ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická
Katedra radioelektroniky

Výukové úlohy na číslicovém signálovém procesoru

Demonstrations on Digital Signal Processor

Bakalářská práce

Studijní program: Komunikace, Multimédia a Elektronika
Studijní obor: Multimediální technika

Vedoucí práce: prof. Ing. Pavel Zahradník, CSc.

Dominik Šmíd

Praha 2016

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra radioelektroniky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Dominik Šmíd**

Studijní program: Komunikace, multimédia a elektronika
Obor: Multimediální technika

Název tématu: **Výukové úlohy na číslicovém signálovém procesoru**

Pokyny pro vypracování:

Pro výukové účely, na výukovém přípravku s číslicovým signálovým procesorem TMS320C6748/OMAPL138 naprogramujte algoritmus pro ladičku akustických tónů a dále algoritmus pro detekci tónů a jejich zjednodušený zápis do notové osnovy. Algoritmy odlaďte pro práci v reálném čase.

Seznam odborné literatury:

[1] Rulph Chassaing, DSP Applications Using C and the TMS320C6x DSK, Wiley 2002.
[2] Rulph Chassaing, Digital Signal Processing and Applications with the C6713 and C6416 DSK, Wiley 2005.
[3] www.ti.com

Vedoucí: prof. Pavel Zahradník Ing., CSc.

Platnost zadání: do konce letního semestru 2016/2017

doc. Mgr. Petr Páta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 19. 2. 2016

Bibliografický záznam

ŠMÍD, Dominik. *Výukové úlohy na číslicovém signálovém procesoru*. Praha, 2016. Bakalářská práce. České vysoké učení technické v Praze, Fakulta elektrotechnická, Katedra radioelektroniky. Vedoucí práce prof. Ing. Pavel Zahradník, CSc.

Anotace

Bakalářská práce se zabývá vývojem algoritmu rozpoznání tónu hudebního nástroje, jeho implementací na vývojovém přípravku s číslicovým signálovým procesorem a odladěním aplikace pro práci v reálném čase. Zjištěné tóny jsou i s dobou jejich trvání přepisovány do zjednodušeného notového zápisu. Výstup ladičky i notový zápis je poté graficky zobrazen na obrazovce VGA monitoru. Základ algoritmu používá Fourierovu transformaci pro následnou analýzu zvukového signálu v kmitočtové oblasti. Zbytek algoritmu se dvěma způsoby snaží nalézt tón ve spektru signálu pomocí všech jeho harmonických složek. Algoritmus je vyvíjen v prostředí programu MATLAB, následně je přepsán do jazyka C a testován v reálných podmínkách pomocí vývojového přípravku OMAP-L138 LCDK.

Abstract

The bachelor's thesis deals with the development of a recognition algorithm for a musical instrument tone pitch, its implementation on the development kit with a digital signal processor and making the application work in real-time. Recognized tones are then processed with their time duration and written as a simplified musical score. Tuner output and musical score are then graphically printed on a VGA monitor screen. The core of the algorithm uses Fourier Transform for the following analysis of the sound signal in the frequency domain. For the rest of the algorithm, there are two ways to find a tone in signal's spectrum with the help of all his harmonics. The rest of the algorithm is developed in the MATLAB software and subsequently rewritten into the C language and challenged in real-time conditions using the OMAP-L138 low cost development kit.

Klíčová slova

Číslicový signálový procesor, zvukový signál, elektronika, rychlá Fourierova transformace, autokorelace, spektrum, vyšší harmonický kmitočet, lokální maximum, C, MATLAB, ladička, tón, notový zápis

Keywords

Digital Signal Processor, audio signal, electronics, Fast Fourier Transform, autocorrelation, spectrum, higher harmonics, local maximum, C, MATLAB, tuner, tone, musical score

Prohlášení

„Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.“

V Praze dne ……

Poděkování

Děkuji prof. Ing. Pavlu Zahradníkovi, CSc., vedoucímu práce, za cenné rady a připomínky při tvorbě bakalářské práce a za zapůjčení potřebného hardwaru.

Obsah

Úvod

Cílem této práce je návrh algoritmu rozpoznání tónu hudebního nástroje v reálném čase a jeho praktická realizace. Pro vývoj algoritmu je použit software MATLAB a vytvořena implementace nepracující v reálném čase, ale pouze s nahraným zvukovým signálem. Pro implementaci algoritmu v reálném čase a případné testování v reálných podmínkách je použit vývojový přípravek OMAP-L138 LCDK a vývojové prostředí Code Composer Studio (dále jen CCS). Ve vývojovém prostředí je vytvořena, zkompilována a debuggována aplikace v jazyce C jako samostatný program neběžící pod žádným operačním systémem. Aplikace používá jen jedno ze dvou jader číslicového signálového procesoru OMAP-L138 a to jádro C6748 (dále jen DSP). Dalším cílem této práce je na detekční algoritmus navázat grafický výstup aplikace – ladičku akustických tónů a průběžný výpis výsledků do zjednodušeného notového zápisu. Jako výstup je použito VGA monitoru připojeného ke konektoru na vývojovém přípravku.

První kapitola se zabývá teorií okolo problematiky zpracování zvukových signálů, jejich analýzou v číslicovém světě a konkrétní analýzou hudebních nástrojů.

Druhá kapitola popisuje samotný algoritmus rozpoznání tónů hudebních nástrojů. Zabývá se převodem signálu do kmitočtového spektra pomocí rychlé Fourierovy transformace (dále jen FFT) a dalšími nástroji, které spektrum zpracují či analyzují. Na základ algoritmu pak naváže autokorelace výkonového spektra nebo na zpracování složitějších zvukových signálu s více tóny se specializuje alternativní polyfonní algoritmus. Ten spočívá v hledání lokálních maxim ve spektru a rozboru nalezených vyšších harmonických kmitočtů. Dále jsou v podkapitolách rozebírána různá omezení, jsou popsány optimalizace na implementaci pro DSP a aplikace je podrobena statistice o výpočetní náročnosti jednotlivých bloků programu. Výsledky vypočtené algoritmem jsou předány do grafické části aplikace, o které pojednává kapitola následující.

Třetí kapitola se zabývá grafickým prostředím aplikace, které se objeví na obrazovce připojené k desce. Zabývá se také mechanismy naprogramovanými v C, které jsou s grafickým prostředím spojené, a návaznostmi na algoritmus rozpoznání tónu. Kapitola se týká pouze implementace pro DSP. V podkapitole jsou popsány i další interaktivní prvky aplikace nesouvisející s obrazem. Jsou to tlačítka, přepínače a LED zabudované na desce vývojového přípravku.

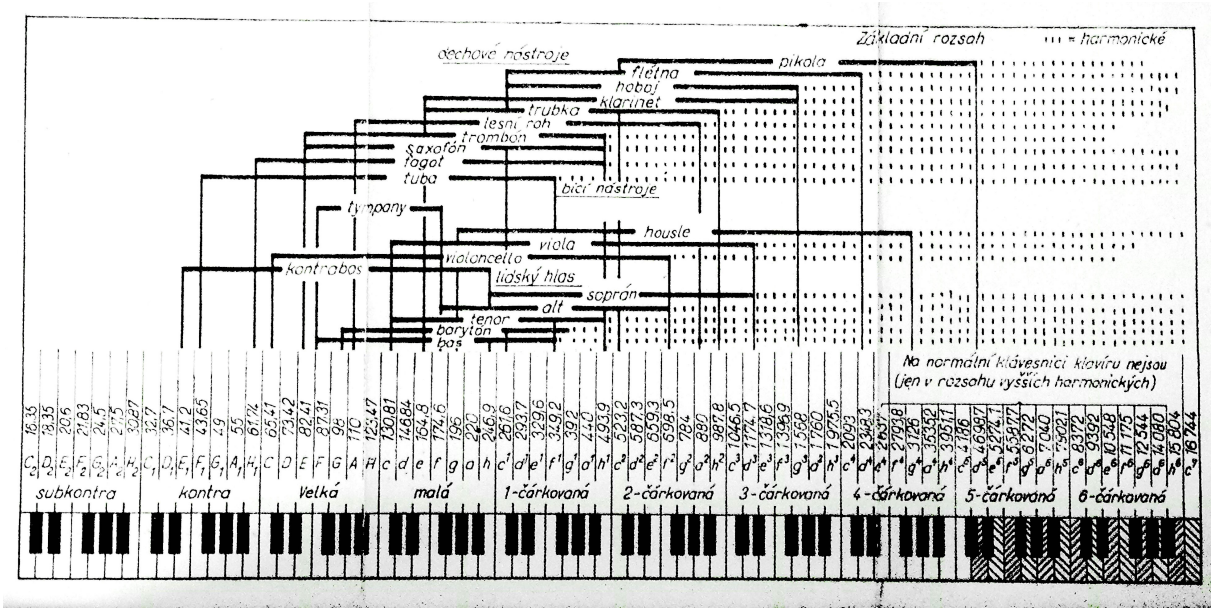
Poslední krátká kapitola popisuje software použitý pro vývoj algoritmu a aplikace. Dále rozebírá optimalizované knihovní funkce použité pro DSP implementaci algoritmu a hodnotí jejich výhody, nevýhody a výpočetní náročnost.

V závěru je pak osobní zhodnocení autora, získané zkušenosti a náročnost praktické části práce. Na konci je posouzeno splnění úkolů zadání.

1 Popis problematiky a analýza

K analýze zvukového signálu lze vhodně použít kmitočtové spektrum. Kdyby se jednalo pouze o čistý sinusový signál, daly by se parametry signálu jednoduše určit i z časového průběhu. Takový signál se ale v reálném světě zvuků téměř nevyskytuje. Často je zvuk složen minimálně z několika takových sinusových průběhů, dokonce i ze stovek různých kmitočtů a amplitud. Analýza ve kmitočtové oblasti tedy bez pochyb podá informace o reálných zvukových signálech mnohem přehledněji, než analýza v časové oblasti.

Další výchozí bod pro tuto práci jsou obecné parametry signálu hudebních nástrojů, spíše tedy kmitočtového spektra jejich tónů. Pro návrh ladičky je dobré si vymezit ve které části spektra kmitočty tónů hledat. Je zbytečné počítat i části spektra, kde pokud nějaké kmitočty signálu najdeme, určitě nejsou vytvářeny hudebním nástrojem. Pro základní představu poslouží obrázek s rozsahy různých nástrojů i lidského hlasu:



0br. č. 1: Rozsahy hudebních nástrojů [13]

V obrázku výše je použito jiné číslování tónů, tzv. Helmholtzovo značení. Tato práce však používá mezinárodní značení. Tón a^1 z obrázku odpovídá tónu A_4, 440 Hz. Pro konkrétnější představu o rozsahu tónů a jejich základních kmitočtů je použito některých údajů z tabulky sloužící pro naladění digitálních ekvalizačních filtrů [1]:

Hudební nástroj	Nejnižší tón	Nejvyšší tón	Kmitočtový rozsah [Hz]
Kytara	E_2	F_6	82 – 1 397
Basová kytara	E_1	C_4	41 – 262
Flétna	C_4	B_6	262 – 1 976
Housle	G_3	G_7	196 – 3 136
Trubka	E_3	B_5	165 – 988
Klarinet	E_3	G_6	165 – 1 568
Klavír (pianino)	A_0	C_8	*28 – 4 186*
Varhany	C_0	A_9	*16 – 7 040*
Soprán (ženský zpěv)	C_4	C_6	262 – 1 047
Alt (ženský zpěv)	F_3	F_5	175 – 698
Tenor (mužský zpěv)	C_3	C_5	130 – 523
Bas (mužský zpěv)	F_2	E_4	87 – 330

Tab. č. 1: Rozsahy nejběžnějších hudebních nástrojů a zpěvu

Spolehlivě tedy i nejhlubší tóny jakéhokoliv nástroje mají kmitočet alespoň 40 Hz a nejvyšší tóny maximálně 3,2 kHz. Výjimkou jsou klavír a varhany, které se ale stejně nedají jednoduše naladit.

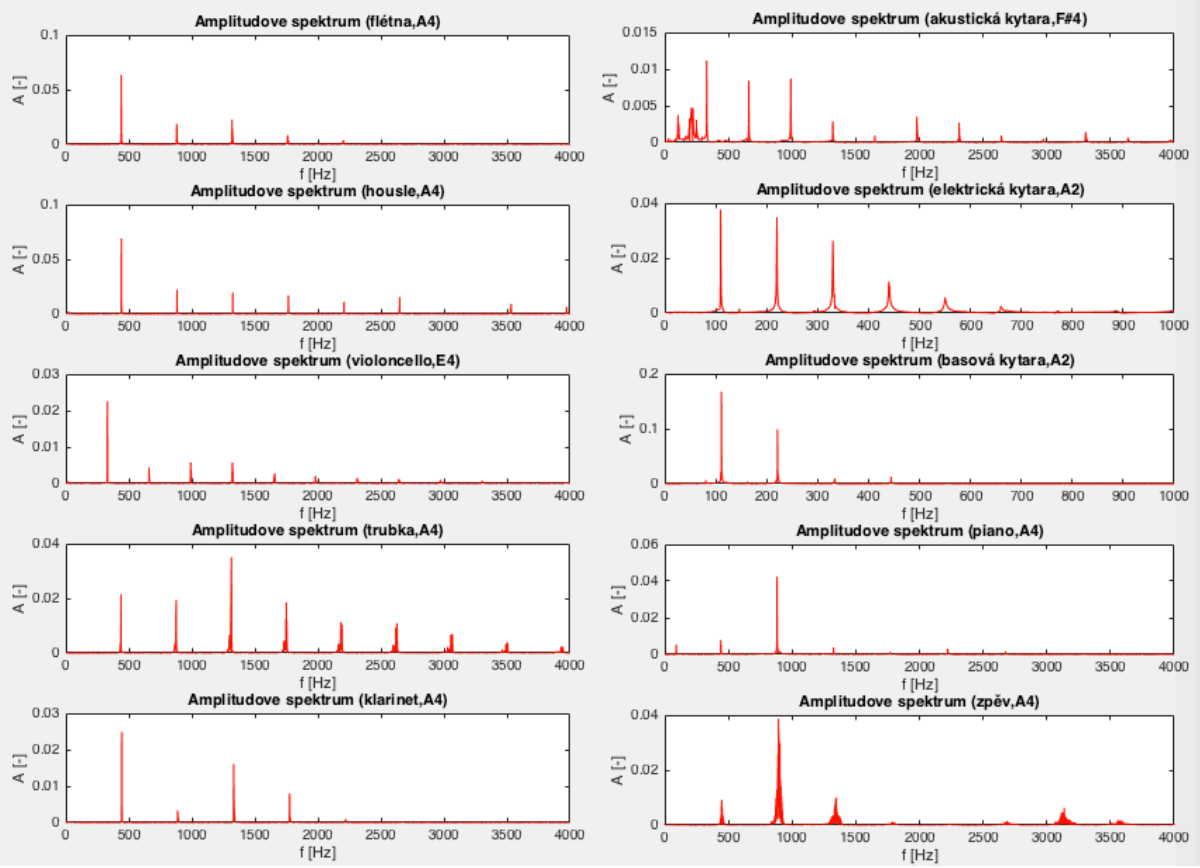
Myšlenka algoritmu rozpoznání tónů s autokorelací výkonového spektra vznikla během mého pročítání různých textů na internetu, které se touto problematikou zabývají. V [2] se objevuje dotaz mířící přesně do černého. Jak vylepšit čistou analýzu kmitočtového spektra, aby byl algoritmus ladičky robustnější? V textu otázky na tomto fóru autor zmiňuje dva různé nástroje zpracování zvukového signálu k účelům ladičky. Prvním je často používaná rychlá Fourierova transformace a druhým je autokorelace. Oba mají podobný výsledek. Zde mě napadlo, zda by šlo použít oba matematické nástroje zároveň. Navíc jsem si všiml, že výkonové spektrum více zvýrazní špičky harmonických kmitočtů oproti šumovému pozadí než amplitudové spektrum. Zkusil jsem tedy navrhnout algoritmus, který na vzorky výkonového spektra signálu navíc aplikuje autokorelaci a pak teprve hledá kmitočet s nejvyšší špičkou. Výsledky tohoto algoritmu jsou oproti algoritmu bez autokorelace mnohem lepší (viz obrázek č.3). Ale je to za cenu více než dvakrát větší výpočetní náročnosti při počítání autokorelace navíc oproti samostatné FFT. Myšlenka algoritmu s hledáním lokálních maxim a práci s nimi pochází ze stejného fóra, tentokrát z odpovědi, která mluví o použití struktury harmonických kmitočtů a práci s více harmonickými zároveň. Tuto myšlenku jsem začal rozvíjet až ve chvíli, kdy jsem začal testovat hotový algoritmus s autokorelací na složitějších hudebních melodiích či celých akordech. Postupně vznikl alternativní algoritmus popsaný v kapitole o modifikacích algoritmu.

1.1 Spektrální složení tónů hudebních nástrojů

Už na střední škole ve fyzice se lze dozvědět jak fungují hudební nástroje a jak vzniká jejich zvuk. Například na brnknuté struně může vzniknout stojaté vlnění odrážející se od obou konců tam i zpět. Vlnění vzniká nejen v základní vlnové délce rovné dvojnásobku délky struny. Může vznikat i stojaté vlnění s vlnovou délkou vydělenou celým číslem tak, aby krajní uzly stojatého vlnění vyšších harmonických tónů vyšly přesně na kraje struny. Podobně je tomu i u dechových nástrojů, kde stojaté vlnění vzniká v dutině vzduchového sloupce. Základní vlnění vzniká s uzly na krajích sloupce, ale i vlnění o více vlnových délkách, které se vejdou do základní vlnové délky několikrát. U všech hudebních nástrojů v jednom tónu tedy zní více kmitočtů o přesně celém násobku základního harmonického kmitočtu. Tentýž tón zahraný různými nástroji může díky tomu znít rozdílně (tzv. barva

tónu). Záleží na složení amplitud vyšších harmonických kmitočtů tónu. Toto teoretické složení tónů se pokusím dokázat i v praxi.

Zde jsou výsledky mé vlastní analýzy nahrávek čistého tónu různých hudebních nástrojů. Nahrávky samostatného tónu zahraného nástrojem jsou zpracovány v MATLABu. Ten spočítá FFT pro každý nástroj a následně přehledně zobrazí amplitudové spektrum:



Obr. č. 2: Rozložení harmonických složek tónů různých hudebních nástrojů

Ukazuje se tedy, že například trubka nebo klavír mají dokonce třetí či druhou harmonickou výraznější než základní kmitočet. U kytary je sice první harmonická nejsilnější, ale jen o kousek. Většinou lze počítat s tím, že hudební nástroje mají nezanedbatelné amplitudy přibližně do 6 až 10 harmonického kmitočtu, tedy do 6ti až 10ti násobku základní kmitočtu. Pro univerzální přístup ke složení tónu můžeme zvolit hranici 7. harmonické.

1.2 Analýza signálů v reálném čase

Po převedení signálu do kmitočtové oblasti získáme přibližný obrázek o tom, z čeho všeho se signál skládá, ale bohužel ztratíme informaci o změnách tohoto složení v čase. Pro velmi přesné údaje o kmitočtovém složení je potřeba vzít co nejdelší úsek signálu, který ale bohužel může v čase hodně měnit. Při analýze reálných signálů se dá vždy najít jen kompromis mezi rozlišením v čase a rozlišením v kmitočtu. Další aspekt související s analýzou signálu v reálném čase je latence. Výsledek nikdy nemůžeme mít okamžitě a čím přesněji je třeba měřit, tím delší úsek signálu musíme vzít. Navíc

výpočet většího množství dat trvá déle. Je tedy třeba si rozmyslet zda je důležitější mít výsledek s co nejmenším zpožděním nebo mít výsledek co nejvíce přesný.

Odpověď na otázku, jaké zvolit rozlišení, se navíc komplikuje faktem, že hudební stupnice je geometrická posloupnost kmitočtů. Znamená to, že u hlubokých tónů potřebujeme mnohem větší rozlišení kmitočtů spektra, než u vysokých tónů. A pokud zvolíme za kritérium, aby ladička měla dobré rozlišení pro hluboké tóny, u vysokých tónů bude rozlišení zbytečně velké a velmi neefektivní pro výpočet. Je třeba najít tedy správný kompromis rozlišení v kmitočtu a podle toho zvolit teoreticky nejnižší tón, u kterého ladička ještě bude mít alespoň minimální rozlišení. Prakticky to znamená do algoritmu zařadit kmitočtový filtr, který omezí možný rozsah laděných tonů.

Nejefektivnější by bylo nastavit parametry algoritmu rozpoznání tónů pro každý hudební nástroj speciální, v této práci je ale snaha ladičku vytvořit spíše univerzální.

2 Algoritmus rozpoznání tónu

Nejdříve se nastaví parametry – vzorkovacího kmitočtu (f_s = 16 kHz), délka okna (300 ms, N_{win} = 4800 vzorků), délka posuvu okna (N_{shift} = ⅔ N_{win} = 3200 vzorků). Navzorkovaný zvukový signál postupně plní kruhový buffer, dokud nedosáhne velikosti N_{win}. Potom se spočítá rychlá Fourierova transformace a z bufferu se smaže N_{shift} nejstarších vzorků. U testovací verze algoritmu v MATLABu stačí mít vzorky celé zvukové nahrávky a jen okno po signálu posouvat o N_{shift} vzorků.

Pro zvýšení přesnosti rozpoznaného kmitočtu je použita metoda interpolace ve spektru [3]. Metoda je v této situaci výhodná. Za vzorky signálu jsou přidány nulové vzorky a tím FFT uměle zvětší počet výstupních vzorků, aniž by se musela zvětšit vzorkovací kmitočet f_s nebo prodloužit okno N_{win}. Lze takto přesněji rozlišit vrchol spektrální čáry tónu. Interpolace se hodí i pro splnění podmínky optimalizované funkce FFT, která většinou vyžaduje počet vstupních vzorků celou mocninu 2 nebo 4. Lze pak volit velikost okna N_{win} libovolně, nezávisle na počtu vstupních vzorků pro FFT. Po přidání nul je tedy počet vstupních vzorků:

$$N_{fft} = 16\,384 = 4^7 \quad (= 2^{14}) \tag{1}$$

Pro automatické dopočítání nejbližší vyšší mocniny pro jakoukoliv délku okna N_{win} lze použít vzorec:

$$N_{fft} = a^{\mathrm{ceil}(\log_a N_{win})} \tag{2}$$

kde funkce *ceil* znamená supremum a za *a* se dosadí 2 nebo 4, podle požadavku použité funkce FFT. Získané rozlišení výstupního kmitočtového spektra s interpolací, neboli rozdíl kmitočtů dvou nejbližších vzorků, lze jednoduše spočítat z počtu vzorků pro FFT:

$$f_{res} = \frac{f_s}{N_{fft}} = \frac{16\,000}{16\,384} \doteq 0{,}977 \text{ Hz} \tag{3}$$

FFT je komplexní funkce. Vstupní i výstupní vzorky jsou komplexní čísla, a proto je třeba reálné vzorky signálu doplnit o nulovou imaginární část. Výstupní spektrum je periodické a pro reálný vstupní signál je pravá polovina spektra jen zrcadlení první poloviny. Ze vzorkovacího teorému [4] také vyplývá, že v signálu by neměly být kmitočty vyšší než polovina vzorkovacího kmitočtu $f_s/2$ = 8 kHz. Konkrétně to zajišťuje kodek AIC31 na desce LCDK pomocí metody oversamplingu na $128f_s$, digitálním antialiasingovým filtrem s max. -3 dB do $0{,}45f_s$ a min. -75 dB za $0{,}55f_s$ a následně decimací zpět na f_s [5]. Pravá polovina spektra je pro reálný signál zrcadlová a není třeba ji uvažovat. Dále bude algoritmus pracovat pouze s levou polovinou spektra.

Po provedení FFT jsou výstupem komplexní vzorky spektra daného úseku signálu vymezeného oknem N_{win}. Jsou tedy k dispozici amplitudy i fáze jednotlivých složek akustického signálu. Informace o fázi není potřeba. U amplitudy je zajímavá spíše její relativní velikost oproti ostatním amplitudám. Spektrum může obsahovat mnoho lokálních maxim, viz předchozí kapitola. Pro správný výsledek je třeba vzorky spektra dále zpracovat. Pro další práci je použita absolutní hodnotou na komplexní vzorky a zbytek výpočtu algoritmu už je pouze v reálných hodnotách. Místo jednoduché funkce

absolutní hodnoty je u implementace na DSP použita pro výpočet amplitudy její definice s Pythagorovo větou:

$$|\boldsymbol{x}| = \sqrt{\Re\{\boldsymbol{x}\}^2 + \Im\{\boldsymbol{x}\}^2} \quad \rightarrow \quad \boldsymbol{x}^2 = \Re\{\boldsymbol{x}\}^2 + \Im\{\boldsymbol{x}\}^2 \tag{4}$$

Zjednodušením vzorce o odmocninu je počítáno rovnou výkonové spektrum, které by se jinak počítalo umocněním vzorků amplitudového spektra na druhou. Tento krok je výhodný tím, že zmenší výpočetní náročnost. Výkonové spektrum zároveň poslouží ještě lépe než amplitudové, protože více ve spektru zvýrazní spektrální čáry tónu oproti šumu na pozadí. V této fázi by algoritmus již rozpoznal mnoho tónů. Základní harmonický kmitočet u mnoha nástrojů bývá ten s největší amplitudou, ale není tomu tak vždy, viz předchozí kapitola.

Určitým mezikrokem algoritmu před provedením autokorelace je použití filtrů, které pomohou ve spektru potlačit amplitudu kmitočtů, které nenesou pro ladičku žádné důležité informace. Filtr horní propusti či spíše dolní zádrž potlačí příliš nízké kmitočty. Prakticky je jejich amplituda nahrazena nulou. Do tohoto intervalu patří kmitočty, jejichž perioda je větší než délka okna 300 ms, což odpovídá kmitočtu 3,3 Hz. Dále hluboké tóny, pro které je příliš malé rozlišení. Rozdíly kmitočtů mezi tóny se s klesajícím kmitočtem stále zmenšují až v nějaké hranici přestane stačit rozlišení FFT, která má rozestupy kmitočtového spektra konstantní. Příliš hluboký tón by pak mohl být mylně rozpoznán jako sousední tón. Pro ladičku je dobré mít alespoň 4 kroky na stupnici ladění pro jeden tón, tedy hranice je $3f_{res}$ = 2,9 Hz a takový rozdíl má od okolních tónů tón G_1 s kmitočtem 49 Hz. Filtr horní propusti mimo jiné potlačí i rušivý nízkokmitočtový brum, vznikající mechanickými otřesy mikrofonu snímajícího nástroj. Rozumná hranice horní propusti je, s ohledem na analýzu v minulé kapitole a rozlišením FFT, f_{min} = 40 Hz. Bude možno naladit i nejhlubší strunu basové kytary i když s horším rozlišením. Z druhé strany je spektrum upraveno filtrem doplní propusti či spíše horní zádrž. V tomto případě není třeba amplitudy vyšších kmitočtů nahradit nulou, ale stačí jen zmenšit počet vzorků a dále zpracovávat pole vzorků o menší velikosti. Kmitočty nad 20 kHz, které už lidský sluch vůbec neslyší, není třeba se zabývat. Nejvyšší tóny hudebních nástrojů můžeme nalézt na kmitočtu do 3,2 kHz, kap. 1. Pro správnou funkci autokorelace v následujícím kroku je nutné zachytit alespoň základní a druhý harmonický kmitočet tónu. Pro účely ladičky hudebních nástrojů, při malém omezení pro klavír a varhany, postačí nastavit horní mezní kmitočet f_{max} = 6,8 kHz. Ve výsledku oba filtry dohromady tvoří pásmovou propust. Je ale třeba zvolené hodnoty mezních kmitočtů f_{min} a f_{max} přepočítat na index vzdálenost vzorků od začátku pole reprezentující tyto kmitočty v diskrétním spektru. Tyto indexy se z rozlišení spočítají a zaokrouhlí na nejbližší celé číslo:

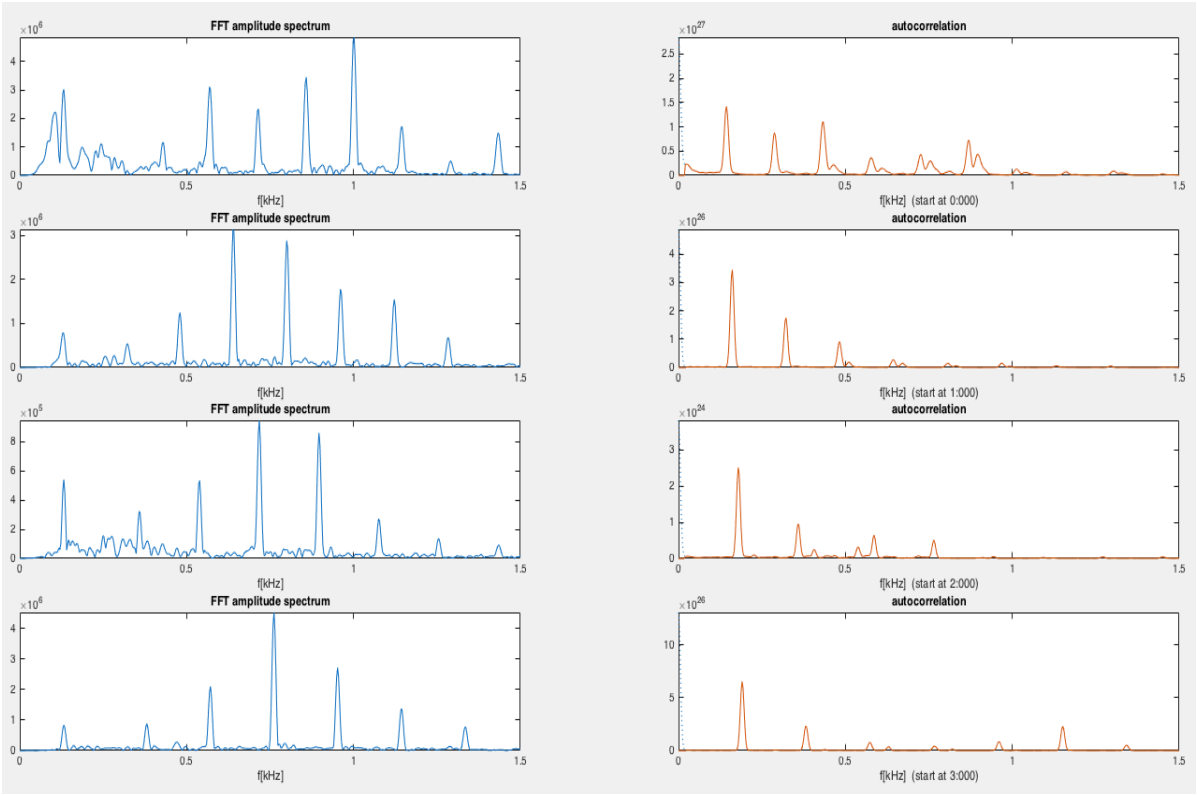
$$N_{min} = \frac{f_{min}}{f_{res}} = \frac{f_{min} \cdot N_{fft}}{f_s} = \frac{40 \cdot 16\,384}{16\,000} \doteq 41 \tag{5}$$

$$N_{max} = \frac{f_{max}}{f_{res}} = \frac{f_{max} \cdot N_{fft}}{f_s} = \frac{6\,800 \cdot 16\,384}{16\,000} \doteq 6963 \tag{6}$$

Hodnoty mezních kmitočtů filtru pro algoritmus se dají definovat jako nastavitelné konstanty a měnit pro různé nástroje a aplikace.

V dalším klíčovém kroku algoritmu je autokorelace, aplikovaná na vzorky filtry omezeného výkonového spektra. Účel tohoto kroku je přizpůsobit algoritmus ladičky pro ladění reálných hudebních nástrojů. Namísto hledání pouze základního kmitočtu tónu se algoritmus zaměří na všechny harmonické kmitočty obsažené v tónu zároveň. Je tedy schopný bez problémů správně rozeznat i tóny

nástrojů, které mají základní harmonický kmitočet velmi slabou nebo úplně potlačenou oproti některým ostatním vyšším harmonickým.



Obr. č. 3: Porovnání amplitudového kmitočtového spektra a výstupem autokorelace

Při zpracování spektra tónu autokorelací vzniknou stejně vzdálené spektrální čáry, ale oproti výkonovému spektru se seřadí podle velikosti. Lépe nežli text toto demonstruje obrázek č.3. Většina výkonu všech harmonických složek se promítne do první čáry, která je umístěna přesně v bodě základního harmonického kmitočtu tónu. Ve výsledném průběhu po autokorelaci je možné si všimnout největšího lokálního maxima v nulovém bodě. Z definice autokorelace [5] je hodnota v nule vždy globální maximum a je rovna celkové energii vstupního signálu. To ale zde neplatí, protože vstupem autokorelace není signál jako takový, ale výkonové spektrum signálu. Lokální maximum v nule nenese pro algoritmus žádnou užitečnou informaci. Je třeba celý vrchol v nule nahradit nulovými hodnotami, aby vyniklo další lokální maximum, které je pro algoritmus klíčové. Pro velkou většinu případů lze vrchol v nule smazat cyklem *for*, který projde všechny hodnoty od začátku. Přepisuje se nulou každá aktuální hodnota, dokud platí podmínka, že aktuální hodnota je větší následující 3 hodnoty.

Dále je použita funkce maximum vracející index nalezeného maxima. Získaný index pořadí maximální hodnoty v poli N_i je vlastně diskrétní hodnota kmitočtu, který algoritmus ladičky hledá. Nakonec je ještě třeba přepočítat výsledek na kmitočet f_x [Hz], opět pomocí rozlišení FFT:

$$f_x = N_i \cdot f_{res} = N_i \cdot \frac{f_s}{N_{fft}} \doteq N_i \cdot 0{,}977 \text{ Hz} \tag{7}$$

Posledním krokem je přepočítat kmitočet na hudební stupnici [6]. Stupnice je vyjádřena vzdáleností n v půltónech od tónu A_4, kde f_{A4} = 440 Hz. Rovnice definující kmitočty jednotlivých tónů je:

$$f_x = f_{A4} \cdot 2^{\frac{n}{12}} \quad \rightarrow \quad n = 12 \cdot \log_2\left(\frac{f_x}{f_{A4}}\right) = 12 \cdot \log_2\left(\frac{f_x}{440}\right) \tag{8}$$

Kmitočty tónů tedy tvoří geometrickou posloupnost a každá oktáva má dvojnásobný kmitočet oproti předchozí oktávě, viz obr. 1.

2.1 Optimalizace algoritmu

U algoritmu rozpoznání tónů implementovaného v MATLABu není potřeba příliš dbát na optimalizaci kódu, protože pracuje pouze s nahrávkou hudebních nástrojů uložených v souboru *.wav. Implementace v MATLABu nepracuje v reálném čase. Při implementaci algoritmu v C pro DSP je však třeba stihnout spočítat každé okno signálu dříve, než do bufferu přijdou všechny vzorky dalšího okna.

Je několik základních kroků, jak zrychlit výpočet, aniž by se snížila přesnost výsledků. První možností je použít pro FFT a autokorelaci knihovní optimalizované funkce podrobněji rozebrané v kapitole č. 4. Výpočty FFT a autokorelace, i bez údajů z profillingu, s velkou pravděpodobností zabírají hodně procesorového času a je tedy třeba se na ně soustředit. Mimo jiné hlavně proto, že jejich výpočetní náročnost roste někdy i kvadraticky s počtem vzorků N. Pro DSP je použita funkce FFT, která počítá efektivně s podmínkou, že počet vzorků N_{fft} je mocnina 4. Přidání nulových vzorků pro interpolaci ve spektru se v tomto případě velmi hodí. Funkce FFT dostane potřebný počet vzorků N_{fft} a zároveň aplikace ladičky nebude omezená na tak hrubé kroky pro velikosti okna N_{win} . Při testování různých velikostí N_{fft} v poměru k velikosti okna N_{win} se ukázalo jako nejlepší řešení nejbližší vyšší mocnina 4. Další zvětšování už způsobuje příliš široké rozmazané spektrální čáry, které pak výsledky ladičky už příliš nevylepší.

Další velkou úsporu strojových cyklů procesoru lze dosáhnout, když algoritmus zbytečně nepočítá kmitočty mimo ohraničený interval pro hudební nástroje. Nejen, že ve vysokých kmitočtech je čím dál menší hustota tónů, ale navíc náročnost výpočtu FFT vzrůstá s rostoucím počtem vstupních vzorků. To jsou hned dva důvody, proč dobře zvážit volbu vzorkovacího kmitočtu. Pro autokorelaci je dokonce náročnost Θ kvadraticky strmá a větší než u FFT. Protože nastavení vzorkovacího kmitočtu u implementace v C pro DSP má poměrně hrubé kroky dané kodekem AIC31 a pro implementaci v MATLABu vzorkovací kmitočet závisí na souboru *.wav, je třeba maximální kmitočet f_{max} doladit filtrem dolní propusti. Opravdové zkrácení počtu zpracovávaných vzorků je zde potřeba, oproti jednoduchému „vynulování“ vzorků, jako je to u filtru horní propusti. Autokorelace už pak počítá pouze N_{max} vzorků namísto původních $N_{fft}/2$ vzorků.

Dalším krokem v optimalizacích pro rychlejší výpočet je profilling. V této oblasti však zvolený procesor klade velké nástrahy. Procesorové jádro C6748 obsahuje speciální hardware, např. cache, který zkresluje či znemožňuje běžné metody debugového profillingu kódu aplikace bez operačního systému. Vytvářet aplikaci ladičky jako proces operačního systému by bylo pro účely této práce příliš složité a tak tedy nezbývá nic jiného, než nejjednodušší metoda profillingu počítáním strojových cyklů. Ve vývojovém prostředí CCS lze použít funkci debuggeru „Clock“. To se ale ukázalo jako nespolehlivé, protože často vypisuje nesmyslné hodnoty a není jisté zda 32bit čítač během měřené doby nepřetekl. Při 456 MHz taktování DSP jádra, je to asi 9 sekund. Spousty neúspěšných pokusů o profilling C6748 DSP si vynutily naprogramování vlastního kódu modulu uvedeného v příloze č. 4.

Tento modul sám pomocí hardwarového 64bit čítače průběžně počítá cykly strávené ve vymezených blocích kódu aplikace. Po nějakém počtu zprůměrovaných měření následně vypíše do debugové konzole přehledně zformátované výsledky. Zajímavostí je, že funkce pro vypsání textu do debugové konzole samotná pak zabere necelou sekundu a tím na chvíli způsobí latenci v aplikaci ladičky, která promešká spočítání několika oken signálu. Je třeba ještě před další statistikou všechny údaje profilling modulu i čítač automaticky vyresetovat.

Ve chvíli, kdy jsou k dispozici profillingové informace lze začít zkoušet, na jakou přesnost je schopný algoritmus stihnout vše počítat v reálném čase. První kritický bod výpočtu byl nalezen ve způsobu počítání druhé mocniny při převodu z komplexního do amplitudového či výkonového spektra. Původně byla použita funkce `pow()` ze standardní knihovny jazyka C a nahrazením za x*x se výpočet řádově zrychlil. Další velké skoky v rychlosti byly zaznamenány po použití interní paměti L2 namísto externí DDR2 RAM. Programovou část kódu aplikace lze celou umístit do blízké paměti, ale s daty už to není tak jednoduché. Z celkového množství přes 1 MB dat lze do 256 kB L2 paměti umístit jen menší datové struktury. Je třeba zkoušet, která data mají vliv na rychlost výpočtu a která ne. Ukázalo se, že je výhodné přesunout pole vzorků pro vstup a výstup autokorelace a také výstupní pole FFT. Pro funkci FFT existují 3 stejně velká pole – vstupní vzorky, twiddle factors a výstupní vzorky. Všechny tyto pole mají dvojnásobnou velikost $2N_{fft}$, kvůli imaginární a reálné části komplexních vzorků. Bohužel se do interní paměti L2 vejde pouze jedno ze tří polí. Největší efekt vyvolá přesunutí výstupního pole. Přesuny některých částí programu a dat do interní paměti L2 posunuly efektivitu výpočtu algoritmu ještě přibližně o další řád. Zde je výsledek optimalizací a statistika, kolik který výpočet zabírá procent procesorového času:

```
Rejnok Tuner v2  fs=16.0kHz  FFT_rozliseni=0.98Hz (3.33Hz)
---- window=4800, shift=3200(=200ms), Nfft=16384, NHCF=6640(fmax=6.8kHz) ----
Profiling statistics (100 cyklu): 28.572% nevyuzito
kopirovani bufferu, priprava pro FFT: 2.096%
CPU meter + level meter: 1.001%
DSPF_sp_fftSPxSP(): 19.306%
komplexni spektrum -> vykonove spektrum: 0.443%
filtry, normalizace urovne: 0.643%
alternativni algoritmus: 2.950%
DSPF_sp_autocor(): 44.522%
smazani vrcholu v nule: 0.002%
DSPF_sp_maxidx(): 0.013%
vykreslovani obrazovky (ladicka + noty): 0.447%
==== cycklu_celkem: 91200440 (456MHz) -> 200ms ====
```

2.2 Omezení algoritmu

V algoritmu použitá autokorelace hledá ve výkonovém spektru alespoň první a druhý harmonický kmitočet. Proto algoritmus paradoxně vůbec nerozpozná umělý tón, který tvoří jediný sinusový průběh bez žádného vyššího harmonického kmitočtu. Žádný hudební nástroj však takové spektrum tónu nemá, takže toto omezení není závažné. Pouze při testování algoritmu generátorem signálů je třeba si na to dát pozor a nastavit pilový průběh namísto sinusového. Obdélníkový signál se také příliš nehodí, protože zase obsahuje čistě liché násobky harmonických kmitočtů a algoritmus má tendenci mylně rozpoznat tón o jednu oktávu vyšší.

Při zvyšování horního hraničního kmitočtu filtru dochází k velkému zvýšení výpočetní náročnosti autokorelace, a proto je nutné nastavit f_{max} co nejníže, ale s ohledem na rozsah laděného nástroje. Je zde i spodní hraniční kmitočet f_{min}, který téměř neovlivní náročnost výpočtů. Slouží spíše pro potlačení

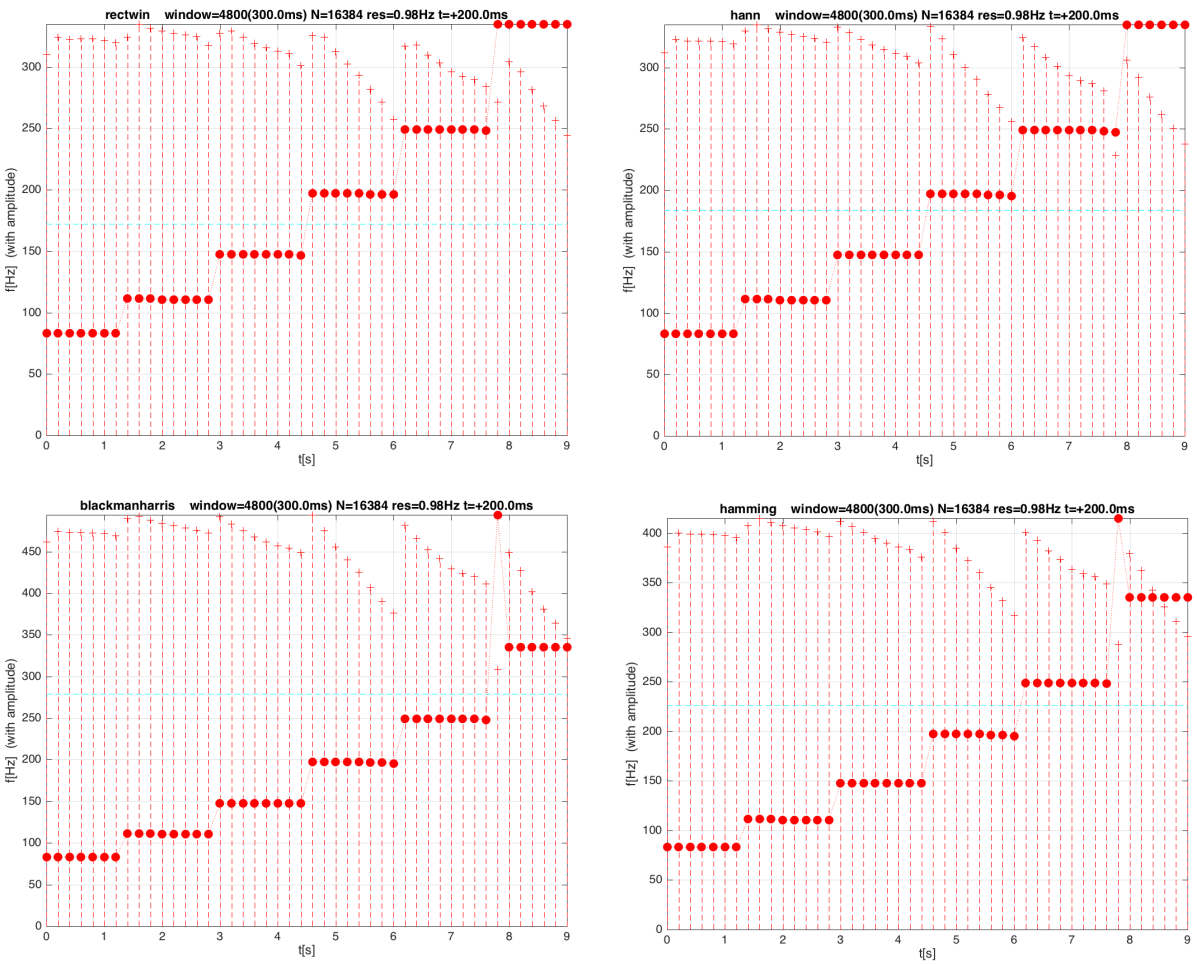
rušivých signálů s velmi nízkým kmitočtem, které by více či méně mohly kazit výsledky ladičky. Horní a spodní kmitočet tvoří uměle přidané omezení algoritmu.

Spodní kmitočet filtru složené pásmové propusti zároveň souvisí s dalším důležitým omezením algoritmu. S klesajícím kmitočtem rozpoznaných tónů klesá i rozlišení FFT v algoritmu. I kdyby byl filtr pásmové propusti vyřazen, bude u nízkých kmitočtů vznikat jen několik málo rozlišovacích kroků kmitočtu na jeden tón. Nebo dokonce některé tóny budou přeskakovány, což je pro ladičku velmi nevhodné. Radši rozumnou volbou dolní hranice filtru pásmové propusti těmto chybám algoritmus předejde.

Na další limit rozpoznávacího algoritmu lze narazit ve chvíli, kdy se z údajů o tónu vytváří notový zápis. V tuto chvíli se už na vstupu může objevovat spíše zvukový signál s více tóny či tóny jdoucími po sobě a znějícími přes sebe, tedy zní ještě i v době, kdy už hrají další tóny. V případě tónů doznívajících přes sebe algoritmus s autokorelací občas vyhodnotí tón chybně jako tón o kmitočtu odpovídajícímu rozdílu dvou znějících tónů nebo ještě hlubší. Při hraní celých akordů, 2 a více tónů najednou, už tento algoritmus vhodný pro aplikaci ladičky akustických tónů nestačí a selhává. Je tedy lepší pro přepis tónů do notové osnovy použít upravený algoritmus bez autokorelace popsaný v kapitole 2.2.1.

2.3 Modifikace a rozšíření algoritmu

Algoritmus nemusí striktně používat pravoúhlé okno pro výřez části signálu. Je na výběr několik jiných tvarů oken. U pravoúhlého okna ale není třeba nic násobit, stačí výřez signálu nechat tak jak je. Pro algoritmus bylo proto vybráno pravoúhlé okno. Při testování různých oken se ukázalo, že na výsledky mají vliv jen velmi málo a kromě pravoúhlého a Hannova okna výsledky spíše nepatrně zhorší:



Obr. č. 4: Porovnání algoritmu s použitím oken: pravoúhlé, Hannovo, Blackman-Harissovo a Hammingovo – nahrávka strun kytary E_2, A_2, D_3, G_3, H_3, E_4

Na grafech je zobrazen výstup algoritmu implementovaného v MATLABu. Algoritmus načetl pokaždé stejný soubor s nahrávkou 6 strun kytary zahraných po sobě, jen byl pozměněn typ okna. Přerušované čáry s křížky na konci znamenají relativní amplitudu v decibelech a slouží pouze k ilustraci hlasitosti příslušných rozpoznaných tónů značených kolečkem.

2.3.1 Alternativní algoritmus pro polyfonní zvuk

Pro lepší práci s polyfonními zvuky, kde hraje více tónu najednou, je vytvořen jiný algoritmus. Tento alternativní algoritmus rozpoznání tónu používá amplitudové spektrum stejně jako původní algoritmus. Místo autokorelace je zde ale jiný princip. Polyfonní algoritmus hledá v amplitudovém spektru vrcholy spektrálních čar všech silných kmitočtů, třídí je a snaží se najít tóny podle všech jeho harmonických kmitočtů.

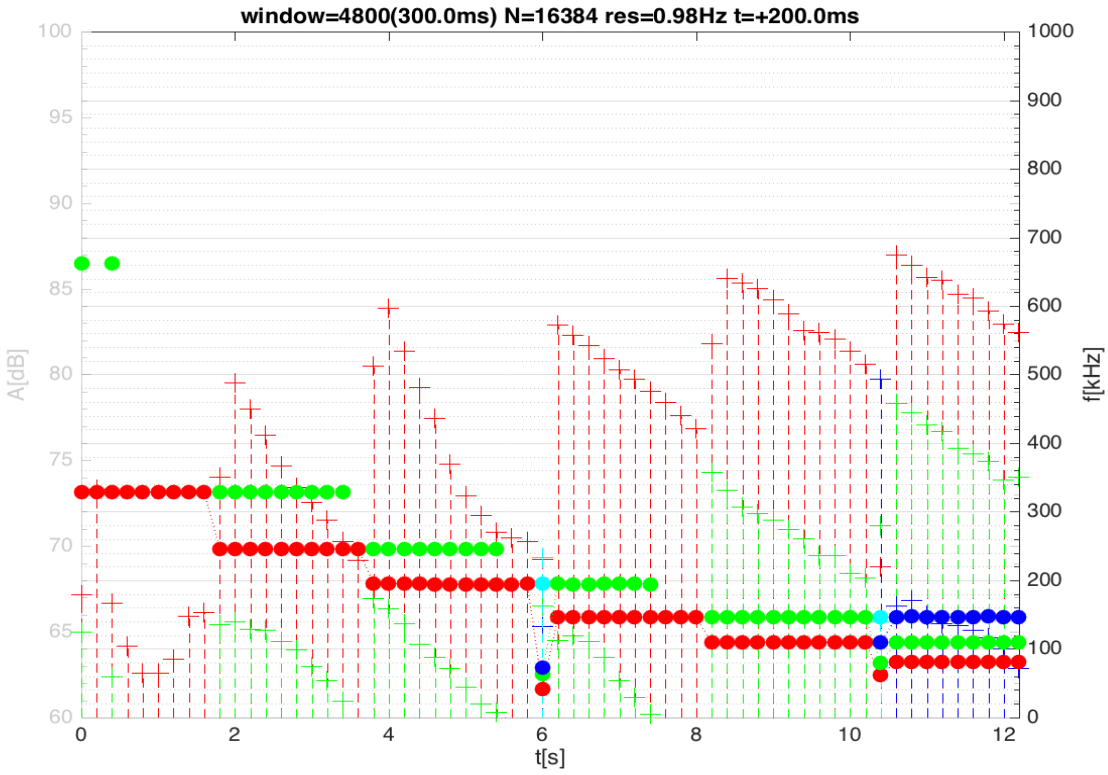
Postup je následující. Nejdříve se najde maximum v amplitudovém spektru signálu. Potom se zvolí nastavitelné parametry polyfonního algoritmu, jako šířka lokálního maxima n_w, minimální amplituda v decibelech A_{dBmin} a maximální počet tónu hraných najednou n_{poly}. Postupně se projíždí všechny vzorky spektra a hledají se lokální maxima splňující dvě podmínky. První podmínka je, že hodnota musí být větší než n_w sousedních hodnot na obě strany. A druhá podmínka je, že hodnota musí být větší než $A_{min} = 10^{\frac{A_{dBmin}}{20}}$. Pokud jsou obě podmínky splněny, vloží se hodnota a její index

pořadí na konec pole obsahujícího lokální maxima jako dvojice index a hodnota. Po dokončení celého amplitudového spektra máme prakticky pole seřazených lokálních maxim s hodnotou kmitočtů i jejich amplitudou.

V dalším kroku je provedena extrakce vyšších harmonických kmitočtů. Znamená to, že každý nalezený kmitočet se porovná se všemi ostatními nalezenými kmitočty z předchozího kroku. Pokud je jeden kmitočet násobkem druhého, ubere se amplituda vyššího harmonického kmitočtu a celá se přičte do amplitudy základního harmonického kmitočtu. Tímto krokem by měly v poli lokálních maxim zůstat s nenulovou hodnotou amplitudy pouze základní harmonické kmitočty všech tónů. Demonstruje to příloha č. 5.

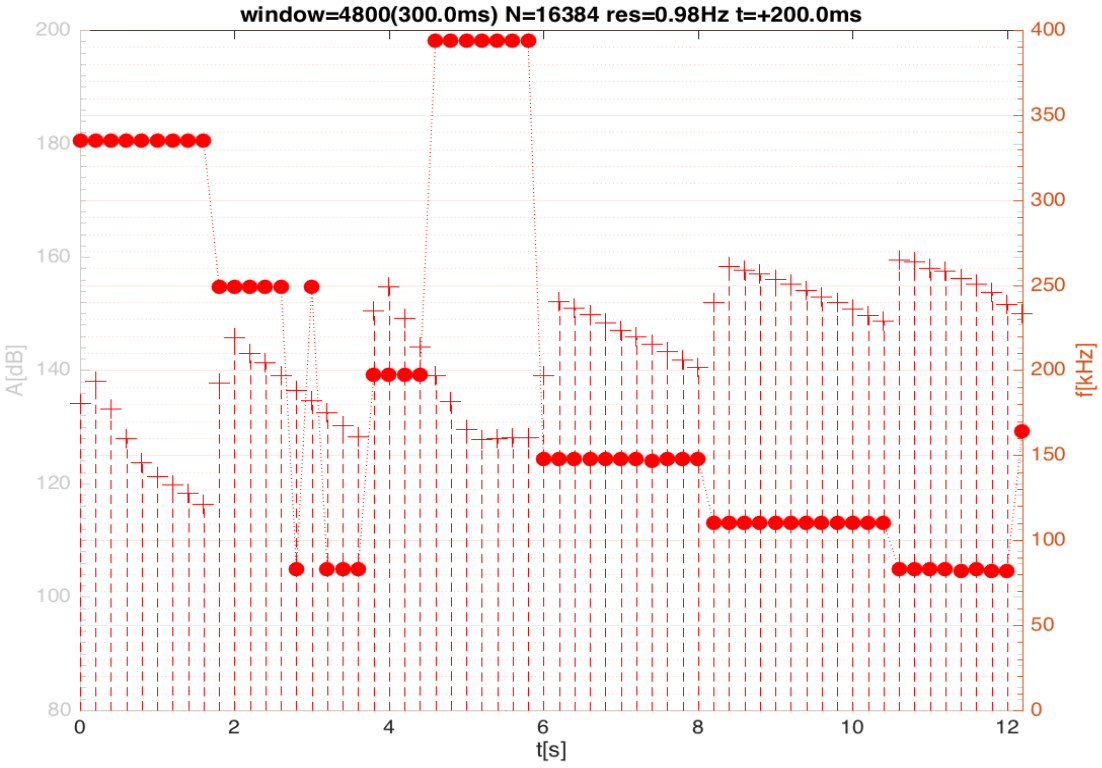
V posledním kroku lze postupně vybrat n_{poly} kmitočtů s největší amplitudou a ty pak ještě přepočítat na vzdálenost půltónů (8). Pro potřeby aplikace a zápisu not do notové osnovy je nastaveno $n_{poly} = 1$.

Zde je ukázka, která demonstruje, jak polyfonní algoritmus dokáže zachytit 6 tónů postupně hraných a doznívajících přes sebe. Konkrétně je to 6 strun kytary hraných od nejvyššího tónu:

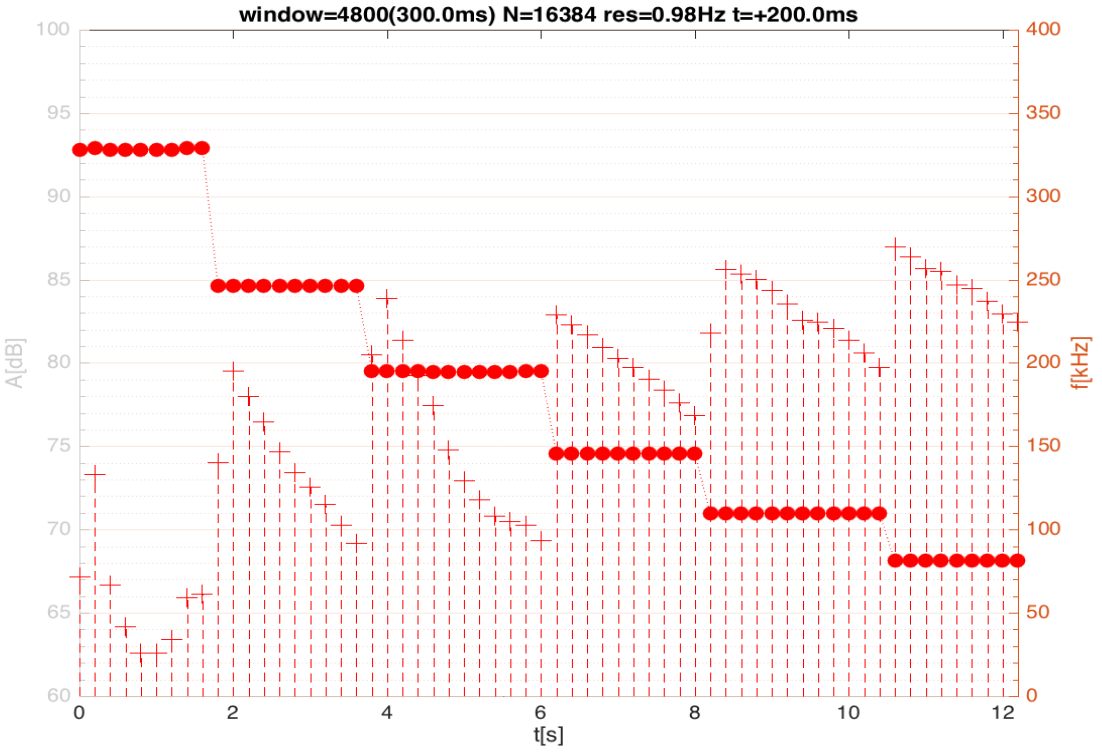


Obr. č. 5: Grafické zobrazení výstupu polyfonního algoritmu (s $n_{poly} = 4$) - struny E_4, H_3, G_3, D_3, A_2, E_2

Pro porovnání je zde i simulovaný výstup algoritmu s autokorelací a výstup polyfonního algoritmu s pouze jedním tónem. Zpracována je stejná nahrávka zvuku:



Obr. č. 6: Grafické zobrazení výstupu algoritmu s autokorelací - struny E_4, H_3, G_3, D_3, A_2, E_2



Obr. č. 7: Grafické zobrazení výstupu polyfonního algoritmu (s $n_{poly} = 1$) - struny E_4, H_3, G_3, D_3, A_2, E_2

2.4 Zdrojový kód v MATLABu a ukázka kódu v jazyce C

Nejprve je zde uveden základ algoritmu implementovaného v jazyce MATLAB. Zbytek kódu je k nalezení v příloze č.1.

```
% MAIN LOOP - compute FFT for all windows: ============
amp_tab=zeros(1,cnt); f_tab=amp_tab; f_tab_a=[]; amp_tab_a=[]; % containers initialization
for (k = 0:cnt-1)
    start = 1+k*shift;
    % FFT of signal window (with zero padding) -> amplitude spectrum:
    y = fft(x(start:start+window-1),N);%.*hann(window)',N); %  <------
    s1 = abs(y(1:N/2)); % convert half of complex spectrum to abs.values
    s1 = s1*(2/window); % normalize amplitude (window instead of N, cause FFT pading)
    % band-pass filter:
    s1(1:imin) = 0; % /¨¨ LC filter
    s1(imax+1:length(s1)) = 0; % ¨¨\ HC filter
  % ALTERNATIVE ALGORITHM for polyphone input signals: %%
    [f_tab_a0,amp_tab_a0] = c6748_alt_algorithm(s1,polyphone_max,db_min,...
            peak_mode,resolution,plotnowif,xmax); % compute the rest down by different way
    f_tab_a = [f_tab_a f_tab_a0']; % add results in matrix columns
    amp_tab_a = [amp_tab_a amp_tab_a0'];
  % end of ALTERNATIVE ALGORITHM %%
    % AUTOCORRELATION (of power spectrum):
    s1 = s1.^2; % power spectrum
    c = xcorr(s1); % <------
    c = c(round(length(c)/2):length(c)); % erase left mirror half of autocorr
    % erase first unwanted peak (starting at zero freq):
    p = 1;
    while (c(p)>c(p+1) || c(p)>c(p+2) || c(p)>c(p+3))
        c(p) = 0;
        p = p+1;
    end
    % find next first peak -> fundamental frequency:
    [m,i] = max(c);
    m = sqrt(m); % compensation of autocorrelation (not exact! dB->dB* )
    amp_tab(k+1) = m; % write amplitude of found fund.frequency
    % erase too small signal -> NaN
    if (m<min_signal || i<=imin) % (with addition of LC filter)
        f_tab(k+1) = NaN; amp_tab(k+1)=0;
    else
        f_tab(k+1) = i*resolution; % convert peak freqency and write
    end
end
```

Manuálně nastavitelné parametry algoritmu jsou `window` (N_{fft}), `shift` (N_{shift}), `fmin` (f_{min}), `fmax` (f_{max}) a `min_signal`, což je hladina intenzity ticha pod kterou se tón nahradí pomlkou. Výstupem jsou pole: `f_tab` - hodnoty základních kmitočtů rozpoznaných tónů, `amp_tab` - hodnoty amplitud vztahujících se k rozpoznaným tónům a `n_tab` - konvertované tóny do hudební stupnice v počtu půltónů vzdálenosti od tónu A_4. Řádek počítání pole půltónů je v části kódu, který vykresluje grafy:

```
n_tab = round(12*log2(f_tab/440)); % convert from f to discrete halftones from A4
```

Pro ukázku je zde i úsek kódu v C, který je pak zkompilován a spouštěn na desce s OMAP-L138, přesněji na jádře C6748:

```
DSPF_sp_fftSPxSP(NFFT, x_sp, w_sp, y_sp, brev, 4, 0, NFFT); // <-FFT-----
```

```
PROF_section();/******profilling******/
int m, o;
for (m=0,o=0; m<NSP; ++m) { // only left half of spectrum and only real values
    amplituda[m] = y_sp[o]*y_sp[o] + y_sp[o+1]*y_sp[o+1]; // power spectrum
    o+=2;
}
PROF_section();/******profilling******/
for (m=0; m<LCF_IDX; ++m) amplituda[m]=0; // LowCut filter (...also later, after autocor)
for (; m<HCF_IDX; ++m) amplituda[m]*=1e-9; // scaling against float overflow
float * phy_sp = &y_sp[HCF_IDX]; // parallely fill right half
for (m=0; m<HCF_IDX; ++m) { // (with HighCut filter)
    y_sp[m] = 0; // for DSPF_sp_autocor() need, using bigger y_sp (not used this time)
    phy_sp[m] = amplituda[m];
}
PROF_section();/******profilling******/
DSPF_sp_autocor(amplituda, y_sp, HCF_IDX, HCF_IDX); // <-autocorr-----
PROF_section();/******profilling******/
for (m=0; m<LCF_IDX; ++m) amplituda[m]=0; // LowCut filter (addition)
for (; amplituda[m]>amplituda[m+1]||amplituda[m]>amplituda[m+2]||
        amplituda[m]>amplituda[m+3]; ++m) amplituda[m]=0; // 1st DC peak erase
PROF_section();/******profilling******/
amplituda[0] = MIN_LVL; // minimum level for peak searching
vysledek = DSPF_sp_maxidx(amplituda,HCF_IDX);
PROF_section();/******profilling******/
vyslfreq = (float)SAMPLING_RATE/NFFT*vysledek;
```

Výstupem této implementace je rozpoznaný kmitočet `vyslfreq`. O zbytek už se starají funkce grafického výstupu ladičky. Tam je kmitočet přepočítán na půltóny. Následně se mu přiřadí název a číslo tónu ve stupnici. Nezaokrouhlená hodnota `vyslfreq` slouží jako přesný kmitočet pro ladičku. Manuálně nastavitelné parametry algoritmu lze měnit jako preprocesorové konstanty:

```
#define SAMPLING_RATE (16000u) //Hz
#define WINDOW (4800u) // samples to use (must be smaller than NFFT!)
#define WIN_SHIFT (WINDOW*2/3)
#define NFFT (16384u)   //radix-4!
#define LOWCUT_FILTER_FREQ (40u) //Hz   (82Hz guitar_min, 41Hz bass_min, 28Hz piano_min)
#define HIGHCUT_FILTER_FREQ (6800u) //Hz  (note: more computing when high!)
#define MIN_LVL (1e9f)
#define NUM_PROFILE_CYCLES (100) // profiling precision (average from x cycles of FFT)
/***** CONSTANTS ************************************************************/
#define NSP (NFFT>>1) // spectrum size (half used)
#define LCF_IDX (LOWCUT_FILTER_FREQ*NFFT/SAMPLING_RATE)
#define HCF_IDX (HIGHCUT_FILTER_FREQ*NFFT/SAMPLING_RATE)
```

Jako poslední ukázka je v příloze č. 4 uveden C kód vlastní knihovny autora práce, která zajišťuje potřebný profilling zabudovaný do aplikace a statistiky běhu aplikace na DSP.

3 Aplikace a její grafický výstup

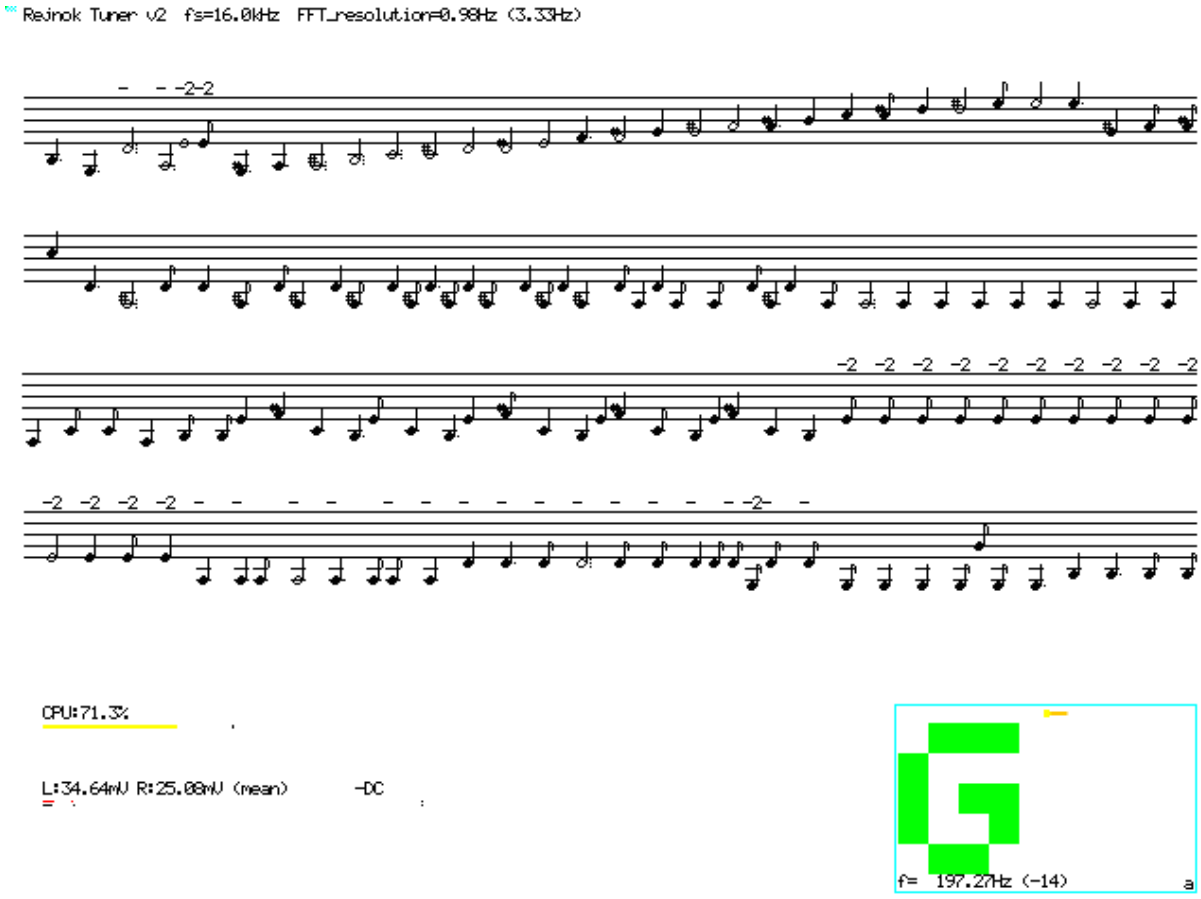
Základem aplikace ladičky akustických tónů je popisovaný algoritmus rozpoznání tónů, který je vyvíjený v MATLABu. Následně je přepsaný do jazyka C a optimalizovaný pro běh v reálném čase na desce s DSP. Zároveň tato aplikace umožňuje v reálném čase rozpoznané tóny zapisovat do zjednodušené notové osnovy. Vstupem aplikace je zvukový signál snímaný přes mikrofon nebo snímač hudebního nástroje, například elektrické kytary. Fyzicky je připojen přes linkový jack konektor umístěný na desce vývojového přípravku OMAP-L138 LCDK. Dále je analogový signál zpracován kodekem AIC31 a v podobě diskrétního signálu zpracováván na DSP, kde běží tato aplikace. Výstupem aplikace je grafické zobrazení ladičky nebo i notového zápisu na VGA monitoru. Fyzicky je připojen ke konektoru D-Sub na desce vývojového přípravku. Obrazový výstup aplikace průběžně posílá obsah pixelové mapy reprezentované v kódu aplikace polem 640x480 hodnot. Signál pro rastrový displej je převeden na analogový signál pro VGA monitor pomocí integrovaného převodníku na desce přípravku.

Aplikace používá pouze pravý kanál stereo signálu. Vzorky vstupního signálu jsou zpracovávány pomocí algoritmu popsaného v předchozí kapitole, pravidelně jednou za 200 ms. Tato doba je závislá na nastavené velikosti N_{shift}, v tomto případě 3200 vzorků. Zpoždění výsledků, latence, se pohybuje od 300 do 500 ms. Spodní hranici udává doba naplnění bufferu. Dokud vzorků není kompletní počet N_{win}, nelze začít počítat. Horní hranice má navíc dobu 200 ms, do které je třeba stihnout spočítat poslední okno a začít počítat další. Tato latence není malá, ale pro účely ladičky i zápisu do not je v pořádku.

Je třeba definovat hladinu hlasitosti signálu, do které je tón příliš slabý a výsledky mohou být zkresleny šumem z pozadí. Nejlépe toho lze dosáhnout nastavitelnou hodnotou minimální amplitudy a tu zapsat namísto první hodnoty v poli, ze kterého se pak hledá maximum v poslední fázi algoritmu rozpoznání tónu. Tím se docílí, že pokud maximální amplituda tónu není větší než tato nastavitelná hodnota, funkce maximum namísto toho vyhodnotí tuto uměle vloženou hodnotu jako maximum. Rozpoznaný kmitočet tedy bude roven nule, což za normálních okolností nemůže nastat, kvůli filtru s mezním kmitočtem f_{min}. Nakonec stačí jednoduchá podmínka – pokud je výsledek algoritmu nula, přesněji řečeno kmitočet 0 Hz, znamená to, že se nezobrazí žádný tón. Ladička je deaktivována a barva přesnosti naladění nebude ani zelená, ani červená, ale šedá. Notový zápis pro nulový kmitočet zastaví vypisování not a čeká, dokud se zase neobjeví signál s dostatečně silným tónem.

Grafický výstup aplikace na LCD zobrazuje vpravo dole menší rámec s velkým písmenem tónu a nad ním barevný ukazatel přesnosti naladění. Přes většinu obrazovky se postupně vypisují tóny do notové osnovy. Vlevo dole se zobrazuje aktuální hlasitost zvukového signálu na vstupu i s indikátorem přebuzeného vstupu. Na LCD se vypisuje také ukazatel aktuálního vytížení procesoru. Zdrojem této informace je „profilling“ modul starající se o statistiky procesorového času strávených v jednotlivých částech kódu algoritmu a o výpis zprůměrovaných hodnot do debugové konzole v CCS.

Pro účely vypisování jakýchkoliv textů na displej byla pro aplikaci vytvořena funkce, která vypíše jakýkoliv ascii řetězec na zadanou souřadnici na displeji. Používá tabulku bitové masky všech znaků. Je použito volně dostupného hlavičkového souboru s definicí všech znaků od Pascala Stanga původně pro AVR MCU. Touto funkcí je například vypsán na začátek obrazovky název aplikace, její verze a kmitočtové rozlišení.



Obr. č. 8: Výstupní obrazovka aplikace vyfocená za běhu

Barvy všech prvků na obrazovce jsou definovány jako preprocesorové konstanty. Definované jsou hlavně základní barvy s jinak těžko čitelným 16-ti bitovým zápisem. Červená složka má prvních 5 bitů, zelená prostředních 6 bitů a modrá posledních 5 bitů. V kódu aplikace pak už stačí jen použít tyto nadefinované barvy pomocí lidsky čitelného slova.

3.1 Ladička akustických tónů

Rámec ladičky je samostatný modul aplikace, který jako vstup získá přesnou nezaokrouhlenou hodnotu v půltónech i hodnotu kmitočtu z algoritmu rozpoznání tónu. Kmitočet je pro informaci vypsán malým písmem na spodu ladičky. Po vydělení hodnoty půltónů 12 je použit zbytek, který může nabývat 12 různých hodnot a těm se dá jednoduše přiřadit do dvojice název tónu. Ten se skládá z písmene tónu a případně křížku. Tento řetězec je pak zobrazen velkými znaky na displej uprostřed rámečku ladičky. Pokud algoritmus rozpoznání tónu vrátí nulový kmitočet, není přepočítán na hodnotu v půltónech, ale namísto toho zůstane text ladičky prázdný. Nejdůležitější součástí ladičky je ukazatel rozladění tónu. Je to hnědý pruh vysouvající se z barevného čtverce uprostřed. Ukazuje, jak moc je rozpoznaný tón rozladěn oproti teoretickému kmitočtu. Pruh je umístěn přímo nad velkým textem tónu. Protahuje se doleva či doprava, podle toho, na kterou stranu je tón rozladěn, a jeho délka odpovídá míře této odchylky. V levé krajní poloze je tón příliš hluboký a v pravé části příliš vysoký. Kmitočet přepočítaný na půltóny odpovídá hraniční hodnotě -0,5 od nejbližšího teoretického tónu. V pravé krajní poloze je tón naopak příliš hluboký. Kmitočet přepočítaný na půltóny odpovídá hraniční hodnotě +0,49 od nejbližšího teoretického tónu. Uprostřed je čtverec, které se zbarvují červeně, pokud

je tón velmi rozladěn, oranžově pokud je tón mírně rozladěn, zeleně pokud je tón naladěn velmi přesně. Šedá barva se objeví, pokud ladička není aktivní, tedy není nalezen dostatečně silný tón.

3.2 Notový zápis

Jako zdroj dat pro notový zápis se více hodí polyfonní algoritmus, kap. 2.2.1. Algoritmus vypočítá pouze první nejsilnější tón a ten předá aplikaci k zápisu do notové osnovy. Zápis je spíše demonstrační, protože implementace všech pravidel a detailů by byla v této situaci rozsahem na další celou práci. Navíc všechny noty, linky a znaménka se kreslí pixel po pixelu od základu. Pro účely této práce byly implementovány pouze nejnutnější značky a pravidla notového zápisu.

V principu notový zápis v aplikaci funguje tak, že se vytvoří nový blok konstantní šířky, kam se postupně vykreslí nota i se všemi značkami k ní potřebné, v závislosti na výšce a délce noty. V intervalu 200 ms přichází nová algoritmem vypočítaná hodnota tónu. Postupně dochází buď k prodlužování délky poslední noty, pokud je tón stále stejný, nebo k vytvoření nové noty s nejkratší délkou, tedy osminová nota. Maximální délka noty je 8x osminová, ta se nazývá celá nota. Pokud délka noty dosáhne maxima, nelze ji už více prodloužit a je namísto toho vytvořen nový blok s osminovou notou. Počet vytvořených bloků je počítán v proměnné. Pokud se další blok nevejde na obrazovku, vytvoří se nový blok na začátku dalšího řádku a počítadlo bloků se vynuluje. Je třeba sledovat počet řádků notového zápisu, aby nedošlo k překryvu s rámcem ladičky či dalších ukazatelů na levé spodní části obrazovky. Při dosažení definovaného maximálního počtu řádků se další bloky s notami už nevykreslují, což zabrání i vykreslování pixelů mimo obrazovku. Tím je vyřešeno riziko přepsání hodnot v paměti RAM mimo prostor pole pixelů obrazovky a následného selhání programu. Jazyk C totiž nemá žádnou ochranu proti neplatným indexům prvků pole, ležícím mimo jeho alokovaný prostor.

Nově vytvořený blok má jako výchozí bod pro další kreslení levý horní roh. Podle něj se pak vykresluje vše ostatní v bloku, kde souřadnice už je pouze relativní k základnímu pixelu bloku. Nejdříve se vykreslují pixely pěti notových linek, pro jakýkoliv blok stejné. Jako další se určí základní bod v bloku pro vykreslování noty a jejích značek. Tento bod má vertikální souřadnici závislou na hodnotě výšky tónu. Poněkud složitější je fakt, že tato závislost není lineární a není ani žádnou jednoduchou matematickou funkcí vyjádřitelná. Konkrétně je třeba definovat tabulku:

Tón	Linka v notové osnově	kmitočet [Hz]	vzdálenost od A_4 [půltónů]	rel. koef. vert. umístění v osnově	křížek
A_5	+1 (pomocná)	880,0	**12**	**7**	
$G\#_5$	(mezera)	830,6	**11**	**6**	#
G_5	(mezera)	784,0	**10**	**6**	
$F\#_5$	5	740,0	**9**	**5**	#
F_5	5	698,5	**8**	**5**	
E_5	(mezera)	659,3	**7**	**4**	
$D\#_5$	4	622,3	**6**	**3**	#
D_5	4	587,3	**5**	**3**	
$C\#_5$	(mezera)	554,4	**4**	**2**	#
C_5	(mezera)	523,3	**3**	**2**	
H_4	3	493,9	**2**	**1**	
$A\#_4$	(mezera)	466,2	**1**	**0**	#
A_4	*(mezera)*	*440,0*	***0***	***0***	
$G\#_4$	2	415,3	**-1**	**-1**	#
G_4	2	392,0	**-2**	**-1**	
$F\#_4$	(mezera)	370,0	**-3**	**-2**	#
F_4	(mezera)	349,2	**-4**	**-2**	
E_4	1	329,6	**-5**	**-3**	
$D\#_4$	(mezera)	311,1	**-6**	**-4**	#
D_4	(mezera)	293,7	**-7**	**-4**	
$C\#_4$	-1 (pomocná)	277,2	**-8**	**-5**	#
C_4	-1 (pomocná)	261,6	**-9**	**-5**	

Tab. č. 2: Přiřazení hodnoty půltónu ke koeficientu relativní výšky v notové osnově

Horní a spodní hranici vertikální souřadnice noty je třeba omezit tak, aby velmi hluboké či velmi vysoké noty nezasahovaly do jiného řádku notové osnovy nebo dokonce mimo obrazovku. Pro tento účel je určen interval zobrazitelných not tónů. Pokud je třeba napsat notu mimo tento interval, odečítá (resp. přičítá) se několikrát k hodnotě tónu číslo 12, dané počtem půltónů v jedné oktávě. Zároveň tuto konverzi do zobrazitelného intervalu graficky indikujeme znaménkem + (resp. –) vykresleným kousek nad vrchní linku v bloku noty. Pokud je nota posunuta o více než jednu oktávu, zobrazí se znaménko i s číslem. Pro posun o jednu oktávu je číslo 1 skryté. V normálním hudebním zápisu [7] se toto značí textem 8va (+), 15ma (+2), resp. 8mb (-), 15mb (-2) a čarou ohraničeným úsekem takto posunutých not. Pro účely této práce ale postačí zjednodušený zápis se znaménkem nad každou notou samostatně.

Ve chvíli, kdy je určen základní pixel pro vykreslení noty, je použit univerzální postup s relativními souřadnicemi pixelů noty samotné a jejích znamének. První se vykreslí tělo noty, pro kratší než půlovou notu plný, jinak prázdný ovál. Dále se vykreslí křížek „#“ k notám, které ho mají v názvu, tab. 2. Následuje v kódu switch struktura s parametrem délka noty. Délka noty je jednička pro osminovou až osmička pro celou notu. Nastaví se tím proměnné, představující jednotlivé části noty či znaménka [7], na true či false. V následujícím kroku se už vykreslí pouze ty prvky, které získaly hodnotu true. Konkrétně nožička ve tvaru svislé čáry u půlové až osminové noty, vlnka z horního konce čáry pouze u osminové noty, tečka u 3/8 a 3/4 noty, dvě tečky u 7/8 noty a tři tečky u 5/8. Tři tečky jsou zjednodušení oproti správnému zápisu pomocí více not za sebou. Pro noty ležící mimo základních 5 notových linek se nakonec přikreslí potřebné pomocné linky.

Implementovaný notový zápis neobsahuje žádné pomlky. Namísto pomlk v době, kdy se nehrají žádné dostatečně silné tóny, se pouze pozastaví proces vypisování not a příští rozpoznaný tón vždy začne novým blokem s vynechanou mezerou.

3.3 Ostatní zobrazované prvky aplikace – CPU meter, level meter

Jako rozšíření pro lepší práci s aplikací a její vývoj je přidán „level meter“, který ukazuje na obrazovce aktuální intenzitu signálu ze vstupu kodeku. Tento ukazatel odpovídá střední hodnotě z absolutní hodnoty všech vzorků v bufferu. Zobrazený je zvlášť červený levý i černý pravý kanál. Malá tečka pohybující se v „level meteru“ znamená maximální hodnotu ze všech vzorků signálu v bufferu. Další funkcí „level meteru“ je indikátor přebuzeného vstupu za koncem linky obou kanálů. Ten se rozsvítí, pokud se v signálu objeví vzorek s absolutní hodnotou vetší než 92% maximální hodnoty. Pro 16bit vzorkování A/D převodníku je maximum 32767 a hraniční hodnota indikátoru tedy 30000. Tento varovný indikátor zůstává trochu svítit, i pokud signál už přebuzený není. Lze tento stav vymazat hardwarovým tlačítkem na desce přípravku. Indikátor s pamětí je vytvořen, aby nedošlo k přehlédnutí této události a aby mohl uživatel zamezit dalšímu nebezpečně vysokému signálu. Tento ukazatel nemá žádnou spojitost s algoritmem rozpoznání tónů, pouze jeho aktualizace má stejný interval jako interval počítání FFT v algoritmu.

Poslední grafický prvek aplikace je „CPU meter“, který pravidelně ve stejném intervalu jako „level meter“ zobrazuje aktuální vytížení procesoru. Tento údaj je zprostředkován modulem pro profilling, údaj je počítán z poměru zbylého neužitečného procesorového času stráveného v nekonečné while smyčce oproti času stráveného v užitečné části kódu. Tato informace je opožděna o jeden cyklus algoritmu. Údaj se za běhu aplikace mění jen minimálně.

3.4 Indikátory LED a funkce tlačítek a přepínačů na desce

Několika parametrů lze nastavit i za běhu aplikace, bez nutnosti zdlouhavě měnit parametry ve zdrojovém kódu a znovu kompilovat. Pro tyto účely je využito mikrotlačítek a DIP přepínačů na desce vývojového přípravku:

	připojeno k	**funkce v aplikaci**
tlačítko „user 1“	GPIO2[4]	krátké přepnutí na alternat. algoritmus vhodný pro přepis do not[1]
tlačítko „user 2“	GPIO2[5]	resetování notového zápisu a všech indikátorů
přepínač č.5	GPIO0[1]	(nevyužito)
přepínač č.6	GPIO0[2]	(nevyužito)
přepínač č.7	GPIO0[3]	zapnutí korigování DC offsetu AIC31 kodeku při výpočtu úrovně signálu ze vstupu line-in[2]
přepínač č.8	GPIO0[4]	přepnutí na alternativní algoritmus vhodný pro přepis do not[1]

Tab. č. 3: Funkce tlačítek a přepínačů v aplikaci

Podobně je využito čtyř LED zabudovaných na desce vývojového přípravku pro nejzákladnější indikaci správného běhu aplikace, nezávislou na připojení obrazovky nebo případné chybě, která způsobí prázdný či nečitelný obraz:

[1] Rozpoznávání tónu alternativním algoritmem na obrazovce indikováno malým písmenem „a“ v pravém dolním rohu rámce ladičky.
[2] Vypisování střední hodnoty úrovně signálu v bufferu bez stejnosměrné složky indikováno textem „–DC“ nad ukazatelem úrovně na obrazovce.

	připojeno k	**funkce v aplikaci**
LED „D4“	GPIO6[13]	bliká při správném běhu aplikace, frekvence odpovídá frekvenci výpočtu algoritmu a střída vytíženosti procesoru
LED „D5“	GPIO6[12]	indikuje aktivní ladičku, když rozpoznaný tón je dostatečně silný
LED „D6“	GPIO2[12]	indikátor přebuzení vstupního signálu (aktuální)
LED „D7“	GPIO0[9]	indikátor přebuzení vstupního signálu (permanentní)

Tab. č. 4: Funkce LED v aplikaci

Na desce je i několik dalších LED, tlačítek a přepínačů, které nejsou připojeny k DSP a nejsou využitelné pro aplikaci. Jsou to LED sloužící pouze k indikaci správného napájení desky a je zde mirkotlačítko „reset“ a DIP přepínače č. 1 - 4 pro nastavení bootovacího módu [8].

4 Použité matematické nástroje a software

K realizací cílů práce byly použity následující matematické nástroje, které jsou stručně shrnuty v této kapitole.

4.1 Knihovna DSPLIB – FFT, autokorelace a maximum

Firma Texas Instruments zdarma nabízí i různé knihovny kódu často používaných funkcí optimalizovaných co nejvíce pro své procesory. Obsahují funkce pro zpracování signálů, zpracování obrazu, matematické funkce, funkce zprostředkovávající jednodušší práci s periferiemi a mnoho dalšího. V této práci je vhodné použít knihovnu DSPLIB [9]. Jsou z ní čerpány funkce pro FFT, autokorelaci a maximum. Knihovna poskytuje nejrychlejší matematické funkce pro práci se signály. Funkce jsou ručně optimalizované v jazyku symbolických adres přímo pro procesorové jádro C6748. Jiné univerzální knihovny funkcí pro běžné procesory by zdaleka nevyužily všech hardwarových nástrojů, které v sobě DSP má, a počítaly by řádově pomaleji.

Diskrétní Fourierova transformace, konkrétně její rychlá varianta FFT, poslouží k převodu signálu z časové oblasti do kmitočtové oblasti. Pro účely implementace na DSP byla vybrána funkce `DSPF_sp_fftSPxSP` [9] počítající v single-precision float hodnotách. Přesnost 32 bitů pro tento algoritmus postačuje. Tato funkce FFT je implementována algoritmem Cooley-Tukey. Ten používá předem vygenerované pole „twiddle factors“, kde hodnoty jsou stále stejné, pokud nezměníme počet vstupních vzorků FFT. Algoritmus funkce používá ještě malé pole 64 neměnných konstant „bit reversal“ definovaných v pomocném hlavičkovém souboru. Menší nevýhodou je, že vždy vyžaduje vstupní data jako komplexní čísla. U reálného signálu je třeba vstupní data doplnit o nulovou imaginární část u všech vzorků. Navíc je třeba zajistit, aby vstupní pole bylo v paměti zarovnáno na „double-word“ adresy, což znamená 8 bytů. Lze to zajistit pomocí zásahu do jinak automatického linkovacího skriptu a direktivou `#pragma DATA_SECTION(promenna,8)`. Náročnost výpočtu FFT podle manuálu odpovídá složitosti přibližně:

$$\Theta(3N \cdot \log_4 N + 2 \cdot N) \tag{9}$$

Autokorelace, neboli korelace signálu s ním samotným, poslouží k vylepšení základního algoritmu rozpoznávání tónu. Funkce hledá míru závislosti hodnot jednoho a téhož signálu při vzrůstajícím časovém posunutí signálu vůči sobě samotnému. V knihovně lze k tomu najít funkci `DSPF_sp_autocor` [9] počítající opět v single-precision float hodnotách. Funkce vyžaduje vstupní pole speciálně formátované o dvojnásobné velikosti doplněné zepředu nulami. Menší nevýhodou je požadavek na velikost vstupního pole, která musí být násobek 2, a velikost výstupního pole, která musí být násobek 4. Pro obě pole je třeba dbát na „double-word“ zarovnání v paměti. Když vstupních hodnot je stejný počet jako výstupních, lze vzorec složitosti funkce z manuálu knihovny zjednodušit přibližně na:

$$\Theta(N^2/4) \tag{10}$$

Funkce maximum vracející index pořadí nalezené hodnoty v poli je `DSPF_sp_maxidx` [9]. Menší nevýhodou je omezení velikosti vstupního pole na násobek 4. To znamená, že někdy je třeba pro algoritmus ladičky doplnit pole o několik nulových hodnot nakonec. Náročnost výpočtu podle manuálu odpovídá složitosti přibližně:

$$\Theta\left(\frac{2}{3}N\right) \tag{11}$$

4.2 StarterWare – knihovna pro práci s periferiemi vývojového přípravku

Tato knihovna [10] obsahuje spoustu funkcí k jednoduchému ovládání všech periferií desky vývojového přípravku OMAP-L138 LCDK. Obsahuje zdrojový kód v C i v assembleru poskytující abstraktní hardwarovou vrstvu k periferiím pro aplikace bez operačního systému i pro RTOS. V rámci knihovny je možné využít i spoustu ukázkových aplikací pro většinu periferií. Z jejich kódu se lze naučit efektivní práci s knihovnou nebo lze jednoduše jejich zdrojového kódu využít jako základ vlastní aplikace. Jsou zde i složitější demonstrační aplikace používající mnoho periferií zároveň.

Aplikace vytvářená pro DSP v této práci využívá právě kód ukázkové aplikace s McASP „mcaspPlayBk.c“. Použitý kód zajistí vše okolo nastavení desky a kodeku AIC31 pro příjem zvukového signálu z jack konektoru. Aplikace ladičky už jen získává potřebná data z bufferu vytvořeného touto ukázkovou aplikací.

Další využité ukázkové aplikace ze StarterWare poskytly pouze kód některých jejich funkcí. Konkrétně z ukázkové aplikace „rasterDisplay.c“ byly vzaty funkce zajišťující nastavení LCDC periferie pro grafický výstup na VGA monitor či malý displej. Pro aplikaci ladičky pak už stačilo pouze používat pole pixelů displeje o 16bit barvách a rozměrech 640x480, které je automaticky vykreslováno na displej nastaveným zobrazovacím kmitočtem. Z ukázkové aplikace „gpioCardDetect.c“ byly vzaty funkce pro inicializaci GPIO modulu a zpřístupnění LED, tlačítek a DIP přepínačů. A poslední kód funkcí k nastavení a ovládání hardwarového 64bit časovače byl vzat z ukázkové aplikace „timerCounter.c“. Časovač byl použit pro účely profilling modulu zabudovaného v kódu aplikace ladičky.

4.3 Code Composer Studio

Vývoj aplikací pro DSP řady C6000 i mnoha jiných probíhá pomocí vývojového prostředí Code Composer Studio od firmy Texas Instruments. Po bezplatné registraci je ke stažení zdarma pro nekomerční využití. Konkrétně pro tuto práci byla použita aktuální verze 6. Obsahuje vše potřebné – editor C/C++, kompilátor, linker i debugger. Debuggování ale nelze vůbec provádět bez externího emulátoru, například XDS100 připojeného přes USB k hostitelskému počítači s CCS. Vývojový přípravek LCDK totiž nemá emulátor integrovaný na desce.

Pro odladění algoritmu rozpoznání tónů je nutné často použít debugger a mít přehled o aktuálních hodnotách proměnných, obsahu RAM i dokonce stavu registrů procesoru. Dobré je využít možnosti breakpointů a krokování programu po jednotlivých řádcích kódu.

4.4 MATLAB

Pro samotný vývoj algoritmu a jeho testování bylo použito software MATLAB 2015b, pracujícím pouze s nahrávkou tónu reálných hudebních nástrojů a melodií z jejich tónů. Zdrojový kód algoritmu v jazyce MATLAB je popsán v kapitole 2.4. MATLAB je vhodný pro vykreslování grafů a simulace namísto běhu v reálných podmínkách. Také lépe než DSP poslouží k porovnání výsledků algoritmu s různými parametry a změnami pomocí stejného a neměnného nahraného zvuku. Zobrazit pěkné přehledné grafy a průběhy na obrazovku v implementaci pro DSP by bylo oproti MATLABu nesrovnatelně obtížné. Algoritmus je přepsán do jazyka C až poté, co je algoritmus vyvinut a funkčně odladěn v MATLABu. Na DSP probíhá spíše odladění pro práci v reálném čase.

Závěr

Název této práce napovídá, že účelem dvou úkolů napsaných v zadání je naučit se pracovat s platformou číslicového signálového procesoru a praktické seznámení se s mnoha aspekty této problematiky. Nejprve bylo třeba přečíst trochu teorie a prakticky se seznámit s vývojovým přípravkem. Musím se přiznat, že jsem byl nadšený ze zapůjčeného přípravku a měl jsem sklony více pracovat s přípravkem, než studovat teorii. Ukázalo se ale, že problematika DSP je daleko složitější, než jsem si dokázal představit. Začátky pro mě byly opravdu těžké i vzhledem k tomu, že jsem moc nerozuměl mikroprocesorům a DSP jsem do té doby ani neviděl. Když mě okolnosti přesvědčily, že musím více číst text, než praktikovat metodu „pokus omyl“, trochu mě vyvedla z míry příručka procesoru OMAP-L138 se svými téměř 2000 stránkami [11].

Mnoho práce mi dalo vůbec poprvé zprovoznit na přípravku hotovou ukázkovou aplikaci, která bliká s LED zabudovanou na desce. Často jsem hodiny a hodiny studoval texty a návody na internetu, proč něco nefunguje, když to fungovat má. Velký pokrok byl po dlouhé době, co se mi podařilo spustit ukázkovou aplikaci mcaspPlayBk, která přeposílá zvuk ze vstupu na výstup, a najít, kde přesně v paměti leží vzorky signálu a jak jsou zformátovány. Pak jsem teprve mohl začít s prací na algoritmu rozpoznání tónu a splnit zadání práce. Po čase stráveném nad praktickou částí této práce mohu uznat, že nyní bych dokázal další takovou aplikaci vytvořit na vlastním základu, který nepoužívá ukázkovou aplikaci mcaspPlayBk. Pro porovnání se stavem na začátku, bylo pro mě velmi složité vůbec najít jak na tuto ukázkovou aplikaci navázat aplikaci svojí a spoustu částí jejího kódu jsem nechápal.

Kód implementace algoritmu v MATLABu je krátký oproti kódu implementace aplikace pro DSP. Kompletní kód z MATLABu lze najít v příloze. Ukázka klíčové části algoritmu v C je uvedena v textu práce. Experimentoval jsem i s využitím Linuxu na ARM jádře pro obsloužení části aplikace. Po dlouhé době jsem ale usoudil, že synchronizace a komunikace mezi dvěma odlišnými procesorovými jádry je pro účely aplikace příliš složitá a musel bych se o tom ještě mnoho naučit. Moje volba ve výsledku také nebyla nejefektivnější. Bylo třeba naprogramovat vlastní rozsáhlý kód pro vykreslování veškeré grafiky a textů aplikace pixel po pixelu. Je tedy tomu také věnována celá kapitola.

Literatura [12] uvedená v zadání této práce mi byla užitečná spíše k pochopení obecných principů fungování DSP řady C6000. Autor knihy používá vývojový přípravek TMS320C6711 DSK, který není zaměnitelný za přípravek používaný v této práci. Kniha vede čtenáře velmi prakticky a používá mnoho konkrétních příkladů. Bohužel mi kniha nepomáhala řešit konkrétní problémy během plnění úkolu ze zadání práce. Hlavní důvod je v použití jiné podpůrné knihovny (tzv. board-support library), než je knihovna StarterWare dodávaná pro OMAP-L138 LCDK. S tím je spojen úplně jiný princip práce s periferiemi DSP v ukázkových úlohách knihy. Druhá kniha z literatury v zadání je spíše jen přepracované vydání té první knihy [12] od stejného autora. Liší se pouze v použití o trochu novějšího ale velmi podobného přípravku v ukázkových úlohách a jinými demonstračními projekty v závěru knihy.

Ve výsledku musím uznat, že přínos bakalářské práce pro mě je opravdu velký, protože jsem se během toho musel vše popsané výše naučit. Naučil jsem se mnohem lépe programovat v C a jak funguje preprocesor, kompilátor i linker a debugger. Získal jsem dovednosti v používání knihoven kódů, vytváření rozsáhlých struktur zdrojových kódů a propojení mezi nimi. Mnoho jsem se dozvěděl o procesorech obecně, z čeho se skládá a jaký význam mají jednotlivé části, případně jak je použít. Prakticky teď umím pracovat s mnoha periferiemi DSP, konkrétně ovládat pinový multiplex, nastavit registry procesoru, použít GPIO, sériový port, modul pro rastrový displej a hardwarový čítač. Naučil

jsem se hodně o pamětích, o externích i vnitřních a cache, jak funguje adresový prostor a jak s ním pracovat ve zdrojovém kódu. Naučil jsem se, jak procesor zpracovává instrukce, jak fungují a k čemu všemu se dají využít přerušení. Osvojil jsem si prakticky zpracování signálů pomocí Fourierovy transformace a práce s vzorky spektra. Užitečné jsou pro mě zkušenosti s odladěním aplikace na DSP, aby pracovala efektivně a rychleji. Prakticky jsem si vyzkoušel práci s mnoha nástroji během debuggování aplikace a práci s emulátorem. Naučil jsem se spoustu nových věcí v MATLABu. Mnohem více už rozumím zvuku hudebních nástrojů a jejich analýze. Shrnu-li to tedy, práce na samotném vývoji algoritmu rozpoznání tónu byla dokonce jen ta menší část veškeré práce na bakalářské práci.

Cíle práce a úkoly zadání byly úspěšně splněny. Algoritmus rozpoznání tónů není bezchybný, ale pro samostatně zahrané tóny funguje velmi dobře. Pro ladičku akustických tónu plně stačí původní verze s autokorelací výkonového spektra. Ve chvíli, kdy je hráno více tónů přes sebe, má lepší výsledky alternativní algoritmus s hledáním lokálních maxim v amplitudovém spektru a práce s vyššími harmonickými kmitočty. Rozpoznání více tónů plně demonstruje implementace v MATLABu, která umí vykreslit kmitočty tónu do jednoho grafu a rozpoznané tóny dokonce zpětně zrekonstruovat a přehrát sinusovými signály.

Tento polyfonní algoritmus by určitě bylo možné vyvíjet ještě dále, kvůli rozsahu této bakalářské práce je spíše jen jako doplněk. Zahrané tóny umí ve většině případů rozpoznat správně a přesně. Jeho nedostatek ale spočívá v nemalém množství tónů rozpoznaných navíc. Při poslechu zrekonstruovaných melodií z rozpoznaných tónů se ale ukazuje, že to jsou často tóny na kmitočtech hudebně ladících ke správně rozpoznaným tónům.

Pro notový zápis bylo v zadání práce záměrně použito slova „zjednodušený“. Naprogramování zápisu více not nad sebou nebo mnoha detailů hudebního notového zápisu by bylo velmi komplikované. Cílem práce není mít do detailu všechna pravidla implementovaná, ale umět tóny rozpoznat a umět je graficky znázornit na obrazový výstup. To aplikace určitě splňuje.

Použité zdroje

[1] ZYTRAX, Inc. Frequency Ranges: Audio Freqencies. Zytrax: Tech Stuff [online]. 2016 [cit. 2016-04-22]. Dostupné z: http://www.zytrax.com/tech/audio/audio.html

[2] STIEFEL. Stack Overflow: *Robust algorithm for chromatic tuner?* [online]. 20. května 2010. [vid. 2016-04-22]. Dostupné z: http://stackoverflow.com/questions/2873589/robust-algorithm-for-chromatic-instrument-tuner

[3] DAVÍDEK, Vratislav a Pavel SOVKA. *Číslicové zpracování signálů a implementace*. Vyd. 2. přeprac. Praha: Vydavatelství ČVUT, 2002. ISBN 80-01-02483-0.

[4] HRDINA, Zdeněk a František VEJRAŽKA. *Signály a soustavy*. Vyd. 1. Praha: Vydavatelství ČVUT, 1998. ISBN 80-01-01726-5.

[5] TEXAS INSTRUMENTS. *TLV320AIC3106 Low-Power Stereo Audio CODEC for Portable Audio/Telephony* [online]. Rev. F. 24 Dec 2014. Dostupné z: http://www.ti.com/lit/ds/symlink/tlv320aic3106.pdf

[6] Piano key frequencies. In: *Wikipedia: the free encyclopedia* [online]. poslední aktualizace 27. 2. 2016 [cit. 2016-04-16]. Dostupné z: https://en.wikipedia.org/wiki/Piano_key_frequencies

[7] List of musical symbols. In: *Wikipedia: the free encyclopedia* [online]. poslední aktualizace 5. 5. 2016 [cit. 2016-05-11]. Dostupné z: https://en.wikipedia.org/wiki/List_of_musical_symbols

[8] TEXAS INSTRUMENTS. L138/C6748 Development Kit (LCDK). In: *Texas Instruments Wiki* [online]. poslední aktualizace 17. 2. 2016 [cit. 2016-05-02]. Dostupné z: http://processors.wiki.ti.com/index.php/L138/C6748_Development_Kit_(LCDK)

[9] TEXAS INSTRUMENTS. *TMS320C67x DSP Library Programmer's Reference Guide* [online]. Rev. C. 31 Jan 2010. Dostupné z: http://www.ti.com/lit/ug/spru657c/spru657c.pdf

[10] TEXAS INSTRUMENTS. StarterWare 01.10.01.01 User Guide. In: *Texas Instruments Wiki* [online]. poslední aktualizace 2. 5. 2013 [cit. 2016-05-02]. Dostupné z: http://processors.wiki.ti.com/index.php/StarterWare_01.10.01.01_User_Guide

[11] TEXAS INSTRUMENTS. *OMAP-L138 DSP+ARM Processor Technical Reference Manual* [online]. Rev. A. 12 Dec 2011. Dostupné z: http://www.ti.com/lit/ug/spruh77a/spruh77a.pdf

[12] CHASSAING, Rulph. *DSP applications using C and the TMS320C6x DSK*. New York: John Wiley & Sons, 2002. ISBN 0-471-22112-0.

Použité obrázky

[13] SVOBODA, J. a J. BRDA. *Elektro-akustika do kapsy*. Vyd. 2. přeprac. Praha: SNTL, 1981, tabulka č. 3.

Přílohy

Příloha č.1: soubor „c6748_tuner.m“ (skript MATLABu)

```
clear all,close all,clc

%==== MAIN SETTINGS: ====
window = 4800; %[samples]
shift_ratio = 2/3; %[-] of window
fmax = 6800; %[Hz]  ®®\ HC filter
fmin = 40; %[Hz]  /®® LC filter
min_signal = 1e+4;%9e+5; % minimal amplitude after autocorrelation
input_file_suffix = 'guitar-eadghe3b';
fs_decimate_ratio = 3; % 48kHz->16kHz
% alternative algorithm settings:
peak_mode = 4; % local max in amplitude spectrum width to both sides
db_min = 60; %[dB] minimal tone amplitude (else->NaN)
polyphone_max = 4; % maximal number of tones in the same time
%=========================

% setting for some detailed demonstration plots(fft,autocorr,peak_detect):
demo_plot = true; % enable/disable
plot_every = 8;
plot_rows = 5;
zoom_ratio = 2^4; %radix-2!

% background variables:
Am = 2^15-1; % signed short 16bit codec imitation (for c6748)
shift = round(window*shift_ratio);

% read test_melody signal:
[x,fs] = audioread(strcat('../../Downloads/test_melody-',...
        input_file_suffix,'.wav'));
x = x(:,1)'*Am;
x = decimate(x,fs_decimate_ratio); % 48kHz->16kHz
fs = fs/fs_decimate_ratio;

% search for nearest 4-root number for fft input:
N = 4^ceil(log(window)/log(4));

% some prepare computation:
resolution = fs/N; %[Hz]
axisratio = resolution/1000; %[kHz/sample]
imax = round(fmax/resolution); % filters
imin = round(fmin/resolution);
cnt = floor((length(x)-window)/shift); % number of fft computed
xmax = N*axisratio/zoom_ratio; % plot horizontal zoom [kHz]
shift_time = shift/fs;
fig1=figure(1); fig1.Units='normalized';
fig1.OuterPosition=[0 0 1 1]; % full screen size demo plot
nf=0:N/2-1; nf=nf*axisratio; % x axis [kHz]
fprintf('%dxFFT->',cnt); %(console log) number of FFTs

% MAIN LOOP - compute FFT for all windows: ============
amp_tab=zeros(1,cnt); f_tab=amp_tab; f_tab_a=[]; amp_tab_a=[]; % containers initialization
for (k = 0:cnt-1)
    start = 1+k*shift;
    % FFT of signal window (with zero padding) -> amplitude spectrum:
    y = fft(x(start:start+window-1).*rectwin(window)',N); %  <------
    s1 = abs(y(1:N/2)); % convert half of complex spectrum to abs.values
    s1 = s1*(2/window); % normalize amplitude (window instead of N, cause FFT pading)
    % band-pass filter:
    s1(1:imin) = 0; % /®® LC filter
    s1(imax+1:length(s1)) = 0; % ®®\ HC filter
```

```
    plotnowif = false;
                %(demo draw - amplitude spectrum)
                if (demo_plot && mod(k,plot_every)==0 && k/plot_every<plot_rows)
                    subplot(plot_rows,3,floor(k/plot_every)*3+1), plot(nf,s1),...
                            xlabel('f[kHz]')
                    axis([0 xmax 0 max(s1)]), title('FFT amplitude spectrum')
                    plotnowif=true; subplot(plot_rows,3,floor(k/plot_every)*3+3)
                end
  % ALTERNATIVE ALGORITHM for polyphone input signals: %%
    [f_tab_a0,amp_tab_a0,all_tones_cnt] = c6748_alt_algorithm(s1,polyphone_max,db_min,...
            peak_mode,resolution,plotnowif,xmax); % +(demo draw - peak search (inside))
    f_tab_a = [f_tab_a f_tab_a0']; % add results in matrix columns
    amp_tab_a = [amp_tab_a amp_tab_a0'];
  % end of ALTERNATIVE ALGORITHM %%
    % AUTOCORRELATION (of power spectrum):
    s1 = s1.^2; % power spectrum
    c = xcorr(s1); % <------
    c = c(round(length(c)/2):length(c)); % erase left mirror half of autocorr
                %(demo draw - autocorrelation)
                if (demo_plot && mod(k,plot_every)==0 && k/plot_every<plot_rows)
                    subplot(plot_rows,3,floor(k/plot_every)*3+2), plot(nf,c,':')
                    axis([0 xmax 0 max(c)]), title('autocorrelation')
                    xlabel(sprintf('f[kHz]  (start at %d:%03d)',round(floor(...
                            shift_time*k)),round(floor(mod(shift_time*k*1000,1000)))))
                end
    % erase first unwanted peak (starting at zero freq):
    p = 1;
    while (c(p)>c(p+1) || c(p)>c(p+2) || c(p)>c(p+3))
        c(p) = 0;
        p = p+1;
    end
                %(demo draw - autocorrelation without peak in zero)
                if (mod(k,plot_every)==0 && k/plot_every<plot_rows)
                    hold on, plot(nf,c), hold off
                end
    % find next first peak -> fundamental frequency:
    [m,i] = max(c);
    m = sqrt(m); % compensation of autocorrelation (not exact! dB->dB* )
    amp_tab(k+1) = m; % write amplitude of found fund.frequency
    % erase too small signal -> NaN
    if (m<min_signal)% || i<=imin) % (with addition of LC filter)
        f_tab(k+1) = NaN; %amp_tab(k+1)=0;
    else
        f_tab(k+1) = i*resolution; % convert peak freqency and write
    end
    fprintf('%d',min([all_tones_cnt 9])); %(console log) number of all found tones (max.9)
end

%==== OUTPUT of results (graphic): ====
colors = ['r','g','b','c','m','y','k']; % for synchronization of plot&stem colors
% draw result plot:
amp_tab_db = 20*log10(amp_tab); % create dB scale of amplitude
n_tab = round(12*log2(f_tab/440)); % convert from f to discrete halftones
nt = 0:shift_time:shift_time*(length(f_tab)-1); % time x-axis
fig2=figure(2); fig2.Units='normalized'; fig2.OuterPosition=[.05 .5 .9 .5];
[ha,hl1,hl2] = plotyy(nt,amp_tab_db,nt,f_tab,'stem','plot');% left:A(t) right:f(t)
ylabel(ha(2),'f[kHz]'), ylabel(ha(1),'A[dB]'), xlabel('t[s]')
set(ha(2),'YGrid','on','YMinorGrid','on','YMinorTick','on');
set(ha(1),'YColor',[.8 .8 .8],'YMinorTick','on'); ha(1).YLim(1)=20*log10(min_signal);
ha(2).YLim(1)=0; ha(1).XLim(2)=nt(length(nt)); ha(2).XLim(2)=nt(length(nt));
ha(1).YTickMode='auto'; ha(2).YTickMode='auto';
hl1.Marker='+'; hl1.LineStyle='--'; hl1.Color=colors(1); hl1.MarkerSize=10; % A
hl2.Marker='.'; hl2.LineStyle=':'; hl2.Color=colors(1); hl2.MarkerSize=26; % f
title(sprintf('window=%d(%0.1fms) N=%d res=%0.2fHz t=+%0.1fms',window,window/fs*1000,N,...
        resolution,shift_time*1000))
```

```
% draw results plot of alternative algorithm:
amp_tab_a_db = 20*log10(amp_tab_a);
%TODO: subtract db_min
n_tab_a = round(12*log2(f_tab_a/440));
xxx=[]; for (k=1:polyphone_max) xxx=[xxx,nt']; end % create special x-axis for all tones
fig4=figure(4); fig4.Units='normalized'; fig4.OuterPosition=[.05 .0 .9 .5];
[ha,hl1,hl2] = plotyy(xxx,amp_tab_a_db',xxx,f_tab_a','stem','plot');% left:A(t) right:f(t)
ylabel(ha(2),'f[kHz]'), ylabel(ha(1),'A[dB]'), xlabel('t[s]')
set(ha(2),'YGrid','on','YMinorGrid','on','YMinorTick','on');
set(ha(1),'YColor',[.8 .8 .8],'YMinorTick','on'); ha(1).YLim(1)=db_min; ha(2).YLim(1)=0;
ha(1).XLim(2)=nt(length(nt)); ha(2).XLim(2)=nt(length(nt));
ha(1).YTickMode='auto'; ha(2).YTickMode='auto';
for (k = 1:polyphone_max)
    clr = colors(mod(k,length(colors))); % circulate in color array
    hl1(k).Marker='+'; hl1(k).LineStyle='--'; hl1(k).Color=clr; hl1(k).MarkerSize=10; % A
    hl2(k).Marker='.'; hl2(k).LineStyle=':'; hl2(k).Color=clr; hl2(k).MarkerSize=26; % f
end
title(sprintf('window=%d(%0.1fms) N=%d res=%0.2fHz t=+%0.1fms',window,window/fs*1000,N,...
        resolution,shift_time*1000))

%==== EXPORT of results (audio, picture): ====
play_sig_old = play_melody(n_tab,amp_tab,shift_time,true,fs);
play_sig = zeros(1,length(play_melody(n_tab_a(1,:),amp_tab_a(1,:),shift_time,true,fs)));
for (k = 1:polyphone_max)
    play_sig = play_sig+play_melody(n_tab_a(k,:),amp_tab_a(k,:),shift_time,true,fs);
end
soundsc([play_sig zeros(1,3*fs) play_sig_old],fs);
audiowrite(strcat('c6748_test/',input_file_suffix,exp_label,sprintf(...
    '_pm%u-db%u-p%u.wav',peak_mode,db_min,polyphone_max)),play_sig/max(abs(play_sig)),fs);
audiowrite(strcat('c6748_test/',input_file_suffix,exp_label,sprintf('_old_ms%u.wav',...
    min_signal)),play_sig_old/max(abs(play_sig_old)),fs);
print(fig4,strcat('c6748_test/',input_file_suffix,exp_label,sprintf('_pm%u-db%u-p%u',...
    peak_mode,db_min,polyphone_max)),'-dpng');
print(fig2,strcat('c6748_test/',input_file_suffix,exp_label,sprintf('_old_ms%u',...
    min_signal)),'-dpng');
```

Příloha č.2: soubor „c6748_alt_algorithm.m“ (funkce pro MATLAB)

```
% Function - alternative algorithm for analysis of amplitude spectrum
%   polyphone: number of tones to return (if not enough tones->NaN)
%   db_min: minimal tone amplitude value (if not enough->NaN)
%   mode_peak_width: number of samples to both sides from founded peak to be
%                    accepted as local max value
%   f_resolution: [Hz/sample]
%   plotnow (optional): if true, output a graph of peak research
%   plotxmax (optional): [kHz] zoom for graph - max freq to show
% Returns table of 'polyphone' number of found tones:
%   table_f: frequencies [discrete freqency(*f_resolution=Hz)]
%   table_a: amplitudes (sum of harmonic tones amplitudes)
%   all_cnt (optional): complete number of all found tones
function [table_f,table_a,all_cnt] = c6748_alt_algorithm(ampl_spectrum,polyphone,...
        db_min,mode_peak_width,f_resolution,plotnow,plotxmax)
numargs = 7;
if (nargin<numargs-1) plotnow=false; end % optional arguments default
if (nargin<numargs) plotxmax=f_resolution*(length(ampl_spectrum)-1); end

ampl_spectrum(1) = 0; % erase DC value (might already be done)
peak_limit = max(ampl_spectrum)/10; % use only peaks with 10% of max value in spectrum
amp_min = 10^(db_min/20); % translate from dB->amplitude
ftable0=[]; atable0=[];
% peak detection (local maxmimum)
for (k = 1+mode_peak_width:length(ampl_spectrum)-mode_peak_width)
    if (ampl_spectrum(k) > peak_limit)
        [maxval,imax] = max(ampl_spectrum(k-mode_peak_width:k+mode_peak_width));
        if (imax == 1+mode_peak_width)
```

```
            ftable0 = [ftable0 k-1]; % frequency [index] (*f_resolution=[Hz])
            atable0 = [atable0 maxval]; % amplitude
end,end,end
atable0_plot = atable0; % save for plot
% process of harmonic extraction (erase harm.freq +add its amplitude to fundamental freq):
table_f0=[]; table_a0=[]; table_f=[]; table_a=[]; table_f0_r=[]; table_a0_r=[];
for (k = 1:length(ftable0)-1) % tone related intensity coeffs:
    if (atable0(k) ~= 0)
        tf = ftable0(k); % 1.harmonic +-0.5
        for (k2 = k+1:length(ftable0)) % sum the amplitude from harmonic frequencies
            tf2 = ftable0(k2); % 2.harmonic +-3  3.harmonic +-3.5
            if (tf2>tf+tf-4&&tf2<tf+tf+4 || tf2>tf*3-4&&tf2<tf*3+4 ||...
                    tf2>tf*4-5&&tf2<tf*4+5 || tf2>tf*5-5&&tf2<tf*5+5 ||...% 4.+-4  5.+-4.5
                    tf2>tf*6-6&&tf2<tf*6+6 || tf2>tf*7-6&&tf2<tf*7+6) % 6.+-5  7.+-5.5
                atable0(k) = atable0(k)+atable0(k2); %TODO: only half ratio of value?
                atable0(k2) = 0; % erase after move (6x the same later..)
        end,end
        if (atable0(k) > amp_min) % erase too small ones
            table_f0 = [table_f0 ftable0(k)]; % colect passed freqs with their amplitude
            table_a0 = [table_a0 atable0(k)];
        else
            table_f0_r = [table_f0_r ftable0(k)]; % colect erased ones for plot
            table_a0_r = [table_a0_r atable0(k)];
end,end,end
endk = length(ftable0);
if (isempty(table_f0) && endk~=0 && atable0(endk)>amp_min) % specially for the last
    table_f0 = [table_f0 ftable0(endk)];
    table_a0 = [table_a0 atable0(endk)];
end
all_cnt = length(table_f0);
% choose only 'polyphone' number of tones with biggest amplitude (if not enough add NaNs)
if (polyphone < all_cnt)
    table_a0_temp = table_a0;
    for (k = 0:polyphone) % find ('polyphone'+1)th amplitude as border 'tempa'
        [tempa,tempi] = max(table_a0_temp);
        table_a0_temp(tempi) = 0;
    end
    for (k = 1:length(table_a0)) % copy only 'polyphone' tones sorted by frequency
        if (tempa<table_a0(k) && length(table_f)<polyphone)
            table_f = [table_f table_f0(k)*f_resolution];
            table_a = [table_a table_a0(k)];
    end,end
else
    table_f = [table_f0.*f_resolution ones(1,polyphone-all_cnt)*NaN];% convert f[-]->f[Hz]
    table_a = [table_a0 ones(1,polyphone-all_cnt)*NaN]; % and append missing tones as NaN
end

% optional ploting (only compatible with special code outside this function)
if (plotnow)
    ffx = 0:f_resolution:f_resolution*(length(ampl_spectrum)-1);
    plot(ffx*0.001,ampl_spectrum), xlim([0 plotxmax]), xlabel('f [kHz]') % input spectrum
    hold on, plot(ffx*0.001,ones(1,length(ampl_spectrum))*peak_limit,':') % 10% limit line
    plot(ffx*0.001,ones(1,length(ampl_spectrum))*amp_min,':r') % minimal amplitude line
    stem(ftable0*0.001*f_resolution,atable0_plot,'LineStyle','none','MarkerEdgeColor','k')
    stem(table_f0_r*0.001*f_resolution,table_a0_r,'LineStyle','none','MarkerFaceColor',...
            'g','MarkerEdgeColor','none','MarkerSize',4)
    stem(table_f0*0.001*f_resolution,table_a0,'LineStyle','none','MarkerFaceColor','r',...
            'MarkerEdgeColor','none','MarkerSize',4),hold off
end
end %of function c6748_alt_algorithm()
```

Příloha č.3: soubor „play_melody.m“ (funkce pro MATLAB)

```
% Play a melody of 'vecf' array samples [halftones_from_A4] of length'time'
```

```
% and amplitude 'veca' array (relative amplitude) on same indexes as'vecf',
% with sample rate 'fs'(optional) and return samples 'out' (default 48kHz).
% If 'mute'(optional) is true, then only output samples 'out' (without
% play), default is false
% (NaN values means silence)
% 'time' and 'veca' is also optional (defaults: 0.4s, all ones)
function out = play_melody(vecf,veca,time,mute,fs)
numargs = 5;
if (nargin<numargs-3) veca=ones(1,length(vecf)); end
if (nargin<numargs-2) time=0.4; end
if (nargin<numargs-1) mute=false; end
if (nargin<numargs) fs=48000; end

veca = veca/max(abs(veca)); % normalization and error correction
nn = round(fs*time);
t = (0:nn-1)/fs;
as_time=0.005; as_n=round(as_time*fs); as_vecr=linspace(1,0,as_n); % anti-shock
vecf = 440*2.^(vecf/12);
out=[]; last_phi=0; last_pause=true;
for (k9 = 1:length(vecf))
    if (isnan(vecf(k9))) % NaN = pause (not tone)
        if (~last_pause) k0=1; % anti-shock
            for (k1 = length(out)-as_n+1:length(out))
                out(k1)=out(k1)*as_vecr(k0); k0=k0+1;
        end,end
        out = [out zeros(1,nn)];
        last_phi=0; last_pause=true;
    else
        temp_out = veca(k9)*sin(2*pi*vecf(k9)*t+last_phi);
        if (k9<length(vecf) && ~isnan(vecf(k9+1))) % anti-shock
            next_ampl_ratio = veca(k9+1)/veca(k9);
            temp_out = temp_out.*[ones(1,length(temp_out)-as_n)...
                    linspace(1,next_ampl_ratio,as_n)];
        end
        out = [out temp_out];
        last_phi = (2*pi*vecf(k9)*t(length(t)) + last_phi);
        if (last_phi/(2*pi)>=1) last_phi=last_phi-2*pi; end
        last_pause = false;
end,end
if (~mute) soundsc(out,fs); end
```

Příloha č.4: soubor „profilling_include.c“ (knihovna funkcí v C pro DSP)

```
/* functions for C6748 profiling purposes */
#define TIMER_GET() (TimerCounterGet(SOC_TMR_2_REGS,TMR_TIMER12)+ /
        ((unsigned long long)TimerCounterGet(SOC_TMR_2_REGS,TMR_TIMER34)<<32ull))
#define TIMER_RESET() (HWREG(SOC_TMR_2_REGS+0x24)&=~TMR_TGCR_TIM12RS);TimerCounterSet /
        (SOC_TMR_2_REGS,TMR_TIMER_BOTH,0);(HWREG(SOC_TMR_2_REGS+0x24)|=TMR_TGCR_TIM12RS)
#define PROF_section() (timestamps[tsi++]=TIMER_GET())
#define PROF_sectionidle() (timestamps[101]=TIMER_GET())

unsigned int cyclecount = 0;
float percentidlecputime;
float percentidlecputimeactual;
unsigned int minidletime = 0xFFFFFFFFu;
float minidlepercentcputime;
unsigned char tsi = 0; // 8bit integer -> array index never be more than 256
// [101]=nonidletime [102]=alltime  [100]=profilingapp_time
unsigned long long timestamps[103]; //103x64bit=824B  [.far]
unsigned long long durationsum[103]; //824B
float averageduration[103]; //412B   [.far]
float percentcputime[103]; //412B
// sum=2490B >2kiB

void PROF_restart(void){
```

```
    timestamps[102]=TIMER_GET();
    TIMER_RESET();
    if (cyclecount){ // ->cyclecount never zero (if 0 it is special PROF_start())
        int i;
        timestamps[tsi++]=timestamps[101]; // for last section (before sectionidle)
        for (i=102;i>99;--i){
            durationsum[i]+=timestamps[i];
            averageduration[i]=(float)durationsum[i]/cyclecount; //[102] need to be first!
            percentcputime[i]=averageduration[i]/averageduration[102]*100.0;
        }
        durationsum[0]+=timestamps[0]-timestamps[100]; // first section
        averageduration[0]=(float)durationsum[0]/cyclecount;
        percentcputime[0]=averageduration[0]/averageduration[102]*100.0;
        for (i=1;i<tsi;++i){
            durationsum[i]+=timestamps[i]-timestamps[i-1];
            averageduration[i]=(float)durationsum[i]/cyclecount;
            percentcputime[i]=averageduration[i]/averageduration[102]*100.0;
        }
        percentidlecputime=100.0-percentcputime[101];
        percentidlecputimeactual=(float)(timestamps[102]-timestamps[101])/
                timestamps[102]*100.0f;
        if(minidletime>timestamps[102]-timestamps[101]) {
            minidletime=timestamps[102]-timestamps[101];
            minidlepercentcputime=percentidlecputimeactual;
        }
    }
    tsi=0;
    ++cyclecount;
    timestamps[100]=TIMER_GET();
}

void Breakpoint(int cycles){
    if (cyclecount>=cycles){
        int i=0;
        float temp;
        printf("\n---- window=%d, shift=%d(=%dms), Nfft=%d, NHCF=%d(fmin=%.1fkHz) ----\n"
                "Profiling statistics (%d cyklu): %.3f%% idle\n"
                "kopirovani bufferu, priprava pro FFT: %.3f%%\n"
                "CPU meter + level meter: %.3f%%\nDSPF_sp_fftSPxSP(): %.3f%%\n"
                "komplexni spektrum -> vykonove spektrum: %.3f%%\n"
                "filtry, normalizace urovne: %.3f%%\nalternatvni algoritmus: %.3f%%\n"
                "DSPF_sp_autocor(): %.3f%%\nmazani vrcholu v nule: %.3f%%\n"
                "DSPF_sp_maxidx(): %.3f%%\n"
                "vykreslovani obrazovky (ladicka + noty): %.3f%%\n",
                WINDOW,WIN_SHIFT,1000*WIN_SHIFT/SAMPLING_RATE,NFFT,NHCF,
                HIGHCUT_FILTER_FREQ/1000.0,cycles,percentidlecputime,percentcputime[i++],
                percentcputime[i++],percentcputime[i++],percentcputime[i++],
                percentcputime[i++],percentcputime[i++],percentcputime[i++],
                percentcputime[i++],percentcputime[i++],percentcputime[i++]);
        for (temp=percentcputime[i];temp>0.0&&i<100;temp=percentcputime[++i])
            printf("(no name): %.3f%%\n",temp);
        printf("==== cycle_total:%.0fc(456MHz)->%.0fms ====\n",averageduration[102]*2,
                averageduration[102]/456000*2);
/*<-BREAKPOINT HERE*/cyclecount=0;
        minidletime = 0xFFFFFFFFu;
        tsi=0;
        for (i=0;i<103;++i) durationsum[i]=0; // erase durationsum[]
    }
}
```

Příloha č.5: Grafy pro demonstraci polyfonního algoritmu při hraní akordu