TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky a mezioborových inženýrských studií

Studijní program: M 2602 – Elektrotechnika a informatika

Studijní obor: 3902T005 – Automatické řízení a inženýrská informatika

Příznakové rozpoznávání pomocí signálového procesoru

Pattern Recognition Using Signal Processor

Diplomová práce

Autor: **Ondřej Merta**

Vedoucí práce: Ing. Lukáš Matela, Ph.D.

Konzultant: Ing. Jiří Bažant

V Liberci 18. 5. 2007

- originál zadání -

PROHLÁŠENÍ

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé diplomové práce a prohlašuji, že **s o u h l a s í m** s případným užitím mé diplomové práce (prodej, zapůjčení apod.).

Jsem si vědom toho, že užít své diplomové práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce a konzultantem.

Datum: 18. 5. 2007

Podpis

Příznakové rozpoznávání pomocí signálového procesoru

Abstrakt

Cílem této práce bylo realizovat na vývojovém přípravku ADSP-BF533 EZ-Kit Lite algoritmy pro rozpoznávání na základě maximální pravděpodobnosti. Pro navržený rozpoznávač byly sestaveny a implementovány postupy pro záznam zvuku z audio vstupu, jeho předzpracování a rozpoznání, s využitím vytvořené statistické analýzy. Zároveň bylo vytvořeno jednoduché ovládání užitím prvků dostupných na přípravku. Řešení bylo otestováno na demonstrační úloze, která prokázala schopnosti a potenciál pro případné reálné nasazení jako samostatný rozpoznávací člen.

Klíčová slova: Signálový procesor, příznakové rozpoznávání, klasifikace

Pattern recognition using signal processor

Abstract

The aim of this work was to realize algorithms for recognition using maximal probability on a device ADSP-BF533 EZ-Kit Lite. Processes for audio registering, preprocessing and recognition using created statistical analysis were put together and implemented for proposed recognizer. Simple control interface with use of available components was created as well. Solution was tested on a demo problem which proved capabilities and potential for possible real use as independent recognition unit.

Keywords: Signal processor, pattern recognition, classification

OBSAH

SEZNAM ZKRATEK A TERMÍNŮ

AD převodník – převodník signálu z analogové do digitální (číslicové) podoby

ALU – Arithmetic Logical Unit – aritmeticko-logická jednotka

ASIC – Application-Specific Integrated Circuits – integrované obvody pro konkrétní aplikace

BTC – Background Telemetry Channels – „kanály telemetrie na pozadí“

CISC – Complex Instruction Set Computer – počítač se sadou komplexních instrukcí

DA převodník – převodník signálu z digitální (číslicové) do analogové podoby

DAG – Data Address Generator – generátor adres dat

DSP – Digital Signal Processing/Processor – digitální signálové zpracování/procesor

FFT – Fast Fourier Transform – rychlá fourierova transformace

FIR filtr – filtr s konečnou časovou odezvou (Finite Impulse Response)

FPGA – Field-Programmable Gate Array – programovatelné hradlové pole

IIR filtr – filtr s nekonečnou časovou odezvou (Infinite Impulse Response)

in-place algoritmus – algoritmus, jehož vstup je za běhu přepsán výstupními hodnotami

JTAG – Joint Test Action Group – obvyklý název rozhraní pro testovací přístup a tzv. boundary scan („okrajové snímání“) umožňující přímé ladění procesoru

LU dekompozice – rozklad matice na horní (Upper) a dolní (Lower) trojúhelníkovou matici

MAC – Multiply-and-Accumulate

MMX – MultiMedia / Multiple Math / Matrix Math eXtension

RISC – Reduced Instruction Set Computer – počítač s redukovanou sadou instrukcí

SIMD – Single Instruction Multiple Data

SPI – Serial Peripheral Interface – sériové periferní rozhraní

TDM – Time Division Multiplex – časový multiplex

TWI – Two-Wired Interface - „dvoudrátové rozhraní“

VLIW – Very Long Instruction Word

ÚVOD

Práce se zabývá vytvořením a implementací postupů sloužících pro příznakové rozpoznávání signálů na základě maximální pravděpodobnosti. Platformou byla vývojová deska osazená signálovým procesorem Blackfin 533 od firmy Analog Devices a vývojovým prostředím VisualDSP++ 4.5, které s podobnými přípravky firma poskytuje. Zařízení obsahuje audio i video vstupy (a výstupy). Pro realizaci byl zvolen zvukový vstupní signál.

Teoretická část práce pojímá postupy zpracování zvuku se zaměřením na digitální zpracování, základy teorie rozpoznávání, některé postupy a používané metody s objasněním v této práci použitých, včetně matematického aparátu. Technická část se zabývá vlastnostmi, možnostmi a nastavením přípravku, na kterém byla úloha realizována a též okrajově vývojovým prostředím, které pro vývoj posloužilo. Následuje popis vlastní realizace, který v několika kapitolách objasňuje implementované postupy, přibližuje ovládání, výstupy a další možnosti rozšíření funkčnosti. Poslední část ukazuje demonstrační úlohu a její výsledky. V příloze jsou pak uvedeny zdrojové kódy vytvořené aplikace.

1 DIGITÁLNÍ ZPRACOVÁNÍ ZVUKU

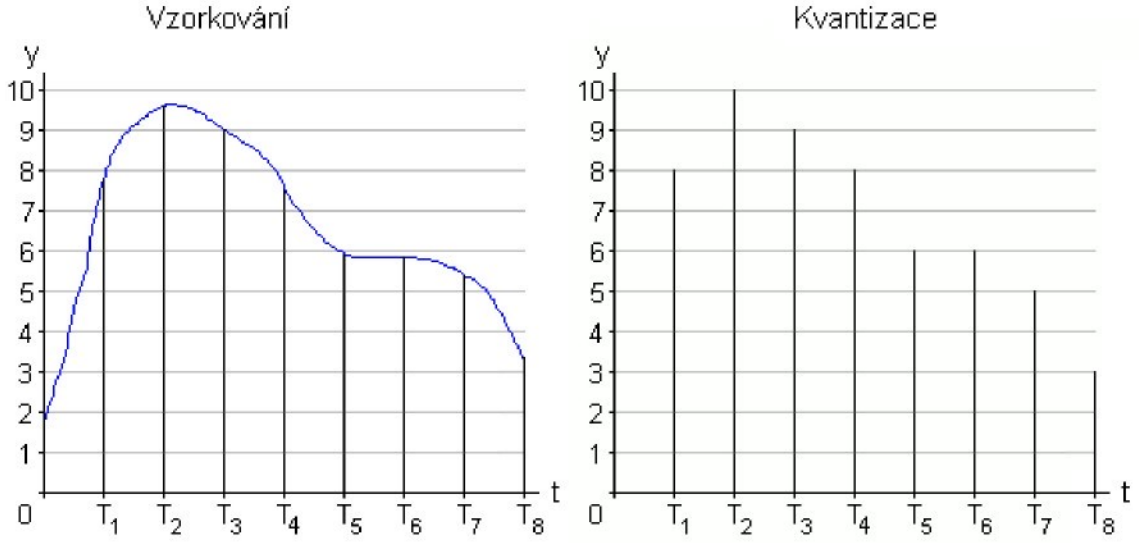
Zvuk je mechanické vlnění šířící se hmotou. Charakteristickými parametry jsou frekvence, vlnová délka, perioda, amplituda a rychlost. Zpracování zvuku se realizuje na reprezentaci zvukových signálů a to buďto analogové nebo digitální. Analogová reprezentace je obvykle elektrická, kde úroveň napětí odpovídá tlaku zvukové vlny. Obdobně digitální reprezentace vyjadřuje tlak zvukové vlny jako symboly, nejčastěji binární hodnoty, které umožňují digitální zpracování. Nutno poznamenat, že všechny reálné zvukové signály jsou v čase spojité analogové. Proto je nutné provádět vzorkování a kvantizaci, pro konverzi spojitého signálu na diskrétní digitální reprezentaci. Přestože tato konverze je ztrátová, většina moderních zvukových systémů používá tento přístup, jelikož postupy digitálního zpracování signálu jsou účinnější než zpracování analogového signálu a uživateli nabízí velice širokou základnu snadno realizovatelných funkcí.

Cílem digitálního zpracování signálů je obvykle měření či filtrace reálných signálů. První krokem je tedy obvykle konverze do digitální formy pomocí AD převodníku. Jako výstup se často požaduje opět analogový signál, který vyžaduje DA převodník. Algoritmy tohoto oboru někdy vyžadují specializované počítače, které používají speciální mikroprocesory – digitální signálové procesory (též zkracované na DSP). Tyto zpracovávají signál v reálném čase a jsou často konstruovány pro konkrétní účel (tzv. ASIC).

1.1 Digitalizace

Digitalizace analogového signálu se provádí ve dvou fázích – vzorkování a kvantizace (Obr. 1). Během vzorkování je signál převeden do diskrétní podoby. Spojitá časová základna je rozdělena na (obvykle rovnoměrné) úseky délky periody vzorkování. Kvantizace aproximuje hodnoty navzorkovaného signálu hodnotami z konečné množiny. Spojitý obor hodnot je rozdělen na konečný počet hodnot. Rozlišení závisí na bitové šířce převodníku.

Pro správnou rekonstrukci analogového signálu musí být dodržen Nyquist-Shannonův vzorkovací teorém. Tento teorém říká, že vzorkovací frekvence musí být alespoň dvakrát větší než šířka pásma (požadovaná).



Obr. 1: Digitalizace signálu

1.2 Časová oblast

Nejběžnějším přístupem zpracování signálu v časové oblasti je zlepšení signálu metodou filtrace. Filtrace obvykle sestává z transformace několika sousedních vzorků okolo aktuální hodnoty. Filtry lze charakterizovat z různých pohledů:

- lineární / nelineární
- kauzální / nekauzální
- časově invariantní / adaptivní
- stabilní / nestabilní
- s konečnou (FIR) / s nekonečnou (IIR) impulsní odezvou

Většinu filtrů lze popsat v Z-rovině pomocí přenosových funkcí. Filtr lze též popsat diferenční rovnicí, pomocí pólů a nul nebo v případě FIR filtru impulsní odezvou, kdy lze výstup filtru získat konvolucí vstupního signálu s impulsní odezvou filtru. Filtry lze též reprezentovat pomocí blokových diagramů, které mohou poté sloužit pro implementaci algoritmu hardwarovou cestou.

1.3 Frekvenční oblast

Signály jsou obvykle převáděny z časové do frekvenční oblasti pomocí Fourierovy transformace. Výsledkem této transformace jsou amplituda a fáze jednotlivých frekvencí (v rozsahu od 0 – stejnosměrná složka až po polovinu vzorkovací frekvence, s velikostí kroku závislou na počtu vzorků vstupujících do transformace). Běžně se tento výstup (reálná a imaginární složka) ještě převádí na modul (absolutní

hodnotu komplexního čísla). Účelem je analýza signálu ve frekvenční oblasti, kdy lze získat informaci a zastoupení jednotlivých frekvencí v signálu obsažených. Výpočet diskrétní fourierovy transformace se v praxi realizuje numerickými metodami pomocí algoritmu FFT (Fast Fourier Transform), tzv. rychlé fourierovy transformace, který je pro tyto účely optimalizován. Existuje celá řada různých implementací FFT algoritmu. Nejpoužívanější je algoritmus Cooley-Tukey publikovaný v roce 1965 (později se zjistilo, že algoritmus používal již Carl Friedrich Gauss kolem roku 1805). Jako další metody lze jmenovat algoritmus prvočíselných faktorů (Prime-factor FFT algorithm), Bruunův, Raderův, Bluesteinův algoritmus. (Čerpáno z článku o FFT v [5].)

1.4 Použití digitálního zpracování signálu

Hlavními aplikacemi DSP jsou zpracování audio signálů, audio komprese, digitální zpracování obrazu, video komprese, zpracování řeči, rozpoznávání řeči, digitální komunikace, radary. Konkrétnějším příkladem může být například komprese řeči pro účely přenosu v mobilních sítích, předpovídání počasí, zpracování seismických dat, analýza a řízení průmyslových procesů, počítačem generované animace ve filmech, zdravotnická diagnostická zobrazení (počítačová tomografie, magnetická rezonance), hudební efekty, apod.

2 STROJOVÉ UČENÍ

Strojové učení je podoblastí umělé inteligence, zabývající se algoritmy a technikami, které umožňují počítačovému systému „učit se". Učením v daném kontextu rozumíme takovou změnu vnitřního stavu, která zefektivní schopnost přizpůsobení se změnám okolního prostředí.

Strojové učení se značně prolíná s oblastmi statistiky a dobývání znalostí a má široké uplatnění. Jeho techniky se využívají např. v biomedicínské informatice (tzv. systémy pro podporu rozhodování), rozpoznávání řeči a psaného textu, či mnohé další.

Algoritmy strojového učení lze podle způsobu učení rozdělit do následujících kategorií:

- učení s učitelem
- učení bez učitele
- kombinace učení s učitelem a bez učitele
- učení posilováním
- transdukce
- „učení učit se"

V této práci je realizováno učení s učitelem. Obecným cílem učení s učitelem je vytvořit funkci na základě trénovacích dat. Trénovací data sestávají z páru *vstup* (typicky vektor) a požadovaný *výstup*. Výstupem této funkce může být spojitá hodnota nebo předpokládaná třída vstupního objektu (pak mluvíme o klasifikaci). Úkolem „žáka" je předpovědět výstupní hodnotu pro libovolný platný vstupní objekt na základě shlédnutí dostatečného množství trénovacích vzorků. Pro dosažení tohoto je třeba vyvodit obecné předpoklady z poskytnutých dat, tak aby bylo možné je rozumně aplikovat na nové situace.

Vzhledem k cílům práce nebudou vysvětleny různé další metody a postupy, které do tohoto velice širokého a neustále se vyvíjejícího oboru spadají. Pro přiblížení lze uvést příklady některých používaných modelů (pod těmito hesly lze dohledat spoustu dalších informací, např. v online encyklopedii [5]): rozhodovací stromy (decision trees), algoritmus k-nejbližších sousedů (k-nearest neighbor), lineární diskriminační analýza (linear discriminant analysis), množina rozhodovacích pravidel, perceptron, Bayesovské sítě (Bayesian network), neuronové sítě (artificial neural network), genetické algoritmy (gene algorithms), dynamické programování (dynamic programming), kvadratický klasifikátor (quadratic classifier), Support vector machines, apod.

2.1 Příznakové rozpoznávání

Příznakové rozpoznávání si klade za cíl vytvoření klasifikátoru, který bude schopen zařadit vstupující příznaky do jedné ze tříd na základě předchozích znalostí nebo pomocí statistických informací získaných z příznaků. *Třídou* se zde rozumí množina prvků s podobnými vlastnostmi, kde *podobnost* (similarity) definujeme jako vlastnost měřitelnou na obrazu objektu umožňující vyjádřit vztah ke každé ze tříd.

Klasifikovanými příznaky je obvykle množina měření či pozorování, definující body v příslušném vícerozměrném prostoru. Příznaky charakterizují objekt pomocí kvantitativního či kvalitativního popisu – tvoří tzv. *obraz* objektu. Rozpoznávání se zásadně děje na obrazu objektu, ne na vlastním objektu, ten je často neuchopitelný. Obraz lze zvolit podle různých kritérií: nejmenší chyba klasifikace, obraz snadno (levně) získatelný, nejefektivnější porovnání obrazů.

Úplný systém s příznakovým rozpoznáváním sestává ze senzoru, který nashromáždí rozpoznávaná pozorování, dále mechanismu extrakce příznaků, který vypočítá číselné nebo symbolické informace z pozorování a rozhodovacích pravidel, která vykonávají vlastní práci zařazení pozorování na základě získaných příznaků.

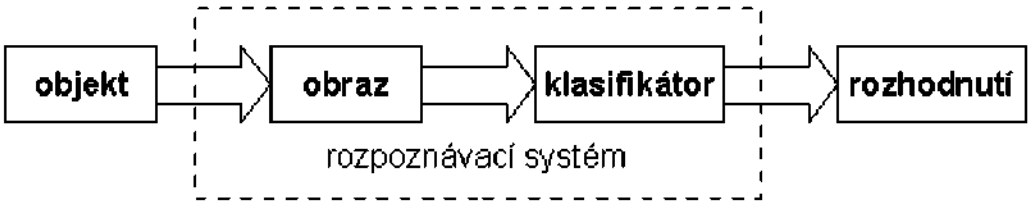
Rozhodovací pravidla jsou obvykle vytvořena na základě dostupnosti již klasifikovaných vzorů (tzv. trénovací množiny), pak se jedná o učení s učitelem. Lze však aplikovat i učení bez učitele. V tomto případě není systému poskytnuta předchozí klasifikace, místo toho si systém vytvoří třídy sám na základě statistických podobností příznaků.

Typickými aplikacemi příznakového rozpoznávání jsou automatické rozpoznávání řeči, klasifikace textu do různých kategorií (např. spam v emailových zprávách), automatické rozpoznávání ručně psaných adres na poštovních obálkách, automatické rozpoznávání obrazů lidských tváří. Poslední dva příklady spadají do oblasti analýzy obrazu, kde jako rozpoznávaná data vstupuje digitální obraz.

2.1.1 Klasifikace

Základní úlohou klasifikace (jednoduché schéma je na Obr. 2), jak již bylo uvedeno, je přiřadit objekt na základě jeho obrazu do jedné ze tříd, přičemž lze zvolit jednu z následujících možností:

- zařadit vždy do jedné ze tříd
- zařadit do jedné ze tříd s možností odmítnutí zařazení (rejection)
- možnost zařadit objekt do více tříd



Obr. 2: Schéma rozpoznávacího systému

Klíčovým aspektem klasifikace je fakt, že neexistují dva identické objekty. Rozpoznávání tedy pracuje s daty, jejichž popis má náhodný charakter. Daný klasifikátor musí tedy uvažovat prvek náhodnosti, určité odchylky od „normálu" musí tolerovat – tzv. *robustnost*.

Klasifikátorem rozumíme systém s N vstupy (vstupním vektorem velikosti N) a jedním výstupem, kde vstupem je *příznakový vektor* $\vec{x}=(x_1, x_2, \ldots, x_N)$, reprezentující obraz z obrazového prostoru (euklidovský prostor dimenze N) a výstupem je *index třídy* t, do které byl vstupní vektor na základě *pravidla* přiřazen. Třídy jsou zpravidla značeny $T_1, T_2, \ldots, T_R$, kde R udává počet tříd.

Pro vlastní klasifikaci lze aplikovat celou řadu algoritmů od jednoduchého Bayesovského klasifikátoru až po neuronové sítě. Nejčastější formy jsou:

- metoda diskriminačních funkcí
- metoda minimální vzdálenosti
- **metoda maximální pravděpodobnosti (minimální chyby)**

Klasifikátor vytvořený v této práci využívá metodu maximální pravděpodobnosti.

2.1.2 Metoda maximální pravděpodobnosti

Základním principem je, že každá třída je reprezentována:

- apriorní pravděpodobností třídy $P(T_r)$, přičemž $\sum_{r=1}^{R} P(T_r)=1$ a
- podmíněnou hustotou pravděpodobnosti $p(\vec{x}|T_r)$,

  která udává rozložení pravděpodobnosti vektoru příznaků $\vec{x}$ pro třídu T_r.

Trénování pak probíhá tak, že se pro každou třídu na trénovací množině určí (odhadnou) výše uvedené pravděpodobnosti.

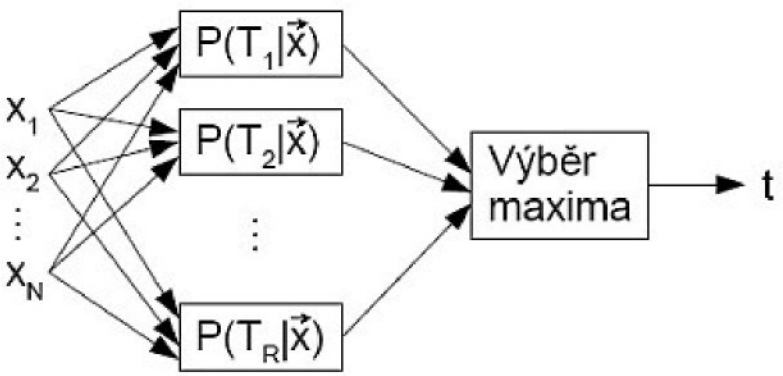
Při rozpoznávání se aplikuje Bayesovo pravidlo

$$P(T_r|\vec{x}) = \frac{p(\vec{x}|T_r)P(T_r)}{p(\vec{x})}, \qquad \text{(Rovnice 1)}$$

kde $P(T_r|\vec{x})$ je aposteriorní pravděpodobnost, že $\vec{x}$ patří do třídy T_r,

$$p(\vec{x}) = \sum_{i=1}^{R} p(\vec{x}|T_i)P(T_i) \qquad \text{(Rovnice 2)}$$

je absolutní pravděpodobnost – hustota rozložení vektoru příznaků (nezávisle na třídě). Klasifikátor poté vybere třídu s největší pravděpodobností a její index (označení) prohlásí za výsledek (Obr. 3).



Obr. 3: Schéma klasifikátoru

Pro hustotu pravděpodobnosti se nejčastěji volí normální (Gaussovo) rozložení

$$p(\vec{x}|T_r)=\frac{1}{\sqrt{(2\pi)^R det\ \Sigma}}e^{-\frac{1}{2}(\vec{x}-\vec{\bar{x}})^T\Sigma^{-1}(\vec{x}-\vec{\bar{x}})}, \qquad \text{(Rovnice 3)}$$

kde $\vec{\bar{x}}$ představuje vektor středních hodnot a

Σ je matice kovariancí definovaná následovně:

$$\Sigma=\begin{bmatrix}\sigma_{11}^2 & \sigma_{12}^2 & \cdots & \sigma_{1n}^2\\ \sigma_{21}^2 & \sigma_{22}^2 & & \sigma_{2n}^2\\ \vdots & & \ddots & \vdots\\ \sigma_{n1}^2 & \sigma_{n2}^2 & \cdots & \sigma_{nn}^2\end{bmatrix}, \qquad \text{(Rovnice 4)}$$

přičemž $\sigma_{ij}^2=E\left[(x_i-\bar{x}_i)(x_j-\bar{x}_j)\right]$ a

E značí operátor průměr.

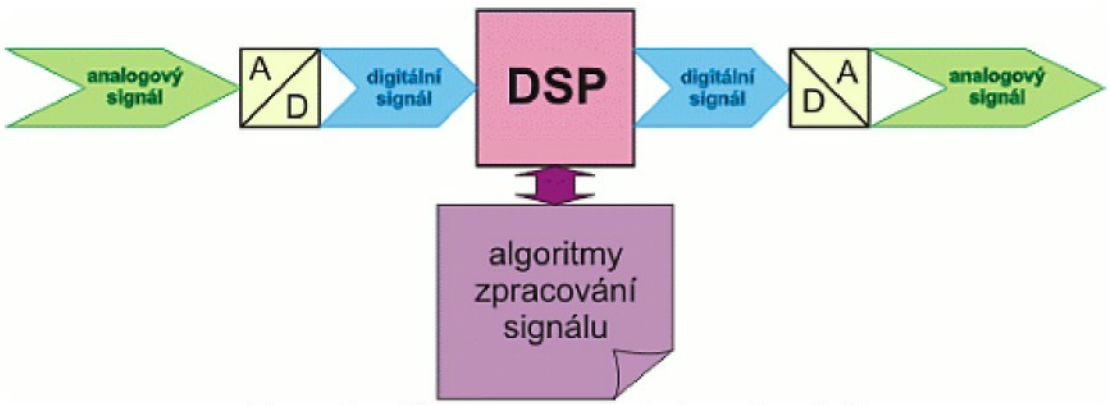
3 SIGNÁLOVÝ PROCESOR

Digitální signálové procesory (DSP) jsou specializované mikroprocesory, jejichž specificky navržený hardware a instrukční soubor jim umožňuje účinně provádět matematické výpočty, které se objevují v metodách číslicového zpracování signálů a to obecně v reálném čase. Tvoří jádro mnoha aplikací ve spotřební elektronice, v telekomunikacích, v medicíně a v průmyslu.

3.1 Obecná charakteristika

Jedním ze základních důvodů k vytvoření DSP byl fakt, že klasické analogové obvody sloužící pro zpracování signálu bývají náročné z hlediska návrhu, nastavení, provedení a reprodukovatelnosti, přičemž jakmile jsou vyrobeny, nelze jejich funkci téměř modifikovat. Ve srovnání s tím cena digitálních obvodů stále klesá a výkon roste.

Na Obr. 4 je typické blokové schéma zařízení využívajícího DSP. Analogový signál je nejprve převeden AD převodníkem na digitální a v této podobě je průběžně zpracováván digitálním signálním procesorem. Zpracovaný digitální signál je DA převodníkem zpět převeden na analogový. V mnoha zařízeních prochází signál tímto řetězcem v reálném čase, ale na některých signálech je potřeba provést tak složité a výpočetně náročné algoritmy, že to ani velmi rychlý DSP procesor v reálném čase nestihne a digitalizovaná data musí být nejprve zaznamenána do paměti a odtamtud teprve postupně zpracovávána.



Obr. 4: Typický řetězec zpracování signálu v DSP

V některých zařízeních je použita jen polovina tohoto typického řetězce nebo jsou sice použity obě poloviny, ale v samostatných oddělených řetězcích zpracovávaných samostatnými procesory. Užití části řetězce nastává například v CD přehrávači, kdy je signál z kompaktního disku čten v digitální podobě, zpracován digitálním signálovým procesorem a nakonec převeden na analogový. Příkladem

odděleného zpracování je mobilní telefon, ve kterém se obvykle zpracovává vysílaný a přijímaný signál odděleně v samostatných procesorech.

3.1.1 Architektura DSP

Typický digitální signální procesor je vystavěn na harvardské architektuře. Tato architektura má oproti von Neumannovu počítači oddělenou paměť pro program od paměti pro data. V praxi to znamená, že data a kód programu využívají vlastní sběrnice, což zvyšuje propustnost systému.

Dalšího zrychlení výpočtů se dosahuje pomocí specializovaných výpočetních jednotek procesoru, které dokáží pracovat paralelně. Typický DSP má kromě aritmeticko-logické jednotky (ALU) navíc rychlou násobičku, která dokáže operaci násobení s přičítáním $A \leftarrow A+B \times k$. Tato operace je základní operací většiny algoritmů digitálního zpracování signálu. DSP zpravidla obsahuje dvě nebo více nezávislých adresních jednotek, tzv. DAG (Data Address Generator), adresujících data v lineárních nebo kruhových bufferech. Typický DSP tak umožňuje během jednoho taktu provést jeden krok skalárního násobení dvou vektorů (vynásobení hodnot ze dvou bufferů, přičtení do akumulátoru, posun na další index v bufferech). Procesor s klasickou architekturou by na stejnou operaci potřeboval několik taktů (např. 1. načtení hodnoty z prvního bufferu, 2. vynásobení hodnotou z druhého bufferu, 3. přičtení výsledku do akumulátoru, 4. posun adresy prvního bufferu, 5. posun adresy druhého bufferu).

3.1.2 Rozdělení DSP

Základním dělením digitálních signálních procesorů je dělení podle použité aritmetiky. Existují DSP pracující:

- v celočíselné aritmetice
- v aritmetice s pevnou řádovou čárkou
- v aritmetice s plovoucí řádovou čárkou

Procesory s celočíselnou aritmetikou jsou sice levné, ale algoritmy výpočtů stále narážejí na nutnost převádět reálná čísla na celá a mezivýsledky výpočtů se musí neustále upravovat tzv. normalizacemi. Proto je vývoj algoritmů v těchto typech procesorů výrazně náročnější. Hodí se proto zejména pro masovou produkci výrobků, kde nevadí poněkud vyšší cena vývoje, ale důležitá je zejména cena samotné součástky.

Procesory s plovoucí řádovou čárkou jsou sice složitější a dražší, ale vývoj softwaru je pro ně výrazně jednodušší. Nevýhodou zde může být rovněž vyšší spotřeba energie.

Procesory s pevnou řádovou čárkou mohou sice být určitým kompromisem, ale prakticky je nelze jasně odlišit od procesorů pracujících v celočíselné aritmetice.

Dalším kritériem pro dělení digitálních signálních procesorů je šířka jejich datové sběrnice. Ta bývá od 16 bitů výše. Další dělení může být na jednojádrové nebo vícejádrové DSP.

3.1.3 Vývoj a výroba DSP

Algoritmy zpracování digitálních signálů jsou často velmi složité a často se celé nebo alespoň jejich podstatné části v různých aplikacích opakují. Proto je často výhodné vyvíjet specializované procesory obsahující již potřebné algoritmy. Firmy zabývající se vývojem DSP se proto dělí na firmy vyvíjející hardware procesorů a firmy vyvíjející algoritmy. Hardwarový návrh DSP je pak často prodáván jako tzv. DSP core (jádro). Toto jádro je použito k výrobě specializovaných integrovaných obvodů, které ho doplňují o další potřebné součástky - například A/D a D/A převodníky a paměť ROM obsahující patřičné algoritmy. Například typický DSP pro mobilní telefony obsahuje 2 - 4 samostatná DSP jádra a veškeré algoritmy potřebné pro zpracování řečového signálu na GSM a naopak.

3.1.4 Historie

V roce 1979 firma Bell Laboratories představila první jednočipový DSP, Mac 4 Microprocessor. V Roce 1980 na konferenci IEEE International Solid-State Circuits Conference '80 firma NEC prezentovala μPD7720 a AT&T představila DSP1 – první samostatné kompletní DSP. Oba inspirované výzkumem v PSTN telecommunicacions.

První DSP vyrobený firmou Texas Instruments, TMS32010, představený v roce 1983 se ukázal ještě větším úspěchem. Byl založen na harvardské architektuře, již obsahoval speciální instrukční sadu s instrukcemi jako load-and-accumulate nebo multiply-and-accumulate. Byl schopen pracovat s 16bitovými čísly a pro operaci multiply-add („vynásob a přičti“) potřeboval 390 ns. Texas Instruments je v současné době vedoucím producentem univerzálních DSP. Dalším úspěšným návrhem byla Motorola 56000.

O pět let později se začala rozšiřovat druhá generace DSP. Ty měly tři paměti pro současné ukládání dvou operandů, obsahovaly hardware pro urychlení krátkých smyček. Některé operovaly na 24bitových proměnných a běžnému modelu stačilo přibližně 21 ns na vykonání operace multiply-and-accumulate (MAC). Zástupci této generace byly například DSP16A od AT&T nebo Motorola DSP56001.

Hlavním zlepšením ve třetí generaci bylo objevení se aplikačně specifických jednotek a instrukcí v datových cestách nebo jako koprocesory. Tyto jednotky dovolovaly přímou hardwarovou akceleraci velmi specifických, ale komplexních matematických problémů, jakými jsou Fourierova transformace nebo maticové výpočty. Některé čipy, jako Motorola MC68356, dokonce obsahovaly více procesorových jader, která pracovala paralelně. Dalšími zástupci z roku 1995 jsou například TMS320C541 nebo TMS320C80 od Texas Instruments.

Čtvrtá řada je charakterizována změnami v instrukční sadě a kódováním/dekódováním instrukcí. Byly přidány SIMD (Single Instruction Multiple Data) a MMX rozšíření (*MultiMedia / Multiple Math / Matrix Math eXtension* – spíše obchodní značka, označuje speciální instrukční sadu přidanou do procesoru pro multimediální/maticové výpočty), objevuje se VLIW (Very Long Instruction Word – velmi dlouhé instrukce – instrukce kóduje několik operací, které se spouští paralelně) a superskalární architektura. A jako obvykle se zvýšily taktovací frekvence, pro MAC (multiply-and-accumulate) nyní stačí 3 ns.

3.1.5 DSP v roce 2007

Dnešní signálové procesory mají podstatně vyšší výkony. A to díky technologickým i návrhovým pokrokům, rychlo přístupovým cache druhé úrovně, (E)DMA obvodům a širším sběrnicím. Samozřejmě ne všechny DSP nabízejí stejné rychlosti a existuje velmi mnoho různých typů, každý vhodný pro jiné specifické uplatnění. Série C6000 od Texas Instruments je taktována na 1 GHz a implementuje oddělené instrukční a datové cache a stejně tak 8 MB L2 cache, přičemž I/O rychlost je pozoruhodná díky 64 EDMA kanálům. Špičkové modely jsou dokonce schopny 8000 MIPS (miliónů instrukcí za sekundu), používají VLIW, provádějí 8 operací během cyklu a jsou kompatibilní s celou škálou externích periférií a různých sběrnic.

Dalším silným zástupcem je i firma Analog Devices, která taktéž nabízí širokou škálu DSP, jejím hlavním portfoliem však jsou multimediální procesory jako kodeky, filtry a DA převodníky. Zástupcem od této firmy je i řada procesorů Blackfin. Tyto

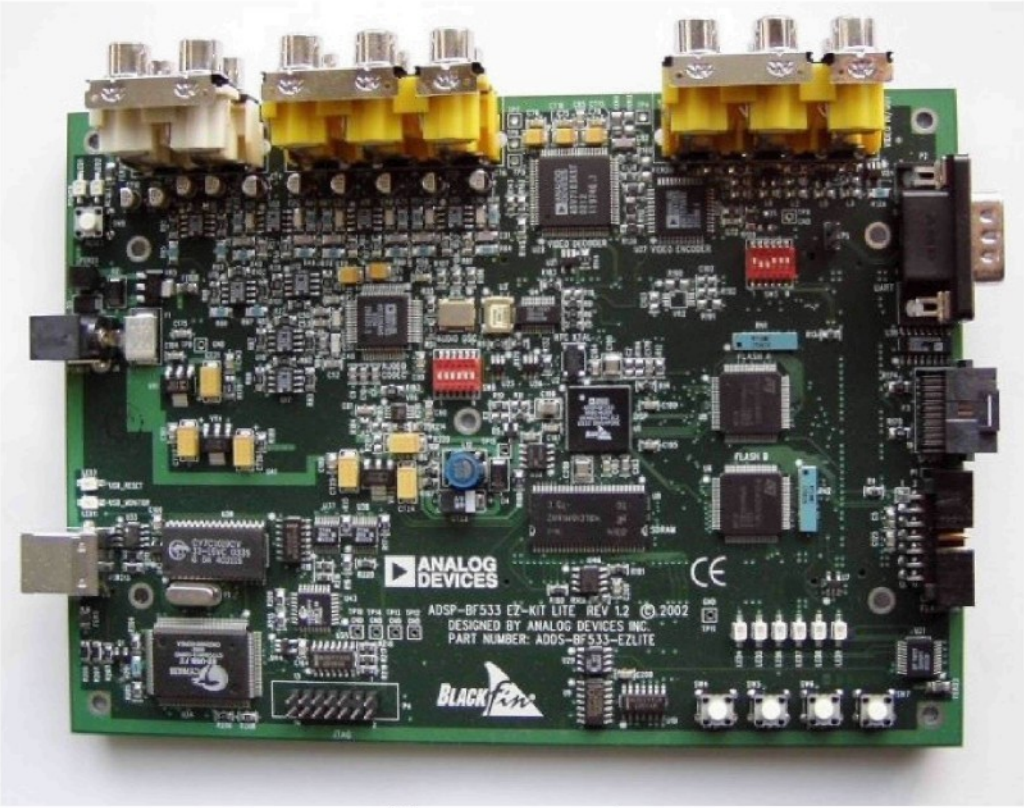
procesory v sobě kombinují schopnosti DSP a procesoru pro běžné použití. Výsledkem je, že na těchto procesorech lze provozovat jednoduché operační systémy jako μCLinux, velOSity nebo Nucleus a přitom relativně efektivně zpracovávají i real-time data.

Většina DSP pracuje s aritmetikou s pevnou řádovou čárkou, protože v reálném nasazení dodatečná přesnost, kterou by přinesla plovoucí řádová čárka není potřeba a zároveň je tím dosažen i rychlostní zisk. DSP s plovoucí řádovou čárkou jsou však nasazovány ve vědeckých aplikacích, kde může být dodatečná přesnost vyžadována. Procesory pro běžné použití se v mnohém inspirovaly u signálových procesorů, například rozšířením MMX. Obecně lze říci, že DSP jsou integrované obvody s úzkým zaměřením. Jejich funkci lze za cenu složitějšího vývoje realizovat i pomocí FPGA čipů.

Řešení v této práci používá procesor Blackfin založený na upravené harvardské architektuře v kombinaci s hierarchickou paměťovou strukturou. Je založen na modelu RISC (Reduced Instruction Set Computer), který oproti CISC (Complex Instruction Set Computer) neobsahuje různá rozšíření instrukční sady (jako např. MMX). Instrukční sada procesoru je optimalizována tak, že nejpoužívanější instrukce jsou kódovány 16 bity, složitější instrukce jsou pak kódovány jako vícefunkční 32bitové. Procesor umožňuje spouštění některých 32bitových instrukcí paralelně se dvěma 16bitovými, čímž lze efektivně využít jádro procesoru v jednom výpočetním cyklu. Jazyk symbolických instrukcí (assembly language) používá algebraickou syntaxi, která je optimalizována pro použití překladače pro programovací jazyk C. Procesor je taktován na frekvenci 600 MHz, přičemž tato frekvence může být procesorem za běhu regulována pro účely úspory energie. Procesor vykoná 1512 milionů MAC operací za sekundu a pracuje s pevnou řádovou čárkou. Další údaje jsou uvedeny v následující kapitole.

3.2 ADSP-BF533 EZ-Kit Lite

Pro účely této práce byl použit vývojový přípravek ADSP-BF533 EZ-Kit Lite od firmy Analog Devices (Obr. 5), na bázi signálového procesoru Blackfin 533, který je vhodný jak pro zpracování signálů z audio vstupů, tak pro zpracování videa. Kit má na sobě implementovány video enkodér ADV7171, video dekodér ADV7183 a audio kodek AD1836.



Obr. 5: Vývojový kit

Procesor ADSP- BF533 má vnitřní paměť SRAM, která může být použita pro instrukce nebo pro ukládání dat. Přípravek obsahuje dva typy externí paměti – SDRAM a FLASH paměť. Velikost SDRAM je 64 MBytů (32M × 16 bit). Flash paměť je implementována se dvěma Dual-Bank Flash paměťovými zařízeními. Tato zařízení zahrnují primární a sekundární flash paměť stejně jako vnitřní SRAM a registry. Primární flash paměť o celkové velikosti 2 MByty je mapovaná do dvou oddělených asynchronních paměťových bloků o velikosti 1 MByte. Sekundární flash paměť, spolu se SRAM a registry, zabírá třetí banku asynchronního paměťového prostoru.

3.2.1 Flash paměť

Každé paměťové zařízení zahrnuje následující části:

- 1 MByte primární flash paměti
- 64 KByty ze sekundární flash paměti
- 32 KByty vnitřní SRAM
- 256 Bytů konfiguračních registrů (IO ovladače)

Přístup ke každé části může být 8 nebo 16bitový. Asynchronní paměťová banka 0 je povolena vždy po tvrdém resetu, zatímco banky 1 a 2 musí být povoleny softwarem. Tab. 1 udává příklad nastavení asynchronního paměťového konfiguračního registru.

Tab. 1: Příklad nastavení asynchromní paměti

Registr	**Hodnota**	**Funkce**
EBIU_AMBCTL0	0x7BB07BB0	Ovládání časování pro banky 1 a 0
EBIU_AMBCTL1 bity 15-0	0x7BB0	Ovládání časování pro banky 1 a 0 (banka 3 nepoužita)
EBIU_AMGCTL bity 3-0	0xF	Povolení všech bank

3.2.2 Video rozhraní

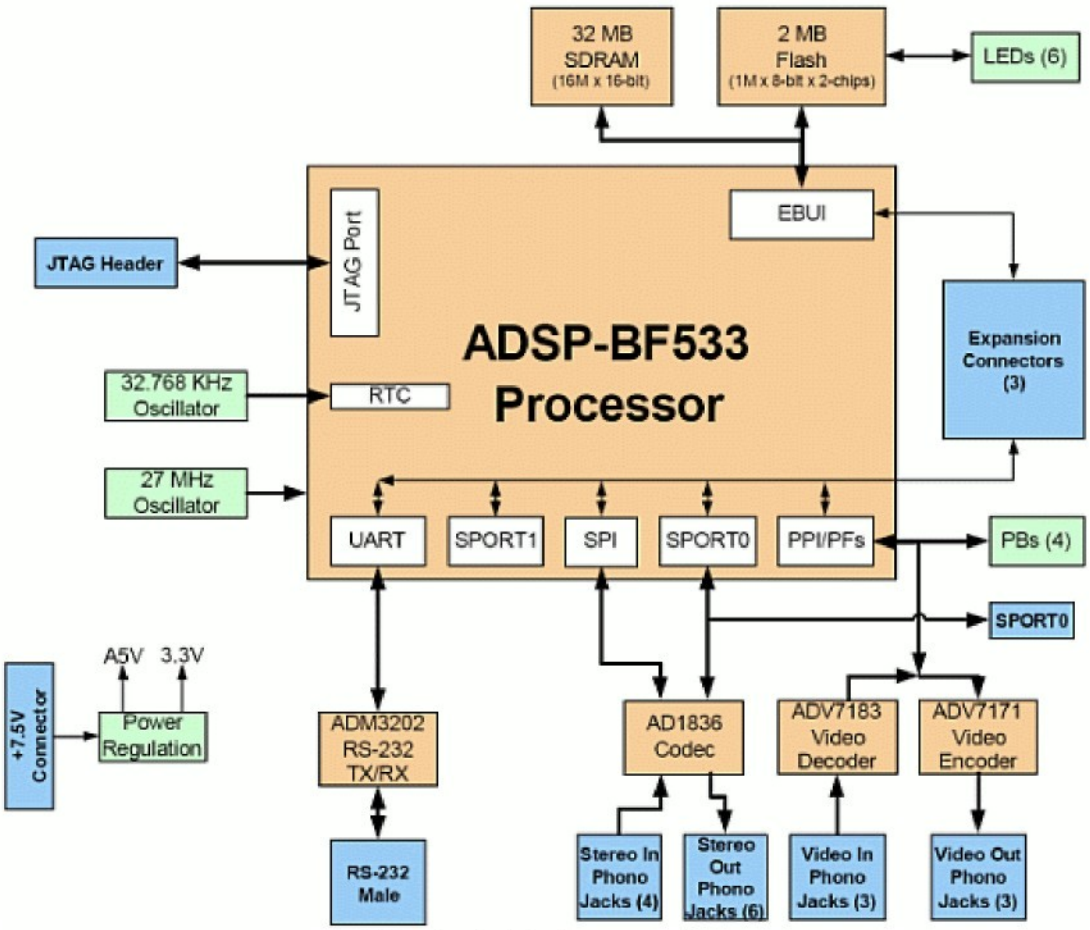
Deska podporuje video vstupní a výstupní aplikace. ADV7171 video kodér poskytuje až tři výstupní kanály analogového videa, zatímco ADV7183 video dekodér poskytuje až tři vstupní kanály analogového videa. Kodér a dekodér jsou připojeny k paralelnímu periferním rozhraní (PPI) procesoru.

3.2.3 Audio rozhraní

Přípravek taktéž obsahuje audio kodér-dekodér AD1836, který nabízí dva vstupní a tři výstupní stereo kanály. Připojen je přes sériové rozhraní SPORT0 a konfiguruje se pomocí portu SPI.

3.2.4 Architektura systému

EZ-Kit Lite byl navržen k tomu, aby demonstroval schopnosti procesoru Blackfin 533. Procesor má IO voltáž 3,3 V. Napětí jádra je odvozené z této 3,3V zásoby a užívá vnitřní napěťový regulátor nebo externí regulátor 1,4 V. Když procesor pracuje na kmitočtech vyšších než 600 MHz, je nezbytné použít 1,4 V regulátor. Napětí jádra a hlavní taktovací kmitočet může být stanoven za běhu procesorem. Architektura je znázorněna na Obr. 6.



Obr. 6: Architektura systému

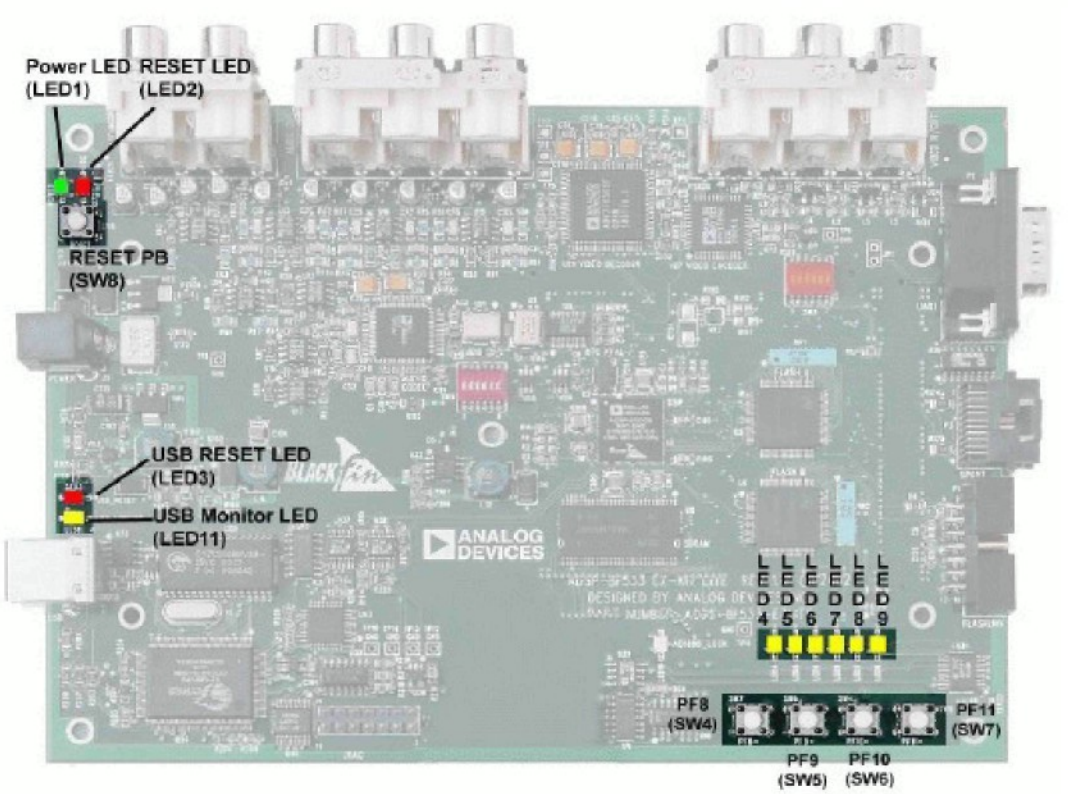
3.2.5 Externí jednotka rozhraní sběrnicového řadiče

Externí jednotka rozhraní sběrnicového řadiče (EBIU) připojuje vnější paměť procesoru ADSP-BF533. EBIU zahrnuje 16bitovou sběrnici dat, adresovou sběrnici a řídicí sběrnici. Jsou podporovány oba přístupy, jak 16 tak i 8bitový. Na EZ-Kit Lite EBIU jednotka spojuje SDRAM a flash paměť.

Zařízení poskytuje celkem 2 MByty primární flash paměti, 128 KBytů sekundární flash paměti a 64 KByty SRAM. Procesor může užívat tuto paměť pro zavádějící programy a pro ukládaní informací během normální činnosti.

3.2.6 LED a tlačítka

Obr. 7 ukazuje umístění všech LED a tlačítek. Funkčnost je dále popsána.



Obr. 7: LED a tlačítka

- **Programovatelná příznaková tlačítka (SW4–7)**

Čtyři tlačítka jsou volně k dispozici a představují univerzální uživatelský vstup. Tlačítko je aktivní při stisknutí a v tomto okamžiku posílá „1“ do procesoru.

- **Resetovací tlačítko (SW8)**

Resetovací tlačítko resetuje všechny prvky umístěné na desce. Jedinou výjimkou je USB interface čip. V případě, že kabel USB je komunikačně neprůchodný nedojde k resetování čipu i přesto, že komunikace s PC byla správně inicializována. Jedinou možností uvolnění komunikace přes USB je resetování čipu odpojením napětí.

- **LED napájení (LED1)**

Zelená LED1 signalizuje správné připojení desky ke zdroji napětí.

- **Resetovací LED (LED2, 3)**

Rozsvícení LED2 signalizuje reset všech hlavních komponent. Svítí-li LED3, znamená to reset čipu USB rozhraní. USB čip se resetuje pouze při startu, nebo pokud USB komunikace není inicializována.

- **Uživatelské LED (LED4-9)**

Šest LED připojených k univerzálním IO pinům flash paměti je k uživatelskému použití. Diody svítí, pokud je do příslušné adresy flash paměti zapsána „1“.

- **LED USB komunikace (LED11)**

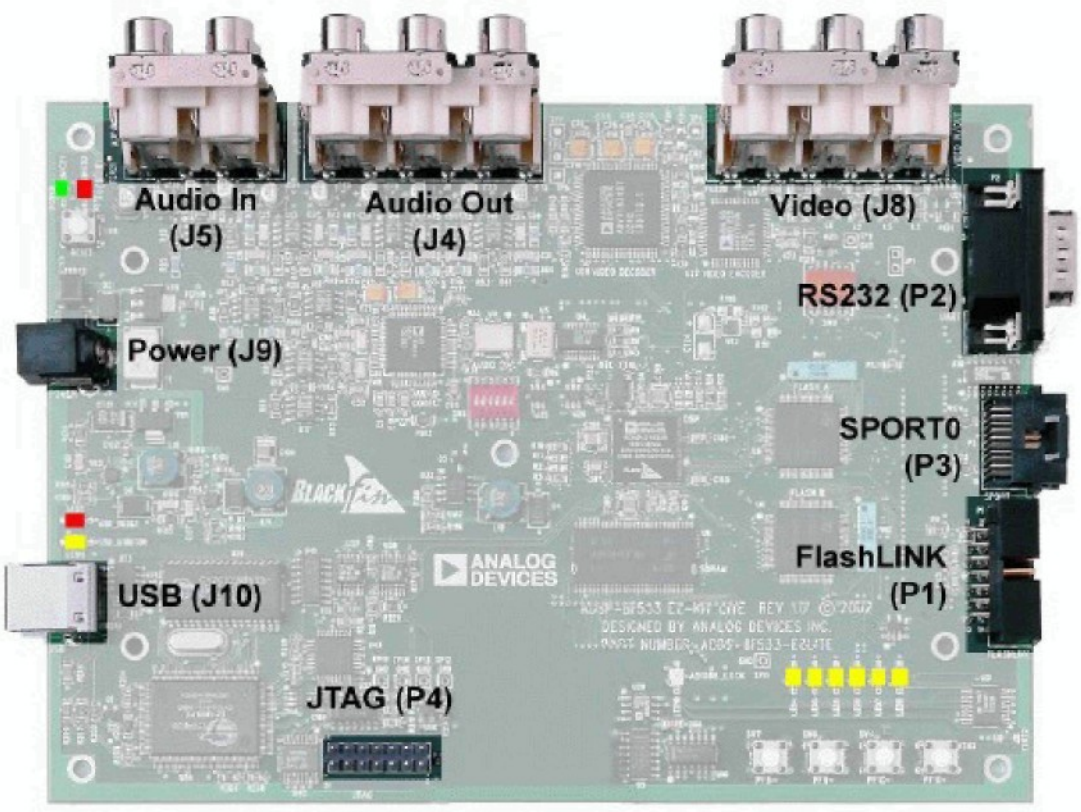
Komunikační LED USB signalizuje, že USB komunikace byla inicializována úspěšně a nyní je možno připojit procesor prostřednictvím VisualDSP++ k PC. Tato inicializace trvá přibližně 15 sekund. Pokud se LED nerozsvítí, je nutno znovu připojit napájení k desce nebo znovu nainstalovat USB ovladače.

- **JTAG emulační port**

JTAG emulační port dovolí emulátoru zpřístupnit nitro procesoru a vnější paměť skrz 6pinové rozhraní. JTAG emulační port procesoru je také připojený k USB ladícímu rozhraní. Je-li emulátor připojen k desce, pak USB ladící rozhraní je vyřazeno.

- **Konektory**

Na Obr. 8 je znázorněno umístění a označení konektorů: Napájení (Power), audio vstup/výstup (Audio in/out), video konektory, sériové rozhraní RS232, USB (typ B), připojení JTAG, port SPORT0 a flash programovací rozhraní.



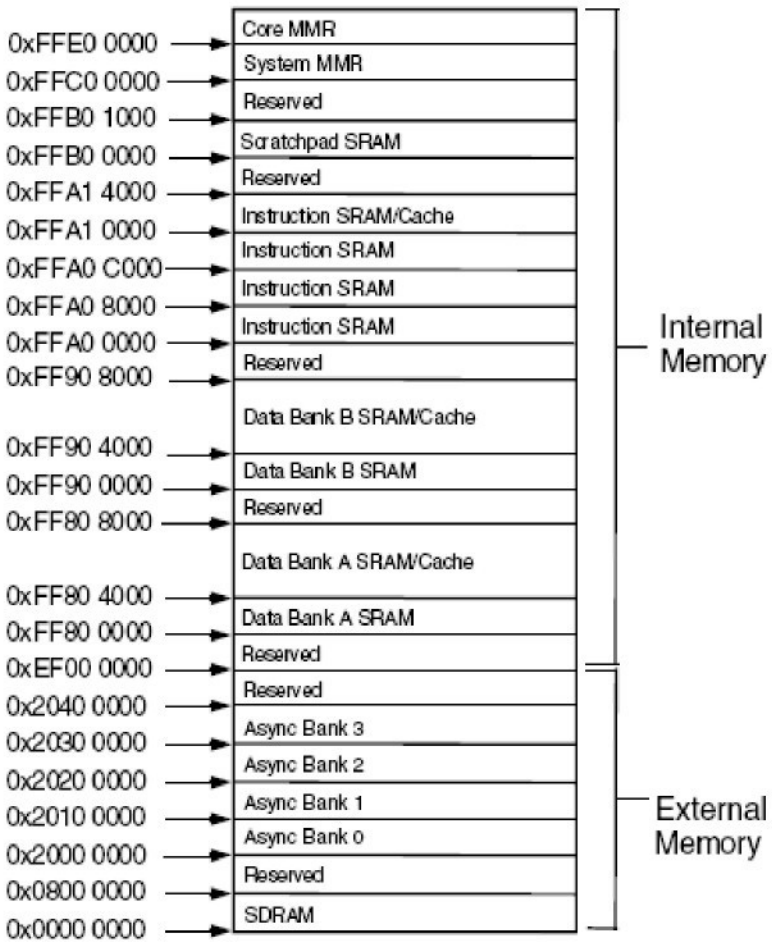
Obr. 8: Konektory

3.3 Paměť procesoru Blackfin 533

Procesor podporuje hierarchický paměťový model s různým výkonem a velikostními parametry, v závislosti na paměťovém místě uvnitř hierarchie. Úroveň 1 (L1) paměti jsou umístěné na čipu a jsou rychlejší než paměťové systémy úrovně 2 (L2). Úroveň 2 (L2) paměti jsou mimo čip a mají delší přístupové latence. Rychlejší L1 paměti, typicky malá zápisníková paměť nebo vyrovnávací paměti, se nacházejí uvnitř jádra.

Procesor má jednotný 4GB adresový rozsah (Obr. 9), který překlenuje kombinaci čipové a mimočipové paměti a mapu paměti I/O zdrojů. Z tohoto rozsahu, je část adresového prostoru věnována interním, čipovým zdrojům. Procesor zabere části z tohoto vnitřního paměťového prostoru:

- L1 statickou pamětí RAM (SRAM) (statický náhodný přístup paměti)
- soubory mapy pamětí registrů (MMRs)
- boot pamětí pouze pro čtení (ROM)



Obr. 9: Mapa paměti

Část vnitřní L1 SRAM paměti může být také konfigurována jako cache. Procesor také poskytuje podporu pro externí paměťový prostor, který zahrnuje asynchronní a synchronní paměťový prostor, DRAM (SDRAM).

Za zmínku stojí, že architektura nedefinuje oddělený I/O prostor. Všechny zdroje jsou mapované přímo skrz 32bitový adresový prostor. Paměť je Byte-adresovatelná. Tab. 2 ukazuje dostupné typy pamětí.

Tab. 2: Typy pamětí procesoru Blackfin 533

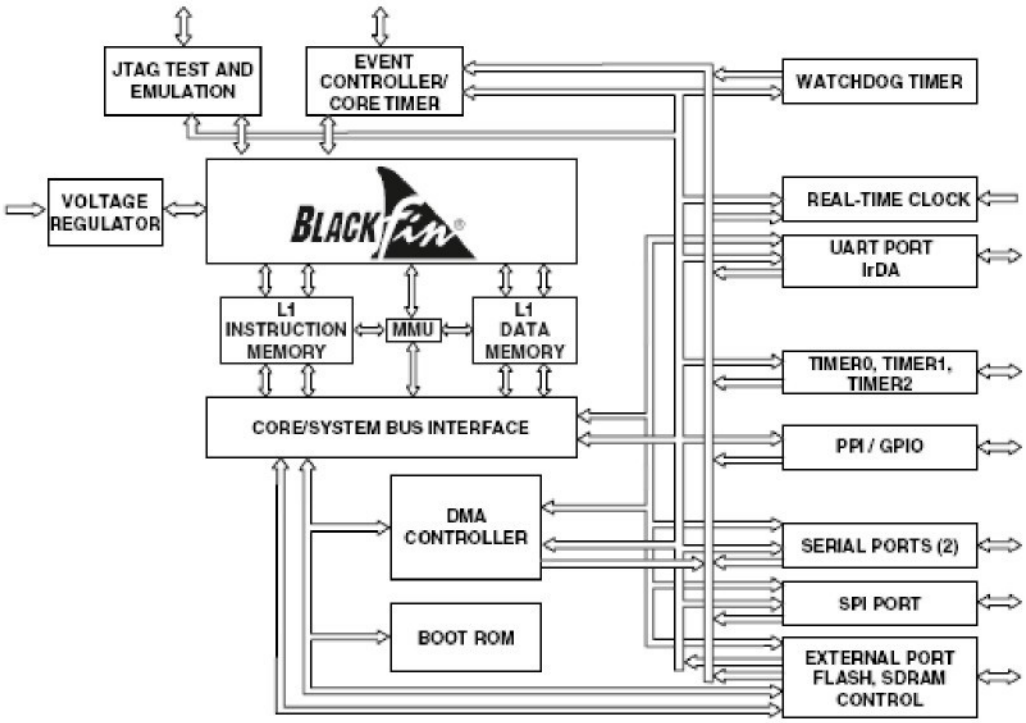
Typ paměti	Velikost
Instrukční SRAM/Cache	16 KByte
Instrukční SRAM	64 KByte
Data SRAM/ Cache	32 KByte
Data SRAM	32 KByte
Data Scratchpad SRAM	4 KByte
Celkem	148 KByte

Horní část vnitřního paměťového prostoru je přidělena jádru a systémovému MMR. Přístup k této oblasti je povolen pouze pokud je procesor v Supervisor nebo emulačním režimu. Nejnižší 1 KByte vnitřního paměťového prostoru je obsazený boot ROM. Po restartování procesoru se příslušný zaváděcí program vykoná z tohoto paměťového prostoru.

Uvnitř externí paměti jsou dostupné čtyři banky z asynchronního paměťového prostoru a jedna banka SDRAM paměti. Každá z asynchronních bank je 1 MByte velká a SDRAM banka stačí na 128 MBytů.

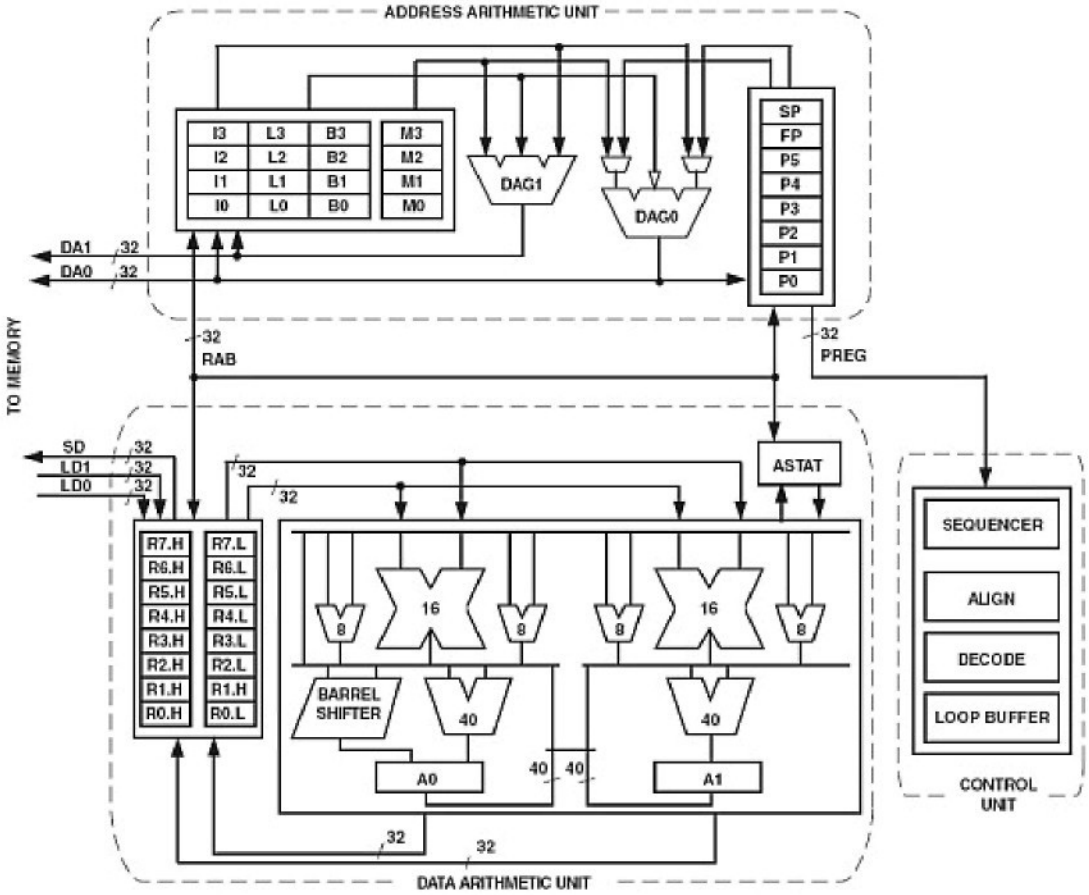
3.4 Vnitřní uspořádání procesoru

Na Obr. 10 je zobrazeno blokové schéma vnitřního uspořádání procesoru Blackfin, kde jsou vidět vzájemné vazby jednotlivých systémových zařízení.



Obr. 10: Blokové schéma procesoru

Schéma jádra procesoru je na Obr. 11. Ukazuje adresovací jednotku a vlastní výpočetní jednotku včetně vnitřního uspořádání jednotlivých prvků (násobičky, sčítačky, registry, propojovací sběrnice).

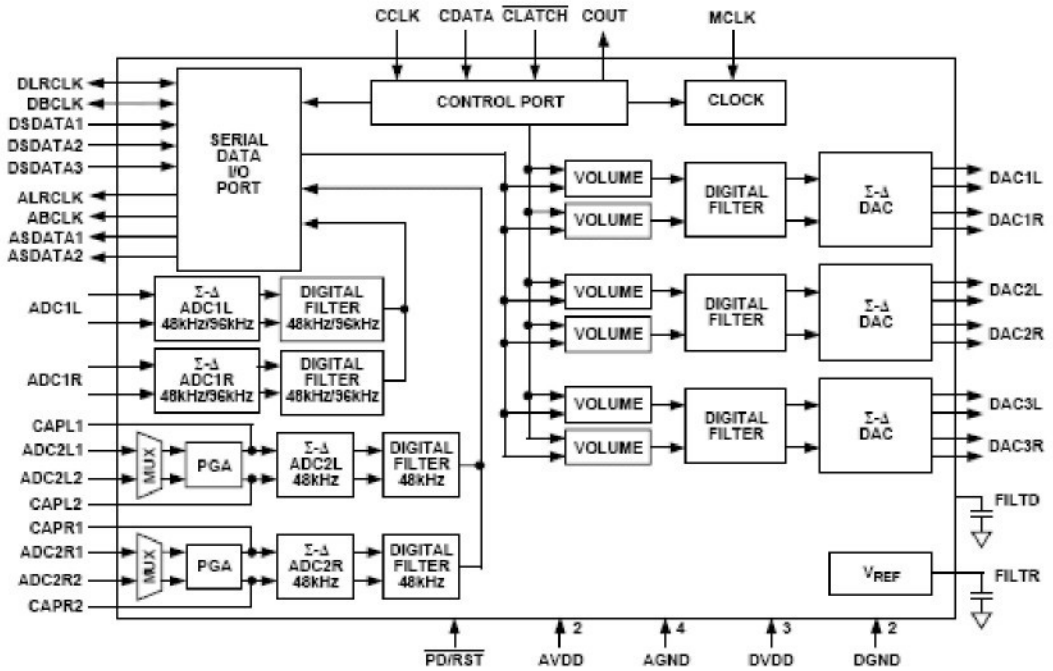


Obr. 11: Jádro procesoru

Jelikož záměrem práce není vyčerpávajícím způsobem popsat architekturu procesoru Blackfin a veškeré jeho možnosti, další konkrétnější popis již nebude uveden. Pro vlastní realizaci je v podstatě nepotřebný. Zájemci lze doporučit přibližně tisícistránkový manuál dodaný s procesorem [4].

3.5 Kodek AD1836

AD1836 je výkonný jednočipový audio kodér-dekodér poskytující tři stereo DA převodníky a dva stereo AD převodníky s použitím tzv. vícebitové delta-sigma modulace (detailní blokové schéma je na Obr. 12). Včleněný SPI port poskytuje možnosti pro řízení úrovně hlasitosti a nastavení různých parametrů. Kodek je dostupný v 52pinovém MQFP (nebo PQFP) pouzdře, určený pro povrchovou montáž.



Obr. 12: Funkční blokové schéma kodeku

Procesor je schopný posílat data do audio kodeku v tzv. TDM – časovém multiplexu nebo tzv. TWI – „dvoudrátovém režimu“. Časový multiplex (TDM) je princip přenosu více signálů jedním společným přenosovým médiem. Jednotlivé signály jsou odděleny tím, že se každý z nich vysílá (přenáší) pouze krátký pevně definovaný časový okamžik. Prakticky ve všech případech se používá rámcové struktury, která je rozdělena na stejně velké časové intervaly pro vysílání, pro každý signál jeden. Tento rámec se v čase neustále opakuje a tedy každý signál se přenáší stále se stejnou pravidelností.

TWI režim umožňuje kodeku pracovat se vzorkovací frekvencí 96 kHz, ale povoluje při tom pouze dva výstupní kanály. TDM režim lze provozovat s maximální vzorkovací frekvencí 48 kHz, ale umožňuje současné použití všech vstupů i výstupů.

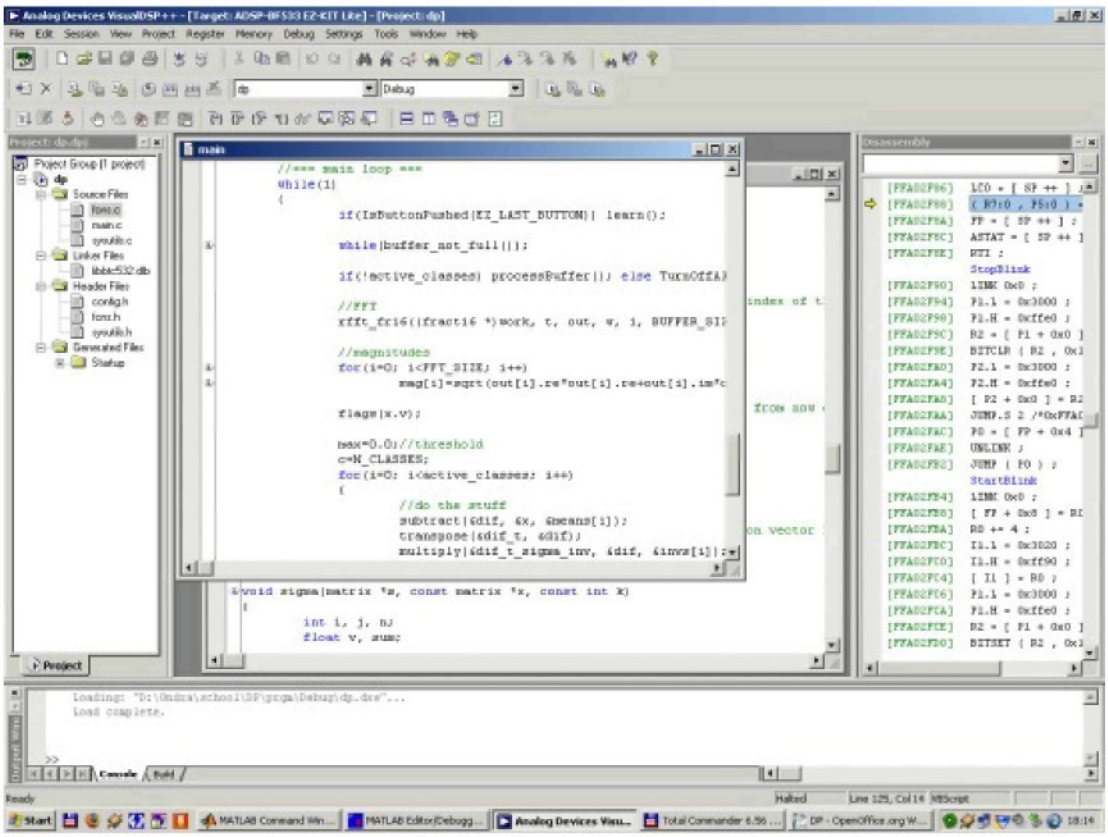
Kodek obsahuje čtyři AD kanály, konfigurované jako dva nezávislé stereo páry. Jeden pár je primární a má plně diferenciální vstupy, druhý pár může být prostřednictvím SPI nastaven do jednoho ze tří vstupních režimů. AD sekce může také pracovat na vzorkovací frekvenci 96 kHz, kdy jsou aktivní pouze primární vstupy. AD převodníky obsahují vlastní digitální decimační filtr s útlumem 120 dB a lineární fázovou odezvou operující v převzorkovacím poměru 128 (pro 4 kanály 48 kHz) nebo 64 (pro 2 kanály 96 kHz).

3.6 Vývojové prostředí

Deska ADSP-BF533 EZ-Kit Lite je navržena pro použití s vývojovým prostředím VisualDSP++ pro testování schopností procesoru Blackfin. Prostředí nabízí rozšířené možnosti pro vlastní vývoj a testování aplikačního kódu, jako:

- vytváření a sestavení aplikací napsaných v jazyce C, C++ a assembleru
- nahrávání, spouštění a krokování aplikace
- načítání a zápis datové a programové paměti
- čtení a zápis registrů
- sledování paměti

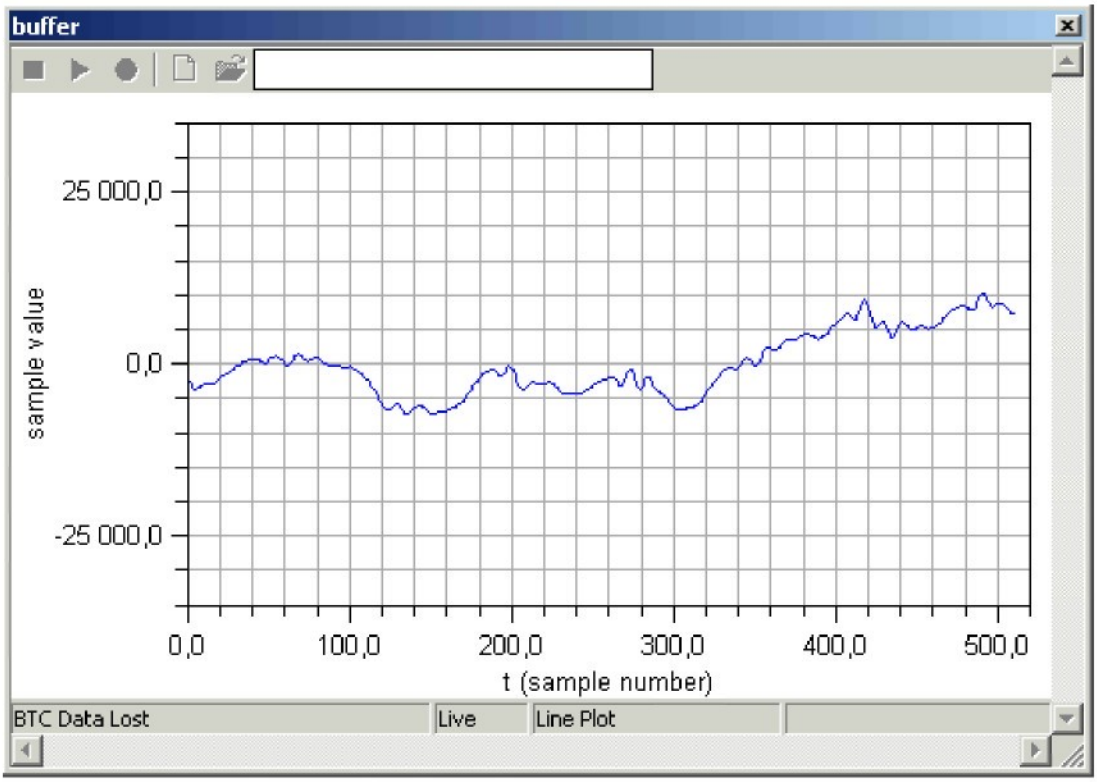
Spojení ADSP-BF533 s osobním počítačem je umožněno pomocí USB rozhraní nebo prostřednictvím přídavného JTAG emulátoru, který nabízí rychlejší vlastní komunikaci a některé další funkce. Vyžaduje však umístění zásuvné PCI karty do PC. Pohled na obrazovku vývojového prostředí je na Obr. 13.



Obr. 13: Náhled vývojového prostředí VisualDSP++ 4:5

Součástí prostředí je i sada příkladů (tzv. examples), které ukazují použití různých programových konstrukcí a funkcí. Nejjednodušší příklad ukazuje nastavení a použití indikačních LED a tlačítek na přípravku.

Jednou z mnoha možností, které VisualDSP++ nabízí, je i podpora tzv. BTC (Background Telemetry Channels – „kanály telemetrie na pozadí“). Pomocí knihovny funkcí, která se přidá do projektu (v případě toho přípravku jde o knihovnu *libbtc532.dlb*) a jednoduchým definováním kanálů ve vlastním zdrojovém kódu projektu, lze za běhu aplikace sledovat právě zpracovávané hodnoty. Prostředí může hodnoty například vykreslovat do grafu, který se automaticky aktualizuje. Náhled takového okna s grafem je na Obr. 14.



Obr. 14: Graf aktuálně zpracovávaných hodnot

Kromě zobrazení v podobě grafu lze paměť sledovat také jako výpis hodnot, s možností exportu do souboru a pro použití v aplikacích zpracovávajících video signál umožňuje prostředí zobrazit obsah paměti i jako obrázek.

4 REALIZACE

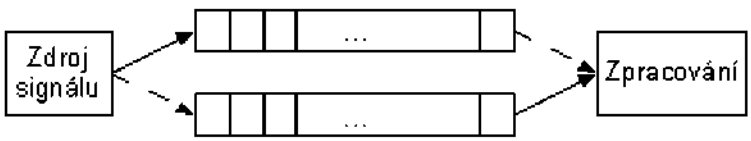
4.1 Záznam zvuku

Vstupní signál je odebírán z primárního stereo vstupu, přičemž je zároveň v nezměněné podobě odesílán na první výstup, aby bylo možno přímo ověřit funkčnost zdroje signálu. Pro demonstrační úlohu je postačující pouze jeden kanál stereo signálu, byl použit levý kanál (horní bílý konektor). Signál je vzorkován frekvencí 48 kHz s automatickým odfiltrováním vyšších frekvencí (více než 24 kHz). Výstupní hodnoty z kodeku jsou 24bitové se znaménkem.

Jako základ zdrojového kódu zde posloužil příklad *Audio Codec Talkthrough – TDM (C)* dodaný spolu s vývojovým prostředím VisualDSP++. Jeho funkce je prosté snímání vstupního signálu, který je po digitalizaci opět převeden do analogové podoby a posílán zpět na výstup. Vzorky jsou přístupné uživateli v odpovídajících proměnných. Tento příklad obsahuje nezbytné inicializace vlastního kodeku, konfigurace sériového rozhraní, nastavení DMA a obsluhy přerušení.

4.2 Zpracování signálu

Jednotlivé vzorky jsou postupně řazeny do vyrovnávacího bufferu, který v této konkrétní realizaci uchovává 512 vzorků, dokud nejsou připraveny pro další zpracování. Buffer je dvojitý (tzv. double buffer), čímž je zajištěno, že při práci na jednom úseku se paralelně plní druhá polovina (jednoduché schéma je na Obr. 15). Po dokončení výpočtů si buffery vymění role a výpočet probíhá na nově zaznamenaných datech. Při dané vzorkovací frekvenci a počtu vzorků uchová jeden buffer přibližně 10,7 ms signálu.

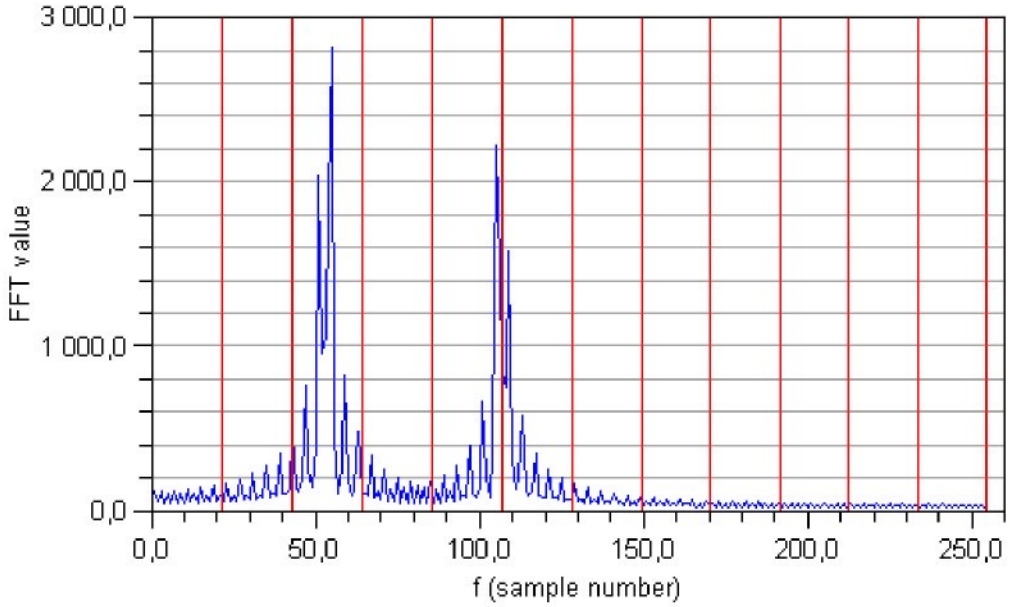


Obr. 15: Dvojitý buffer

Jelikož FFT algoritmus, který je na vzorky dále aplikován je optimalizován pro 16bitové hodnoty jsou vzorky do bufferu ukládány přepočtené na 16bitovou přesnost, která je pro tyto účely dostatečná. Jako vlastní FFT funkce byla použita funkce *rfft_fr16* z knihovny obsažené ve vývojovém prostředí, která je navržena pro reálné vstupní hodnoty. Komplexní výstup je dále přepočten na absolutní hodnoty a takto vzniklé spektrum je poté použito k výpočtu příznaků.

4.3 Výpočet příznaků

Spektrum získané v předešlém kroku je podle zvoleného počtu příznaků rovnoměrně rozděleno na stejný počet pásem a jednotlivé příznaky jsou vypočteny jako součet hodnot v konkrétním pásmu. Rozdělení spektra je naznačeno na Obr. 16.



Obr. 16: Rozdělení spektra na pásma

Lineární rozdělení spektra samozřejmě není jedinou možností, jak příznaky určovat. Lze například použít logaritmické rozdělení, nebo různé vlastní dělení. Volba samozřejmě závisí na plánovaném použití rozpoznávače a do jisté míry také ovlivňuje jeho vlastnosti. V uvedeném příkladu na Obr. 16 je například vidět, že druhý extrém (zde šlo o 10 kHz) se nachází na rozhraní dvou sousedních pásem. Při výpočtu tedy bude částečně zahrnut do obou. Specifická harmonická frekvence se zde rozprostře do dvou příznaků jejichž hodnoty budou nižší než by byla hodnota příznaku, do jehož pásma by extrém spadal úplně (jako v případě prvního extrému na obrázku). Tento fakt lze ovlivnit pouze pokud jsou předem známy charakteristiky rozpoznávaného signálu. Vliv má pochopitelně i počet příznaků. Při vyšším počtu příznaků lze teoreticky dosáhnout vyšší přesnosti, avšak spolu s tím narůstá i výpočetní náročnost algoritmu, na kterou musí též být brán ohled.

Realizace v této práci používá dvanáct příznaků – spektrum je rozděleno na 2kHz pásma (nejvyšší frekvence je 24 kHz), přičemž při výpočtu prvního příznaku je z pásma 0 Hz – 2 kHz vypuštěna první hodnota, která představuje stejnosměrnou složku signálu, která by pro účely frekvenčního vyhodnocení pouze zkreslovala.

4.4 Analýza

Další zpracování získaného příznakového vektoru závisí na tom, v jakém režimu se přípravek nachází – fáze učení nebo fáze rozpoznávání.

Pro statistickou analýzu byla vytvořena podpora základních maticových operací:

- inicializace matice, dle zadaných rozměrů
- transpozice matice
- maticové násobení a odčítání
- inverze matice
- výpočet determinantu matice
- výpočet kovarianční matice

Algoritmus výpočtu inverze matice byl realizován podle [1], je tzv. in-place a zahrnuje dva kroky. Na vstupní matici se nejprve provede LU rozklad (dekompozice), který rozdělí (čtvercovou) matici na horní (upper) a dolní (lower) trojúhelníkovou matici, jejichž součin je roven původní matici (obecný příklad pro matici 4×4 ukazuje Rovnice 5).

$$\underbrace{\begin{bmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{bmatrix}}_{L} \cdot \underbrace{\begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix}}_{U} = \underbrace{\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}}_{A} \qquad \text{(Rovnice 5)}$$

Takovýto rozklad je obecně nejednoznačný. V praxi se algoritmus rozkladu běžně realizuje tak, aby diagonální prvky dolní trojúhelníkové matice L byly rovny 1. To zároveň přináší i úsporu paměti, jelikož zmíněné diagonální prvky nemusíme ukládat a obě matice (L a U) lze uložit společně do jedné. V tomto řešení je výstup (rozklad) zapisován přímo do původní matice A během výpočtu. Výsledek rozkladu matice 4×4 tedy dostaneme ve tvaru $\begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ \alpha_{21} & \beta_{22} & \beta_{23} & \beta_{24} \\ \alpha_{31} & \alpha_{32} & \beta_{33} & \beta_{34} \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \beta_{44} \end{bmatrix}$.

Ve druhém kroku výpočtu inverze je poté zpětnou substitucí po řádcích vypočtena vlastní inverze. LU dekomponovaná matice se též používá pro výpočet determinantu, který je v tomto případě pouhým součinem prvků diagonály. Matematický aparát a vlastní realizace je podrobně objasněna v [1] v kapitole 2.3 na stranách 43–49.

4.4.1 Učení

Ve fázi učení jsou příznakové vektory ukládány pro statistickou analýzu. Počet uchovávaných vektorů (velikost trénovací množiny) ovlivňuje přesnost rozpoznávání. Pro tuto úlohu bylo zvoleno pevně 100 trénovacích prvků, což při dané vzorkovací frekvenci a velikosti bufferu odpovídá přibližně jedné sekundě záznamu signálu.

Po zaznamenání všech prvků (vektorů $\vec{x}_1 - \vec{x}_{100}$) trénovací množiny pro konkrétní třídu je provedena statistická analýza – vypočtou se průměry pro jednotlivé příznaky ($\vec{\bar{x}}$) a poté se podle Rovnice 4 vypočte úplná kovarianční matice Σ, která zachycuje vztahy mezi různými příznaky. Pro účely rozpoznávání se poté vypočítá inverze a determinant této matice, který figuruje v pravděpodobnostních výpočtech (Rovnice 3) a spolu s vektorem průměrů jsou tyto hodnoty uloženy pro konkrétní třídu do „znalostní databáze".

4.4.2 Rozpoznávání

Při rozpoznávání se příznakový vektor $\vec{x}$, reprezentující daný úsek signálu, dosazuje do maticových operací spolu s naučenými hodnotami (Rovnice 3). Pro každou třídu se určí pravděpodobnost, že do ní tento vektor patří a třída s největší hodnotou je prohlášena za výsledek. Pokud není nalezeno maximum (všechny třídy mají nulovou pravděpodobnost) výsledkem není žádná třída (tzv. odmítnutí). Mají-li dvě třídy stejnou pravděpodobnost, je za výsledek označena třída s nejnižším indexem.

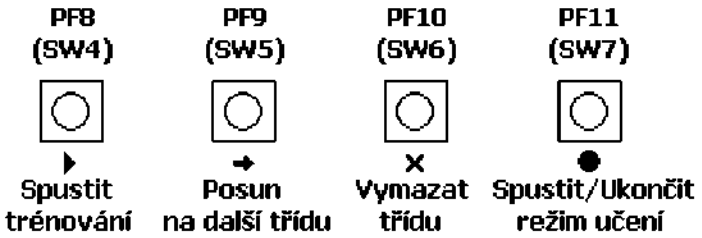
Rozložení apriorních pravděpodobností tříd $P(T_r)$ aplikované v Rovnici 1 bylo pro tyto účely zvoleno jako rovnoměrné – všechny třídy mají stejnou pravděpodobnost výskytu.

4.5 Výstup

Jako výstup byly pro jednoduchost zvoleny indikační LED na přípravku. Každá LED zastupuje jednu třídu a její svit během rozpoznávání signalizuje vyhodnocení příslušnosti daného úseku signálu do odpovídající třídy. Při učení jednotlivé blikající diody signalizují učení konkrétní třídy. Pokud není definována žádná třída, neprobíhá rozpoznávání, přípravek pouze vypočítává celkovou energii úseku a reprezentuje ji rozsvícením příslušného počtu diod.

4.6 Ovládání

Pro úlohu bylo vytvořeno jednoduché ovládání pomocí dostupných programovatelných tlačítek. Na Obr. 17 je znázorněno rozmístění a identifikace tlačítek, spolu s funkcemi, které jim byly přiřazeny.



Obr. 17: Popis ovládacích tlačítek

Jak již bylo zmíněno, úloha pracuje ve dvou základních režimech – učení a rozpoznávání. Po zapnutí je výchozí rozpoznávání. Nejsou však definovány žádné třídy a přípravek pouze indikuje energii signálu. Během rozpoznávání je aktivní pouze tlačítko pro spuštění učení (SW7), po jeho stisku přechází program do režimu učení.

Tento stav je indikován blikáním první uživatelské LED (LED4), která zároveň indikuje učení (definování) první třídy, ostatní LED jsou zhasnuté. V tuto chvíli jsou již aktivní všechna tlačítka, jejich funkce jsou následující:

Spustit trénování – Spouští proces trénování, který zaznamená trénovací množinu, provede statistickou analýzu, uloží výsledky do „znalostní databáze" a nastaví příslušnou třídu jako aktivní (pro použití při rozpoznávání). Před stiskem tohoto tlačítka je nutné již přivádět požadovaný signál zastupující trénovanou třídu. Po stisku se indikační dioda třídy rozsvítí trvale a zůstane svítit dokud není definice příslušné třídy odstraněna funkcí *Vymazat třídu*. Vlastní učení trvá přibližně jednu sekundu (závisí na počtu trénovacích vzorků). Dokončení je signalizováno blikáním následující LED v řadě (po natrénování poslední třídy se cykluje opět od první).

Posun na další třídu – Posouvá vybranou třídu, indikovanou blikáním příslušné diody, na následující v řadě (na konci se opět cykluje od začátku). Ovládání umožňuje při definování tříd některé třídy vynechat (– neaktivní třídy, nepoužijí se při rozpoznávání). Pokud se během posunu narazí na třídu, která je již definována, její indikátor bliká s vyšší frekvencí než u třídy, která dosud nebyla definována (nebo byla vymazána). Indikátory ostatních tříd (vybraná je vždy pouze jedna), svítí pokud je příslušná třída aktivní.

Vymazat třídu – Nastaví příslušnou vybranou třídu jako neaktivní (tedy bez definice). Třída zůstane vybraná, přičemž indikátor bliká opět se základní frekvencí.

Ukončit režim učení – Ukončuje učení a přepíná přípravek do rozpoznávacího režimu. Pokud nebyly definovány žádné aktivní třídy je realizována pouze indikace energie signálu, jinak probíhá rozpoznávání a pokud je daný signál klasifikován je rozsvícena odpovídající dioda.

4.7 Možnosti rozšíření

V demonstrační úloze byl pro názornost zvolen maximální počet tříd 6 (stejně jako je množství indikačních LED na přípravku) a třídy jsou reprezentovány rozsvícením jedné diody z šesti. Při realizaci většího počtu tříd by se daly třídy signalizovat například binární kombinací (zde tedy až 64 třídy).

Pro účely dalšího zpracování by se dal výstup posílat do jiných zařízení například pomocí standardního sériového portu, což může jednoduchou indikaci rozšířit například o připojení řídícího systému, který by poté mohl libovolně naložit s výsledkem rozpoznávání a realizovat odpovídající odezvu. Příkladem by mohl být systém sledování elektrického pohonu zařízení (například akcelerometrem), který by po zjištění nežádoucího stavu odeslal varování obsluze.

Nezbytnou úpravou pro případ reálného nasazení přípravku v praxi by bylo ukládání naučené „znalostní databáze“ do vlastní flash paměti, aby tato data zůstala zachována i v případě vypnutí (například při výpadku napájení).

Další možností rozšíření by mohla být funkce „přitrénování“. Jde o postup trénování třídy, kdy jsou zachovány původní charakteristiky (průměry, matice kovariancí), které jsou spolu s nově zanalyzovanými daty přepočteny, tak aby postihovaly novou rozšířenou trénovací množinu (původní trénovací množina není uchovávána). Například natrénujeme 100 prvků a poté se rozhodneme, že chceme ještě přidat dalších 100 prvků. Tento postup vyžaduje určitá rozšíření matematických vztahů (např. Rovnice 4) – pro účely průměrování je potřeba uchovávat počet natrénovaných prvků, matici kovariancí je nutno přepočíst se zakomponováním nových hodnot, apod.

5 DEMONSTRAČNÍ ÚLOHA

Pro demonstrační úlohu byly programem Matlab generovány čtyři signály složené z různých harmonických frekvencí a náhodného šumu:

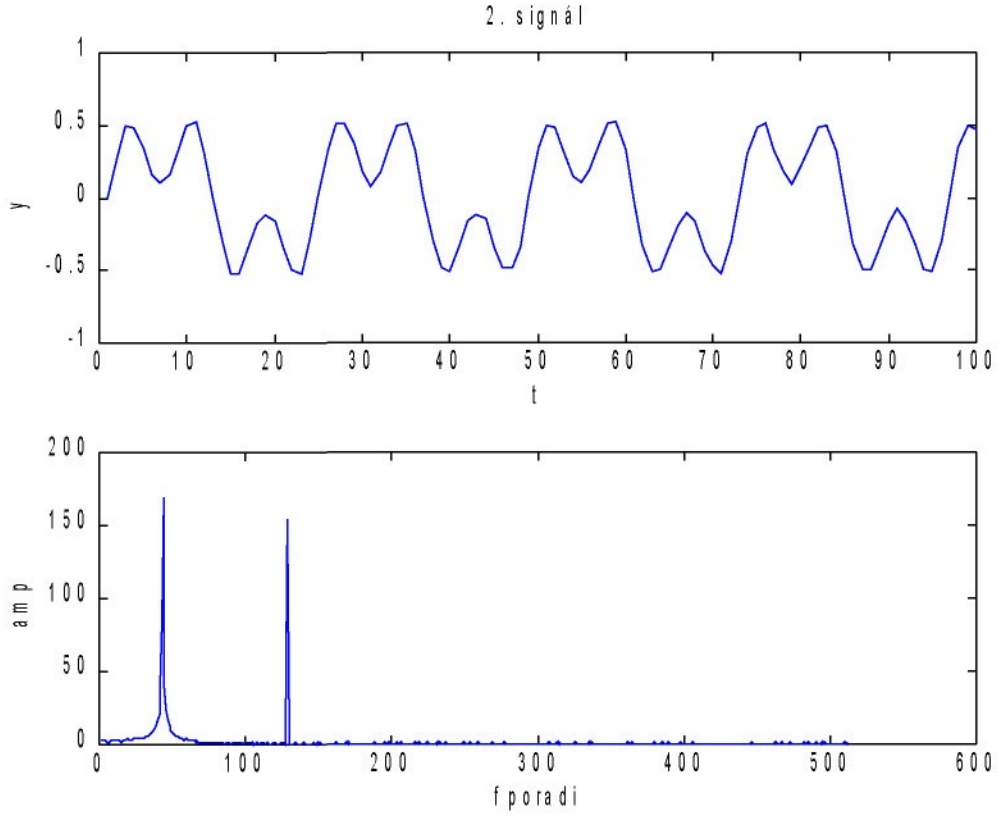
$$s_1 = 0{,}3 \cdot \sin(2 \cdot \pi \cdot 7000 \cdot t) + 0{,}05 \cdot \nu(t)$$

$$s_2 = 0{,}4 \cdot \sin(2 \cdot \pi \cdot 4000 \cdot t) + 0{,}3 \cdot \sin(2 \cdot \pi \cdot 12000 \cdot t) + 0{,}05 \cdot \nu(t)$$

$$s_3 = 0{,}3 \cdot \sin(2 \cdot \pi \cdot 7000 \cdot t) + 0{,}3 \cdot \sin(2 \cdot \pi \cdot 12000 \cdot t) + 0{,}05 \cdot \nu(t)$$

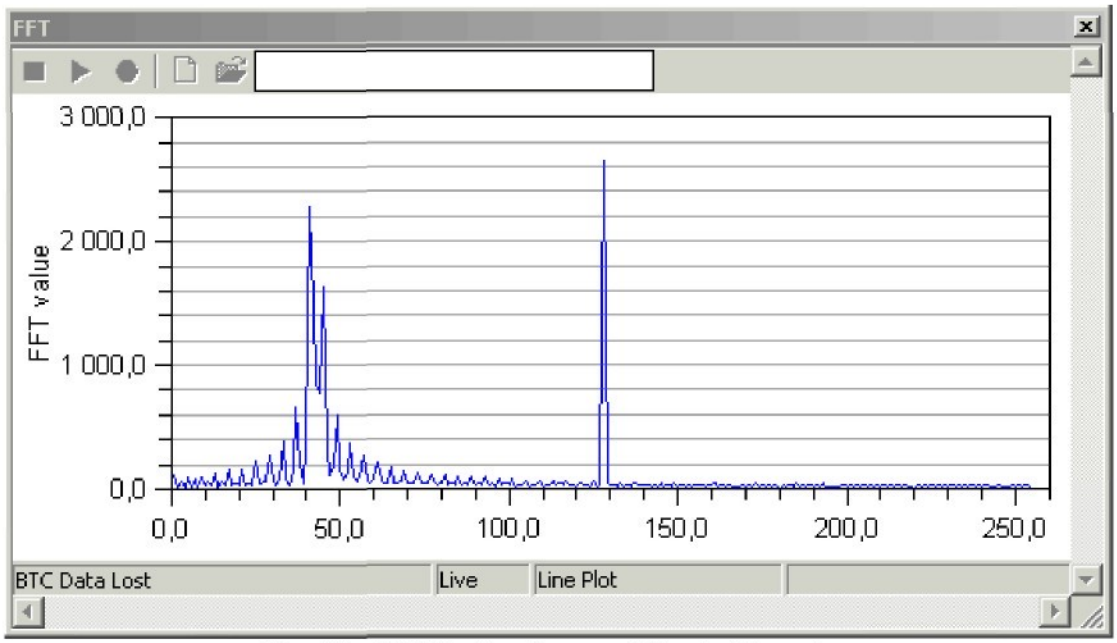
$$s_4 = 0{,}3 \cdot \sin(2 \cdot \pi \cdot 4000 \cdot t) + 0{,}4 \cdot \sin(2 \cdot \pi \cdot 6000 \cdot t) + 0{,}5 \cdot \sin(2 \cdot \pi \cdot 12000 \cdot t) + 0{,}05 \cdot \nu(t),$$

kde $\nu(t)$ značí bílý šum s nulovou střední hodnotou a jednotkovou amplitudou. Ukázka časového průběhu a FFT jednoho ze signálů je na Obr. 18 (grafy jsou též z programu Matlab).

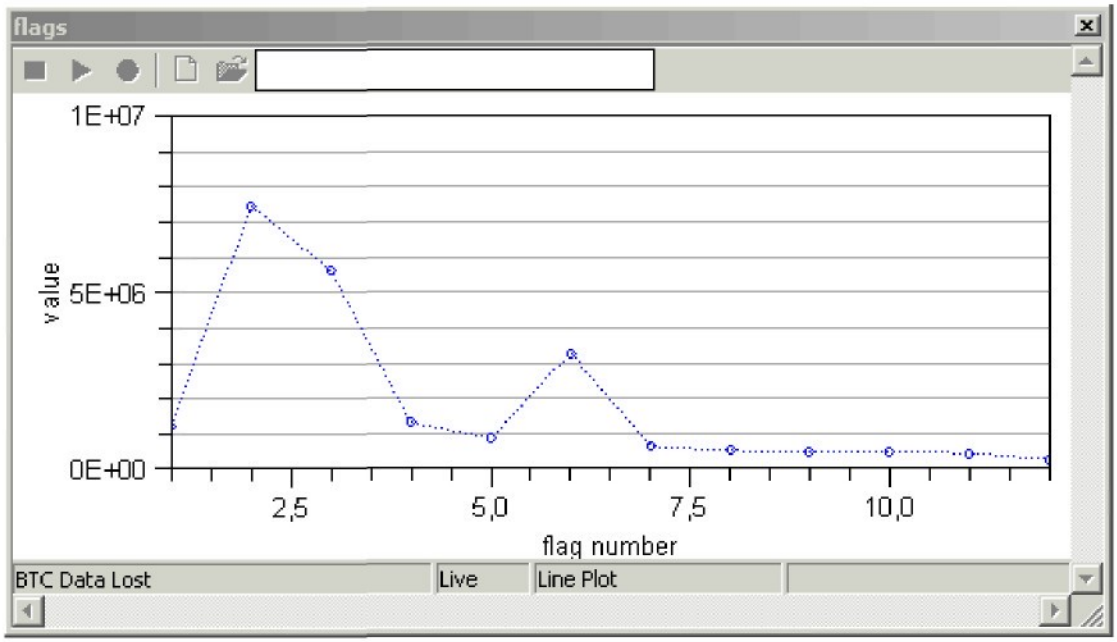


Obr. 18: Časová a frekvenční charakteristika 2. signálu

Všechny signály byly propojením vstupu na přípravku s výstupem z PC postupně natrénovány a poté v režimu učení posloužily k otestování rozpoznávače. Během rozpoznávání byly pomocí BTC monitorovány snímané průběhy. Obr. 19 ukazuje spektrum 2. signálu tak, jak ho vypočítal přípravek a Obr. 20 znázorňuje již hodnoty příznaků z tohoto spektra vypočtené.

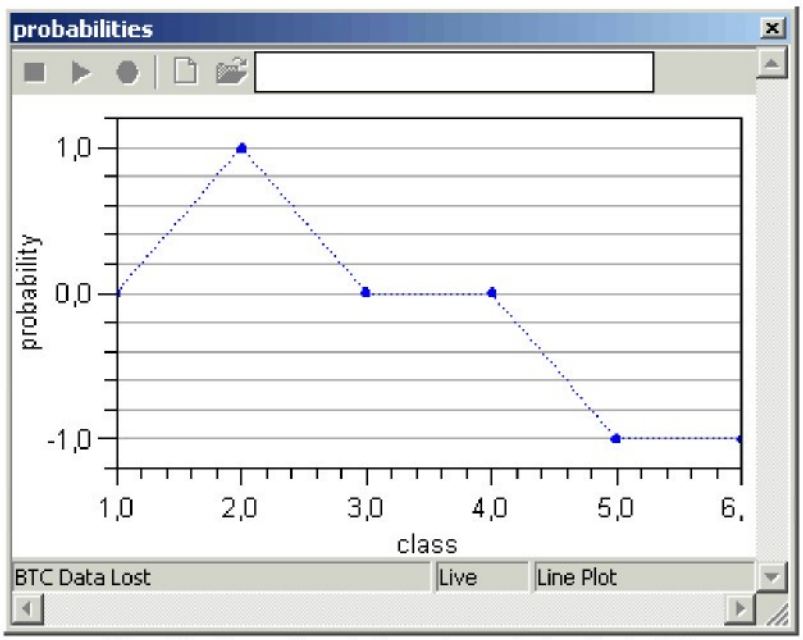


Obr. 19: FFT 2. signálu vypočítané přípravkem



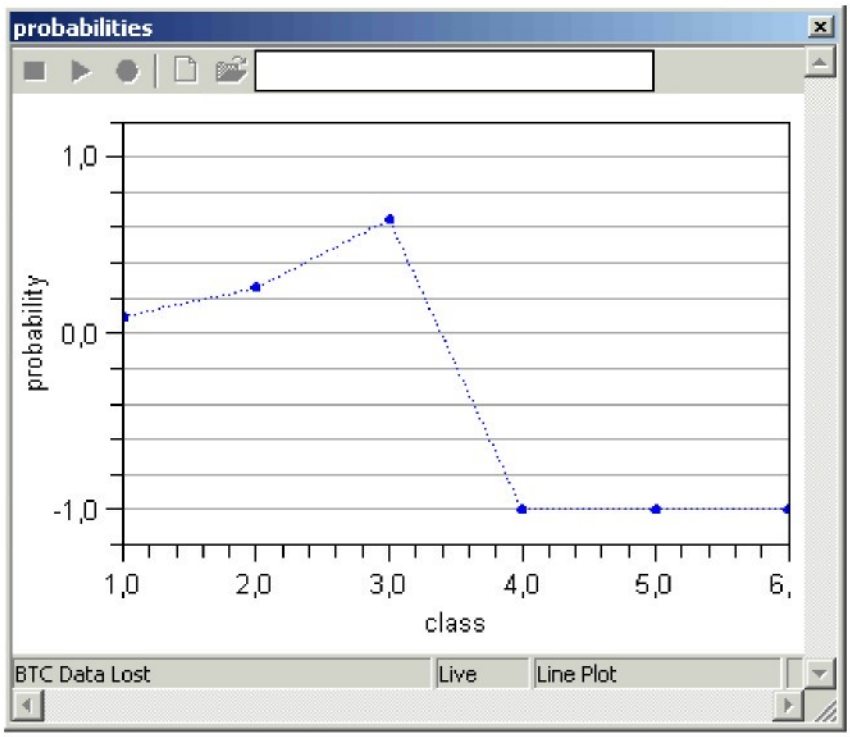
Obr. 20: Hodnoty příznaků 2. signálu

Výsledek rozpoznávání lze sledovat (kromě vlastní indikace rozsvícením diody) opět pomocí BTC. Obr. 21 ukazuje jednoznačnou klasifikaci signálu do 2. třídy. Hodnoty -1 u posledních dvou tříd v grafu značí, že třídy nejsou aktivní.



Obr. 21: Pravděpodobnosti pro jednotlivé třídy

Jiný příklad (na Obr. 22) ukazuje situaci, kdy jsou si tři natrénované třídy více podobné. Hodnoty pravděpodobností tedy obecně nejsou tak jednoznačné jako v předchozím případě, proto se provádí výběr maximální pravděpodobnosti.



Obr. 22: Příklad pravděpodobností pro podobné třídy

ZÁVĚR

V práci byl nejprve řešen matematický aparát, sloužící jako prostředek příznakového rozpoznávání (podpora maticových výpočtů, LU rozklad, inverze matice) a následně statistické funkce (výpočet kovarianční matice, aposteriorní pravděpodobnosti).

Demonstrační úloha ověřila funkčnost řešení, které poskytuje základnu pro další možný vývoj na této nebo podobné platformě. Jelikož vlastní aplikace je vytvořena v programovacím jazyku C, kód je s určitými úpravami použitelný i pro jiné typy signálových procesorů.

Výhodou toho řešení oproti použití klasického PC je kompaktnost, stabilita a především rychlost, kterou by v tomto případě bylo možné dále zvýšit optimalizací statistických výpočtů pro práci s pevnou řádovou čárkou nebo použitím procesoru, pracujícím s řádovou čárkou plovoucí.

Co se týká praktického využití, je možné takovéto řešení použít například pro on-line diagnostiku technických signálů pro určení stavu zařízení (např. elektrického pohonu), dále též v oblasti zdravotnictví pro analyzování biologických signálů jako EKG (např. detekce srdeční arytmie) či EEG (např. rozpoznání epileptického záchvatu) nebo dokonce pro detekci nálady včelstva na základě akustické analýzy.

REFERENCE

[1] Press, William H. - Teukolsky Saul A. - Vetterling, William T. - Flannery, Brian P. *Numerical Recipes in C: The Art of Scientific Computing*. Second Edition. Cambridge: Cambridge University Press, Reprinted 2002. 994 p. [online] URL: <http://www.nrbook.com/a/bookcpdf.php>. ISBN 0-521-43108-5 Citováno: listopad 2006

[2] Krupka, Petr. *Metody počítačového zpracování obrazu užitím signálového procesoru*. Liberec, 2005. 66 s. Diplomová práce. Technická univerzita v Liberci.

[3] Analog Devices, Inc. *ADSP-BF533 EZ-KIT Lite Evaluation System Manual*. Revision 3.0, June 2006. 89 p. [online] URL: <http://www.analog.com/UploadedFiles/Associated_Docs/68863602ADSP_BF533_EZ_KIT_Lite_Manual_Rev_3.0.pdf>. Citováno: 12.12.2006

[4] Analog Devices, Inc. *ADSP-BF533 Blackfin Processor Hardware Reference*. Revision 1.0, December 2003. 936 p.

[5] http://en.wikipedia.org

[6] http://www.analog.com

[7] http://www.blackfin.org

[8] http://www.dsprelated.com

PŘÍLOHA – Zdrojové kódy

Hlavní program (*main.c*)

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sysreg.h>
#include <ccblkfn.h>
#include <filter.h>
#include <complex.h>
#include <fract.h>
#include "sysutils.h"
#include "fcns.h"
#include "config.h"

#ifdef _BTC_
      #include <btc.h>
#endif

//    Disconnect RSCLK0 and TSCLK0 (Turn SW9 pin 6 OFF)
//    Disconnect RFS0 and TFS0 (Turn SW9 pin 5 OFF)

//knowledge base
int n_active_classes=0, active_classes[N_CLASSES];
float dets_sqrts[N_CLASSES];
matrix means[N_CLASSES], invs[N_CLASSES];

//double buffer
int buffer_count=0, buffer_num=0;
volatile fract16 *buffer;//active buffer address (for sample storing)
volatile fract16 *work;//working buffer address (for processing)
volatile fract16 *buffers[2];

//FFT vars
complex_fract16 t[BUFFER_SIZE];//temp working
complex_fract16 out[BUFFER_SIZE];//FFT output
complex_fract16 w[BUFFER_SIZE];//twiddle sequence
fract16 mag[FFT_SIZE];//magnitudes

//DMA buffer
volatile int iRxBuffer[8], iTxBuffer[8];

//BTC definitions
#ifdef _BTC_
      fract16 BTC_CHAN0[BUFFER_SIZE+8];
      fract16 BTC_CHAN1[FFT_SIZE+8];
      float BTC_CHAN2[N_FLAGS+8];
      float BTC_CHAN3[N_CLASSES+8];

      BTC_MAP_BEGIN
      BTC_MAP_ENTRY("BUFFER", (long)&BTC_CHAN0, sizeof(BTC_CHAN0))
      BTC_MAP_ENTRY("FFT", (long)&BTC_CHAN1, sizeof(BTC_CHAN1))
      BTC_MAP_ENTRY("FLAGS", (long)&BTC_CHAN2, sizeof(BTC_CHAN2))
      BTC_MAP_ENTRY("CLASSES", (long)&BTC_CHAN3, sizeof(BTC_CHAN3))
      BTC_MAP_END
#endif
```

```
void main(void)
{
	int i, c;
	float max, p, probs[N_CLASSES], aposterior[N_CLASSES];

	matrix x=matrix_create(1, N_FLAGS);//actual flags
	matrix dif=matrix_create(1, N_FLAGS);//x-means
	matrix dif_t=matrix_create(N_FLAGS, 1);//x-means transposed
	matrix dif_t_sigma_inv=matrix_create(1, N_FLAGS);//dif*sigma_inv
	matrix scalar=matrix_create(1, 1);//result

	//buffers init
	buffer=buffers[0]=
		(fract16 *) malloc((size_t) BUFFER_SIZE*sizeof(fract16));
	work=buffers[1]=
		(fract16 *) malloc((size_t) BUFFER_SIZE*sizeof(fract16));

	sysreg_write(reg_SYSCFG, 0x32);

	//FFT twiddle
	twidfft_fr16(w, BUFFER_SIZE);

	//knowledge base matrices init
	for(i=0; i<N_CLASSES; i++)
	{
		means[i]=matrix_create(1, N_FLAGS);
		invs[i]=matrix_create(N_FLAGS, N_FLAGS);
	}

	//init
	#ifdef _BTC_
		btc_init();
	#endif

	Init_EBIU();
	Init_Flash();
	InitButtons();

	//initialize the timer
	initTimer();
	register_handler(ik_timer, timerISR);

	//codec, interrupt, DMA
	Init1836();
	Init_Sport0();
	Init_DMA();
	Init_Sport_Interrupts();
	Enable_DMA_Sport0();
```

```
//main loop
while(1)
{
	if(IsButtonPushed(BUTTON_LEARN)) learn();

	while(buffer_not_full());//wait for full buffer

	if(!n_active_classes)
	{//no class defined, no recognition, just energy indicator
		processBuffer();
		continue;
	}
	else
		TurnOffAllLEDs();

	//FFT
	rfft_fr16((fract16 *)work, t, out, w, 1, BUFFER_SIZE, 0,0);

	//magnitudes
	for(i=0; i<FFT_SIZE; i++)
		mag[i]=sqrt(out[i].re*out[i].re+out[i].im*out[i].im);

	flags(x.v);

	p=0;
	//conditional probabilities
	for(i=0; i<N_CLASSES; i++)
	{
		if(active_classes[i])
		{//do the stuff
			subtract(&dif, &x, &means[i]);
			transpose(&dif_t, &dif);
			multiply(&dif_t_sigma_inv, &dif, &invs[i]);
			multiply(&scalar, &dif_t_sigma_inv, &dif_t);
			probs[i]=(expf(-0.5*scalar.v[0]))/
					dets_sqrts[i];//probability
			//absolut probability
			p+=probs[i];
		}
	}

	max=0.0;//threshold
	c=N_CLASSES;
	for(i=0; i<N_CLASSES; i++)
	{
		if(active_classes[i])
		{
			if(p>0.0)//to prevent division by zero
			{
				aposterior[i]=probs[i]/p;
				if(aposterior[i]>max)
				{
					max=aposterior[i];
					c=i;
				}
			}
			else
				aposterior[i]=0.0;
		}
```

```
                else
                        aposterior[i]=-1;//indicates non active classes
            }
            //if class established, indicate
            if(c<N_CLASSES) TurnOnLED(c+EZ_FIRST_LED);

            //BTC write output
            #ifdef _BTC_
                btc_write_array(0, (unsigned int *)work,
                                    sizeof(fract16)*BUFFER_SIZE);
                btc_write_array(1, (unsigned int *)mag, sizeof(mag));
                btc_write_array(2, (unsigned int *)x.v,
                                    sizeof(float)*N_FLAGS);
                btc_write_array(3, (unsigned int *)aposterior,
                                    sizeof(float)*N_CLASSES);
                btc_poll();
            #endif
        }
}
```

Základní konfigurace (*config.c*)

```
#ifndef CONFIG_H
#define CONFIG_H

//Background Telemetry Channels active
//(if disabled also exclude libbtc532.dlb library from build)
#define _BTC_


//defines
#define BUFFER_SIZE     512
#define N_CLASSES       6
#define N_SAMPLES       100
#define N_FLAGS         12

#define FFT_SIZE  BUFFER_SIZE/2

//control buttons
#define BUTTON_TRAIN    4
#define BUTTON_NEXT     5
#define BUTTON_DELETE   6
#define BUTTON_LEARN    7

#endif
```

Funkce (*fcns.c*)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <math_const.h>
#include <complex.h>
#include <filter.h>
#include "fcns.h"
#include "sysutils.h"
#include "config.h"

#ifdef _BTC_
      #include <btc.h>
#endif

//double buffer
extern int buffer_count;
extern int buffer_num;
extern volatile fract16 *buffer;
extern volatile fract16 *work;
extern volatile fract16 *buffers[];

//knowledge base
extern int n_active_classes;
extern int active_classes[];
extern float dets_sqrts[];
extern matrix means[], invs[];

//FFT vars
extern complex_fract16 t[];//temp working
extern complex_fract16 out[];//FFT output
extern complex_fract16 w[];//twiddle sequence
extern fract16 mag[FFT_SIZE];//magnitudes

matrix matrix_create(int r, int c)
{//initializes matrix structure according to given size
      long i;
      matrix a;

      a.r=r; a.c=c;

      //allocate pointers to rows
      a.m=(float **) malloc((size_t)(r*sizeof(float*)));

      //allocate rows (as one block - array) and set pointers to them
      a.v=a.m[0]=(float *) malloc((size_t)(r*c*sizeof(float)));
      for (i=1; i<r; i++) a.m[i]=a.m[i-1]+c;

      return a;
}

void matrix_free(matrix *a)
{//deallocates memory occupied by matrix (arrays)
      free((*a).v);
      free((*a).m);
}
```

```
void multiply(matrix *x, const matrix *a, const matrix *b)
{//destination x should be initialized previously
      int i, j, k;
      float sum;

      for(j=0; j<(*x).r; j++)
            for(i=0; i<(*x).c; i++)
            {
                  sum=0.0;
                  for(k=0; k<(*a).c; k++)
                  {
                        sum+=((*a).m[j][k])*((*b).m[k][i]);
                  }
                  (*x).m[j][i]=sum;
            }
}


void subtract(matrix *x, const matrix *a, const matrix *b)
{//destination x should be initialized previously
      int i, j;

      j=((*a).c)*((*a).r);
      for(i=0; i<j; i++)
            (*x).v[i]=((*a).v[i])-((*b).v[i]);
}


void transpose(matrix *x, const matrix *a)
{//destination x should be initialized previously
      int i, j;

      if((*a).c>1 && (*a).r>1)
      {
            for(i=0; i<(*a).r; i++)
                  for(j=0; j<(*a).c; j++) (*x).m[i][j]=(*a).m[j][i];
      }
      else
      {
            memcpy((*x).v, (*a).v, sizeof(float)*((*a).r)*((*a).c));
      }
}


void inversion(const matrix *s, int *indx, int k)
{//input should be LU decomposed square matrix!!!
 //output will be in global array invs
      int i, j;
      float *col;
      col=(float *) malloc((size_t) ((*s).r)*sizeof(float));

      for(j=0; j<(*s).r; j++)
      {//find inverse by columns
            for(i=0; i<(*s).r; i++) col[i]=0.0;
            col[j]=1.0;
            lubksb(s, indx, col);
            for(i=0; i<(*s).r; i++) invs[k].m[i][j]=col[i];
      }
}
```

```
void ludcmp(matrix *m, int *indx, float *d)
/*replaces given square matrix by the LU decomposition of a rowwise
permutation of itself; indx is an output vector that records the row
permutation efected by the partial pivoting; d is output as ±1
depending on whether the number of row interchanges was even or odd,
respectively; this routine is used in combination with lubksb to solve
linear equations or invert a matrix*/
{
      int n, i, imax, j, k;
      float **a, big, dum, sum, temp;
      matrix vv;//vv stores the implicit scaling of each row

      a=(*m).m;
      n=(*m).r;

      vv=matrix_create(n, 1);

      *d=1.0;//no row interchanges yet.
      for(i=0; i<n; i++)
      {//loop over rows to get the implicit scaling information
            big=0.0;
            for(j=0; j<n; j++) if((temp=fabs(a[i][j]))>big) big=temp;
            //no nonzero largest element
            vv.v[i]=1.0/big;//save the scaling
      }

      for(j=0; j<n; j++)
      {//this is the loop over columns of Crout's method
            for(i=0; i<j; i++)
            {
                  sum=a[i][j];
                  for (k=0; k<i; k++) sum-=a[i][k]*a[k][j];
                  a[i][j]=sum;
            }

            big=0.0;//init for the search for largest pivot element
            for(i=j; i<n; i++)
            {
                  sum=a[i][j];
                  for(k=0; k<j; k++) sum-=a[i][k]*a[k][j];
                  a[i][j]=sum;
                  if((dum=vv.v[i]*fabs(sum))>=big)
                  {
                        big=dum;
                        imax=i;
                  }
            }

            if(j!=imax)
            {//do we need to interchange rows?
                  for(k=0; k<n; k++)
                  {//yes, do so...
                        dum=a[imax][k];
                        a[imax][k]=a[j][k];
                        a[j][k]=dum;
                  }
                  *d=-(*d);//...and change the parity of d
                  vv.v[imax]=vv.v[j];//interchange the scale factor
            }
```

```
            indx[j]=imax;
            if(a[j][j]==0.0) a[j][j]=1.0e-20;
            //if the pivot element is zero the matrix is singular
            //(at least to the precision of the algorithm)
            //for some applications on singular matrices,
            // it is desirable to substitute TINY for zero
            if (j<n)
            {//now, finally, divide by the pivot element
                  dum=1.0/(a[j][j]);
                  for(i=j+1; i<n; i++) a[i][j]*=dum;
            }
      }
      matrix_free(&vv);
}


void lubksb(const matrix *m, int *indx, float b[])
//solves the set of n linear equations A·X = B; input is a LU
decomposition of original matrix, determined by the routine ludcmp;
indx is input as the permutation vector returned by ludcmp; b is input
as the right-hand side vector B, and returns with the solution vector
X; input matrix is not modified in this routine and can be left in
place for successive calls with different right-hand sides b; this
routine takes into account the possibility that b will begin with many
zero elements, so it is eficient for use in matrix inversion
{
      int n, i, ii=-1, ip, j;
      float **a, sum;

      a=(*m).m;
      n=(*m).r;

      for(i=0; i<n; i++)
      {//when ii is set to a positive value, it will become the index
       // of the first nonvanishing element of b
            //we now do the forward substitution, the only new wrinkle
            // is to unscramble the permutation as we go
            ip=indx[i];
            sum=b[ip];
            b[ip]=b[i];
            if(ii>-1)
                  for(j=ii; j<=i-1; j++) sum-=a[i][j]*b[j];
            else if(sum)
                  ii=i;//nonzero element was encountered, so from now
                  // on we will have to do the sums in the loop above
            b[i]=sum;
      }

      for(i=n-1; i>=0; i--)
      {//now we do the backsubstitution
            sum=b[i];
            for(j=i+1; j<n; j++) sum-=a[i][j]*b[j];
            b[i]=sum/a[i][i];//store a component of the solution
      }
}
```

```
void sigma(matrix *s, const matrix *x, const int k)
//returns covariance matrix of x (where each row is an observation,
//and each column a variable) in matrix s
//k determines which means vector from global array will be used
{
	int i, j, n;
	float v, sum;

	//diagonal
	for(i=0; i<N_FLAGS; i++)
	{
		sum=0;
		for(n=0; n<N_SAMPLES; n++)
		{
			v=((*x).m[n][i])-(means[k].v[i]);
			sum+=v*v;
		}
		(*s).m[i][i]=sum/N_SAMPLES;
	}

	//symetric
	for(j=1; j<N_FLAGS; j++)
	{
		for(i=0; i<j; i++)
		{
			sum=0;
			for(n=0; n<N_SAMPLES; n++)
			{
				sum+=(((*x).m[n][i])-(means[k].v[i]))*
					(((*x).m[n][j])-(means[k].v[j]));
			}
			(*s).m[j][i]=(*s).m[i][j]=sum/N_SAMPLES;
		}
	}
}


void means_vector(const matrix *x, const int k)
//counts column means of matrix x
//and stores them in global array means as k-th element
{
	int i, n;
	float sum;

	for(i=0; i<N_FLAGS; i++)
	{
		sum=0;
		for(n=0; n<N_SAMPLES; n++) sum+=(*x).m[n][i];
		means[k].v[i]=sum/N_SAMPLES;
	}
}
```

```
int buffer_not_full(void)
{//returns zero value if buffer (defined globaly) is full
      if(buffer_count<BUFFER_SIZE) return 1;

      //buffer full
      work=buffer;//set full active buffer as working
      buffer_num=!buffer_num;//switch buffer numbers
      buffer=buffers[buffer_num];//set actual address
      buffer_count=0;//reset counter (enable sample storing)
      return 0;
}


void processBuffer(void)
//counts the energy of samples in work buffer
//and turns corresponding number of LEDs on (bar indicator)
{
      int i;
      long e=0;
      short bits;

      for(i=0; i<BUFFER_SIZE; i++) e+=work[i]*work[i];

      //set up the flags bit pattern according to the peak value
      if(e>0xf0000000) bits=0x3F;
      else if(e>0xc0000000) bits=0x1F;
      else if(e>0xa0000000) bits=0xF;
      else if(e>0x80000000) bits=0x7;
      else if(e>0x60000000) bits=0x3;
      else if(e>0x40000000) bits=0x1;
      else bits = 0;

      *pFlashA_PortB_Out=bits;//set flags
}


void flags(float *d)
//divides spectrum in global array mag into equidistant zones
//and sums them into flags in array d
//1. magnitude value (DC offset) is skipped
//output is scaled down by factor 16
//to prevent overflows in subsequent calculations
{
      int i, j, k=0;
      int n=FFT_SIZE/N_FLAGS;
      float sum=0.0;

      for(i=j=1; i<FFT_SIZE; i++, j++)
      {
            if(j>n)
            {
                  j=0;
                  d[k++]=sum/16;
                  sum=0.0;
            }
            sum+=mag[i];
      }
      d[k]=sum/16;//last one
}
```

```
void learn(void)
{//learning procedure
	int i, j, k=-1, indx[N_FLAGS];
	float d, freq=2000.0;
	matrix s, x;

	buffer_count=BUFFER_SIZE;//stop samples storing

	x=matrix_create(N_SAMPLES, N_FLAGS);//for flags vectors
	s=matrix_create(N_FLAGS, N_FLAGS);//for sigma

	ClearAllButtons();
	TurnOffAllLEDs();

	while(1)//classes are cycled indefinitly unless user cancels
	{
		if((++k)>=N_CLASSES) k=0;

		//set blink period (quicker blinking if class is defined)
		if(active_classes[k]) *pTPERIOD = 0x0001FFFF;
			else *pTPERIOD = 0x0003FFFF;

		//already defined classes have their LEDs lit
		for(i=0; i<N_CLASSES; i++)
			if(active_classes[i]) TurnOnLED(i+EZ_FIRST_LED);
				else TurnOffLED(i+EZ_FIRST_LED);

		StartBlink(k);
		while(IsNoButtonPushed());//wait for control button
		StopBlink();

		if(IsButtonPushed(BUTTON_LEARN))
		{//stop learning
			ClearAllButtons();
			break;
		}

		if(IsButtonPushed(BUTTON_NEXT))
		{//skip to next class
			ClearAllButtons();
			continue;
		}

		if(IsButtonPushed(BUTTON_DELETE))
		{//set class not active (will not be used in recognition)
			ClearAllButtons();
			active_classes[k--]=0;
			continue;
		}

		//if none of previous buttons, then start training
		ClearAllButtons();
		TurnOnLED(k+EZ_FIRST_LED);//turn on corresponding LED
		buffer_count=0;//start samples storing
```

```
            //collect flags
            for(i=0; i<N_SAMPLES; i++)
            {
                  while(buffer_not_full());//wait for full buffer

                  rfft_fr16((fract16 *)work, t, out, w, 1,
                                                BUFFER_SIZE, 0, 0);

                  //magnitude
                  for(j=0; j<FFT_SIZE; j++)
                        mag[j]=sqrt(out[j].re*out[j].re+
                                                out[j].im*out[j].im);

                  flags(x.m[i]);

                  //BTC write output
                  #ifdef _BTC_
                        btc_write_array(0, (unsigned int *)work,
                                          sizeof(fract16)*BUFFER_SIZE);
                        btc_write_array(1, (unsigned int *)mag,
                                          sizeof(mag));
                        btc_write_array(2, (unsigned int *)x.m[i],
                                          sizeof(float)*N_FLAGS);
                        btc_poll();
                  #endif
            }
            buffer_count=BUFFER_SIZE;//stop samples storing

            //statistical analysis
            means_vector(&x, k);
            sigma(&s, &x, k);

            //matrix LU-decomposition (needed for other operations)
            ludcmp(&s, indx, &d);//this returns d as ±1.

            //determinant
            for(i=0; i<N_FLAGS; i++) d*=s.m[i][i];
            dets_sqrts[k]=sqrt(d);

            //inversion of matrix
            inversion(&s, indx, k);

            active_classes[k]=1;
      }
      //determine number of active classes
      for(n_active_classes=k=0; k<N_CLASSES; k++)
            if(active_classes[k]) n_active_classes++;

      //cleaning
      matrix_free(&s);
      matrix_free(&x);
      TurnOffAllLEDs();
      buffer_count=0;//start samples storing again
}
```

Hlavičkový soubor funkcí (*fcns.h*)

```
#ifndef FCNS_H
#define FCNS_H

//types
typedef struct
{
      int c, r;//columns, rows
      float **m;//matrix data
      float *v;//vector data
} matrix;

matrix matrix_create(int, int);
void matrix_free(matrix *);
void multiply(matrix *, const matrix *, const matrix *);
void subtract(matrix *, const matrix *, const matrix *);
void transpose(matrix *, const matrix *);
void inversion(const matrix *, int *, int);
void ludcmp(matrix *, int *, float *);
void lubksb(const matrix *, int *, float []);
void sigma(matrix *, const matrix *, const int);
void means_vector(const matrix *, const int);
int buffer_not_full(void);
void processBuffer(void);
void flags(float *);
void learn(void);

#endif
```

Systémové utility (*sysutils.c*)

```
#include <services/services.h>      //system service includes
#include <sysreg.h>                 //system config definitions
#include <fract_typedef.h>
#include "sysutils.h"
#include "config.h"

extern int buffer_count;
extern volatile fract16 *buffer;
int BlinkLED;

//array for registers to configure the ad1836
volatile short sCodec1836TxRegs[CODEC_1836_REGS_LENGTH] =
{
      DAC_CONTROL_1     | 0x000,
      DAC_CONTROL_2     | 0x000,
      DAC_VOLUME_0      | 0x3ff,
      DAC_VOLUME_1      | 0x3ff,
      DAC_VOLUME_2      | 0x3ff,
      DAC_VOLUME_3      | 0x3ff,
      DAC_VOLUME_4      | 0x3ff,
      DAC_VOLUME_5      | 0x3ff,
      ADC_CONTROL_1     | 0x000,
      ADC_CONTROL_2     | 0x180,
      ADC_CONTROL_3     | 0x000
};
```

```
//SPORT0 DMA receive, transmit buffer
extern volatile int iRxBuffer[], iTxBuffer[];


void Init_EBIU(void)
//initializes and enables asynchronous memory banks in External
//Bus Interface Unit so that Flash A can be acessed
{
      //access time=7 cycles, read access time=11 cycles, no ARDY
      *pEBIU_AMBCTL0    = 0x7bb07bb0;
      //hold time=2, setup time=3, transition time=4 (cycles)
      *pEBIU_AMBCTL1    = 0x7bb07bb0;
      *pEBIU_AMGCTL     = 0x000f;//enable all memory banks
}


void Init_Flash(void)
//sets up the flash on the board for use
{
      *pFlashA_PortA_Dir = 0x1;//everything on port A as outputs
      *pFlashA_PortB_Out = 0;//resets port B to initial value
      *pFlashA_PortB_Dir = 0x3f;//everything on port B as outputs
}


void Init1836(void)
//sets up the SPI port to configure the AD1836
//the content of the array sCodec1836TxRegs is sent to the codec
{
      int i;
      static unsigned char ucActive_LED = 0x01;

      //write to Port A to reset AD1836
      *pFlashA_PortA_Out = 0x00;

      //write to Port A to enable AD1836
      *pFlashA_PortA_Out = ucActive_LED;

      //wait to recover from reset
      for (i=0; i<0xf000; i++) asm("nop;");

      //enable PF4
      *pSPI_FLG = FLS4;
      //set baud rate SCK = HCLK/(2*SPIBAUD) SCK = 2MHz
      *pSPI_BAUD = 16;
      //configure spi port
      //SPI DMA write, 16-bit data, MSB first, SPI Master
      *pSPI_CTL = TIMOD_DMA_TX | SIZE | MSTR;

      //set up DMA5 to transmit
      //map DMA5 to SPI
      *pDMA5_PERIPHERAL_MAP   = 0x5000;

      //configure DMA5
      //16-bit transfers
      *pDMA5_CONFIG = WDSIZE_16;
      //start address of data buffer
      *pDMA5_START_ADDR = (void *)sCodec1836TxRegs;
      //DMA inner loop count
```

```
        *pDMA5_X_COUNT = CODEC_1836_REGS_LENGTH;
        //inner loop address increment
        *pDMA5_X_MODIFY = 2;

        //enable DMAs
        *pDMA5_CONFIG = (*pDMA5_CONFIG | DMAEN);
        //enable spi
        *pSPI_CTL = (*pSPI_CTL | SPE);

        //wait until dma transfers for spi are finished
        for (i=0; i<0xaff0; i++) asm("nop;");

        //disable spi
        *pSPI_CTL = 0x0000;
}


void Init_Sport0(void)
//configure Sport0 for TDM mode, to transmit/receive data to/from
//the AD1836; configure Sport for external clocks and frame syncs
{
        //Sport0 receive configuration
        //external CLK, External Frame sync, MSB first, 32-bit data
        *pSPORT0_RCR1 = RFSR;
        *pSPORT0_RCR2 = SLEN_32;

        //Sport0 transmit configuration
        *pSPORT0_TCR1 = TFSR;
        *pSPORT0_TCR2 = SLEN_32;

        //enable MCM 8 transmit channel
        *pSPORT0_MTCS0 = 0x000000FF;

        //enable MCM 8 receive channel
        *pSPORT0_MRCS0 = 0x000000FF;

        //set MCM configuration register and enable MCM mode
        *pSPORT0_MCMC1 = 0x0000;
        *pSPORT0_MCMC2 = 0x101c;
}


void Init_DMA(void)
//initialize DMA1 in autobuffer mode to receive
{
        //set up DMA1 to receive
        //map DMA1 to Sport0 RX
        *pDMA1_PERIPHERAL_MAP = 0x1000;

        //configure DMA1
        //32-bit transfers, interrupt on completion, Autobuffer mode
        *pDMA1_CONFIG = WNR | WDSIZE_32 | DI_EN | FLOW_1;
        //start address of data buffer
        *pDMA1_START_ADDR = (void *)iRxBuffer;
        //DMA inner loop count
        *pDMA1_X_COUNT = 8;
        //inner loop address increment
        *pDMA1_X_MODIFY   = 4;
```

```
	//set up DMA2 to transmit
	//map DMA2 to Sport0 TX
	*pDMA2_PERIPHERAL_MAP = 0x2000;

	//configure DMA2
	//32-bit transfers, Autobuffer mode
	*pDMA2_CONFIG = WDSIZE_32 | FLOW_1;
	//start address of data buffer
	*pDMA2_START_ADDR = (void *)iTxBuffer;
	//DMA inner loop count
	*pDMA2_X_COUNT = 8;
	//inner loop address increment
	*pDMA2_X_MODIFY   = 4;
}


void Init_Sport_Interrupts(void)
//initialize interrupt for Sport0 RX
{
	//set Sport0 RX (DMA1) interrupt priority to 2 = IVG9
	*pSIC_IAR0 = 0xffffffff;
	*pSIC_IAR1 = 0xffffff2f;
	*pSIC_IAR2 = 0xffffffff;

	//assign ISRs to interrupt vectors
	//Sport0 RX ISR -> IVG 9
	register_handler(ik_ivg9, Sport0_RX_ISR);

	//enable Sport0 RX interrupt
	*pSIC_IMASK = 0x00000200;
	ssync();
}


void Enable_DMA_Sport0(void)
//enable DMA and Sport0 RX
{
	//enable DMAs
	*pDMA1_CONFIG     = (*pDMA1_CONFIG | DMAEN);
	*pDMA2_CONFIG     = (*pDMA2_CONFIG | DMAEN);

	//enable Sport0 TX and RX
	*pSPORT0_RCR1     = (*pSPORT0_RCR1 | RSPEN);
	*pSPORT0_TCR1     = (*pSPORT0_TCR1 | TSPEN);
}
```

```
//====
//LEDS
//====

void TurnOnLED(u32 LEDNumber)
{
	switch(LEDNumber)
	{
		case 4: *pFlashA_PortB_Out |= 0x01; break;
		case 5: *pFlashA_PortB_Out |= 0x02; break;
		case 6: *pFlashA_PortB_Out |= 0x04; break;
		case 7: *pFlashA_PortB_Out |= 0x08; break;
		case 8: *pFlashA_PortB_Out |= 0x10; break;
		case 9: *pFlashA_PortB_Out |= 0x20; break;
	}
}


void TurnOffLED(u32 LEDNumber)
{
	switch(LEDNumber)
	{
		case 4: *pFlashA_PortB_Out &= ~0x01; break;
		case 5: *pFlashA_PortB_Out &= ~0x02; break;
		case 6: *pFlashA_PortB_Out &= ~0x04; break;
		case 7: *pFlashA_PortB_Out &= ~0x08; break;
		case 8: *pFlashA_PortB_Out &= ~0x10; break;
		case 9: *pFlashA_PortB_Out &= ~0x20; break;
	}
}


void ToggleLED(u32 LEDNumber)
{
	if(IsLEDOn(LEDNumber))
		TurnOffLED(LEDNumber);
	else
		TurnOnLED(LEDNumber);
}


void TurnOnAllLEDs(void)
{
	*pFlashA_PortB_Out = 0x3f;
}


void TurnOffAllLEDs(void)
{
	*pFlashA_PortB_Out = 0x0;
}

u8 GetDisplay()
{//returns the current status of the LED display
	return *pFlashA_PortB_In;
}
```

```
void SetDisplay(u8 display)
{//sets the LED display
	*pFlashA_PortB_Out = display;
}


void CycleLEDs(void)
{//changes each time called
	static int LED = EZ_FIRST_LED;

	TurnOffLED(LED);
	LED++;
	if (LED > EZ_LAST_LED) LED = EZ_FIRST_LED;
	TurnOnLED(LED);
}


u32 IsLEDOn(u32 LEDNumber)
{//returns TRUE if an LED is lit, FALSE otherwise
	switch(LEDNumber)
	{
		case 4: if(*pFlashA_PortB_In & 0x01) return(TRUE); break;
		case 5: if(*pFlashA_PortB_In & 0x02) return(TRUE); break;
		case 6: if(*pFlashA_PortB_In & 0x04) return(TRUE); break;
		case 7: if(*pFlashA_PortB_In & 0x08) return(TRUE); break;
		case 8: if(*pFlashA_PortB_In & 0x10) return(TRUE); break;
		case 9: if(*pFlashA_PortB_In & 0x20) return(TRUE); break;
	}
	return(FALSE);
}


//=======
//Buttons
//=======

void InitButtons(void)
{//initializes the push button as input flags
	*pFIO_INEN=0x0f00;//enable flags as inputs
	*pFIO_DIR=0xF0FF;//set the flag direction as input
	*pFIO_EDGE=0x0f00;//set to edge sensitive (when enabled)
}


u32 IsButtonPushed(u32 ButtonNumber)
{//returns TRUE if a button has been pushed, FALSE otherwise
	switch(ButtonNumber)
	{
		case 4: if(*pFIO_FLAG_D & 0x0100) return(TRUE); break;
		case 5: if(*pFIO_FLAG_D & 0x0200) return(TRUE); break;
		case 6: if(*pFIO_FLAG_D & 0x0400) return(TRUE); break;
		case 7: if(*pFIO_FLAG_D & 0x0800) return(TRUE); break;
	}
	return(FALSE);
}
```

```
u32 IsNoButtonPushed(void)
{//returns TRUE if any button has been pushed, FALSE otherwise
      if(*pFIO_FLAG_D & 0x0F00) return(FALSE);

      return(TRUE);
}


void ClearButton(u32 ButtonNumber)
//clears a push button latch; this must be called to reset the latch
//for the push button, if a button has been pressed
{
      switch (ButtonNumber)
      {
            case 4: *pFIO_FLAG_C = 0x0100; break;
            case 5: *pFIO_FLAG_C = 0x0200; break;
            case 6: *pFIO_FLAG_C = 0x0400; break;
            case 7: *pFIO_FLAG_C = 0x0800; break;
      }
}


void ClearAllButtons(void)
{//clears all push button latches
      *pFIO_FLAG_C = 0x0F00;
}


//========
//Blinking
//========

void initTimer(void)
{
      *pTCNTL = 5;//timer control register

      *pTPERIOD = 0x00038FFF;//timer period register

      *pTSCALE = 0x0000FFFF;//timer scale register

      *pTCOUNT = 0x0000FFFF;//timer count register
}


void StartBlink(int i)
{
        BlinkLED=i+EZ_FIRST_LED;

      *pTCNTL |= 0x2;   //enable Timer
}


void StopBlink(void)
{
      *pTCNTL &= ~0x00000002; //disable Timer
}
```

```
//====
//ISRs
//====

EX_INTERRUPT_HANDLER(timerISR)
{//timer interrupt handler
      ToggleLED(BlinkLED);
}

EX_INTERRUPT_HANDLER(Sport0_RX_ISR)
//executed after a complete frame of input data has been received
{
      //confirm interrupt handling
      *pDMA1_IRQ_STATUS = 0x0001;

      if(buffer_count<BUFFER_SIZE)
      {//copy input data from DMA input buffer into main buffer
            buffer[buffer_count++]=(iRxBuffer[0]>>16);
            //only high 16 bits of INTERNAL_ADC_L0 is used (from 24)
      }

      //copy data from DMA input to output
      iTxBuffer[0] = iRxBuffer[0];
      iTxBuffer[4] = iRxBuffer[4];
}
```

Hlavičkový soubor systémových utilit (*sysutils.h*)

```
#ifndef SYSUTILS_H
#define SYSUTILS_H

//header files
#include <sys\exception.h>
#include <services\services.h>
#include <cdefBF533.h>
#include <ccblkfn.h>
#include <sysreg.h>

//symbolic constants
#define pFlashA_PortA_In      ((volatile unsigned char *)0x20270000)
#define pFlashA_PortA_Out     ((volatile unsigned char *)0x20270004)
#define pFlashA_PortA_Dir     ((volatile unsigned char *)0x20270006)
#define pFlashA_PortB_In      ((volatile unsigned char *)0x20270001)
#define pFlashA_PortB_Out     ((volatile unsigned char *)0x20270005)
#define pFlashA_PortB_Dir     ((volatile unsigned char *)0x20270007)

//names for codec registers, used for iCodec1836TxRegs[]
#define DAC_CONTROL_1         0x0000
#define DAC_CONTROL_2         0x1000
#define DAC_VOLUME_0          0x2000
#define DAC_VOLUME_1          0x3000
#define DAC_VOLUME_2          0x4000
#define DAC_VOLUME_3          0x5000
#define DAC_VOLUME_4          0x6000
#define DAC_VOLUME_5          0x7000
#define ADC_0_PEAK_LEVEL      0x8000
#define ADC_1_PEAK_LEVEL      0x9000
```

```
#define ADC_2_PEAK_LEVEL      0xA000
#define ADC_3_PEAK_LEVEL      0xB000
#define ADC_CONTROL_1         0xC000
#define ADC_CONTROL_2         0xD000
#define ADC_CONTROL_3         0xE000

//size of array iCodec1836TxRegs and iCodec1836RxRegs
#define CODEC_1836_REGS_LENGTH     11

//SPI transfer mode
#define TIMOD_DMA_TX 0x0003

//SPORT0 word length
#define SLEN_32   0x001f

//DMA flow mode
#define FLOW_1    0x1000

//Enumerations for the first and last buttons and LEDs
#define EZ_FIRST_LED    (4)//first LED number
#define EZ_LAST_LED     (9)//last LED number
#define EZ_FIRST_BUTTON (4)//first push button
#define EZ_LAST_BUTTON  (7)//last push button

//Functions provided by the utilities
void  Init_EBIU                (void);
void  Init1836                 (void);
void  Init_Sport0              (void);
void  Init_DMA                 (void);
void  Init_Sport_Interrupts    (void);
void  Enable_DMA_Sport0        (void);
void  Init_Flash               (void);
void  initTimer                (void);
void  StartBlink               (int);
void  StopBlink                (void);

void  TurnOnLED          (u32 LEDNumber);//turns on an LED
void  TurnOffLED         (u32 LEDNumber);//turns off an LED
void  ToggleLED          (u32 LEDNumber);//toggles an LED
u32   IsLEDOn            (u32 LEDNumber);//test to see if an LED is lit

void  TurnOnAllLEDs      (void);//turns on all LEDs
void  TurnOffAllLEDs     (void);//turns off all LEDs
void  CycleLEDs          (void);//cycles through LEDS
u8    GetDisplay         (void);//returns current state of LED display
void  SetDisplay         (u8 display);//sets LED display state

void  InitButtons        (void);//initialized push buttons
u32   IsButtonPushed     (u32 ButtonNumber);//is a push button pressed
u32   IsNoButtonPushed   (void);//is no push button pressed
void  ClearButton        (u32 ButtonNumber);//clears push button latch
void  ClearAllButtons    (void);//clears all push button latches


EX_INTERRUPT_HANDLER(Sport0_RX_ISR);//Sport0 ISR
EX_INTERRUPT_HANDLER(timerISR);//timer interrupt

#endif
```