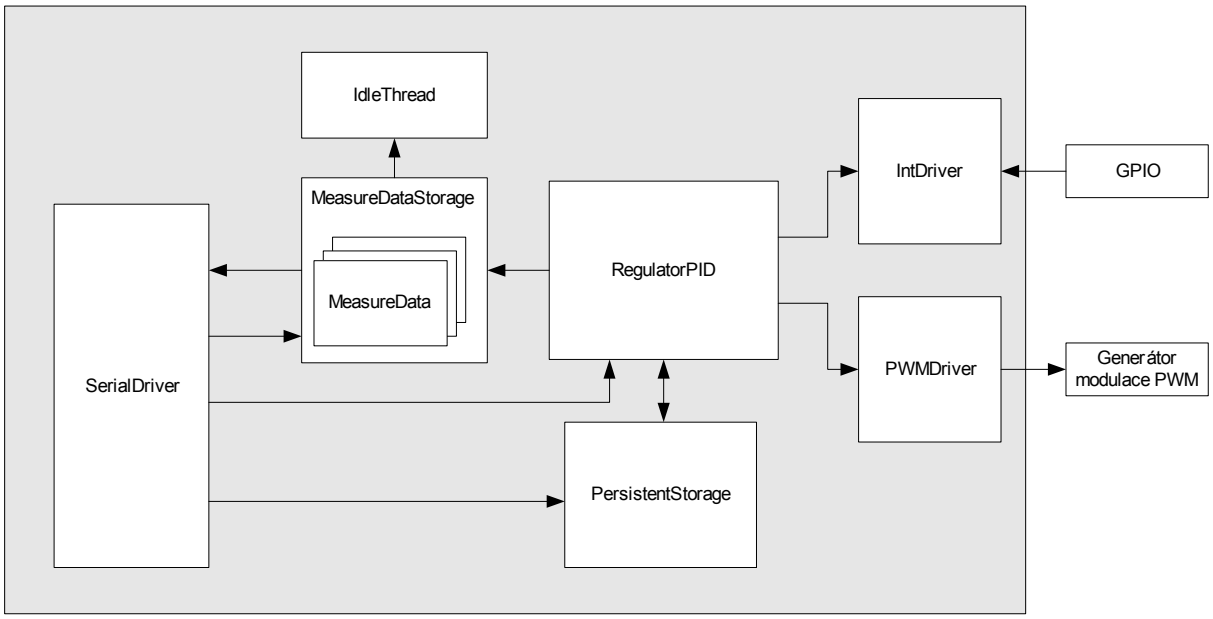
Obrázek 6.1 - Schéma řídícího systému

V zařízení JSTAMP je implementován programový regulátor typu PID. Ten na základě údajů o požadované hodnotě (nastavené pomocí nadřazeného systému) a skutečné hodnotě (získané měřením) nastavuje střídu generovaného signálu PWM a ukládá údaje do úložiště. Přes sériový port komunikuje s počítačem PC, odkud přijímá nastavení parametrů regulace a další údaje, které se mohou nastavovat. Program v počítači PC přijímá data o průběhu sledovaných signálů a zobrazuje je ve formě grafů.

Zdrojové kódy nejsou uvedeny v přílohách z důvodu jejich rozsáhlosti. Zdrojové kódy jsou na přiloženém CD. Struktura CD je popsána v příloze C.

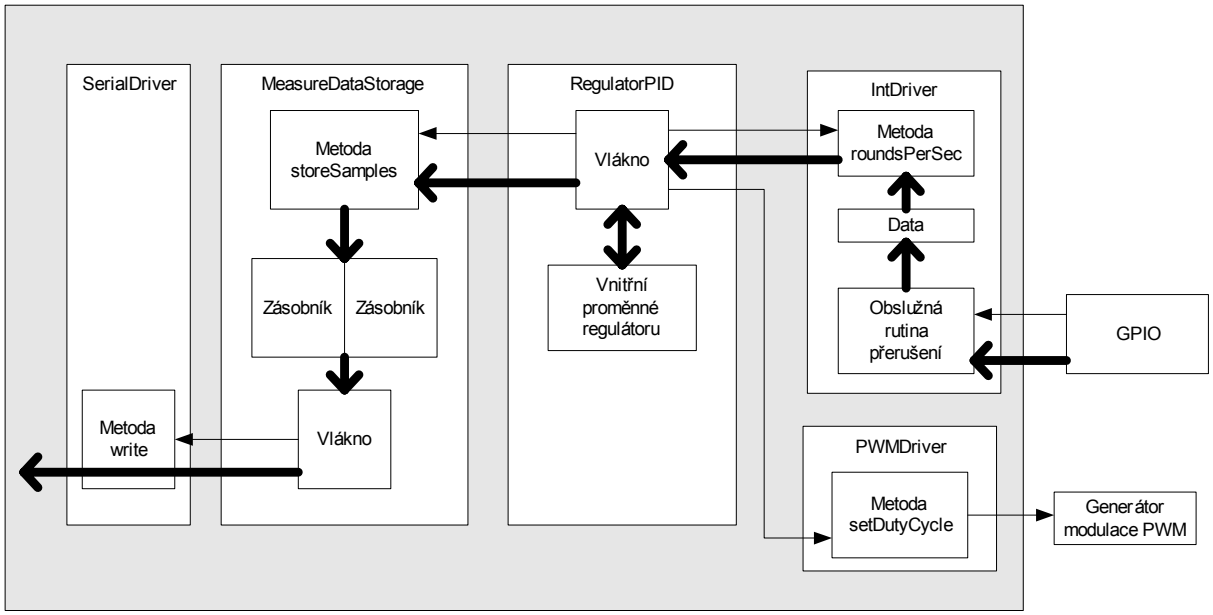
6.1 Část v JSTAMP

Blokové schéma s naznačenými řídícími toky aplikace v JSTAMP je znázorněno na obrázku 6.2.



Obrázek 6.2 - Blokové schéma aplikace v JSTAMP

Hlavní výkonnou částí aplikace je třída RegulatorPID, která na základě informace o změřeném počtu otáček a na základě nastavení vnitřních parametrů posílá požadavky třídě PWMDriver. Třída PWMDriver komunikuje s generátorem modulace PWM a na základě požadavků od třídy RegulatorPID nastavuje střídu signálu PWM. Do objektu třídy MeasureDataStorage se dočasně ukládají sledovaná data regulace třídy RegulatorPID. Tato dočasně uschovávaná data se pomocí třídy SerialDriver transportují do počítače PC. Třída SerialDriver navíc přijímá požadavky z počítače PC a ty poté distribuuje mezi jednotlivé třídy. Třída PersistentStorage umožňuje trvale ukládat vnitřní parametry třídy RegulatorPID do paměti FLASH.



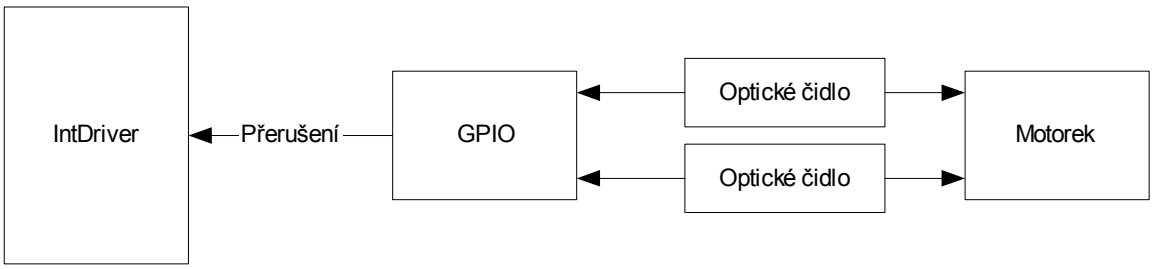
Obrázek 6.3 - Popis vláken a synchronizace

Na obrázku 6.3 je znázorněno schéma aplikace z pohledu datových toků a činnosti vláken. Datové toky jsou na obrázku vyznačeny tučnými čarami. Nejvyšší prioritu má obslužná metoda přerušení třídy IntDriver. Vlákno třídy RegulatorPID má prioritu MAX_PRIORITY-2. Priorita vlákna třídy MeasureDataStorage je nastavena na hodnotu NORM_PRIORITY+1. Synchronizaci přístupu ke sdíleným objektům musíme zajistit na dvou místech aplikace. Je to sdílená datová oblast třídy IntDriver a dvojitý zásobník třídy MeasureData Storage. V prvním případě k datovým položkám přistupuje jak obslužná rutina přerušení, tak metoda roundsPerSec, která je volána vláknem třídy RegulatorPID. Zde je „synchronizace“ zajištěna tím, že datová položka (údaj o aktuálním počtu otáček za sekundu) je ukládána do registru procesoru. Čtení i zápis z registru je atomická (nedělitelná) operace, proto se nemohou data dostat do nekonzistentního stavu. Nekonzistence by mohla vzniknout, pokud by vlákno s nízkou prioritou měnilo datové položky způsobem, který je dělitelný a při své činnosti by bylo přerušeno díky připravenosti k běhu vlákna s vyšší prioritou. Potom bychom museli uvažovat o synchronizaci například použitím klíčového slova „synchronized“ nad sdíleným objektem.

U dvojitého zásobníku třídy MeasureDataStorage je exkluzivita přístupu ke sdílenému zdroji řešena programově pomocí mechanismu semaforů. Ty určují, do kterého ze dvou zásobníků zapisuje metoda StoreSamples a ze kterého čte vlákno třídy MeasureDataStorage

6.1.1 Třída IntDriver

Na obrázku 6.4 je blokové schéma obvodů, s nimiž spolupracuje třída IntDriver. Optická čidla snímají pohyb kotouče připojeného k ose motorku. Kotouč obsahuje 16 přechodů mezi průhlednými a neprůhlednými částmi. Polohy optických čidel jsou fázově posunuty tak, aby při otáčení kotouče docházelo ke změně mezi průhlednou a neprůhlednou částí pouze u jednoho čidla. Je tak možno zjišťovat směr otáčení a odhalovat chyby měření. Protože sekvence změn průhledných a neprůhledných částí se nemění.



Obrázek 6.4 - Měření otáček motorku

Signál z optických snímačů je přiveden fyzickými vodiči do bloku GPIO. Tam je generováno přerušení procesoru v taktu změn jednotlivých průhledných a neprůhledných částí kotouče. Přerušení je obslouženo metodou IntDriver.InterruptEvent.

Metoda InterruptEvent kontroluje, zda je kombinace vstupních hodnot přípustná. Pokud není, pak inkrementuje počítadlo chyb. Pokud je kombinace přípustná, zvýší se o jednu počítadlo platných pulzů. Pokud tento počet dosáhne šestnácti (jedna kompletní otáčka), je pozastaven čítač/časovač a je uložena jeho aktuální hodnota. Poté je čítač/časovač vynulován a znovu spuštěn.

Třída IntDriver dále poskytuje metodu roundsPerSec, která na základě aktuální uložené hodnoty čítače/časovače a na základě hodnot registrů bloku GPTC vrací aktuální hodnotu počtu otáček za sekundu.

6.1.2 Třída PWMDriver

Třída PWMDriver komunikuje s generátorem modulace PWM. Pomocí metody setDutyCycle nastavujeme střídu modulace a to v rozsahu 1/5000 až 4999/5000. Metoda tmrAllSetup se používá k inicializaci všech tří čítačů/časovačů bloku GPTC.

6.1.3 Třída RegulatorPID

Tato třída implementuje numerický regulátor PID. Na základě měřeného počtu otáček (Y) a požadované hodnoty otáček (W) počítá hodnotu akčního zásahu (U). Akční zásah se počítá pomocí přírůstkového (rychlostního) tvaru rovnic pro PID regulátor:

$$\Delta u(k) = u(k) - u(k-1) = r_0 \left\{ e(k)\left[1 + \frac{T_v}{2T_i} + \frac{T_d}{T_v}\right] + e(k-1)\left[\frac{T_v}{2T_i} - 1 - 2\frac{T_d}{T_v}\right] + e(k-2)\frac{T_d}{T_v} \right\},$$

kde e(k) = w(k) – y(k), r_0 je zesílení proporcionální složky, T_d je časová konstanta derivační složky, T_i je časová konstanta integrační složky a T_v je perioda vzorkování.

Výpočet akčního zásahu a volání metody setDutyCycle třídy PWMDriver se provádí v samostatném vlákně s vysokou prioritou. Časový interval mezi jednotlivými výpočty a akčními zásahy je dán parametrem perioda vzorkování (Sample Period).

Při vlastní implementaci vlákna jsem uvažoval o použití speciálního typu vlákna – periodického vlákna (třída PeriodicThread). Při vývoji se ale ukázalo, že periodickému vláknu nelze při jeho běhu dynamicky měnit periodu spouštění. Vlákno by se muselo zrušit a následně vytvořit jiné s novou dobou periody. Proto jsem nakonec ponechal stávající řešení pomocí klasického Java vlákna, kde nám sice vzniká chyba, protože doba provádění kódu vlákna není nezanedbatelná. Ale protože se jedná jen o několik instrukcí, pohybuje se doba provádění kódu vlákna pod hranicí 30 μs. Doba vzorkování je minimálně 1ms, takže chyba vzniklá prováděním kódu není velká a lze ji, podle mého názoru, zanedbat.

V této ukázkové aplikaci jsem preferoval demonstraci různých vlastností zařízení JSTAMP, proto jsem použil řešení umožňující měnit při běhu aplikace periodu vzorkování regulátoru. Pokud by se jednalo o reálný

řídící systém, tak by požadavky na něj byly zřejmě opačné. Tedy mít přesnou periodu vzorkování i za cenu toho, že půjde změnit jen pokud bude aplikace pozastavena. Příklad použití periodického vlákna je na přiloženém CD v projektu PeriodicThread.

Třída RegulatorPID kromě výše uvedených metod poskytuje i soubor metod pro nastavování a čtení vnitřních parametrů regulátoru.

6.1.4 Třída MeasureDataStorage

Třída MeasureDataStorage slouží k dočasnému uložení dat, která jsou transportována přes sériový port do nadřazeného počítače PC. Třída obsahuje implementaci dvou zásobníků typu FIFO s indikací přetečení. Velikost zásobníku je nastavitelná. Výkonné vlákno třídy RegulatorPID volá při každém průchodu hlavní smyčkou metodu storeSamples třídy MeasureDataStorage. Ta má jako parametr objekt třídy MeasureData. Třída MeasureData je takovou pseudo třídou, protože obsahuje pouze atributy a nemá žádné metody. Je tedy jakousi formou implementace typu struktura do jazyka Java. Konkrétně třída MeasureData obsahuje datové položky W, Y, U a IdleThreadCount (viz třída IdleThread).

Výše zmíněná metoda storeSamples třídy MeasureDataStorage uloží položky předaného objektu MeasureData do právě aktivního zásobníku FIFO. Druhý „neaktivní“ zásobník FIFO je kompletně přečten vláknem, které je součástí třídy MeasureDataStorage. Toto vlákno komunikuje s třídou SerialDriver, která posílá přečtené údaje do nadřazeného počítače PC. Po naplnění aktivního zásobníku plněného pomocí metody storeSamples se úlohy obou zásobníků FIFO vymění. Toto řešení pomocí dvou zásobníků je z důvodu nutnosti oddělení častého zapisování jednotlivých dat do úložiště MeasureDataStorage vláknem třídy RegulatorPID a méně častého, ale blokového odesílání dat do nadřazeného počítače PC pomocí třídy SerialDriver.

6.1.5 Třída SerialDriver

Pomocí třídy SerialDriver jsou spojeny objekty aplikace v JSTAMP s nadřazeným počítačem PC. Třída SerialDriver obsahuje metodu initializeSerialPort, která inicializuje sériový port. Metoda je totožná s inicializační metodou třídy SerialPort, jejíž popis je uveden v kapitole 5.2.

Třída SerialDriver posílá data obsažená v objektu třídy MeasureData do počítače PC a také z něj přijímá požadavky pro jednotlivé třídy aplikace v JSTAMP. Detekce typu požadavku a jeho distribuce je prováděna metodou serialHandler.

6.1.6 Třída PersistentStorage

Třída PersistentStorage pracuje se souborovým systémem v paměti FLASH. Pomocí metody writeFlash jsou zapsány parametry regulátoru (třída

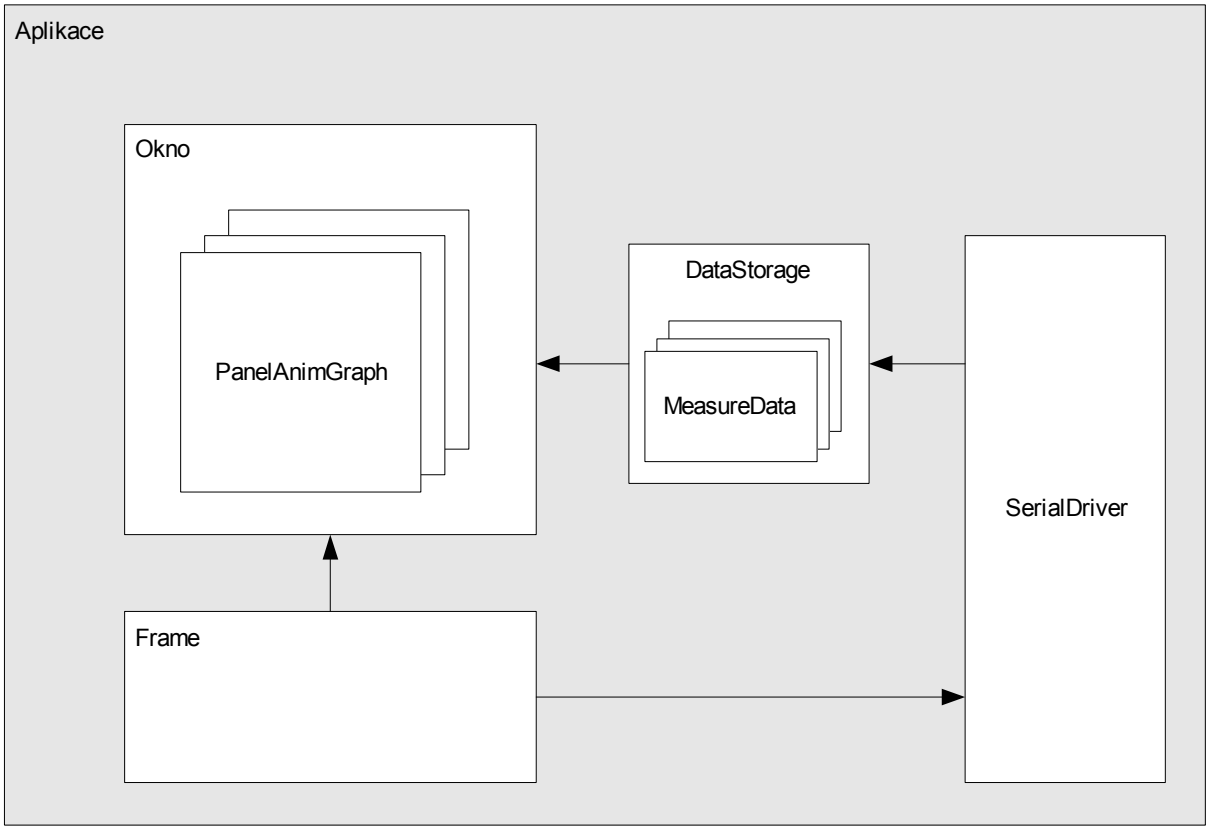
RegulatorPID) do souboru measuredata.dat. Metoda readFlash čte data ze souboru measuredata.dat a nastavuje příslušné parametry regulátoru. Souborový systém v paměti je stálý, to znamená, že zaznamenaná data v paměti FLASH přetrvávají. Nejsou smazána při odpojení napájení.

6.1.7 Třída IdleThread

Třída IdleThread je implementována, aby bylo možno (z nadřazeného počítače PC) sledovat aktuální zatížení systému. Třída obsahuje vlákno, které běží s nejnižší možnou prioritou. V tomto vlákně je smyčka, ve které se pouze inkrementuje vnitřní počítadlo. Aktuální hodnotu počítadla zjišťujeme pomocí metody getCount. Hodnota načítaná ve vnitřním počítadle obsahuje informaci o zatížení systému, protože pokud je systém více zatížen, vlákno třídy IdleThread se do aktivního stavu dostane méně často.

6.2 Klientský program v PC

Klientská aplikace v nadřazeném počítači PC má za úkol přijímat data týkající se průběhu regulace a zobrazovat je ve formě grafů. Navíc na základě požadavků uživatele ohledně nastavení parametrů regulace zasílá klientská aplikace požadavky přes sériový port aplikaci v JSTAMP. Aplikace je napsána v programovacím jazyce Java, verze J2SE. Jako IDE byl použit produkt firmy Borland – jBuilder, verze 8.0. Pro tvorbu GUI byly použity knihovny awt a swing. Blokové schéma aplikace je uvedeno na obrázku 6.5.



Obrázek 6.5 - Blokové schéma klientské aplikace

6.2.1 Třída SerialDriver

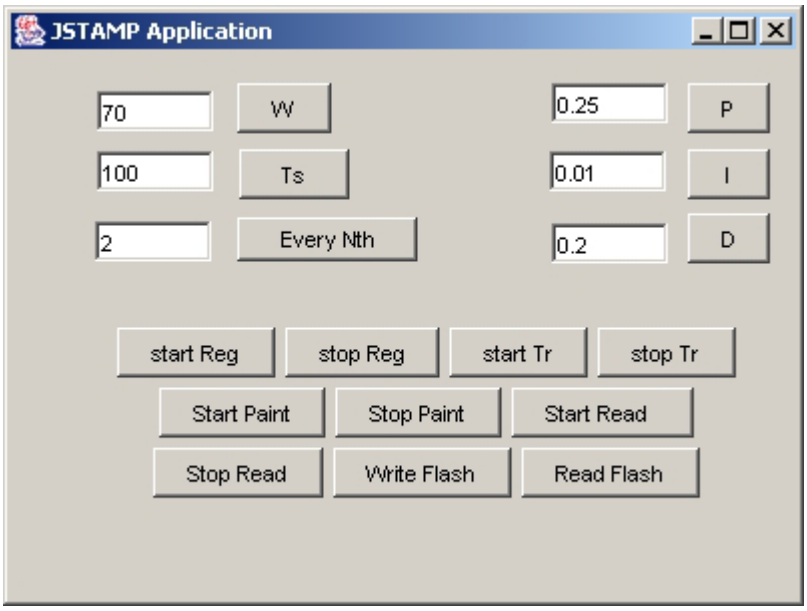
Třída SerialDriver je určena pro komunikaci přes sériové rozhraní s aplikací v JSTAMP. Obdobně, jako u aplikace v JSTAMP, obsahuje metodu initializeSerialPort, kterou se provádí počáteční nastavení parametrů sériové komunikace (sériového portu).

6.2.2 Třída DataStorage

Třída DataStorage je implementována z důvodů potřeby vyrovnat proměnnou rychlost přijímání dat přes sériový port a měnící se rychlost zobrazování dat na obrazovku. Třída obsahuje zásobník FIFO s konfigurovatelnou velikostí. Prakticky je zásobník tvořen polem objektů třídy MeasureData na principu kruhové fronty. Třída DataStorage disponuje metodami push a pop, pomocí nichž se odebírají (nebo ukládají) položky ze zásobníku.

6.2.3 Třída Frame

Třída Frame je GUI komponentou typu okno, vizuální podobu je možno shlédnout na obrázku 6.6.



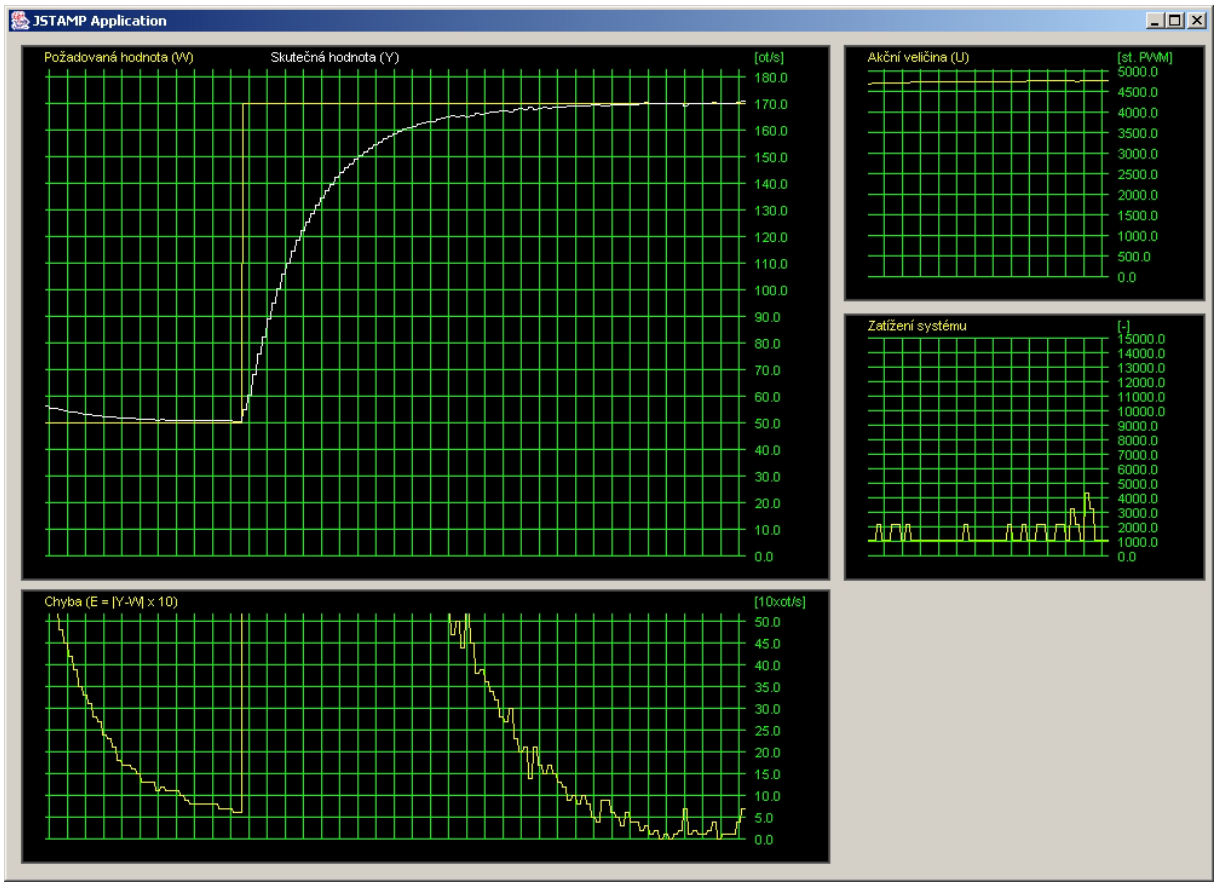
Obrázek 6.6 - Okno třídy Frame

Pomocí tlačítek a textových polí třídy Frame ovládáme parametry regulátoru a chování programu jak v JSTAMP, tak na počítači PC. Tlačítky P, I a D inicializujeme hodnoty jednotlivých položek regulátoru. Tlačítkem Ts nastavujeme periodu vzorkování (v ms). Tlačítko W slouží k nastavení požadované hodnoty otáček (v ot/s). Tlačítkem EveryNth nastavujeme, kolikátý každý vzorek regulačních dat bude přenesen po sériové lince. Pokud je hodnota nastavena na 1, přenáší se každý vzorek. Tlačítky Start Reg, Stop Reg spouštíme (pozastavujeme) činnost regulátoru PID. Tlačítko Start Tr a Stop Tr slouží ke spuštění (pozastavení) přenosu regulačních dat z JSTAMP do počítače PC.

Tlačítky Start Paint a Stop Paint spouštíme, či pozastavujeme zobrazování dat v grafech. Tlačítko Write Flash uloží parametry regulátoru do paměti Flash a tlačítko Read Flash nastaví parametry regulátoru podle dat uložených v paměti Flash.

6.2.4 Třída Okno

Třída Okno je GUI komponenta typu okno. Její vizuální podoba je na obrázku 6.7.



Obrázek 6.7 - Třída Okno

Třída Okno v sobě sdružuje čtyři vizuální komponenty třídy PanelAnimGraph. Tyto komponenty slouží k zobrazení regulačních dat. Jsou to požadovaná a skutečná hodnota otáček motorku, regulační odchylka, průběh akčního zásahu a zatížení systému.

6.2.5 Třída PanelAnimGraph

Třída PanelAnimGraph rozšiřuje třídu javax.swing.JPanel. Pomocí metody java.awt.Graphics.drawLine jsou nakresleny osy grafu, mřížka a zobrazovaný průběh vstupních dat. S každým zavoláním metody paintSamples třídy PanelAnimGraph je obraz grafu posunut o jeden pixel doleva a do vzniklého prostoru je zobrazen příslušný vzorek vstupních dat. Při zobrazování je použit tzv. double buffering, což znamená, že požadovaný obraz se nejprve kompletně sestaví v paměti a potom je najednou zobrazen na obrazovce. Tato metoda je použita, aby eliminovala různě rychlé zobrazování částí obrazu, což by mohlo vyvolat poblikávání obrazu a podobné vady zobrazení.

7 Závěr

V diplomové práci jsem otestoval vlastnosti a chování zařízení JSTAMP a s ním spojených technologií. Také jsem naprogramoval funkční ukázkovou aplikaci, která vlastnosti a možnosti zařízení JSTAMP demonstruje.

Regulační část ukázkové aplikace je velmi jednoduchá. Nastavení jednotlivých složek regulátoru (P,I,D) bylo provedeno více méně intuitivně. Nebyl zde implementován žádný složitější typ regulátoru, ani nebyla provedena identifikace fyzikálních parametrů řízeného systému. Ukázková aplikace má demonstrovat možnosti zařízení JSTAMP a neklade si za cíl provádět dokonalou regulaci. Vlastní regulace je v rámci aplikace spíše okrajovou záležitostí.

Aplikace napsané v jazyce Java, i když provozované na procesorech provádějících přímo Java bajtový kód, nebudou nikdy tak rychlé, jako programy napsané v některém z nižších programovacích jazyků. Java je již ze svého principu méně efektivní při provádění programu a prací s pamětí. Ale jak dokazuje ukázková aplikace vyvinutá v rámci této práce, je možno pomocí jazyka Java v kombinaci se zařízením JSTAMP vytvořit program, jehož výkonnost je dostatečná. Navíc s nepoměrně vyšším programátorským komfortem, než by tomu bylo při použití některého z nižších programovacích jazyků.

Přínos této práce spočívá v tom, že může zájemcům o popisované technologie posloužit jako „odrazový můstek“. Proto jsem se snažil vytvořit popis technologií v co nejširším záběru. Zájemci se mohou po přečtení této práce sami dále detailněji seznámit s konkrétními vlastnostmi JSTAMP, které budou ve své aplikaci potřebovat. A budou moci zařízení JSTAMP využít v aplikacích, které mají blíže k reálným problémům řešeným v praxi.

Jedním z doporučení, jak navázat na tuto diplomovou práci, je prozkoumat přiložené knihovny aJile CLDC Runtime Classes a mít tak možnost přesněji určit chování systému. Zdrojové kódy bohužel nejsou v současné době k těmto knihovnám k dispozici, což považuji za nedostatek ze strany výrobce, protože to velice snižuje důvěru v přesné chování systému za všech okolností. Dokumentace navíc existuje pouze k procesoru na technické úrovni. Navíc je bohužel zpracována nevhodným způsobem, neboť pojmy, které je snadné pochopit, jsou popisovány do velkých podrobností a často několika způsoby. Naopak pojmy, které jsou složitějšího charakteru, jsou „vysvětleny“ několika větami. Informace o obecném chování zařízení JSTAMP s vazbami na nastavení v programu JEMBuilder také nejsou k dispozici. V síti Internet jsem sice našel několik článků, které se zabývají touto tématikou, ale vesměs se jedná o obecný přehled vlastností. Žádná konkrétní vysvětlení neobsahují.

Pokud bych měl možnost vyvíjet ukázkovou aplikaci se znalostmi, které jsem v průběhu práce načerpal, mohla by určitě vzniknout rozsáhlejší

aplikace. Myslím si však, že ukázková aplikace splnila svůj účel a navíc je díky modulárnímu členění její činnost snadno pochopitelná.

Považuji za nutné poděkovat katedře řídící techniky FEL ČVUT v Praze, že mi umožnila seznámit se s uvedenými moderními technologiemi.

Zvláště děkuji vedoucímu mé diplomové práce panu Ing. Františku Vackovi za podnětné konzultace a za jeho celkový vstřícný přístup.

8 Použitá literatura

[1] Herbert Schildt, JAVA2 – příručka programátora, Softpress, 2001

[2] Qusay H. Mahmoud, JAVA2 – Micro Edition, Grada, 2002

[3] Joshua Bloch, JAVA efektivně, Grada, 2002

[10] aJile Java Direct Execution Microprocessors :
< http://isavix.net/whitepapers/1014166430514.pdf >

[11] Silicon based Java (A. Muir) :
<http://www.microjava.com/jvm/hardware/native/ajile2>

[12] The Real-Time Specification for Java (D. Hardin) :
<http://www.ddj.com/documents/s=888/ddj0002g/0002g.htm>

[13] Enabling Java in Embedded Systems :
<http://www.cotsjournalonline.com/pdfs/2001/12/cots12-ruggedside.pdf>

[14] RT Specification for Java :
<http://www.rtj.org/>

[15] The Memory Management Glossary :
<http://www.memorymanagement.org/glossary/>

[16] aJ-80 Data Sheet :
<přiložené CD: /dokumenty/aJ-80/aJ80Datasheet.pdf>

[17] aJ-100 Reference Manual :
<přiložené CD: /dokumenty/Dokumentace_aJ100/aJ-100ReferenceManual.pdf>

[18] Byte code instruction set :
<http://www.seas.gwu.edu/~shmuel/CRAY/Jikes%20CRAY%20Code.htm>

[19] Byte code basics :
<http://www.artima.com/underthehood/bytecode.html>

[20] Virtual Machine specification :
<http://java.sun.com/docs/books/vmspec/2nd-edition/html/Instructions.doc.html>

[21] Sběrnice CAN :
<http://www.ehl.cz/can.htm>

[22] Charade User's Guide :
< přiložené CD: /dokumenty/charade/ Charade Users Guide.pdf >

[23] Stránky společnosti Systronixs :
<http://www.jstamp.com>

[24] Stránky výrobku JSIMM :
<http://www.jsimm.com>

[25] Stránky společnosti aJile :
<http://www.ajile.com>

[26] Stránky společnosti Aidonic :
<http://www.aidonic.co.jp>

9 Přílohy

9.1 Příloha A – Slovník použitých zkratek

GC	Garbage Collector
RT	Real Time
RTOS	RT Operating System
JVM	Java Virtual Machine
JEM	Java direct Execution Microprocessor
JRE	Java Runtime Enviroment
JDK	Java Development Kit
IDE	Integrated Development Enviroment
GUI	Graphical User Interface
J2ME	Java 2 Micro Edition
CLDC	Connected Limited Device Configuration
GPIO	General Purpose Input Output
GPTC	General Purpose Timer Counter

9.2 Příloha B – Instalace a nastavení programů

9.2.1 JBuilder 8 – Personal Edition

- Instalace
- výběr instalačního adresáře
- automaticky nainstaluje JRE 1.4 do svého adr.
- ve verzi Personal se nedá nastavení JDK změnit
- nainstaluje JB8 – uninstall – do svého adr.
- nainstalovat JBuilder Documentation jb8docs\ doc-install.exe

- Nastavení

```
| Tools -> Configure Libraries
    | Name : a Jile CLDC
    | Location: JBuilder
                | Class: ../a Jile/Runtime-cldc/Rts
                | Source
                | Documentation:../a Jile/Runtime_cldc/doc
                                ../a Jile/J2ME CLDC specification

        | Project -> Default Project Properties
    | Paths
        | JDK– nelze změnit – verze Personal
        | Source
        | Documentation
        | Required Libraries -> a Jile CLDC

    | Build
                | Target VM = All JAVA SDKs
```

- První projekt

```
        | File -> New Project
```

- Comm API
- práce se sériovým portem
- vykopírovat soubory z adresáře JAvaCOMM-API na disk
- v |Tools -> Configure Libraries - nastavit cestu ke classes a documentation – tam kam jsme výše uvedené soubory zkopírovali

9.2.2 aJile Tools

verze 3.16.09
spustit \aJileTools3.16.09\install3.16.09.exe
- adresář např. D:\JSTAMP\aJile\
- install for EV-100

<u>Kopírování souborů</u>

zkopírovat z \JStamp config\Systronix-aJile–config-3-16-09.zip
do a Jile\Configurations\
adresáře
JStampPlusRamConfiguration
JStampPlusFlashConfiguration

kopírovat adr. J2MECLDCspecification
třeba do ..\JSTAMP\ aJile\
to je dokumentace k CLDC - od aJile jsou dodávány jen kompletní runtime classes, dokumentace od aJile je jen ke speciálním třídám, specifickým pro aJ-80, proto musíme nakopírovat výše uvedený adresář s dokumentací od firmy Sun.

9.2.3 Jem Builder

Po prvním spuštění
File -> new Runtime
1/3 -> např. Runtime_cldc_jméno
2/3 -> Runtime dirctory
.. \aJile \ Runtime_cldc
Class Path
.. \ aJile \ Runtime_cldc \ Rts
3/3 -> JDK class path
.. \ aJile \ Runtime_cldc \ classes.jar

Project -> Properties
Embedded Runtime : Runtime_cldc
Target Config : JstampPlusRamContig, potom stisk tl. Default
Nový projekt
File -> New Project
1/2 – pracovní adresář projektu
2/2 – Runtime + Configuration – viz Project -> Properties

new JWM
1/4 – jméno
2/4 – package + název třídy, kde se nachází třída s metodou main
např. com.tp.tests.pwmtst.PWMTestAll
3/4 – class path k package
4/4 – ovladače potřebné pro projekt

Tools -> Libraries
možno nadefinovat cesty nebo JARy pro používané knihovny

9.2.4 Charade

- při spuštění programu musí být JSTAMP připojen přes JTAG (v PC paralelní port), jinak program zahlásí chybu a ukončí se

- menu->Device – nastavit aJ–80(378) nebo (288) (adresa paralelního portu PC)

9.3 Příloha C – Struktura přiloženého CD

/dokumenty – obsahuje dokumenty stažené ze sítě Internet

/dp – obsahuje tuto diplomovou práci ve formátu PDF

/JBuilder – obsahuje instalační soubory a dokumentaci k programu JBuilder verze 8.0 Personal

/Jstamp CD – kopie instalačního CD dodávaného se zařízením JSTAMP

/soubory - obsahuje programy stažené ze sítě Internet

9.4 Příloha D – Příklady zdrojových kódů

9.4.1 SpecifikaceRTSJ

9.4.1.1 Výpis A

```
import javax.realtime.*;
/** Example of using Asynchronous Event/Event Handling facility to provide
an interface to hardware events, i.e. interrupts. A hardware interrupt
conceptually fires an AsyncEvent, which causes the associated handler to
run.
 */
public class HardwareEventExample extends AsyncEvent {
    private int interruptNum;
    /** Construct a new Hardware Event for a given interrupt.
     * @param num  Interrupt number
     */
    public HardwareEventExample(int num) {
    interruptNum = num;
    }
    /** Bind a handler to the interrupt.
     * @param h  Handler for the interrupt
     */
    public void setHandler(AsyncEventHandler h) {
    super.setHandler(h);
    Hardware.bind(interruptNum, h);
    }
}
class HardwareEventHandler extends AsyncEventHandler
{
    private int interruptCount = 0;
    /** Interrupt handler method. */
    public void handleAsyncEvent() {
        interruptCount++;
    // Driver code follows
```

```
    }
}
```

9.4.1.2 Výpis B

```
import javax.realtime.*;
/** Example of the use of "Immortal" memory in a periodic processing
context. This example eschews heap allocation, thus avoiding garbage
collection overhead.
 */
public class ImmortalMemoryExample {
    public static void main(String[] Args) {
    NoHeapRealtimeThread t = null;

    // Set up periodic processing of 1 msec each 10 msec
    PeriodicParameters timeParams = new PeriodicParameters();
    timeParams.cost = new RelativeTime(1, 0);      // 1 msec computation
    timeParams.period = new RelativeTime(10, 0);   // 10 msec period
    // Set up immortal memory; size is given in RealtimeSystem.
    MemoryParameters memParams = new
        MemoryParameters(ImmortalMemory.instance());
        // Processing is encapsulated in a Runnable
    Runner r = new Runner();
    // Create a NoHeapRealtimeThread with Periodic scheduling parameters
    // and ImmortalMemory memory parameters.
    try {
        t = new NoHeapRealtimeThread(timeParams, memParams, r);
    } catch (AdmissionControlException e) {}

   // Start processing
    t.start();
    }
}
/** Perform periodic processing in Immortal memory */
class Runner implements Runnable {
    public void run() {
        // Processing code here
    }
}
```

9.4.1.3 Výpis C

```
import javax.realtime.*;

/** Example of the use of an AsynchronouslyInterruptedException (actually,
the Timed subclass of AIE) to timeout a long-running computation, and
return a "rough" answer instead. Use of AIE avoids overhead of a polling
approach.
 */
public class AIEExample {
  /** Timeout class */
  private class MyTimed
      extends Timed {
    public MyTimed(HiResTime timeout) {
      super(timeout);
    }
  }

  /** Long-running computation that may be timed out.
   *  @param T  Timeout
```

```
   */
  int computeRefinedAnswer(MyTimed T) throws MyTimed {
    int refinedAnswer = 0;
    T.enable();
    // hairy computation goes here
    T.disable();
    return refinedAnswer;
  }

  /** Public interface to the potentially timed-out computation.  If a
   *  timeout occurs while computing the "refined" answer, a "rough"
   *  answer is returned instead.
   */
  public int computeAnswer() {
    int roughAnswer = 3;
    // Set up 100 usec timeout
    MyTimed T = new MyTimed( (new RelativeTime(0, 100000)));
    try {
      return computeRefinedAnswer(T);
    }
    catch (MyTimed t) { // Computation timed out
      return roughAnswer;
    }
  }
}
```

9.4.1.4 Výpis D

```
import javax.realtime.*;

/** Example use of RawMemoryAccess (actually, a subclass of RawMemoryAccess
 *  for accessing the Intel x86 I/O space) to directly address memory. This
 *  (elided) example provides an interface to the Intel 8253 Programmable
 *  Interval Timer, based on code originally developed by Gerald H.
Hilderink.
 */
public class IOAccess
    extends RawMemoryAccess { /* ... */}

public class I8253Example {
  /* ... */
  private long counter0Offset = 0;
  private long counter1Offset = 1;
  private long counter2Offset = 2;
  private long controlWordOffset = 3;

  private byte controlWord0 = 0x00;
  private byte controlWord1 = 0x00;
  private byte controlWord2 = 0x00;

  /** Create instance of the I8253 class.
   *  @param baseAddr  base address
   */
  IOAccess iox;
  public I8253Example(long baseAddr) {
    iox = IOAccess.create(baseAddr, (long) 8);
  }

  /* ... */
  /** Write a 16-bit value to counter 2.
```

```
   *  @param value  value for counter 2
   */
  public void setCounter2(short value) {
    setControlWord(controlWord2);
    iox.setByte(counter2Offset, (byte) (value & 0xFF));
    iox.setByte(counter2Offset, (byte) (value >> 8));
  }

  /** Write a byte value to the control register.
   *  @param value  value for control register
   */
  public void setControlWord(byte value) {
    iox.setByte(controlWordOffset, value);
  }

  /* ... */
  /** Read a 16-bit value from counter 2.
   * @return value of counter 2
   */
  public short getCounter2() {
    short value;
    setControlWord(COUNTER2);
    value = (short) iox.getByte(counter2Offset);
    value |= ( (short) iox.getByte(counter2Offset) << 8);
    return value;
  }
}
```

9.4.2 Testování vlastností aJ-80

9.4.2.1 Doba reakce na vnější událost

Třída Convert

```
package com.tp.utils;

/**
 * <p>Title: Utility tools</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: </p>
 * @author Tomáš Procházka
 * @version 1.0
 */

import com.ajile.lang.Math;

public class Convert {
  // this array is used to display hex values
  static final char[] digit = {
      '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd',
'e',
      'f'};

// prevadi cislo v String bin tvaru na int
  public static int strBinToInt(String str) {
    int res = 0;
    int mocnina2 = 1;
    String pom;

    for (int i = str.length(); i > 0; i--) {
```

```
      pom = str.substring(i - 1, i);
      if (pom.compareTo("1") == 0 || pom.compareTo("0") == 0) {
        if (pom.compareTo("1") == 0) {
          res += mocnina2;
        }
        mocnina2 = mocnina2 * 2;
      }
    }
    return res;
  }

  // prevadi byte na string v hexa tvaru
  public static String toStrHex(byte b) {
    return "" + digit[ (b >> 4) & 0x0f] + digit[b & 0x0f];
  }

  // prevadi cislo ve formatu float na jeho reprezentaci typu String
  public static String floatToStr(float f, int decimals) {
    String deciPart = String.valueOf( (long) (Math.round( (f -
Math.floor(f)) *
        Math.pow(10, decimals))));
    while (deciPart.length() < decimals) {
      deciPart = "0" + deciPart;
    }
    return "" + String.valueOf( (long) Math.floor(f)) + "." + deciPart;
  }
}
```

Třída TmrTestDemo

```
package com.tp.tests.tmrtst;

/**
 * <p>Title: Timer test</p>
 * <p>Description: Testing Timer/Counter features</p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: </p>
 * @author Tomáš Procházka
 * @version 1.0
 */

import com.ajile.jem.rawJEM;
import com.tp.utils.Convert;

public class TmrTestDemo {

  public static void tmrSetup() {
//                GPTC_IOPR    098765432109|876|543|210
    final String GPTC_IOPR = "------------|001|001|000";

//                GPTC_GMR    09876543210|98|765|4|3|210
    final String GPTC_GMR = "-----------|00|000|1|1|001";

//                GPTC0_MR    0|9|8|7|6|5|4|32|1|09|8|76|5|43|2|10
    final String GPTC0_MR = "-|1|1|1|1|1|1|00|0|00|1|11|1|10|0|01";

//                GPTC_IOMR    09|87|65|43|21|09|87|65|43|21|09|87|65|432|10
    final String GPTC_IOMR = "--
|00|00|00|00|00|00|00|00|00|00|01|00|000|01";

    rawJEM.set(0x9001008, Convert.strBinToInt(GPTC_IOPR));
    rawJEM.set(0x9001000, Convert.strBinToInt(GPTC_GMR));
```

```
    rawJEM.set(0x9001018, Convert.strBinToInt(GPTC0_MR));
    rawJEM.set(0x9001004, Convert.strBinToInt(GPTC_IOMR));

// nastaveni primo hodnot int
    rawJEM.set(0x9001014, 1); //GPTC_PRLR - nastaveni deliciho koef. delice
//    rawJEM.set(0x9001014, 36863); //GPTC_PRLR - nastaveni deliciho koef.
delice
    // 36863 jeden tick casovace je 1ms
    // 1 = jeden tick casovace je (1+1)/36.864.000 s

    rawJEM.set(0x900101c, 0x0); //GPTC0_RLR - nastaveni Reload Registru
    // 0 = chci, aby se citac zastavil na 0

    rawJEM.set(0x9001020, 0xffff); //GPTC0_CTR - nastaveni Current Time
Register

    rawJEM.set(0x900100c, 0x2); //GPTC_GTR - nastaveni Global Trigger
Register
    // ---1- = zastavit timer 0

    /*    System.out.println(rawJEM.getInt(0x9001020)); // GPTC0_CTR -
Current Time Register
        System.out.println(Convert.strBinToInt(GPTC_IOPR));
        System.out.println(Convert.strBinToInt(GPTC_GMR));
        System.out.println(Convert.strBinToInt(GPTC0_MR));
        System.out.println(Convert.strBinToInt(GPTC_IOMR));
     */
  }

  public static void setupMeasure() {
    rawJEM.set(0x9002004, 0x0); //GPIOA_OR - nataveni vsech vyst pinu portu
A na 0
    rawJEM.set(0x900100c, 0x2); //GPTC_GTR - nastaveni Global Trigger
Register
    // ---1- = zastavit timer 0
    rawJEM.set(0x9001014, 1); //GPTC_PRLR - nastaveni deliciho koef. delice
    // 36863 jeden tick casovace je 1ms
    // 1 = jeden tick casovace je (1+1)/36.864.000 s
    rawJEM.set(0x9001020, 0xffff); //GPTC0_CTR - nastaveni Current Time
Register

    System.out.println("I'm ready to measure");
  }

  public static float getIntervalInSec() {
    long prl, ctr;

    ctr = rawJEM.getInt(0x9001020); //  GPTC0_CTR  - current time reg.
    prl = rawJEM.getInt(0x9001014); //  GPTC_PRLR  - delitel delice
//    System.out.println(ctr + " " + prl);
    return (float) (prl + 1) * (0xffff - ctr) / 36864000;
    // 36864000 Internal peripheral clock = 1/2 * CPU freq.
  }

  public static float getIntervalInMicSec() {
    long prl, ctr;

    ctr = rawJEM.getInt(0x9001020); //  GPTC0_CTR  - current time reg.
    prl = rawJEM.getInt(0x9001014); //  GPTC_PRLR  - delitel delice
    return (float) (prl + 1) * (0xffff - ctr) / 36864000 * 1000000 / 18 *
10;
```

```
  }

  public static void main(String[] args) {
    // TmrTestDemo tmrTestDemo1 = new TmrTestDemo();
    tmrSetup();
    System.out.println("---- Start Aplikace v0");
    System.out.println("------  Timer - nastaveno");
    System.out.println(Convert.floatToStr(54544.5444F, 2));

    //cekaci smycka
    int i = 0;
    int pom = 0;
    while (i++ < 999999999) {
      if (i % 1000000 == 0) {
        pom++;
        System.out.println("i = " + String.valueOf(i) + ", pom = " +
                           String.valueOf(pom));
        // System.out.println("cas [s] = " + Convert.floatToStr(
        //     getIntervalInSec(),5));
        System.out.println("cas [us] = " + Convert.floatToStr(
            getIntervalInMicSec(), 5));
        if (pom == 5) {
          setupMeasure();
        }
      }

    }
    System.out.println("---- Konec Aplikace");

  }
}
```

Třída MyThread

```
package com.tp.tests.inttest;
import java.lang.JEM_Thread;

public class MyThread
    extends Thread {
  int th_id;
  public MyThread(int id) {
    super();
    th_id = id;
    start();
  }

  public void run() {
    System.out.println("Thread c." + th_id + " starting");
    while (true) {
      int i = 0;
      while (i++ < 100000) {}
      System.out.println("Thread c." + th_id + " i=" + i);

    }
  }
}
```

Třída IntTestSharp100

```
package com.tp.tests.inttest;

// pokus o prekryti metody
com.ajile.drivers.irq.InterruptController.gpioAInterrupt()V
/**
 * <p>Title: Interrupt Test</p>
 * <p>Description: Testing interrupt capabilities and threading load
 * pridano serial echo</p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: </p>
 * @author Tomáš Procházka
 * @version 1.0
 */
import com.ajile.drivers.irq.*;
import com.ajile.drivers.gpio.GpioPin;
import com.ajile.jem.rawJEM;
import com.tp.tests.tmrtst.TmrTestDemo;
import com.tp.utils.Convert;

public class IntTestSharp100  implements InterruptEventListener{
  private static InterruptController ic;
  private GpioPin pinIn;
  private GpioPin pinOut;
  private static final int IN_PIN = ic.GPIOA_BIT4_INTERRUPT; // nastaveni
vstupniho pinu - preruseni
  private static final int PIN_IN = GpioPin.GPIOA_BIT4; //  nastaveni
vstupniho pinu - prace v ramci GPIO
  private static final int PIN_OUT = GpioPin.GPIOA_BIT3; //  nastaveni
vystupniho pinu - prace v ramci GPIO

  public void interruptEvent(){
       System.out.println("Toto by se nemelo provest");
       rawJEM.breakPoint("Spustila se intEvent - to se NESMI");
   }
  static void myInterruptEvent() { // obsluzna rutina preruseni
    //  System.out.println("Start rutina");
   // rawJEM.breakPoint("BP in rutina");
    //pinOut.setPinState(!pinOut.getPinState());
    //  for (int i=0;i<2000000;i++){} //cekaci smycka

    //   rawJEM.set(0x9002004,~ rawJEM.getByte(0x9002008));
    rawJEM.set(0x9002004, 0xff);
    rawJEM.set(0x9002024, 8); // GPIOA_EDCR edge detector clear
    // 8 = by melo resetnout vsechny hranove detektory
//    System.out.println("End rutina");
    // System.out.println("cas [us] = " + Convert.floatToStr(
    //         TmrTestDemo.getIntervalInMicSec(),5));

  }

  public void setupIrq() {
    pinOut = new GpioPin(PIN_OUT); // vytvoreni instance objektu vystupniho
pinu
    pinOut.setOutputPin(true);
    ic.disableInterrupt(IN_PIN); // zakazani preruseni na vst pinu
    System.out.println("1");
    ic.addInterruptListener(this,IN_PIN); // nastaveni obsluzne rutiny
preruseni pro vst pin
    System.out.println("2");
```

```
    ic.setPinInterruptCondition(IN_PIN, ic.RISING_EDGE); // nastaveni druhu
spusteni preuseni - nabezna hrana
    System.out.println("3");
    pinIn = new GpioPin(PIN_IN); // vytvoreni instance objektu vstupniho
pinu
    System.out.println("4");
    pinIn.setOutputPin(false); // nastaveni vstupniho pinu jako vstupni pin
    System.out.println("5");
    ic.enableInterrupt(IN_PIN); // povoleni preruseni na vst pinu

  }

  public static void main(String[] args) {
    System.out.println("---- Start Aplikace v Sharp4");
    IntTestSharp100 app = new IntTestSharp100();
    Thread.currentThread().setPriority(10);

    System.out.println("------  Objekt vytvoren");
    app.setupIrq();
    System.out.println("------  IRQ nastaveno");
    SerialEcho se = new SerialEcho();
    TmrTestDemo.tmrSetup();
    TmrTestDemo.setupMeasure();

   //vytvoreni vlaken

    System.out.println("priorita main th " +
Thread.currentThread().getPriority());

    MyThread[] mojeThredy= new MyThread[150];
    for (int i=0;i<150;i++){
      mojeThredy[i]= new MyThread(i+1);
      System.out.println("vlakno"+(i+1)+" vytvoreno");
      mojeThredy[i].setPriority(5);
    }


    //cekaci smycka
    System.out.println("1.cas [us] = " + Convert.floatToStr(
            TmrTestDemo.getIntervalInMicSec(), 5));
    int i = 0;
    while (i++ < 999999999) {
     // if (i % 10000 == 0) {
        System.out.println("cas [us] = " + Convert.floatToStr(
            TmrTestDemo.getIntervalInMicSec(), 5));
        try {
          Thread.sleep(5000);
        }
        catch (InterruptedException e) {
          System.out.println("WaitThread error in main th" + e);
        }


    }
    System.out.println("---- Konec Aplikace");
  }

}
```

9.4.2.2 Komunikace přes sériový port

Třída SerialEcho

```
package com.tp.examples;

/**
 * <p>Title: Serial Echo</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: </p>
 * @author Tomas Prochazka
 * @version 1.0
 */

import java.io.IOException;
import java.io.OutputStream;
import java.io.InputStream;

import javax.comm.SerialPort;
import javax.comm.SerialPortEvent;
import javax.comm.CommPortIdentifier;

import com.systronix.util.*;

public class SerialEcho
    implements javax.comm.SerialPortEventListener {
  /** reference to the serial input stream */
  protected InputStream inStream;
  /** reference to the serial output stream */
  protected OutputStream outStream;

  // obsluzna rutina spoustena pri prichodu dat do serioveho portu
  public void serialEvent(SerialPortEvent ev) {
    try {

      if (ev.getEventType() == SerialPortEvent.DATA_AVAILABLE) {
        //System.out.println (".");

        try {
          while (inStream.available() > 0) {
            //int numBytes = inStream.read(readBuffer);
            echo(inStream, outStream);
          }
        }
        catch (IOException e) {
          System.out.println("IOException caught in serialEvent");
        }
      }

    }
    catch (Exception e) {
      System.out.println("Exception caught outside serialEvent: " +
e.toString());
      e.printStackTrace();
    }

  } // end serialEvent

  // inicializace serioveho portu
  public void initializeSerialPort() {
```

```
    SerialPort serialPort;
    CommPortIdentifier commPortId;

    try {
      //commPortId = CommPortIdentifier.getPortIdentifier("com1");
      commPortId = CommPortIdentifier.getPortIdentifier("com1");
      if (!commPortId.isCurrentlyOwned()) {
        if (commPortId.getPortType() == CommPortIdentifier.PORT_SERIAL) {
          try {
            // Open the serial port. The name is unimportant,
            // The second (timeout) parameter is unimplemented
            serialPort = (SerialPort) commPortId.open("Serial Test", 0);

            // Set the baud rate, etc, use 115 kbaud to make it more
stressful
            serialPort.setSerialPortParams(115200,

javax.comm.SerialPort.DATABITS_8,

javax.comm.SerialPort.STOPBITS_1,

javax.comm.SerialPort.PARITY_NONE);

            // Register for notification of data changes
            try {
              serialPort.addEventListener(this);
            }
            catch (Exception tml) {
              System.out.println("addEventListener error " +
tml.toString());
              System.exit(1);
            }
            serialPort.notifyOnDataAvailable(true);

            // Get output and input streams
            outStream = serialPort.getOutputStream();
            inStream = serialPort.getInputStream();

          }
          catch (javax.comm.UnsupportedCommOperationException ucoe) {
            System.out.println("Error: UnsupportedCommOperationException");
            System.exit(1);
          }
          catch (javax.comm.PortInUseException piue) {
            System.out.println("Error: PortInUseException");
            System.exit(1);
          }
          catch (java.io.IOException ioe) {
            System.out.println("Error: IOException");
            System.exit(1);
          } // end catch block
        }
        else {
          System.out.println("Error: comm port is not a serial port");
          System.exit(1);
        } // end if port is serial
      } //  end if port is not owned
    }
    catch (javax.comm.NoSuchPortException nsp) {
      System.out.println("NoSuchPortException");
      System.exit(1);
```

```
    }

  } // end initializeSerialPort

  // metoda, ktera vycka nastaveny casovy okamzik prijata data posle zpet a
vytiskne je na obrazovku
  public void echo(InputStream inStream, OutputStream outStream) {
    try {
      byte rcvVal = (byte) inStream.read(); // blocking call
      WaitThread wt = new WaitThread(500);
      wt.run();

      outStream.write(rcvVal);
      System.out.println("0x" + Util.toHex(rcvVal) + " " + (char) rcvVal);
    }
    catch (IOException e) {
      System.out.println("IOException caught in echo");
    } // end catch
  } // end echo

  /**
   * Instantiate our class then wait this thread.
   * All the action then occurs in the serial event handler.
   */
  public static void main(String[] args) {

    SerialEcho mEcho = new SerialEcho();
    mEcho.initializeSerialPort();
    System.out.println("Pokus - serial port echo START");

    // block this thread
    // the serial events handler will be called
    // from the serialThread
    mEcho.blockThread();
  }

  public synchronized void blockThread() {
    try {
      wait();
    }
    catch (InterruptedException ie) {
      System.out.println("block thread exception: " + ie.toString());
      ie.printStackTrace();
    }
  }

}
```

9.4.2.3 Generování PWM

Třída PWMTestDemo3

```
package com.tp.tests.pwmtst;

/**
 * <p>Title: Testy PWM</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2003</p>
```

```
 * <p>Company: </p>
 * @author Tomáš Procházka
 * @version 1.0
 */

import com.ajile.jem.rawJEM;
import com.tp.utils.Convert;

public class PWMTestDemo3
    implements PWMDemoInterface {
  int RLR;
  PWMTestDemo3() {

    RLR = rawJEM.getInt(0x9001030); //GPTC1_RLR - cteme nastaveni Reload
Registru
    //  kolikrat ma nacist clock na jednu periodu PWM
    System.out.println("RLR:" + String.valueOf(RLR));
  }

  public void SetDutyCycle(int perCent) {
    int FIFO = (RLR+1) / 100 * perCent;
    rawJEM.set(0x9001060, FIFO); //WF1_WPR - FIFO registr
    // urcuje stridu
    rawJEM.set(0x9001064, 3); // WF1_CP WF1 Clear Port
    // |11 = 3 - nuluji chybu a flag preteceni
//    System.out.println("DutyCycle:" + String.valueOf(RLR + 1 - FIFO) +
"/" +
//                       String.valueOf(RLR + 1));
  }

  public static void tmrSetup() {
//              GPTC_IOPR    098765432109|876|543|210
    final String GPTC_IOPR = "------------|--0|000|000";

//              GPTC_GMR    09876543210|98|765|4|3|210
    final String GPTC_GMR = "-----------|00|000|1|1|010";

//              GPTC1_MR    0|9|8|7|6|5|4|32|1|09|8|76|5|43|2|10
    final String GPTC1_MR = "-|0|0|0|0|0|0|00|0|00|0|00|0|00|0|01";

//              GPTC_IOMR    09|87|65|43|21|09|87|65|43|21|09|87|65|432|10
    final String GPTC_IOMR = "--
|00|00|00|00|11|00|00|00|00|00|00|00|000|01";

//              WF1_MR    43|2|10
    final String WF1_MR = "--|-|11"; // set PWM mode

    rawJEM.set(0x9001008, Convert.strBinToInt(GPTC_IOPR));
    rawJEM.set(0x9001000, Convert.strBinToInt(GPTC_GMR));
    rawJEM.set(0x900102C, Convert.strBinToInt(GPTC1_MR));
    rawJEM.set(0x9001004, Convert.strBinToInt(GPTC_IOMR));
    rawJEM.set(0x9001054, Convert.strBinToInt(WF1_MR));

// nastaveni primo hodnot int

    rawJEM.set(0x9001014, 20); //GPTC_PRLR - nastaveni deliciho koef.
delice
//    rawJEM.set(0x9001014, 36863); //GPTC_PRLR - nastaveni deliciho koef.
delice
    // 36863 jeden tick casovace je 1ms
    // 1 = jeden tick casovace je (1+1)/36.864.000 s
```

```
    rawJEM.set(0x9001030, 199); //GPTC1_RLR - nastaveni Reload Registru
    //  kolikrat ma nacist clock na jednu periodu PWM

    rawJEM.set(0x9001064, 3); // WF1_CP WF1 Clear Port
    // |11 = 3 - nuluji chybu a flag preteceni

    rawJEM.set(0x9001060, 20); //WF1_WPR - FIFO registr
    // urcuje stridu

    rawJEM.set(0x9001064, 3); // WF1_CP WF1 Clear Port
    // |11 = 3 - nuluji chybu a flag preteceni

// 36863,1000,500 = střída 0.5, perioda 1s

//    rawJEM.set(0x900100c, 0x2); //GPTC_GTR - nastaveni Global Trigger
Register
    // ---1- = zastavit timer 0

//    System.out.println(rawJEM.getInt(0x9001020)); // GPTC0_CTR - Current
Time Register
    System.out.println("GPTC_IOPR:" + Convert.strBinToInt(GPTC_IOPR));
    System.out.println("GPTC_GMR:" + Convert.strBinToInt(GPTC_GMR));
    System.out.println("GPTC1_MR:" + Convert.strBinToInt(GPTC1_MR));
    System.out.println("GPTC_IOMR:" + Convert.strBinToInt(GPTC_IOMR));

  }

  public static float getIntervalInSec() {
    long prl, ctr;

    ctr = rawJEM.getInt(0x9001020); //  GPTC0_CTR  - current time reg.
    prl = rawJEM.getInt(0x9001014); //  GPTC_PRLR  - delitel delice
//    System.out.println(ctr + " " + prl);
    return (float) (prl + 1) * (0xffff - ctr) / 36864000;
    // 36864000 Internal peripheral clock = 1/2 * CPU freq.
  }

  public static float getIntervalInMicSec() {
    long prl, ctr;

    ctr = rawJEM.getInt(0x9001020); //  GPTC0_CTR  - current time reg.
    prl = rawJEM.getInt(0x9001014); //  GPTC_PRLR  - delitel delice
    return (float) (prl + 1) * (0xffff - ctr) / 36864000 * 1000000;
  }

  public static void main(String[] args) {

    System.out.println("---- Start Aplikace v2");
    for (int i=0;i<2000000;i++){} //cekaci smycka
    tmrSetup();
    System.out.println("------  Timer na PWM - nastaveno");
    PWMTestDemo3 pwmTestDemo = new PWMTestDemo3();
    SerialEcho se = new SerialEcho(pwmTestDemo);
    IntDriver id = new IntDriver();
    id.setupIrq();
    int i = 0;
    int pom = 0;
    int otacky, chyby;
    while (i++ < 999999999) {
     try{
```

```
        Thread.sleep(2000);
      }catch (Exception e) {
         System.out.println("Preruseni sleepu");
         }
         otacky = id.getOtacky();
         chyby= id.getChyby();
         System.out.println("pocet otacek: " + String.valueOf(otacky));
         System.out.println("pocet chyb: " + String.valueOf(chyby));
         if (otacky != 0)
         System.out.println("pomer chyby/pulzy[%]: "
                                 +
Convert.floatToStr((float)chyby/otacky*100/16,4));

     }
     System.out.println("---- Konec Aplikace");

   }
}
```

9.5 Příloha E – Tabulka vývodů zařízení JSTAMP

JStamp Pinouts			
Pin #	**Signal Name**	**I/O**	**Description**
1	+3.3V	I/O	If you power JStamp through its VRAW input (pin 40) JStamp's power converter provides 3.3V @ 100 mA on this pin for your use off-module. You can alternatively provide JStamp with regulated 3.3 VDC +/- 5% on this pin and not provide power on the VRAW input.
2	GND		System ground
3	IOA4	I/O	Bit programmable I/O signal and external interrupt input. 24 mA sink/source
4	IOA3	I/O	Bit programmable I/O signal and external interrupt input. 24 mA sink/source
5	IOA2	I/O	Bit programmable I/O signal and external interrupt input. 24 mA sink/source
6	IOA1	I/O	Bit programmable I/O signal and external interrupt input. 24 mA sink/source
7	IOA0	I/O	Bit programmable I/O signal and external interrupt input. 24 mA sink/source Also drives the JStamp heartbeat and status LED through a PNP transistor (so that it doesn't load down this pin).
8	GND		System ground
9	CLK0	O	aJ-80 Clkout signal, a programmable divider output.

			Derived from the internal processor clock divided by 2, 4, or 8. The clock output may also be disabled when an external clock is not necessary.
10	GND		System ground
11	IOB5	I/O	8 mA sink/source I/O pin
12	IOB4	I/O	8 mA sink/source I/O pin
13	IOC6	I/O	8 mA sink/source I/O pin. Also SPI Transfer Clock.
14	IOC5/FA1	I/O	8 mA sink/source I/O pin. Also SPI Master In/Slave Out. (Also used by JStamp during flash programming - so be careful to not drive this pin externally during flash programming operations).
15	IOC4/FA0	I/O	8 mA sink/source I/O pin. Also SPI Master Out/Slave In. When operating in master mode MOSI is an output used to transmit data to the slave device. When operating in the slave mode MISO is input which receives data from the slave device. (Also used by JStamp during flash programming - so be careful to not drive this pin externally during flash programming operations).
16	IOC3	I/O	8 mA sink/source I/O pin, Also SPI Slave Chip Select 3.
17	IOC1	I/O	8 mA sink/source I/O pin. Also SPI Slave Chip Select 1.
18	IOC0	I/O	8 mA sink/source I/O pin. Also SPI Slave Chip Select 0/Slave Mode Select.
19	IOD6	I/O	8 mA sink/source I/O pin
20	GND		System ground
21	GND		System ground
22	IOD5/RXDA	I/O	8 mA sink/source I/O pin, also UARTA RXD - more RS-232 info
23	IOD4/TXDA	I/O	8 mA sink/source I/O pin, also UARTA TXD - more RS-232 info
24	IOD1/RXDB	I/O	8 mA sink/source I/O pin, also UARTB RXD - more RS-232 info
25	IOD0/TXDB	I/O	8 mA sink/source I/O pin, also UARTB TXD - more RS-232 info
26	IOE7	I/O	8 mA sink/source I/O pin
27	IOE6	I/O	8 mA sink/source I/O pin

28	IOE4	I/O	8 mA sink/source I/O pin
29	IOE3	I/O	8 mA sink/source I/O pin
30	GND		System ground
31	SWAP_MEM(L)	I	This pin has a pullup resistor. If left floating high, Flash memory is located at location 0, the boot vector. If pulled to ground, SRAM will be at the boot location 0. This pin will typically be pulled low to use SRAM for your code during development, then allowed to float high once your application is deployed in flash.
32	JTAG_TDO	O	JTAG Test Data Output, see note 2
33	JTAG_TDI	I	JTAG Test Data Input, see note 2
34	JTAG_TMS	I	JTAG Test Mode Select. The TMS input controls sequencing through the 1149 test access port (TAP) state machine. see note 2
35	JTAG_TCK	I	JTAG Test Clock input, see note 2
36	CRST(L)	I/O	Open-drain reset to/from JStamp's reset controller and the aJ-80. JStamp circuitry drives this pin low whenever voltage at the JStamp is below acceptable levels. This pin can also be driven by the JTAG debugger interface. JStamp has a 10K pullup to 3.3V on this pin. You may connect it to other logic in your system which you want to be reset by JStamp. Connect only to other open-drain or open-collector devices or to inputs of your system. (Do not drive this signal with a totem-pole output from your system or you will prevent JStamp from functioning properly.) You can drive this with an open-drain or open-collector output from your system to cause JStamp to reset (you can also use the RESET_PB input).
37	N/C		Not connected or used. Do not connect anything to this pin. May be used in a future JStamp revision.
38	RESET_PB(L)	I	Input from a reset pushbutton. Circuitry on JStamp debounces this input. Use to reset JStamp.
39	GND		System ground
40	VRAW	I	Raw power input, 5-14 VDC (4.75V min), unregulated. JStamp itself draws 30-80 mA at 5V (dependent on processor speed which is controlled by your firmware). At 7.3728 MHz, JStamp draws about 10 mA from a 9V battery.

Notes on JStamp GPIO (General Purpose I/O)	
1	JStamp GPIO pins are 3.3V max Voh, compatible with TTL levels, and are 5V I/O tolerant. They interface with no additional circuitry to 3.3V and 5V TTL-level devices.
2	JTAG pins must be connected only to a Systronix JTAG adapter or Xilinx Parallel III programming adapter. Any other connection voids your warranty and may damage JStamp.
3	JStamp grounds are all connected together, so in a minimal system you only need to connect at least one to your system. More is better, so connect them all if possible.
4	Each GPIO pin may be individually configured as input or an output. Every GPIO pin may also be configured to generate a CPU interrupt. Interrupt flexibility is provided by allowing interrupts to be triggered on a rising edge, falling edge, either edge, high level, or low level. To minimize pin-count most of the GPIO signals are multiplexed with other I/O signals of the aJ-80. On a reset the multiplexed signals are configured as GPIO inputs. Operation of the multiplexed signals is controlled with the I/O configuration registers.
5	JStamp assigns some GPIO pins to specific purposes when using the JSimm interface.