Vysoké učení technické v Brně

Fakulta informačních technologií

Fixed-point implementácia rozpoznávača reči

Diplomová práca

Brno, január 2007 Tomáš Král

Zadanie diplomovej práce

Fixed-point implementácia rozpoznávania reči

Vedúci: Burget Lukáš, Ing., Phd.D., UPGM FIT VUT

Zadanie:

1. Implementujte parametrizáciu reči pomocou MFCC koeficientov.
2. Parametrizujte rečovú databázu, nad ktorou vedúci natrénuje akustické modely.
3. Vytvorte funkciu pre vyhodnotenie hodnoty zmesi Gaussových rozdelení.
4. Vytvorte jednoduchý dekodér pre rozpoznávanie reči alebo kľúčových slov.
5. Otestujte na reálnych rečových dátach a vyhodnoťte úspešnosť.

Prehlásenie

Prehlasujem, že tento ročníkový projekt som vypracoval samostatne pod vedením Ing. Lukáša Burgeta, Ph.D. a Doc. Dr. Ing. Jana Černockého.

Všetky zdroje z ktorých som čerpal informácie, ako knižné tituly, výukové materiály a iné dokumenty, som uviedol v záverečnej časti dokumentu.

V Brne, dňa 13.decembra 2006 .....

Tomáš Král

Abstrakt

Táto diplomová práca sa zaoberá problematikou automatického rozpoznávania reči na systémoch s obmedzenými hardwarovými prostriedkami – embedded systems. Cieľom projektu je navrhnúť a implementovať systém rozpoznávania reči na embedded systémy, ktoré nedisponujú floating-point výpočetnými jednotkami. V prvom rade bola zvolená vhodná hardwarová architektúra a s ohľadom na dostupné prostriedky, ktorými vybraná architektúra disponuje, bolo navrhnuté riešenie rozpoznávania reči. Jednotlivé časti systému rozpoznávania boli následne v priebehu vývoja optimalizované do takej podoby, aby mohli byť nasadené na zvolený HW. Výsledkom práce je dosiahnutie rozpoznávania českých čísloviek na embedded systéme.

Kľúčové slová

rozpoznávanie reči, embedded systémy, DSP, fix-point rozpoznávanie, Viterbiho algoritmus, skryté Markovove modely, MFCC, extrakcia rečových príznakov

Abstract

Master thesis is related to the problematics of automatic speech recognition on systems with restricted hardware resources – embedded systems. The object of this work was to design and implement speech recognition system on embedded systems, that do not contain floating-point processing units. First objective was to choose proper hardware architecture. Based on the knowledge of available HW resources, the recognition system design was made. During the system development, optimalization was made on constituent elements so they could be mounted on chosen HW. The result of the the project is successful recognition of czech numerals on embedded system.

Key words

speech recognition, embedded systems, DSP, fix-point recognition, Viterbi algorithm, Hidden Markov Models, MFCC, feature extraction

Obsah

Poďakovanie

Na tomto mieste by som chcel vyjadriť poďakovanie Ing. Lukášovi Burgetovi a Doc.Dr.Ing. Janu Černockému, ktorý mi poskytli nevyhnutné informácie a rady pri riešení tohto projektu a bez ktorých pomoci by dokončenie práce trvalo oveľa dlhšie. Tak isto by som chcel poďakovať aj Ing. Petrovi Schwarzovi za poskytnuté konzultácie.

1. Úvod

1.1 O tejto práci

Systém rozpoznávania reči adaptovaný na embedded systéme: toto si dáva za cieľ projekt zadaný Ústavom počítačovej grafiky a multimédií, na ktorom som pracoval a o ktorom pojednáva táto práca. Dokument je koncipovaný tak, aby nás najskôr stručne uviedol do problematiky, vysvetlil dôležité pojmy, postupy a metódy, a aby nás následne postupne previedol cez jednotlivé fázy vývoja tohto systému. Dokument nemá za úlohu detailne vysvetlovať teóriu počítačového rozpoznávania reči a preto sa predpokladá, že čitateľ má základné teoretické vedomosti o tejto problematike naštudované. Na druhú stranu to, na čo sa v tejto práci najviac sústredíme, sú skôr praktické skúsenosti, zistenia a návody, ktoré by mali pomôcť pri vývoji systémov rozpoznávania reči na embedded systémoch.

1.2 Stručný obsah kapitol

Dokument je rozdelený do šiestich logických častí. Úvodná **1.kapitola** nás oboznámi s dokumentom, uvedie nás do širšieho kontextu problematiky rozpoznávania reči a definuje ciele tohto projektu. **2.kapitola** sa zaoberá teóriou rozpoznávania reči, na ktorú sa pozrieme len zo široka a spomenieme len základné informácie na pripomenutie si vedomostí, bez ktorých sa v ďalšom pokračovaní nezaobídeme. Čo sú to embedded systémy, ich vlastnosti a možnosti implementácie rozpoznávania reči na takýchto systémoch, to sú už témy ktoré si objasníme v nasledujúcej **3.kapitole**. Okrem toho si na tomto mieste už povieme aj o konkrétnej hardwarovej architektúre a integrovanom vývojovom prostredí v ktorom bol projekt vyvýjaný. Následne sa dopracujeme k jadru celej problematiky, ktoré je spracované vo **4.kapitole.** Návrh systému rozpoznávania. Definujeme si jednotlivé postupy, mechanizmy a metódy, ktoré budeme využívať a tak isto si definujeme aj vlastnosti jednotlivých blokov použitých v systéme rozpoznávania. Realizáciu navrhnutého systému preberieme v **5.kapitole**. Tu nájdeme detailný popis implementácie rozpoznávania reči na hardwarovej architektúre embedded systému. Definujeme si štruktúru systému rozpoznávania, uvedieme použité hardwarové prostriedky, spôsob ich nastavenia a metódy, ktorými bude HW komunikovať a obsluhovať ďalšie aplikačné vrstvy. Potom preberieme implementáciu jednotlivých algoritmov, ktoré tvoria jadro rozpoznávania reči. Záverom v **6.kapitole** zhrnieme dosiahnuté výsledky, úspešnosť a navrhneme možnosťi ďalšieho smerovania projektu.

1.3 Náväznosť projektu

Na tomto mieste by som chcel uviesť náväznosť tohto projektu na predchádzajúcu prácu, ktorá bola riešená ako ročníkový projekt a ktorá sa zaoberala parametrizáciou (feature extraction) rečového signálu na embedded systéme – DSP. Dosiahnuté výsledky ročníkového projektu boli použité aj v tejto diplomovej práci. Feature Extraction, resp. metódy parametrizácie, prevzaté z predchádzajúceho

projektu museli byť avšak v tejto práci ďalej optimalizované, aby neboli prekročené výpočetné možnosti, ktoré je daná hardwarová architektúra svojimi prostriedkami schopná poskytnúť.

1.4 Rozpoznávanie reči

Systémy pre rozpoznávanie reči sú v dnešnej dobe veľmi aktuálnou, mnohými odborníkmi diskutovanou a vo veľa výskumných laboratóriách skúmanou oblasťou, ktorá do dnešnej doby priniesla tak ako určité pokroky a úspechy, aj neúspechy a stále nové a nové otázky na ktoré potrebujeme nájsť odpoveď tak, aby bol zachovaný určitý technologický rast a pokrok. To, že po technológiách týkajúcich sa systémov rozpoznávania reči je obrovský dopyt je nepopierateľné. Výrobcovia spotrebnej elektroniky by určite radi poskytli spotrebiteľovi možnosť prijať hovor na mobilnom telefóne obyčajným vyslovením príkazu „accept“, či pohodlné zníženie hlasitosti výkonnej audio Hi-Fi sústavy vyslovením príkazu „volume down“ bez toho, aby sme museli prekričať našu obľúbenú pesničku, ktorá nám vyhráva na najvyššej hlasitosti. Posunutím o stupienok vyššie v technológií rozpoznávania reči by sme sa mohli dostať do automobilového priemyslu. Tu už okrem štandartných príkazov ktoré poznáme z ovládania spotrebnej elektroniky požadujeme aj príkazy ako „find hotel“, ktorý aktivuje systém GPS, alebo príkaz „report speed limit“, ktorý v kooperácií s ostatnými systémami informuje o povolenej rýchlosti v danom mieste.

Tieto príklady predstavujú základnú myšlienku využitia systémov rozpoznávania reči v praxi. Vzniká teda otázka, ako ďaleko sme sa v tomto smere dostali. Všetko závisí od kvality rozpoznávania ktorú požadujeme. Jednoduché rozpoznávače, ktoré poznáme z mobilných telefónov a iných malých elektronických zariadení sú založené na jednoduchom porovnávaní rozpoznávaného signálu a referenčného vzorku, ktorý bol v minulosti do telefónu nahraný užívateľom. Zariadenie teda reaguje len na hlas jedného užívateľa od ktorého sa očakáva, že príkaz zopakuje rovnakým tónom a rovnakou rýchlosťou. Takýmto systémom chýba akákoľvek robustnosť. Princíp technológie o niečo dokonalejších rozpoznávačov pri ktorých nezáleží či rozpoznávané slovo povedalo dieťa so základným tónom na vysokých frekvenciách, alebo muž so základným tónom na nízkych frekvenciách, prípadne že rozpoznávané slovo bolo povedané rozdielnou rýchlosťou, je tiež zvládnutý. Avšak s technologickou dokonalosťou sa úmerne zvyšujú aj hardwarové nároky. Tu už narážame na problém aplikácie takýchto vyspelejších systémov napríklad do mobilných telefónov, ktorých hardwarové vybavenie nieje dostatočné. Ďalším problémom, ktorý je za potreby v každom prípade nejakým spôsobom minimalizovať je vplyv šumu na rozpoznávanie reči. Aj rozpoznávače pochádzajúce z renomovaných výskumných ústavov majú veľké problémy so šumom, ktorý je prítomný všade okolo nás. Šoférom by sa napríklad určite nepáčilo, ak by ich auto prestalo reagovať na hlasové pokyny zakaždým keď okolo prejde hlučný kamión.

Až po vyriešení týchto, svojím spôsobom elementárnych problémov sa v budúcnosti budeme môcť začať zaoberať vysoko sofistikovanými systémami pre rozpoznávanie reči. V prvom rade sa jedná o systém, ktorý bude schopný z hovoreného slova vytvárať textové dokumenty. Ďalej systémy, ktoré budú splňovať požiadavky národných bezpečnostných inštitúcií. Jedná sa najmä o rozpoznávače, ktoré by mohli byť nasadené na rizikových miestach ako letiská, vlakové stanice, či veľké obchodné centrá a boli by schopné poradiť si so šumom, ktorý dav ľudí spôsobuje a filtrovať len dôležité, resp. podozrivé signály.

1.5 Rozpoznávanie na čipe

Rozpoznávače reči sú z pochopiteľných dôvodov najefektívnejšie na osobných počítačoch, ktoré im poskytujú dostatočné hardwarové i softwarové prostriedky. Bohužiaľ v praxi, na miestach kde sa bežne stretávame, alebo zatiaľ by sme sa len chceli stretávať s kvalitnými rozpoznávačmi väčšinou nieje priestor pre fyzicky veľké a energeticky náročné osobné počítače.

Z tohto dôvodu nemenej dôležitou úlohou pri vývoji systémov pre rozpoznávanie reči je aj ich optimalizácia do takej podoby, aby boli schopné fungovania aj na systémoch s obmedzenými hardwarovými či softwarovými prostriedkami, na tzv. embedded systémoch.

Cieľom tohto projektu je implementovať rozpoznávač reči na čip. Pri hľadaní vhodného čipu, ktorý by sa na túto úlohu najlepšie hodil nebolo treba dlho premýšľať. Voľba padla na DSP. Digitálne signálové procesory poskytujú prostriedky, ktoré sú pre funkčnosť rozpoznávača reči nevyhnutne potrebné. Disponujú dostatočným výpočetným výkonom, sú pripojiteľné k externým zariadeniam a prostriedkom ako je pamäť, audio kodek, paralelné, sériové a iné rozhrania. Bližšie si o týchto zariadeniach povieme v kapitole venovanej DSP.

2. Teória rozpoznávanie reči

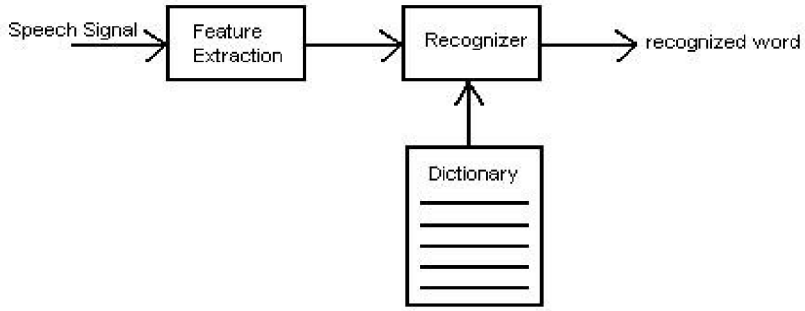
2.1 Rozpoznávanie reči a veda

Problematika počítačového rozpoznávania reči je stará takmer ako počítače samotné. Vychádza to z faktu, že najprirodzenejšou formou komunikácie pre človeka je verbálna forma. Už od samotných počiatkov vedného oboru Computer Science, vedci začali premýšľať o možnosti naučiť počítače porozumieť tejto forme komunikácie. Postupom času sa však prišlo na to, že táto, na prvý pohľad možno nie až tak komplikovaná úloha, v sebe ukrýva veľké množstvo problémov, ktoré je potrebné vyriešiť. Pozrime sa teda pre názornosť, vedomosti ktorých vedných oborov potrebujeme pri vývoji systémov rozpoznávania reči:

– spracovanie signálov	- z rečového signálu potrebujeme získať relevantné dáta
– fyzika: akustika	- porozumenie fyziologických mechanizmov
– pattern recognition	- klasterizácia dát a tvorba prototypov reprezentujúcich jednotlivé klustery
– komunikácia a teória informácií	-odhadovanie parametrov, detekcia rečových vzorkov
– fonetika	- vzťahy medzi zvukmi
– syntaxa, sémantika	- gramatika a význam slov
– informatika	- návrh efektívnych algoritmov
– a mnoho ďalších	

2.2 Dekompozícia problému rozpoznávania

Systém rozpoznávania reči je proces, ktorý je značne komplikovaný a pre pochopenie jeho činnosti je potrebná dekompozícia jednotlivých častí do logických blokov.



Obrázok 2.1 *Blokové schéma rozpoznávača*

Rečový signál, ktorý je produkovaný hlasovým ústrojenstvom na jednej strane a prijímaný posluchovým ústrojenstvom na druhej, v sebe nesie obrovský informačný obsah, ktorý je ľudský mozog schopný bez problémov spracovať a extrahovať si potrebné informácie. Bohužiaľ dnešné počítače ešte nedisponujú takou výpočetnou silou, ktorá by tohto bola schopná. Z tohto dôvodu potrebujeme na začiatku rečový signál upraviť do takej formy, aby bol použiteľný v počítačovom spracovaní. Okrem relevantných informácií sú v rečovom signále prenášané aj informácie o človeku, ktorý rozpráva, farbe jeho hlasu, nálade, prostredí, … Tieto informácie sú pre systémy rozpoznávania reči nadbytočné a preto ich môžeme odstrániť. K takejto úprave rečového signálu je určená parametrizácia, tzv. Feature Extraction.

Po získaní parametrov reči nasleduje samotné rozpoznávanie založené na princípe porovnávania vstupného vzorku s referenčnými vzorkami. V tomto momente vzniká ďalší problém. Vzorky, resp. slová, ktoré rozpoznávame totiž vo väčšine prípadov niesú rovnako dlhé. Riešením by mohla byť lineárna transformácia a zarovnanie dĺžky porovnávaných slov, avšak v praxi sa ukázalo, že toto nieje ideálne riešenie. V prípade, že rozpoznávané slovo, resp. vektor parametrov slova by bol dlhší ako vektor parametrov referenčnej vzorky, museli by sme v zásade niektoré parametre vylúčiť a tým by sme mohli stratiť práve dôležité informácie popisujúce reč. Podobný problém nastáva aj opačne, ak porovnávaný vektor parametrov musíme doplniť o ďalšie parametre, aby sme dosiahli zarovnanie. Týmto by sme zaviedli nepresnosti, ktoré môžu negatívne ovplyvniť výsledok. Ideálnym riešením je tzv. dynamické programovanie. Dynamic Time Wrapping (DTW) a Hidden Markov Models (HMM) sú techniky dynamického programovania, ktoré dokážu porovnávať vzorky rozdielnej dĺžky a vrátiť nám mieru podobnosti dvoch porovnávaných vektorov, resp. rozpoznať slovo.

Nepostrádateľnou súčasťou systémov rozpoznávania sú modely rozpoznávaných slov. Získavame ich postupným trénovaním nad množinou referenčných vzoriek. Pri DTW, najjednoduchším spôsobom popisu modela je jedna referenčná sekvencia vektorov reprezentujúca rozpoznávané slovo. V ideálnom prípade je to viac referenčných sekvencí pre slovo, alebo vytvorenie priemerného vzoru z viacerých referenčných sekvencí pre každé rozpoznávané slovo. Modely slov reprezentované HMM sú o niečo zložitejšie. Jednotlivé stavy HMM sú popisované zmesou Gaussovských rozložení pravdepodobnosti a definované sú pravdepodobnosti prechodov medzi jednotlivými stavmi.

2.3 Extrakcia rečových príznakov

Z vedných oborov, ktoré boli spomenuté, že sa významnou mierou podieľajú na rozpoznávaní reči, by sme mohli spracovanie signálov postaviť do pozície akéhosi základu. Tento fakt je daný tým, že rečový signál musíme nejakým spôsobom parametrizovať tak, aby boli tieto parametre prijateľné a spracovateľné systémami rozpoznávania. Vlastnosťou parametrov, ktorú sa zakaždým snažíme dosiahnuť je to, aby parametre zachytávali minimálnu množinu tých vlastností rečového signálu, ktoré sú pre zvolenú úlohu potrebné. Ostatné informácie a vlastnosti by mali zostať neznáme a nemali by sme nimi systém zbytočne zaťažovať.

Jednotlivé parametre rečového signálu sú vlastne sekvencie vektorov čísiel, ktoré zachytávajú dôležité vlastnosti krátkeho časového úseku signálu, tzv. rámca. Aby sme boli schopný zachytiť všetky dôležité vlastnosti a zmeny signálu, potrebujeme pracovať nad segmentami, rámcami signálu špecifickej dĺžky. Prvým krokom pri extrakcii rečových príznakov, resp. parametrov je vytvorenie rámcov. Na signál, ktorý považujeme vo všeobecnosti za náhodný sa potrebujeme v rámci pozerať ako na signál stacionárny. Rámec musí byť teda dostatočne krátky na to, aby sme signál mohli považovať za stacionárny, ale na druhú stranu aj dostatočne dlhý na to, aby sme boli schopný zachytiť požadované relevantné vlastnosti. Na rámcoch, ktoré sa navyše navzájom aj prekrývajú, zachytíme aj malé výchylky signálu. Hodnoty, ktoré sa ukázali v praxi ako vyhovujúce sú nasledovné: postačujúca vzorkovacia frekvencia rečového signálu je 8000Hz. Dĺžka jednotlivých rámcov sa pohybuje v rozmedzí 20-25ms (tj. 160-200 vzoriek pri 8kHz) a prekrytie rámcov sa nastavuje na hodnotu 10ms. Použitím týchto hodnôt získame pre signál dlhý jednu sekundu až 100 rámcov.

Existuje široké spektrum možností parametrického vyjadrenia rečového signálu. Skalárne parametre (jedna hodnota pre jeden rámec) počítané na základe strednej krátkodobej energii dokážu detekovať rečovú aktivitu, rozlíšiť znelé a neznelé hlásky. Sú však veľmi náchylné na šum. Počet priechodov nulou, počet priechodov úrovňou a iné skalárne parametre sa tak isto obmedzujú len na špecifické použitie, ktoré sú pre rozpoznávanie reči nevyhovujúce. Potrebujeme teda vektorové parametre, ktorých informačná hodnota je oveľa väčšia. Spektrálna analýza sa ukazuje ako veľmi silný nástroj pre popis parametrov rečového signálu. Medzi dve najrozšírenejšie metódy spektrálnej analýzy patria: spektrálna analýza založená na banke filtrov a spektrálna analýza založená na Linear Predictive Coding (LPC). V nasledujúcom výklade sa budeme venovať analýze založenej na banke filtrov.

Banky filtrov

Nasledujúci krok po získaní rečového rámca pozostáva z použitia okienkovej funkcie (window function). Typicky sa používa Hammingovo okno, ktoré svojím spôsobom utlmí význam vzoriek, ktoré sa nachádzajú na okrajoch jednotlivých rámcov a vyzdvihne význam hodnôt v okolí stredu rámca. Na takto pripravený rámec aplikujeme Fourierovu transformáciu a umocnením modulu získaných komplexných čísiel na druhú získame výkonové spektrum rámca.

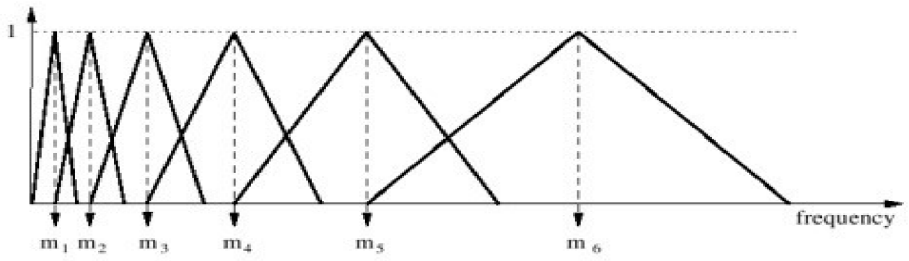
Cepstrálne koeficienty: ich úlohou je z vykonového spektra odstrániť nepotrebné informácie o budení signálu (t.j. šum a základný tón reči - funkcia hrtanu a hlasiviek) a získať len informácie popisujúce modifikáciu signálu (t.j. funkcia hlasového, artikulačného traktu). Rečový signál je vlastne

konvolúciou budenia a impulznej odozvy filtra, ktorý je tvorený artikulačným traktom. Našou úlohou teda je tieto dve zložku od seba oddeliť a zdroj budenia do ďalšieho spracovania už nezaradiť. DFT dokáže túto konvolúciu rozdeliť.

$$c(n) = \mathcal{F}^{-1}\left\{\ln|\mathcal{F}[s(n)]|^2\right\},$$

Výsledné cepstrálne koeficienty c(n), sú už kvalitatívne na vyhovujúcej úrovni a mohli byť použité pri rozpoznávaní reči.

Mel-frekvenčné cepstrálne koeficienty (MFCC): Avšak až zavedením Mel-frekvenčných cepstrálnych koeficientov získavame koeficienty, ktoré sa v praxi bežne využívajú. Zlepšenie oproti klasickým cepstrálnym koeficientom vychádza z faktu, že DFT má na celom spektre rovnaké frekvenčné rozlíšenie, na rozdiel od ľudské ucha, ktorého rozlišovacia schopnosť klesá s narastajúcou frekvenciou. Z tohto dôvodu umiestnime na frekvenčnú osu nelineárne filtre(tzv. Mel banky, ktorých je 23), zmeriame energie na ich výstupoch a použijeme ich pre výpočet cepstra (namiesto DFT).



Obrázok 2.2 *Melfrekvenčné filtre*

Na výkonové spektrum, ktoré sme si vypočítali aplikujeme 23 Mel-baniek (tj. vynásobenie trojuholníkovým oknom a sčítanie cez jednotlivé hodnoty). 23 hodnôt, ktoré týmto spôsobom získali, zlogaritmujeme a aplikujeme spätnú Fourierovu transformáciu. V skutočnosti sa ale nejedná o klasickú spätnú Fourierovu transformáciu, ale realizujeme diskrétnu kosínusovú transformáciu, z ktorej nám vypadne 13 čísiel. Získali sme teda 13 Mel-frekvenčných cepstrálnych koeficientov (MFCC). V praxi sa k týmto 13 koeficientom, ešte dopočíta 26 derivačných koeficientov (delta koef.), čo nám dáva dokopy typický počet, 39 MFCC. V tejto práci budeme ďalej pracovať len s pôvodnými 13 koeficientami, ktoré sú pre potreby rozpoznávania reči postačujúcich.

2.4 Mechanizmus rozpoznávania

V tomto momente teda máme k dispozícii rečové príznaky MFCC, ktoré nám popisujú rečový signál a môžeme začať so samotným rozpoznávaním. Systémy rozpoznávania reči môžeme rozdeliť podľa funkcie na:

- rozpoznávače izolovaných slov
- rozpoznávače spojených slov z obmedzeného slovníka
- rozpoznávače plynulej reči s veľkým slovníkom

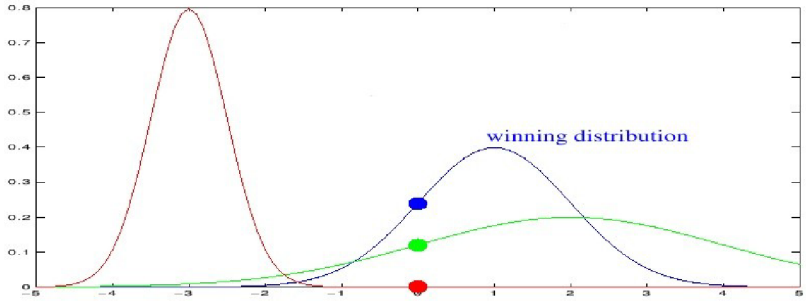
V nasledujúcom výklade sa obmedzíme len na rozpoznávanie izolovaných slov. Majme teda slovo X, ktoré chceme rozpoznať. Toto slovo bude po parametrizácií reprezentované sekvenciou vektorov O = [o(1),o(2),o(3), … o(T)]. Na druhej strane, v slovníku[1] máme uložené natrénované modely, ktoré popisujú rozpoznávané slová. Našou úlohou je určiť, či sa slovo, ktoré je privedené na vstup nachádza v slovníku a prípadne ho identifikovať. Na výber máme možnosť rozpoznávania na základe určenia vzdialenosti medzi dvoma vektormi, alebo rozpoznávanie na základe štatistického modelovania. Pre zopakovanie treba pripomenúť, že dĺžky jednotlivých sekvencií niesu rovnaké a preto musíme použiť metódy dynamického programovania.

Dynamic Time Wrapping je metóda dynamického programovania, ktorá počíta vzdialenosť medzi rôzne dlhými sekvenciami referenčných a testovacích vektorov. Jej použitie sa obmedzuje na rozpoznávače izolovaných slov. V tejto práci DTW nebudeme používať, detaily možno naštudovať v literatúre [2].

Hidden Markov Models

Táto metóda dynamického programovania bola použitá pri riešení tohto projektu, takže sa na ňu pozrieme trocha podrobnejšie. HMM využíva štatistické modelovanie, ktoré spočíva v získaní najväčšej možnej pravdepodobnosti (správnejšie vierohodnosti), že zadaný model generuje požadovanú sekvenciu vektorov.

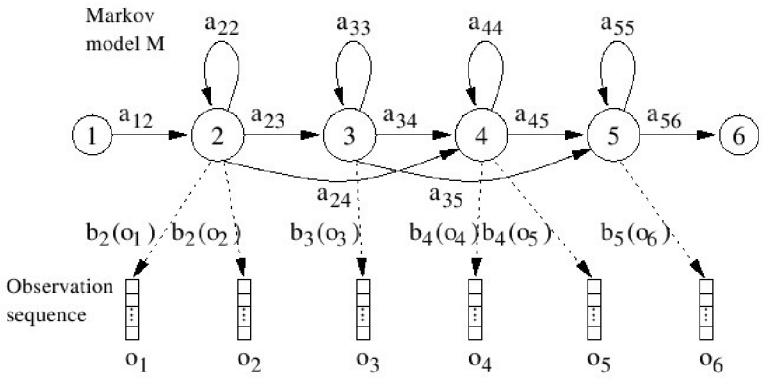
V prvom rade treba pochopiť akou formou sú natrénované slová, ktoré chceme rozpoznávať uložené v slovníku. Tou formou je Gaussovské rozloženie pravdepodobnosti. Ak by slová boli reprezentované jedným skalárom (čo je v praxi nepoužiteľné), po natrénovaní 3 slov by výsledné 1-rozmerné Gaussovské rozloženia mohli vypadať nasledovne:



Obrázok 2.4 *Gaussovské rozloženia pravdepodobností*

1 Slovník – v tejto práci budeme týmto slovom označovať množinu modelov rozpoznávaných slov.

Reálnejší príklad: ak sú jednotlivé slová popisované jedným celým vektorom, predchádzajúce výsledné rozloženie sa zmení len v tom zmysle, že sa počíta s ďalšími n-rozmermi Gaussových rozložení. My ale vieme, že jednotlivé slová sú reprezentované celou sekvenciou, 13 resp. 39 rozmerných vektorov. Logicky sa ponúka riešenie, vytvoriť pre každý vektor jedno Gaussovské rozloženie. Pripomeňme si však, že slová majú väčšinou rozdielnu dĺžku, takže potrebujeme iné riešenie. Zavedieme teda modely, v ktorých sa jednotlivé Gaussovské rozloženia môžu opakovať. Týmto krokom v skutočnosti vytvoríme skryté Markovove modely (HMM). Jednotlivé slová v slovníku budú popísané Markovými modelmi, v ktorých budú logicky umiestnené a poprepájané Gaussovské rozloženia pravdepodobnosti.

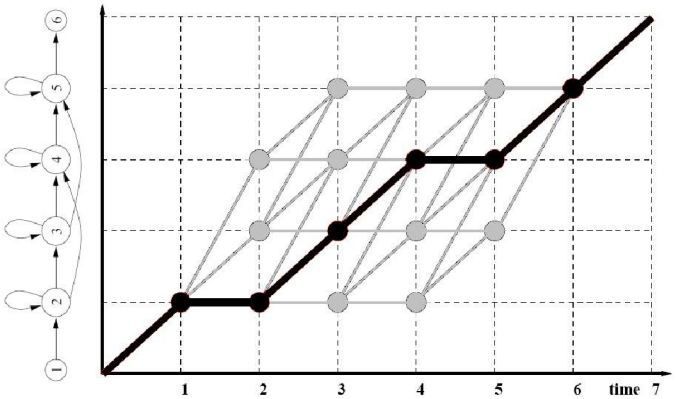


Obrázok 2.5 *Hidden Markov Model*

Jednotlivé funkcie hustoty rozloženia vysielacích pravdepodobností pre P-rozmerné Gaussovské rozloženie sa počítajú:

$$b_j[o(t)] = \prod_{i=1}^{P} \mathcal{N}(o(t); \mu_{ji}, \sigma_{ji}) = \prod_{i=1}^{P} \frac{1}{\sigma_{ji}\sqrt{2\pi}} e^{-\frac{[o(t)-\mu_{ji}]^2}{2\sigma_{ji}^2}} \tag{2.1.}$$

Teraz keď už vieme počítať vysielacie pravdepodobnosti Gaussovských rozložení (=hodnoty stavov HMM) môžeme vypočítať celkovú pravdepodobnosť, že model M generuje sekvenciu vektorov O. Jednotlivé vektory sekvencie O, sa pokúsime rozložiť na HMM stavy tak, aby sme získali najväčšiu možnú vierohodnosť. V nasledujúcom obrázku sú zobrazené všetky možné prechody v HMM:



Obrázok 2.6 *Priebeh pravdepodobnosti generovania sekvencie O*

… vyznačená postupnosť prechodov X=[1,2,2,3,4,4,5,6] nám dáva (teoreticky) najväčšiu vierohodnosť generovania sekvencie O modelom M. Pri pohľade na tento graf je hneď jasné, že všetkých možných postupností prechodov je veľmi veľa a výpočet vierohodností pre všetky prechody by bol veľmi náročný. Pri riešení tohto problému nám našťastie pomôže efektívny Viterbiho algoritmus. Ten si priebežne uchováva najlepšie dočasné výsledky a na konci, keď sa dostane v čase *t* k poslednému prvku vektora O(t) je okamžite schopný určiť najväčšiu pravdepodobnosť generovania sekvencie O modelom M.

Rozpoznávač reči pracujúci na princípe HMM teda funguje nasledovne. V prvom rade si natrénujeme modely HMM pre slová ktoré chceme rozpoznávať. Na vstup nám príde sekvencia parametrov slova O. Sekvenciu aplikujeme na všetky modely v slovníku za pomoci Viterbiho algoritmu a ten nám postupne vráti vierohodnosti generovania O jednotlivými modelmi. Model s najväčšou vierohodnosťou predstavuje rozpoznané slovo.

3. Rozpoznávanie reči na embedded systémoch

3.1 DSP

Ako už bolo povedané v úvodnej kapitole, v praxi pri využití menej či viac dokonalých a inteligentných rozpoznávačov reči sa kladie veľký dôraz na to, aby tieto systémy boli použiteľné na takých výpočtových architektúrach, ktoré majú väčšinou do určitej miery obmedzené, jak hardwarové tak i softwarové prostriedky. Hovoríme teda o takzvaných embedded systémoch. Do skupiny embedded systémov môžeme teda zaradiť väčšinu spotrebnej elektroniky, ktorá je riadená rôznymi mikroprocesormi, mikrokontrolérmi či inými riadiacimi výpočtovými jednotkami. Digitálne signálové procesory nevynímajúc.

Procesory ktoré sú navrhované na digitálne spracovanie signálov (DSP) sa vyznačujú určitými vlastnosťami, ktoré na jednej strane ostatné procesory postrádajú, ale naopak nedisponujú niektorými vlastnosťami, ktoré sú samozrejmosťou na iných HW architektúrach. Preto je vždy dôležité zodpovedať si na základné otázky týkajúce sa požadovaného výkonu, HW a SW prostriedkov, fyzických vlastností, a v neposlednom rade aj otázky ohľadne obstarávacích a prevádzkových nákladov. Po zvážení všetkých týchto faktorov by sme mali vybrať vhodnú hardwarovú architektúru pre náš systém.

Podobnými úvahami som sa dopracoval k odpovedi na otázku, aký čip použiť na implementáciu systému pre rozpoznávanie reči. Digitálny signálový procesor splňuje všetky požiadavky systému, ktorý by mal byť schopný poskytnúť potrebné zdroje pre rozpoznávač reči. Výpočtový výkon je síce o niečo menší ako poskytujú iné procesorové jednotky, avšak keďže je sústredený len na výpočty týkajúce sa rozpoznávania, je tento výkon dostačujúci. Po pripojení externej pamäte RAM odpadá aj problém nedostatku pamäti. Avšak najväčšou výhodou tohto riešenia sú rozmery takto koncipovaného systému, ktoré by nemali presiahnuť rozmery napríklad bežne používaného mobilného telefónu.

Systém rozpoznávania reči bol v tomto projekte implementovaný na vývojový KIT DSK TMS320C6416 osadený DSP procesorom C6416 od firmy Texas Instruments. Vývojový Kit obsahuje všetky potrebné prostriedky ako pamäť, audio kodek, rozhranie k pripojeniu k PC a iné.

3.2 Architektúra TMS320-C64x

DSP procesory z rady TMS320 sa delia na fix-point, floating-point a multiprocesorové signálové procesory. Ich architektúra je navrhnutá špeciálne pre real-timeové signálové spracovanie.

TMS320 procesory:

- floating-point DSPs: C3x, C4x
- multiprocessor DSPs: C8x
- fix-point DSPs: C1x, C2x, C5x, C6x

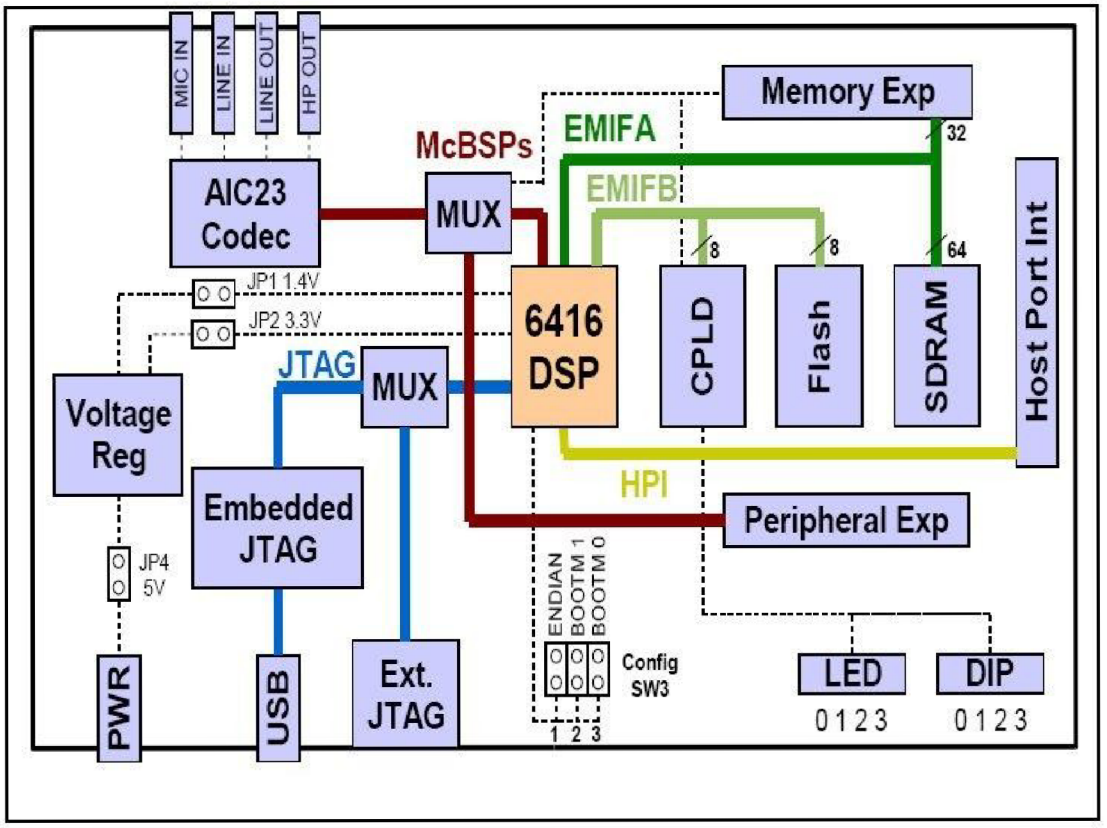
Ako už z názvu vyplýva, KIT ktorý bol použitý pre rozpoznávač reči je osadený procesorom z rady C64x. Tento procesor patrí do skupiny fix-pointových procesorov. Obsahuje 64, voľne použiteľných, všeobecných 32-bitových registrov, dvojúrovňovú cache pamäť (32KB L1, 1024KB L2), 8 funkčných jednotiek: 2 násobičky, 6 aritmeticko-logických jednotiek. Avšak i napriek tomu, že CPU postráda floating-point výpočtové jednotky, táto architektúra sa s floating-point číslami dokáže vysporiadať a to tak, že floating-point operácie sa emulujú. Samozrejme, že efektívnosť výpočtov týmto pádom klesá, avšak pri neveľkom využívaní tejto emulácie je pokles výkonnosti zanedbateľný. Tento VLIW CPU je schopný vykonávať až 8 32-bitových inštrukcií v jednom cykle. Podporuje 8/16/32 bitové dátové typy čo zvyšuje efektívnosť využitia pamäti. 40-bitové aritmetické operácie zaručujú väčšiu presnosť výpočtov. 32-bitový adresový priestor dovoľuje adresovať internú pamäť, ktorá má oddelený programový a dátový priestor. Pri použití externej pamäte je adresový priestor zjednotený spolu s internou pamäťou za pomoci External Memory InterFace (EMIF). EMIF podporuje externé SDRAM, SBSRAM, SRAM. K dispozícii je ďalej DMA radič, ktorý má 4 programovateľné kanály. Funkcionalitu DMA radiča poskytuje aj dokonalejší EDMA radič, ktorý má dokopy 16 kanálov a aj pamäťový priestor pre uloženie rôznych konfigurácií DMA prenosov pre neskoršie použitie. Komunikácia s okolím prebieha cez paralelný port HPI, alebo multikanálový sériový port McBSP. Vo výbave nájdeme aj 3 32-bitové časovače, ktoré slúžia napr. na časovanie udalostí, generovanie pulzov, generovanie prerušení, či synchronizačné účely DMA/EDMA radičov.

3.3 Vývojový KIT DSK TMS320C6416

Vývojový KIT DSK TMS3206416 od firmy Texas Instruments (ďalej len DSK) je zariadenie určené na edukačné a vývojové účely. Pre širšie zavedenie do praxe nieje primárne určený a preto sa s ním v žiadnom komerčnom produkte nestretneme. Svoje uplatnenie avšak nachádza na samom začiatku procesu vývoja produktov na báze embedded systémov – vývoj, development. Po ukončení vývoja aplikácie na vývojovom KITe, sa výrobcovi zašle špecifikácia zariadenia, ktoré by malo byť osadené

DSP procesorom a ďalšími potrebnými prostriedkami aké boli použité na vývojovom KITe. Takto vyrobené špecifické zariadenie bude určené na ďalšie komerčné, alebo iné využitie.

Pozrime sa teda na blokové schéma a prostriedky, ktoré nám táto architektúra poskytuje.



Obrázok 3.1 *Blokový diagram DSK C6416*

3.3.1 Hardware resources

DSK obsahuje mnoho „on-board“ zariadení ktoré uspokoja potreby širokého spektra aplikácií, ktoré je možno na DSK vyvíjať.

- DSP procesor TMS320C6416 pracujúci na frekvencii 1000MHz
- stereo audio codec AIC23
- 16MB DRAM
- 512KB non-volatile Flash pamäť
- 4 užívateľom ovládané LED a DIP prepínače
- softwarovo nastaviteľné konfigurácie karty cez registre
- nastaviteľné boot-ovanie
- slot na zapojenie prídavných kariet
- JTAG emulácia pre UBS port
- zdroj napájania +5V

DSP CPU a ostatné on-board periférie sú prepojené cez dve zbernice. 64-bitovú EMIFA a 8-bitovú EMIFB. AIC23 kodek dovoľuje DSK prijímať a odosielať analógové signály. McBSP1 zbernica zabezpečuje nastavenie kodeku a McBSP2 slúži na prenos dát medzi kodekom a DSP. CPLD (programmable logic device) zabezpečuje zjednodušenie použitia jednotlivých komponentov na DSK. Napríklad použitím CPLD registrov, ktorými sa nastavujú jednotlivé zariadenia. 4 LED a 4 DIP prepínače poskytujú užívateľovi spätnú odozvu. CodeComposerStudio (vývojové prostredie, ďalej len CCS) komunikuje s DSK cez JTAG emulator s USB rozhraním.

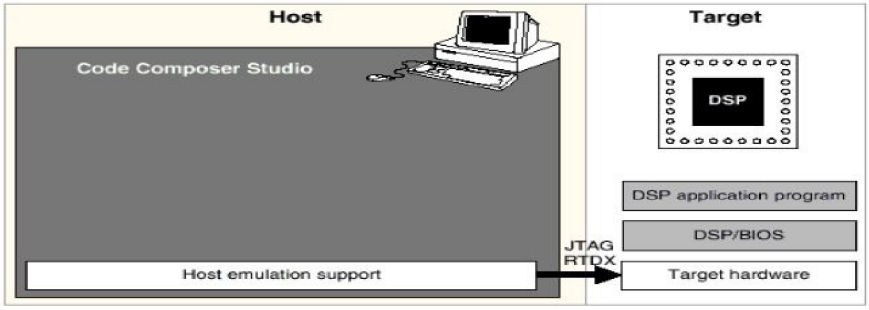
3.3.2 Software resources

DSK poskytuje okrem rôznorodého hardwarového vybavenia aj softwarové prostriedky, ktoré spolu vytvárajú veľmi silný nástroj pre vývoj aplikácií na DSP. Dvoma základnými stavebnými prvkami sú DSP/BIOS a Code Composer Studio.

DSP/BIOS je vo svojej podstate operačný systém bežiaci na strane DSK. Tak isto ako bežné operačné systémy pre PC, aj DSP/BIOS poskytuje programátorovi, resp. aplikácii bežiacej na DSK, služby ktoré sú dostupné cez API. Ďalej okrem iného zabezpečuje real-time scheduling, real-time analýzu (RTA), real-time data exchange (RTDX) a takisto zabezpečuje jednoduché statické nastavenie systém za pomoci grafického užívateľského rozhrania CCS a konfiguračného súboru.

Code Composer Studio je na druhú stranu aplikácia bežiaca na strane PC, ktorá priamo zabezpečuje komunikáciu s DSK, či už pri statickom nastavení systému alebo pri real-time komunikácií počas behu aplikácie.

Schematicky by sa vzťah týchto dvoch aplikačných prvkov dal znázorniť nasledovne:



Obrázok 3.2 *Host-Target connection*

3.4 Code Composer Studio

Ako už bolo v predchádzajúcej časti vysvetlené, integrované vývojové prostredie Code Composer Studio (CCS) patrí k základnému softwarovému vybaveniu DSK. Tak isto ako samotný DSK aj CCS je produktom firmy Texas Instruments. Aplikácia je vytvorená iba pre platformu MS Windows, takže vývoj aplikácií pre DSK sa obmedzuje len na túto jednu platformu. Na druhú stranu toto vývojové prostredie poskytuje všetky potrebné nástroje, ktoré sú pri vývoji aplikácie nevyhnutné. To znamená, že vo všetkých fázach vývojového cyklu aplikácie si vystačíme s CCS. Prejdime si teda najdôležitejšie časti Studia.

3.4.1 Project Manager

V prvom rade sa jedná o nástroje správy projektov, ktoré sú pre programátora asi najdôležitejšou časťou. Ľavá strana aplikacie je vyhradená pre hlavnú lištu, v ktorej sú logicky usporiadané jednotlivé súbory projektov (súborovú štruktúru projektov vo všeobecnosti si vysvetlíme neskôr). Najväčší priestor je pochopiteľne venovaný samotnému editoru zdrojových kódov. Tento editor je plne integrovaný do prostredia a interaktívne spolupracuje s ostatnými nástrojmi Studia. V spolupráci s prekladačom okamžite dokáže zobrabraziť inštrukcie assembleru tak ako sú generované pre každý riadok zdrojového kódu a tým dovolí programátorovi pochopiť a v prípade potreby optimalizovať beh programu na DSK. V prípade vývoja aplikácie je neoceniteľnou výhodou spolupráca editora s debugovacími nástrojmi vo forme nastavovania *breakpointov* a *probepointov*.

3.4.2 Kompilátor

V tomto momente teda vieme spravovať projekty a editovať zdrojové kódy. Ďaľšie nástroje ktoré teraz bezpochyby potrebujeme sú nástroje generovania kódu. Základ tvorí C/C++ kompilátor, ktorý prekladá štandartné ANSI/ISO C/C++ zdrojové kódy na inštrukcie assembleru optimalizované na konkrétnu platformu DSP procesora osadeného na použitom DSK. Optimalizované inštrukcie assembleru sú ďalej posunuté assembleru, ktorý preloží tieto inštrukcie na strojový kód a vytvorí objektové súbory COFF(Common Object File Format). V poslednej fáze prichádza na rad samozrejme linker, ktorý vezme jednotlivé COFF súbory a zlinkuje ich do výsledného COFF súboru so strojovým kódom spustiteľným na DSK. Tento výsledný súbor sa nahrá do pamäte DSK a aplikácia je pripravená na spustenie.

3.4.3 Debugging

Proces vývoja aplikácie môže byť výrazne uľahčený v prípade, ak má programátor k dispozícii kvalitné nástroje určené na debugging. CCS ich má hneď niekoľko. V prvom rade sú to *Breakpointy*, ktoré po prerušení programu dovoľujú skúmať jeho stav. O aktuálnom obsahu lokálnych a globálnych

premenných, poprípade C/C++ výrazov nás informuje tzv. *Watch Window*.

Pri programovaní nám neraz príde vhod možnosť nejakým spôsobom zaznamenavať priebeh udalostí do LOG súboru. Pri klasickom programovaní na PC je najjednoduchším spôsobom použiť *stdout*, *stderr* (v prípade jazyka C). Pochopiteľne, že v prípade DSK, ktorý priamo nieje pripojený k žiadnemu zobrazovaciemu periférnemu zariadeniu potrebujeme iný mechanizmus. K tomuto účelu slúži tzv. Message LOG. V prvom rade si musíme nadefinovať jednotlivé Message Log objekty na strane CCS. V priebehu programu sa potom na tieto objekty posielajú funkciou `LOG_printf()` formátované výstupy. Toto je plne v réžii DSP/BIOSu, ktorý komunikačným kanálom napojeným na CCS posiela dáta do logovacích objektov. Logovanie má v rámci DSP/BIOSu najnižšiu prioritu, takže najmä pri real-time aplikáciách sa môže stať, že do logovacích objektov sa kvôli prísnym deadline-om, ktoré sú kladené na procesy s vyššou prioritou, nestihnú niektoré dáta ani poslať a tým vzniká vo výsledných logovacích dátach akési vákuum, ktoré rozhodne nieje žiadané a dokonca nás môže občas aj pomýliť. Z tohto poznatku teda vyplýva, že Message Log by sa mal používať len pri offline spracovaní údajov, kde nehrozia žiadne real-time deadlines.

Ďalším užitočným nástrojom sú tzv. ProbePoints. Tieto majú obdobnú funkciu ako BreakPoints s tým rozdieľom, že ProbePoints sa používajú na prerušenie behu programu za účelom načítania vstupných dát zo súboru na PC, zápisu výstupných dát do súboru na PC, prípadne k aktualizácii CCS okien zobrazujúcich stav programu. V mieste kde chceme načítať vstupné dáta nastavíme ProbePoint, vytvoríme referenciu na vstupný súbor a odkaz na miesto v pamäti kam sa majú dáta uložiť a všetko ostatné je už v réžii DSP/BIOSu resp. CCS. Podobný postup platí aj pre zápis do súboru.

Keďže vývojový KIT s DSP procesorom poskytuje len obmedzené prostriedky, veľmi dôležitým faktorom je efektivita vyvíjaných aplikácií. Optimalizačné procesy sú vo všeobecnosti veľmi zložitým problémom a v prípade aplikácií pre DSP to platí dvojnásobne. V CCS je pre tieto účely určený optimalizačný nástroj Profiler, ktorý programátorovi podáva dôležité informácie o tom, aký dlhý čas strávili jednotlivé procesy v kritických miestach, ktoré pamäťové segmenty boli použité a veľa ďalších informácií na základe ktorých dokážeme určiť problematické časti kódu a prípadne zvoliť vhodnú optimalizačnú metódu na zefektívnenie výpočetných procesov.

3.4.4 Štruktúra programu na DSK

Na tomto mieste by sme si povedali niečo bližšie o samotnej štruktúre programu vyvýjaného v CCS. DSK disponuje mnohými hardwarovými prostriedkami a zariadeniami, ktoré je potrebné nejaký spôsobom nastavovať, riadiť a ovládať tak, aby bola dosiahnutá požadovaná činnosť. Najprirodzenejším spôsobom je pochopiteľne použitie API funkcií, ktoré sú súčasťou DSP/BIOSu. V priebehu programu sa za pomoci API dynamicky nastavujú riadiace registre jednotlivých zariadení a tým sa dosahuje požadované správanie. K tomu aby sme mohli začať používať jednotlivé API, program musí mať k dispozícií konfiguračný súbor.

Prvou základnou súčasťou programu je teda konfiguračný súbor s príponou *.cdb*. V tomto súbore sa definujú jednotlivé objekty a vlastnosti objektov, ktoré bude aplikácia používať. CCS poskytuje nástroj na tvorbu konfiguračných súborov (`FILE→NEW→DSP/BIOS Configuration`). Konfigurácia je rozdelená do jednotlivých sekcií:

- ***System*** - základné systémové nastavenia, rozdelenie pamäťových sekcií (FLASH, ISRAM, SDRAM)
- ***Instrumentation*** - nastavenia LOG objektov, ...
- ***Scheduling*** - nastavenia hodinového kmitočtu, HW a SW prerušení,..
- ***Synchronization***
- ***Input/Output*** - nastavenia vstup/výstupných prostriedkov
- ***Chip Support Library*** - nastavenia špecifických zariadení, ktoré sú na DSK k dispozícii

Po vytvorení konfiguračného súboru *program.cdb* a nastavení všetkých potrebných atribútov sa automaticky vygenerujú nasledovné súbory:

- *programcfg.cmd* – príkazy adresované Linker-u
- *programcfg.h* – deklarácia externých premenných a objektov definovaných v konfiguračnom súbore
- *programcfg.s62* – nastavenia DSP/BIOSu napísané inštrukciami Assembleru
- *programcfg.h62* – hlavičkový súbor k *programcfg.s62*

Z vygenerovaných súborov je pre nás najdôležitejší súbor *programcfg.h*, pretože práve na tomto mieste sa nachádzajú hlavičkové súbory jednotlivých modulov (napr.: *prd.h*, *swi.h*, *tsk.h*, ...) a deklarácie externých premenných a objektov, ktoré sme si nadefinovali v konfiguračnom súbore. V zdrojových kódoch aplikácie teda na každom mieste kde budeme používať tieto premenné a objekty musíme vložiť konfiguračný hlavičkový súbor *programcfg.h*.

Teraz keď máme nadefinovaný konfiguračný súbor (hoci aj s implicitne danými hodnotami) možeme začať používať tzv. Chip Support Library (CSL). CSL je súbor špecifických API funkcií pre konkrétnu čipovú súpravu osadenú na DSK (v našom prípade C6416), ktoré konfigurujú a ovladajú periférie na DSK. Týmto získavame akúsi abstrakciu nad HW a odbremeňujeme sa od nízko úrovňových operácií nad hardwarovými prostriedkami. Pomocou CSL dokážeme jednoducho ovládať napr. DMA resp. EDMA, McBSP, IRQ, časovače a mnoho ďalších zariadení.

Ďalšou dôležitou súčasťou programu je tzv. Board Support Library (BSL). Táto knižnica patrí tak isto k programovému vybaveniu DSK a rozširuje súbor CSL API funkcií o ďalšie rozhrania, tentokrát špecifické pre konkrétny DSK (bez ohľadu na použitú čipovú súpravu). V našom prípade sa jedná o knižnicu *dsk6416bsl.lib*, ktorá nám na vývojovej doske DSK sprístupňuje okrem iného API funkcie pre LED, DIP prepínače, FLASH pamäť, AIC23 kodek a iné ...

Na tomto mieste sa dostávame k poslednej súčasti programu a tou sú bezpochyby samotné zdrojové kódy produkované programátorom. Ako už bolo spomenuté, programovanie embedded

systémov sa do istej miery líši od klasického programovania na PC a to hlavne z toho dôvodu, že nemáme k dispozícii také prostriedky a výpočetný výkon aké nám poskytujú bežné počítače. V našom prípade sa napríklad musíme zaobísť bez floating-point výpočetných jednotiek. To znamená, že všetky operácie nad reálnymi číslami sa musia emulovať a ako vieme takáto emulácia nieje optimálnym riešením čo sa týka rýchlosti výpočtov. Aby sme sa do určitej miery mohli vyhnúť problémom s výkonom, k dispozícii máme knižnicu užitočných algoritmov, ktoré sú napísané v assembleri a sú optimalizované pre konkrétny DSP čip tak, aby poskytovali čo najlepší výkon pri zložitých výpočtoch. V tejto knižnici (*dsp64x.lib*) nájdeme okrem iného algoritmy na výpočet FFT, FIR a IIR filtrov, operácie nad vektormi a maticami, výpočet autokorelácie, atď...

Týmto výčtom boli zhrnuté všetky najdôležitejšie časti programu, ktoré sú nepostrádateľné pri vývoji efektívnych aplikácií pre DSK.

4. Návrh rozpoznávania na DSP

V tejto kapitole si povieme niečo o návrhu vhodného systému rozpoznávania reči na DSP, z akých častí sa tento systém bude skladať a akú funkciu bude v konečnom dôsledku vykonávať. Zameriame sa skôr na spôsob akým dosiahnuť požadovanú kvalitu rozpoznávania, opis vlastností a algoritmov použitých v jednotlivých častiach a tak isto sa zameriame aj na popis samotného procesu rozpoznávania, od spôsobu prijatia vstupných signálov až po následnú odozvu systému. Detaily realizácie a implementácie rozpoznávania si preberieme v kapitole 5.

4.1 Popis použitého rozpoznávača

Začnime tak trocha od konca a povedzme si najskôr aký druh rozpoznávania reči bude na embedded systéme implementovaný. Zmyslom tejto práce je akýmsi spôsobom preskúmať a získať nejaké východiskové pozície v problematike rozpoznávania reči na embedded systémoch, s dôrazom na možnosti využitia architektúr, ktoré postrádajú výpočetné jednotky s pohyblivou rádovou čiarkou. Z tohto teda vyplýva, že cieľom nieje vytvoriť sofistikovaný rozpoznávač, ktorý dokážeme vtestnať do zariadenia malých rozmerov. Cieľom by mal byť jednoduchší rozpoznávač, na ktorom by sme mohli skúmať možnosti a odhalovať limity, ktoré nám HW architektúry embedded systémov stavajú. Na základe výsledkov dosiahnutých takýmto rozpoznávačom by sme potom ďalej mohli odhadnúť možnosti realizácie napr. už spomínaného sofistikovaného rozpoznávača.

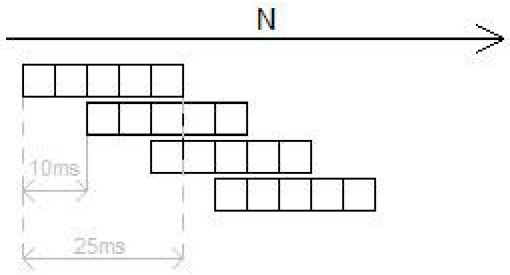
Po zvážení týchto faktov sme sa rozhodli pre systém, ktorý by mal byť schopný rozpoznávať vo vstupnom rečovom signále číslovky v českom jazyku. Veľkou výhodou HW architektúry DSK je, že táto disponuje vlastným audio kodekom. Po pripojení mikrofónu (príp. iného zvukového zdroja) na vstup audio kodeku a následnej reprodukcii rečového signálu, systém bude musieť tento signál zachytiť, spracovať ho a spätnou odozvou informovať užívateľa o výsledku rozpoznávania. Presný popis mechanizmu ovládania systému z užívateľského pohľadu preberieme v podkapitole 4.2.2.

Prejdime teda k samotnému návrhu. V teoretickej časti tejto práce na obrázku 2.1 (str.13) sme si uviedli, že systém rozpoznávania reči sa dá rozdeliť do troch základných blokov. V prvom rade to je parametrizácia, ktorá nám z rečového signálu extrahuje rozhodujúce parametre a posunie ich k ďalšiemu spracovaniu. Nasleduje samotné rozpoznávanie, ktoré úzko spolupracuje s poslednou časťou systému – slovníkom rozpoznávaných slov. V nasledujúcich podkapitolách si podrobne popíšeme zvolené metódy, algoritmy a vlastnosti jednotlivých častí systému.

4.1.1 Parametrizácia (Feature Extraction)

Z našich doterajších poznatkov vieme, že existujú rôzne prístupy k parametrizácii rečových signálov, tzv. Feature Extraction. V praxi sa vo veľkej väčšine používajú systémy rozpoznávania reči, ktoré sú založené na Melfrekvenčných cepstrálnych koeficientoch (MFCC). V našej aplikácii sme sa preto tiež rozhodli pre použitie práve MFCC.

Na vstup je privádzaný rečový signál, ktorý je vzorkovaný na frekvencii 8kHz. Táto vzorkovacia frekvencia nieje pre potreby rozpoznávania reči najideálnejšia, avšak postačujúca. Frekvencia 8kHz bola zvolená z čisto praktických dôvodov a to preto, lebo referenčné akustické vzorky nad ktorými boli modely rozpoznávaných slov trénované, mali vzorkovaciu frekvenciu práve 8kHz (viac sa o modeloch a trénovaní dozvieme v následujúcich podkapitolách). Ako už vieme, rečový signál je vo všeobecnosti považovaný za náhodný a preto pri parametrizácií potrebujeme v prvom rade získať úseky, resp. rámce signálu, ktoré by sme mohli považovať za stacionárne. Zvolená bola, v praxi bežne používaná, štandartná dĺžka rámcov 25ms, uchovávajúca rečový signál, ktorý má vo všeobecnosti nemenný charakter. Pri 8kHz je to presne 200 vzoriek na jeden rámec. Ďalej bol medzi jedenotlivými rámcami zvolený posuv 10ms (tj. 80 vzoriek). Z tohto vyplýva, že susedné rámce majú 160 spoločných vzoriek.



Obrázok 4.1 *Rámce*

Na jednotlivé rámce je následne aplikovaná okienková funkcia (window function):

$$w[n] = \begin{cases} 0.54 - 0.46\cos\dfrac{2\pi n}{200-1} & \text{pro} \quad 0 \leq n \leq 200-1 \\ 0 & \text{jinde} \end{cases}$$

Jedná sa teda o Hammingovo okno, ktoré zabezpečí utlmenie signálu na okrajoch rámca.

Z takto pripraveného a predspracovaného rámca nás v nasledujúcom kroku zaujíma frekvenčné spektrum. Najskôr si za pomoci rýchlej Fourierovej transformácie FFT vypočítame spektrum definované komplexnými číslami a následne z prvej polovice spektra (druhá polovica je komplexne združená) vypočítame moduly komplexných čísiel umocnené na druhú. Týmto získavame požadované spektrum. V ďalšom kroku nasleduje aplikácia banky 23 filtrov. Výpočet a parametre jednotlivých Mel-filtrov možno nájsť v [4]. Dôvody použitia týchto nelineárnych filtrov sme si už vysvetlili v teoretickej časti. Po aplikácii filtrov sa dostávame do stavu, keď je jeden 25ms rečový rámec popisovaný 23 koeficientami. K získaniu kompletných MFCC potrebujeme týchto 23 čísiel ešte logaritmovať a aplikovať diskrétnu kosínusovú transformáciu na ktorej výstupe sa nám objavý 13 MFCC parametrov.

4.1.2 Modely rozpoznávaných slov

Aby systém rozpoznávania reči mohol plniť úlohu ktorú od neho očakávame, potrebuje mať informácie o rozpoznávaných slovách. Slovník slov ktorým musí disponovať je vlastne množina modelov reprezentujúcich jednotlivé slová. V závislosti od použitého rozpoznávača existuje viacero spôsobov modelovania požadovaných slov. V našom prípade sme sa rozhodli pre rozpoznávač založený na štatistickom modelovaní za pomoci Hidden Markov Models (HMM).

Rozpoznávané slová sú teda v našej aplikácii reprezentované skrytými Markovými modelmi. Pri tvorbe modelov nebola do úvahy bratá fonémová skladba jednotlivých slov. Dĺžka modelov, resp. počet jednotlivých stavov HMM bol pevne daný, bez ohľadu na počet fonémov vyskytujúcich sa v jednotlivých slovách. Vzhľadom na priemernú dĺžku slov českých čísloviek, ktoré chceme rozpoznávať, bol stanovený počet stavov HMM na 16. K tomu sa pridajú ešte pomocný počiatočný a pomocný koncový stav. HMM reprezentujúce jednotlivé číslovky sú teda definované 18 stavmi.

Ďalším dôležitým rozhodnutím bolo definovanie jednotlivých stavov. Z teoretickej časti vieme, že jednotlivé HMM stavy sú popisované Gaussovskými rozloženiami pravdepodobností. V našom prípade bola zvolená jednoduchšia varianta. To znamená, že stavy sú reprezentované len jedným Gaussovským rozložením pravdepodobnosti. Typicky sa v praxi používa zmes viacerých Gaussovských rozložení na jeden stav, avšak pre naše účely nám jedno rozloženie zatial stačí. Následná zmena algoritmov pri použití zmesy viacerých rozložení by bola relatívne jednoduchá. Detaily týkajúce sa uloženia modelov v systéme si preberieme v kapitole 5.4.5.

4.1.3 Trénovanie modelov

Trénovanie modelov rozpoznávaných slov bolo pochopitelne robené výlučne na strane PC. Samotná implementácia algoritmov trénovania modelov slov nebola v tejto práci zahrnutá. Namiesto toho sa na trénovanie použili overené nástroje HTK, ktoré boli aplikované na rozsiahlu rečovú databázu. Týmto spôsobom sme získali vierohodné slovné modely, ktoré sa pri rozpoznávaní použijú. Avšak v prvom rade bolo treba vyriešiť jeden problém týkajúci sa trénovania. Ten spočíval v samotnom mechanizme tvorby modelov. Na tomto mieste nebudeme rozoberať tento mechanizmus, povieme si len podstatu problému. Problém vzniká hneď na začiatku pri tvorbe MFCC, ktoré budú použité pri trénovaní. Najjednoduchšie by sa zdalo použitie priamo HTK nástrojov parametrizácie. Bol by to overený a zaručený spôsob získania MFCC, avšak tento neprichádza do úvahy. Treba si totiž uvedomiť, že *Feature Extraction* portovaný na embedded systém je do istej miery modifikovaný a prispôsobený na požadovanú architektúru, a tým v konečnom dôsledku produkuje mierne odlišné MFCC koeficienty ako HTK *Feature Extraction*. Rozpoznávanie reči na embedded systéme s HMM modelmi trénovanými nad HTK MFCC koeficientami by dosahovalo velmi zlú kvalitu, prípadne by vôbec ani nefungovalo.

Z tohto dôvodu potrebujeme na strane PC vytvoriť nástroj, ktorý bude produkovať rovnaké MFCC koeficienty ako aplikácia bežiaca na DSP. Na základe algoritmov použitých v aplikácii pre DSP sme teda vytvorili nástroj parametrizácie aj na strane PC (priložený v elektronickej prílohe). Z

dôvodu použitia odlišných HW architektúr a odlišných implementácii funkcií FFT (vlastná implementácia na PC vs. optimalizovaná implementácia z knižnice *dsp64x.lib* na DSP) sme si museli dávať pozor na drobné nepresnosti vznikajúce vo výsledkoch. Aplikáciou DCT v procese parametrizácie sme však nepresnosti vo výsledných koeficientoch posunuli až za 3-4 desatinné miesto, čo považujeme za dostatočné. V prípade portovania iných algoritmov z DSP na PC musíme v budúcnosti brať do úvahy vznik možných nepresností. Tento nástroj bol následne použitý pri trénovaní modelov, ktoré budú týmto pádom vierohodné a budú poskytovať základ pre úspešné rozpoznávanie zadaných slov.

4.1.4 Mechanizmus rozpoznávania

Ako už bolo v predchádzajúcom odstavci uvedené, systém rozpoznávania reči, ktorý sme sa rozhodli implementovať na DSK je založený na štatistickom modelovaní za pomoci Hidden Markov Models. Základ tvoria modely slov uložené v slovníku a samotné rozpoznávanie spočíva v nájdení naväčšej možnej vierohodnosti generovania vstupného rečového signálu daným modelom slova. Model s najväčšou vierohodnosťou nám určuje rozpoznané slovo.

Na celý tento proces rozpoznávania nám postačí Viterbiho algoritmus. Najvhodnejším spôsobom sa javí použitie takej modifikácie Viterbiho algoritmu, ktorý by bol schopný zvládnuť *online* rozpoznávanie. Jedným vstupom algoritmu budú postupne MFCC koeficienty jednotlivých rámcov signálu a druhým vstupom HMM reprezentujúci model slova. Po získaní MFCC z rámca sa tieto koeficienty Viterbiho algoritmom postupne aplikujú na všetky modely v slovníku a priebežne sa získané výsledky budú ukladať. Po tom ako vo vstupnom signále identifikujeme koniec reči, vyhodnotíme dosiahnuté výsledky a určíme model s najväčšou vierohodnosťou.

Asi najdôležitejšou časťou Viterbiho algoritmu je výpočet hodnoty funkcie hustoty rozdelenia pravdepodobnosti (tzv. vysielacia pravdepodobnosť) pre jednotlivé stavy HMM, ktoré sú v našom prípade popisované 13-rozmernými Gaussovskými rozloženiami. V teoretickej časti sme si uviedli vzorec (2.1.) na výpočet funkcie hustoty rozdelenia $b_j[o(t)]$. V našom prípade sa vo vzorci hodnota P rovná počtu MFCC koeficientov, tj. 13. Pre zjednodušenie výpočtov používame vo Viterbiho algoritme logaritmy jednotlivých hodnôt. Preveďme si teda výpočet $\log(b_j[o(t)])$ do zrozumiteľnej formy, z ktorej budeme môcť jednoducho vytvoriť algoritmus.

$$\log\ b_j[o(t)] = \log \prod_{i=1}^{P} \mathbb{N}(o(t); \mu_{ji}; \sigma_{ji}) = \log \prod_{i=1}^{P} \frac{1}{\sigma_{ji}\sqrt{2\pi}} \cdot e^{-\frac{[o(t)-\mu_{ji}]^2}{2\sigma_{ji}^2}} =$$

$$= \log\left[\prod_{i=1}^{P} \frac{1}{\sigma_{ji}\sqrt{2\pi}} \times \prod_{i=1}^{P} e^{-\frac{[o(t)-\mu_{ji}]^2}{2\sigma_{ji}^2}}\right] = \log \prod_{i=1}^{P} \frac{1}{\sigma_{ii}\sqrt{2\pi}} + \log \prod_{i=1}^{P} e^{-\frac{[o(t)-\mu_{ji}]^2}{2\sigma_{ji}^2}}$$

Jednotlivé logaritmy tohto súčtu možeme ďalej zjednodušiť na ...

$$\log \prod_{i=1}^{P} \frac{1}{\sigma_{ii}\sqrt{2\pi}} = \log \frac{1}{\prod_{i=1}^{P} \sigma_{ji} \cdot (2\pi)^{\frac{P}{2}}} = -\log\left(\prod_{i=1}^{P} \sigma_{ji} \cdot (2\pi)^{\frac{P}{2}} \right) =$$

$$= -\frac{1}{2}\log\left(\prod_{i=1}^{P} \sigma_{ji} \cdot (2\pi)^{P} \right)$$

$$\log \prod_{i=1}^{P} e^{-\frac{[o(t)-\mu_{ji}]^2}{2\sigma_{ji}^2}} = \sum_{i=1}^{P} \log\left(e^{-\frac{[o(t)-\mu_{ji}]^2}{2\sigma_{ji}^2}} \right) = \sum_{i=1}^{P} -\frac{[o(t)-\mu_{ji}]^2}{2\sigma_{ji}^2} =$$

$$= -\frac{1}{2}\sum_{i=1}^{P}\left(\frac{o(t)-\mu_{ji}}{\sigma_{ji}} \right)^2$$

... a výsledkom teda je:

$$\log\ b_j[o(t)] = -\frac{1}{2}\log\left(\prod_{i=1}^{P} \sigma_{ji} \cdot (2\pi)^{P} \right) - \frac{1}{2}\sum_{i=1}^{P}\left(\frac{o(t)-\mu_{ji}}{\sigma_{ji}} \right)^2 =$$

$$= -\frac{1}{2}\left(\log\left(\prod_{i=1}^{P} \sigma_{ji} \cdot (2\pi)^{P} \right) + \sum_{i=1}^{P}\left(\frac{o(t)-\mu_{ji}}{\sigma_{ji}} \right)^2 \right) \qquad (4.1.)$$

Pri pohľade na rovnicu zistíme, že jedinou premennou hodnotou je *o(t)* – vektor 13 MFCC koeficientov. Naskytuje sa nám teda možnosť optimalizácie výpočtu a to tým, že si dopredu vypočítame konštanty, ktoré budú nemenné. Zo zadaných stredných hodnôt $\boldsymbol{\mu}$ a rozptylu $\boldsymbol{\sigma}$, ktorými sú

definované Gaussovské rozloženia, môžeme z predošlej rovnice dopredu vypočítať hodnotu

$$\log\left(\prod_{i=1}^{P}\sigma_{ji}\cdot(2\pi)^{P}\right) \qquad (4.2.)$$

pre každý stav HMM, resp. každé Gauss. rozloženie. Táto hodnota sa zvykne označovať aj ako ***gConst***. Veľkou výhodou dopredného vypočítania tejto konštanty pre jednotlivé stavy je najmä to, že sa zbavíme vypočetne náročnej operácie logaritmu, ktorá by nás pri real-time rozpoznávaní mohla do určitej miery spomalovať. Na základe týchto poznatkou sme teda schopný vytvoriť efektívny Viterbiho algoritmus, ktorý si detailne popíšeme v kapitole venovanej implementácii.

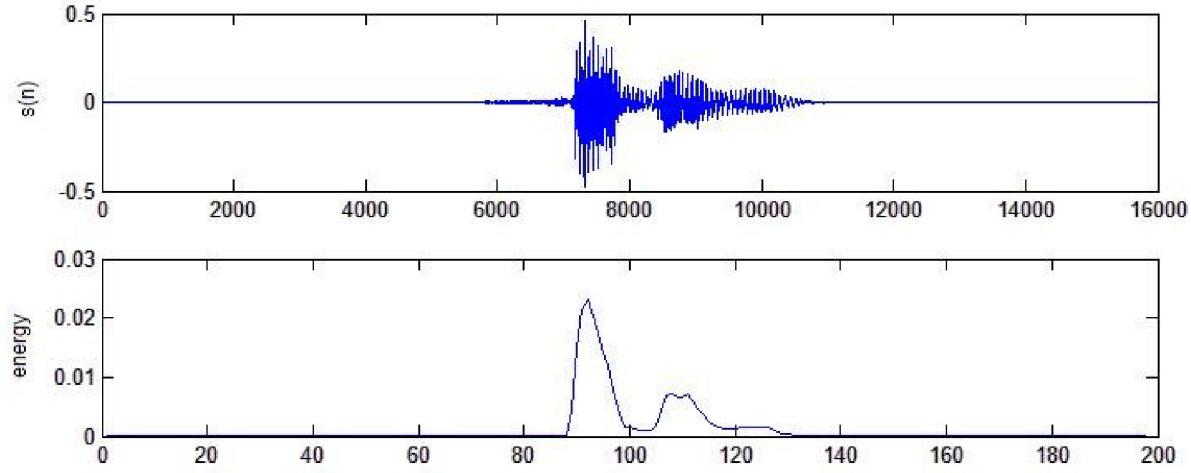
4.2 Aplikácia systému rozpoznávania na DSP

V tejto časti by sme sa pozreli na aplikáciu rozpoznávača z užívateľského hľadiska. Povieme si akým spôsobom bude rozpoznávač fungovať, reagovať na užívateľa, prijímať rečové signály a akým spôsobom bude zase užívateľ prijímať odozvu systému a výsledky rozpoznávania reči.

4.2.1 Princíp fungovania na DSP

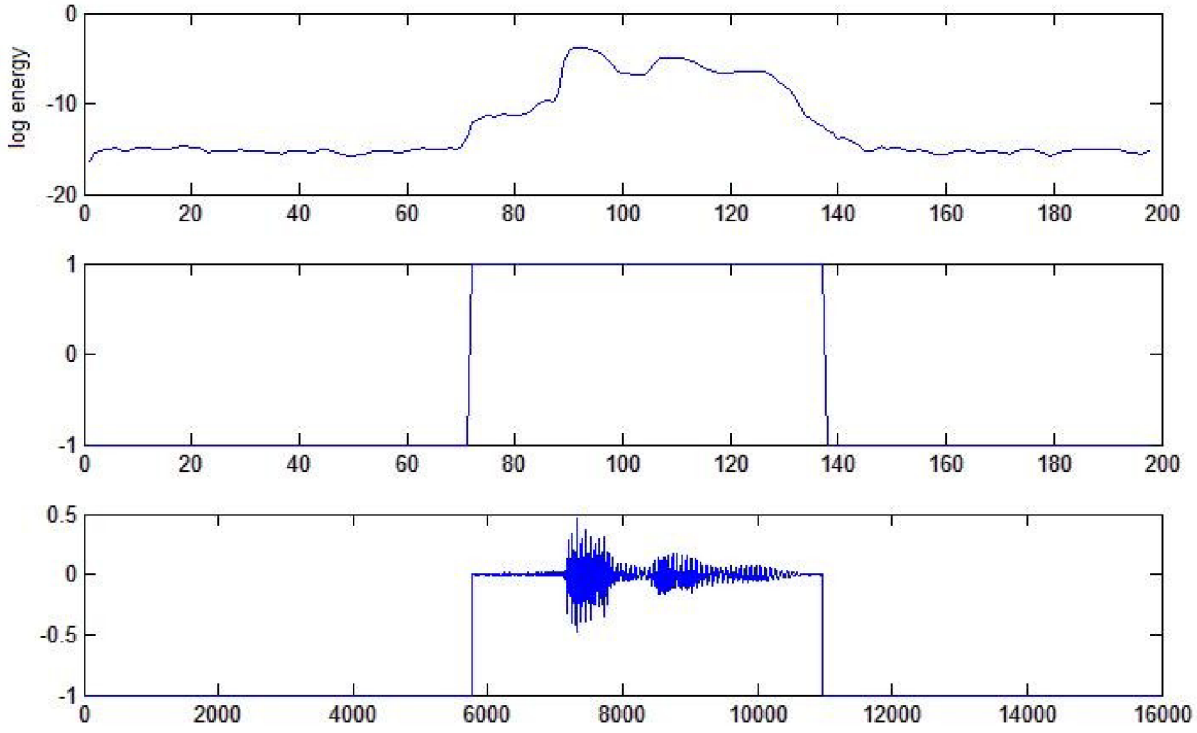
V zásade máme na výber z dvoch princípov prístupu k rozpoznávaniu reči. Jedná sa o tzv. "online" resp. "offline" rozpoznávanie. Pri online rozpoznávaní sa rečový signál okamžite parametrizuje a priebežne sa parametre jednotlivých rámcov posielajú algoritmom rozpoznávania. Výhodou teda je, že výsledky rozpoznávania máme okamžite k dispozícii. Na druhú stranu offline rozpoznávanie nieje také dynamické. Vstupný rečový signál sa najskôr uloží do vyhradenej pamäte, podľa požiadavkov sa zanalyzuje a upraví do požadovanej formy. Následne sa takto upravený signál pošle do rozpoznávača reči. Nevýhodou teda je, že na výsledky rozpoznávania si musíme v závislosti na rýchlosti systému chvíľu počkať.

V našej aplikácii sa vyberieme akousi strednou cestou medzi týmito dvoma princípmi a využijeme mechamizmy používané ako v online tak aj v offline rozpoznávaní. Tento prístup by sa dal nazvať ako "pseudo-online". Z mechanizmov použitých pri offline rozpoznávaní použijeme práve tie, ktoré nám zabezpečujú uloženie vstupného signálu na vyhradené miesto v pamäti. Berúc do úvahy fakt, že rozpoznávať sa budú len české číslovky, priestor na uloženie 4 sekundového rečového signálu nám určite s obrovskou rezervou bude stačiť. Pre 4 sekundový signál vzorkovaný na frekvencii 8kHz a popisovaný 16-bitovými číslami si teda v pamäti potrebujeme alokovať 8000 x 4 x 2B = 64kB. Dôvod prečo si pôvodný signál ukladáme do pamäte je ten, že z tohto signálu si potrebujeme vyseknúť len tú časť, v ktorej sa nachádza hovorená číslovka a bezprostredné úseky ticha, ktoré sa nachádzajú pred a za vyslovenou číslovkou. Ostatné ticho, prípadne ďalšie zvuky sa do rozpoznávača posielať nebudú . Jedná sa teda o detektor rečovej aktivity. Najjednoduchším spôsobom ako detegovať rečovú aktivitu je výpočet strednej krátkodobej energie E, resp. výpočet logaritmu energie log(E). Uveďme si príklad...



Obrázok 4.1 *Číslovka "sedum" a priebeh jej energie*

Po zlogaritmovaní energie si musíme určiť nejaký limit, ktorý nám bude oddelovať reč od ticha. Experimentálne som zistil, že stredná hodnota energie, resp. logaritmu energie, nám dáva hodnotu, ktorá nám pre naše účely dostatočne presne oddelí reč od ticha. V príklade, ktorým si tu ilustrujeme tento mechanizmus nám stredná hodnota logaritmu energie vyšla na lim= -12,6434. Všetky hodnoty energie pod touto úrovňou považujme za ticho a týmto spôsobom získame práve ten úsek signálu, v ktorom sa nachádza vyslovená číslovka.



Obrázok 4.2 *Logaritmus energie, oddelenie reči od ticha a výsledný osekaný signál*

Na tomto mieste treba ešte pripomenúť jednu nutnú modifikáciu tohto algoritmu. Detektor je musíme modifikovať tak, aby bol schopný odhaliť pokles energie pod definovanú hranicu uprostred vyslovovaného slova. Toto sa prejavuje najmä pri číslovke "čtyři", keď pri vyslovovaní písmena "T"

klesá energia hlboko pod definovanú hranicu. Detektor teda musí brať ohľad na bezprostredné okolie poklesu energie a rozhodnúť, či sme sa v skutočnosti dostali na koniec vyslovovaného slova, alebo nastal len dočasný pokles energie z dôvodu vlastností práve vyslovovaných fonémov.

Získali sme teda osekaný rečový signál, ktorý posielame do rozpoznávača. V tomto momente začíname používať mechanizmy a metódy *online* rozpoznávania reči. Simulujeme príchod osekaného rečového signálu presne tak, akoby prichádzal priamo z kodeku, resp. z pamäťového miesta na ktoré bol za pomoci DMA prenosu uložený. Algoritmy rozpoznávania teda signál postupne parametrizujú a vyhodnocujú vierohodnosť generovania jednotlivými modelmi slov uložených v slovníku. Na konci signálu máme okamžite k dispozícii výsledky rozpoznávania. Na základe opísaného mechanizmu rozpoznávania som si dovolil tento princíp nazvať ako "pseudo-online".

4.2.2 Obsluha aplikácie

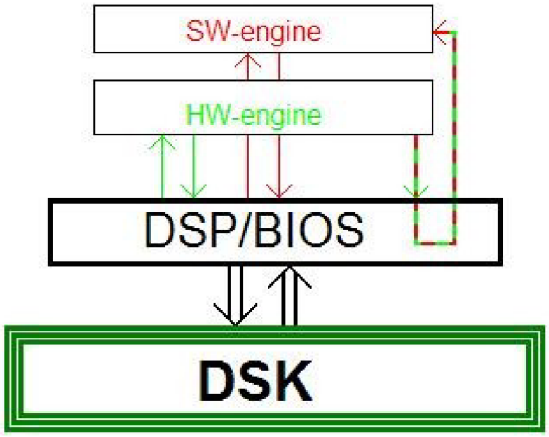
Popíšme si teraz aplikáciu z čisto užívateľského hľadiska. DSK pripojíme pomocou USB na PC a za pomoci CCS do neho nahráme kompletnú aplikáciu rozpoznávania reči. Po nahratí je aplikácia pripravená k použitiu. Zostáva nám už len pripojiť na audio vstup mikrofón a aplikáciu spustiť. Pri používaní mikrofónu treba dať pozor na to, aby sme používali mikrofón s dostatočne silným signálom. Ja som to riešil použitím PC ako zosilovača. Možnosti priamej interakcie užívateľa a DSK sú obmedzené na použitie štyroch LED diód a štyroch DIP prepínačov. Po spustení aplikácie nám systém signalizuje svoju pripravenosť sústavným blikaním všetkých štyroch diód. Tým nám oznamuje, že je pripravený prijímať rečový signál. Ako už vieme, v prvom rade potrebuje do systému nahrať maximálne 4-sekundový rečový signál. Nahrávanie sa zapne v momente stlačenia DIP prepínača číslo 0. Reč sa nahráva až do opätovného uvolnenia DIP prepínača, poprípade sa nahrávanie ukončí aj po uplynutí 4 sekúnd. V tomto momente prichádza na rad spomínané osekanie signálu a samotné rozpoznávanie. Po ukončení rozpoznávania potrebujeme informovať užívateľa o dosiahnutých výsledkoch. Primárne sú k tomuto účelu určené LED diódy. Keďže ich máme k dispozícii iba 4, najvhodnejším spôsobom bude reprezentovať rozpoznané číslo v binárnej forme. V prípade ak máme DSK stále pripojený cez USB k CCS, výsledky rozpoznávania si môže pozrieť aj logovacom okne. V okamihu ohlásenia výsledku je systém opätovne pripravený k ďalšiemu rozpoznávaniu a celý postup môžeme znovu opakovať. V prípade nejakých problémov funguje DIP prepínač číslo 4 ako softwarový reset systému rozpoznávania. Ďalšie detaily si postupne odhalíme v nasledujúcej kapitole o implementácii.

5. Implementácia a testovanie

V tejto kapitole si podrobne prejdeme celú implementáciu aplikácie a vysvetlíme si ako celý rozpoznávač reálne funguje na HW. Od záležitostí týkajúcich sa HW nastavení, cez definovanie použitých dátových typov a inicializáciu dátových štruktúr, až po jednotlivé fázy rozpoznávania. Dôraz sa bude klásť najmä na vysvetlenie použitých dátových typov: fix-point vs. floating-point. Malými kúskami zdrojového kódu, poprípade obrázkami a grafmi, sa posnažíme vysvetliť si mechanizmy fungovania jednotlivých algoritmov. Taktiež si urobíme prehľad ako jednotlivé algoritmy svojím priebehom zaťažujú DSP procesor a v miestach kde sa nám podarilo optimalizačnými metódami dosiahnuť určité zvýšenie efektivity algoritmov si uvedieme postup a mieru zvýšenia dosiahnutej efektivity.

5.1 Architektúra rozpoznávača

Z implementačného hľadiska by sme celý systém rozpoznávania reči mohli rozdeliť na dve horizontálne aplikačné vrstvy ležiace jedna nad druhou. Pomenujme ich výstižne "*HW-engine*" a "*SW-engine*". Už samotné názvy nám prezrádzajú, akú rolu budú jednotlivé vrstvy v systéme zohrávať. Aplikačná vrstva *HW-engine* je umiestnená v spodnej časti systému a zabezpečuje nám správne nastavenie a fungovanie HW prostriedkov a ukladanie dát na vyhradené pamäťové miesta. Vrstva bude poskytovať služby nadradenej aplikačnej vrstve – *SW-engine*. Táto sa nachádza priamo nad *HW-engine* a obsahuje samotné riešenie systému rozpoznávania reči. Dôležitou časťou je spôsob komunikácie medzi týmito dvoma aplikačnými vrstvami, ktorá je vcelku jednoduchá. V skutočnosti pozostáva len z jedného softwarového prerušenia (SW Interrupt) smerujúceho z *HW-engine* do *SW-engine*, ktoré signalizuje príchod nových vstupných dát. Samozrejme, že toto prerušenie sprostredkováva operačný systém DSP/BIOS.



Obrázok 5.1 *Schéma architerktúry systému. SW Interrupt znázornený prerušovanou čiarov.*

5.2 Statické nastavenia systému

Prejdime teda k samotnej implementácii systému. Ako už bolo uvedené v odseku 3.4.4, pred tým ako začneme s programovaním si potrebuje v prvom rade vytvoriť konfiguračný súbor, ktorý obsahuje jednotlivé statické nastavenia HW prostriedkov použitých v aplikácii. V CCS za pomoci integrovaného konfigurátora vytvoríme súbor `main.cdb` a postupne nakonfigurujeme nasledovné časti systému: pamäť (ISRAM,SDRAM), logovacie objekty, PRD (Periodic Function Manager), HWI (HW interrupts) a SWI (SW interrupts).

5.2.1 Pamäť

Nastavenia týkajúce sa použitia pamäte nájdeme v sekcii `SYSTEM→MEM`. Zaujímajú nás len časti ISRAM a SDRAM. (pamäť FLASH v aplikácii nebudeme používať). Konfigurácia pamäte je dosť háklivá záležitosť a aj pri vývoji systému som sa stretol s mnohými problémami a výpadkami, ktoré boli spôsobené prístupom do nesprávne nakonfigurovanej pamäte.

Integrovaná ISRAM má kapacitu 1MB a nachádza sa priamo na čipe. Práve do tejto pamäte sa ukladá program spolu s niektorými dátami, ku ktorým potrebujeme mať rýchly prístup. Fyzický adresový priestor ISRAM je definovaný v rozsahu 0x000000 – 0x100000. Kapacitu užívateľskej časti tejto pamäti si definujme na 0x0B0000, zvyšnú časť využíva DSP/BIOS k vlastným potrebám. K tejto hodnote som sa postupne dopracoval na základe skúseností získaných v priebehu vývoja systému.

Obdobne nastavíme aj externú SDRAM. Táto sa skladá z dvoch 8MB čipov, ktoré sú o niečo pomalšie ako integrovaná ISRAM. Sem budeme ukladať výhradne iba dáta. Nastavením kapacity na hodnotu 0x800000 sprístupníme užívateľovi 8MB pamäte. Nakoniec túto časť pamäťe označme ešte identifikátorom `_SEG1`, ktorým ju budeme môcť neskôr explicitne adresovať.

5.2.2 Logovacie objekty

Sekcia `INSTRUMENTATION→LOG` je určná na definovanie logovacích objektov, na ktoré budeme v priebehu programu posielať dáta. Počas vývoja bola táto sekcia často využívaná, avšak po dokončení logovacie objekty v podstate vôbec nepotrebujeme. Vytvoríme si tu len jeden, nazvaný `results`. V prípade ak budeme mať DSK počas rozpoznávania reči pripojený k PC, resp. CCS, výsledky rozpoznávania budeme posielať práve na tento logovací objekt. V opačnom prípade, ak DSK bude pracovať autonómne, logovací objekt stratí svoj význam.

5.2.3 PRD – Periodic Function Manager

Pri popise obsluhy rozpoznávača sme si povedali, že užívateľ bude rozpoznávač ovládať (iniciovať a ukončievať rozpoznávanie) za pomoci DIP prepínačov. Vlastnosťou DIP však je, že samé

o sebe nedokážu generovať hardwarové ani softwarové prerušenia. Riešením teda je použitie periodickej funkcie, ktorá bude kontrolovať stav prepínačov a následne volať požadované metódy.

V sekcii SCHEDULING→CLK si najskôr nastavíme `timer0` tak, aby generoval prerušenie každých 1000 mikrosekúnd. Následne v sekcii SCHEDULING→PRD vytvoríme položku `checkDipPRD`, ktorá bude vlastne referenciou na samotnú periodickú funkciu `dipCheck()`. Zostáva nám ešte zvoliť vhodnú periódu opakovania funkcie. Interval 50ms sa po odskúšaní aplikácie osvedčil.

5.2.4 Prerušenia – HWI a SWI

Posledná vec ktorú musíme v konfiguračnom súbore nastaviť je registrácia prerušení a vytvorenie referencií na ISR (Interrupt Service Routine), ktoré budú tieto prerušenia obsluhovať. V našom prípade sa v systéme používajú dva druhy prerušení. V prvom rade to sú hardwarové prerušenia vyskytujúce sa medzi HW a aplikačnou vrstvou *HW-engine* a softwarové prerušenia pre komunikáciu medzi jednotlivými aplikačnými vrstvami. Bez ďalších podrobných informácií si uvedieme len samotnú konfiguráciu a implementačné detaily obsluhy prerušení si rozoberieme v podkapitole 5.3.2.

V sekcii SCHEDULING→HWI máme zoznam všetkých možných HW prerušení, ktoré sú implicitne deaktivované. Pre naše účely potrebujeme zaregistrovať `HWI_INT8` – prerušenie EDMA kontroléru, ktorý bude riadiť tok vstupných dát. Funkciu `edmaHWI()` nastavíme ako rutinu obsluhy prerušenia. Ďalej si zaregistrujeme `HWI_INT15` – prerušenie časovača `timer1`, ktoré bude mať menej významnú funkciu – obslužná rutina `timer1Isr()` riadi iba frekvenciu blikania LED.

Nakoniec ešte v sekcii SCHEDULING→SWI vytvoríme softwarové prerušenie `processBufferSwi`. Toto bude generované na úrovni aplikačnej vrstvy *HW-engine* a prostredníctvom funkcie `processBuffer()`, ktorú zaregistrujeme ako obslužnú rutinu tohto prerušenia, bude aplikačnej vrstve *SW-engine* signalizovať pripravenosť vstupných dát k spracovaniu.

5.3 Hardware engine

Na tomto mieste si popíšeme mechanizmy fungovania spodnej aplikačnej vrstvy *HW-engine*. Hlavnou úlohou tejto vrstvy sú inicializačné procedúry, zabezpečenie správneho fungovania hardwarových prostriedkov a v neposlednom rade obsluha HW prerušení generovaných jednotlivými zariadeniami na DSK. Súbor týchto mechanizmov nájdeme v súbore `hw_engine.c`.

5.3.1 Inicializácia HW

Ako sme si už uviedli v predošlých častiach, z HW zariadení budeme v našom systéme využívať časovač, audio kodek, seriové rozhranie McBSP a EDMA kontrolér. Všetky tieto prostriedky musíme v prvom rade inicializovať. Väčšinu zariadení je možné staticky nastaviť aj za pomoci spomínaného konfiguračného súboru, avšak API funkcie a abstraktné rozhrania pre priamy zápis do registrov nám poskytujú mocné nástroje, s ktorými získame absolútnu kontrolu nad HW.

Časovač `timer1`, ktorý nám bude udávať frekvenciu blikania LED, inicializujeme funkciou `init_timer()`. Zápisom hodnoty 12500000 do príslušného registra PRD nám časovač bude generovať prerušenia s periódou 100ms.

Nasleduje nastavenie audio kodeka. V súbore `defs.h` si nadefinujeme konfiguráciu `AIC23_Params` kam zapíšeme všetky požadované nastavenia kodeku. Dôležitými parametrami vzhľadom k rozpoznávaniu reči sú hlasitosť vstupného kanálu a vzorkovacia frekvencia, ktorú nastavíme zápisom do registru `DSK6416_AIC23_SAMPLERATE` na hodnotu 8kHz. Funkcia `AIC23_setParams()` prevedie požadovanú inicializáciu. V prvom rade vytvorí na zbernici McBSP konfiguračný kanál vďaka ktorému následne zapíše definované nastavenia priamo do registrov kodeku.

Pripomeňme, že dátový prenos z kodeku, resp. do kodeku prebieha po zbernici McBSP. Okrem konfiguračného kanálu, ktorý sa vytvorí pri inicializácii kodeku teda potrebuje aj regulárny dátový komunikačný kanál na McBSP, po ktorom budú prichádzať vstupné dáta. Funkcia `init_Mcbsp()` nám tento kanál vytvorí.

Ak by prenos po dátovom komunikačnom kanále mal byť plne v réžii DSP procesoru, zabralo by to veľkú časť jeho prostriedkov. Procesor preto odbremeňuje od tejto práce a činnosť týkajúca sa transferu dát z audio kodeku sa prenecháva DMA kontroléru. Kodek dodáva vstupné dáta konštantnou rýchlosťou 8000 vzoriek za sekundu. Samozrejme že od rozpoznávača požadujeme aby stihol prijať a spracovať všetky vstupné dáta a preto riešením zabezpečenia priebežného spracovania vstupného signálu je použitie dvoch vstupných bufferov, *PING* a *PONG*. Pokým sa jeden buffer plní vstupnými dátami, druhý je spracovávaný (*SW-engine*). Veľkosť bufferov je dimenzovaná na 200 vzoriek. Pri frekvencii 8kHz je teda jeden buffer naplnený za 25ms. Z tohto faktu vyplýva, že aplikácia stihne real-time deadline, len ak spracovanie dát prebehne v tomto časovom horizonte. Funkcia `initEdma()` inicializuje EDMA kontrolér. Vytvorí dve DMA konfigurácie (pre PING a PONG) v ktorých nastavíme adresy bufferov a ich zreťazením v príslušnej ISR sa zabezpečí striedavé napĺňanie bufferov. DMA kanál zaregistrujeme ku generovaniu DMA prerušení (tzn. odštartujeme dátový transfer z kodeku do vstupných buffrov) až potom, keď `dipCheck()` identifikuje stlačenie príslušného DIP prepínača. Podobne DMA prerušenia po uvoľnení DIP odregistrujeme.

Týmto sme uzavreli výčet nevyhnutných inicializačných rutin. Pre hlbšie pochopenie jednotlivých nastavení, sposôbu zápisu do registrov a významu hodnôt zapisovaných do registrov je potrebné si tieto informácie vyhladať v manuáloch [6].

5.3.2 Obsluha prerušení – ISR

Aplikácia používa dve hardwarové prerušenia. Pripomeňme si, že obidve prerušenia sme si už v konfiguračnom súbore zaregistrovali a prideli im obslužné rutiny ISR – Interrupt Service Routine.

Po každom vygenerovanom prerušení časovača `timer1` sa zavolá obslužná rutina `timer1Isr()`. Táto na základe príznakov `led0_on` až `led3_on` zapína, resp. vypína jednotlivé LED diódy danou frekvenciou.

Pre nás je však dôležitejšia obsluha DMA prerušení, ktoré sú generované pri ukončení DMA

prenosu, resp. pri zaplnení jednotlivých vstupných bufferov PING a PONG. Audio kodek vo funkcii A/D prevodníka produkuje na frekvencii 8kHz 16b čísla. Na základe týchto vlastností som nadefinoval vstupné buffre ako statické pole 16-bitových čísiel `Int16` s kapacitou 200 hodnôt:

```
Int16 gBufferRcvPing[200];
Int16 gBufferRcvPong[200];
```

DMA kanály sú teda nasmerované práve na tieto dve pamäťové miesta. Po ich naplnení, tzn. každých 25ms je volaná obsluha prerušenia `edmaHwi()`. Táto následne identifikuje buffer ktorý bol práve naplnený a sama vygeneruje softwarové prerušenie signalizujúce aplikačnej vrstve *SW-engine* pripravenosť vstupných dát k spracovaniu. SW prerušenie je generované funkciou: `SWI_or(&processBufferSwi,{PING|PONG});`. *SW-engine* potom priamo pristupuje k *PING/PONG* bufferom a spracováva vstupný signál.

5.4 Software engine

Algoritmy rozpoznávania reči sú implementované v aplikačnej vrstve *SW-engine*. Popíšme si teda podrobne všetky mechanizmy rozpoznávania od prijatia vstupného rečového signálu až po vyhodnotenie samotných výsledkov, ku ktorým nás jednotlivé algoritmy priviedli. Z návrhu, ktorý bol popísaný vo 4.kapitole si pripomeňme princíp rozpoznávania nazvaný "pseudo online", ktorý rozdeluje systém na časť pracujúcu "offline" (predspracovanie vstupného signálu) a časť pracujúcu "online" (rozpoznávanie). Najskôr začneme pochopiteľne s predspracovaním signálu a pri prechode do "online" časti si jasne definujeme hranicu. Nasledujúce funkcie nájdeme v súbore `sw_engine.c`.

5.4.1 Predspracovanie signálu

Aplikačná vrstva *SW-engine* začína svoju činnosť príchodom prvého softwarového prerušenia a zavolaním obslužnej rutiny `processBuffer()`. Táto na základe príslušného parametru identifikuje buffer obsahujúci vstupné dáta prichádzajúce z audio kodeka. Pri predspracovaní si potrebujeme v prvom rade vstupný signál uložiť na vyhradené miesto v pamäti. Za týmto účelom si vytvoríme ukazateľ `Int16 *audioBuff` na pamäťové miesto v ktorom dynamicky alokujeme priestor pre 4s záznamu, tzn. pre 4*8000 = 32000 16bitových čísiel. Funkcia `save2buff()` nám vstupné dáta z PING a PONG bufferov ukladá do pamäte a v prípade zaplnenia 4 sekundového bufferu automaticky nahrávanie ukončí. Po ukončení nahrávania, ktoré môže byť vyvolané zaplnením buffera, alebo uvoľnením príslušného DIP prepínača užívateľom, musíme najskôr ukončiť DMA prenos zápisom do príslušného miesta `IRQ_mask` a až potom zahájime predspracovanie uloženého signálu funkciou `off_process()`.

Hlavnou úlohou predspracovania signálu je odstránenie nepotrebných úsekov ticha pred a po vyslovení rozpoznávanej číslovky. Táto potreba vychádza zo zvolenej metódy rozpoznávania, kde jednotlivé modely čísloviek (HMM) niesú dynamicky poprepájané na modely ticha, ktoré by na seba dokázali naviazať úseky ticha v signále a tým zabezpečiť presnejšie rozpoznávanie. Tohto ticha sa

teda potrebujeme zbaviť manuálne. Mechanizmus detektora rečovej aktivity sme si už popísali v 4.kapitole, pozrime sa teda len dôležité implementačné časti.

Pôvodný rečový signál, ktorý máme uložený v `audioBuff` si rozdelíme na rámce. Funkcia `framesOffline()`, ktorá nám na tomto mieste delí signál na rámce je takmer identická s funkciou `getFrame()` používanou samotným rozpoznávačom, takže implementačné detaily si uvedieme až neskôr. Následne si pri každom rámci vypočítame logaritmus jeho energie a výsledok uložíme do `frames_energy`. Výpočtom strednej hodnoty energie získame hranicu oddelujúcu rámce ticha od rámcov reči, na základe ktorej môžeme dostatočne presne identifikovať začiatok a tak isto aj koniec rečovej aktivity, resp. vyslovovanej číslovky. Práve na tomto mieste potrebujeme ešte použiť algoritmus, ktorý nám dokáže odhaliť a následne ignorovať pokles energie, spôsobený vlastnosťami vyslovovaných fonémov a nie skutočným koncom rečovej aktivity. Detaily algoritmu nájdeme v **prílohe A**. Premenné `speech_start` a `speech_end` sú teda nástavené na presný začiatok, resp. koniec vyslovovanej číslovky v signále uloženom v `audioBuff`. Fáza predprípravy signálu sa týmto pádom ukončila.

5.4.2 Zahájenie rozpoznávania

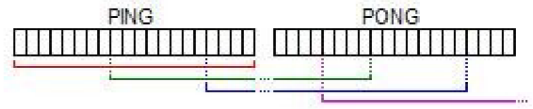
Rozpoznávanú reč máme pripravenú, nastáva zahájenie rozpoznávania. V tomto momente sa presúvame z časti systému, ktorá pracovala *offline,* do *online* časti využívajúcej algoritmy splňujúce podmienky realtime-ového spracovania reči. Zaujímať nás začne dodržiavanie real-time deadlines a preto sa budeme pozerať aj na výpočetný čas strávený v jednotlivých algoritmoch a samozrejme aj na použitie vhodných dátových typov (fix vs. floating -point), ktoré ovplyvňujú rýchlosť.

Prechod z *offline* predprípravy signálu na *online* rozpoznávanie budeme musieť simulovať. Jedná sa o simuláciu DMA kontroléru takým spôsobom, že manuálne budeme zaplňovať PING|PONG buffre pripraveným signálom a následne aplikačnej vrstve *SW-engine* posielať signál oznamujúci pripravenosť vstupného buffera na spracovanie. V tom čase by už *SW-engine* po splnení deadline limitov mal byť pripravený nasledujúci buffer spracovať. Toto všetko sa bude odohrať rovnako ako pri použití skutočného DMA prenosu, tj. každých 25ms.

Z *HW-engine* nám teda prichádzajú prerušenia, resp. simulácia prerušení a sme na začiatku procesu rozpoznávania. Pozrime sa do *Profiler-u* na zaťaženie DSP procesora. Na tomto počiatočnom mieste spotrebúvame 0,25% celkového výkonu procesoru. Z tejto referenčnej hodnoty budeme ďalej určovať zaťaženie pri jednotlivých procesoch.

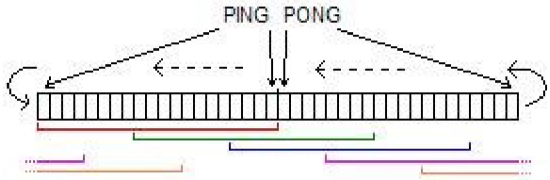
5.4.3 Parametrizácia (Feature Extraction)

Prvým krokom pri rozpoznávaní reči je *Feature Extraction.* Ako už vieme, pri parametrizácii vytvárame rečové rámce, ktoré sa navzájom prekrývajú. Z tohto dôvodu nieje možné jednotlivé rámce extrahovať priamo z *PING* a *PONG* bufferov. Dôvod je ten, že niektoré rámce sa budú nachádzať na hraniciach bufferov a jednotlivé dáta sa budú rozprestierať v obidvoch častiach. Vieme, že pri spracovaní jedného z bufferov sa zároveň druhý napĺňa nasledujúcimi vstupnými dátami a staré dáta sú prepisované. Týmto by sme stratili potrebné informácie.



Obrázok 5.1 Rámcovanie na vstupných bufferoch

Riešením je teda použitie kruhového bufferu dostatočnej veľkosti, z ktorého si pohodlne vyberieme požadovaný rámec bez toho, aby sme ho predtým stihli prepísať vstupnými dátami.



Obrázok 5.2 Rámcovanie na kruhovom bufferi

Kapacita kruhového buffera `disp_buffer` bola zvolená na 400 vzoriek. Na toto miesto dokážeme uložiť dva plné vstupné buffre (*PING,PONG*) a tým odstrániť nežiadúce prepisovanie dát.

Rámce

Po prijatí signálu z aplikačnej vrstvy *HW-engine*, funkcia `getFeatures()` vezme príslušný vstupný buffer a dáta uloží do kruhového buffera. Následne vytvorí z dostupných dát maximálny počet úplných, 25ms (200 vzoriek) dlhých rámcov pomocou funkcie `getFrame()`. Vytváranie rámcov je plne v réžii tejto funkcie. V prípade, že nedokáže vytvoriť nasledujúci rámec z dôvodu nedostatku dát v bufferi, jednoducho si počká na ďalšie softwarové prerušenie signalizujúce pripravenosť nových dát.

Hammingovo okno

Prvú manipuláciu so vstupným signálom na úrovni *SW-engine,* ktorá sa deje práve vo funkcii `getFrame()`, možeme hneď využiť na aplikovanie Hammingova okna. Aby sme sa na tomto mieste zatiaľ vyhli zavedeniu *floating-point* čísiel, Hammingovo okno pri inicializácii normalizujeme na rozsah $0..2^{15}$-1, teda 15-bitové celé čísla.

```
frame[n]=( disp_buffer[framePos] * hamming[n] ) >> 15;
```

Násobením jednotlivých 16-bitových vstupných vzoriek s príslušnými hodnotami Hammingova okna získame 31-bitové číslo, ktoré si bitovým posunom doprava o 15 prevedieme na číslo 16-bitové (*scaling* >>15). Týmto spôsobom získavame rámce, ktoré sú už automaticky vynásobené Hammingovým oknom. Hodnota zaťaženia procesora sa po tejto operácii mierne zvýšila o 0,5% na hodnotu 0,75%.

V tomto momente prichádza na rad funkcia `processFrame()`, ktorá priebežne spracováva jednotlivé rečové rámce tak ako sú postupne vytvárané. V tejto funkcii je vo svojej podstate implementovaný celý zvyšný mechanizmus rozpoznávania. To znamená jednotlivé bloky parametrizácie pre vytvorenie MFCC koeficientov, nasledované blokom obsahujúcim samotnú logiku rozpoznávania. Keďže nás zaujíma zaťaženie procesora a záleží nám najmä na spĺňaní real-time deadlines pri rozpoznávaní, potrebujeme mať určitý prehľad nad tým, koľko prevedených výpočtov sa

ukrýva za percentami spotrebovaného výkonu. Pre názornosť si teda odvoďme v akých intervaloch je funkcia `processFrame()` volaná. Uviedli sme si, že vstupné buffre PING a PONG sú zaplnené vstupnými dátami každých 25ms. Pri pohľade na *obrázok 5.2* vidíme, že príchodom prvých dát sme schopný vytvoriť len jeden rámec. Po uplynutí 25 ms prijimeme buffer PONG s ďalšími 200 vzorkami, z ktorých sme schopný postupným prekrývaním rámcov extrahovať 2 rámce. Nasledujúci buffer PING nám tentoraz poskytne dáta z ktorých vytvoríme 3 rámce. Týmto spôsobom sú každých 25ms vytvorené striedavo 2, resp. 3 rámce. Každý rámec následne znamená volanie funkcie `processFrame()`. Touto úvahou sme sa teda dostali k výsledku, že v rámci dosiahnutia plynulosti rozpoznávania, resp. splnenia real-time deadlines, potrebujeme zabezpečiť, aby funkcia `processFrame()`, dokázala spracovať 3 rámce za 25ms.

FFT

`processFrame()` začína svoju činnosť výpočtom spektra nad rámcom. Využijeme hotovú funkciu `DSK_fft32x32()` [6] z knižnice `dsp64x.lib`, ktorá je optimalizovaná, napísaná v assembleri a preto sa vyznačuje vysokou efektivitou.

```
memset(FFT_vector, 0, (FFT_SIZE*2+1)*sizeof(Int32));
for(i=0;i<VECTOR_SIZE;i++){ FFT_vector[i*2]= frame[i]; }
DSP_fft32x32(w32, FFT_SIZE, FFT_vector, FFT_out);
```

Vstupom funkcie je vektor 256 komplexných čísiel `FFT_vector`, v ktorom sa reálne zložky tvorené jednotlivými hodnotami rámca striedajú s imaginárnymi zložkami nastavenými na nulu. Zvyšných 56 komplexných čísiel je taktiež nastavených na nulu. Po aplikácii FFT získame spektrum, ktoré je uložené vo vektore `FFT_out`. Formát výsledného vektora je rovnaký ako pri vstupnom vektore. Hodnoty typu *integer* striedavo reprezentujú reálne a imaginárne zložky komplexných čísiel spektra. Zdôraznime, že v tomto momente nám presnosť, ktorú poskytujú 16-bitové čísla prestáva vyhovovať a preto musíme prejsť k 32-bitovým číslam. Efektivitu použitého algoritmu FFT nám dokazuje *Profiler*, ktorý nám ukazuje nárast zaťaženie o prijateľných 0,36% na hodnotu 1,07%.

Výkonové spektrum (Power function)

Po získaní spektra potrebujeme vypočítať jednotlivé moduly komplexných čísiel umocnené na druhú. Funkcia `fPower()` je zaujímavá z toho dôvodu, akým spôsobom spracováva vstupné čísla. V teoretickej rovine ide o to, že dve dvojice 32-bitových čísiel majú byť násobené, medzi sebou sčítané a opäť uložené na 32 bitoch (Re*Re + Img*Img). Ak by sme chceli zabezpečiť úplnú bezpečnosť tejto funkcie, 32-bitové čísla by v prvom rade museli byť prevedené na 16-bitové (scaling >>16). Výsledok vypočítaný na základe takto upravených čísiel by sme potom mohli bezpečne uložiť na 32 bitoch. Problém riešenia založeného na tomto spôsobe spočíva v tom, že spektrum čísiel nad ktorými sa funkcia počíta, sa rozprestiera relatívne blízko okolo nuly a najväčšie čísla sú zriedkakedy definované na viac ako 16 bitoch. Použitím bitového posunu doprava o 16 by sme prišli o veľkú väčšinu dôležitých informácii, ktorých nositeľmi sú práve čísla definované na posledných 16 bitoch 32-bitových čísiel. Experimentovaním som dospel k záveru, že "scaling" bitového posunu o 4 (tj. funkcia delenia číslom 16) by mal v dostatočnej miere eliminovať možné problémy a informačný obsah z

globálneho hľadiska by mal zostať nezmenený.

```
pwr[i]=(      (FFT_out[2*i]>>4)   * (FFT_out[2*i]>>4))       //Re*Re
       +(     (FFT_out[2*i+1]>>4) * (FFT_out[2*i+1]>>4));    //Img*Img
```

V krátkom zhrnutí: funkcia `fPower()` prijíma vektor komplexných čísiel. Vezme reálnu a imaginárnu zložku každého čísla, prevedie *scaling* (>>4), zložky umocní na druhú a sčíta. Výsledok uloží na povodné miesto `FFT_out`. Funkcia žiadnym výrazným spôsobom nezaťažuje procesor, pretože v*Profiler-y* zaznamenávame iba nepatrný nárast o 0,14%.

MelBanks

V nasledujúcom kroku prichádza na rad aplikácia banky filtrov na frekvenčné spektrum za pomoci funkcie `applyMelBanks()`. Táto funkcia vytvára okrem iného aj akúsi hranicu v tom, akým spôsobom sa na spracovávané dáta pozeráme. Doteraz sme totiž pracovali výhradne s celočíselnými 16, resp. 32 bitovými dátovými typmi. V tomto momente sa pohľad na dáta mení, pretože do systému zavádzame dátové typy s pohyblivou rádovou čiarkou. Pripomeňme, že použitie reálnych čísiel je zaujímavé z toho dôvodu, že HW architektúra nedisponuje floating-point funkčnými jednotkami a práca s reálnymi číslami musí byť emulovaná. Dôvod prechodu na pohyblivú rádovú čiarku je v zjednodušení použitia nasledujúcich algoritmov. Dôsledky ktoré z toho plynú budeme skúmať v *Profiler-i*. Na prvú polovicu výkonového spektra (druhá polovica symetrická) aplikujeme banku 23 filtrov, ktorých koeficienty sme si dopredu vypočítali. Inicializáciu filtrov si rozoberieme v podkapitole 5.4.5. Vynásobením výkonového spektra a koeficientov príslušných filtrov, resp. aplikovaním trojuholníkových okien na spektrum a následnou sumou cez jednotlivé hodnoty získame 23 koeficientov vo vektore `Mel_feat`. Detaily algoritmu nájdeme v [4]. Celkové zaťaženie procesora po zaradení tejto funkcie do výpočtov nám stúplo na hodnotu 2,17%, čo znamená nárast o takmer jedno celé percento. S ohľadom na počet vykonávaných operácii a použitie reálnych čísiel je tento nárast primeraný.

Logaritmus

`fLn()` je funkcia, ktorá vezme 23 koeficientov z predchádzajúceho kroku a prevedie logaritmus nad týmito číslami. Na prvý pohľad celkom jednoduchá operácia, avšak na *fix-point* architektúre v spojení s reálnymi číslami dokáže situáciu celkom skomplikovať. Na výpočet logaritmu použijeme klasickú funkciu `log()` z knižnice `cmath`. Zistíme však, že *Profiler* nám ukazuje nárast zaťaženia o takmer 2%, čo je pri výpočte iba 23 logaritmov veľký nárast. Zefektívnenie výpočtu dosiahneme použitím vyhladávacej tabuľky (look-up table) pre logaritmus. V prvom rade si uveďme, že hodnoty vektoru `Mel_feat` získaného v predchádzajúcej funkcii `applyMelBanks()` sú reálne (samozrejme kladné) čísla v rozsahu rádovo 10^3 až 10^5, v lokálnych extrémoch až 10^7. Z tohto poznatku som teda prišiel k záveru, že desatinnú časť čísiel možeme pokojne odstrániť bez toho, aby sa to prejavilo na ďalších výsledkoch a look-up tabuľku teda definovať pre celé čísla. Detaily implementácie tabuľky preberieme v podkapitole 5.4.5.

```
v[i] = getLog((unsigned int)Mel_feat[i]);
```

Vyhľadávaním dopredu vypočítaných výsledkov logaritmu funkciou `getLog()` dosiahneme

zefektívnenie, ktoré sa prejaví v *Profiler-i* nárastom zaťaženie o prijateľných 0,69% (v porovnaní s 2% pri `log()`) na celkovú hodnotu 2,76%.

DCT a C0

Poslednými dvoma krokmi potrebnými k získaniu MFCC koeficientov sú funkcie `applyDCT()` a `fCalcC0()`. Výpočet DCT [4] je podobne ako v prípade `applyMelBanks()` odľahčený použitím konštánt, ktoré sme si pri inicializácii dopredu vypočítali (kapitola 5.4.5). Výsledkom DCT je prvých 12 MFCC koeficientov. Posledný 13 nám dopočíta funkcia `fcalcC0()`.

```
applyDCT(23, Mel_feat, 12, feats);
feats[13] = fCalcC0(23, Mel_feat);
```

V tomto momente sme sa dostali do stavu, keď vo vektore `feat` máme uložených 13 MFCC koeficientov a parametrizácia rámca je tým pádom ukončená. Výpočtom diskrétnej kosínovej transformácie a koeficientu C0 sa nám zaťaženie procesoru zvýšilo o 0,81%. Celkový spotrebovaný výkon procesoru sa po parametrizácii rečového rámca ustálil na prijateľných 3,57%. Zostáva nám teda dostatočne veľké množstvo nevyužitého strojového času, ktorý môžeme čerpať v nasledujúcich algoritmoch rozpoznávania reči. Pre porovnanie uvádzam, že v práci [7] sa nám na tomto mieste hodnota záťaže procesora pohybovala okolo 10%.

5.4.4 Viterbiho algoritmus

Na tomto mieste sa dostávame k samotnému mechanizmu rozpoznávania reči. Na výpočet pravdepodobností, reps. vierohodností generovania požadovaných sekvencií rečových parametrov jednotlivými modelmi slov, použijeme techniky dynamického programovania, ktoré v našom prípade reprezentuje Viterbiho algoritmus [4]. Metódy využívané v algoritme sú v dôsledku svojej zložitosti implementované na úrovni floating-point čísiel. Budeme sa musieť teda snažiť o optimalizáciu, aby výpočty prebiehali v rámci povolených limitov.

Algoritmus je v programe reprezentovaný funkciou `viterbi()`. Na vstup funkcie posielame MFCC koeficienty `feat[]` aktuálneho rečového rámca a model rozpoznávaného slova definovaného v štrukrúre `Model`, ktorú si popíšeme v podkapitole 5.4.5.

V prvom rade si obnovíme konfiguráciu, v ktorej sa model nachádzal pri poslednom prechode Viterbiho algoritmu.

```
float *tokens  = model->token;
```

Konfigurácia je definovaná čiastkovými vierohodnosťami generovania sekvencie v každom stave HMM, ku ktorým sme sa v doterajšom priebehu rozpoznávania dostali. Na obrázku *2.6* nám konfiguráciu v čase *t* definuje množina sivých bodov. Na základe vstupných MFCC koeficientov sa do nasledujúcej konfigurácie presunieme tak, že ku čiastkovým vierohodnostiam `tokens[j]` pripočítame logaritmus pravdepodobností prechodu z príslušného stavu `model->probs[j][k]` a logaritmus hodnoty funkcie hustoty rozdelenia pravdepodobnosti pre Gauss. rozloženie cieľového stavu HMM.

```
//j = 0 .. 18   -> HMM initial state
//k = 0 .. 18   -> HMM target state

float prob_log  = model->probs[j][k];
float gauss_log = LogGaussPDF(feats, model->means[k], model->vars[k],
                                  model- >pars, model->gConst[k]);
float res = prob_log + gauss_log + tokens[j];
```

LogGaussPDF

Výpočet logaritmu hodnoty funkcie hustoty rozdelenia pravdepodobnosti pre Gaussovské rozloženia nám zabezpečuje funkcia `LogGaussPDF()`, ktorej na vstup posielame MFCC koeficienty a ich počet, Gaussovské rozloženie a hodnotu `gConst` (4.2.), ktoré máme uložené v štruktúre `Model` (podkapitola 5.4.5). Funkcia je presným zápisom vzťahu (4.1.), ktorý sme si odvodili vo 4.kapitole.

```
float logPdf = 0;
int i;
for(i = 0; i < size; i++)
{
logPdf+=(((observation[i]-means[i])*(observation[i]-means[i]))/variances[i]);
}
logPdf = -0.5 * (gconst + logPdf);
```

Po príchode MFCC koeficientov posledného rečového rámca prevedieme prechod do koncového stavu HMM a tým získame celkovú vierohodnosť generovania rozpoznávanej sekvencie reči daným modelom. Pri pohľade na *Profiler* zistíme, že nárast zaťaženia procesora po aplikácii Viterbiho algoritmu nám vzrastie o 4,6%. Aplikáciu nakoniec skompletizujeme tým, že Viterbiho algoritmus postupne aplikujeme na všetkých 16 modelov slov a na konci vyberieme ten, v ktorom sme dosiahli najväčšiu vierohodnosť generovania sekvencie rozpoznávaného slova. Nastavením príslušných príznakov `led0_on` ... `led3_on` informujeme užívateľa o výsledku. Po zapojení všetkých 16 modelov do výpočtov, nám *Profiler* ukazuje výsledné zaťaženie procesoru, ktoré dosiahlo hodnoty okolo 77%.

5.4.5 Dátové štruktúry, konštanty a modely

Na tomto mieste si ešte doplňme implementačné detaily, ktoré sme si doteraz neuviedli a v predchádzajúcich častiach sme sa na ne iba odkazovali. Jedná sa najmä o inicializačné metódy, v ktorých si po bezprostrednom spustení aplikácie inicializujeme dátové štruktúry a dopredu vypočítame konštanty opakovane používané v jednotlivých algoritmoch. Týmto spôsobom potom v priebehu aplikácie významnou mierou šetríme strojový čas, ktorý môžme následne využiť pri dôležitejších výpočtoch. Teraz keď máme prehľad o kontexte v ktorom sa tieto štruktúry budú používať, ľahšie pochopíme ich odvodenie.

Hammingovo okno

Koeficienty okna uložené vo vektore `hamming[]` si incializujeme funkciou `createHamming()`. Jednotlivé hodnoty počas inicializácie hneď normalizujeme na rozsah $0..2^{15}$-1

```
hamming[i] = (short)_round((0.54f-0.46f*(cos(2.0f*PI*i/(n-1))))*32767 );
```

Banky filtrov

V procese parametrizácie vo funkcii `applyMelBanks()` používame filtre, ktoré je tak isto vhodné si na začiatku inicializovať. Koeficienty jednotlivých filtrov sa funkciou `melBanks_init()` [4] dopredu počítajú a ukladujú do dátovej štruktúry:

```
struct sMelBanks{
	int Count;
	int SampleFreq;
	int WindowLength;
	int WindowLength_2;
	float f0[MEL_COUNT+1];
	float f0m[MEL_COUNT+1];
	float Coeffs[FFT_SIZE/2];
	short Banks[FFT_SIZE/2];
	float lo;
	float hi;
	float mlo;
	float mhi;
	int fftlo;
	int ffthi;
};
```

Diskrétna kosínová transformácia

Pozrime sa na funkciu výpočtu DCT [4] a ukážme si, ktoré nemenné hodnoty budeme opakovane používať a ich výpočet v každom volaní by bol nefektívny.

```
inline void sDCT(int n, float *re, int nOut, float *Out)
{
	int j, k;
	float NormC = sqrtf(2.0f/(float)n);
	float PiByN = (float)PI/(float)n;
	float v;

	for(k = 0; k < nOut; ++k)
	{
		Out[k] = 0;
		v = PiByN * (float)(k + 1);
		for(j = 0; j < n; ++j)
			Out[k] += re[j] * cosf(v * ((float)j + 0.5f));
		Out[k] *= NormC;
	}
}
```

Inicializačná metóda `DCT_init()` teda pre známe hodnoty `n=23` a `nOut=12` vypočíta hodnoty konštánt `NormC`, `PiByN` a do vektora `csf[276]` (12*23=276) postupne uloží výsledky funkcie `cosf()`. Výpočet DCT v priebehu aplikácie sme tým pádom redukovali na približne 300 operácii násobenia reálnych čísiel oproti pôvodnému dvojnásobnému počtu násobení a funkcií `cosf()`.

Look-Up Table

Použitie Look-Up tabuľky pre získanie logaritmu čísla, výrazným spôsobom urýchluje prevedenie funkcie `fPower()`. Princíp fungovania použitej vyhladávacej tabuľky je nasledovný. Máme definované dva vektory rovnakej dĺžky, `nums[]` a `logs[]`. Do prvého si uložíme čísla *x*, ktorých logaritmy *y=log(x)* sú dopredu vypočítané a uložené do druhého vektora na príslušné miesto s rovnakým indexom. Pri volaní funkcie `getLog(x)` sa binárnym vyhľadávaním nájde v `nums[]` číslo *x*, reps. číslo najbližšie ku *x* a získaným indexom sa z druhej tabuľky `logs[]` určí hodnota logaritmu log(*x*).

Na základe predpokladov z 5.4.3, že logaritmus budeme počítať len nad celými číslami, sa tomu podriadila aj konštrukcia tabuľky. Tá bola dimenzovaná tak, aby na vstup funkcie `getLog(x)` mohli byť privádzané celé čísla v rozsahu 1 - 2^{32}. Obor hodnôt funkcie je teda <0 ... 22,18>. S ohľadom na pamäťové nároky tabuľky a s ohľadom na požadovanú presnosť funkcie logaritmu bolo treba zvoliť vhodnú veľkosť tabuľky. Kompromisom bolo zvolenie veľkosti 98304, ktorá je použiteľná aj algoritmom binárného vyhľadávania, keďže $98304=2^{16} + 2^{15}$. Presnosť, resp. minimálný rozdieľ dvoch po sebe idúcich logaritmov uložených v tabuľke s touto veľkosťou je na úrovni 22,18/98304=0,0002256.

Hodnoty v tabuľkách získame tak, že pre jednotlivé dané výsledky logaritmu, y = 0 ... 22,18 s krokom 0,0002256, ktoré zapíšeme do `logs[]`, vypočítame zaokrúhlenú hodnotu x=exp(y), ukladanú do `nums[]`. V danom rozsahu pri použitom kroku a zaokrúhlení získame hodnoty x, ktoré sa budú vo veľkom počte opakovať. Tieto opakujúce sa hodnoty do tabuľky `nums[]` nezapíšeme, pretože pri binárnom vyhľadávaní potrebujeme pre jednu hodnotu *x* len jeden výsledok *y*. Týmto spôsobom sa nám však tabuľky `nums[]` a `logs[]` zaplnia približne iba do jednej polovice. Experimentálne som prišiel na to, že zvolením kroku 22,18/151808=0,0001461 pre hodnoty *y* z ktorých spätne vupočítame *x=exp(y)*, získame taký počet rôznych výsledkov x, ktorým zaplníme tabuľku `nums[]` v celej jej kapacite.

`getLog(x)` je implementácia klasického binárneho vyhľadávania s drobnými modifikáciami. V prvom rade treba povedať, že vo veľkej väčšine prípadov presnú hodnotu *x* v tabuľke nenájdeme a preto budeme musieť nájsť hodnotu najbližšiu k *x*. Najneskôr v 17 krokoch ($2^{16}<98304<2^{17}$) sa teda dostaneme k číslu, ktoré sa nachádza v blízkom okolí *x*. Testovaním bezprostredne susedných čísiel (aktuálne, predchádzajúce, nasledujúce) v tabuľke `nums[]` nájdeme hodnotu najbližšiu k *x* a získaným indexom vrátime z tabuľky `logs[]` výsledok *log(x)* s presnosťou +- 0,0001461. Inicializáciu Look-Up tabuľky zabezpečuje funkcia `lookup_init()`.

Modely rozpoznávaných slov

Ako sme si už v 4.kapitole uviedli, trénovanie modelov rozpoznávaných slov prebiehalo na PC za pomoci nástrojov HTK. Našou úlohou bolo zvoliť vhodný spôsob uloženia týchto modelov, reprezentovaných skrytými Markovými modelmi, na DSP. V súbore `model_def.h` si nadefinujme štruktúru, ktorá bude uchovávať HMM:

```
typedef struct {
  int value;          //identifikátor modelu
  int states;         //počet stavov HMM
  int pars;           //počet parametrov (13 MFCC)
  float gConst[N];    //konštanta gConst pre každý stav
  float means[N][P];  //str.hodnoty Gauss. rozložení
  float vars[N][P];   //rozptyl Gauss. rozložení
  float probs[N][N];  //prechod. pravdepodobnosti HMM stavov
  float tokens[N];    //informácie o aktuálnom stave HMM
}Model;
```

Vieme, že na jednotlivé modely, resp. HMM, bude opakovane aplikovaný Viterbiho algoritmus pre všetky prichádzajúce rečové rámce a tým bude dynamicky meniť ich aktuálny stav, konfiguráciu. Je preto výhodné si priamo do štruktúry `Model` pridať vektor `tokens[]`, ktorý bude obsahovať informácie o aktuálnej hodnote jednotlivých stavov HMM v priebehu rozpoznávania. Keďže Viterbiho algoritmus počíta s logaritmami, hodnoty uložené v `probs[N][N]` sú priamo logaritmy prechodových pravdepodobností. Poslednou vecou je dopredné vypočítanie konštanty `gConst` (4.2) pre každé Gaussovské rozloženie popisujúce stavy HMM. Do súboru `model.h` si v definovanej štruktúre uložíme modely slov českých čísloviek: “NULA”, “JEDNA”, “DVA”, “DVĚ”, ”TŘI”, “ČTYŘI”, “ČTYRI”, “ŠTYŘI”, “ŠTYRI”, “PĚT”, “ŠEST”, “SEDM”, “SEDUM”, “OSM”, “OSUM”, “DEVĚT”.

5.5 Testovanie

Testovanie systému rozpoznávania reči prebiehalo počas celého vývoja. V prvých fázach vývoja pri implementácii jednotlivých algoritmov sme testovali použité metódy na presne zadaných dátach, ktorých výsledky sme priamo porovnávali s výsledkami produkovanými overenými referenčnými algoritmami na PC. Týmto spôsobom sme validovali jednotlivé časti systému a overili správnosť použitých metód.

Po celkovom dokončení implementácie systému rozpoznávania reči na embedded system bolo prevedené testovanie funkčnosti systému ako celok. Keďže k dispozícii nemáme žiadny podobný systém, ktorého výsledky by sme mohli považovať za referenčné, testovanie malo neformálny charakter. To pozostávalo z rozpoznávania českých čísloviek vyslovených priamo užívateľom. Po dôkladnom otestovaní sme odhalili jednodlivé charakteristické vlastnosti a črty systému rozpoznávania. V zásade môžeme skonštatovať, že s číslovkami “JEDNA”, ”DVA”, ”PĚT”, ”ŠEST”, ”SEDM”, “OSM” a “OSUM” nemá rozpoznávač vážnejšie problémy a úspešnosť rozpoznania uvedených čísloviek je vysoká. Číslovka 4 je tiež rozpoznávaná s vysokou úspešnosťou, avšak naviazanie na konkrétny model “ČTYŘI”, “ČTYRI”, “ŠTYŘI”, “ŠTYRI” je vo väčšine prípadov nesprávne. Rozpoznávanie ďalších čísloviek “NULA”, “TŘI” a “SEDUM” je problematické. Pre správne rozpoznanie si musíme dať záležať na výslovnosti a zdôrazniť charakteristické fonémy slova. Úspešnosť rozpoznávanie posledných dvoch slov “DVĚ” a “DEVĚT” je v systéme minimálna.

Subjektívnym vyhodnotením dosiahnutých výsledkou by sa dala úspešnosť rozpoznávania čísloviek vyjadriť hodnotou okolo 75-80%.

6. Záver

Hlavným cieľom diplomovej práce bolo vytvoriť systém rozpoznávania reči adaptovaný na hardwarovú architektúru s obmedzenými hardwarovými prostriedkami. Požiadavkou bolo použitie cieľového embedded systému, ktorý nedisponuje vypočetnými jednotkami pracujúcimi nad číslami s pohyblivou rádovou čiarkou. Práca priamo naväzuje na ročníkový projekt [7], v ktorom sa riešila úvodná fáza vývoja tohoto systému.

Na základe získaných poznatkov bol vytvorený návrh rozpoznávača, ktorý bolo možné na embedded systém portovať. Postupne sme implementovali algoritmy parametrizácie pre výpočet MFCC koeficientov. Tak isto sme nástroj parametrizácie portovali na PC, aby sme trénovanie slovných modelov mohli robiť nad identickými rečovými príznakmi. Modely, ktoré sme si na PC natrénovali sme museli vhodným spôsobom presunúť na embedded system, na ktorom budú pripravené k použitiu. Implementovaním Viterbiho algoritmu a optimalizáciou jednotlivých metód sme skompletizovali systém rozpoznávania reči.

Vo všeobecnosti sú rozpoznávače reči výpočetne náročné aplikácie. Na začiatku vývoja systému nebolo možné presne určiť, do akej miery bude zvolená hardwarová architektúra schopná obsluhovať a zvládať systém rozpoznávania reči. Z tohto dôvodu sme preto v priebehu vývoja pozorne sledovali hodnoty záťaže CPU v jednotlivých krokoch.

CPU load		
Overall	Partial	
0,25%		DSP/BIOS
0,75%	0,50%	Buffering, frames, Hamming w.
1,07%	0,36%	FFT
1,21%	0,14%	Power spectrum
2,17%	0,96%	MelBanks
2,76%	0,69%	Logarithm
3,75%	0,81%	DCT, C0
	4,60%	Viterbi
	73,00%	16 x Viterbi (pre každý model)
77,00%		Výsledné rozpoznávanie

Výsledná záťaž procesora na úrovni 77% nám poskytuje určitý priestor pre rozšírenie systému. V ďalšom pokračovaní vývoja bude vhodné zamerať sa na dosiahnutie zvýšenej uspešnosti rozpoznávania reči. V našej práci sme sa priblížili k úrovni 75-80%. Prvým krokom by mohlo byť zavedenie tzv. derivačných (delta) koeficientov, ktoré by nám zvýšili počet MFCC koeficientov na 39. Podľa predpokladov by sa použitím tejto metódy nárast zaťaženia procesora mal pohybovať pod úrovňou 23%, ktorú máme v tejto fáze projektu stále k dispozícii. Úspešnosť rozpoznávania by sa

mohla dostať až na úroveň okolo 90%. Ďalšie zvýšenie by prinieslo použitie zmesy viacerých Gaussovských rozložení v každom stave HMM. V tomto prípade však bude potrebné zefektívniť použité algoritmy, pretože výpočetný výkon procesora nebude pravdepodobne dostatočný. Prvými kandidátmi sú funkcie aplikovania banky filtrov a DCT, ktoré by sme mohli modifikovať tak, aby pracovali nad číslami s pevnou rádovou čiarkou. Optimalizácia Viterbiho algoritmu bude taktiež nevyhnutná. Po vyriešení spomínaných problémov bude ešte vhodné odstrániť *offline* predspracovanie signálu a celý systém implementovať ako *online* rozpoznávač. V tomto momente by sme získali kvalitný systém rozpoznávania reči implementovaný na embedded systéme.

Literatúra a iné zdroje informácií:

[1] Lawrence Rabiner, Biin-Hwang Juang: Fundamentals of Speech Reocognition, Prentice Hall

[2] Doc.Dr.Ing Černocký Jan: materiály k prednáškam Číslicové spracovanie reči

[3] Schwarz P., Matějka P., Černocký J.: Towards Lower Rates in Phoneme Recognition, VUT, Brno

[4] Schwarz P., Burget L.: Feature Extraction algorithms, Viterbi algorithm

[5] Steve Young, Mark Gales, ... : The HTK book, Cambridge University Engineering Department

[6] TMS320 C6416 DSK manual pages : Texas Instruments Inc.

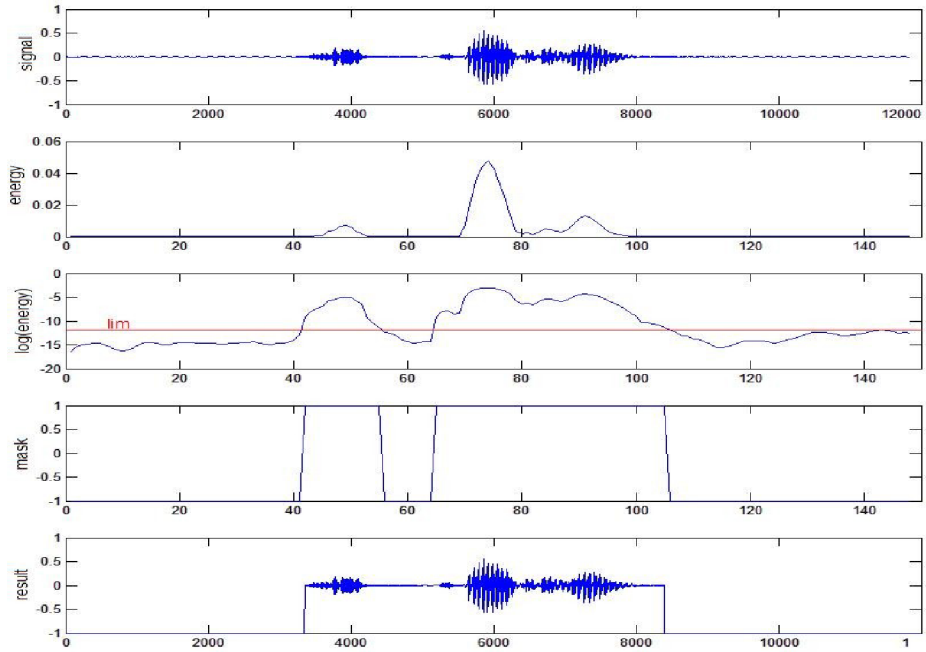
[7] Král Tomáš, Rozpoznávač/detektor slov na čipu [Ročníkový projekt], FIT VUT v Brne, 2006

Prílohy

Príloha A: Algoritmus identifikácie poklesu energie rečového rámca spôsobeného vlastnosťami vyslovovaných fonémov.

```
%Nram - pocet ramcov
%mask - pre vsetky ramce - 1= recovy ramec; -1= ramec ticha
sp_start =0;                                %cislo pociatocneho recoveho ramca
sp_end = 0;                                 %cislo koncoveho recoveho ramca
speech = 0;                                 %priznak sekvencie reci - true/false
sil_cnt =0;                                 %pocet ramcov ticha v rade (uprostred reci)
for ii=1:Nram,                              %pre vsetky ramce
    if(mask(ii)==1)                         %ak recovy ramec....
        if(speech == 0)                     %ak v sekvencii ticha...
            if(sil_cnt==0) sp_start = ii;   %a ziadny predosly recovy ramec - > zaciatok reci
            end;
           speech = 1;                      %priznak ze sa nachadzame v sekvencii reci
           sil_cnt = 0;                     %vynulovanie pocitadla ramcov ticha
        end;
    else                                    %ak ramec ticha
        if(speech == 0)                     %a sme v sekvencii ticha
            if(sp_start~=0) sil_cnt=sil_cnt+1;%ak zaciatok reci definovany...
            end;                            %inkrementuj pocitadlo
        else                                %ak v sekvencii reci
            sil_cnt=sil_cnt+1;              %inkrementuj pocitadlo ramcov ticha
            sp_end = ii;                    %rec mozno skoncila -> nastav sp_end
            speech =0;                      %zacina sekvence ticha
        end;
    end;
    if((sil_cnt>15)&&(sp_start~=0))         %ak zaciatok reci definovany a pocet...
        sp_end = ii-16;                     %ramcov ticha>15 -> rec konci
        if(sp_end-sp_start > 20)break;      %ak sekvencia dostatocne dlha -> KONIEC
        else                    %...ak nie, vznikol nahodny sum -> pokracuj v identifikovani
reci
            sp_start = 0;
            sp_end   = 0;
            speech = 0;
            sil_cnt = 0;
        end;
    end;                                    %sil_cnt<15 -> pokles energie v ramci tolerancie
end
 % sp_start a sp_end obsahuju cislo pociatocneho a koncoveho recoveho signalu
```

Príklad nízkoenergetickej spoluhlásky "T" v slove "štyři"



Príloha B: Code Composer Studio a editácia konfiguračného súboru