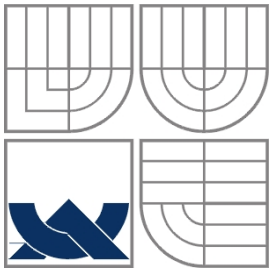
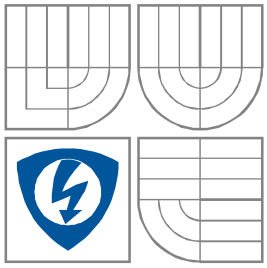


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV RADIOELEKTRONIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF RADIO ELECTRONICS

ALGORITMY ZPRACOVÁNÍ SIGNÁLU NA PLATFORMĚ AVR32

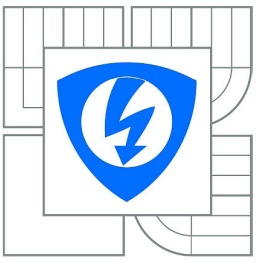
SIGNAL PROCESSING ALGORITHMS ON AVR32 PLATFORM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. FILIP ZÁPLATA
AUTHOR

VEDOUCÍ PRÁCE — Ing. ZBYNĚK FEDRA, Ph.D.
SUPERVISOR

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav radioelektroniky

Diplomová práce

magisterský navazující studijní obor
Elektronika a sdělovací technika

Student:	Bc. Filip Záplata	***ID:***	72765
Ročník:	2	***Akademický rok:***	2010/2011

NÁZEV TÉMATU:

Algoritmy zpracování signálu na platformě AVR32

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s architekturou procesorů Atmel AVR32 UC a s vývojovým prostředím pro tyto procesory. Prostudujte možnosti vývojové desky ATEVK1100 s tímto procesorem.
Otestujte dostupné algoritmy pro zpracování signálu na dané platformě a zhodnoťte jejich implementaci. Uvažte i možnosti získávání vstupních dat (AD převodník, komunikační rozhranní) a navrhněte vhodnou ukázkovou aplikaci.
Po domluvě s vedoucím projektu vyberte vhodnou aplikaci nebo aplikace pro demonstraci možností dané platformy při zpracování signálu. Připravte jednoduchý manuál pro usnadnění práce při zpracování signálů na dané platformě a jako návod pro předvedení možností vývojové desky.

DOPORUČENÁ LITERATURA:

[1] AVR32 Architecture Manual. Atmel, 2007 - [cit. 15.1.2009]. Dostupné na http://www.atmel.com/dyn/resources/prod_documents/doc32000.pdf

[2] ATEVK1100 description. Atmel, 2009 - [cit. 15.1.2009]. Dostupné na http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4114

Termín zadání:	7.2.2011	***Termín odevzdání:***	20.5.2011

Vedoucí práce: Ing. Zbyněk Fedra, Ph.D.

prof. Dr. Ing. Zbyněk Raida
Předseda oborové rady

UPOZORNĚNÍ:

ABSTRAKT

Diplomová práce zhodnocuje vlastnosti architektury AVR32, mikroarchitektury AVR32UC a hlavně mikrokontroléru AT32UC3A0512. Tento mikrokontrolér je osazen na desce EVK1100, která je zde použita pro odlaďování aplikací. Celý rozbor je zaměřen na možnost zpracování audio-signálů na této desce. K desce je vytvořeno AD/DA rozhraní a ovládací knihovna. Následuje nutný popis použité knihovny DSP-lib. Poslední částí je popis teorie a implementace dvou hudebních efektů a implementace operačního systému FreeRTOS.

KLÍČOVÁ SLOVA

architektura Atmel AVR32, mikroarchitektura AVR32UC, mikrokontrolér AT32UC3A0512, vývojová deska EVK1100, programátor JTAGICE mkII, vývojové prostředí AVR32 Studio, DSP-lib, AVR32 Framework, 3-pásmový digitální grafický ekvalizér, efekt pitch-scale, STFT analýza/syntéza, CORDIC algoritmy, FreeRTOS

ABSTRACT

Master‘s thesis reviews the characteristics of the AVR32 architecture, AVR32UC microarchitecture, and especially AT32UC3A0512 microcontroller. This microcontroller is mounted on the board EVK1100, which is used for debugging applications. The entire analysis is focused on the ability to process audio signals on this board. For the board is created AD/DA interface and its control library. Follows necessary description of used DSP-lib library. The last part is a description of the theory and implementation of two sound effects and implementation of operating system FreeRTOS.

KEYWORDS

architecture Atmel AVR32, microarchitecture AVR32UC, microcontroller AT32UC3A0512, evaluation board EVK1100, on-chip debugging tool JTAGICE mkII, development environment AVR32 Studio, DSP-lib, AVR32 Framework, 3-band digital graphic equaliser, pitch-scale effect, STFT analysis/synthesis, CORDIC algorithms, FreeRTOS

ZÁPLATA, F. *Algoritmy zpracování signálu na platformě AVR32*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií. Ústav radioelektroniky, 2011. 60 s, 8 příl. Diplomová práce. Vedoucí práce: Ing. Zbyněk Fedra, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma Algoritmy zpracování signálů na platformě AVR32 jsem vypracoval samostatně pod vedením vedoucího semestrální práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

V Brně dne ....

(podpis autora)

PODĚKOVÁNÍ

Děkuji vedoucímu diplomové práce Ing. Zbyňku Fedrovi, Ph.D. za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé diplomové práce.

V Brně dne ....

(podpis autora)

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

ÚVOD

Číslicové zpracování zvukových signálů je z výpočetního hlediska dost náročný proces. Pro takové operace jsou vyvíjeny speciální procesory, které mají instrukční sady úzce zaměřeny na signálové výpočty a umožňují tím rychlejší průchod vzorků signálu.

Procesory architektury AVR32 nepatří sice mezi signálové procesory, nicméně jejich výkon by měl i pro takovéto využití postačit. K vývoji aplikací na těchto procesorech je možné využít vývojové desky fy. Atmel, jednou z nich je také EVK1100.

Zvukové signály jsou distribuovány v analogové formě, k digitálnímu systému je tedy nutné přiřadit rozhraní schopné tyto signály převést. Vstupní signál může být buď již číslicový z nějakého záznamu, nebo analogový. Výstup do koncového zesilovače a reproduktorů je ale, až na výjimky, vždy analogový. Pro takové rozhraní jsou používány kvalitní stereofonní AD a DA převodníky. Časté jsou sigma-delta převodníky s vysokým bitovým rozlišením a velkým odstupem SNR.

Tato architektura je pro DSP tak podporovaná, že je k ní dokonce dodávána knihovna s funkcemi DSP zpracování. Jedním z cílů práce je zjistit použitelnost těchto funkcí pro zpracování audio-signálů a kde jsou hranice výkonu procesoru real-time zpracování. Tyto schopnosti je možné ověřit přímo implementací náročných hudebních efektů.

Mezi základní operace zpracování signálů patří filtrace, v audio-technice jsou široce nastavitelnými filtry např. ekvalizéry. Mezi náročnější DSP operace pak patří efekty pracující se spektrální analýzou a syntézou, tyto efekty zde umožní ověřit zároveň univerzálnost knihovny DSP-lib a výkonnost procesoru architektury AVR32.

Pokročilejší systémy s výkonnými procesory si žádají inteligentnější software. Takový software může být již velice rozsáhlý a i zkušený programátor se časem začne ztrácet, navíc je třeba zajišťovat dostatečnou modulárnost a výsledné bloky nakonec propojit. Výraznou úlevu zajišťuje multitasking. I ve vývojovém prostředí je implicitně k dispozici operační systém FreeRTOS. Cílem je naznačit, jak je možné vytvořit snadno ovladatelné aplikace realizující zvukové efekty na vytvořeném rozhraní za pomoci operačního systému FreeRTOS.

1 ARCHITEKTURA AVR32

AVR32 je nové 32-bitové jádro s RISCovou instrukční sadou. Tato architektura byla vytvořena pro cenově dostupné aplikace s nízkou spotřebou a vysokou hustotou kódu. 32-bitové jádro umožňuje pracovat výhodně i s 64-bitovými slovy, což zajišťuje vysokou flexibilitu využití.

1.1 Základní vlastnosti procesorů AVR32

Architektura AVR32 je mírně odlišná, a proto nekompatibilní s předchozí architekturou AVR. Pro zvýšení pružnosti a hustoty kódu jsou instrukce rozděleny do dvou skupin. První skupina obsahuje 16-bitové instrukce. Je to většina standardních instrukcí známých z architektury AVR. Druhá skupina zabírá 32-bitové instrukce, jež mají rozšířené schopnosti. Obě skupiny instrukcí mohou být v kódu libovolně využívány.

Rozdělením instrukční sady do těchto dvou skupin bylo dosaženo velké efektivity kódu. Například pro načítání konstanty do registru jsou k dispozici 2 verze instrukce. Pro načítání malých konstant postačí 16-bitová instrukce, pro větší konstanty pak 32-bitová. Jiné instrukce se pak mohou lišit počtem operandů.

Jádro AVR32 je rozděleno s ohledem na náročnost aplikací na další mikroarchitektury. Jednotlivé typy procesorů spadají do různých architektur, mají tak různé vlastnosti více či méně vhodné pro konkrétní aplikace, tento rozdíl ve výkonu se samozřejmě odráží i na ceně produktu.

Mikroarchitektura AVR32A je zaměřena na levnější aplikace, kde nevadí nižší výkon procesoru. Obsluha přerušení používá místo HW registrů zásobník, to ušetří plochu čipu a tím i cenu, ale na úkor zpomalení obsluhy přerušení. Po spuštění obsluhy přerušení jsou některé registry automaticky zálohovány také do zásobníku, stejně jako status registr.

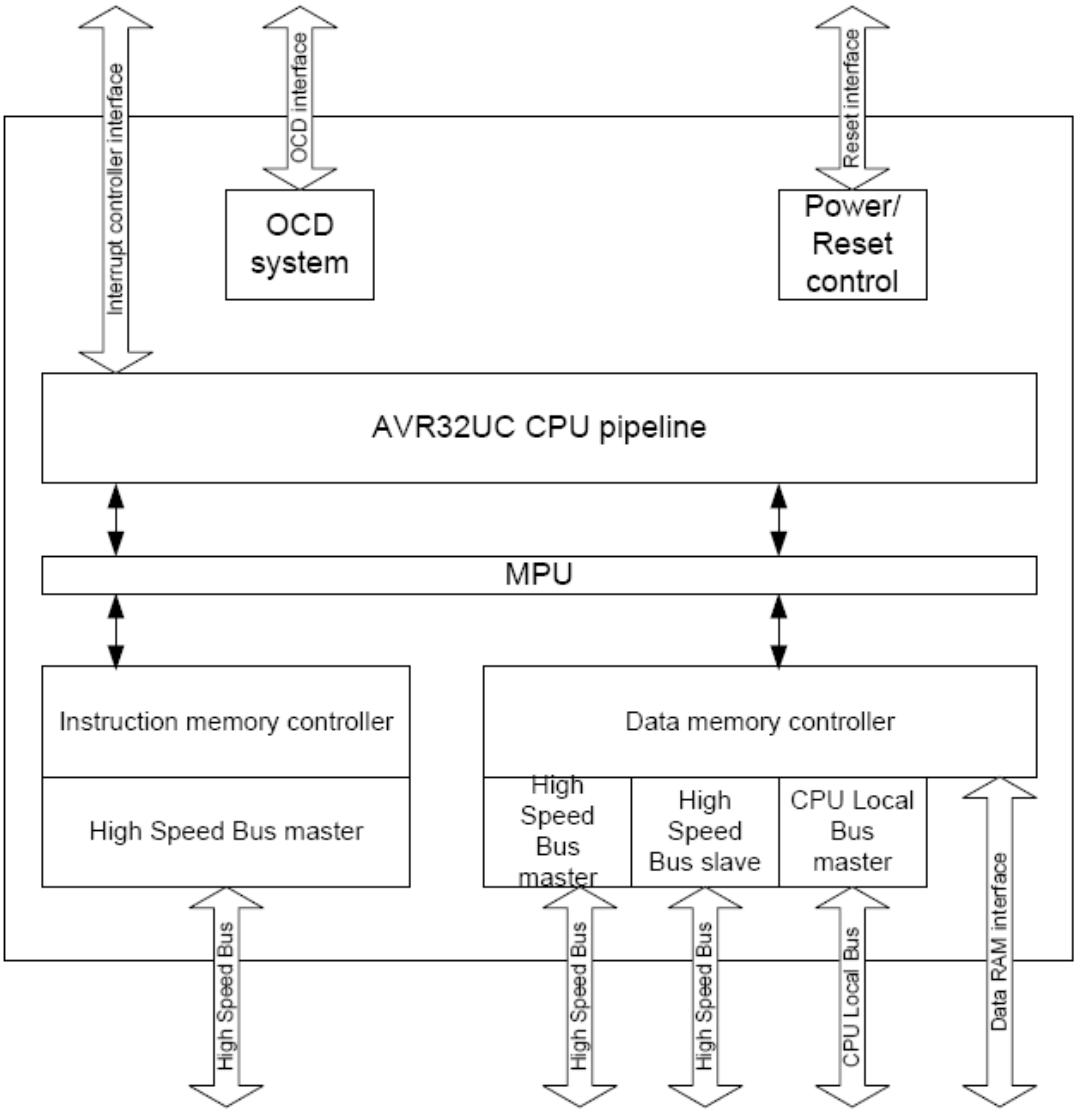
U mikroarchitektury AVR32B probíhá obsluha přerušení odlišně. Zálohování pracovních registrů je realizováno pomocí HW souboru registrů k tomu určených. Status registr a ostatní registry jsou také zálohovány do speciálních hardwarově vytvořených registrů.

Rozdíl mezi těmito dvěma mikroarchitekturami je tedy především v obsluze přerušení a to konkrétně ve způsobu zálohování registrů. Mikroarchitektura AVR32B umožňuje významně zrychlit obsluhu přerušení za cenu nutnosti přítomnosti nadstandardních zálohovacích registrů.

Architektura AVR32UC je první implementací mikroarchitektury AVR32A. Zajišťuje rozšířený systém OCD, optimální jednotku MPU a nepodporuje Java akceleraci. Obsahuje 3 paměťová rozhraní, pro vyzvedávání instrukcí, přístup do paměti a jednu pro přístup k dalším paměťovým rozhraním. Veškerý přístup do paměti je kontrolován MPU jednotkou, a pokud není povolen přístup, je vyvolána výjimka. Blokové schéma AVR32UC je na obr. 1.1.

Další text je zaměřen pouze na architekturu AVR32UC, což předurčuje směr

k vývojovému kitu EVK1100.



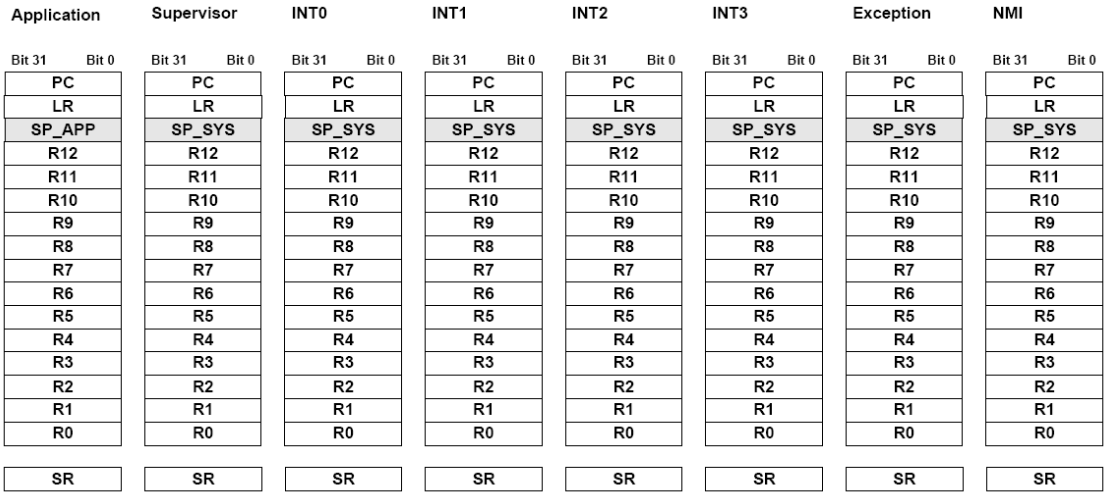
Obr. 1.1 Blokový popis architektury AVR32UC (převzato z [3])

1.2 Programátorský model

Tato kapitola popisuje soubor registrů, které je nutno znát pro správné naprogramování procesoru. Architektury jsou časem mírně upravovány, AVR32UC již prošla třemi takovými úpravami, během nichž došlo především k rozšíření instrukční sady. Revize lze do budoucna ještě očekávat, uvedené informace jsou v souladu s [3].

Procesory architektury AVR32 umí pracovat s daty typu *byte* (8 bitů) až *double word* (64 bitů). K vyjádření znaménkového typu je standardně využíván dvojkový doplněk. Některé instrukce jsou, jako je to i u AVR architektury, určeny pro práci ve zlomkovém tvaru. Rozsah těchto čísel je pak -1 až $+1-2^{-(N-1)}$ také ve dvojkovém doplňku. Více-bytová slova jsou do paměti ukládána metodou *big-endian*, nicméně pokud je třeba metody *little-endian*, jsou pro to vyhrazeny speciální instrukce.

Na obr. 1.2 je popsán soubor registrů. Sestává z bloku šestnácti registrů a jednoho status registru. Všechny registry jsou 32-bitové. Programový čítač, linkový registr a ukazatel na zásobník jsou umístěny v bloku registrů společně s registry pro všeobecné použití. Pro chod programu je volně dostupných tedy jen 13 registrů, nicméně některé z nich jsou přednostně využívány k jistým operacím. Registr *R12* je využíván pro předávání návratových hodnot funkcí. Výhodné je, že všechny tyto registry je možné použít jako ukazatele.



Obr. 1.2 Soubor registrů architektury AVR32A (převzato z [2])

Spuštěním obsluhy přerušení jsou registry zálohovány do zásobníku, pouze ukazatel na zásobník je v privilegovaných módech odložen do zálohovacího registru.

Linkový registr je využíván jako registr návratové hodnoty při volání podprogramu a to ve všech módech. Pokud není podprogram volán, je možné tento registr volně využívat jako registr *R14*.

T – Scratch bit Nenastavuje jej žádná instrukce, je určen pro obecné použití. Při resetu je nulován.

L – Lock flag Využíván při podmíněném ukládání do paměti. Je nulován instrukcí *rete* a při resetu.

Q – Saturation flag Signalizuje přetečení u saturovaných aritmetických operací. (saturované násobení apod.) Je nulován pouze manuálně instrukcí *csrf*.

V – Overflow flag Signalizuje přetečení aritmetické operace.

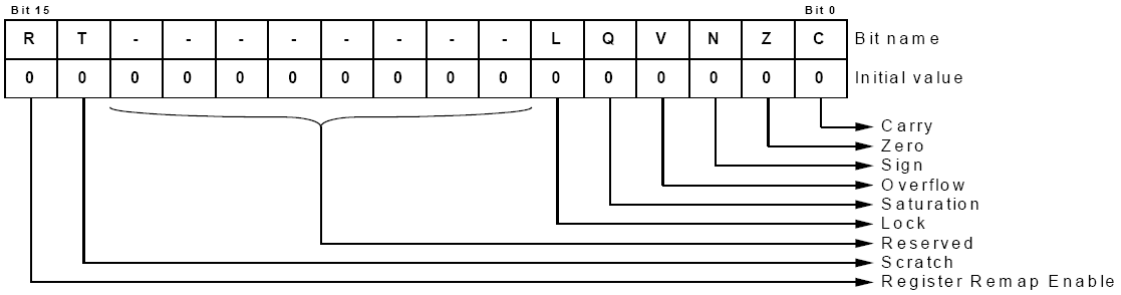
N – Negative flag Informace o polaritě výsledku. Je nastavován/nulován aritmetickými nebo logickými operacemi.

Z – Zero flag Nastaven, pokud je výsledek aritmetické nebo logické operace nulový.

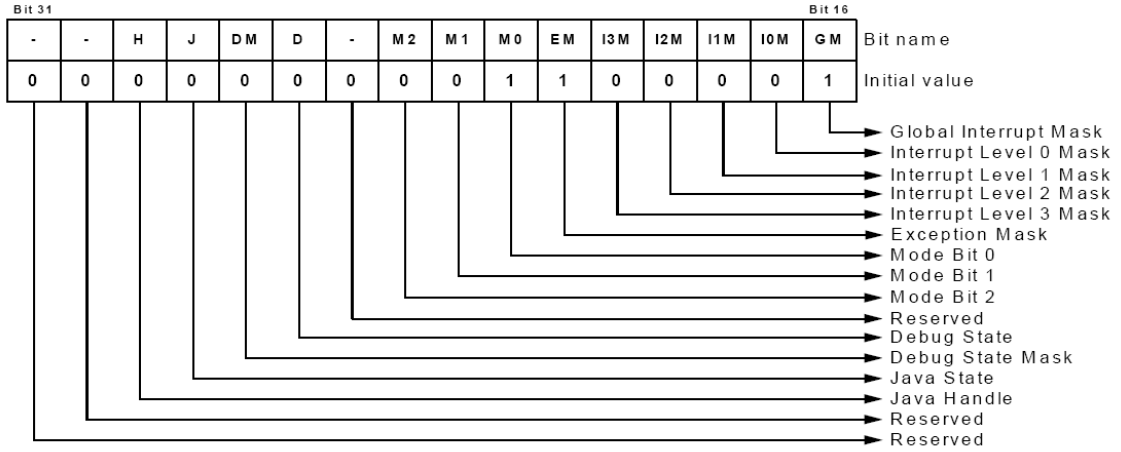
C – Carry flag Indikace přenosu při aritmetické nebo logické operaci.

Spodních 16 bitů status registru je využíváno obvykle jako příznaky vykonaných instrukcí, jejich popis je uveden na obr. 1.3. Dále následuje popis jednotlivých příznaků.

Horní polovina status registru je přístupná pouze v privilegovaném módu a informuje o stavu a módu, ve kterém se procesor právě nachází. Jeho stručný popis uvádí obr. 1.4.



Obr. 1.3 Status registr bity 0 – 15 (převzato z [3])



Obr. 1.4 Status registr bity 16 – 31 (převzato z [3])

Tabulka 1.1 Stavy procesorů architektury AVR32

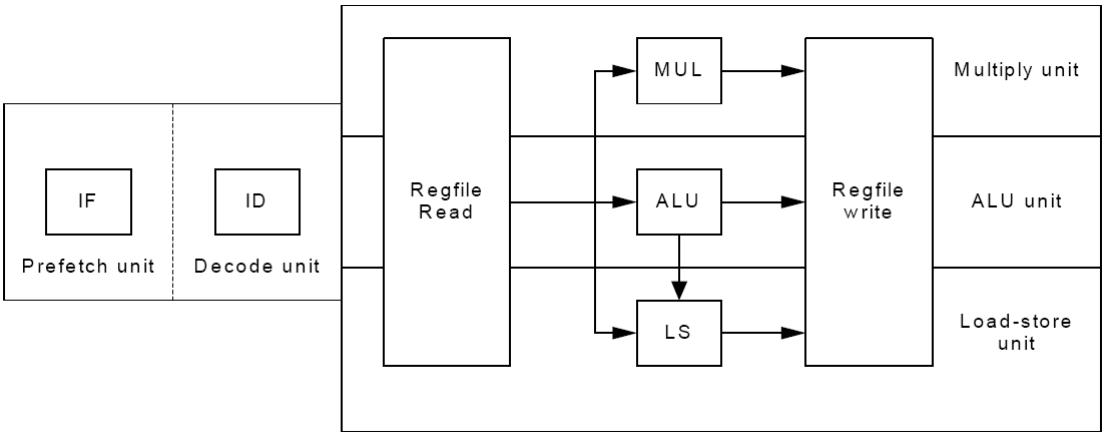
Priorita	Mód	Zabezpečení	Popis
1	nemaskované přerušení	privilegované	přerušení vysoké priority
2	výjimka	privilegované	výjimka programu
3	přerušení 3	privilegované	obecné přerušení
4	přerušení 2	privilegované	obecné přerušení
5	přerušení 1	privilegované	obecné přerušení
6	přerušení 0	privilegované	obecné přerušení
-	správce	privilegované	správcovské aplikace
-	aplikace	neprivilegované	normální mód programu

Systémové registry jsou posazeny mimo virtuální paměť a jsou přístupné pouze pomocí speciálních instrukcí *mfsr* a *mtsr*. Některé z nich mohou být měněny hardwarově. Je to 256 32-bitových registrů, některé nejsou v architektuře AVR32UC využity, některé jsou volné pro budoucí využití. Jejich kompletní popis je uveden v [2] nebo [3]. Status registr je umístěn na začátku tohoto systémového bloku.

Jak již bylo naznačeno, tyto procesory umožňují více pracovních stavů, jejich seznam uvádí tabulka 1.1 a detailnější popis pak [2]. Změny těchto stavů mohou být způsobeny jak externě (přerušením) nebo přímo programem. Tato práce je však zaměřena na práci v aplikačním módu, tak není podstatné se těmito stavy více zaobírat.

1.3 Pipelining

Architektura AVR32UC umožňuje zřetězení instrukcí ve třech fázích, vyzvednutí instrukce (IF), dekódování instrukce (ID) a její vykonání (EX). Blokové schéma pipeliningu je znázorněno na obr. 1.5.



Obr. 1.5 Blokové schéma pipeliningu (převzato z [3])

Jednotka IF má za úkol načíst z programové paměti 32 bitů instrukce a vložit ji do FIFO zásobníku, odkud si v ten samý okamžik (instrukční cyklus) vyzvedne jednotka ID předchozí instrukci.

Dekódovací jednotka převzatou instrukci rozpozná a na základě toho z ní vygeneruje potřebné adresy operandů a řídící signály pro výkonové jednotky. To vše také v jednom instrukčním cyklu. Pro „více-cyklové“ instrukce pak připraví prostor k jejich vykonání. Pokud nedokáže instrukci dekódovat, generuje výjimku.

Výkonová jednotka potom operuje s daty v registrech a v datové paměti. ALU vykonává většinu operací, především aritmeticko-logické operace. Zajišťuje i znaménkové i neznaménkové dělení.

Do procesorů AVR je vložena hardwarová násobička, která výrazně urychluje program. Vykonávání instrukcí násobení a násobení-součet (MAC) probíhá v jednom instrukčním cyklu. Pouze v případě, že jsou oba operandy 32-bitové a je očekáván 64-bitový výsledek, je k tomu třeba dvou cyklů. Při výkonu instrukcí MAC je výsledek odkládán, jak do speciálního registru přímo v MUL jednotce, tak do cíleného datového prostoru. Takové odložení umožní ušetřit jeden instrukční cyklus u další instrukce MAC. Pokud další instrukce již manipuluje s registrem, jehož adresa nesouhlasí s MAC jednotkou, pak předchází přepsání MUL-registru jiným operandem. Naopak jedná-li se o totožný registr, MUL-registr je aktualizován automaticky i s instrukcemi jako je *add*, *mul* nebo *load*. MUL-registr akceptuje 64-bitová i 32-bitová slova, při zápisu 32 bitů je horní polovina automaticky restartována.

Paměťová jednotka (load-store) přijme adresu z ALU a čte/ukládá z/do paměti vše v jednom cyklu. Paměťové instrukce s přidruženými operacemi, jako např. *ldins* nebo *ldswp,* jsou také „jedno-cyklové“. Instrukce *memc*, *mems* a *memt* jsou vykonávány jako čtení a zápis bez možnosti přerušení. Při vykonávání delší instrukce využívající datovou paměť je pipelining pozastaven, pokud datová paměť není dále využívána, pipelining se nezastavuje i přesto, že není dokončen zápis nebo čtení.

1.4 Instrukční sada AVR32UC

Sada instrukcí architektury AVR32UC je velice obsáhlá. Instrukce jsou to buď 16-bitové, nebo 32-bitové. Některé mají stejnou funkci, avšak více formátů, jež jejich funkci mírně upravují. Výběr správných instrukcí pak urychlí a zjednoduší program. Některé instrukce dovolují dokonce rozšíření o podmínku při zachování počtu zabraných instrukčních cyklů. Pomocí tečkového rozšíření je vykonání některých instrukcí ještě upraveno pro práci s různými datovými typy. Bližší informace lze vyčíst přímo z instrukční sady uvedené v [3].

Aritmeticko-logické instrukce jsou jednocyklové instrukce pro jednotku ALU. Do této skupiny se řadí instrukce součtu, rozdílu, kopírování registrů, standardní logické operace apod. Dále jsou zde k dispozici aritmetické operace se saturovanými výsledky, nebo logické operace, jako např. bitový posun, navíc se zaokrouhlením.

Součinové instrukce 32 a 48-bitového výsledku jsou také pouze jednocyklové díky jednotce MUL. Mimo standardní instrukce jsou k dispozici součinové instrukce se saturovaným výsledkem a umožňují zaokrouhlení, což pomůže snížit chybovost DSP při případném přebuzení.

MAC instrukce roznásobí dva operandy s 32 nebo 48-bitovým výsledkem, který následně přičte do registru. Pokud je v MUL jednotce v MUL-registru k dispozici hodnota právě přičítaná, trvá tato instrukce 1 cyklus, v opačném případě je nutný jeden cyklus navíc pro vyzvednutí této hodnoty z paměti.

64-bitové MAC nebo MUL instrukce vyžadují dva cykly pro součin, při čtení je pak totožné s méně-bitovými MAC.

Instrukční sada AVR32UC obsahuje také instrukce pro dělení. Vydělení je tvořeno více iteracemi, proto vyžaduje více instrukčních cyklů. Při přerušení se nečeká na dokončení těchto instrukcí, ale je obslouženo okamžitě.

Další velkou skupinu tvoří instrukce pro práci s datovou pamětí. Výpočty přístupových adres jsou prováděny sčítačkou ve výkonové části, sčítačka také realizuje automatické inkrementování/dekrementování adres, a to po jedné, tzn., že načítání z o w posunutou adresou vyžaduje $1 + w$ instrukčních cyklů. Při načítání dat méně než 32-bitových jsou slova doplněna a rotována. Podmíněná práce s pamětí vyžaduje dva případně tři instrukční cykly. Načítání 64-bitových slov vyžaduje cyklus pro každou polovinu slova. Stejně tak je tomu i u ukládání do datové paměti. Pipelining je v těchto případech vždy pozastaven. Architektura umožňuje i vícenásobné načítání/ukládání, výkon těchto operací je úměrný počtu načítaných/ukládaných registrů. Při přerušení jsou vícenásobné operace přerušeny stejně jako u dělení.

Větvení programu zabere normálně pouze jeden instrukční cyklus, při skoku je, dí-

ky nutnosti přerušení pipeliningu, program o dva cykly zpožděn. Volání podprogramu se chová stejně, pipelining je nutné přerušit.

Instrukce pro prohození registrů *xchg*, trvá dva instrukční cykly, jeden pro vyzvednutí a jeden pro uložení do paměti.

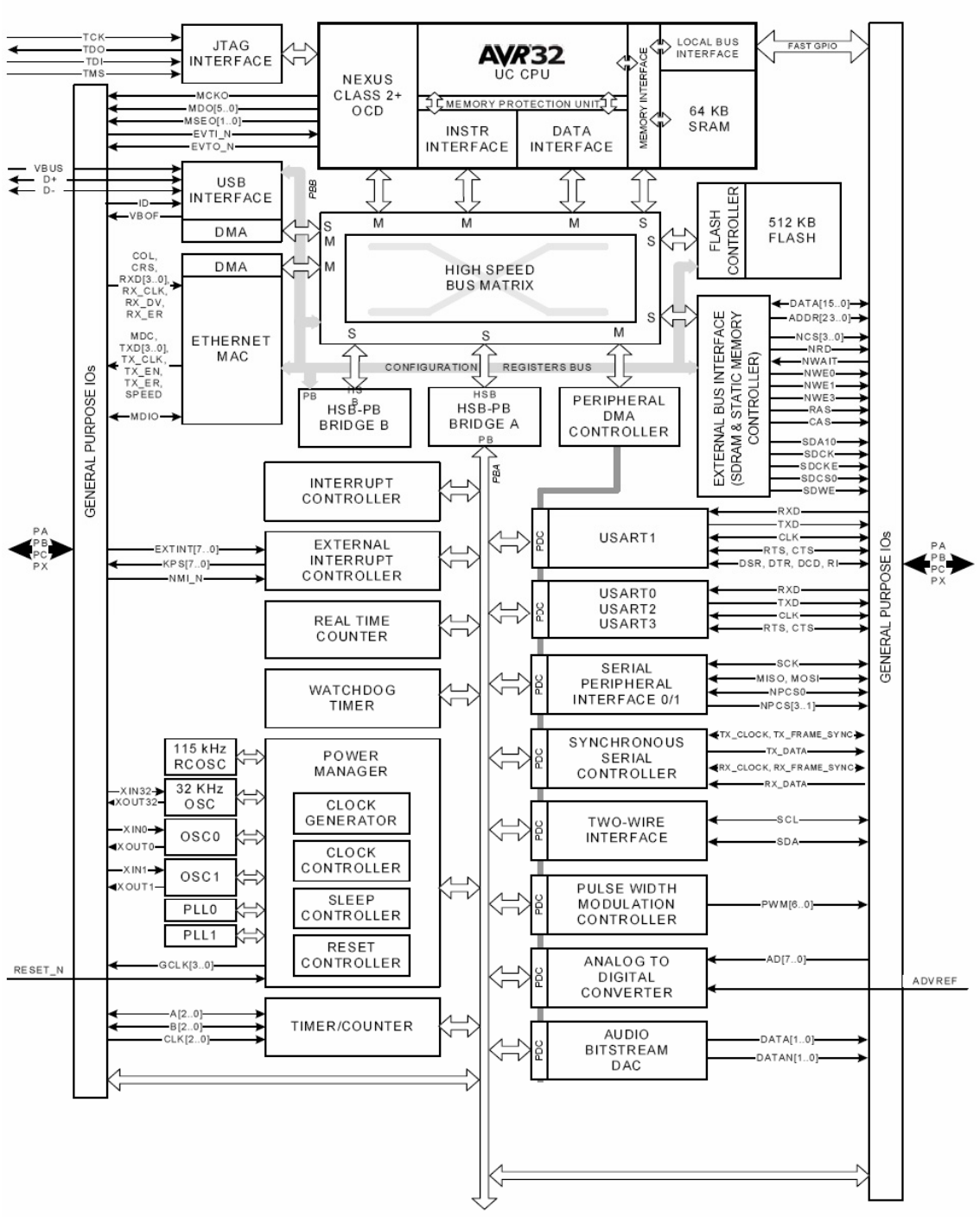
V paměti je možné skupinou instrukcí upravovat jednotlivé bity ve slovech. Tyto instrukce vyžadují více iterací, takže si žádají také větší množství cyklů, avšak mohou pracovat paralelně, pokud nenásledují další operace s pamětí.

Instrukční sada nabízí ještě množství dalších operací, převážně systémových a pro nestandardní pracovní módy. Je vidět, že nejen výběrem ale i řazením instrukcí je možné výrazně ovlivnit funkci pipeliningu a program tak zefektivnit.

1.5 Procesor AT32UC3A0512

Vývojový kit EVK1100 je osazen procesorem AT32UC3A0512. Tento procesor je, jak napovídá označení mikrokontrolér s architekturou AVR32UC s 512kB programové paměti flash. Jeho RISCové jádro je schopno pracovat až do kmitočtu 66MHz, což umožňuje zpracovat až 91DMIPS. K dispozici je 64kB paměti SRAM jako datového bloku, kterou je možné dále externě rozšířit pomocí implementovaných komunikačních modulů. Takové rozšíření je možné u procesorů s 0 v označení (AT32UC3A**0**512). Jednotka PDCA (Peripheral Direct Memory Access) umožňuje přímé propojení komunikačních modulů, jako je USART, SPI apod., s datovou pamětí, aniž by bylo nějak zatíženo jádro procesoru. Napájení hlídá jednotka Brown-Out osazená na čipu. Časovat jádro je možné buď z interního RC oscilátoru, nebo jedním externím oscilátorem. Časovač/čítač má k dispozici tři nezávislé 16-bitové kanály, které mohou být nezávisle nakonfigurovány k měření kmitočtu, pulzů, časových intervalů, ke generování pulzů, časového zpoždění, nebo k PWM modulaci. Samostatné PWM moduly poskytují 7 nezávislých kanálů, z nichž jeden může spouštět AD převodník. Dále pomocí implementovaných modulů umožňuje komunikaci přes různá rozhraní, jako UART, SPI, TWI, USB, Ethernet MAC nebo nastavitelnou synchronní sériovou linku SSC.

Na obr. 1.6 je blokové schéma procesoru.



Obr. 1.6 Blokové schéma mikrokontroléru AT32UC3A0512 (převzato z [4])

1.6 Periferie pro zpracování analogových signálů

V této podkapitole jsou blíže probrány periferie, kterými výše uvedený mikrokontrolér disponuje a jsou zajímavé z pohledu zpracování signálů.

První důležitou periferií pro DSP je jistě analogově digitální převodník. Procesor

AT32UC3A0512 nabízí 10-bitový ADC s postupnou aproximací s možností přepínání až osmi kanálů vložených multiplexerem. Je schopen převádět s rychlostí $384kSas^{-1}$ při rozlišení 10 bitů nebo až $533kSas^{-1}$ s rozlišením 8 bitů. Pro zpracování audiosignálů by rychlost převodníku postačila i pro stereofonní zpracování, ovšem jeho rozlišení by mělo být alespoň 16 bitů pro zajištění $SNR > 90dB$.

ABDAC je modul pro převod 16-bitových dat na analogový signál. Tento modul je určen přímo pro stereofonní zvukový výstup, takže podporuje (maximálně) dva kanály. Jsou v něm obsaženy dva sigma-delta DA převodníky se 128-násobným převzorkováním. Pro interpolaci po převzorkování je použito kaskádně sestaveného filtru 4. řádu (Comb4) s impulsní charakteristikou tvaru sinc. Předtím je spektrum signálu ještě upraveno jednoduchým filtrem FIR 3. řádu. Výstupní signál by měl být dále filtrován dolním propustím pro odstranění harmonických modulátoru. ABDAC může využívat modulu PDCA, aniž by obtěžoval jádro procesoru zřizováním komunikace mezi pamětí a komunikačními rozhraními. Navíc ovšem potřebuje vlastní hodinový signál, jsou předpokládány kmitočty oscilátoru do $12{,}288MHz$ což odpovídá max. $48kSas^{-1}$.

Zvukový výstup může být realizován i pomocí PWM modulů. K dispozici je 7 kanálů. Lze pomocí registrů jednoduše nastavit periodu i střídu výstupního signálu. PWM moduly jsou řízeny čítačem, jež dělením odvozuje hodiny od MCU (Master Clock Unit), to může způsobovat značné problémy s realizací přesného vzorkovacího kmitočtu. Vztah určující výstupní vzorkovací kmitočet určuje (1.1).

$$\begin{aligned} & f_{MCU} = 66MHz \\ & CPRD = 2^{16} \\ & X = 1 \\ & \downarrow \\ & f_S = \frac{f_{MCU}}{X \cdot CPRD} = \frac{66 \cdot 10^6}{2^{16}} = 1{,}007kHz \end{aligned} \qquad (1.1)$$

Je vidět, že ani při maximálním kmitočtu jádra nelze dosáhnout 16-bitové hloubky rozlišení pro audio-signály ($f_S \geq 32kHz$).

K vytvoření některých efektů, např. echo, je potřeba uchovávat delší sekvence signálu v datové paměti. Implementovaná datová paměť pro tyto funkce nemusí dostačovat, avšak mikrokontrolér umožňuje tuto paměť efektivně rozšířit pomocí bloků SMC pro statickou RAM a SDRAMC pro dynamickou RAM. SMC má 4 čipy a na každém může naadresovat až 64MB. Komunikuje přes 26-bitovou adresní a 32-bitovou datovou sběrnici. Datová sběrnice může být navíc překonfigurována na 8 nebo 16-bitovou. Signály pro čtení a zápis jsou plně programovatelné. SDRAMC podporuje 16 nebo 32-bitové paměti s velikostí stránek od 2048 do 8192 řádků a od 256 do 2048 sloupců. Podporuje taktéž 8, 16 nebo 32-bitový přístup. Navíc pro snížení spotřeby je k dispozici několik módů ovládání. Samotný přenos dat je zprostředkován modulem EBI, který jmenovaná paměťová rozhraní využívá.

Komunikaci s AD/DA převodníky, nebo s jinými externími zařízeními (PC) je možné realizovat různými rozhraními implementovanými na mikrokontroléru. Pro zajiš-

tění komunikace mezi periferiemi uvnitř zařízení jsou k dispozici standardní rozhraní SPI, TWI kompatibilní s I^2C, nebo SSC. SSC je modul pro sériovou synchronní komunikaci podporující mnoho protokolů používaných i v audiotechnice. SSC má nezávislý přijímač a vysílač a poskytuje 2 kanály pro PDCA jednotku. Komunikace s externími zařízeními, např. příjem signálových dat z PC, pak může být realizována pomocí rozhraní USART doplněného o převodník na RS232, nebo pomocí sériové linky USB.

Jednotka PDCA umožní načítat data přes přednastavené rozhraní přímo do datové paměti, aniž by komunikace obtěžovala procesor. Pokud jsou data přijata/vyslána, může být vyvoláno přerušení. Modulu PDCA se nastaví ukazatel na zásobník, který je v paměti vyhrazen pro tato data (datové pole), počet datových slov a nastaví se mu komunikační periferie. Pro každý kanál je možné nadefinovat 2 zásobníky.

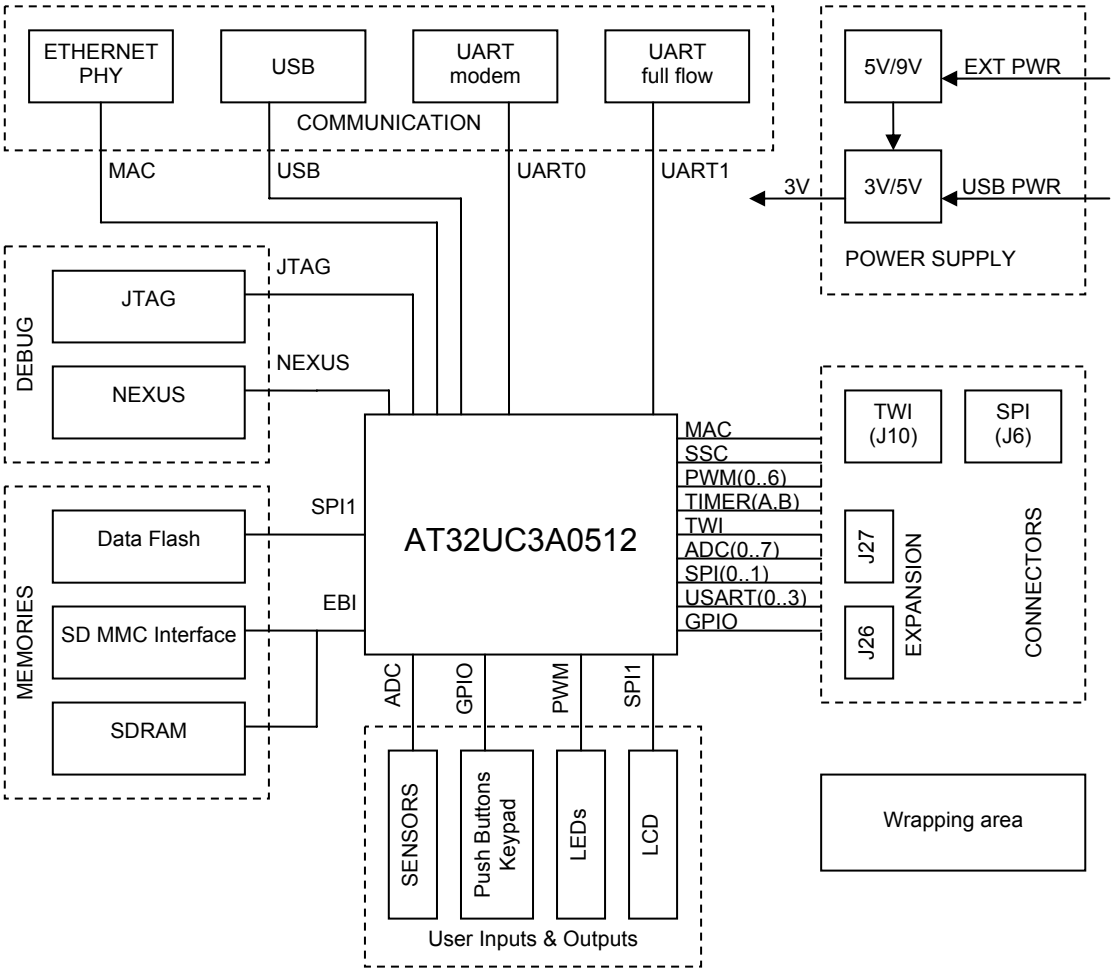
Více informací je k dispozici v [4].

2 VÝVOJOVÁ DESKA EVK1100

EVK1100 je vývojový kit fy. ATMEL určený pro odlaďování aplikací na platformě AVR32. Je osazen mikrokontrolérem AT32UC3A0512. K bližšímu popisu nejsou dostupné žádné originální dokumenty, jediným zdrojem zůstává nápověda AVR32 Studia.

2.1 Základní přehled

Obr. 2.1 ukazuje blokové schéma EVK1100, jsou na něm přehledně zobrazeny dostupné periferie.



Obr. 2.1 Blokové schéma EVK1100

Napájení desky je možné v rozsahu 8-20VDC, potřebná napětí jsou pak získána pomocí spínaných měničů. Napětí 1,8V pro napájení jádra procesoru je získáváno lineárním regulátorem v mikrokontroléru AT32UC3A0512 (výstup VDDOUT). Měniče jsou proudově omezeny na 1A pro napětí 5V a 0,8A pro napětí 3,3V, regulátor 1,8V je

také interně proudově omezen. Zdrojem může být externí napáječ nebo port USB, což je manuálně volitelné přepínačem.

Na desce je osazeno 8MB paměti flash a 32MB SDRAM. Paměti mohou být rozšířeny MMC nebo SD kartami pomocí osazeného rozhraní.

Vyvedeny jsou komunikační porty pro Ethernet, USB a dva RS232.

Programování a ladění je zprostředkováno portem JTAG. Je možné emulovat i prostřednictvím portu NEXUS, ale konektor není standardně osazen.

Seznam periferií:

- 2× USART (s převodníky RS232)
- ETHERNET (10/100, konektor RJ45)
- USB 2,0 mini
- 6 indikačních LED (2 dvoubarevné)
- 3 tlačítka
- joystick
- potenciometr (pro ADC)
- světelný senzor (fototranzistor TEMT6000)
- teplotní senzor (termistor NCP18WF104)
- LCD 4×20 znaků modře podsvícený
- SD/MMC slot
- 8MB data-flash (AT35DB642)
- 32MB SDRAM (MT48LC16M16A2)
- 3 oscilátory (hlavní 12MHz a náhradní 12MHz a RTC 32,768kHz)
- SPI port
- TWI port

2.2 Fyzické propojení procesoru a periferií

V tabulka 2.1 jsou uvedeny veškeré periferie, doplňky nebo výstupy, které jsou spojeny přímo s procesorem a jejich mapování na jednotlivé piny nebo porty mikrokontroléru. AT32UC3A0256 poskytuje čtyři obecně využitelné porty, 31-bitový PA, 32-bitový PB, 40-bitový PX a 6-bitový PC, jednotlivě s možnostmi využití implementovanými periferiemi mikrokontroléru. Dále pak, tři vstupy pro USB rozhraní, 4 piny jsou rezervovány pro rozhraní JTAG. Zbývá resetovací vstup, 9 napájecích a 17 zemnících pinů a jeden vývod je nezapojen. Dohromady 144 vývodů pouzdra VQFP144. Oscilátory jsou připojeny přes PC port.

Tabulka 2.1 Připojení periferií EVK1100

Periferie	**PORT/PIN**
3,3V napájení	VDDANA (pin 81), VDDIO (pin [92,7,36,69,73,]), VDDIN (pin 17)
VDDOUT 1,8V	VDDCORE (pin [29,51,55,138]), VDDPLL (pin 84)
Reset	reset (pin 23)
Oscilátor 12MHz hlavní	PC[2,3]
Oscilátor 12MHz vedlejší	PC[4,5]
Oscilátor 32,768kHz RTC	PC[0,1]
ETHERNET	reset (pin 23), PA24, PB[0..3,5..9,15]
USB	DM (pin 70), DP (pin 71), PA11
SPI	PA[10..13]
data flash	reset (pin 23), PA[14..17]
SDRAM	PB[10..14,16], PX[0..11,14,17,18,20,21,23..32,34..39]
SD/MMC	PA[2,7,15..18]
USART 0	PA[0,1,3,4]
USART 1	PA[5,6,8,9], PB[23..26]
TWI	PA[29,30]
termistor	PA21
fototranzistor	PA23
potenciometr	PA22
joystick	PA[20,25..28]
LED	PB[27..30]
LED 2 barvy	PB[19..22]
tlačítka	PX[16,19,22]
NEXUS	reset (pin 23), PB[4,10..14,16,17,19..21]
JTAG	reset (pin 23), TCK (pin 129), TDO (pin 130), TDI (pin 131), TMS (pin 128), PB20
LCD 4×20 znaků	reset (pin 23), PA[15..17,19], PB18
wrapping area (15×16+16VDD+16VCC)	

rozšíření	reset (pin 23), PA[0..30], PB[0..31], PC[4,5], PX[0..39]

Wrapping area je pole dutinek (prokovených otvorů) 15×16 se dvěma 16-bodovými řadami napájení 3,3V. Jsou určeny pro konstrukci jednoduchých obvodů bez nutnosti vytváření DPS nebo připojování nepájivého pole.

Pro přímý přístup k portům mikrokontroléru slouží konektory rozšíření, jenž jsou přímo spojeny se všemi porty s výjimkou PC[0..3]. Z konektorů je přístupné i napájecí napětí 3,3V pro napájení připojených obvodů. Možné je procesor i externě resetovat.

Zatím nebyla zmíněna žádná periferie, která by se starala o zpracování audio-signálů. Taková možnost není ani přímo podporována, jediným východiskem je využití portů rozšíření, odkud jsou přímo přístupny vývody jednotky ABDAC. Jako audio vstup, jak již bylo rozebráno, je k dispozici široká škála komunikačních rozhraní, která lze nakonfigurovat ke komunikaci s mnoha ADC. Pro možnost zpracování zvuku je tedy nutno doplnit tento kit ještě rozšiřující deskou.

2.3 Softwarové propojení periferií a GPIO portů

Z důvodu velkého množství implementovaných periferií a velké flexibility použití mikrokontroléru je ovládání výstupních portů mnohem složitější než tomu bylo například u architektury AVR. Vstupně-výstupní piny mikrokontroléru jsou přímo připojeny k jednotce GPIO, která zajišťuje jejich funkci.

Porty jsou ovládány jako pět 32-bitových portů, což nekoresponduje s fyzickým stavem, PA, PB, PC a PX. Fyzický port PX je například nutno ovládat dvěma virtuálními porty, proto je celkový počet virtuálních portů 5. Každý virtuální port (dále jen port) lze uživatelsky nastavit pomocí bezmála 50 registry. Každému pinu samostatně je možné nastavit zdvihací rezistor nebo i výstup s otevřeným kolektorem. Na všech portech je možné vyvolat externí přerušení se 4 módy spouštění. Porty jsou také vybaveny filtrem zákmitů, jenž lze opět nastavit na libovolný pin.

Připojení periferií k IO pinům je nutno příslušně nastavit registrem *GPER*, který rozhoduje, zda bude pin ovládán GPIO jednotkou přímo, nebo zda k němu bude připojena periferie. O tom, jaká periferie bude k danému pinu připojena rozhoduje dvojice registrů *PMR1* a *PMR0*. Registry *PMR* vybírají ze 3 skupin A, B a C, ve kterých je pro každý pin pevně definována periferie. Pro každý pin tedy nelze vybrat jakoukoli periferii, ale jen z maximálně 3 přiřazených. Tabulka 2.2 ukazuje, jakých pinů je možné využít pro jednotku ABDAC a některá komunikační rozhraní.

Jednotka PDCA se konfiguruje také pomocí několika registrů. Jednou komunikační stranou je datová paměť mikrokontroléru, registrem *MAR* se nastaví ukazatel na volné pole, které bude sloužit jako buffer. Velikost bufferu, nebo počet zpracovávaných slov musí být udáno v registru *TCR*. V případě, že je vyžadován kontinuální režim, *PDCA* umožňuje využití dvou bufferů. Registry pro náhradní buffer jsou *MARR* a *TCRR*. Po

obsloužení prvního bufferu se registry *MAR* a *TCR* automaticky přepíší registry *MARR* a *TCRR*. Velikost slova může být různá, určuje ji registr *MR*. Druhou komunikační stranou je periferie mikrokontroléru. Toto propojení je zprostředkováno nastavením pouze registru *PSR*, kam se uloží PID periferie. PID udává jak adresy datových registrů periferie, tak i směr, kterým je komunikace realizována. PID pro vybrané periferie jsou uvedeny v tabulka 2.3. PDCA jednotka umožňuje použití několika kanálů DMA, ovládací registry jsou pro každý kanál zvlášť a jsou v paměti po skupinách za sebou.

Tabulka 2.2 Možné multiplexování vybraných periferií AT32UC3A0512

Periferie	**Signál**	**Pin(funkce)**
ABDAC	DATA[0]	PA03(C), PB02(B)
	DATAN[0]	PA04(C), PB03(B)
	DATA[1]	PA23(C), PB05(B)
	DATAN[1]	PA23(C), PB06(B)
SSC	TX_FRAME_SYNC	PA14(A)
	TX_CLOCK	PA15(A)
	TX_DATA	PA16(A)
	RX_FRAME_SYNC	PA19(A)
	RX_CLOCK	PA18(A)
	RX_DATA	PA17(A)
SPI0	NPCS[0]	PA10(A), PX30(B)
	NPCS[1]	PA08(B), PX31(B)
	NPCS[2]	PA09(B), PX32(B)
	NPCS[3]	PA07(C), PX33(B)
	MISO	PA11(A), PX34(B)
	MOSI	PA12(A), PX35(B)
	SCK	PA13(A), PX36(B)
SPI1	NPCS[0]	PA14(B), PX37(B)
	NPCS[1]	PA18(B), PX38(B)
	NPCS[2]	PA19(B), PX39(B)
	NPCS[3]	PA20(B)
	MISO	PA16(B), PX34(B)
	MOSI	PA17(B), PX35(B)
	SCK	PA15(B), PX36(B)
TWI	SDA	PA29(A)
	SCL	PA30(A)

Tabulka 2.3 PID vybraných periferií

Periferie	**PID**
ABDAC	17
SSC-RX	1
SPI0-RX	7
SPI1-RX	8
TWI-RX	6

2.4 Časování mikrokontroléru

Mikrokontrolér je vybaven několika zdroji kmitočtu pro časování jádra a periferií. Základním zdrojem je RC oscilátor nastaven na kmitočet 115kHz, který je defaultně nastaven jako výchozí. Dále jsou k dispozici dva nezávislé oscilátory s nutností připojení externího krystalu v rozmezí 0,45-16MHz. Tyto 3 oscilátory je možné doplnit smyčkou PLL (2 k dispozici), bez kterých by nebylo možné dosáhnout maximálního hodinového kmitočtu 66MHz pro jádro. Jako poslední zdroj časování je pomalý oscilátor pro připojení externího krystalu 32,768kHz, užívaný ke generování přesného času.

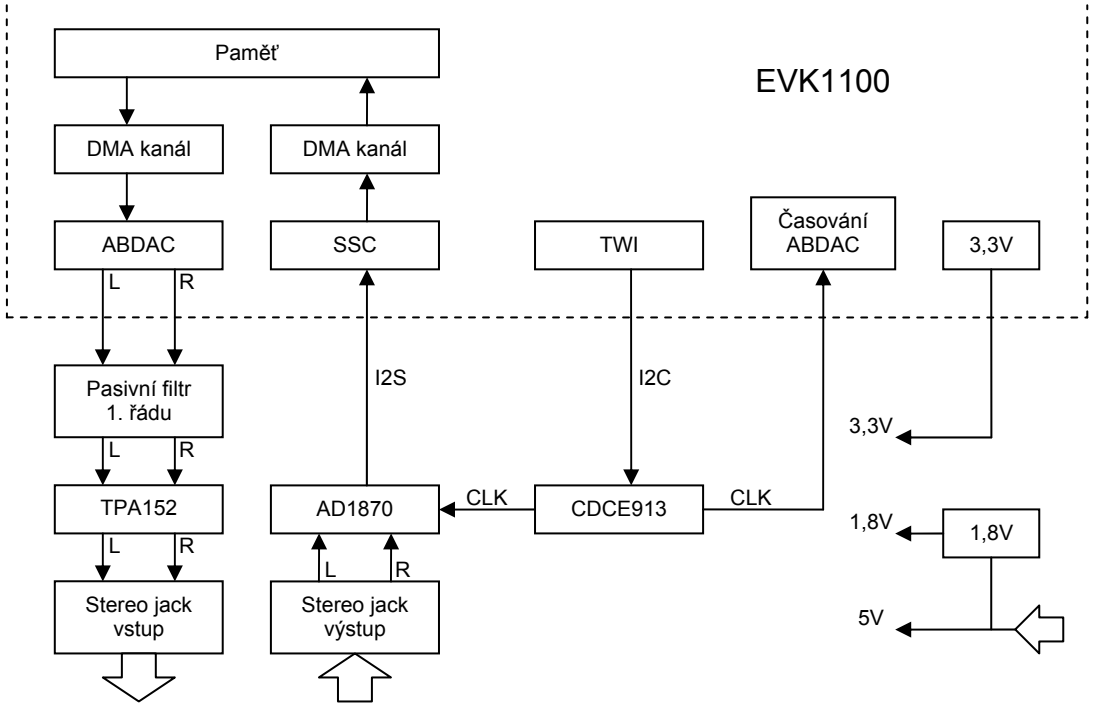
Krystaly jsou připojovány k portu PC. Rychlý oscilátor (OSC0) je možné využít pro časování všech periferií, ale jako jediný může časovat jádro procesoru. Na desce EVK1100 jsou osazeny pouze 2 krystaly, OSC0 (12MHz) a OSC32 (32,768kHz). OSC1 je neosazen a jeho připojovací body jsou k dispozici na portech rozšíření a je tedy takto možné zajistit externí časování periferií.

Jednotka analogového výstupu ABDAC vyžaduje specifické a přesné časování, které se od časování jádra liší. Pro tyto potřeby se využívá tzv. generic clock, toto časování může svůj kmitočet odvozovat ze všech jmenovaných oscilátorů a to s využitím děličky a PLL smyčky. Od dostupného oscilátoru 12MHz však tyto kmitočty získat nelze, je proto nutno využít oscilátoru OSC1.

K řešení časování OSC1 se nabízejí dvě možnosti. První z nich je doplnit kit o krystal a odrušovací kondenzátory, což je jednoduché, ale částečně omezující řešení. Napevno připájený krystal znemožní pozdější modifikace časování procesoru a omezí jednotku ABDAC na práci s jedním, nebo maximálně dvěma vzorkovacími kmitočty. Druhou možností je doplnit rozšiřující modul ještě o zdroj externího časování se smyčkou PLL, jako je např. obvod CS2200 nebo CDCE913. Tyto obvody umožní variabilní vzorkování (48kHz, 44,1kHz, 32kHz a jiné). Navíc většina AD převodníků vyžaduje také toto externí časování, lze tak tento modul efektivně využít.

3 HW AUDIO ROZHRANÍ

Zpracování zvukových signálů na desce EVK1100 není možné bez doplňující desky. Deska musí zajistit analogově digitální převod jako vstup, rekonstrukci signálu z vestavěného DA převodníku a také přesné časování těchto převodníků. Blokové schéma takto navržené desky je na obr. 3.1.



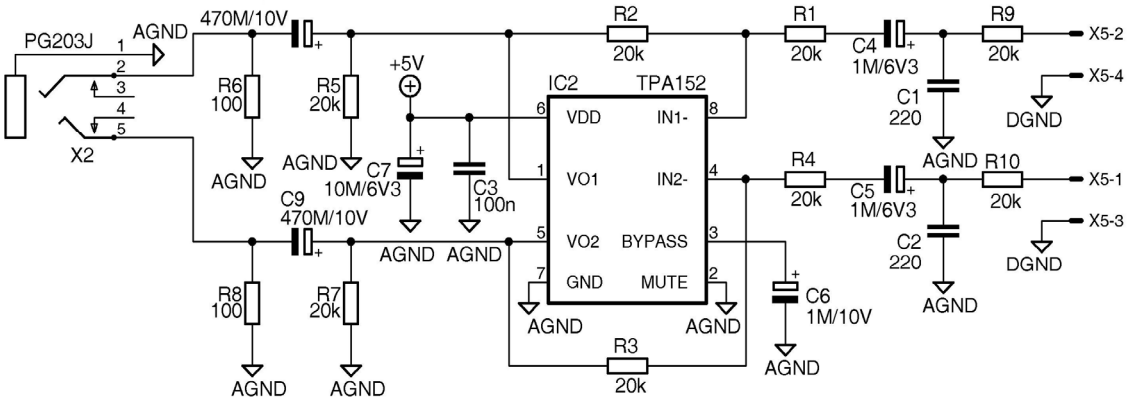
Obr. 3.1 Blokové schéma audio rozhraní

3.1 Audio výstup

Periferie ABDAC je 16-bitový sigma-delta DA převodník. Výstup je symetrický, nicméně pro testovací účely není nutno na tak krátkou vzdálenost vytvářet symetrické vedení. Desymetrizační obvody by komplikovaly celé zapojení, tak jsou invertující výstupy nezapojeny. Schéma výstupní části je na obr. 3.2.

V ABDAC není implementován rekonstrukční filtr, proto je nutno oba kanály doplnit o dolní propustí. Sigma-delta modulátory využívají převzorkování (ABDAC 128x), takže rušivá složka signálu je ve frekvenční oblasti posazena vysoko. K filtraci tedy postačí jednoduchý RC integrační člen.

Výstup DA převodníku je vysokoohmový, pro obecnou zátěž je tedy vhodné jej posílit. K tomuto účelu byl vybrán sluchátkový zesilovač od fy. Texas Instruments TPA152. Výhoda tohoto zesilovače je v tom, že je napájen pouze nesymetrickým napětím 5V a nevyžaduje o mnoho více externích součástek jako operační zesilovače. Zesílení



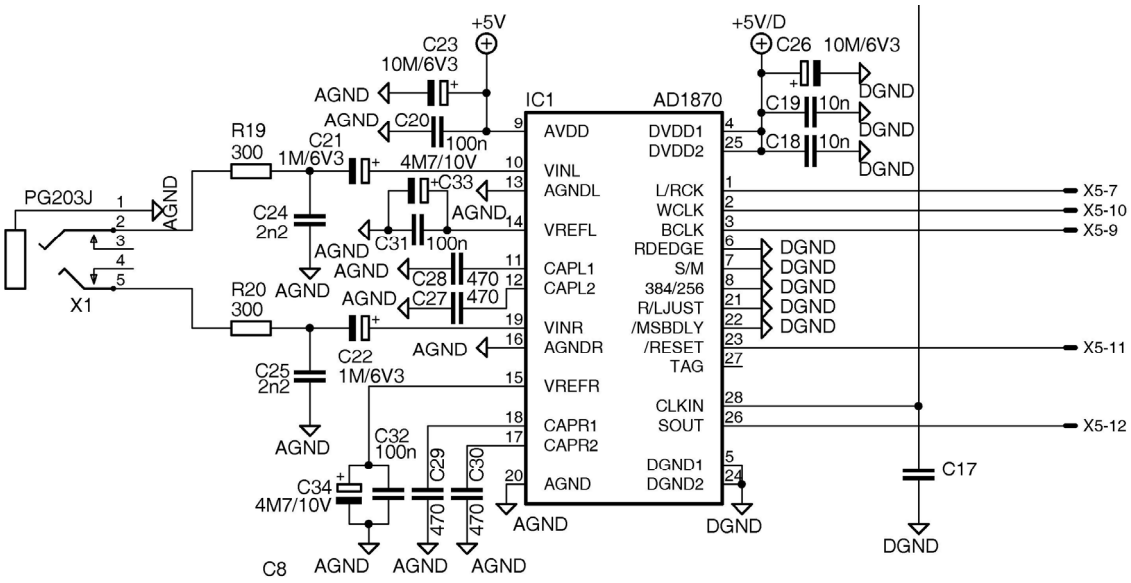
Obr. 3.2 Schéma výstupní části

obou kanálů je jednotkové. Výstup je zatížen impedancí 20kΩ pro potlačení vybíjecího impulzu kondenzátoru při vypnutí do vysokoohmové zátěže a impedancí 100Ω pro potlačení vybíjecího impulzu při připojení sluchátek (nízkoohmová zátěž).

Výstupní konektor je stereo 3,5 mm jack.

3.2 Audio vstup

Vstupní část desky tvoří pouze AD převodník, antialiasingový filtr a nutné externí součástky. Převodník AD1870 je 16-bitový sigma-delta stereo AD převodník, Analog Devices bohužel s jeho produkcí již končí, nejspíš právě proto, že je pouze 16-bitový. Pro audio-aplikace se v dnešní době používají spíše 24-bitové převodníky, ale vzhledem k tomu, že DA převodník je také 16-bitový, tento plně postačuje.



Obr. 3.3 Schéma výstupní části

Schéma výstupní části je na obr. 3.3. Převodník využívá 64-násobného převzorkování a následné FIR filtraci pro potlačení kvantizačního šumu. Tento systém umožňuje

použití velmi jednoduchého antialiasingového filtru. Opět postačuje RC dolní propust prvního řádu.

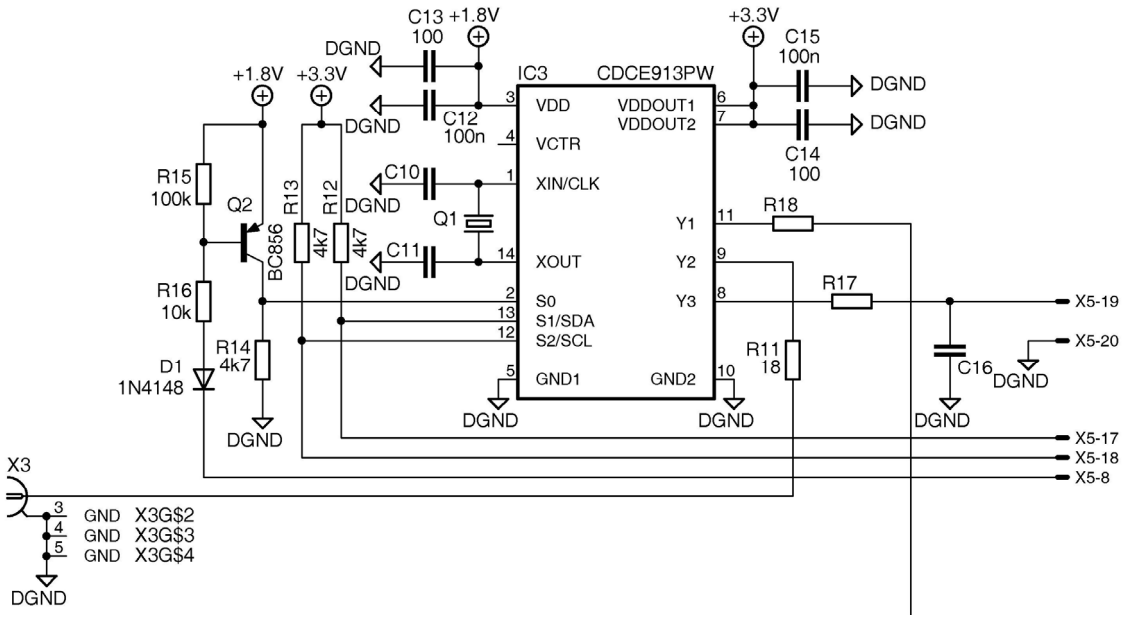
Výstupní formát je nastaven na master I2S komunikaci. Tato komunikace využívá 3 vodičů, hodinový bitový signál, datový signál a synchronizační rámcový signál. Dále je dostupný synchronizační signál pravého/levého kanálu. Data jsou posílána v 32-bitových blocích s 16 bity dat a 16 nulami, data z jednotlivých kanálů se střídají. Jaký kanál je právě vysílán je určeno právě signálem *L/RCK*. Tento signál lze výhodně využít místo synchronizačního rámcového signálu.

3.3 Časovací jednotka

Časování převodníků zajišťuje obvod CDCE913 od fy. Texas Instruments. Je to programovatelný hodinový PLL syntezátor s výstupním kmitočtem až do 230MHz. Disponuje třemi nezávislými výstupy se zvláštním napájením až 3,3V. Generátor má nízký jitter, typicky 50ps, což je pro přesné časování také důležité. Schéma zapojení časovače je na obr. 3.4.

Vstupní referenční kmitočet je generován oscilátorem s externě připojeným krystalem 12,288MHz. Tento kmitočet je vybrán záměrně, neboť je to 256-násobek 48kHz, tj. kmitočtu hojně využívaného jako vzorkovací pro audio-signály. Kompenzační kondenzátory často připojované ke krystalovým oscilátorům mohou být v tomto zapojení vynechány, jsou již implementovány v obvodu a jsou plně programovatelné.

Výstupy mají impedanci okolo 32Ω, jsou k nim připojeny ještě rezistory 18Ω, aby výstupní impedance byla normalizována na 50Ω, což ovšem není příliš významné. Dále jsou k výstupům připojeny kondenzátory pro možnost doplnění na dolní propust omezující vysoké kmitočty. Jeden výstup by byl nezapojen, je proto vyveden na konektor BNC k možnému externímu využití.



Obr. 3.4 Schéma časovací jednotky

Programování syntezátoru umožňují 3 programovatelné piny. 2 z nich jsou defaultně nastaveny jako slave port I2C komunikace a na této desce jsou tak i využívány. Třetí vstup je defaultně nastaven jako enable pin všech výstupů a také je vyveden k procesoru. Tento pin snese pouze napěťové úrovně LVCMOS, je ale externě doplněn o obvod schopný zpracovat napěťové úrovně v širším rozsahu.

3.4 Napájení a fyzická realizace

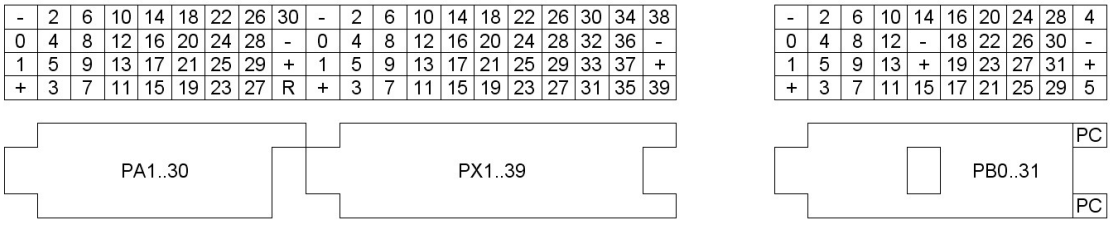
Pro správný běh všech obvodů je nutné zajistit různá napájecí napětí a oddělit od sebe napájení analogové a digitální části. Sluchátkový zesilovač vyžaduje napájení 5V, AD převodník také 5V, ale digitální a analogové, a syntezátor pracuje s napájecím napětím 1,8V a hradla výstupů jsou napájena 3,3V.

Deska EVK1100 standardně poskytuje napájení periferních obvodů 3,3V. 5V lze získat buď externě, k čemuž slouží napájecí konektor na rozšiřující desce, nebo jej lze vyvést dodatečně z desky za USB konektorem nebo napájecím konektorem EVK1100. POZOR, rozšiřující deska neobsahuje stabilizátor 5V, je tedy nutné, aby toto přiváděné napětí bylo stabilizované na maximálně 5,5V, jinak hrozí zničení obvodů. Napětí 1,8V je získáváno z 5V větve stabilizátorem LM1117-1.8.

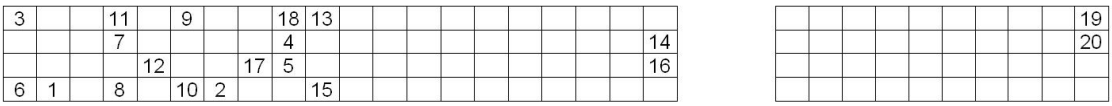
Oddělení napájení pro číslicové a analogové obvody je nutné pro snížení průchodu vysokofrekvenčního rušení. Rozdělit je nutno pouze 5V větev a zemnící plochu. Napájení pro analogové obvody je odděleno Π-článkem a zemnící plocha je spojena v jednom bodě nejblíže vstupu napájení. Zemní plochu je ještě vhodné doplnit prokovy v místech soustředění největšího rušení. Vzájemné ovlivňování je ještě omezeno kombinací blokovacích kondenzátorů nejblíže napájecím pinům jednotlivých integrovaných obvodů.

K propojení s vývojovou deskou slouží plochý kabel. Kit EVK1100 bylo ještě nutno opatřit redukcí, která umožní snadné propojení, avšak nevyžaduje příliš invazivní zásah. Mapa připojení kabelu je na obr. 3.6, číslování vodičů plochého kabelu je čitelné z osazovacího plánu desky v příloze B. Na obr. 3.5 je uvedeno mapování portů na redukci.

Celé schéma zapojení a obrazce plošných spojů jsou uvedeny v přílohách A a B.



Obr. 3.5 Mapa portů na redukci (pohled shora)



Obr. 3.6 Mapa připojení rozšiřující desky (pohled shora)

4 VÝVOJOVÉ PROSTŘEDÍ PRO AVR32

Vývojovým prostředím pro architekturu AVR32 je program AVR32 Studio od fy. Atmel. K naprogramování desky je využíváno JTAG programátoru JTAGICE mkII.

4.1 JTAGICE mkII

AVR JTAGICE mkII je adaptér od fy. Atmel pro odlaďování programu na čipu přes rozhraní JTAG nebo debugWIRE. Adaptér umožňuje sledovat paměti flash, EEPROM i SRAM, registrový soubor, programový čítač, fuse a lock bity a všechny vstupně výstupní moduly, a další možnosti odlaďování.

Adaptér se s počítačem propojuje buď přes RS232, přičemž potřebuje externí napájení v rozsahu 9-12V. Alternativou je komunikace přes USB, čímž je zajištěno i napájení. Propojení s deskou je realizováno plochým kabelem s koncovkou JTAG, ke které jsou dodávány i redukce.

Momentální stav a připojení adaptéru signalizují 3 LED. Červeně je signalizováno připojení napájení. Korektní připojení napájené desky je signalizováno zelenou LED. Poslední LED svítí v případě právě probíhající komunikace mezi deskou a PC.

Instalace ovladačů programátoru by měla proběhnout společně s instalací AVR32 Studia. V opačném případě jsou pro ruční instalaci všechny ovladače dostupné ve složce AVR Tools.

4.2 AVR32 Studio

AVR32 Studio je prostředí pro vývoj aplikací na platformě AVR32. Podporuje všechny procesory AVR32 a odlaďování obou typů aplikací, bez operačního systému a systémů na bázi Linuxu. Podporuje také programátory Atmel pro AVR32 jako je JTAGICE mkII.

Verze 2.6 je podporována na operačních systémech Windows 2000 a XP nebo Linux FEDORA, UBUNTU a SUSE. Nyní je již podporována i na 64-bitových operačních systémech. Ve vývoji je i nová verze AVR32 Studio 5.

Nejnovější verze je volně ke stažení na webových stránkách fy. Atmel nebo na DVD společně s AVR32 knihovnami. Nutností je pouze registrace na webu. Dále je rozebrán popis verze pouze pro Windows XP. Dodatečné informace jsou uvedeny v [8].

Instalace by měla probíhat bez větších problémů, je však nutno doinstalovat nástrojovou sadu AVR32 GNU toolchain, která je volně dostupná na webu Atmel. Dalším požadavkem je přítomnost novější verze Javy.

AVR32 Studio obsahuje podrobnou příručku, ve formě helpu, k používání jak samotného programu, tak i většiny programátorů a vývojových desek od fy. Atmel. Pro všechny tyto desky jsou předvytvořeny i demoaplikace, které je možno načíst, zkompilovat a odzkoušet tak na desce.

Nový projekt se zakládá položkou *File→New→AVR32 C Project From Template*. Zadá se název projektu a adresář pro uložení. Potom je nutné vybrat cílovou desku EVK1100 a osazený mikrokontrolér UC2A0512. Je možné zadat ještě několik doplňujících informací. Tímto je vytvořen projekt s GCC souborem s předpřipravenými direktivami *include* a funkcí *main*.

Přidání nového souboru, (např. *.c nebo *.h) lze jednoduše přes *File→New*. Soubor se automaticky přidá do otevřeného projektu a otevře se pro editaci. Přidání již existujícího souboru se provádí podobně přes *File→Import*.

Hlavičkové soubory je nutno přidat nejenom direktivou *include*, ale i do projektu. Soubor musí být přítomný v okně Project Explorer, kam se přidávají z menu Framework→Select Drivers/Components/Services. Tímto průvodcem se lehce vkládají potřebné knihovny ovladačů periferií, ovladačů vnějších periferií a programových služeb.

Překlad programu se spustí z menu *Project→Build Project*. Může pracovat ve dvou módech, *Release* a *Debug*. *Release* vytvoří holý zkompilovaný program pro „vypuštění“, zatímco *debug* do kódu přidá data nutná pro ladění, takže je program déle kompilován a je větší.

AVR32 Studio má k dispozici nástroje, díky nimž je schopno přímo komunikovat s programátorem a desku tak naprogramovat. Zařízení je nejprve nutno přidat pomocí položky *Scan Targets* a nakonfigurovat jej. Funkčnost signalizuje ikonka v okně AVR32 Targets. Konfigurace musí probíhat při připojeném a aktivovaném cílovém zařízení. Všechny akce jsou přístupné z menu tohoto okna. Program je uložen v souboru *.bin nebo *.elf v adresáři Debug projektu.

Před zahájením odlaďování projektu je ještě nutno nastavit zdrojový soubor v menu *Run→Debug Configurations…*, veškeré nastavení je provedeno automaticky při správném zvolení cílového zařízení. Pak už stačí jen spustit ladění tlačítkem *Debug* a odsouhlasit pár výstrah a režim ladění je zahájen. Takto lze libovolně krokovat program na reálném procesoru a nahlížet při tom volně do registrů. Po ukončení ladění se program přepne zase do standardního vývojového prostředí.

5 OVLADAČE ROZŠIŘUJÍCÍ DESKY

V této kapitole je popsána knihovna, která byla vytvořena za účelem zjednodušení práce s hardwarovým audio-rozhraním. Funkce v této knihovně se postarají o správné nastavení veškerých nutných periferií procesoru, nastavení obvodů rozšiřující desky a uvedení v činnost.

Veškeré funkce jsou zapsány v souboru *FZ102010.c*, k němuž náleží hlavičkový soubor *FZ102010.h*, kde jsou všechny nutné definice a hlavičky funkcí. Název knihovny je totožný s označením desky, je tak na první pohled jasná jejich sounáležitost Funkce knihovny jsou také doplněny o tento řetězec, není tak možná nechtěná záměna s funkcemi jiných knihoven. Knihovna využívá funkce standardních knihoven pro AVR32 platformu, ty je nutno mít samozřejmě k dispozici.

5.1 Ovladač kmitočtového syntezátoru

Na desce je osazen syntezátor CDCE913, jenž je ovladatelný po I2C sběrnici. Následující nastavení vychází především z datových listů obvodů [4] a [12].

5.1.1 I2C komunikace

Komunikace I2C je zajištěna pomocí modulu TWI implementovaného na procesoru, ten je plně kompatibilní s I2C. Funkce ovládající tento modul jsou k dispozici v knihovně *TWI* a není nutno je doplňovat.

Funkce *init_synth_FZ102010* se postará o nastavení GPIO portů, zapnutí TWI modulu a nastavení přenosové rychlosti. Poté zašle do syntezátoru nastavení velikosti blokovacího kondenzátoru oscilátoru. Syntezátor se pak rozběhne podle nastavení v EEPROM nebo defaultního nastavení.

Syntezátor má množství konfiguračních registrů, k nimž se přistupuje standardně pomocí 8-bitové adresy. Tabulka registrů je podrobně rozepsána v [12] ovšem potřebné registry jsou definovány jako symbolické názvy v knihovně.

Paket zápisu

S	Adresa slave	WR	A	Vnitřní adresa	A	Datový Byte	A	P

Paket čtení

S	Adresa slave	WR	A	Vnitřní adresa	A	S	Adresa slave	RD	A

Datový byte	A	P

Obr. 5.1 Datové pakety I2C

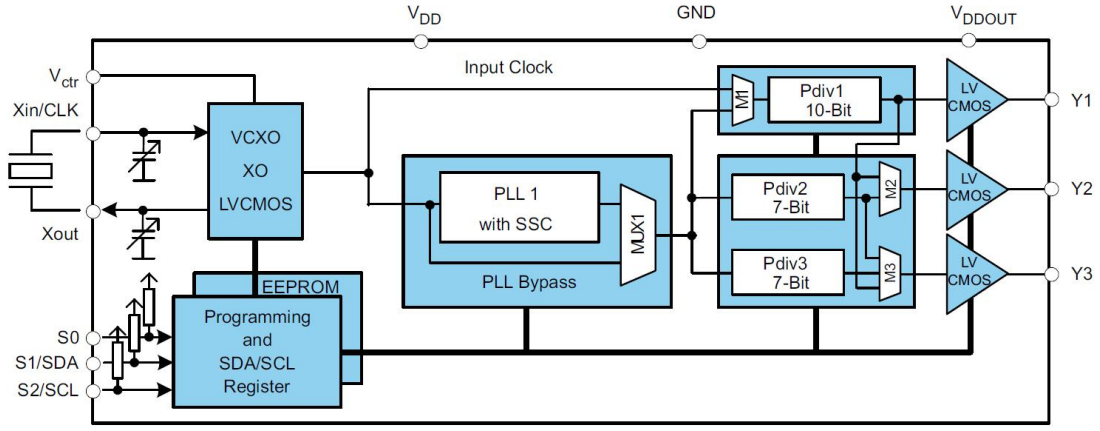
Data lze zasílat jak po 1 bytu, tak i po blocích. V tomto případě je zvolena komunikace po 1 bytu, jenž je mírně jednodušší, i když v některých případech pomalejší. Komunikační pakety jsou na obr. 5.1. Jsou vždy uvedeny start-bitem *S*, poté následuje 7-bitová adresa slave zařízení, 8. bit informuje, zda master zařízení chce číst nebo zapisovat. Tento byte musí být potvrzen *ACK* bitem. Následuje vnitřní adresa zařízení, tj. ad-

resa bytu v paměti slave, a je opět potvrzen příjem *ACK* bitem. Dále se komunikace liší, při zápisu je zaslán ještě datový byte. Čtení pokračuje zahájením nové komunikace start bitem a opětovným zasláním adresy slave zařízení s bitem *RD*. Od této chvíle master poslouchá a slave zašle požadovaný byte. Úspěšný přenos master potvrdí bitem *ACK*. Celá komunikace je v obou případech ukončena stop bitem *P*.

Veškerá konfigurace je takto uložena do paměti RAM, po restartu je tedy tato paměť smazána. Data je možno převést do EEPROM a tím zachovat nastavení i po restartu zařízení, tato funkce ovšem není využita.

5.1.2 Nastavení kmitočtu syntezátoru

Z obr. 5.2 je patrná bloková struktura syntezátoru. Zdrojem kmitočtu je oscilátor s externím krystalem. Dále je obvod vybaven jednou PLL smyčkou, jenž pracuje na kmitočtech v rozmezí 80-230MHz. Každý výstup obsahuje vlastní děličku. Celý systém je ještě doplněn několika multiplexery pro kombinování kmitočtů.



Obr. 5.2 Blokové schéma syntezátoru (převzato z [12])

Nastavení PLL jednotky je trochu složitější, než by se na první pohled mohlo zdát, přesto jej funkce *set_VCO_FZ102010* plně automatizuje. Vstupním parametrem je pouze požadovaný kmitočet PLL smyčky. Na začátku je nejprve zkontrolováno, zda je frekvence ve správném rozsahu. Zadáním neaplikovatelného kmitočtu je vráceno chybové hlášení. Nulou se PLL blok přemostí, přepíná tedy MUX1 do polohy PLL Bypass.

V prvním kroku se vstupní kmitočet přepočítá na násobek referenčního kmitočtu ve formě zlomku. Aby dělení probíhalo beze zbytku, musí se získat největší společný dělitel (NSD) obou kmitočtů, ten je získán funkcí *get_gcd_FZ102010*, jenž využívá Euklidův algoritmus. Jmenovatel i čitatel získaného zlomku nesmí být větší než 511 a 4095. Pokud NSD nesplňuje tyto podmínky, přejde se k úpravě vstupního kmitočtu. Přidává se postupně na obě strany rostoucí odchylka a počítá se další iterace, dokud nejsou podmínky splněny, nebo odchylka nepřekročí danou mez. Ve většině „rozumných“ kmitočtů postačí jedna iterace a chyba nastavení je tedy nulová, pokud přeci dojde ke korekci, je výsledná chyba nastavení zapsána do globální proměnné *SYNTH_SET_ERROR*. Získaný zlomek je předán funkci *send_PLL_FZ102010*, která pokračuje v přepočítávání. Zlomek je přepočítán na 4 parametry, které jsou poté zapsány do

paměti syntezátoru. Výpočet probíhá podle předepsaného postupu v [12]. Podle vstupního kmitočtu je nutno ještě nastavit registr rozsahu. Všechny tyto parametry jsou v paměti rozmístěny poněkud nešikovně, logické operace s proměnnými při vytváření datových paketů tak mohou být nepřehledné. Na závěr je ještě aktivován celý blok, a pokud vše proběhlo správně, funkce vrací nulovou hodnotu.

Další částí nastavení je již jednoduché nastavení jednotlivých děliček. O to se stará jediná funkce *set_Pdiv_FZ102010*. Vstupními parametry jsou dělitel a označení kýženého kanálu. Podmínkou je pouze aby dělitele byly v požadovaném rozsahu, pro první kanál 10-bitová hodnota, pro 2. a 3. kanál pak 7-bitová hodnota. Parametry jsou opět zaslány do paměti syntezátoru. V některých případech jeden datový byte obsahuje hodnoty, které se nesmí změnit, pak se před zápisem data nejprve čtou a změna se provede pouze na inkriminovaných bitech.

Přepnout multiplexery je možné funkcí *set_MPX_FZ102010*. Jako vstupní parametry jsou definovány symbolické konstanty umožňující všechny možné kombinace.

5.2 Ovladače audio rozhraní

AD převodník komunikuje s procesorem přes rozhraní I2S. Procesor ke komunikaci využívá univerzální jednotku SSC ve slave módu. Pro nastavení SSC je opět dostupná knihovna, ta ovšem neumožňuje dostatečně detailní nastavení, registry bylo tedy nutno nastavit přímo. Protože se předpokládá velká vytíženost procesoru, bylo snahou procesor odlehčit od ošetřování všech komunikací, proto byl AD převodník nastaven jako master. Zamezí se tak ztrátám informace a možnost vyvolání přerušení po dokončení přenosu zajistí plnou samostatnost periferií.

Převod DA zajišťuje jednotka ABDAC implementovaná přímo v procesoru. Její výstup je ve formátu digitální modulace, takže je třeba ještě externích obvodů na odfiltrování vysokofrekvenčních složek a posílení signálu. Tyto obvody ovšem nevyžadují žádnou obsluhu, proto jediným požadovaným nastavením je zapnutí jednotky, nastavení GPIO portů a nastavení časování. Převodník je zapnut jedinou funkcí, která je dostupná z framework knihoven. Nastavení portů je totožné s nastavením portů SSC jednotky a také je obslouženo dostupnými GPIO funkcemi. Časování převodníku je odlišné od časování procesoru, je využito externího hodinového signálu. Celé zpracování a přenos hodinového signálu zajišťuje generic clock jednotka, jenž je ovládána funkcemi z knihovny power management.

Oba převodníky vyžadují velké datové přenosy rovnoměrně rozložené v čase, což by výrazně zpomalovalo procesor. Z tohoto důvodu jsou k periferiím SSC a ABDAC připojeny 2 DMA kanály zajišťující přenos dat z periferie přímo do paměti. PDCA jednotka (DMA kanály) pracuje přímo s libovolně velkým a libovolně definovaným datovým prostorem v paměti. Pro každý kanál musí být definována dvě, ne nutně stejně velká, datová pole, s nimiž bude jednotka střídavě pracovat. Datová pole jsou definována uživatelem a ukazatele na ně jsou společně s jejich velikostmi přímo vstupními proměnnými funkcí *init_ADC_FZ102010* a *init_DAC_FZ102010*.

Jakmile je práce s jedním datovým blokem odvedena (blok je naplněn/vyprázdněn) automaticky se přejde ke stínovému poli, přičemž je vyvoláno přerušení. Přerušení je implicitně obslouženo jednoduchými handlery, v nichž se pouze provede přepnutí dato-

vého pole, vykonává tak střídání datových bloků, a změna příznaku *ADC_DATA_A_RDY* resp. *DAC_DATA_A_RDY*. Příznaky jsou globálními proměnnými a informují uživatele, který datový blok je momentálně k dispozici. Pokud by se pracovalo s datovým blokem, který zároveň obsluhuje PDCA jednotka, dojde ke znehodnocení informace a ke zkreslení signálu.

Poslední částí knihovny je velice užitečná funkce inicializace. Jejím účelem je nastavení vybraných módů, které jsou nejčastěji používané. Celá inicializace je tak provedena touto jedinou funkcí a nevyžaduje inicializaci jednotlivých jednotek zvlášť. Vstupním parametrem je struktura obsahující mód převodníků a ukazatele na datová pole a jejich velikosti.

Poslední, ale velice důležitou poznámkou je, že funkci *INTC_init_interrupts,* definovanou ve framework knihovnách, je možné volat jen jednou. Z tohoto důvodu není zahrnuta v inicializačních knihovnách a tak je nutné se před použitím AD a DA převodníků ujistit, že jsou přerušení inicializována.

Použití knihovny ovladačů je popsáno v aplikačním listu přílohy D.

6 KNIHOVNA DSP-LIB

Knihovna DSP-lib je vytvořena společností Atmel a je dodávána jako součást vývojového prostředí AVR32 Studio. Obsahuje funkce používané pro číslicové zpracování signálů, jako jsou např. základní operace s vektory nebo filtrační algoritmy.

Knihovna je dostupná hned v několika verzích podle stupně optimalizace. Možné je se tak zaměřit na nejrychlejší zpracování, nejmenší kód apod.

Funkcí je mnoho a jsou rozděleny do několika skupin:

- Filtrace
- Generování signálu
- Operátory
- Transformace
- Práce s vektory
- Váhování okny
- Pokročilé operace

Ne všechny funkce jsou potřebné pro zde zpracovávané aplikace, proto budou podrobněji probrány pouze vybrané skupiny.

Většina funkcí je dostupná ve dvou verzích, pro 16-bitové a pro 32-bitové operace. Seznam všech funkcí dostupných v DSP-lib je uveden v příloze E. Podrobnější informace ke knihovně je možné nalézt v [15] a [16]. Informace v těchto dokumentech jsou ovšem často nedostatečné, proto nejlepším manuálem je stále nápověda AVR32 studia.

6.1 Filtrace

Do této skupiny patří funkce pokročilejší práce s vektory jako jsou filtrační algoritmy a funkce s těmito postupy spojené. Nejdůležitější jsou zde FIR a IIR filtry.

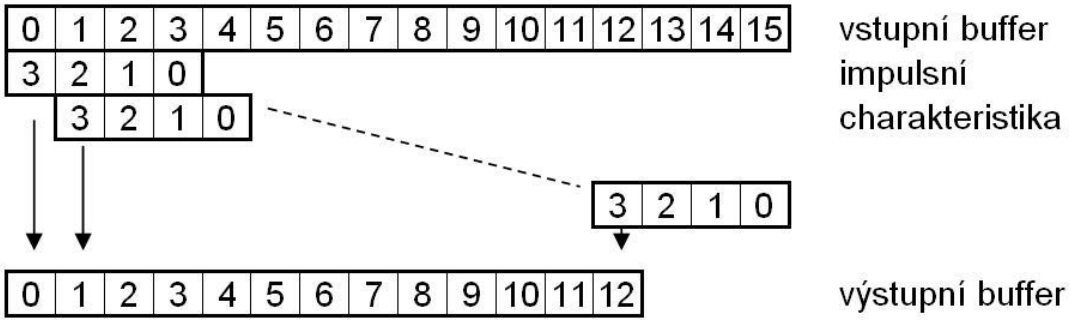
6.1.1 FIR filtr

Filtry s konečnou impulsní charakteristikou jsou zpravidla realizovány jako přímá konvoluce s impulsní charakteristikou. Stejně je tomu tak i u funkce *dspXX_filt_fir*, je to jen přepis volání funkce částečné konvoluce *dspXX_vect_convpart*.

Data jsou zpracovávána po blocích a nejsou ukládána „in place“, což znamená, že pro výpočet musí být vyhrazeno místo pro vstupní a výstupní buffer. Vstupními parametry jsou pak ukazatele na deklarovaná datová pole. Z realizačních důvodů musí být tato pole v paměti zarovnána na 32 bitů a velikost výstupního bufferu musí být násobkem čtyř. Při nedodržení těchto podmínek dojde s velkou pravděpodobností k chybě programu z důvodu přepsání jiné proměnné, jenž je v paměti umístěna těsně u bufferů.

Dalšími vstupními proměnnými je ukazatel na blok koeficientů, resp. impulsní cha-

rakteristiku, dále pak velikost tohoto bloku a velikost vstupního bufferu. Opět z realizačních důvodů musí být velikost impulsní charakteristiky větší jak 7.

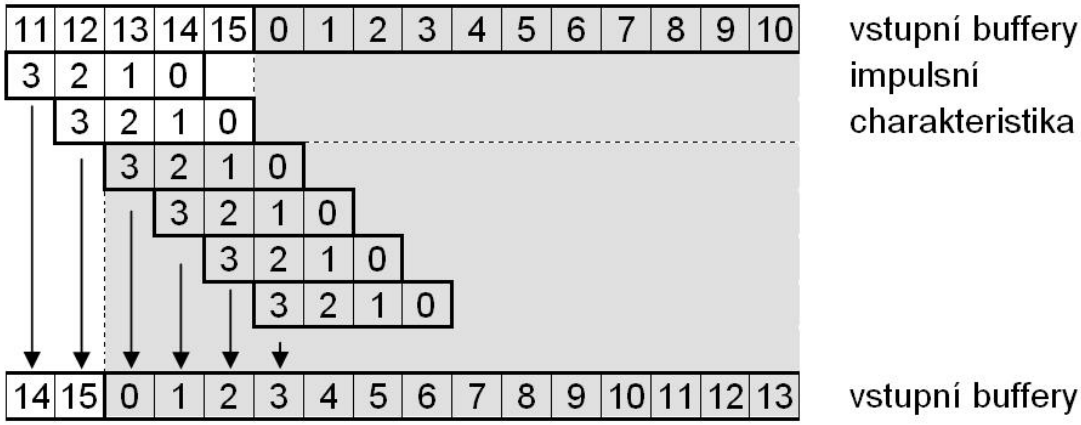


Obr. 6.1 Konvoluce při výpočtu FIR filtru

Na obr. 6.1 je ukázán příklad výpočtu konvoluce pro vstupní buffer velikosti 16 slov a o velikosti impulsní charakteristiky 4 slova. Vyplývá z něho, že velikosti vstupního a výstupního bufferu nejsou stejné, ale jsou závislé na velikosti impulsní charakteristiky podle vztahu (6.1).

$$buffer\_OUT\_size = buffer\_IN\_size - coef\_size + 1 \qquad (6.1)$$

Důvodem tohoto rozdílu je, že výpočet konvoluce je ukončen předčasně. Kdyby byl ovšem výpočet dotažen do konce, musel by se vstupní buffer na konci doplnit o $coef\_size - 1$ nul, čímž by se ale výsledek zkreslil a znemožnila by se tím možnost bezchybného navázání na další vstupní data.



Obr. 6.2 Navázání dat FIR filtru

Správné navázání dat je ukázáno na obr. 6.2. Vstupní buffer je na začátku doplněn o $coef\_size - 1$ dat z předchozího kroku, tím se výpočet dokončí vždy při následném volání.

K otestování funkce byla vytvořena identická funkce pro 16-bitová data v matlabu. Její název i vstupní proměnné jsou zavedeny identicky ke knihovně DSP-lib, pouze výstupní buffer je předáván jako návratová hodnota. Zdrojový kód je na zdr. kód 6.1.

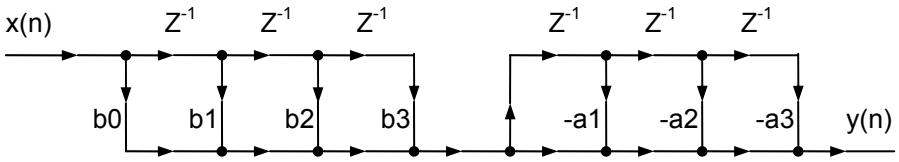
```
function [buffer_OUT]=dsp16_filt_fir(buffer_IN, size, h, h_size)

SAMPLES=size-h_size+1;
resolution=2^15-1;
buffer_OUT=zeros(1,SAMPLES);

for sample=1:SAMPLES
  for delayed=1:h_size
    buffer_OUT(sample)=buffer_OUT(sample)
        +round((buffer_IN(sample+delayed-1)
        *h(h_size-delayed+1))/resolution);
  end
end
```

Zdr. kód 6.1 Funkce FIR filtru pro matlab

6.1.2 IIR filtr

K výpočtu IIR filtru jsou v knihovně k dispozici dvě funkce. Funkce *dspXX_filt_iirpart* je určena pro jednorázovou filtraci a není vhodná pro real-time aplikace. Funkce *dspXX_filt_iir* je naopak určena pro sekvenční filtraci. Její algoritmus je založen na stejném postupu jako tomu bylo u FIR filtru, jen je její algoritmus komplikovanější o zpětnou vazbu a také vstupní buffery jsou definovány odlišně.

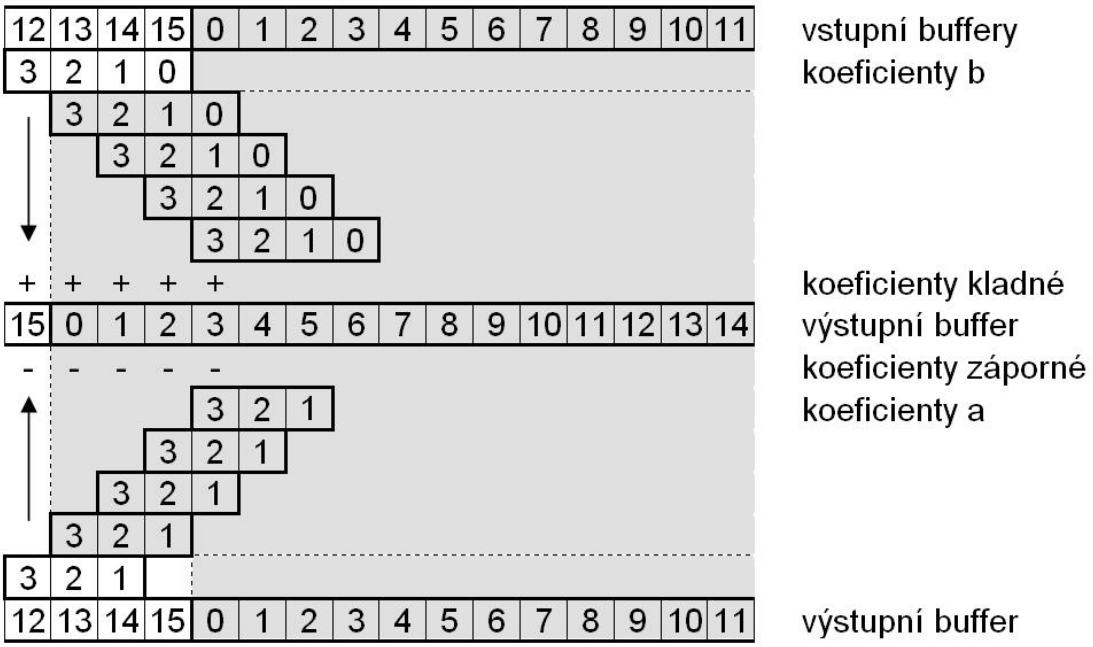
Signálová návaznost byla u FIR filtru zajišťována opakováním části dat pro dva kroky výpočtu. Zde tomu není jinak, a dokonce i zpětná vazba má tento princip shodný. Na obr. 6.3 je připomenut graf signálových toků IIR filtru 3. řádu.



Obr. 6.3 IIR filtr 3. řádu

Přímá část je počítána shodně podle obr. 6.2 (impulsní charakteristika je nahrazena koeficienty čitatele přenosové funkce), u zpětné vazby je navíc nutno si uvědomit, že jsou výpočty prováděny se stejným bufferem, do kterého jsou data postupně ukládána. Postup výpočtu je uveden na obr. 6.4. Koeficient a_0 je po normalizaci roven 1 a proto je vynechán, je to naznačeno indexací koeficientů od 1. V programu tedy pole koeficientů začíná koeficientem a_1, tj. na pozici 0 je koeficient a_1.

První dvě vstupní proměnné jsou ukazatele na výstupní a vstupní buffer. Buffery je nutné deklarovat o velikosti podle (6.2). Datová pole musí být v paměti zarovnána na 32 bitů. Filtr začne počítat z nulové pozice, čemuž na obr. 6.4 odpovídá sloupec nulového indexu vstupních dat. Tmavá část ovšem ukazuje, že v prvních krocích koeficienty zasahují na pozice před toto pole. Tato skutečnost je poněkud neobvyklá a matoucí, navíc je srozumitelně popsána pouze v nápovědě AVR32 studia!



Obr. 6.4 Příklad výpočtu IIR filtru

$$\begin{aligned} buffer\_IN\_size &= data\_size + coef\_B\_size - 1 \\ buffer\_OUT\_size &= data\_size + coef\_A\_size \end{aligned} \qquad (6.2)$$

Buffery jsou tedy deklarovány jako datová pole o velikosti podle (6.2), ale nejsou předávány jako ukazatele na první pozici, nýbrž jako ukazatele na první pozici nových dat, tj. podle (6.3).

$$\begin{aligned} ptr\_IN &= \&buffer\_IN[coef\_B\_size - 1] \\ ptr\_OUT &= \&buffer\_OUT[coef\_A\_size] \end{aligned} \qquad (6.3)$$

Jako třetí parametr je udávána velikost nových vstupních dat, tj. $data\_size$. Dále jsou předávány informace o koeficientech. Je uváděno, že je nutné, aby počet koeficientů *a* i *b* byl sudý. To je vzhledem k tomu, že koeficientů *a* bývá stejné množství jako *b* a po zanedbání a_0 pak bude vždy jeden lichý počet, zvláštní požadavek. Vysvětlení je nasnadě, datová pole musí být zarovnána na 32 bitů a to musí splňovat i ukazatele na buffery. Stačí tedy pouze zajistit, aby ukazatel ukazoval na sudou pozici deklarovaného 32-bitově zarovnaného pole a problém je odstraněn. V jednoduchých řešeních tak zůstane 16 bitů RAM zablokováno, ale nevyužito. Posledním požadavkem je nutnost deklarovat výstupní buffer o velikosti násobků 6, a to z optimalizačních důvodů.

Dva poslední vstupní parametry slouží k redukci velikosti koeficientů. Koeficienty mnoha filtrů vycházejí i několikrát větší než 1, zvláště koeficienty *a*, což je pro zlomkový tvar nepřípustné. Dále může vhodná kombinace způsobovat přetékání. Koeficienty je tedy vhodné předem zmenšit a informovat o tom filtr. Filtr poté provede násobení jednotlivých sum a výpočet pak odpovídá reálným koeficientům. Násobení je realizováno bitovým posuvem, takže dělitelé koeficientů musí být mocninami dvou.

Filtr byl opět testován v matlabu a byla vytvořena ekvivalentní funkce s maximálně podobnými vlastnostmi. Zdrojový kód je na zdr. kód 6.2.

```
function [vect1] =
  dsp16_filt_iir(
        vect1, size_v1,
        vect2, size_v2,
        num, num_size,
        den, den_size,
        num_prediv, den_prediv)

resolution=2^15;

for n=1:size_v2-num_size+1
  sum1=0;
  for m=1:num_size
        sum1=sum1+num(m)*vect2(n+num_size-m);
  end
  sum2=0;
  for m=1:den_size
        sum2=sum2+den(m)*vect1(n+den_size-m);
  end
  vect1(n+den_size)=(((sum1*2^num_prediv)
        -(sum2*2^den_prediv))/resolution);
end
```

Zdr. kód 6.2 Funkce IIR filtru pro matlab

6.2 Transformace

Knihovna poskytuje funkce pro výpočet rychlé fourierovy transformace. Obecně je to transformace komplexní, k tomu se vztahuje funkce *dspXX_trans_complexfft* pro převod z časové do frekvenční oblasti a *dspXX_trans_complexifft* pro transformaci opačným směrem. Vstupní signály získávané měřením nějaké veličiny, jako je např. audiosignál, mají často reálný charakter. To výpočet FFT zjednodušuje a urychluje, v knihovně je proto dostupná ještě funkce *dspXX_trans_realcomplexfft*. Bohužel zpětná reálná verze FFT není dostupná.

Použití těchto funkcí není již tak komplikované, jako tomu bylo u filtrů. Transformace opět nepočítá „in place“ takže je nutné založit vstupní a výstupní buffer. V závislosti na použité funkci musí být buffery specifického typu. Reálný vstup je pole stejného typu, jako tomu je u filtrů *dspXX_t*, což je při standardním nastavení AVR32 studia typ *short int* (16-bitový) resp. *int* (32-bitový). Komplexní data obsahují reálnou a imaginární složku, proto byla definována struktura *dspXX_complex_t*, jenž má právě složku *real* a *imag*, obě pak typu *dspXX_t*. Obě pole jsou stejné velikosti (stejný počet vzorků) zarovnaná opět na 32-bitů.

Pokud bude vyžadováno zpracovávání „real time“, pak při segmentaci bude nutné uvážit vhodné překrývání a váhování vstupního signálu.

Posledním vstupním parametrem je délka transformované sekvence. Výpočet je realizován algoritmem radix-4 decimace v čase, proto je schopen zpracovat pouze datovou posloupnost délky přirozených mocnin čtyř. Délka dat je zadávána jako mocnina

dvou, která tedy musí být sudá. To znamená omezení na délky $\{4,16,64,256...\}$ což může být v některých případech nepříjemné. FFT je například použita ve výpočtu koeficientů FIR filtru ekvalizéru. 256 koeficientů je pro takový ekvalizér málo a 1024 již může mírně zpomalovat výpočty. Algoritmus výpočtu není v této práci nutné popisovat, je analogický k radix-2, který je podrobně popsán např. v [17].

6.3 Ostatní funkce

Další skupiny funkcí jsou popsány již jen okrajově, neboť jejich použití je intuitivní, nebo nebudou dále aplikovány.

Skupina generování signálů obsahuje funkce, jež slouží k vytvoření posloupnosti signálu předem daného tvaru. Je dostupné poměrně velké množství funkcí zahrnující všechny základní průběhy používané v elektrotechnice včetně širokopásmového šumu. Všechny funkce jsou dostupné jak v 16, tak v 32-bitové verzi. Aplikace je srozumitelně popsána v [15].

Operátory jsou funkce zajišťující základní matematické operace nedostupné v základním GCC. Dostupné goniometrické funkce jsou nutné například k převodu mezi tvary komplexních čísel. Dostupné jsou i exponenciální a logaritmické funkce. Zajímavou funkcí je generátor náhodného čísla.

Pro vektorové operace je určeno také velké množství funkcí. Umožňují jednoduše realizovat základní operace s vektory, jako součet, rozdíl a různé druhy součinů. Dále také operace s komplexními vektory.

Další skupinou je soubor váhovacích oken. Jsou to funkce pro generování různých váhovacích oken v 16 nebo 32-bitovém zlomkovém formátu. Vytvořený vektor lze pak aplikovat na datovou posloupnost. Nejběžnější využití je ve spojení s transformacemi.

Poslední skupina obsahuje funkce pokročilejších operací, z nichž jsou zajímavé pouze funkce pro převzorkování.

7 3-PÁSMOVÝ GRAFICKÝ EKVALIZÉR

Grafické ekvalizéry slouží pro úpravu kmitočtových vlastností signálu. Ekvalizéry s malým počtem pásem se používají pro hrubé doladění nedokonalostí hudební soustavy a prostředí reprodukce. Analogové řešení takového filtračního obvodu není nikterak složité, i pro číslicovou realizaci je možné použít jednoduchých technik FIR filtrace.

7.1 Princip činnosti

Malá pásmová rozlišitelnost ekvalizéru dovoluje k filtraci použít FIR filtr, neboť nebude vyžadovat neúnosné množství koeficientů. Zároveň se tak využije jeho jednoduchosti a možnosti přímého výpočtu koeficientů. Odpadá starost o zajištění stability filtru.

Filtrace probíhá jako postupná konvoluce vstupního signálu s koeficienty filtru. Koeficienty jsou přímo vzorky impulsní charakteristiky, která odpovídá žádané frekvenční charakteristice převedené do časové oblasti. Pro reálné signály je požadavkem, aby i impulsní charakteristika byla reálná. Podle FT pro reálná data je zřejmá nutnost symetrie frekvenční charakteristiky. Návrh filtru tedy spočívá ve vytvoření žádané frekvenční charakteristiky, její symetrizaci prostým zrcadlením a zpětné fourierově transformaci pro získání ICH.

7.2 Výpočet koeficientů

Délka impulsní charakteristiky se odvíjí od požadavku na kmitočtovou rozlišitelnost ekvalizéru. Standardně využívané frekvenční pásmo pro hudební signály je 20Hz až 20kHz. Toto pásmo je logaritmicky rovnoměrně rozděleno na 3 části, dělicí kmitočty ukazuje vztah (7.1).

$$f_{\frac{n}{N}} = 10^{\left(\frac{n}{N}(\log(f_b)-\log(f_a))+\log(f_a)\right)} = f_a \cdot \left(\frac{f_b}{f_a}\right)^{\frac{n}{N}}$$

$$N \ldots pocet\ pasem$$

$$n \ldots poradi\ deliciho\ kmitoctu \qquad (7.1)$$

$$f_{\frac{1}{3}} = 20 \cdot \left(\frac{20000}{20}\right)^{\frac{1}{3}} = 200Hz$$

$$f_{\frac{2}{3}} = 20 \cdot \left(\frac{20000}{20}\right)^{\frac{2}{3}} = 2kHz$$

Rozlišení filtru určuje vztah (7.2). Je vztaženo k vzorkovacímu kmitočtu, s pomalejším vzorkováním roste rozlišovací schopnost. Pásmo hlubokých tónů je nutné ošetřit dostatečným počtem vzorků frekvenční charakteristiky a tím je tedy i určena nutná délka

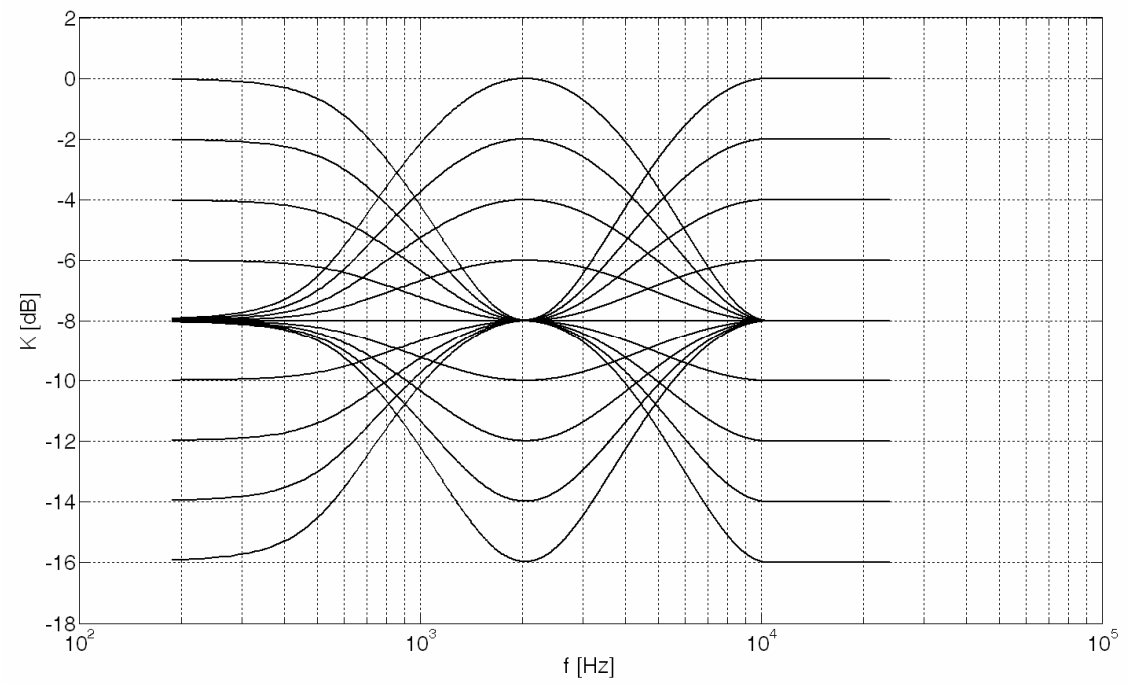
impulsní charakteristiky filtru. V příkladu je zvolena délka 256 vzorků, výpočet (7.3) ukazuje, jaké parametry bude filtr mít pro vzorkovací kmitočet 48kHz.

$$\Delta f = \frac{f_{vz}}{N} \tag{7.2}$$

$$\begin{aligned} \Delta f_{256} &= \frac{48000}{256} = 187.5Hz \\ &\downarrow \\ f_0 &= 0Hz \\ f_1 &= 187.5Hz \\ f_2 &= 375Hz \\ &\vdots \\ &\downarrow \\ n_{bass} &= 2 \end{aligned} \tag{7.3}$$

Výsledkem jsou 2 vzorky na pásmo hloubek, což je až příliž nízká hodnota. Řešením může být zvolení delšího filtru, nebo snížení požadavků. Spodní kmitočet děleného pásma je možné zvednout.

Díky lineárnímu rozložení vzorků frekvenční charakteristiky není v logaritmickém zobrazení strmost pásem stejná. Jednotlivá pásma jsou proto maskována podobným průběhem jako má hannovo okno. Pásma se překrývají tak, aby výsledné spektrum při stejném zesílení bylo rovnoměrné.



Obr. 7.1 Kmitočtové charakteristiky ekvalizéru

Výsledné kmitočtové charateristiky ekvalizéru jsou na obr. 7.1. Mezní kmitočty jsou vyšší, vzorkovací kmitočet je 48kHz. Útlum pásem je nastavován v logaritmické míře v rozsahu od 0 do -16dB.

Kmitočtová charakteristika je sestavována násobením příslušné masky koeficientem útlumu odpovídajícího nastavené hodnotě. Výpočtem zpětné FFT jsou získány koeficienty filtru, jež není nutné váhovat žádným oknem, neboť je výchozí frekvenční charakteristika dostatečně zaoblená.

7.3 Implementace na platformu AVR32

Funkce a konstanty efektu jsou uzavřeny do knihovny *music_3equalizer*, popis knihovny je v příloze G, bližší návod na sestavení programu je v příloze F.

Filtrace je realizována filtrem FIR z knihovny DSP-lib. Na každý kanál je aplikován jeden filtr, oběma jsou nastaveny stejné parametry filtrace, tj. i společná impulsní charakteristika. Ta je uložena v datovém poli *FIR_COEF[FIR_COEF_SIZE]*.

K nastavení ekvalizéru slouží funkce *set_coef*, která vypočte impulsní charakteristiku ze zadaných hodnot nastavení pásem. Vstupním parametrem je pole obsahující kroky útlumu pro každé pásmo, možnost nastavení je tedy od 0 do 8 pro 3 pásma. Nejprve je vytvořena kmitočtová charakteristika podle zdr. kód 7.1. Tvarovací masky pro každé pásmo jsou uloženy v globálních datových polích *band_1 – band_3*, ty jsou násobeny útlumovým koeficientem z globálního pole *band_amp*. Takto sestavená frekvenční charakteristika je uložena a zrcadlena jako reálná část do komplexního datového pole, imaginární část je nulována. Toto pole je vstupním parametrem následné zpětné FFT.

V DSP-lib jsou transformační funkce založeny na algoritmu radix-4, proto je nutné dodržet délku vstupní posloupnosti v mocninách 4. Výsledná impulsní charakteristika má pak také délku mocnin 4. Pokud není FIR filtr stejně dlouhý, musí se ICH vhodně decimovat.

Vypočtená ICH je přeskládána a uložena do pole *FIR_COEF*, filtr je tak aktualizován.

```
for (i=0; i<(FIR_COEF_SIZE>>1); i++)
{
    freq_char[i].real=band_amp[band_setting[0]]*band_1[i];
    freq_char[i].real+=band_amp[band_setting[1]]*band_2[i];
    freq_char[i].real+=band_amp[band_setting[2]]*band_3[i];
    freq_char[i].imag=0;
    freq_char[FIR_COEF_SIZE-i-1].real=
        band_amp[band_setting[0]]*band_1[i];
    freq_char[FIR_COEF_SIZE-i-].real+=
        band_amp[band_setting[1]]*band_2[i];
    freq_char[FIR_COEF_SIZE-i-1].real+=
        band_amp[band_setting[2]]*band_3[i];
    freq_char[FIR_COEF_SIZE-i-1].imag=0;
}
```

Zdr. kód 7.1 Vytvoření frekvenční charakteristiky

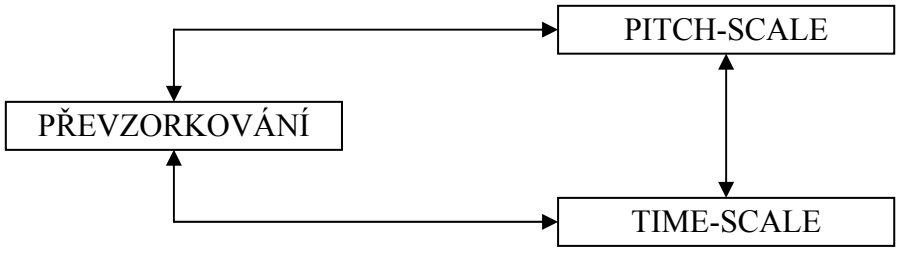
8 ZVUKOVÝ EFEKT „PITCH-SCALE“

Nezávislá obměna časového vývoje nebo kmitočtového zabarvení zvukového signálu je poměrně žádanou signálovou úpravou.

Změnou časového vývoje se myslí zpomalení nebo zrychlení signálové stopy při zachování její kmitočtové skladby. Tato úprava je často využívána například k synchronizaci zvukové a obrazové stopy ve videostudiích, nebo ke zkrácení času projevu např. v záznamnících nebo i reklamách. Tento efekt je v anglickém jazyce nazýván „time-scale“, pro omezení nesrovnalostí jsou anglické názvy dále zachovány.

Duálním efektem k time-scale je „pitch-scale“, je to naopak posunutí kmitočtového spektra signálu aniž by se změnil jeho časový vývoj. Využití je především jako hudební efekt v hudebních záznamových studiích, nebo pro úpravu řečových signálů. Transformací, která tyto efekty spojuje, je převzorkování. Jak je naznačeno na obr. 8.1, je možné převzorkováním realizovat jeden efekt pomocí efektu duálního.

Protože je tato práce zaměřena na real-time aplikace a i vzhledem k dualitě obou efektů je následně popsán pouze efekt pitch-scale.



Obr. 8.1 Dualita efektu pitch-scale a time-scale

8.1 Pitch-scale algoritmy

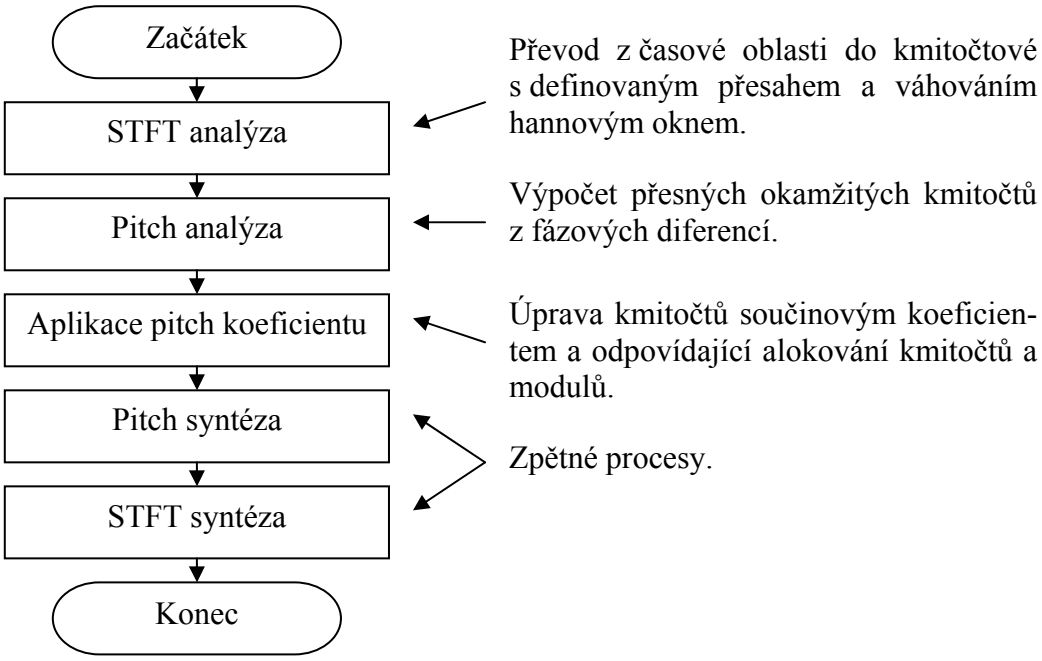
První přístroje realizující tyto efekty byly čistě mechanické a byly založeny na pravidelném opakování, nebo přeskakování částí záznamu na magnetofonové pásce. Touto metodou byly inspirovány algoritmy založené na zpracování v časové oblasti. Číslicová reprezentace spočívala v kruhovém registru, do něhož se zapisovalo rychlostí jinou než se kterou se z něho vyčítalo. Tyto algoritmy jsou snadno implementovatelné do číslicových systémů a jsou tak vhodné pro aplikace v reálném čase. Jedním z propracovanějších algoritmů v časové oblasti je PSOLA. Je atraktivní především pro zpracování řeči, neboť minimálně ovlivňuje formanty, jenž jsou klíčové pro zachování charakteru řeči.

S vývojem DSP algoritmů bylo umožněno modifikovat spektra v kmitočtové oblasti. Algoritmy jsou založeny na krátkodobé fourierově transformaci (STFT), která je výpočetně náročná, a proto nebyla implementace do real-time systémů zpočátku možná. Nyní jsou již k dispozici procesory s početními výkony dalece přesahující tyto požadavky. V této kapitole je popsáno, jakým způsobem je algoritmus pitch-scale

v kmitočtové oblasti implementován právě na platformu AVR32.

8.2 Algoritmus v kmitočtové oblasti

Základní myšlenkou úpravy v kmitočtové oblasti je posun bodů spektra v závislosti na vstupujícím „pitch“ koeficientu. Dochází tak ke zhuštění nebo roztažení spektra vstupního signálu. Algoritmus pitch-scale ve frekvenční oblasti popisuje obr. 8.2.

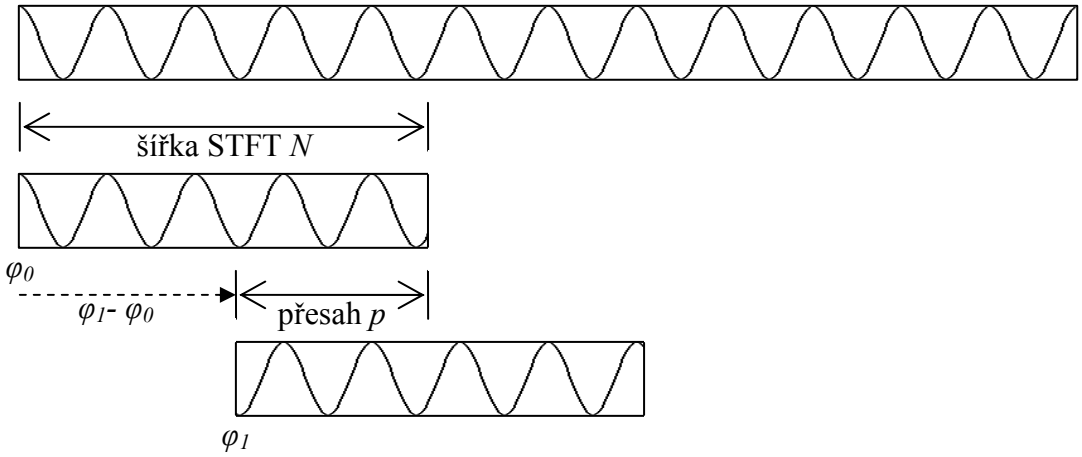


Obr. 8.2 Vývojový diagram pitch-scale efektu

Vstupní datová posloupnost je rozdělena na stejně velké datové bloky, jejichž velikost odpovídá šířce a tím i rozlišení STFT. Bloky se překrývají danou velikostí, přesahem. Tyto dva parametry významně ovlivňují kvalitu výsledného efektu a bývají různé pro různé typy signálu. Všechny datové bloky jsou postupně násobeny váhovacím oknem a převedeny do frekvenční oblasti. Výsledkem FFT dostupné z DSP-lib je komplexní spektrum v algebraickém tvaru, to je nutné ještě převést do polárního tvaru.

Získané spektrum je diskrétní v kmitočtu a jednotlivé složky odpovídají tedy přesně daným a přesně od sebe vzdáleným kmitočtům. Pokud se ve vstupním signálu vyskytuje kmitočet, který jim přesně neodpovídá, což je vzhledem k předpokládanému náhodnému charakteru signálu téměř 100% pravděpodobné, dojde k jeho rozprostření do více složek. Toto spektrum, přestože je plně rekonstruovatelné, nepodává přesnou informaci o kmitočtu a po posunutím složek spektra by došlo k nepřípustnému zkreslení díky neshodě fáze. Přesné kmitočty je ovšem možné vypočítat z rozdílů fáze dvou po sobě jdoucích bloků STFT. Postup je znázorněn na obr. 8.3. Rovnice (8.1) až (8.7) popisují výpočet přesných kmitočtů z takto získaných rozdílů fáze a zpětnou rekonstrukci. Výsledkem je vektor, v němž každé amplitudě spektra již není přidělena fáze, ale přesný kmitočet. V ideálním případě budou kmitočty složek, do nichž byl signál rozprostřen,

stejné. V reálu to závisí na velikosti přesahu, čím větší, tím přesnější, bohužel ale na úkor zhoršení jiných parametrů. Výpočet je veden přes odchylku od očekávané fáze (fáze odpovídající ekvidistantním kmitočtům STFT), proto jsou zavedeny vektory $\vec{f}_{map}$ a $\vec{\varphi}_{map}$. Při implementaci by zjednodušený postup mohl způsobovat zanedbávání násobků 2π a to by mělo za následek špatné určení přesného kmitočtu. Tabulka 8.1 a obr. 8.4 jsou spojeny s příkladem, který demonstruje vliv velikosti přesahu na přesnost určení okamžitého kmitočtu. Testovaný kmitočet byl 400Hz, šířka STFT 1024 a vzorkovací kmitočet 10kHz. Je pozorovatelné, že při malém přesahu jsou složky amplitudy rozprostřeny do kmitočtů okolo skutečného více než při velkém přesahu. S dále rostoucím přesahem ovšem dochází ke zúžení aktivní části bloku STFT pro výpočet rozdílu fáze a tím se snižuje rozlišovací schopnost. Tento jev se projevuje prudkým ujetím kmitočtu při přesahu větším jak cca 90%.



Obr. 8.3 Rozdíl fází z STFT

$$\vec{f}_i = \left[f_{i(0)}, f_{i(1)}, \cdots, f_{i(N-1)}\right] \quad (8.1)$$

$$f_{step} = \frac{f_S}{N}, \text{ kde } f_S \text{ je vzorkovací kmitočet a } N \text{ je šířka STFT} \quad (8.2)$$

$$\vec{f}_{map} = \left[0, f_{step}, 2\cdot f_{step}, 3\cdot f_{step}, \cdots, (N-1)\cdot f_{step}\right] \quad (8.3)$$

$$\vec{\varphi}_{map} = \left[0, 2\pi(1-p), 2\cdot 2\pi(1-p), \cdots, (N-1)\cdot 2\pi(1-p)\right], \text{ kde } p \text{ je přesah} \quad (8.4)$$

$$\varphi_{dif(n)} = \left(\varphi_{1(n)} - \varphi_{0(n)}\right) - \varphi_{map(n)} \quad (8.5)$$

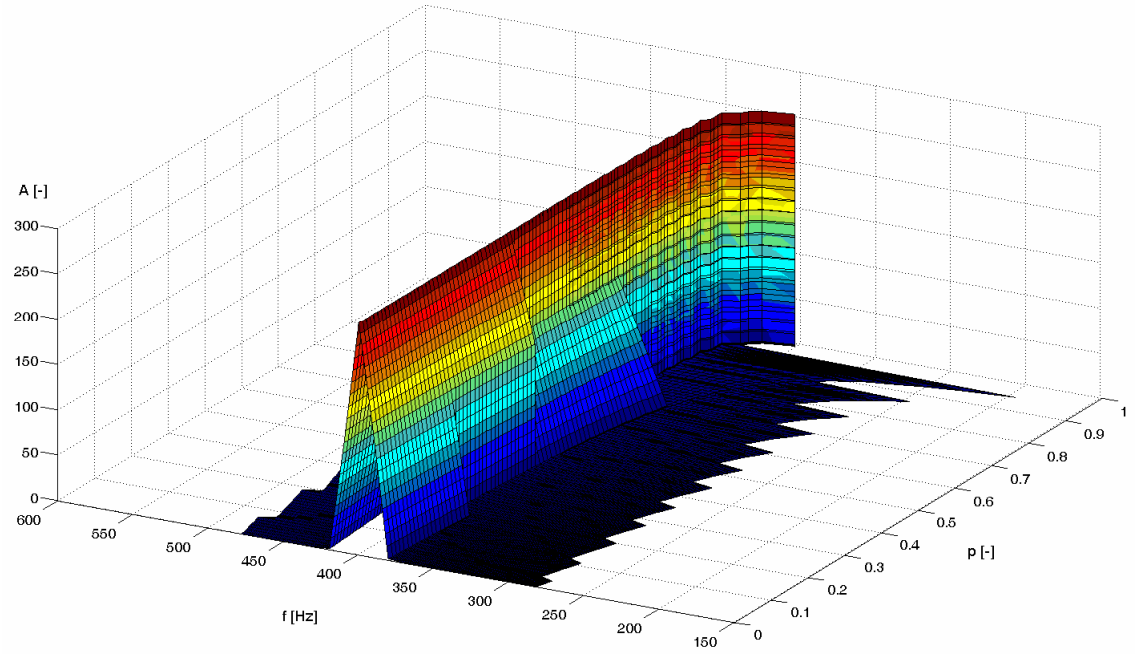
$$f_{i(n)} = f_{map(n)} + f_{step}\frac{\varphi_{dif(n)}}{2\pi(1-p)} \quad (8.6)$$

$$\varphi_{dif(n)} = \frac{f_{i(n)} - f_{map(n)}}{f_{step}} \left(2\pi\left(1-p\right)\right) \qquad (8.7)$$

Získané kmitočty je již možné zpracovat dle vstupního parametru. Zde je tímto parametrem pitch koeficient. Kmitočty se koeficientem vynásobí a přesunou na pozice, které jim odpovídají. Takto vytvořené vektory jsou připraveny ke zpětným procesům. Kmitočty se přepočítají zpět na fáze, čímž se získá komplexní spektrum. Spektrum se převede zpětnou FFT do časové oblasti, kde je blok podruhé váhován oknem a poté přičten k výstupní datové posloupnosti.

Tabulka 8.1 Vliv velikosti přesahu na přesnost určení okamžitého kmitočtu

Okamžitý kmitočet [Hz]										
Amplituda [-]		0,19	0,48	1,93	**135,74**	**255,49**	**120,42**	1,51	0,38	0,15
Přesah [%]	0	361	370	380	390	400	409	419	429	439
	30	358	372	386	386	399	413	413	427	441
	60	351	374	374	398	398	398	422	422	446
	70	367	367	367	398	398	398	430	430	430
	80	352	352	398	398	398	398	398	445	445
	90	394	394	394	394	394	394	394	394	483



Obr. 8.4 Vliv velikosti přesahu na přesnost určení okamžitého kmitočtu

8.3 Chyby určení okamžitého kmitočtu

Vzhledem k tomu, že zde jde o číslicové zpracování, potýkají se tyto metody s různými chybami. Diskretizace signálu v čase zavádí absolutní chybu fáze. Tato chyba se projeví kvantováním fáze spektra po provedené FFT. Velikost tohoto kvanta určuje (8.8), chyba fáze je potom poloviční.

$$\Delta_\varphi = 2\pi \cdot \frac{f}{f_S} \qquad (8.8)$$

Lineárně roste s kmitočtem a v maximu, kdy je ještě splněn vzorkovací teorém, dosahuje skok hodnoty π. Tato chyba rovněž způsobuje jednu z vlastností diskrétního spektra reálného signálu a to, že první vzorek spektra, odpovídající stejnosměrné složce, a vzorek odpovídající polovině vzorkovacího kmitočtu jsou vždy reálné.

Okamžitý kmitočet je závislý na změně fáze v celém bloku STFT, ne jen na odchylce od ekvidistantně předpokládané fáze. Odchylka je zatížena výše uvedenou absolutní chybou a velikost rozdílu fáze se mění také s kmitočtem. Relativní chyba fáze je potom konstantní pro celý kmitočtový rozsah, jak ukazuje (8.9).

$$\delta_\varphi = \frac{\Delta_\varphi}{\varphi} = \frac{2\pi \dfrac{f}{f_S}}{2\pi \dfrac{f}{f_S} \cdot N} = \frac{1}{N} \qquad (8.9)$$

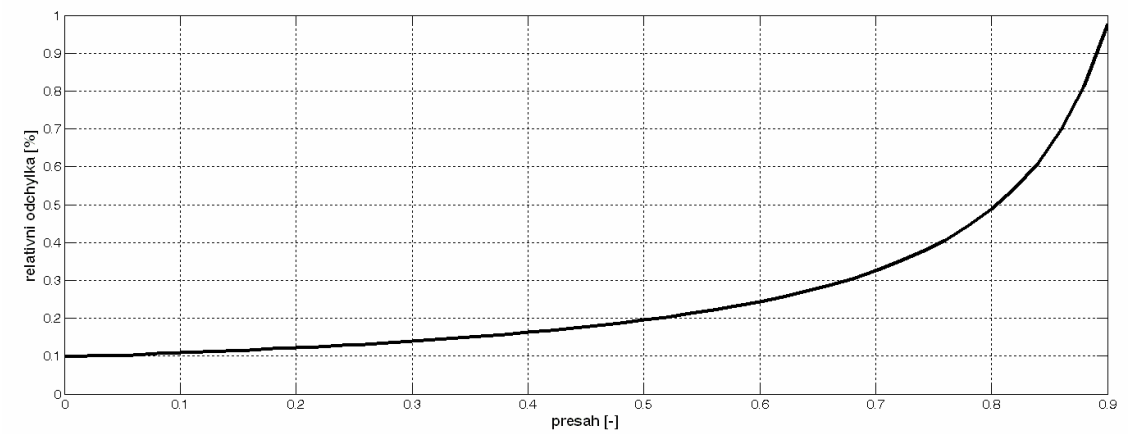
Zavedení přesahu do výpočtů ovlivní i tuto chybu. S přesahem se úměrně mění rozdíl fáze, absolutní chyba zůstává nezměněna. Výsledná relativní odchylka je potom (8.10).

$$\delta_\varphi = \frac{1}{N(1-p)} \qquad (8.10)$$

Relativní odchylka se vztahuje přímo na kmitočet podle (8.11).

$$\begin{aligned} f_i &= f_{map} + f_{step}\frac{\varphi_{dif}}{2\pi(1-p)} = f_{step}\frac{\varphi_{map}}{2\pi(1-p)} + f_{step}\frac{\varphi_{dif}}{2\pi(1-p)} \\ f_i &= f_{step}\frac{\varphi}{2\pi(1-p)} \\ &\downarrow \\ f_i + \delta_f \cdot f_i &= f_{step}\frac{\varphi}{2\pi(1-p)}\left(1+\frac{1}{N(1-p)}\right) = f_i(1+\delta_\varphi) \end{aligned} \qquad (8.11)$$

Jako příklad je závislost relativní chyby na velikosti přesahu uvedena na obr. 8.5 (FFT šířky 1024). Průběh přímo koresponduje s obr. 8.4, kde způsobuje právě ujetí kmitočtu při velkém přesahu.



Obr. 8.5 Relativní chyba v závislosti na velikosti přesahu

8.4 STFT analýza/syntéza

Signál převedený do kmitočtové oblasti dopřednou FFT je plně rekonstruovatelný zpětnou FFT. Krátkodobá analýza STFT je postupně počítána přes vybrané bloky vstupního signálu, ty jsou vybírány váhovacím oknem. Nejjednodušší obdélníkové okno pouze čistě vyřízne datový blok, vnáší tím ovšem do signálu nespojitost, která narušuje výsledné spektrum. To pak obsahuje složky, které se ve spojitém signálu nevyskytují a při úpravách způsobují zkreslení. Za účelem potlačení parazitních složek jsou k dispozici váhovací okna speciálních tvarů.

Zde využité Hannovo okno je popsáno rovnicí (8.12).

$$w_n = \frac{1}{2}\left(1-\cos\left(\frac{2\pi n}{N-1}\right)\right) \qquad (8.12)$$

Použití tvarovaného váhovacího okna vnáší do rekonstruovaného signálu zkreslení ve formě amplitudové modulace. Tuto modulaci je možné potlačit zavedením přesahu přesné velikosti. Pro Hannovo okno vyhovuje přibližně 50% přesah, jak ukazuje (8.13).

$$\begin{aligned}&\frac{1}{2}\left(1-\cos\left(\frac{2\pi n}{N-1}\right)\right)+\frac{1}{2}\left(1-\cos\left(\frac{2\pi\left(n+\frac{N}{2}\right)}{N-1}\right)\right)\cong \\ &\cong 1-\frac{1}{2}\left(\cos\left(\frac{2\pi n}{N-1}\right)+\cos\left(\frac{2\pi n}{N-1}+\pi\right)\right)=1\end{aligned} \qquad (8.13)$$

V tomto případě je signál rekonstruován bez většího zkreslení. Pitch-scale efekt ovšem, jak bylo ukázáno výše, neumožňuje libovolný výběr přesahu bez vlivu na fázové zkreslení. Pro potlačení šumů způsobených neodstranitelnými chybami fáze je váhování okny dvojité, druhá aplikace je prováděna při syntéze. Tím se mění i požadavek na přesah

(nyní okolo 66%), což mírně zlepšuje situaci.

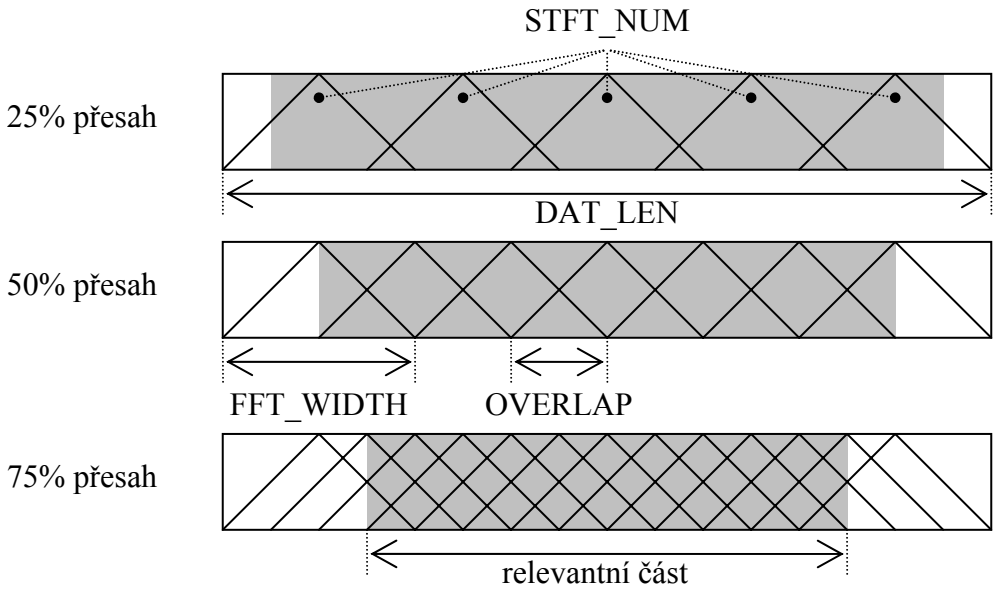
Dosud uvedené aspekty je nutné zvážit a zvolit určitý kompromis podle toho, jaký signál je zpracováván a které zkreslení je v konkrétním případě méně rušivé. Často uváděnou hodnotou přesahu je 75%.

8.5 Implementace na platformu AVR32

Efekt pitch-scale byl na platformu AVR32 implementován jako funkce zpracovávající blok vstupních dat definované velikosti. Funkce pracuje s velkým množstvím časově náročných výpočtů a její parametry proto musí splňovat přesné požadavky na úkor pohodlí uživatele. Všechny funkce a konstanty jsou uzavřeny ve vlastní knihovně *music_pitch*. Popis knihovny a návod k vytvoření projektu uvádí přílohy H a F.

Vstupními parametry funkce *pitch_scale* jsou ukazatele na vstupní a výstupní datový blok a samotný pitch koeficient, zbylé nutné parametry jsou zadány jako konstanty definované v hlavičkovém souboru. Vše bude následně podrobně popsáno, nejprve je však nutné probrat způsob zpracování datového bloku.

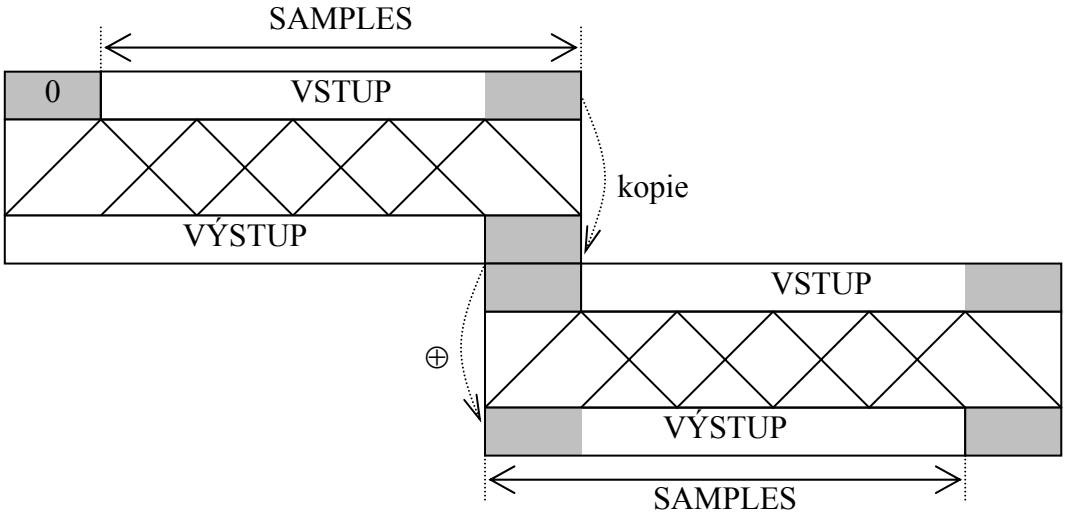
Obr. 8.6 ukazuje, jak probíhá dělení na STFT bloky, pro jednoduchost je aplikováno trojúhelníkové okno. Délka STFT bloku je dána šířkou FFT (*FFT_WIDTH*). Jelikož jsou jak pro analýzu, tak i pro syntézu využity funkce dostupné v DSP-lib, které využívají algoritmus radix-4, je třeba dodržet velikost STFT bloku v mocninách čtyř. Krok s jakým se okno STFT bloku posouvá je dáno přesahem (*OVERLAP*), který určuje procentuální překrytí sousedních STFT oken. Třetím a posledním parametrem, který je nutno před implementací definovat, je žádaný počet STFT provedených v jednom volání funkce, tj. počet STFT aplikovaný na jeden vstupní datový blok (*STFT_NUM*). Tato konstanta musí být sudá, proč, je uvedeno dále.



Obr. 8.6 Zpracování jednoho datového bloku

Z obr. 8.6 vyplývá, že výstup funkce není správný v celém rozsahu, ale pouze ve

zvýrazněných intervalech, je tedy jasné, že musí být zajištěn překryv datových bufferů, stejně jako tomu bylo u filtrů. Návaznost těchto polí je popsána na obr. 8.7. Konec vstupního bloku je kopírován a předkládán jako začátek následujícího vstupního bloku. K výstupnímu bloku je na začátku nutné přičíst konec předchozího výstupního bloku. Tato operace může být provedena ještě před voláním funkce, vzorky jsou do výstupního bufferu přičítány. To zároveň znamená, že výstupní pracovní buffer musí být uživatelem nulován.



Obr. 8.7 Návaznost datových bloků funkce

Konstanta *SAMPLES* udává počet nových vzorků pro jeden krok zpracování. *DAT_LEN* je velikost pracovního bufferu funkce, je vždy větší než *SAMPLES*! Obě konstanty jsou počítány implicitně překladačem a jsou velice důležité pro uživatele, neboť ony právě říkají, jak velká datová pole je pro práci nutné deklarovat.

Funkce využívá ještě další pomocné konstanty, které urychlují výpočty. *DATA_STEP* je prakticky počet vzorků, o které se posouvá výpočet STFT. *OVERLAP_SAMP* je přepočtené procento přesahu na počet vzorků STFT.

8.5.1 Reálná STFT

DSP-lib disponuje třemi funkcemi pro rychlou fourierovu transformaci. Dvě funkce plně komplexní pro dopřednou a zpětnou FFT a jednu dopřednou FFT pro reálná vstupní data. STFT analýza/syntéza prováděná v této aplikaci pracuje pouze s reálnými daty v časové oblasti, pro syntézu tedy není k dispozici potřebný prototyp FFT. Jednou z možností je využít pro syntézu plně komplexního výpočtu, kde imaginární část signálu bude nulová. Toto řešení je ovšem poněkud „humpolácké“ a zbytečně svou náročností prodlužuje výpočet.

$$S_k = \sum_{n=0}^{N-1} s_n e^{-j\frac{2\pi nk}{N}} = \sum_{n=0}^{N-1} s_n W_N^{nk} \qquad (8.14)$$

Matematicky je následně dokázáno, jak lze komplexní výpočet FFT využít efektivněji. Fourierova transformace je vyjádřena vztahem (8.14). Vstupní data jsou komplexní, jak je vyjádřeno ve vztahu (8.15). Postupnými úpravami lze pomocí dvou reálných signálů získat dvě komplexní spektra (8.16).

$$S_k = \sum_{n=0}^{N-1} \left(\mathrm{Re}(s_n) + j\,\mathrm{Im}(s_n)\right) W_N^{nk} \qquad (8.15)$$

$$\begin{aligned} &\mathrm{Re}(s_n) = f_n \\ &\mathrm{Im}(s_n) = g_n \\ &\qquad\downarrow \\ &S_k = \sum_{n=0}^{N-1} f_n W_N^{nk} + j\sum_{n=0}^{N-1} g_n W_N^{nk} = F_k + jG_k \end{aligned} \qquad (8.16)$$

$$\begin{aligned} F_k &= F_{N-k}^* \\ G_k &= G_{N-k}^* \\ S_k &= F_k + jG_k = F_{N-k}^* + jG_{N-k}^* \\ S_{N-k}^* &= F_{N-k}^* - jG_{N-k}^* = F_k - jG_k \end{aligned} \qquad (8.17)$$

Rovnice (8.17) připomínají vlastnosti spektra reálných signálů. Cílenými úpravami jsou získány následující vztahy (8.18).

$$\begin{aligned} F_k &= S_{N-k}^* + jG_k = S_{N-k}^* + S_k - F_k \rightarrow F_k = \frac{1}{2}\left(S_{N-k}^* + S_k\right) \\ G_k &= \frac{1}{j}\left(S_k - F_k\right) = \frac{1}{j}\left(S_k - S_{N-k}^* - jG_k\right) \rightarrow G_k = \frac{j}{2}\left(S_{N-k}^* - S_k\right) \end{aligned} \qquad (8.18)$$

Je zřejmé, že spektra obou reálných signálů jsou se spektrem na výstupu komplexní fourierovy transformace svázány jednoduchými vztahy.

V paměti procesoru jsou komplexní čísla ukládána samozřejmě samostatně po složkách, rovnice (8.19) a (8.20) již jen upravují vztahy tak, aby získání jednotlivých spekter bylo přímo aplikovatelné do procesoru.

$$\begin{aligned} &F_k = \frac{1}{2}\left(\mathrm{Re}(S_{N-k}) - j\,\mathrm{Im}(S_{N-k}) + \mathrm{Re}(S_k) + j\,\mathrm{Im}(S_k)\right) \\ &\qquad\downarrow \\ &\mathrm{Re}(F_k) = \frac{1}{2}\left(\mathrm{Re}(S_k) + \mathrm{Re}(S_{N-k})\right) \\ &\mathrm{Im}(F_k) = \frac{1}{2}\left(\mathrm{Im}(S_k) - \mathrm{Im}(S_{N-k})\right) \end{aligned} \qquad (8.19)$$

$$
\begin{aligned}
G_k &= \frac{j}{2}\left(\mathrm{Re}(S_{N-k}) - j\,\mathrm{Im}(S_{N-k}) - \mathrm{Re}(S_k) - j\,\mathrm{Im}(S_k)\right) \\
&\downarrow \\
\mathrm{Re}(G_k) &= \frac{1}{2}\left(\mathrm{Im}(S_k) + \mathrm{Im}(S_{N-k})\right) \\
\mathrm{Im}(G_k) &= \frac{1}{2}\left(\mathrm{Re}(S_{N-k}) - \mathrm{Re}(S_k)\right)
\end{aligned}
\qquad (8.20)
$$

Analýza STFT je tedy prováděna hned pro dva STFT bloky najednou, kde první je dosazen jako reálná část vstupu a druhý blok jako imaginární. Odtud pochází zmiňovaná nutnost sudosti konstanty *STFT_NUM*. Po FFT naznačený výpočet rozdělí spektrum. Syntéza je prováděna obdobným způsobem, spektrum je sestaveno podle vztahů (8.21).

$$
\begin{aligned}
S_k &= F_k + jG_k = \mathrm{Re}(F_k) + j\,\mathrm{Im}(F_k) + j\,\mathrm{Re}(G_k) - \mathrm{Im}(G_k) \\
&\downarrow \\
\mathrm{Re}(S_k) &= \mathrm{Re}(F_k) - \mathrm{Im}(G_k) \\
\mathrm{Im}(S_k) &= \mathrm{Re}(G_k) + \mathrm{Im}(F_k)
\end{aligned}
\qquad (8.21)
$$

Proti přetečení jsou funkce zřejmě zabezpečeny dodatečným váhováním, bitovým posuvem po každém výpočtu motýlka FFT. Toto váhování bezpečně ochrání proti přetečení, ale na úkor rozlišení výstupního spektra. V praxi při zpracovávání audio-signálů je zeslabení takto zpracovaného signálu velké a musí se zpětně kompenzovat násobením konstantou. Nicméně bitová hloubka výstupního signálu zůstává nízká a tím se dosti zhorší šumové vlastnosti celého systému. Tato vada by byla odstranitelná pouze úpravou algoritmu využité FFT, což ale není náplní této práce.

8.5.2 Konverze komplexního spektra

Získané spektrum je v algebraickém tvaru, jenže podle algoritmu je potřeba znát fázi jednotlivých složek. Je nutné převést spektrum do polárního tvaru. Matematický způsob ukazují vztahy (8.22).

$$
\begin{aligned}
|S| &= \sqrt{\mathrm{Re}(S)^2 + \mathrm{Im}(S)^2} \\
\varphi_S &= \arctan\left(\frac{\mathrm{Im}(S)}{\mathrm{Re}(S)}\right)
\end{aligned}
\qquad (8.22)
$$

DSP-lib ovšem nedisponuje funkcí arkus tangens, ale pouze arkus kosinus nebo arkus sinus. Výpočet fáze lze upravit na vztah (8.23).

$$
\varphi_S = \arcsin\left(\frac{\mathrm{Im}(S)}{|S|}\right) \qquad (8.23)
$$

Tyto vztahy byly aplikovány a testovány, ale neúspěšně. Jejich výpočet je příliš zdlou-

havý. Nejlepším nabízejícím se řešením bylo nahradit tyto vztahy CORDIC algoritmy.

CORDIC algoritmus pro převod algebraického tvaru komplexního čísla na polární je založen na iterační eliminaci imaginární složky. Jde o postupné snižování imaginární složky v absolutní hodnotě o určitou část, která odpovídá známé změně fáze. Pomocí součtových vzorců lze odvodit, že posunem fázoru o úhel φ se složky změní podle rovnic (8.24). Tyto vztahy je třeba poupravit na tvar (8.25).

$$\begin{aligned} \mathrm{Re}(S_{i+1}) &= \mathrm{Re}(S_i)\cos(\varphi) - \mathrm{Im}(S_i)\sin(\varphi) \\ \mathrm{Im}(S_{i+1}) &= \mathrm{Im}(S_i)\cos(\varphi) + \mathrm{Re}(S_i)\sin(\varphi) \end{aligned} \qquad (8.24)$$

$$\begin{aligned} \mathrm{Re}(S_{i+1}) &= \cos(\varphi)\cdot(\mathrm{Re}(S_i) - \mathrm{Im}(S_i)\tan(\varphi)) \\ \mathrm{Im}(S_{i+1}) &= \cos(\varphi)\cdot(\mathrm{Im}(S_i) + \mathrm{Re}(S_i)\tan(\varphi)) \end{aligned} \qquad (8.25)$$

Nyní je možné přistoupit k definování změn při každé iteraci. Nejvýhodnějším způsobem je zřejmě dělení dvěma, které je snadno realizovatelné bitovým posuvem. Úhel otočení se při každé iteraci musí zmenšovat (8.26).

$$\begin{aligned} &\tan(\varphi_i) = 2^{-i} \rightarrow \varphi_i = \arctan(2^{-i}) \\ &\qquad\downarrow \\ &\mathrm{Re}(S_{i+1}) = \cos(\arctan(2^{-i}))\cdot(\mathrm{Re}(S_i) - 2^{-i}\,\mathrm{Im}(S_i)) \\ &\mathrm{Im}(S_{i+1}) = \cos(\arctan(2^{-i}))\cdot(\mathrm{Im}(S_i) + 2^{-i}\,\mathrm{Re}(S_i)) \end{aligned} \qquad (8.26)$$

Po I iteracích je imaginární složka považována již za nulovou, reálná složka tedy odpovídá amplitudě (8.27). Fáze odpovídá součtu úhlů, které byly postupně odečítány a to s ohledem na znaménko (8.27).

$$\begin{aligned} &\mathrm{Im}(S_I) = 0 \\ &\qquad\downarrow \\ &|S| = \mathrm{Re}(S_I) \\ &\varphi_S = \sum_{i=0}^{I} \mathrm{sign}(\mathrm{Im}(S_i))\cdot\arctan(2^{-i}) \end{aligned} \qquad (8.27)$$

Z iteračního cyklu je možné vyjmout onu složitou funkci kosinus, úprava reálné a imaginární složky pak spočívá pouze v součtu nebo rozdílu a bitovém posuvu. Algoritmus poměrně rychle konverguje, takže pro 16-bitovou přesnost postačí již 14 cyklů. V tomto případě je pak velice snadné pro funkci arkus tangens vytvořit převodní tabulku, což zajistí maximální možnou rychlost. Funkce $\cos(\arctan(2^{-i}))$ je dokonce pro více jak 7 iterací v 16-bitovém rozlišení zaokrouhlena na 1, což znamená, že pro více jak 7 iterací je možné funkci K (8.28) nahradit konstantou k (8.29).

$$\begin{aligned} \mathrm{Re}(S_I)_{impl} &= \mathrm{Re}(S_{I-1}) - 2^{1-I}\,\mathrm{Im}(S_{I-1}) \\ |S| &= K(I)\cdot \mathrm{Re}(S_I)_{impl} \\ K(I) &= \prod_{i=0}^{I} \cos\left(arctg\left(2^{-i}\right)\right) = \prod_{i=0}^{I} \frac{1}{\sqrt{1-2^{-2i}}} \end{aligned} \qquad (8.28)$$

$$\begin{aligned} K(7) - K(8) &= 0{,}6072591 - 0{,}6072545 = 4{,}6\cdot 10^{-6} < 2^{-16} \\ &\downarrow \\ K(7) &\cong k = 0{,}6072540 = 39797\cdot 2^{-16} \end{aligned} \qquad (8.29)$$

8.5.3 Změna spektra

Úprava spektrálních vlastností signálu aplikací pitch koeficientu spočívá ve vynásobení získaných okamžitých kmitočtů a jejich přemístění na odpovídající pozice ve vektoru kmitočtů. Vektor amplitud je upraven pouze přemístěním složek paralelně s kmitočty.

Velikost koeficientu může teoreticky nabývat hodnot od 0 do nekonečna, prakticky je ale dobře použitelné rozmezí odpovídající posunu o jednu oktávu, tj. v intervalu od cca 0,5 do 2. Formát pitch koeficientu v implementované funkci je dán konstantou PITCH_FRACT, která udává počet bitů za desetinnou čárkou. Přednastavená hodnota konstanty je 14, tj. interval $\langle 0, 4)$.

Pitch koeficient P je na okamžitý kmitočet aplikován podle rovnice (8.30). Okamžitý kmitočet je počítán podle rovnice (8.31). Tento vzorec je pro rychlý výpočet poměrně složitý, neboť dělení musí probíhat iteračně a trvá tak dlouhou dobu. Naštěstí je možné jej zjednodušit.

$$\begin{aligned} r &= P\cdot n \\ f_{i(r)} &= P\cdot f_{i(n)} \end{aligned} \qquad (8.30)$$

$$\begin{aligned} f_{i(n)} &= f_{map(n)} + f_{step}\,\frac{\varphi_{dif(n)}}{2\pi(1-p)} \\ &= f_{step}\left(n + \frac{\varphi_{dif(n)}}{2\pi(1-p)}\right) \end{aligned} \qquad (8.31)$$

Zpětná rekonstrukce diference fáze je pak následovná (8.32).

$$\begin{aligned} \varphi_{dif(r)} &= \left(\frac{f_{i(r)}}{f_{step}} - r\right)(2\pi(1-p)) \\ &= \left(P\left(n + \frac{\varphi_{dif(n)}}{2\pi(1-p)}\right) - r\right)(2\pi(1-p)) \end{aligned} \qquad (8.32)$$

Po úpravách je výsledný vztah (8.33) mnohem jednodušší a lépe implementovatelný.

$$\varphi_{dif(r)} = 2\pi(1-p)(P \cdot n - r) + P \cdot \varphi_{dif(n)} \qquad (8.33)$$

9 OPERAČNÍ SYSTÉM FREE RTOS

Operační systém FreeRTOS je licencován pod GNU General Public licence. Pro komerční využití je možné jej využít právě pod touto licencí. Pro nekomerční účely je tento OS volně dostupný a je často implementován ve vývojových prostředích jako knihovna. Tak je tomu i u AVR32 Studia, kde je do projektu vkládán jako služba pomocí frameworku.

FreeRTOS sestává z několika knihoven, které obsahují všechny potřebné funkce. Rozděleny jsou pro přehlednost a modulárnost do skupin tasky, semafory a fronty. V této kapitole jsou stručně popsány principy RTOS a funkce, které jsou pro ně potřebné, dále pak využití RTOS s vytvořeným audio-rozhraním a aplikace hudebních efektů. Popis vytvoření konkrétních aplikací je uveden v příloze F.

9.1 Principy RTOS

Multitasking je prakticky jediným schůdným východiskem při vytváření složitých aplikací. Výkony i levnějších procesorů embeded systémů jsou dnes naprosto postačující pro implementaci jednoduchých operačních systémů.

Koncepci celého programu s RTOS tvoří malé podprogramy, tasky, které svou pozornost zaměřují pouze na určitou část systému. Taskům je nadřazen plánovač, který jim na omezený časový interval podle daných pravidel přiděluje výkon procesoru.

Task tvoří obyčejná funkce bez návratové hodnoty s předepsanými vstupními parametry s nekonečnou smyčkou. Aby se z funkce stal task, je nutné jej do RTOS zavést funkcí zdr. kód 9.1.

```
portBASE_TYPE xTaskCreate
(
      pdTASK_CODE pvTaskCode,
      const signed portCHAR * const pcName,
      unsigned portSHORT usStackDepth,
      void *pvParameters,
      unsigned portBASE_TYPE uxPriority,
      xTaskHandle *pxCreatedTask
)
```

Zdr. kód 9.1 Funkce vytvoření tasku

```
void vTaskDelete
(
      xTaskHandle pxTaskToDelete
)
```

Zdr. kód 9.2 Funkce odstranění tasku

Tímto je pro task alokován prostor v operační paměti, nastavena priorita spouštění a funkce je zavedena do tabulky plánovače. Alokace paměti je dynamická, takže je

možné task vytvořit libovolně při běhu programu, zároveň je ale alokace časově náročnější. Přesnější popis zde uváděných funkcí lze nalézt v [22]. Task je možné stejným způsobem i odstranit funkcí zdr. kód 9.2 a uvolnit tak paměťový prostor.

Přepínání jednotlivých tasků může probíhat několika způsoby a podle toho se i operační systémy dělí na:

- **Kooperativní**, kde se tasky samy vzdávají procesoru a předávají ho zpět plánovači, jenž spustí další task.
- **Preemptivní**, kde je procesor pod plnou kontrolou OS a tasku ponechává pouze určitý čas. Task je pak bez dotazu zmražen a pokračuje další task.
- **Kombinace předchozích typů.**

FreeRTOS je založen na preemptivním přístupu, ale umožňuje i ostatní přístupy. Vytížení tasků často nebývá úplné, musí se čekat na vstupní data např. z periferií, nebo je nutné pouze počkat určitý interval. V této době není procesor vytížen a může být využit pro jiné tasky. Task přejde do blokovaného stavu, uvolní procesor a plánovač předčasně pokračuje ve střídání tasků. Tasky se v programu mohou nacházet v těchto stavech:

- **Běžící** – Task, jenž je právě vykonáván.
- **Blokovaný** – Dostal se do stavu, kdy musí počkat na určitou událost, nebo jen počkat určitý časový interval, a uvolnil proto procesor pro jiné tasky.
- **Čekající** – Je připraven ke spuštění a pouze čeká, až na něho dojde řada a plánovač jej spustí.
- **Pozastavený** – Dočasně vyřazen z plánovače, není tedy spouštěn dokud ho nějaký jiný task znovu nespustí. Pozastaven může být jiným taskem, nebo i sám sebou.

Plánovač významně ovlivňuje systém priorit. Každému tasku je při vytváření uživatelem přidělena priorita. Při přepínání úloh plánovač vybírá z čekajících tasků právě podle priority. Nejprve spouští tasky s nejvyšší prioritou. Pokud jsou v čekajícím stavu dva nebo více tasků se stejnou prioritou, je spuštěn právě ten, který čeká delší dobu. Priorita tasků může být měněna i za běhu funkcí zdr. kód 9.3.

```
void vTaskPrioritySet
(
      xTaskHandle pxTask,
      unsigned portBASE_TYPE uxNewPriority
)
```

Zdr. kód 9.3 Funkce nastavení priority tasku

Plánovač musí mít vždy k dispozici task, který je možné spustit. Sám si tedy při spuštění vytvoří tzv. Idle task, který tvoří prakticky jen nekonečná smyčka. Idle task má vždy nejnižší možnou prioritu (0). Idle task sám může spouštět jednoduché funkce, tzv. Hook funkce, které jsou spouštěny jednou za cyklus smyčky Idle tasku. Díky Hook funkcím je možné např. měřit vytížení procesoru.

Předávání dat mezi tasky je možné uskutečnit pomocí globálních proměnných, tak

ale není přímo jasné, kdy jsou data určena k vyzvednutí. Elegantnějším řešením je zavedení fronty. Funkcí zdr. kód 9.4, se vytvoří zásobník, do kterého task uloží buď samotná data určená pro jiný task, nebo, v případě rozsáhlejších dat, ukazatel. Příjemce si poté tato data z fronty vyzvedne funkcí zdr. kód 9.6 a je jasné, že jsou data aktuální. Zápisu do nebo čtení z fronty je možné nastavit čekací dobu od 0 do nekonečna, tj., při nulovém intervalu task zkontroluje frontu a pokud je prázdná, pracuje dál, při nenulovém intervalu task přejde do blokovaného stavu a čeká po dobu tohoto intervalu. Probuzen je buď vyprázdněním/naplněním fronty, nebo po uplynutí intervalu.

```
xQueueHandle xQueueCreate
(
      unsigned portBASE_TYPE uxQueueLength,
      unsigned portBASE_TYPE uxItemSize
)
```

Zdr. kód 9.4 Funkce vytvoření fronty

```
portBASE_TYPE xQueueSendToBack
(
      xQueueHandle xQueue,
      const void *pvItemToQueue,
      portTickType xTicksToWait
)
```

Zdr. kód 9.5 Funkce zaslání na konec fronty

```
portBASE_TYPE xQueueReceive
(
      xQueueHandle xQueue,
      const void *pvBuffer,
      portTickType xTicksToWait
)
```

Zdr. kód 9.6 Funkce vyzvednutí z fronty

Embeded systémy vyžadují práci s přerušením a aplikace pak přímý přístup k jejich obsluhám. FreeRTOS umožňuje libovolné ovládání přerušení s výjimkou funkčně nutných. K synchronizaci tasků přerušením jsou používány semafory, základním typem je binární semafor. Jeho funkce je podobná frontě, pouze nepředává žádná data. Jakmile je semafor aktivován, task, jenž je semaforem ovládán, přejde z blokovacího stavu do čekajícího. Pokud má tento task vyšší prioritu, než ten, který byl přerušen, měl by být přeřazen plánovač, aby prioritní task nečekal. Funkce pro ovládání semaforu ukazují zdr. kód 9.7, zdr. kód 9.8 a zdr. kód 9.9.

Čítací semafory jsou schopny zaznamenat několik přerušení najednou.

```
void vSemaphoreCreateBinary
(
      xSemaphoreHandle xSemaphore
)
```

Zdr. kód 9.7 Funkce vytvoření binárního semaforu

```
portBASE_TYPE xSemaphoreGiveFromISR
(
      xSemaphoreHandle xSemaphore,
      portBASE_TYPE *pxHigherPriorityTaskWoken
)
```

Zdr. kód 9.8 Funkce aktivace semaforu

```
portBASE_TYPE xSemaphoreTake
(
      xSemaphoreHandle xSemaphore,
      portTickType xTicksToWait
)
```

Zdr. kód 9.9 Funkce převzetí semaforu

Dalším problémem, který multitasking přináší je nutnost organizovaného sdílení periferií. Pokud si přejí jednu periferii (display, sériová linka…) využít současně dva nebo více tasků, může dojít ke kolizi. Je nutné zajistit vzájemnou vylučitelnost. FreeRTOS nabízí tři způsoby:

- **Kritická část a pozastavení plánovače** – Základní postupy, které vyřadí operační systém z činnosti a nemůže tak dojít k přepnutí tasku. Zastavením OS se někdy vypínají i přerušení a to může působit problémy.
- **MUTEX** – Speciální druh semaforu, při němž se předává MUTEX. Task, který vlastní MUTEX, má přístup k periferii. Po dokončení práce task periferii uvolní vrácením MUTEXu.
- **Gatekeeper** – Je to task, který periferii vlastní a pouze přijímá požadavky k odeslání dat.

O MUTEX může požádat task, právě když jej vlastní task s nižší prioritou, dochází tak k inverzi priorit. Pokud se do toho připlete ještě task se střední prioritou, může se stát, že task s vysokou prioritou nikdy nedostane MUTEX. Tento nedostatek částečně řeší dědičnost priorit. Task, který právě vlastní MUTEX dědí vyšší prioritu tasku čekajícího na MUTEX. Pokud tedy na MUTEX čekají dva tasky, oba s vyšší, ale rozdílnou prioritou, po vrácení MUTEXU je task preemptivně přerušen pouze prioritním taskem. Task se střední prioritou tak nemůže ihned dostat MUTEX.

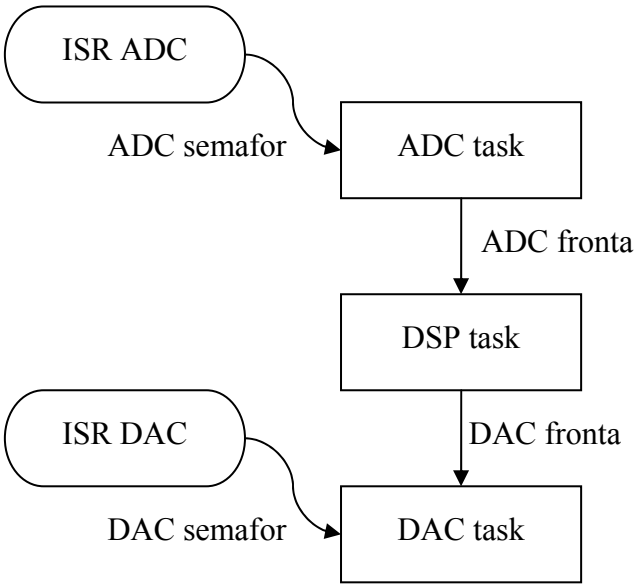
MUTEXy mohou způsobit také „deadlock“ a to v případě, že dva tasky vyžadují dva MUTEXy a task je preemptivně přerušen v nevhodnou chvíli. Každý task pak vlastní jeden MUTEX a zároveň čeká na uvolnění druhého MUTEXu.

Těmto komplikacím lze v jednodušších embeded systémech předcházet prostým zvážením situace a programovými úpravami. Ve složitějších systémech se již vyplatí použití gatekeeperu.

9.2 Aplikace RTOS pro ovládání audio-rozhraní

Ovládání audio-rozhraní je pro RTOS rozděleno na 3 základní části, příjem dat z ADC, zpracování dat DSP a odesílání dat na DAC. Celou koncepci popisuje obr. 9.1.

Přerušení od ADC a DAC ovládají semafory, proto musí být knihovna *FZ102010* mírně upravena doplněním náležitostí se semafory spojenými. Pro RTOS je tak určena knihovna *FZ102010_RTOS*. Deklarace pracovních polí pro DMA kanály zůstává stejná, pouze inicializace rozhraní je prováděna odlišně. Po inicializaci se ihned rozběhnou přerušení a DMA kanály pracují s datovými poli, vyžadují tedy obsluhu semaforů. Obsluha semaforů je aktivní až s běžícím plánovačem, proto k inicializaci rozhraní slouží samostatný task s vysokou prioritou, který se na konci sám odstraní.



Obr. 9.1 Vývojový diagram ovládání audio-rozhraní

Dojde li k přerušení od ADC DMA kanálu, obsluha obstará přepnutí pracovních bufferů a nastaví ADC semafor. Na tento semafor čeká ADC task, který přeskládá data dle uživatelských požadavků, následně předá ukazatel na zpracovaná data do ADC fronty. Na data z fronty čeká DSP task, který má za úkol vykonat veškeré požadované DSP operace. Tyto dva tasky jsou takto zřetězeny a mohly by tvořit pouze jeden task, ale pro přehlednost byly operace rozděleny. Jakmile jsou data zpracována, předá je opět pomocí ukazatele do fronty DAC. DAC task čeká na dvě události, nejprve očekává data ve frontě a jakmile jsou k dispozici, čeká na semafor od DAC přerušení. Poté zapíše výstupní data do DMA bufferu. Je tak zajištěna nemožnost kolize vlivem čekání na data. ADC a DAC tasky by měly mít vysokou prioritu, aby při přepisu bufferů nebyly přerušeny jiným nedůležitým taskem, což by mohlo způsobit chyby ve výstupním signálu.

Před rozběhem plánovače musí proběhnout inicializace. Nejprve je nastaveno časování jádra a inicializace přerušení, alokovány fronty a semafory. Semafory je ještě nutné zavést do ovládací knihovny, aby je mohla přerušení využívat. Poté jsou vytvořeny tasky. Velikost alokované paměti zde postačuje minimální, ovšem DSP task zde nemá žádnou funkci, ve složitějších aplikacích už je nutné alokovat dostatečný paměťový prostor. Poslední funkce již spustí plánovač. První se provede task s nejvyšší prioritou, tj. inicializace rozhraní, čímž se aktivují přerušení a semafory pak spouští příslušné tasky.

9.3 3-pásmový grafický ekvalizér v RTOS

Předchozí aplikace pro ovládání rozraní je doplněna o výkonnou část kódu DSP tasku a ovládací task. DSP task je doplněn o zdr. kód 9.10. Jsou to dvě stejné funkce z knihovny ekvalizéru pro oba kanály. Funkce je upraveným prototypem filtru z knihovny DSP-lib. Vstupní buffery musí být rozšířeny o *FIR_COEF_SIZE*-1. Konstanta je definována v hlavičkovém souboru knihovny.

```
/*filter buffer*/
equalizer_3(
    data_out.left,
    data_in.left,
    SAMPLES
);
equalizer_3(
    data_out.right,
    data_in.right,
    SAMPLES
);
/*fill the overlap*/
dsp16_vect_copy(
    data_in.left,
    &data_in.left[SAMPLES],
    FIR_COEF_SIZE-1
);
dsp16_vect_copy(
    data_in.right,
    &data_in.right[SAMPLES],
    FIR_COEF_SIZE-1
);
```

Zdr. kód 9.10 Filtrace v DSP tasku

Ovládací task se stará o periodické čtení tlačítek a joysticku, zobrazuje vybranou funkci na LED a počítá nové koeficienty FIR filtru podle zadání. Pokud dojde ke změně parametru, tj. hlasitosti některého z pásem, task změní hodnoty pole *eq_set* a zavolá funkci *set_coef*. Tato funkce podle těchto parametrů přepočítá impulsní charakteristiku a uloží ji do vnitřního pole *FIR_COEF*. Priorita tasku je nízká, požadavek není časově kritický. Perioda opakování je nastavena na 100ms funkcí *vTaskDelay*, která task uvádí do blokovaného stavu. Alokovaná paměť pro task je určena odhadem z proměnných, které jsou zde využity, nejnáročnější částí je právě funkce *set_coef*.

9.4 „Pitch-scale“ efekt v RTOS

Implementace tohoto efektu do RTOS spočívá opět v doplnění DSP tasku o výkonnou část a vytvoření nového ovládacího tasku. Navíc, protože nároky na operační paměť jsou kritické, je nutno připojit externí SDRAM.

DSP task je doplněn o část kódu uvedenou ve zdr. kód 9.11. Zajišťuje volání funkce *pitch_scale*, kde je potřeba zajistit správnou návaznost dat, a přijímá informace z fronty. Fronta spojuje DSP task s ovládacím taskem a putuje jí koeficient pitch scale. DSP task frontu kontroluje cyklicky, ale nečeká na její naplnění, pouze koeficient vy-

zvedne, když je k dispozici.

```
xQueueReceive(ADC_queue, &data_in, portMAX_DELAY);
xQueueReceive(pitch_queue, &pitch_coef, 0);

dsp16_vect_copy(data_out.left, &data_out.left[SAMPLES],
OVERLAP_SAMP);
dsp16_vect_zeropad(&data_out.left[OVERLAP_SAMP], SAMPLES,
SAMPLES);
pitch_scale(data_out.left, data_in.left, pitch_coef);
dsp16_vect_copy(data_in.left, &data_in.left[SAMPLES],
OVERLAP_SAMP);

xQueueSendToBack(DAC_queue, &data_out, portMAX_DELAY);
```

Zdr. kód 9.11 Doplnění DSP tasku

Ovládací task je malá smyčka kontrolující stav tlačítek. Při stisku vybraného tlačítka změní požadovaným způsobem hodnotu koeficientu a vyšle jej do fronty. Priorita tohoto tasku je opět nízká a cyklus se opakuje cca po 100ms, což pro kontrolu stisku tlačítka stačí. Alokovaná paměť postačí minimální.

Naopak paměťový prostor nutný pro DSP task je značný, alokovat je třeba 36kB. Vzhledem k tomu, že interní paměť procesoru je 64kB, tak funkce *malloc*, užívaná zde v heap3 FreeRTOS, není schopna paměť alokovat. Deska EVK1100 ovšem disponuje externí 32MB SDRAM, kterou je možné po správném nastavení využívat přímo jako operační paměť systému.

K nastavení SDRAM jako operační paměti je nutné provést tyto kroky:

- Inicializace SDRAM jako hardwaru
- Doplnění startovní (startup) funkce
- Nastavení linkeru

Aby byl procesor schopen pracovat s externí pamětí, je nutné nejprve inicializovat periferie, které zprostředkovávají komunikaci s SDRAM a vnitřní datovou sběrnicí procesoru. Komunikačním rozhraním je jednotka EBI, která je do vnějšku připojena přes jednotku PIO k IO pinům procesoru, a uvnitř je připojena na HMATRIX (HSB). HSB je vícevrstvá sběrnice, která umožňuje komunikaci mezi větším množstvím vnitřních zařízení. Inicializaci zajišťuje funkce *sdramc_init*, dostupná v knihovně *sdram.c* z framework knihoven. Jediným parametrem této funkce je hodinový kmitočet sběrnice. Kmitočet sběrnice se po této inicializaci již nesmí měnit, proto je nutné před inicializací SDRAM inicializovat časování jádra procesoru a sběrnice. Paměť je využívána jako operační paměť systému, je tedy nezbytné ji inicializovat na samém začátku, aby nebylo možné spustit nějakou funkci, která by zasahovala do paměti např. alokací. Proto se inicializace časování a SDRAM umisťuje do tzv. startup funkce.

Startovací funkci *_init_startup* je buď nutné vytvořit, nebo v případě implementovaného FreeRTOS je již vytvořena a umístěna v knihovně *port.c*. Tato funkce pak bude obsahovat mj. inicializaci časování a inicializaci SDRAM. Je nutné naincludovat ještě všechny potřebné knihovny. Funkce musí být spouštěna z assemblerovské startup funkce *crt0.x*, tu je třeba doplnit o kód zdr. kód 9.12. Globální proměnné si linker do paměti

umisťuje sám podle potřeby. Je možné jej však přimět k tomu, aby proměnné, často velká datová pole, umístil přímo do SDRAM, pro tento účel je startup funkce doplněna o kód v asembleru, jenž tyto proměnné umístí. Kód je obdobný s kódem pro interní RAM. Skládá se ze dvou bloků, pro inicializované proměnné, do nichž je uložena požadovaná hodnota, a pro neinicializované proměnné, které jsou vynulovány. Pokud interní paměť pro globální proměnné nestačí, linker sám začne přiřazovat proměnným adresy z prostoru SDRAM a tyto bloky jsou pak spouštěny automaticky.

```
//Call the startup customization function.
     call     _init_startup

//Load initialized external SDRAM data having a global
//lifetime from the data_sdram LMA.
     lda.w    r0, _data_sdram
     lda.w    r1, _edata_sdram
     cp       r0, r1
     brhs     idata_sdram_load_loop_end
     lda.w    r2, _data_sdram_lma
idata_sdram_load_loop:
     ld.d     r4, r2++
     st.d     r0++, r4
     cp       r0, r1
     brlo     idata_sdram_load_loop
idata_sdram_load_loop_end:

//Clear uninitialized external SDRAM data having a global
//lifetime in the blank static storage section.
     lda.w    r0, __bss_sdram_start
     lda.w    r1, _bss_sdram_end
     cp       r0, r1
     brhs     udata_sdram_clear_loop_end
     mov      r2, 0
     mov      r3, 0
udata_sdram_clear_loop:
     st.d     r0++, r2
     cp       r0, r1
     brlo     udata_sdram_clear_loop
udata_sdram_clear_loop_end:
```

volání uživatelské startup funkce

inicializované proměnné

neinicializované proměnné

Zdr. kód 9.12 Doplňující kód assemblerovské startup funkce *crt0.x*

O tom kolik paměti je k dispozici je nutné linker informovat a slouží k tomu soubor *link_uc3a0512.lds*. Nastavení paměti ukazuje zdr. kód 9.13, počáteční adresa je podle EBI SRAM CS1/SDRAM CS0 v [4] 0xD0000000 a velikost je 32MB. Pro dynamickou alokaci je potřeba rozšířit ještě velikost haldy. Defaultně je maximální velikost nastavena na -1, což znamená maximální využití interní RAM. Tuto hodnotu stačí přepsat hodnotou velikosti SDRAM. Nakonec se nastaví konstanty pro inicializaci proměnných v SDRAM a přesměruje se halda do SDRAM. Vše ukazuje zdr. kód 9.14, důležité části jsou zvýrazněny.

```
MEMORY
{
    FLASH (rxai!w) : ORIGIN = 0x80000000, LENGTH = 0x00080000
    INTRAM (wxa!ri) : ORIGIN = 0x00000004, LENGTH = 0x0000FFFC
    SDRAM (wxa!ri) : ORIGIN = 0xD0000000, LENGTH = 0x02000000
    USERPAGE : ORIGIN = 0x80800000, LENGTH = 0x00000200
}

PHDRS
{
    FLASH PT_LOAD;
    INTRAM_ALIGN PT_NULL;
    INTRAM_AT_FLASH PT_LOAD;
    INTRAM PT_NULL;
    SDRAM_AT_FLASH PT_LOAD;
    SDRAM PT_NULL;
    USERPAGE PT_LOAD;
}
```

Zdr. kód 9.13 Nastavení paměti linkeru, soubor *link_uc3a0512.lds*

```
__heap_size__ = DEFINED(__heap_size__) ? __heap_size__ :
LENGTH(SDRAM);
```

nastavení velikosti haldy

```
. = ORIGIN(SDRAM);
.data_sdram ORIGIN(SDRAM):AT(LOADADDR(.balign)+SIZEOF(.balign))
{
  PROVIDE(_data_sdram = .);
  *(.data_sdram)
  . = ALIGN(8);
  PROVIDE(_edata_sdram = .);
} >SDRAM :SDRAM_AT_FLASH

PROVIDE(_data_sdram_lma = ABSOLUTE(LOADADDR(.data_sdram)));

. = ALIGN(8);
.bss_sdram :
{
  PROVIDE(__bss_sdram_start = .);
  *(.bss_sdram)
  PROVIDE(_bss_sdram_end = .);
} >SDRAM AT>SDRAM :SDRAM

.heap :
{
  __heap_start__ = .;
  *(.heap)
  . = __heap_size__;
  __heap_end__ = .;
} >SDRAM AT>SDRAM :SDRAM
```

zavedení startup bloku inicializovaných proměnných

zavedení startup bloku neinicializovaných proměnných

přesměrování haldy

Zdr. kód 9.14 Nastavení haldy a doplňující části linkeru, soubor *link_uc3a0512.lds*

10 ZÁVĚR

Mikrokontrolér AT32UC3A0512 je dostatečně výkonný, aby zvládal výpočty pro DSP v reálném čase. Zřetězení instrukcí a tomu přizpůsobená instrukční sada umožňuje při správném řazení instrukcí v programu rychlé provádění operací MUL a MAC.

Vývojová deska EVK1100 není přímo přizpůsobena pro práci s audio-signály, ale většina IO portů je vyvedena na rozšiřující port. K tomuto portu je připojena rozšiřující deska se všemi potřebnými obvody. Jako vstup je na této desce osazen 16-bitový AD převodník, který s procesorem komunikuje po lince I2S. Na procesoru je implementován 16-bitový DA převodník, jenž vyžaduje už jen jednoduchý filtr a zesilovač. Časování celé soustavy zajišťuje kmitočtový syntezátor, který je široce nastavitelný po rozhraní I2C.

Rozšiřující deska splnila kladené požadavky a její oživení bylo téměř bezproblémové. Koncepce byla zvolena s důrazem na malé rozměry a univerzálnost připojení k jiným vývojovým deskám. Ovládací knihovna tohoto rozhraní je poměrně rozsáhlá, ale díky tomu umožňuje široké možnosti nastavení, a to nejen AD/DA převodníků, ale i samostatného nastavení syntezátoru. Inicializaci lze provést buď jedinou funkcí pro přednastavené, často používané módy, nebo libovolně nastavit každou jednotku zvlášť podle vlastních požadavků.

Vývojový software je volně dostupný a je vybaven framework knihovnami, které významně zjednodušují práci vývojáře. Mezi dodávanými knihovnami je i DSP-lib, která je předurčena pro využití v aplikacích zpracovávajících digitální signály. Obsahuje množství funkcí realizujících základní a některé pokročilé operace s vektory a signály. Mezi vytížené funkce patří především filtry, transformace a vektorové operace.

Vytvořeny byly dva hudební efekty. 3-pásmový ekvalizér, jenž je založen na FIR filtraci s přímou syntézou koeficientů, je jednoduchým efektem schopným pracovat na maximálním vzorkovacím kmitočtu. Zvolený filtr je nejkratší možný, ale celkově přes svojí jednoduchost je subjektivní hodnocení ekvalizace velice dobré. Druhým efektem je „pitch-scale“. Algoritmus ve frekvenční oblasti byl nejprve pečlivě odsimulován a vyladěn v matlabu na plovoucí desetinné čárce. Byly analyzovány chyby, které se při zpracování vyskytují a degradují výsledný signál. Implementace do procesoru byla další komplikovanou operací. Problematika spočívala v převodu na pevnou desetinnou čárku, ale především v optimalizaci algoritmu. Ve výsledku se podařilo i tento efekt implementovat, ovšem již ne v plném rozsahu. V reálném čase je schopen pracovat pouze v monofonním režimu a vzorkovacím kmitočtu 32kHz. To vše se zapnutými optimalizacemi kompilátoru.

Oba efekty byly nakonec implementovány pod operační systém FreeRTOS. Součástí aplikací jsou ovládací tasky, které zajišťují jednoduché nastavení efektů za běhu. Ekvalizér stále nevyžaduje žádná omezení a je schopen běžet při maximální vzorkovací frekvenci. Efekt „pitch scale“ vyžadoval rozšíření operační paměti. Toto rozšíření si zřejmě vyžádalo snížení výkonnosti procesoru, navíc samotná implementace snížila dostupný čas procesoru. Muselo tak dojít k dalším opatřením ve formě snížení vzorkovacího kmitočtu až na 20kHz.

Součástí výsledků práce jsou také přílohy, v nichž jsou uvedeny plány sestaveného rozhraní, pomocné dokumentace ke knihovnám a návod na sestavení aplikací s vytvořeným rozhraním a knihovnami. Softwarovou přílohu tvoří zmíněné knihovny ovládání rozhraní, efektu „pitch-scale“ a 3-pásmového ekvalizéru, aplikace s efekty a aplikace s efekty pod FreeRTOS. Součástí jsou ještě některé pomocné programy pro Matlab.

LITERATURA

[1] ZAJÍC, J. *Demo aplikace s procesorem Atmel AVR32 UC.* Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií. Ústav radioelektroniky, 2009.

[2] *AVR32* [online]. ATMEL Architecture document [cit. 1. března 2010]. Dostupný z www: <http://www.atmel.com/dyn/resources/prod_documents/doc32000.pdf>

[3] *AVR32UC* [online]. ATMEL Technical reference manual [cit. 1. března 2010]. Dostupný z www: <http://www.atmel.com/dyn/resources/prod_documents/doc32002.pdf>

[4] *AT32UC3A* [online]. ATMEL 32-bit microcontroller datasheet [cit. 1. března 2010]. Dostupný z www: <http://www.atmel.com/dyn/resources/prod_documents/doc32058.pdf>

[5] *AVR32 ABDAC audio bitstream DAC driver example* [online]. ATMEL Application note AVR32120 [cit. 4. dubna 2010]. Dostupný z www: <http://remotemediaconnect.googlecode.com/files/AVR32%20-%20Using%20the%20DAC.pdf>

[6] *AVR JTAGICE mkII* [online]. ATMEL doc2489 [cit. 27. dubna 2010]. Dostupný z www: <http://www.atmel.com/dyn/resources/prod_documents/doc2489.pdf>

[7] *Connecting to a target board with the AVR JTAGICE mkII* [online]. ATMEL JTAGICE mkII Quick Start Guide [cit. 27. dubna 2010]. Dostupný z www: <http://www.atmel.com/dyn/resources/prod_documents/doc2562.pdf>

[8] *AVR32 Studio getting started* [online]. ATMEL Application note AVR32015 [cit. 11. března 2010]. Dostupný z www: <http://www.atmel.com/dyn/resources/prod_documents/doc32086.pdf>

[9] *AVR ONE! Quick start guide* [online]. ATMEL doc32103 [cit. 11. března 2010]. Dostupný z www: <http://www.atmel.com/dyn/resources/prod_documents/doc32103.pdf>

[10] *EVK1100 schematics* [online]. ATMEL EVK1100 schematics rev C [cit. 21. prosince 2009]. Dostupný z www: <http://www.atmel.com/dyn/resources/prod_documents/EVK1100_SCHEMATICS_REVC.pdf>

[11] *EVK1100 bill of materials* [online]. ATMEL EVK1100 bill of materials rev C [cit. 20. dubna 2010]. Dostupný z www: <http://www.atmel.com/dyn/resources/prod_documents/EVK1100_%20BILL%20OF%20MATERIALS_RevC.xls>

[12] *CDCE913* [online]. Texas Instruments programmable clock synthesizer datasheet [cit. 26. září 2010]. Dostupný z www: <http://focus.ti.com/lit/ds/symlink/cdce913.pdf>

[13] *TPA152* [online]. Texas Instruments stereo audio power amplifier datasheet [cit. 26. září 2010]. Dostupný z www: <http://focus.ti.com/lit/ds/symlink/tpa152.pdf>

[14] *AD1870* [online]. Analog Devices 16-bit stereo ADC datasheet [cit. 26. září 2010]. Dostupný z www: <http://www.analog.com/static/imported-files/data_sheets/AD1870.pdf>

[15] *DSPLib reference manual* [online]. ATMEL Application note AVR32765 [cit. 19. prosince 2010]. Dostupný z www: <http://www.atmel.com/dyn/resources/prod_documents/doc32120.pdf>

[16] *Series software framework DSPLib* [online]. ATMEL Application note AVR32718 [cit.

25. listopadu 2010]. Dostupný z www: <http://www.atmel.com/dyn/resources/prod_documents/doc32076.pdf>

[17] ZÁPLATA, F. *Digitální filtrace na 8-bitových procesorech: bakalářská práce.* Brno: FEKT VUT v Brně, 2009

[18] KAHRS, M.; BRANDENBURG, K. *Applications of digital signal processing to audio and acoustics*. United states of America : Kluwer Academic Publishers, 2002. 545 s. ISBN 0-3064-7042-X

[19] MOULINES, E.; LAROCHE, J. Non-parametric techniques for pitch-scale and time-scale modification of speech. *Speech communication*. 1995, 16, [cit. 8. února 2011]. s. 175-205. Dostupný z www: <http://www.atc.creative.com/users/jeanl/Papers/non_param.pdf>.

[20] BERNSEE, S. M. *The DSP Dimension* [online]. 21. září 1999 [cit. 9. února 2011]. Pitch Shifting Using The Fourier Transform. Dostupné z www: <http://www.dspdimension.com/admin/pitch-shifting-using-the-ft/>.

[21] LIS, F.; PAN, G.. *EETimes : Design* [online].Analog Devices, 14. září 2008 [cit. 1. března 2011]. Speeding up the CORDIC algorithm with a DSP. Dostupné z www: <http://www.eetimes.com/design/signal-processing-dsp/4007665/Speeding-up-the-CORDIC-algorithm-with-a-DSP>.

[22] BARRY, R. *Using the FreeRTOS real time kernel : A practical guide*. 2009. 163 s.

[23] *Using the AVR32 SDRAM controller* [online]. ATMEL Application note AVR32102 [cit. 26. dubna 2011]. Dostupný z www: <http://www.atmel.com/dyn/resources/prod_documents/doc32013.pdf>

[24] *Placing data and the heap in external SDRAM* [online]. ATMEL Application note AVR32733 [cit. 27. dubna 2011]. Dostupný z www: <http://www.atmel.com/dyn/resources/prod_documents/doc32121.pdf>

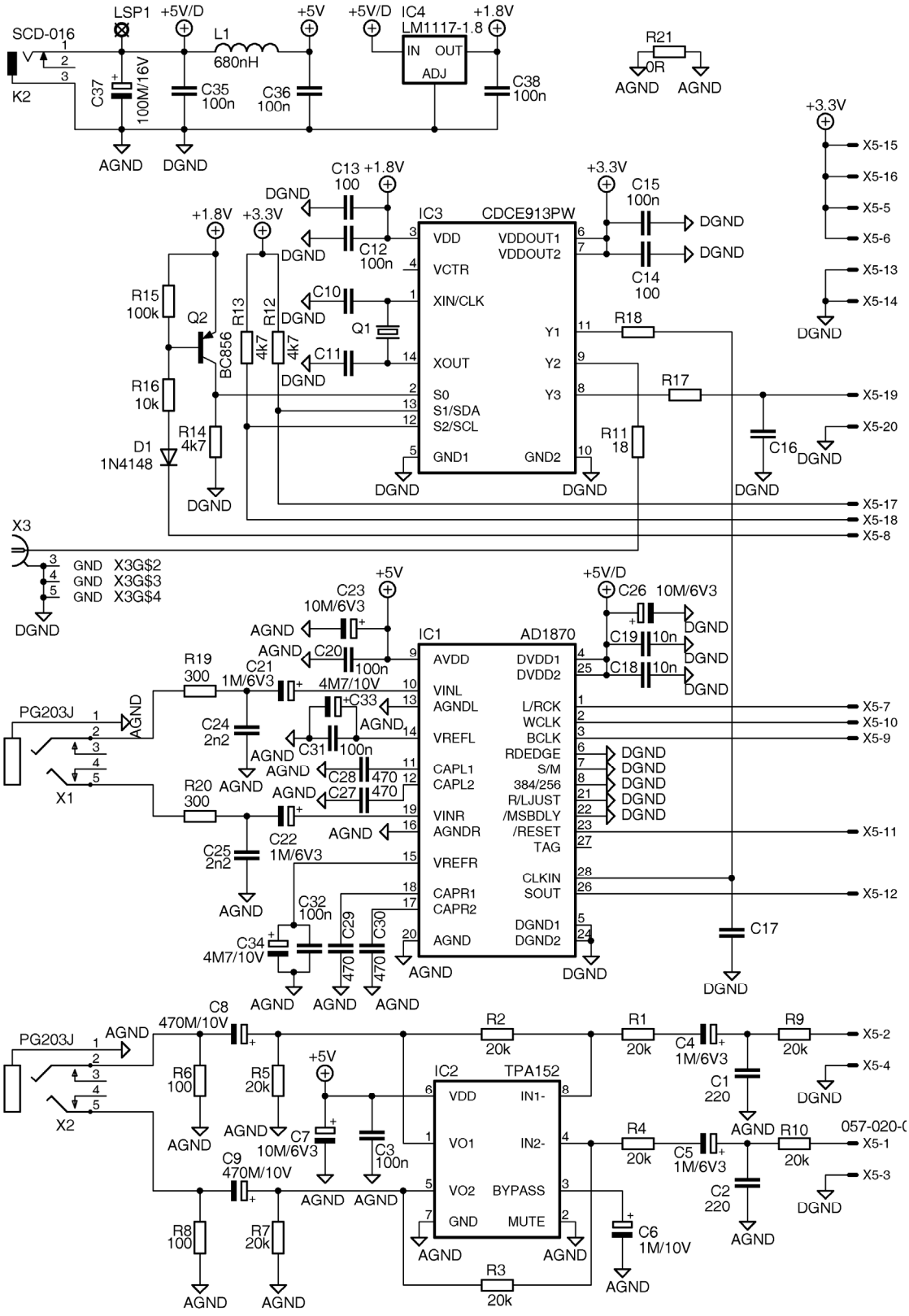
SEZNAM ZKRATEK

ABDAC	Audio Bitstream Digital to Analog Converter, číslicově-analogový převodník datového toku
ADC	Analog to Digital Converter, analogově číslicový převodník
ALU	Arithmetic-Logical Unit, aritmeticko-logická jednotka
CORDIC	COordinate Rotation DIgital Computer, rotační algoritmy
DMA	Direct Memory Access, přímý přístup do paměti
DMIPS	Dhrystone MegaInstruction Per Second, jednotka počtu instrukcí za sekundu
DSP	Digital Signal Processing, číslicové zpracování signálu
EBI	External Bus Interface, rozhraní externí sběrnice
EX	Execution, vykonání instrukce
FIFO	First In First Out, typ zásobníku
GNU	Gnu is Not Unix, „gnu“ z anglického „pakůň“, název projektu
GPIO	General Purpose Input/Output, obecně použitelné vstupní/výstupní porty
HSB	Hingh Speed Bus, vysokorychlostní sběrnice
ID	Instruction Decode, dekódování instrukce
IF	Instruction Fetch, vyzvednutí instrukce
ICH	Impulsní Charakteristika
IO	Input Output, vstupně výstupní
LMS	Least Mean Square, nejmenší kvadratická odchylka
MAC	Multiply and ACcumulate, přičtení součinu
MAC	Media Address Controll, zkratka pro adresování rozhraní Ethernet
MCU	Master Clock Unit, hlavní časová základna
MPU	Memory Protection Unit, jednotka správy paměti
MUL	MULtiply, součin
MUTEX	MUTual EXclusion, vzájemná vylučitelnost
NSD	Největší Společný Dělitel
OCD	On-Chip Debug, ladění na procesoru
OS	Operating System, operační systém
PDCA	Peripheral DMA Controller, periferie ovládání DMA kanálů

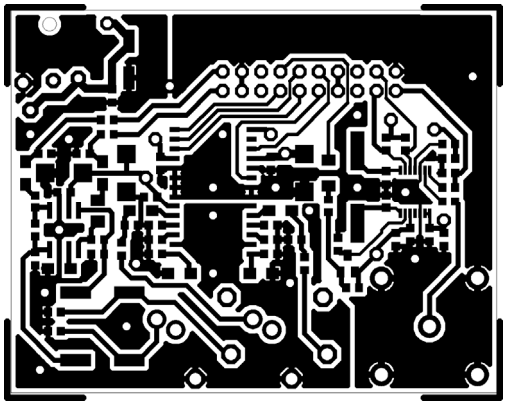
PID	Peripheral IDentifier, identifikační číslo periferie
PIO	Parallel Input Output, paralelní rozhraní
PSOLA	Pitch Synchronous OverLap Add
RISC	Reduced Instruction Set Computer
RTOS	Real Time Operating System, operační systém reálného času
SDRAMC	Synchronous Dynamic RAM controller, řadič dynamické RAM
STFT	Short Time Fourier Transform, krátkodobá fourierova transformace
SMC	Static Memory Controller, řadič statické RAM
SSC	Synchronous Serial Controller, synchronní sériové rozhraní
TWI	Two Wire Interface, komunikační rozhraní fy. Atmel kompatibilní s I2C

SEZNAM PŘÍLOH

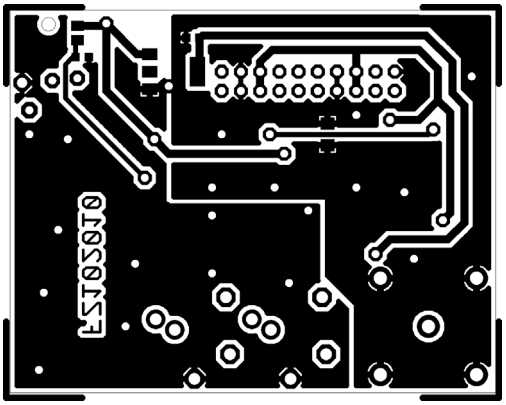
A SCHÉMA ROZŠIŘUJÍCÍ DESKY



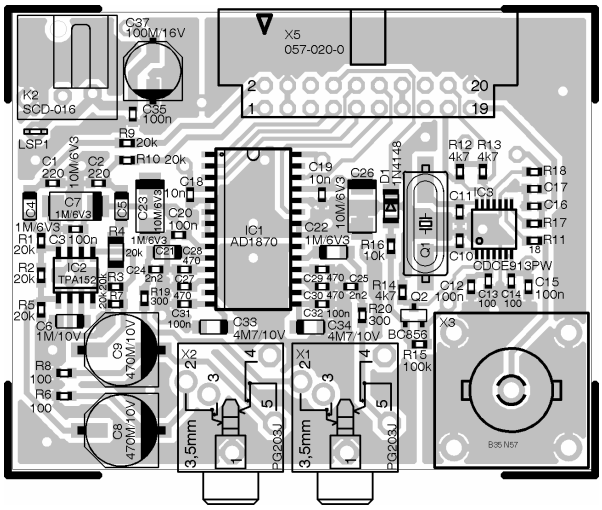
B OBRAZCE DESKY PLOŠNÝCH SPOJŮ A OSAZOVACÍ PLÁNY



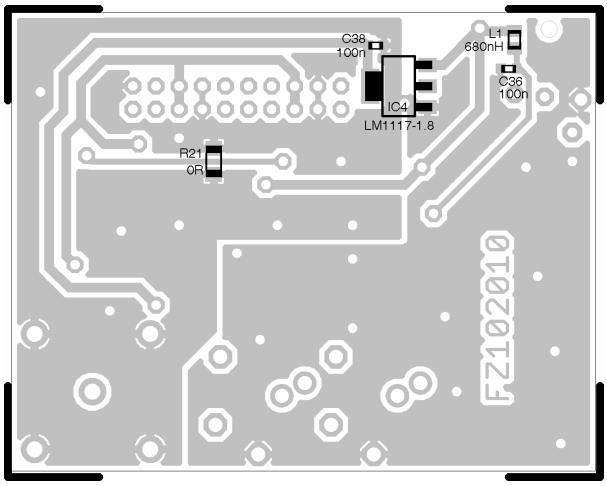
Příloha B1 Obrazec DPS strana součástek



Příloha B2 Obrazec DPS strana spojů



Příloha B3 Osazení strana součástek



Příloha B4 Osazení strany spojů

C SEZNAM SOUČÁSTEK ROZŠIŘUJÍCÍ DESKY

Označení	Hodnota	Pouzdro
Kondenzátory keramické		
C10, C11	0*	0603
C13, C14, C16, C17	100	0603
C1, C2	220	0603
C27, C28, C29, C30	470	0603
C24, C25	2n2	0603
C18, C19	10n	0603
C3, C12, C15, C20, C31, C32, C35, C36, C38	100n	0603
Kondenzátory tantalové		
C4, C5, C6, C21, C22	1M/10V	SMCA
C33, C34	4M7/10V	SMCA
C7, C23, C26	10M/6V3	SMCC
Kondenzátory elektrolytické		
C37	100M/16V	EUD
C8, C9	470M/10V	EUF
Rezistory		
R21	0R	1206
R11, R17, R18	18	0603
R6, R8	100	0603
R19, R20	300	0603
R12, R13, R14	4k7	0603
R16	10k	0603
R1, R2, R3, R5, R7, R9, R10	20k	0603
R4	20k	1206
R15	100k	0603
Polovodiče		
D1	1N4148	MINIMELF
Q2	BC856	SOT23
IC1	AD1870	SOL28
IC2	TPA152	SO-08
IC3	CDCE913PW	R-PDSO-G14
IC4	LM1117-1.8	SOT223
Ostatní		
Q1	12,288MHz	SM49
L1	680nH	0805
K2	napájecí konektor	SCD-016
LSP1	pin	LSP10
X1, X2	jack 3,5mm	PG203J
X3	BNC	B35N57
X5	konektor pro plochý kabel	057-020-0

*kondenzátory nejsou standardně osazeny, osadit pouze v případě, že je nutno krystal doplnit blokovacími kondenzátory většími než 2x 40pF.

D APLIKAČNÍ LIST OVLÁDACÍ KNIHOVNY FZ102010

D.1 Funkce

Popis:

Inicializace syntezátoru

Definice:

```
int init_synth_FZ102010
(
      void
)
```

Vstupní parametry:

žádné

Výstupní parametry:

0 inicializace proběhla bez problémů
1 při inicializaci se vyskytla chyba

Poznámky:

Popis:

Nastavení kmitočtu PLL

Definice:

```
int set_VCO_FZ102010
(
      unsigned int f
)
```

Vstupní parametry:

f požadovaný kmitočet PLL v rozmezí 80e6 – 230e6, 0 přemosťuje PLL

Výstupní parametry:

0 nastavení proběhlo bez problémů
1 při nastavování se vyskytla chyba

Poznámky:

Popis:

Nastavení děličky syntezátoru

Definice:

```
int set_Pdiv_FZ102010
(
      unsigned short P,
      unsigned char channel
)
```

Vstupní parametry:

P	hodnota děličky, rozmezí 1 – 127 pro děličky 2 a 3, rozmezí 1 – 1023 pro děličku 1
channel	identifikace děličky, definováno v hlavičkovém souboru

Výstupní parametry:

0	nastavení proběhlo bez problémů
1	při nastavování se vyskytla chyba

Poznámky:

Popis:

Nastavení multiplexeru

Definice:

```
int set_MPX_FZ102010
(
      unsigned int mpx
)
```

Vstupní parametry:

mpx	adresa, maska a hodnota multiplexeru, definováno v hlavičkovém souboru

Výstupní parametry:

0	nastavení proběhlo bez problémů
1	při nastavování se vyskytla chyba

Poznámky:

Pro lepší porozumění lze podrobný popis nalézt v aplikačním listu obvodu CDCE913 [12]

Popis:

Inicializace AD převodníku

Definice:

```
int init_ADC_FZ102010
(
      const signed short *data_block_A,
      const unsigned short block_size_A,
      const signed short *data_block_B,
      const unsigned short block_size_B
)
```

Vstupní parametry:

*data_block_A	ukazatel na paměťový blok A
block_size_A	velikost bloku A
*data_block_B	ukazatel na paměťový blok B
block_size_B	velikost bloku B

Výstupní parametry:

0	inicializace proběhla bez problémů
1	při inicializaci se vyskytla chyba

Poznámky:

Předem je nutno inicializovat přerušení!
Datové bloky jsou střídavě využívány DMA kanálem, je třeba je definovat jako libovolně velké datové pole. Pravý a levý kanál jsou umisťovány střídavě počínaje pravým.

Popis:

Inicializace DA převodníku

Definice:

```
int init_DAC_FZ102010
(
      const signed short *data_block_A,
      const unsigned short block_size_A,
      const signed short *data_block_B,
      const unsigned short block_size_B
)
```

Vstupní parametry:

*data_block_A	ukazatel na paměťový blok A
block_size_A	velikost bloku A
*data_block_B	ukazatel na paměťový blok B
block_size_B	velikost bloku B

Výstupní parametry:

0	inicializace proběhla bez problémů
1	při inicializaci se vyskytla chyba

Poznámky:

Předem je nutno inicializovat přerušení!
Datové bloky jsou střídavě využívány DMA kanálem, je třeba je definovat jako libovolně velké datové pole. Pravý a levý kanál musí být umisťovány střídavě počínaje pravým.

Popis:
Inicializace předdefinovaných módů
Definice:
void init_board_mode_FZ102010 (**const board_options_FZ102010_t** *opt)
Vstupní parametry:
*opt ukazatel na strukturu s nastavením
Výstupní parametry:
žádné
Poznámky:
Předem je nutno inicializovat přerušení! Struktura nese informace o přednastavovaném módu, jež jsou předdefinovány v hlavičkovém souboru, a informace o datových blocích pro DMA kanály. Inicializace je prováděna funkcemi init_ADC_FZ102010 a init_DAC_FZ102010.

D.2 Globální proměnné

Popis:	
Absolutní odchylka nastavení kmitočtu syntezátoru	
Definice:	**Počáteční stav:**
int SYNTH_SET_ERROR	nedefinován
Poznámky:	

Popis:	
Příznak volného bloku A AD převodníku	
Definice:	**Počáteční stav:**
char ADC_DATA_A_RDY	nedefinován
Poznámky:	
1 když je k dispozici blok A, 0 když je k dispozici blok B	

Popis:	
Příznak volného bloku B DA převodníku	
Definice:	**Počáteční stav:**
char DAC_DATA_A_RDY	nedefinován
Poznámky:	
1 když je k dispozici blok A, 0 když je k dispozici blok B	

D.3 Datové typy

Popis:

Definice:

```
board_options_FZ102010_t
{
      unsigned int mode;
      const signed short *data_block_ADC_A;
      const unsigned short block_size_ADC_A;
      const signed short *data_block_ADC_B;
      const unsigned short block_size_ADC_B;
      const signed short *data_block_DAC_A;
      const unsigned short block_size_DAC_A;
      const signed short *data_block_DAC_B;
      const unsigned short block_size_DAC_B;
}
```

Parametry:

mode	přednastavený mód, definováno v hlavičkovém souboru
*data_block_ADC_A	ukazatel na datový blok A AD převodníku
block_size_ADC_A	velikost datového bloku A AD převodníku

Poznámky:

D.4 Definované konstanty

Popis:

Definice kmitočtů jádra procesoru a sběrnice

Definice:	**Implicitní hodnota:**
FCPU	60000000
FPBA	30000000

Poznámky:

Definice jsou definovány pouze v případě, že nejsou již definovány v jiné knihovně.

Popis:

Přiřazení DMA kanálu periferiím SSC a ABDAC

Definice:	**Implicitní hodnota:**
SSC_DMA_CHANNEL	0
ABDAC_DMA_CHANNEL	1

Poznámky:

Popis:	
Přiřazení handlenu přerušení DMA kanálu	
Definice:	**Implicitní hodnota:**
IRQ_DMA_SSC	AVR32_PDCA_IRQ_0
IRQ_DMA_ABDAC	AVR32_PDCA_IRQ_1
Poznámky:	
Přísně spojeno s definicemi DMA kanálu.	

Popis:	
Definice kanálů syntezátoru	
Definice:	**Číselná hodnota:**
SYNT_ADC_CHAN	1
SYNT_DAC_CHAN	3
SYNT_OUT_CHAN	2
Poznámky:	
Hardwarově propojeno.	

Popis:	
Definice nastavení multiplexerů	
Definice:	**Číselná hodnota:**
MADC_INCLK	(0<<M1_OFFSET)+(0x02<<8)+(M1_MASK<<16)
MADC_PLLCLK	(1<<M1_OFFSET)+(0x02<<8)+(M1_MASK<<16)
MOUT_P1	(0<<M2_OFFSET)+(0x14<<8)+(M2_MASK<<16)
MOUT_POUT	(1<<M2_OFFSET)+(0x14<<8)+(M2_MASK<<16)
MDAC_PADC	(0<<M3_OFFSET)+(0x14<<8)+(M3_MASK<<16)
MDAC_POUT	(1<<M3_OFFSET)+(0x14<<8)+(M3_MASK<<16)
MDAC_PDAC	(2<<M3_OFFSET)+(0x14<<8)+(M3_MASK<<16)
Poznámky:	
Konstanta zahrnuje nastavení, adresu a masku paměti syntezátoru.	

Popis:	
Příznak dostupnosti datového bloku	
Definice:	**Číselná hodnota:**
BLOCK_A_RDY	1
BLOCK_B_RDY	0
Poznámky:	

Popis:

Předdefinované módy převodníků

Definice:	**Číselná hodnota:**
ADC_OFF_DAC_OFF	0
ADC_32_DAC_32	1
ADC_441_DAC_441	2
ADC_48_DAC_48	3
ADC_OFF_DAC_32	4
ADC_OFF_DAC_441	5
ADC_OFF_DAC_48	6
ADC_32_DAC_OFF	7
ADC_32_DAC_48	9
ADC_441_DAC_OFF	10
ADC_48_DAC_OFF	13
ADC_48_DAC_32	14
ADC_48_DAC_441	15

Poznámky:

Naznačené kmitočty v definicích odpovídají vzorkovacím kmitočtům 32kHz, 44,1kHz a 48kHz.

E SEZNAM FUNKCÍ KNIHOVNY DSP-LIB

E.1 Filtrace

FIR filtr

```
inline static void dsp16_filt_fir
  (dsp16_t*,dsp16_t*,int,dsp16_t*,int)
inline static void dsp32_filt_fir
  (dsp32_t*,dsp32_t*,int,dsp32_t*,int)
```

IIR filtr real time

```
void dsp16_filt_iir
  (dsp16_t*,dsp16_t*,int,dsp16_t*,int,dsp16_t*,int,int,int)
void dsp32_filt_iir
  (dsp32_t*,dsp32_t*,int,dsp32_t*,int,dsp32_t*,int,int,int)
```

IIR filtr

```
void dsp16_filt_iirpart
  (dsp16_t*,dsp16_t*,int,dsp16_t*,int,dsp16_t*,int,int,int)
void dsp32_filt_iirpart
  (dsp32_t*,dsp32_t*,int,dsp32_t*,int,dsp32_t*,int,int,int)
```

Interpolace vektoru

```
void dsp16_filt_interpolation
  (dsp16_t*,dsp16_t*,int,dsp16_t*,int,int)
```

Přerovnání koeficientů FIR filtru pro interpolaci

```
void dsp16_filt_interpolation_coefsort(dsp16_t*,int,int);
```

Filtrace s nejmenší kvadratickou odchylkou (LMS filtr)

```
void dsp16_filt_lms
  (dsp16_t*,dsp16_t*,int,dsp16_t,dsp16_t,dsp16_t*,dsp16_t*)
void dsp32_filt_lms
  (dsp32_t*,dsp32_t*,int,dsp32_t,dsp32_t,dsp32_t*,dsp32_t*)
void dsp32_filt_lms_fir
  (dsp32_t*,dsp32_t*,int,dsp32_t*,int)
```

Návrh DP FIR filtru

```
void dsp16_filt_lpfirdesign
  (dsp16_t*,int,int,int,dsp16_win_fct_t,dsp_filt_design_options_t)
```

Normalizovaný LMS filtr

```
void dsp16_filt_nlms
  (dsp16_t*,dsp16_t*,int,dsp16_t,dsp16_t,dsp16_t*,dsp16_t*)
void dsp32_filt_nlms
  (dsp32_t*,dsp32_t*,int,dsp32_t,dsp32_t,dsp32_t*,dsp32_t*)
void dsp32_filt_nlms_fir
  (dsp32_t*,dsp32_t*,int,dsp32_t*,int)
```

E.2 Generování signálu

Kosinusoida

```
dsp16_t dsp16_gen_cos(dsp16_t*,int,int,int,dsp16_t)
dsp32_t dsp32_gen_cos(dsp32_t*,int,int,int,dsp32_t)
```

Pole diracových impulsů

```
void dsp16_gen_dcomb(dsp16_t*,int,int,int,dsp16_t)
void dsp32_gen_dcomb(dsp32_t*,int,int,int,dsp32_t)
```

Diracův impuls

```
void dsp16_gen_dirac(dsp16_t*,int,int)
void dsp32_gen_dirac(dsp32_t*,int,int)
```

Šum

```
void dsp16_gen_noise(dsp16_t*,int,dsp16_t)
void dsp32_gen_noise(dsp32_t*,int,dsp32_t)
```

Pila

```
void dsp16_gen_ramp(dsp16_t*,int,dsp16_t)
void dsp32_gen_ramp(dsp32_t*,int,dsp32_t)
```

Obdélník

```
void dsp16_gen_rect(dsp16_t*,int,int,int,dsp16_t,dsp16_t)
void dsp32_gen_rect(dsp32_t*,int,int,int,dsp32_t,dsp32_t)
```

Trojúhelník

```
dsp16_t dsp16_gen_saw(dsp16_t*,int,int,int,dsp16_t,dsp16_t)
dsp32_t dsp32_gen_saw(dsp32_t*,int,int,int,dsp32_t,dsp32_t)
```

Sinusoida

```
dsp16_t dsp16_gen_sin(dsp16_t*,int,int,int,dsp16_t)
dsp32_t dsp32_gen_sin(dsp32_t*,int,int,int,dsp32_t)
```

Obdélník se střídou 50%

```
inline static void dsp16_gen_sqr(dsp16_t*,int,int,int,dsp16_t)
inline static void dsp32_gen_sqr(dsp32_t*,int,int,int,dsp32_t)
```

Jednotkový skok

```
void dsp16_gen_step(dsp16_t*,int,dsp16_t,dsp16_t,int)
void dsp32_gen_step(dsp32_t*,int,dsp32_t,dsp32_t,int)
```

E.3 Operátory

Inicializace generátoru náhodného čísla

```
void dsp_op_srand(int)
```

Absolutní hodnota

```
inline static dsp16_t dsp16_op_abs(dsp16_t)
inline static dsp32_t dsp32_op_abs(dsp32_t)
```

Arkus kosinus

```
inline static dsp16_t dsp16_op_acos(dsp16_t)
inline static dsp32_t dsp32_op_acos(dsp32_t)
```

Arkus sinus

```
dsp16_t dsp16_op_asin(dsp16_t)
dsp32_t dsp32_op_asin(dsp32_t)
```

Kosinus

```
inline static dsp16_t dsp16_op_cos(dsp16_t)
inline static dsp32_t dsp32_op_cos(dsp32_t)
```

Podíl

```
inline static dsp16_t dsp16_op_div(dsp16_t,dsp16_t)
inline static dsp32_t dsp32_op_div(dsp32_t,dsp32_t)
```

Exponenciála

```
dsp16_t dsp16_op_exp(dsp16_t)
dsp32_t dsp32_op_exp(dsp32_t)
```

Přirozený logaritmus

```
dsp16_t dsp16_op_ln(dsp16_t)
dsp32_t dsp32_op_ln(dsp32_t)
```

Desítkový logaritmus

```
dsp16_t dsp16_op_log10(dsp16_t)
dsp32_t dsp32_op_log10(dsp32_t)
```

Dvojkový logaritmus

```
dsp16_t dsp16_op_log2(dsp16_t)
dsp32_t dsp32_op_log2(dsp32_t)
```

Součin

```
inline static dsp16_t dsp16_op_mul(dsp16_t,dsp16_t)
inline static dsp32_t dsp32_op_mul(dsp32_t,dsp32_t)
```

Mocnina

```
dsp16_t dsp16_op_pow(dsp16_t,dsp16_t)
dsp32_t dsp32_op_pow(dsp32_t,dsp32_t)
```

Generátor náhodného čísla

```
dsp16_t dsp16_op_rand()
dsp32_t dsp32_op_rand()
```

Sinus

```
dsp16_t dsp16_op_sin(dsp16_t)
dsp32_t dsp32_op_sin(dsp32_t)
```

Odmocnina

```
dsp16_t dsp16_op_sqrt(dsp16_t)
dsp32_t dsp32_op_sqrt(dsp32_t)
```

E.4 Transformace

Komplexní FFT

```
void dsp16_trans_complexfft(dsp16_complex_t*,dsp16_complex_t*,int)
```

Komplexní inverzní FFT

```
void dsp16_trans_complexifft
  (dsp16_complex_t*,dsp16_complex_t*,int)
```

Komplexní FFT s reálným vstupem

```
void dsp16_trans_realcomplexfft(dsp16_complex_t*,dsp16_t*,int)
void dsp32_trans_realcomplexfft(dsp32_complex_t*,dsp32_t*,int)
```

E.5 Práce s vektory

Součet vektorů

```
void dsp16_vect_add(dsp16_t*,dsp16_t*,dsp16_t*,int)
void dsp32_vect_add(dsp32_t*,dsp32_t*,dsp32_t*,int)
```

Součet vektorů se saturací

```
void dsp16_vect_add_and_sat(dsp16_t*,dsp16_t*,dsp16_t*,int)
void dsp32_vect_add_and_sat(dsp32_t*,dsp32_t*,dsp32_t*,int)
```

Absolutní hodnota komplexního vektoru

```
void dsp16_vect_complex_abs(dsp16_t*,dsp16_complex_t*,int)
void dsp32_vect_complex_abs(dsp32_t*,dsp32_complex_t*,int)
```

Komplexní součet vektorů

```
void dsp16_vect_complex_add
  (dsp16_complex_t*,dsp16_complex_t*,dsp16_complex_t*,int)
void dsp32_vect_complex_add
  (dsp32_complex_t*,dsp32_complex_t*,dsp32_complex_t*,int)
```

Konjugace vektoru

```
void dsp16_vect_complex_conj
  (dsp16_complex_t*,dsp16_complex_t*,int)
```

Komplexní rozdíl vektorů

```
void dsp16_vect_complex_sub
  (dsp16_complex_t*,dsp16_complex_t*,dsp16_complex_t*,int)
void dsp32_vect_complex_sub
  (dsp32_complex_t*,dsp32_complex_t*,dsp32_complex_t*,int)
```

Konvoluce vektorů

```
void dsp16_vect_conv(dsp16_t*,dsp16_t*,int,dsp16_t*,int)
void dsp32_vect_conv(dsp32_t*,dsp32_t*,int,dsp32_t*,int)
```

Částečná konvoluce

```
void dsp16_vect_convpart(dsp16_t*,dsp16_t*,int,dsp16_t*,int)
void dsp32_vect_convpart(dsp32_t*,dsp32_t*,int,dsp32_t*,int)
```

Kopírování vektorů

```
inline static void dsp16_vect_copy(dsp16_t*,dsp16_t*,int)
inline static void dsp32_vect_copy(dsp32_t*,dsp32_t*,int)
```

Podíl vektorů bod po bodu

```
void dsp16_vect_dotdiv(dsp16_t*,dsp16_t*,dsp16_t*,int)
void dsp32_vect_dotdiv(dsp32_t*,dsp32_t*,dsp32_t*,int)
```

Součin vektorů bod po bodu

```
void dsp16_vect_dotmul(dsp16_t*,dsp16_t*,dsp16_t*,int)
void dsp32_vect_dotmul(dsp32_t*,dsp32_t*,dsp32_t*,int)
```

Podíl vektoru a celého čísla

```
void dsp16_vect_intdiv(dsp16_t*,dsp16_t*,int,int)
void dsp32_vect_intdiv(dsp32_t*,dsp32_t*,int,int)
```

Součin vektoru s celým číslem

```
void dsp16_vect_intmul(dsp16_t*,dsp16_t*,int,int)
void dsp32_vect_intmul(dsp32_t*,dsp32_t*,int,int)
```

Maximum vektoru

```
dsp16_t dsp16_vect_max(dsp16_t*,int)
dsp32_t dsp32_vect_max(dsp32_t*,int)
```

Minimum vektoru

```
dsp16_t dsp16_vect_min(dsp16_t*,int)
dsp32_t dsp32_vect_min(dsp32_t*,int)
```

Negace vektoru

```
void dsp16_vect_neg(dsp16_t*,dsp16_t*,int)
void dsp32_vect_neg(dsp32_t*,dsp32_t*,int)
```

Kvadrát vektoru bod po bodu

```
void dsp16_vect_pow(dsp16_t*,dsp16_t*,int,dsp16_t)
void dsp32_vect_pow(dsp32_t*,dsp32_t*,int,dsp32_t)
```

Součet vektoru a zlomku

```
void dsp16_vect_realadd(dsp16_t*,dsp16_t*,int,dsp16_t)
void dsp32_vect_realadd(dsp32_t*,dsp32_t*,int,dsp32_t)
```

Podíl vektoru a zlomku

```
void dsp16_vect_realdiv(dsp16_t*,dsp16_t*,int,dsp16_t)
void dsp32_vect_realdiv(dsp32_t*,dsp32_t*,int,dsp32_t)
```

Součin vektoru se zlomkem

```
void dsp16_vect_realmul(dsp16_t*,dsp16_t*,int,dsp16_t)
void dsp32_vect_realmul(dsp32_t*,dsp32_t*,int,dsp32_t)
```

Rozdíl vektoru a zlomku

```
void dsp16_vect_realsub(dsp16_t*,dsp16_t*,int,dsp16_t)
void dsp32_vect_realsub(dsp32_t*,dsp32_t*,int,dsp32_t)
```

Rozdíl vektorů

```
void dsp16_vect_sub(dsp16_t*,dsp16_t*,dsp16_t*,int)
void dsp32_vect_sub(dsp32_t*,dsp32_t*,dsp32_t*,int)
```

Nulování vektoru

```
inline static void dsp16_vect_zeropad(dsp16_t*,int,int)
inline static void dsp32_vect_zeropad(dsp32_t*,int,int)
```

E.6 Váhování okny

Bartlettovo (trojúhelníkové)

```
void dsp16_win_bart(dsp16_t*,int)
void dsp32_win_bart(dsp32_t*,int)
```

Blackmanovo

```
void dsp16_win_black(dsp16_t*,int)
void dsp32_win_black(dsp32_t*,int)
```

Gaussovo

```
void dsp16_win_gauss(dsp16_t*,int)
void dsp32_win_gauss(dsp32_t*,int)
```

Hammingovo

```
void dsp16_win_hamm(dsp16_t*,int)
void dsp32_win_hamm(dsp32_t*,int)
```

Hannovo

```
void dsp16_win_hann(dsp16_t*,int)
void dsp32_win_hann(dsp32_t*,int)
```

Kaiserovo

```
void dsp16_win_kaiser(dsp16_t*,int,int)
void dsp32_win_kaiser(dsp32_t*,int,int)
```

Obdélníkové

```
void dsp16_win_rect(dsp16_t*,int)
void dsp32_win_rect(dsp32_t*,int)
```

Welchovo

```
void dsp16_win_welch(dsp16_t*,int)
void dsp32_win_welch(dsp32_t*,int)
```

E.7 Pokročilé operace

IMA/DVI ADPCM dekodér

```
void dsp_adpcm_ima_decode(S16*,void*,int,S16*,S16*)
```

IMA/DVI ADPCM dekodér jednoho vzorku

```
S16 dsp_adpcm_ima_decode_nibble(S8,S16*,S16*)
```

IMA/DVI ADPCM kodér

```
void dsp_adpcm_ima_encode(void*,S16*,int,S16*,S16*)
```

IMA/DVI ADPCM kodér jednoho vzorku

```
S8 dsp_adpcm_ima_encode_nible(S16,S16*,S16*)
```

Převzorkování

```
void dsp16_resampling_compute
  (dsp_resampling_t*,dsp16_t*,dsp16_t*,int)
```

Uvolnění paměti po struktuře dsp_resampling_t

```
void dsp16_resampling_free(dsp_resampling_t*,free_fct_t)
```

Aktuální počet vzorků výstupního signálu

```
int dsp16_resampling_get_output_current_buffer_size
  (dsp_resampling_t*);
```

Maximální počet vzorků výstupního signálu

```
int dsp16_resampling_get_output_max_buffer_size(dsp_resampling_t*)
```

Inicializace převzorkování

```
dsp_resampling_t* dsp16_resampling_setup
  (int,int,int,int,int,malloc_fct_t,dsp_resampling_options_t)
```

F POSTUP VÝVOJE PROJEKTŮ S AUDIO-ROZHRANÍM

F.1 Ovládání audio-rozhraní

K vytvoření aplikace využívající audio-rozhraní v AVR32 studiu je nutné provést tyto kroky:

1. založení projektu
2. vložení knihoven framework
3. vložení uživatelských knihoven
4. vložení všech knihoven do zdrojového kódu
5. definice globálních bufferů pro DMA kanály
6. inicializace časování procesoru
7. inicializace desky
8. obsluha DMA kanálů

ad. 1: Nejjednodušší způsob založení projektu pro podporovanou vývojovou desku je vybrání projektu z šablony. *File→New→AVR32 C Project From Template*, zadá se název příp. umístění, typ desky a cílový procesor MCU. Vygenerovaný kód může být dle libosti nahrazen. Hlavní výhoda šablony spočívá ve správném nastavení debuggeru a kompilátoru.

Tip: V některých případech nelze znovuotevřený projekt zkompilovat, kompilátor hlásí nesmyslné chyby. Založení nového jiného projektu z šablony a jeho smazání tento problém odstraní.

ad 2: Komunikace s rozšiřující deskou vyžaduje vnitřní knihovny z frameworku. *Framework→Select Drivers/Components/Services* a vybrat následující drivery:

- Audio Bitstream Digital-to-Analog Converter
- Flash Controller
- General Purpose I/O Controller
- Interrupt Controller
- Peripheral DMA Controller
- Power Manager
- Synchronous Seriál Controller
- Two-wire Interface

ad. 3: Ovládací knihovnu rozšiřující desky je třeba vložit z nabídky projektu. *Import: General→File Systém*. Vybrat cestu k hlavičkovému a zdrojovému souboru knihovny *FZ102010* a vložit ji do složky *src*.

ad. 4: Všechny knihovny je nutné ještě vložit do zdrojového kódu hlavního souboru, jak ukazuje zdr. kód F.1.

```
#include "board.h"
#include "abdac.h"
#include "gpio.h"
#include "intc.h"
#include "pdca.h"
#include "pm.h"
#include "ssc_i2s.h"
#include "twi.h"
```

Zdr. kód F.1 Vkládané knihovny do zdrojového kódu

ad. 5: DMA kanály potřebují externě deklarovat paměťový prostor pro ukládání vzorků. Jsou to 4 dvourozměrná 16-bitová pole stejné délky. Pokud budou využívány funkce z knihovny DSP-lib, musí být 32-bitově zarovnána. Deklarace může vypadat takto zdr. kód F.2.

```
#define SAMPLES     1024

__attribute__ ((aligned(4))) volatile short OUT_A[SAMPLES][2];
__attribute__ ((aligned(4))) volatile short OUT_B[SAMPLES][2];
__attribute__ ((aligned(4))) volatile short IN_A[SAMPLES][2];
__attribute__ ((aligned(4))) volatile short IN_B[SAMPLES][2];
```

Zdr. kód F.2 Deklarace globálních proměnných

ad. 6: DSP operace si vyžádají maximální výkon procesoru. Časování je nastaveno funkcí z power manageru podle zdr. kód F.3. Tato inicializace by měla být provedena na úplném začátku funkce *main*.

```
static pm_freq_param_t hodiny =
{
    .cpu_f         = 66000000,
    .pba_f         = 66000000,
    .osc0_f        = FOSC0,
    .osc0_startup  = OSC0_STARTUP
};

pm configure clocks(&hodiny);
```

Zdr. kód F.3 Inicializace časování procesoru

ad. 7: Inicializaci rozhraní je možné provést dvěma způsoby. Jednodušším je, nastavení do přednastaveného módu podle zdr. kód F.4. Součástí je nutná inicializace přerušení a zapnutí syntezátoru. Druhý způsob je komplikovanější, ale umožňuje přesné nastavení požadovaných parametrů. Postup ukazuje zdr. kód F.5. Popis použitých funkcí je uveden v příloze D. K nastavení syntezátoru je nutné prostudovat blokové zapojení obvodu CDCE913, uvážit kmitočet krystalu 12,288MHz a převzorkování ADC a DAC 256-krát. Po inicializaci začnou převodníky pracovat a DMA kanály periodicky plní buffery daty.

```
static const board_options_FZ102010_t brd_options =
{
  .mode = ADC_32_DAC_32,
  .data_block_ADC_A = (void *)IN_A,
  .block_size_ADC_A = SAMPLES,
  .data_block_ADC_B = (void *)IN_B,
  .block_size_ADC_B = SAMPLES,
  .data_block_DAC_A = (void *)OUT_A,
  .block_size_DAC_A = SAMPLES,
  .data_block_DAC_B = (void *)OUT_B,
  .block_size_DAC_B = SAMPLES
};

SYNT_ON;
Disable_global_interrupt();
INTC_init_interrupts();
Enable_global_interrupt();
init_board_mode_FZ102010(&brd_options);
```

Zdr. kód F.4 Inicializace desky přednastaveným módem

```
Disable_global_interrupt();
INTC_init_interrupts();
Enable_global_interrupt();
init_synth_FZ102010();
ADC_ON;
SYNT_ON;
set_MPX_FZ102010(MDAC_PDAC);
set_MPX_FZ102010(MADC_PLLCLK);
set_Pdiv_FZ102010(20, SYNT_ADC_CHAN);
set_Pdiv_FZ102010(20, SYNT_DAC_CHAN);
set_VCO_FZ102010(100000000);
init_ADC_FZ102010((void*)IN_A, SAMPLES, (void*)IN_B, SAMPLES);
init_DAC_FZ102010((void*)OUT_A, SAMPLES, (void*)OUT_B, SAMPLES);
```

Zdr. kód F.5 Uživatelská inicializace desky

ad. 8: Datové buffery jsou naplňovány střídavě (ping-pong), pracovat je možné pouze s právě volným bufferem. O tom informuje globální proměnná *DAC_DATA_A_RDY* knihovny *FZ102010*. Nabývá hodnot 1 a 0 a určuje, zda je datový blok A volný (1), pokud ne (0) je umožněna práce s datovým blokem B.

F.2 Aplikace efektů

Efekty mohou být implementovány do aplikací vytvořených postupem z předchozí části. Dále je nutné je doplnit o některé další části.

- 3-pásmový ekvalizér je vytvořen těmito kroky:
    1. doplnění knihovny DSP-lib
    2. doplnění knihovny *music_3equalizer*
    3. inicializace ekvalizéru

4. vytvoření filtrační části
5. vytvoření ovládací části

ad. 1: Knihovna DSP-lib je vložena z framework knihoven v části services v poli library výběrem stupně optimalizace DSPLib.

ad. 2: Knihovna je importována do projektu stejným způsobem jako ovládací knihovna rozhraní. Stejně musí být doplněna i direktiva *include* do zdrojového kódu.

ad. 3: Před použitím ekvalizéru je nutné vypočítat impulsní charakteristiku filtru, to zajistí jedno zavolání funkce *set_coef* s požadovanými výchozími parametry.

ad. 4: Funkce *equalizer_3* zpracovává kanály samostatně, proto je nejprve nutné oba kanály z bufferů oddělit do pomocných polí. Tato pole jsou přeposílána přímo FIR filtru knihovny DSP-lib, práce s nimi je tedy totožná. Bližší popis ukazuje zdr. kód F.6.

```
A_ALIGNED signed short buffer_IN_L[SAMPLES+FIR_COEF_SIZE-1];
A_ALIGNED signed short buffer_OUT_L[SAMPLES];
…
buffer_IN_L[FIR_COEF_SIZE+i-1]=IN_A[i][1];
…
equalizer_3(buffer_OUT_L, buffer_IN_L, SAMPLES);
dsp16_vect_copy(buffer_IN_L,
&buffer_IN_L[SAMPLES],FIR_COEF_SIZE-1);
…
OUT_A[i][1]=buffer_OUT_L[i];
…
```

Zdr. kód F.6 Implementace funkce *equalizer_3*

ad. 5: Tato část je čistě na uživateli, zadávání parametrů je realizováno funkcí *set_coef*.

- Efekt pitch scale je implementován těmito kroky:
    1. doplnění knihovny DSP-lib
    2. doplnění knihovny *music_pitch*
    3. vytvoření výkonné části
    4. vytvoření ovládací části
    5. zapnout optimalizace

```
A_ALIGNED signed short buffer_R1[DAT_LEN];
…
buffer_R1[i+OVERLAP_SAMP]=IN_A[i][0];
…
dsp16_vect_copy(buffer_R2, &buffer_R2[SAMPLES], OVERLAP_SAMP);
dsp16_vect_zeropad(&buffer_R2[OVERLAP_SAMP], SAMPLES, SAMPLES);
pitch_scale(buffer_R2, buffer_R1, pitch_coef);
dsp16_vect_copy(buffer_R1, &buffer_R1[SAMPLES], OVERLAP_SAMP);
…
OUT_A[i][1]=buffer_R2[i];
…
```

Zdr. kód F.7 Implementace funkce *pitch_scale*

ad. 3: Funkce pitch_scale pracuje podobným způsobem jako filtry, jen prodloužení polí je odlišné. Vše ukazuje zdr. kód F.7. **Použité konstanty jsou v tomto případě definovány v hlavičkovém souboru knihovny *music_pitch*!**

ad. 5: Algoritmus efektu už je příliš náročný, je proto nutné kód optimalizovat. Při vybraném projektu *Project→Properties*, v menu *C/C++ Build→Settings* v záložce *ToolSettings* v menu *AVR32/GNU C Compiler→Optimization* vybrat požadovaný *Optimization Level*.

Ostatní kroky jsou totožné s implementací ekvalizéru.

F.3 Zprovoznění operačního systému

Operační systém FreeRTOS je také součástí framework knihoven. Aplikace s tímto systémem jsou vytvářeny tímto postupem:

1. vložení knihoven FreeRTOS z frameworku
2. vložení potřebných knihoven do zdrojového kódu
3. úprava časování procesoru
4. deklarace front a semaforů
5. alokace tasků

ad. 1: Knihovny FreeRTOS se nacházejí v bloku services ve frameworku. Do projektu jsou vloženy všechny součásti FreeRTOS.

ad. 2: Základem jsou 3 bloky, jimž je nadřazena jedna knihovna. Bloky *task*, *queue* a *semphr* nemusí být vždy vkládány, záleží na využití. Direktivy ukazuje zdr. kód F.8.

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
```

Zdr. kód F.8 Vkládané knihovny FreeRTOS

ad. 3: Časování procesoru probíhá stejným způsobem, jako doposud. Dále je ale nutné nastavit OS také na tyto kmitočty, tj. v souboru *src→CONFIG→FreeRTOSConfig h* nastavit vlastní kmitočty CPU a PBA.

ad. 4: Mimo již popsané deklarace musí být vytvořeny fronty a semafory pro komunikaci mezi tasky. Dvě fronty *ADC_queue* a *DAC_queue* pro přenos ukazatelů a hlavně jako přenos informace o dostupnosti zpracovaných dat převodníků. Dva semafory *ADC_semaphore* a *DAC_semaphore* pro zajištění bezprostřední reakce na přepnutí paměťového bloku DMA kanálu.

ad. 5: Následující tasky zajistí běh rozhraní:

- Inicializace rozhraní, *task_board_init.*
  Po inicializaci desky musí být OS již aktivní, proto inicializace desky probíhá jako samostatný task, který se provede jen jednou a poté se sám dealokuje. Jeho priorita by měla být co nejvyšší. Velikost alokované

paměti postačí minimální.

- Zpracování dat z AD převodníku, *task_ADC_proc.*
  Task obsluhuje semafor, odděluje kanály a zasílá data do fronty pro DSP zpracování. Priorita tasku je vysoká, nároky na paměť minimální. Obsah tasku ukazuje zdr. kód F.9.
- Zpracování dat pro DA převodník, *task_DAC_proc.*
  Task skládá kanály dat z fronty do jednoho datového pole. Synchronizuje se se semaforem DA převodníku a posílá mu data. Nároky na paměť jsou opět minimální a priorita vysoká. Zdrojový kód je obdobný ADC tasku ve zdr. kód F.9.
- DSP zpracování, *task_DSP_proc.*
  Pracuje s příchozí a odchozí frontou s daty rozdělenými na kanály. Provádí DSP operace. Nároky na paměť jsou dány funkcemi, které jsou používány, může dosahovat až několika MB (po nastavení SDRAM). Priorita je stejná jako u ADC a DAC tasků, pouze v některých kritických částech může být vyšší. Zdr. kód F.9 také ukazuje DSP task.

```
void task_ADC_proc(void *param){
  signal_buffer_t *data;
  int i;

  data=(signal_buffer_t *)param;
  for (;;)
  {
        xSemaphoreTake(ADC_semaphore, portMAX_DELAY);
        if (ADC_DATA_A_RDY)
              for (i=0; i<SAMPLES; i++)
              {
                    data[i].left=IN_A[i][1];
                    data[i].right=IN_A[i][0];
              }
        else
              for (i=0; i<SAMPLES; i++)
              {
                    data[i].left=IN_B[i][1];
                    data[i].right=IN_B[i][0];
              }
        xQueueSendToBack(ADC_queue, &data, portMAX_DELAY);
  }
}

void task_DSP_proc(void *param){
  signal_buffer_t *data;

  for (;;)
  {
        xQueueReceive(ADC_queue, &data, portMAX_DELAY);
        …
        xQueueSendToBack(DAC_queue, &data, portMAX_DELAY);
  }
}
```

Zdr. kód F.9 ADC a DSP tasky

Tip: Při kompilaci se mohou vyskytnout potíže s errory způsobené duplikátními soubory. Jeden soubor je pak nutno vyřadit z kompilace. např. *src→ SOFTWARE_FRAMEWORK→DRIVERS→INTC→exception.x* a z nabídky *Exclude from build…*

F.4 Aplikace efektů pod FreeRTOS

Implementace efektů pod operační systém je kombinací postupů předchozích částí.

- Ekvalizér vyžaduje následující postup:
    1. doplnění knihovny DSP-lib
    2. doplnění knihoven FreeRTOS
    3. doplnění knihovny *music_3equalizer*
    4. doplnění knihoven do zdrojového kódu
    5. úprava časování FreeRTOS
    6. inicializace ekvalizéru
    7. vytvoření front, semaforů a tasků

Opět jsou vynechány kroky vytvoření aplikace pro audio-rozhraní, které jsou pro všechny aplikace stejné.

- Pitch scale efekt je implementován podobným způsobem:
    1. doplnění knihovny DSP-lib
    2. doplnění knihoven FreeRTOS
    3. doplnění knihovny *music_pitch*
    4. doplnění knihoven do zdrojového kódu
    5. úprava časování FreeRTOS
    6. nastavení externí operační paměti
    7. vytvoření front, semaforů a tasků
    8. zapnout optimalizace

ad. 6: Externí paměť doposud nebyla vyžadována, paměť je dostupná na desce EVK1100 a vyžadována je jen inicializace podle následujících bodů:

- doplnění knihoven SDRAM z frameworku
  Protože je využíván další HW, je nutné ještě přidat ovládací knihovnu SDRAM, jak do projektu, tak i do zdrojového kódu.
- nastavení assemblerovské startup funkce
  Soubor *src→SOFTWARE_FRAMEWORK→UTILS→STARTUP_FILES →GCC→crt0.x* je třeba přepsat, nebo nahradit připraveným souborem stejného názvu.
- nastavení startup funkce FreeRTOS

V souboru *src→SOFTWARE_FRAMEWORK→SERVICES →FREERTOS→Source→portable→GCC→AVR32_UC→port.c* se nachází funkce *_init_startup*, tu je potřeba doplnit inicializačním kódem pro SDRAM. Tj. na konec této funkce připsat volání funkce *sdramc_init* s požadovaným vstupním kmitočtem (kmitočet sběrnice procesoru). Funkce by měla obsahovat i inicializační sekvenci časování jádra a sběrnice.

- nastavení linkeru
  Linkovací soubor je třeba změnit, nebo jednodušeji jej nahradit připraveným souborem *link_uc3a0512_SDRAM.lds*. Soubor se nachází v *src→SOFTWARE_FRAMEWORK→UTILS→LINKER_SCRIPTS→AT32UC3A→0512→GCC*. Pokud se změní název linkovacího souboru, je nutné ještě změnit nastavení linkeru, přenastavit cestu k tomuto souboru v *Project→Properties* z nabídky *C/C++ Build→Settings* v záložce *Tool Settings* v nabídce *AVR32/GNU C Linker→Miscellaneous* v poli *Linker Flags* přepsat příkaz -*T../src/SOFTWARE_FRAMEWORK/UTILS/LINKER_SCRIPTS/AT32UC3A/0512/GCC/link_uc3a0512.lds* správnou adresou nebo nový příkaz přidat. Ve výjimečných případech je potřeba ještě překontrolovat ostatní nastavení linkeru.

G APLIKAČNÍ LIST EFEKTOVÉ KNIHOVNY MUSIC_3EQUALIZER

G.1 Funkce

Popis:

Nastavení charakteristiky ekvalizéru

Definice:

```
void set_coef
(
     signed char* band_setting
)
```

Vstupní parametry:

band_setting ukazatel na pole nastavení ekvalizéru, 3x od 0 do 8

Výstupní parametry:

žádné

Poznámky:

Popis:

Filtrační funkce ekvalizéru

Definice:

```
void equalizer_3
(
     dsp16_t *d_out,
     dsp16_t *d_in,
     int out_size
)
```

Vstupní parametry:

```
d_out     ukazatel na výstupní datový buffer
d_in      ukazatel na vstupní datový buffer,
          velikost je out_size + FIR_COEF_SIZE - 1
out_size  velikost výstupního datového bufferu
```

Výstupní parametry:

žádné

Poznámky:

G.2 Definované konstanty

Popis:	
`Definice velikosti impulsní odezvy filtru ekvalizéru`	
Definice:	**Implicitní hodnota:**
`FFT_SIZE` `FIR_COEF_SIZE`	`8` `(1<<FFT_SIZE)`
Poznámky:	

H APLIKAČNÍ LIST EFEKTOVÉ KNIHOVNY MUSIC_PITCH

H.1 Funkce

Popis:

CORDIC převod z algebraického tvaru do polárního

Definice:

```
void CORD_alg2polar
(
      dsp16_t *magnitude,
      dsp16_t *argument,
      dsp16_complex_t algebraic,
      int iterations
)
```

Vstupní parametry:

magnitude	cílová adresa modulu komplexního čísla
argument	cílová adresa fáze komplexního čísla
algebraic	vstupní komplexní číslo v algebraickém tvaru
iterations	počet iterací (doporučeno 9 – 14)

Výstupní parametry:

žádné

Poznámky:

Počtem iterací je možné optimalizovat výpočet.

Popis:

CORDIC převod z polárního do algebraického tvaru

Definice:

```
void CORD_polar2alg
(
      dsp16_complex_t *algebraic,
      dsp16_t *magnitude,
      dsp16_t *argument,
      int iterations
)
```

Vstupní parametry:

algebraic	cílová adresa komplexního čísla v algebraickém tvaru
magnitude	ukazatel na vstupní modul komplexního čísla
argument	ukazatel na vstupní argument komplexního čísla
iterations	počet iterací (doporučeno 9 – 14)

Výstupní parametry:

žádné

Poznámky:

Počtem iterací je možné optimalizovat výpočet.

Popis:

Efekt pitch-scale

Definice:

```
int pitch_scale
(
      dsp16_t *data_out,
      dsp16_t *data_in,
      unsigned short pitch
)
```

Vstupní parametry:

data_out	ukazatel na výstupní buffer, velikost DAT_LEN
data_in	ukazatel na vstupní buffer, velikost DAT_LEN
pitch	pitch koeficient ve formátu unsigned Q(16-PITCH_FRACT).(PITCH_FRACT)

Výstupní parametry:

0	výpočet proběhl bez problémů

Poznámky:

H.2 Definované konstanty

Popis:

Definice parametrů pitch scale efektu

Definice:	**Implicitní hodnota:**
FFT_WIDTH_2	10
OVERLAP	49152
STFT_NUM	8

Poznámky:

Hodnota OVERLAP implicitně nastavena na 75% (49152)

Popis:

Definice konstant pitch scale efektu

Definice:	**Implicitní hodnota:**
FFT_WIDTH	(1<<FFT_WIDTH_2)
FFT_WIDTH_HALF	(1<<(FFT_WIDTH_2-1))
OVERLAP_SAMP	((FFT_WIDTH*OVERLAP)>>16)
DATA_STEP	(FFT_WIDTH-OVERLAP_SAMP)
SAMPLES	(DATA_STEP*STFT_NUM)
DAT_LEN	(SAMPLES+OVERLAP_SAMP)

Poznámky:

Tyto definice jsou striktně dány a určují i některé parametry pro uživatele. SAMPLES – velikost vstupních dat, DAT_LEN – velikost pomocných bufferů (vstupní parametry *pitch_scale*), OVERLAP_SAMP – offset dat v pomocných bufferech.