TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Aplikace ovládání a vizualizace mikrofonového pole

Diplomová práce

Studijní program: N2612 – Elektrotechnika a informatika
Studijní obor: 1802T007 – Informační technologie

Autor práce: **Veronika Fulínová**
Vedoucí práce: Ing. Martin Rozkovec, PhD.

Liberec 2016

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

Microphone array control and visualization application

Diploma thesis

Study programme:	N2612 – Electrical Engineering and Informatics
Study branch:	1802T007 – Information technology
Author:	**Veronika Fulínová**
Supervisor:	Ing. Martin Rozkovec, PhD.

Liberec 2016

Technická univerzita v Liberci
Fakulta mechatroniky, informatiky a mezioborových studií
Akademický rok: **2015/2016**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Veronika Fulínová**
Osobní číslo:	**M13000182**
Studijní program:	**N2612 Elektrotechnika a informatika**
Studijní obor:	**Informační technologie**
Název tématu:	**Aplikace ovládání a vizualizace mikrofonového pole**
Zadávající katedra:	**Ústav informačních technologií a elektroniky**

Zásady pro vypracování:

1. Seznamte se s technologií MEMS mikrofonů, principem funkce a způsoby zpracovní výstupních signálů. Seznamte se s platformou Xilinx APSoC.
2. Seznamte se s technologiemi WPF, .NET a C#. Důraz klaďte na realtime zobrazování, síťovou komunikaci a vícevláknové/asynchronní zpracování úloh.
3. Navrhněte aplikaci po stávající FW APSoC, která umožní ovládat pole mikrofonů a v reálném čase z něj snímat, vizualizovat a ukládat data.
4. Aplikaci doplňte o moduly zpracování získaných dat, například FFT, IIR, FIR filtry.

Rozsah grafických prací:	**Dle potřeby dokumentace**
Rozsah pracovní zprávy:	**cca 40-50 stran**
Forma zpracování diplomové práce:	**tištěná/elektronická**

Seznam odborné literatury:

[1] **Charles Petzold, Mistrovství ve WPF, Computer Press, EAN: 978-8025121412**

[2] **McClellan J.H., Schaefer R., Yoder M.A.: DSP First - A Multimedia Approach. Prentice Hall, 1998, ISBN: 978-0132431712**

Vedoucí diplomové práce:	**Ing. Martin Rozkovec, Ph.D.** Ústav informačních technologií a elektroniky
Datum zadání diplomové práce:	**14. září 2015**
Termín odevzdání diplomové práce:	**16. května 2016**

prof. Ing. Václav Kopecký, CSc.
děkan

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
vedoucí ústavu

V Liberci dne 14. září 2015

Prohlášení

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

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval(a) samostatně s použitím uvedené literatury a na zá- kladě konzultací s vedoucím diplomové práce a konzultantem.

Současně čestně prohlašuji, že tištěná verze práce se shoduje s elektronickou verzí, vloženou do IS STAG.

Datum: 31.08.2016

Podpis

Poděkování

Na tomto místě bych ráda poděkovala především vedoucímu diplomové práce Ing. Martinu Rozkovcovi, Ph.D. za cenné rady, pomoc a také za podporu při tvorbě této diplomové práce.

Velké poděkování bych také ráda vyjádřila mé rodině a blízkým, kteří mě během vypracovávání práce ve všem podporovali.

Abstrakt

Diplomová práce se zabývá studiem audio signálu z třech různých kategorií vstupů, tj. audio soubor, PC HW/mikrofony pomocí rozhraní WASAPI a MicroZed™ deska osazená MEMS PDM mikrofony. Cílem práce je vyvinout desktopovou aplikaci v jazyce C# za využití frameworku WPF, která by na základě uživatelem volených parametrů umožňovala zobrazení průběhu signálu z výše uvedených třech typů vstupu.

Klíčová slova: audio signál, C#, WPF, MVVM, MicroZed™, AP SoC, MEMS, mikrofony

Abstract

Diploma thesis deals with the study of the three different categories of inputs, i.e. audio file, PC HW/microphones using WASAPI interface and MicroZed™ board equipped with MEMS PDM microphones. The aim is to develop an application in the language C# using the WPF framefork, which, on the basis of user-selectable parameters, would allowed to display an audio waveform of the three inputs mentioned above.

Key words: audio signal, C#, WPF, MVVM, MicroZed™, AP SoC, MEMS, microphones

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

Seznam zkratek

AP SoC	All-Programmable System on Chip
API	Application Programming Interface
ARM	Acorn RISC Machine
ARP	Address Resolution Protocol
ARPANET	The Advanced Research Projects Agency Network
ASCII	American Standard Code for Information Interchange
BiQuad	BiQuadratic
CIC	Cascaded Integrator-Comb
COM port	Communication port
DFT	Discrete Fourier Transform
DLL	Dynamic-Link Library
DNS	The Domain Name System
DPI	Dots Per Inch
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
FPGA	Field-Programmable Gate Array
FTP	The File Transfer Protocol
ICMP	The Internet Control Message Protocol
IEEE	The Institute of Electrical and Electronics Engineers
IIR	Infinite Impulse Response
ISO	Open Systems Interconnection
LTI	Linear Time-Invariant
MEMS	Micro Electro Mechanical Systems
MVC	Model-View-Controller
MVVM	Model-View-ViewModel

OSI	International Organisation for Standards
PCM	Pulse-Code Modulation
PDM	Pulse-Density Modulation
RARP	Reverse ARP
SBC	Single Board Computer
SiP	System in Package
SMTP	Simple Mail Transfer Protocol
SNR	Signal-to-Noise Ratio
SOM	System On Module
TCP/IP	Transmission Control Protocol/Internet Protocol
UART	Universal Asynchronous Receiver/Transmitter
UDP	The User Datagram Protocol
WASAPI	The Windows Audio Session API
WPF	Windows Presentation Foundation
XAML	eXtensible Application Markup Language
XML	eXtensible Markup Language

Úvod

Cílem diplomové práce je vytvořit desktopovou aplikaci v .NET frameworku WPF v jazyce C#, jež by umožňovala zpracování a vizualizaci zvukového signálu na základě uživatelovy volby zdroje toku dat a nastavení vstupních parametrů.

Hlavním úkolem aplikace je možnost zobrazování audio streamů z vývojové desky osazené polem MEMS PDM mikrofonů čítající 64 kusů. Na základě tohoto požadavku je nutné detailněji se seznámit jak se základním principem MEMS PDM mikrofonů, tak obecnými vlastnostmi MEMS čidel.

Diplomová práce je pro přehlednost dělena do několika kapitol, jež se dají zároveň rozlišovat dle dvou kategorií informací – teoretické a praktické.

Pro návrh aplikace jsou potřeba znalosti z oblasti signálů, zejména těch, co nesou informaci o zvuku. Základní přehled z této oblasti se nachází v první kapitole. Druhá kapitola je věnována studiu hlavních atributů mikrofonů řazených dle jejich základních vlastností.

Dalším nezbytným bodem je třetí kapitola, která popisuje obecné charakteristiky programovacího jazyka C#. Sekce, která se zabývá .NET frameworkem WPF, je však popsána detailněji, zejména část týkající se návrhového vzoru MVVM (zkr. *Model-View-ViewModel*). Tématu této kapitoly byla věnována velká pozornost především z vlastní potřeby ucelit si možnosti a správné užívání zmíněného frameworku.

Poté následují dvě kapitoly kategorie praktické – popis testovaného hardwaru a návrh samotné aplikace na straně klienta.

1. Základní pojmy a teorie

1.1. Akustický signál a jeho vlastnosti

Zvuk je mechanické (akustické) vlnění šířící se pružným prostředím. Jedná se o zhušťování a zřeďování částic. Je vymezen frekvenčním rozsahem lidského ucha, tj. 16 Hz – 20kHz.

Rychlost šíření zvuku je dána pro dané prostředí a zejména závisí na jeho teplotě. Ve vzduchu o teplotě t [°C] má zvuk rychlost danou vztahem:

$$v_t = 331{,}82 \cdot 0{,}61t \; [m \cdot s^{-1}]$$

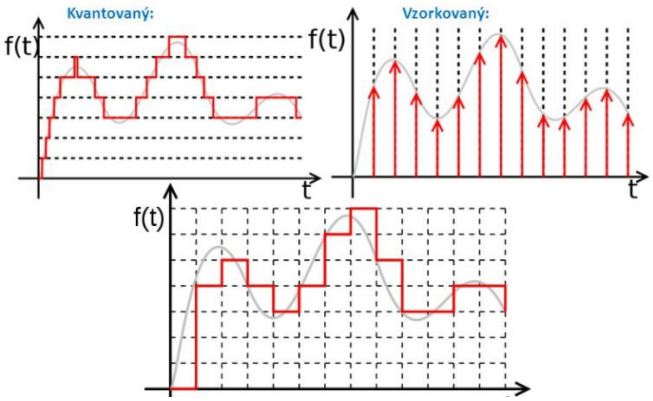
Tabulka 1 - Rychlost šíření zvuku v různých prostředích [35]

LÁTKA/PROSTŘEDÍ	RYCHLOST [m·s^{-1}]
vzduch (13,41°C)	340
voda (25°C)	1500
led	3200
sklo	5200
beton	1700
ocel	5000
helium (25°C)	965

Zvukový signál se v přírodě vyskytuje jako spojitá funkce – jedná se tedy o analogový zvukový signál. Druhým typem je digitální zvukový signál, který představuje číselnou reprezentaci analogového audio signálu. Digitální signál je možné dále zpracovávat pomocí výpočetní techniky [9]. Digitální signál získáme z analogového signálu pomocí A/D převodníku (viz.). Převodem však dojde ke snížení přesnosti záznamu.

Zmíněná snížená přesnost digitálního signálu je omezena [1]:

- **diskretizací** – hodnoty analogového signálu jsou navzorkovány v ekvidistantních časových intervalech
- **kvantizací** – hodnoty jednotlivých vzorků jsou reprezentovány čísly s omezenou přesností (mohou nabývat jen konečně mnoha hodnot)



Obrázek 1 - Vzorkování a kvantování signálu[1]

Kvalitu digitálního signálu určuje vzorkovací frekvence a bitové rozlišení:

- **vzorkovací frekvence**
  - udává počet vzorků za časovou jednotku, definuje tedy kvalitu diskretizace
  - podle Shannonova vzorkovacího teorému by měla být alespoň dvakrát větší, než je nejvyšší frekvence rekonstruovaného signálu
- **bitové rozlišení (Bit rate)**
  - udává počet bitů, kolika je kódován jeden vzorek digitálního signálu, definuje tedy kvalitu kvantizace
  - počet různých hodnot, kterých může jeden vzorek nabývat, je dán vztahem *2n*, kde *n* je počet bitů na vzorek (nejčastěji CD – 16 bit; DVD – 24 bit)

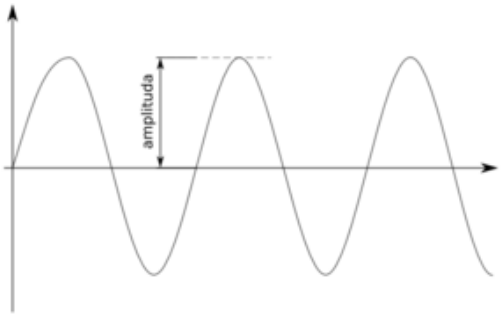
Hustota vzorkování signálu se upravuje dvěma procesy:

- **decimace** – proces, kdy je snižována vzorkovací frekvence
- **interpolace** – proces, kdy je zvyšována vzorkovací frekvence

Dalšími vlastnostmi popisující signál jsou [3]:

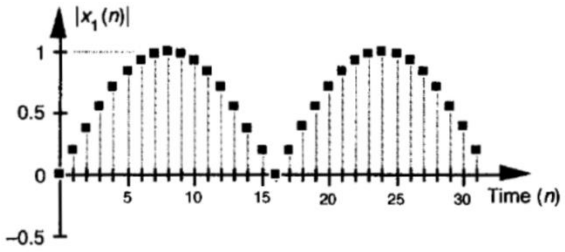
- **amplituda** – maximální odchylka periodicky měnící se veličiny od její střední hodnoty

[1] http://www.wikiwand.com/en/Digital_signal_(signal_processing)



Obrázek 2 - Amplituda[2]

- **magnituda** – absolutní hodnota amplitudy, tj. amplituda převedená pouze do kladných hodnot



Obrázek 3 - Magnituda[3]

- **impulsní odezva (impulzní charakteristika)** – odezva LTI systému (angl. *Linear Time-Invariant* – v překladu *Lineární časově invariantní*), značena malým písmenem *h[t]*, na tzv. *Diracův jednotkový impuls* značen znakem delty *δ[t]*.

  Převod na frekvenční charakteristiku:

$$H(j\omega) = \int_{-\infty}^{\infty} h[t]e^{-j\omega t}dt$$

- **hlasitost zvuku** – subjektivní veličina závislá na velikosti akustického tlaku *p*, kterým zvukové vlnění působí na sluch Užívá se pro ni logaritmického vyjádření v jednotkách decibel (*dB*):

$$L_p = 20log\frac{p}{p_0}$$

[2] https://cs.wikipedia.org/wiki/Amplituda
[3] http://flylib.com/books/en/2.729.1.13/1/

1.2. DSP

DSP (angl. *Digital Signal Processing*) znamená v překladu do češtiny *Digitální Zpracování Signálu*. Jak již z názvu vyplývá, jedná se o číselnou manipulaci se signálem, obvykle za účelem jeho změření, filtrace nebo komprese spojitého analogového signálu [1].

1.3. Fourierova transformace

Fourierova transformace (FT) je integrální transformace převádějící signál z časové oblasti do frekvenční (kmitočtové) [3][27].

Základní myšlenkou Fourierovy transformace je, že každou spojitou funkci lze zapsat jako součet nekonečné řady

$$f(t) = a_0 + \sum_{n=1}^{\infty}(a_n \cos n\omega_0 t + b_n \sin n\omega_0 t) = \sum_{n=1}^{\infty} c_n e^{jn\omega_0 t}$$

Spojitá Fourierova transformace je definována vztahem

$$X(j\omega) = \int_{-\infty}^{\infty} x(t) \cdot e^{-j\omega t} dt$$

1.3.1. DFT

Protože Fourierovy řady lze aplikovat pouze na spojité funkce času a v současné době se číslicové (diskrétní) signály[4] používají daleko častěji než analogové spojité signály, byla vyvinuta metodika použitelná pro zpracování periodických vzorkovaných číslicových signálů.

$$F_n = \sum_{k=0}^{N-1} f_k e^{-j\frac{2\pi}{N}kn}, \qquad n = 0,1,2,\dots N-1$$

DFT (angl. *Discrete Fourier Transform*) bylo založeno na požadavku získat stejný výsledek pro vzorkovaný periodický signál, jako pro původní spojitý.

[4] signál zadán v podobě číslicových řad, často konečných

1.3.2. FFT

Zmíněná metoda DFT má vysokou složitost – $O(N^2)$. Její zdlouhavost se projevuje zejména v případech, kdy dochází k opakovaným výpočtům na mnoha signálech.

Proto v roce 1965 pánové J. W. Cooley a J. W. Tukey představili algoritmus, jež výpočet výrazně zredukoval.

$$F_n = \sum_{k=0}^{N-1} f_k e^{\frac{-2\pi i n k}{N}}, \qquad N = 2^k, kde\ k \in \boldsymbol{N}$$

$$F_n = F'_n + \left(e^{\frac{-2\pi i}{N}}\right)^n F''_n, \qquad pro\ n = \left\{1,2,3, \ldots \frac{N}{2} - 1\right\}$$

$$F_n = F'_n - \left(e^{\frac{-2\pi i}{N}}\right)^n F''_n, \qquad pro\ n = \left\{\frac{N}{2}, \frac{N}{2} + 1, \ldots, N - 1\right\}$$

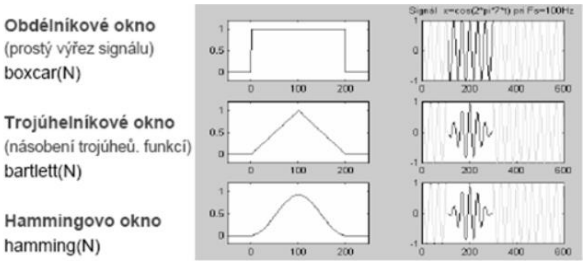
FFT (angl. *Fast Fourier Transform*), neboli Rychlá Fourierova transformace, je způsob výpočtu diskrétní Fourierovy transformace, kterým získáme stejné výsledky, ale mnohem rychleji. Klasická metoda DFT potřebuje $O(N^2)$ operací zatímco FFT pouze $O(N \log(N))$, tj. pro N = 1024 je FFT cca 200 x rychlejší než DFT.

1.3.3. Okénkovací funkce

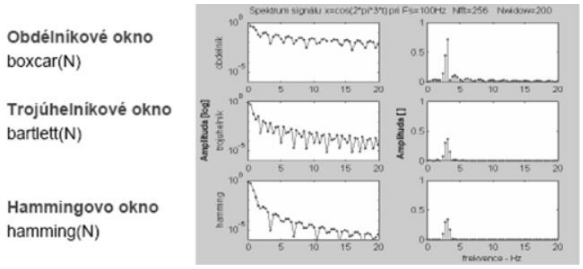
Jelikož signál ve většině případů není periodický, může dojít k rozmazání spektra (objeví se neexistující složky). To lze eliminovat využitím tzv. okénkovacích funkcí (angl. *window function*).

Tato funkce "vyřezává okénka" v analyzovaném signálu a délka okna se stává periodou. Výřezem části signálu se rozumí jeho vynásobení okénkovací funkcí. Násobení v čase se převádí na konvoluci ve spektru. Mezi nejčastěji používané okénkovací funkce se řadí:

- obdélníkové okénko
- trojúhelníkové okénko
- Hammingovo okénko



Obrázek 4 - Okénkovací funkce 1



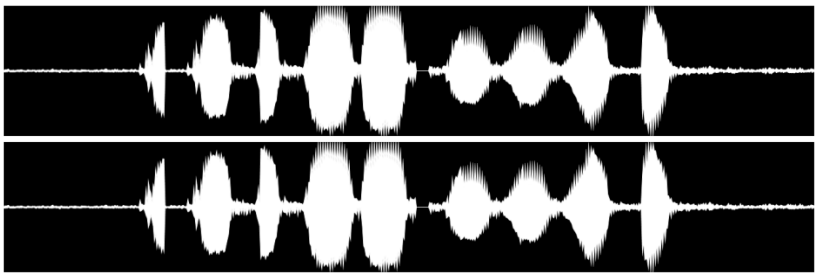
Obrázek 5 - Okénkovací funkce 2

1.4. Vizualizace signálu

Aby bylo možné signál analyzovat, je důležité jej nějakým způsobem vizualizovat. Mezi nejčastěji používané vizualizace patří prosté zobrazení časového průběhu, kmitočtové (frekvenční) spektrum a spektrogram.

1.4.1. Časový průběh signálu

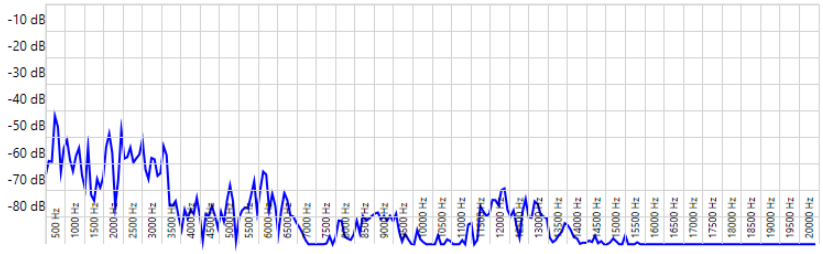
Zobrazení časového průběhu signálu je nejzákladnějším znázorněním audio signálu, kde na osu x je přenesena doba trvání signálu a na osu y jeho maximální/minimální hodnota.



Obrázek 6 - Časový průběh zvukového signálu (ukázka z vyvíjené aplikace)

1.4.2. Kmitočtové spektrum signálu

Dalším způsobem zobrazení zvukového signálu je kmitočtové spektrum, které znázorňuje, jaké frekvence jsou zastoupeny. Obecně se pro výpočet a následné zobrazení signálu v kmitočtové oblasti používá Fourierova transformace [viz. 1.3].



Obrázek 7 - Frekvenční spektrum (ukázka z vyvíjené aplikace)

1.5. Modulace signálu

Modulace je nelineární proces, kterým se mění charakter nosného signálu pomocí modulačního signálu [28][29].

Existuje mnoho různých typů modulací a podle typu nosného signálu se rozdělují na:

- **spojité analogové modulace** – nosným signálem je signál s harmonickým průběhem a modulačním signálem je analogový signál
- **spojité digitální modulace** – nosným signálem je signál s harmonickým průběhem a modulačním signálem je diskrétní/digitální signál
- **diskrétní modulace** – nosným signálem je signál s nespojitým průběhem, často nazýván tzv. *taktovací signál*

Nejčastěji používané druhy modulací signálu:

- PAM (angl. *Pulse-Amplitude Modulation*)
- PWM (angl. *Pulse-Width Modulation*)
- PDM (angl. *Pulse-Density Modulation*)
- PPM (angl. *Pulse-Position Modulation*)
- PCM (angl. *Pulse-Code Modulation*)

- DPCM (angl. *Differential PCM*)
- ADPCM (angl. *Adaptive DPCM*)

1.5.1. PCM

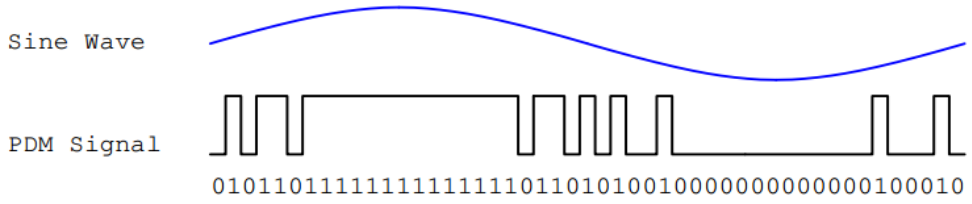
Pulzně kódová modulace je modulační metoda převodu analogového zvukového signálu na signál digitální, vytvořená roku 1937 Britem Alecem Reevsem.

Princip PCM spočívá v pravidelném odečítání hodnoty signálu pomocí A/D převodníku a jejím záznamu v binární podobě.

Audio formát založen na PCM modulaci je nejjednodušší a nejběžněji používaný. Data jsou ukládána nekomprimovaná v podobě celých čísel. Tento formát podporuje bitovou šířku 8, 16 a 32bit.

1.5.2. PDM

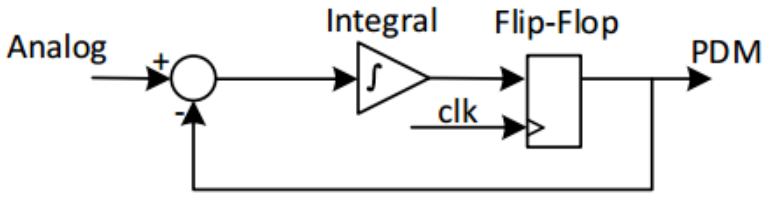
PDM (angl. Pulse-Density Modulation) je modulační metoda převádějící spojitý signál na diskrétní.



Obrázek 8 - PDM signál [19]

Pomocí PDM mohou být data dvou mikrofonů přenášena pouze pomocí dvou drátů. V datovém toku reprezentuje ‘1’ kladný pulz a ‘0’ záporný.

Signál je generován z analogového signálu, který je zpracováván tzv. Delta-Sigma převodníkem [viz. 1.7.1].



Obrázek 9 - Sigma-Delta převodník pro PDM[19]

1.5.3. IEEE Float

IEEE Float formát ukládá data v podobě čísel s plovoucí čárkou – oproti PCM, který je zaznamenává v celých číslech. To vede k úspoře místa potřebného k uložení záznamu.

Tento formát podporuje pouze 32 bit nebo 64 bit bitovou šířku.

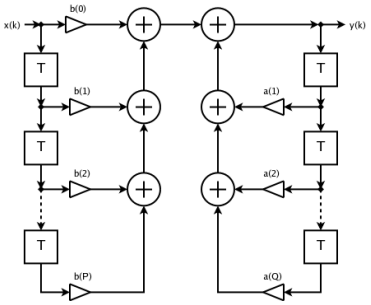
1.6. Filtry

Signál je pro některé vyhodnocovací obvody nutno frekvenčně upravit, např. odstranit střídavé či stejnosměrné složky signálu, omezit frekvenční pásmo nebo snížení šumu resp. brumu.

Základními typy jsou FIR (angl. *Finite Impulse Response*, filtr s konečnou odezvou) a IIR filtry. Jelikož je do aplikace implemetován pouze IIR filtr, nebude zde FIR filtr popisován.

1.6.1. IIR

Filtr s nekonečnou impulzní odezvou (angl. *Infinite Impulse Response*) je diskrétní lineární filtr, který má nekonečnou impulzní odezvu. Vyžaduje minimálně jednu zpětovazební smyčku – IIR je rekurzívní filtr [4][5].



Obrázek 10 - IIR filtr[5]

Příklad IIR filtru 1. řádu:

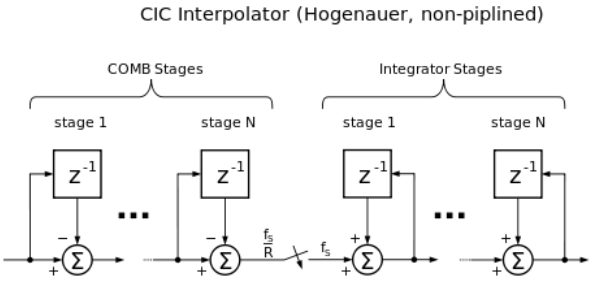
$$y[n] = C_1 y[n-1] + C_2 x[n]$$

Základní vlastnosti:

- filtr může být nestabilní
- filtr je rekurzívní (tj. má zpětnou vazbu)
- malý řád přenosové funkce
- velká citlivost na kvantování

1.6.2. CIC

CIC (angl. Cascaded Integrator-Comb) je optimalizovaným FIR filtrem zahrnující interpolátor a decimátor do své architektury.



Obrázek 11 - CIC filtr[6]

[5] https://upload.wikimedia.org/wikipedia/commons/d/d5/IIR-filter.png

[6] https://upload.wikimedia.org/wikipedia/commons/thumb/4/4b/CIC_interpolator.svg/500px-CIC_interpolator.svg.png

CIC filtr byl vynalezen E. B. Hogenauerem jako třída FIR filtru využívaný v multi-rychlostním zpracování digitálního signálu. Skládá se z jednoho nebo více párů integrátorů a diferenciátorů (COMB) [4][5].

$$H(z) = \left[\sum_{k=0}^{RM-1} z^{-k}\right]^{N} = \left(\frac{1 - \backslash z^{-RM}}{1 - z^{-1}}\right)^{N}$$

R…interpolace či decimace

M…množství vzorků ve fázi

N…počet fází filtru

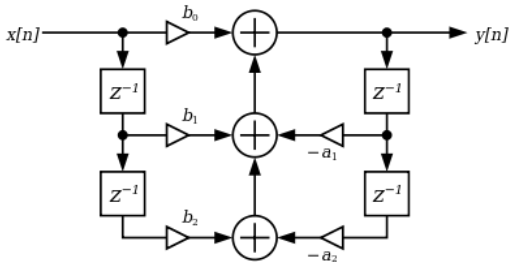
1.6.3. BiQuad filtr

BiQuad filtr je lineární IIR filtr definován diferenční rovnicí [15]:

$$y[n] = \frac{b_0}{a_0} * x[n] + \frac{b_1}{a_0} * x[n-1] + \frac{b_2}{a_0} * x[n-2] - \frac{a_1}{a_0} * y[n-1] - \frac{a_2}{a_0} * y[n-2]$$

Název filtru je zkratka slova bi-kvadratický, protože jeho přenosová funkce obsahuje dvě kvadratické rovnice:

$$H(z) = \frac{b_0 + b_1 * z^{-1} + b_2 * z^{-2}}{a_0 + a_1 * z^{-1} + a_2 * z^{-2}} = \frac{\frac{b_0}{a_0} + \frac{b_1}{a_0} * z^{-1} + \frac{b_2}{a_0} * z^{-2}}{1 + \frac{a_1}{a_0} * z^{-1} + \frac{a_2}{a_0} * z^{-2}}$$



Obrázek 12 - BiQuad filtr[7]

[7] https://upload.wikimedia.org/wikipedia/commons/thumb/c/c3/Biquad_filter_DF-I.svg/400px-Biquad_filter_DF-I.svg.png

1.7. A/D převodník

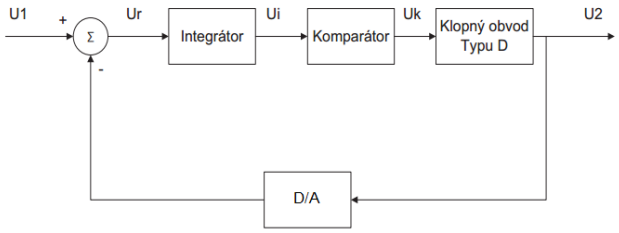
Analogově digitální převodník je elektronická součástka určená pro převod spojitého/analogového signálu na signál diskrétní/digitální.

1.7.1. Sigma-Delta převodník

Delta-Sigma převodník umožňuje dosáhnout velmi vysoké linearity převodu při vysokém rozlišení (až 24 bit). Nižší je však rychlost převodu.

Převodník se skládá ze sigma-delta modulátoru a číslicového filtru. Základními obvody modulátoru jsou:

- integrátor (dolní propust)
- napěťový komparátor
- klopný obvod typu D (překlápěný hodinovým signálem s frekvencí f_0
- zpětno-vazební větev s jednobitovým D/A převodníkem



Obrázek 13 - Sigma-Delta převodník

2. Mikrofon

Mikrofon je elektroakustické zařízení, které přeměňuje akustický signál na signál elektrický. Vlastnosti a zvolená konstrukce mikrofonu ovlivňují kvalitu přeměny akustického signálu.

První mikrofon byl vynalezen 4. března 1877 tvůrcem gramofonu Emilem Berlinerem [20][35].

2.1. Základní vlastnosti mikrofonů

Citlivost [mV/Pa]

Citlivost mikrofonu je dána poměrem výstupního napětí k akustickému tlaku, který toto napětí vybudil v místě akustického vstupu mikrofonu.

SNR

SNR je anglická zkratka *Signal-to-Noise Ratio*, která vyjadřuje poměr signálu a šumu. Signálem se rozumí hlavní informace a šum je vedlejší informací měnící signál. Šum nelze zcela odstranit, ale lze jej minimalizovat. Hlavní část šumu je způsobena snímacím prvkem.

$$SNR = \frac{signál}{šum} = \frac{aritmetický\ průměr}{směrodatná\ odchylka} = \frac{\overline{X}}{s} = \frac{1}{rsd}$$

Vnitřní (výstupní) impedance

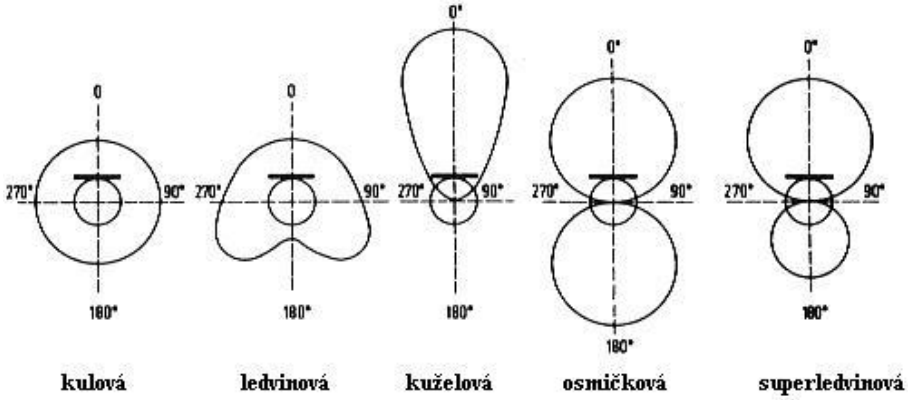
Impedance je měřena na výstupních svorkách mikrofonu. Je dána poměrem výstupního napětí mikrofonu naprázdno k výstupnímu proudu nakrátko. Impedance je podstatná pro správné připojení mikrofonu k zesilovači.

Směrová charakteristika

Směrová charakteristika udává citlivost mikrofonu na úhlu, který svírá akustická osa mikrofonu s osou akustického zdroje. Tyto charakteristiky se znázorňují jako polární diagramy. Zpravidla se dělí na:

- kulové – citlivost mikrofonu je ve všech směrech stejná

- osmičkové – citlivý zpředu a zezadu, po stranách necitlivý
- kardioidní (ledvinová)
- hyperkardioidní (superledvinová)
- kuželové

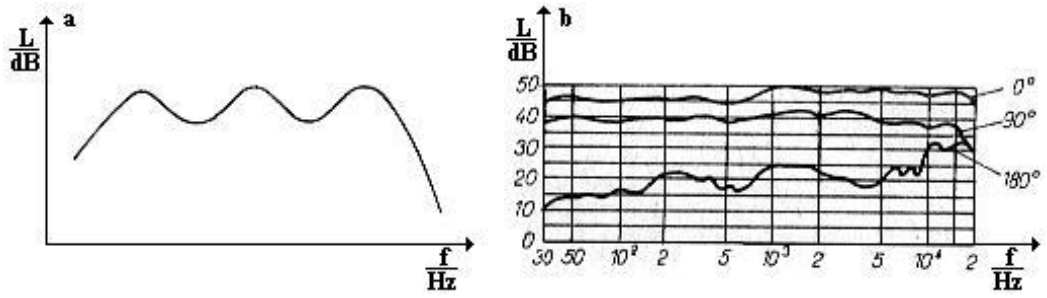


Obrázek 14 - Směrová charakteristika mikrofonu[8]

Frekvenční charakteristika

Tato charakteristika udává rozsah přenášeného pásma akustických signálů. Jedná se o závislost výstupního napětí mikrofonu na frekvenci (při konstantním akustickém tlaku).

Pro kvalitní záznam je potřeba zajistit rovnoměrnou frekvenční charakteristiku. Její nerovnoměrnost (tj. zvlnění) by nemělo být větší než ±5dB.



Obrázek 15 - Frekvenční charakteristika (a – uhlíkový mikrofon; b – dynamický mikrofon, stupně udávají, z jakého úhlu byl mikrofon proměřován)[9]

[8] http://fyzika.jreichl.com/data/E_elektroakustika_soubory/image020.jpg
[9] http://fyzika.jreichl.com/data/E_elektroakustika_soubory/image014.jpg

2.2. Nejběžnější typy mikrofonů dle jejich principu fungování

Do nejběžnějších mikrofonů se řadí:

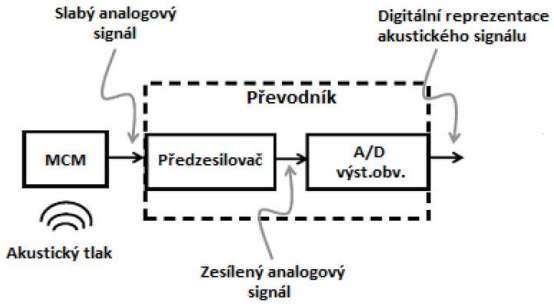
- piezoelektrické (krystalové) [Obrázek 31]
  - využívají piezoelektrického jevu, při němž deformací výbrusu krystalu Seignettovy soli vzniká na jeho plochách elektrický náboj
  - v minulosti se využívaly převážně v systémech veřejného ozvučení (v 50. letech 20. století)
  - dodnes se na tomto principu vyrábějí bezmembránové mikrofony pro snímání zvuku pod vodou
- elektrostatické (kondenzátorové/kapacitní) [Obrázek 32]
  - složeny ze dvou od sebe izolovaných elektrod
  - fungují na kondenzátorovém principu
  - patří k nejkvalitnějším snímačům zvuku – vyznačují se vyrovnanou kmitočtovou charakteristikou, vysokou citlivostí, malým zkreslením a obecně vysokou stabilitou
  - složitější konstrukce mikrofonu se však promítá i do jeho vyšší ceny
  - náchylnost na vlhkost
- elektretové [Obrázek 34]
  - druh elektrostatického mikrofonu, jehož pevná elektroda je opatřena vrstvou elektretu[10]
  - jejich výhodou je jeho nízká cena a stále s kvalitou elektrostatického mikrofonu
  - díky své kvalitě najde uplatnění v profesionálním využití a díky ceně je vyráběn i pro nenáročné aplikace – do počítačů, telefonů apod.
- elektrodynamické
  - vyrábějí se ve dvou provedeních – cívkovém s membránou [Obrázek 35] a páskovém bez membrány [Obrázek 36]

[10] dielektrický materiál s velmi dobrými izolačními vlastnostmi a časovou stálostí nesoucí permanentní elektrický náboj

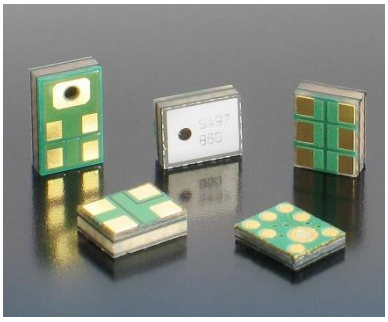
- fungují na principu elektromagnetické indukce – při pohybu vodiče v magnetickém poli se ve vodiči indukuje napětí
- méně citlivé než elektrostatické mikrofony – lépe zpracují například hlasitý zpěv při živých vystoupeních, ozvučení veřejných shromáždění apod.
- odolné vůči nepříznivým vlivům

2.3. MEMS mikrofony

Jedná se o miniaturní křemíkový mikrofon, který je vyroben v podobě jediné integrované součástky (CMOS čipu). Součástka obsahuje křemíkovou membránu upevněnou na křemíkový čip, integrovaný předzesilovač a případně i další pomocné obvody, např. aktivní filtry pro odstranění nežádoucích složek přijímaného zvuku. Do MEMS mikrofonů jsou již z výroby integrovány A/D převodníky, což zajišťuje jednodušší zapojení do digitálních systémů. Ve většině případů se jedná kapacitní mikrofony, snímá se tedy změna tlaku. Celý mikrofon je jedna kompaktní součástka velikosti maximálně několika mm [21].



Obrázek 16 - Blokové schéma MEMS mikrofonu složené ze snímací části a převodníku [35]



Obrázek 17 - MEMS mikrofony [21]

2.3.1. MEMS technologie

MEMS je anglická zkratka slov „Micro-Electro-Mechanical Systems“, do češtiny přeloženo jako Mikro-Elektro-Mechanické Systémy. Jedná se o integraci co nejvíce elementů do jedné součástky o miniaturních rozměrech. Mezi vkládané prvky patří různé mechanicke elementy, senzory, řídící či vyhodnocovací elektrotechnika. Jejich integrace probíhá za využití různých výrobních technologií.

Zatímco elektronické části jsou vyráběny standardními postupy (CMOS, Bipolar či BiCMOS), mikromechanické komponenty jsou zhotovovány prostřednictvím vhodných mikroobráběcích procesů, které selektivně vyleptávají části křemíkového plátku, nebo přidávají nové strukturální vrstvy. Vytvářejí tak mechanická a elektromechanická zařízení[22][23].

Pomocí technologie MEMS lze vytvářet mikroskopické systémy o rozměrech několika milimetrů až mikrometrů.

Technologie MEMS nalézá své uplatnění v širokém spektru oborů [23]:

- biotechnologie
    - mikrosystémy pro identifikaci a rozšiřování DNA
    - mikroobráběcí rastrovací tunelové mikroskopy (STM)
    - biočipy pro detekci riskantních chemických a biologických agentů
- komunikace
    - induktory a laditelné kondenzátory (RF-MEMS)
- akcelerometry
    - airbagy
- medicína
    - mikro kapsle produkující inzulin

3. Programovací jazyk C#

Desktopová aplikace byla vytvořena v jazyce C#. Jedná se o vysokoúrovňový objektově orientovaný jazyk vyvinutý firmou Microsoft, běžící na platformě .NET. Je založen na vlastnostech vícero jazyků, z čehož největší podíl tvoří Java a C/C++ (podobnost s jazykem Java je cca 80%).

První verze vyšla v roce 2002, kdy Bill Gates prohlásil, že na vývojářský trh přichází naprostá špička mezi ostatními. Byl tehdy představen spolu s celým vývojovým prostředím .NET.

K tomu, abyste mohli začít programovat v C# a využívali všechny jeho nabízené možnosti a funkce, potřebuje vývojář šikovný nástroj, jakým je například Microsoft Visual Studio – na tvorbu simulátoru byla využita verze 2015.

Plná verze zmíněného softwaru není zadarmo, ale můžeme využít jeho Community verzi [33], která je zdarma pro samostatné vývojáře, open source projekty, vědecký výzkum, vzdělávání a malé profesionální týmy – ve velkých profesionálních týmech by se již neměla využívat.

Jak již bylo výše zmíněno, C# je spjatý s platformou .NET. Chcete-li tedy spustit aplikace vytvořené v tomto jazyce, musíte mít na svém počítači nainstalován Microsoft .NET Framework, což je prostředí nutné pro běh .NET aplikací – jak spouštěcí rozhraní, tak potřebné knihovny. Pro majitele operačního systému Windows je k dispozici zdarma jako samostatná komponenta, která se do systému může doinstalovat.

3.1. WPF (Windows Presentation Foundation)

Během vývoje jsem použila technologie Windows Presentation Foundation (WPF), ve kterém jsem dodržovala návrhový vzor Model-View-ViewModel (MVVM).

Windows **P**resentation **F**oundation je framework umožňující vytvářet grafické uživatelské prostředí a je vhodný pro komplexní tvorbu bohatých formulářových aplikací. Součástí .NET frameworku je od verze 3.0. Disponuje širokou paletou formulářových

prvků a nabízí spoustu možností jejich stylování. Pro vytváření GUI lze využívat značkovací jazyk XAML, čímž je možné oddělit funkčnost od vzhledu aplikace [6].

3.1.1. WPF vs. WinForms

WPF je nástupcem starší technologie WinForms, oproti které se ale v mnoha aspektech liší [14].

Hlavními výhodami WPF oproti WinForms jsou:

- **DataBinding** - lepší oddělení dat/funkcí od samotné grafické interpretace. Ve spojení s návrhovým vzorem MVVM (**M**odel, **V**iew, **V**iew**M**odel) se jedná o velmi užitečný nástroj.
- **XAML** - cokoliv, co vytvoříte v designeru, má svou interpretaci v XAMLu
- **Stylování** – stylování ovládacích prvků probíhá podobně jako stylování webových stránek pomocí CSS. Je možné vytvořit více stylů a dle potřeby je využívat.
- **Hardwarová akcelerace GUI** - podstatně rychlejší vykreslování.

3.2. XAML

Deklarativní jazyk XAML slouží ke kódování prezentační vrstvy aplikace, tj. pro popis, jak má okno aplikace vypadat. XAML vychází z XML [26].

Zkratka XML označuje e**X**tensible **M**arkup **L**anguage, tedy rozšiřitelný značkovací jazyk. XAML potom označuje e**X**tensible **A**pplication **M**arkup **L**anguage, tedy jednoduše řečeno XML se značkami pro tvorbu aplikací.

XML dokument se skládá z elementů. Struktura dokumentu je stromová. Elementy v sobě tedy mohou obsahovat libovolné množství dalších elementů. Každý XML dokument však obsahuje právě jeden kořenový element.

Elementy se zapisují do lomených závorek, např. `<Textblock />` či `<Textblock></Textblock>`.

3.3. Kód na pozadí (Code Behind)

Prezentační část aplikace je napsána v XAMLu, což ale popisuje pouze to, jak bude aplikace vypadat. To, co bude aplikace dělat, je popsáno v kódu na pozadí, tzv. Code Behind, který obsahuje volání logiky aplikace.

Code Behind hlavního prezentačního okna (*MainWindow.xaml*) vypadá takto:

```
using System.Windows;
namespace DP_Fulinova
{
    // Interaction logic for MainWindow.xaml
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

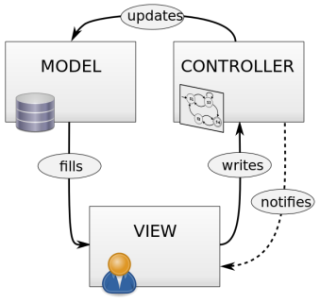
Formulář je reprezentovaný třídou *MainWindow*, která dědí z *Window* (protože MainWindow.xaml je typu Window).

V konstruktoru formuláře se volá metoda *„InitializeComponent()“*, která vnitřně naparsuje XAML a vytvoří podle něj instance jednotlivých ovládacích prvků.

Pokud je některému ovládacímu prvku přidělena událost pomocí postranního panelu s vlastnostmi prvků, objeví se tato událost v kódu na pozadí. Toto řešení je sice funkční, ale ne zcela správné. Pro framework WPF byl speciálně vyvinut návrhový vzor MVVM, při jehož dodržování bychom měli Code Behind nechat prázdný a k volání událostí využít tzv. ***Data Binding*** a ***Command***.

3.4. Návrhový vzor MVC

Model-View-Controller je návrhový vzor, který dělí aplikaci do tří nezávislých komponent – na datový model aplikace (*Model*), uživatelské rozhraní (*View*) a řídící logiku, která má na starosti tok událostí (*Controller*). Modifikace některé z komponent má jen minimální dopad na ostatní.



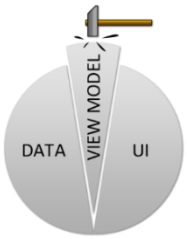
Obrázek 18 - Model - View - Controler[11]

Přestože se jedná o velice používaný a široce rozšířený návrhový vzor, jeho využití ve WPF není zcela vhodné – nevyužívá funkci oboustranného Data Bindingu [7].

3.5. Návrhový vzor MVVM

Model-View-ViewModel je návrhový vzor pro WPF vytvořen architektem WPF/Silverlight Johnem Grossmanem, který vychází z výše zmíněného MVC (Model-View-Controller) [25][30].

Jak již bylo zmíněno výše, události volané v kódu na pozadí jsou nahrazeny Data Bindingem a Commandem (obdoba volání událostí z WinForms).



Obrázek 19 – MVVM [25]

Hlavní myšlenkou tohoto návrhového vzoru je vytvořit třídu, která si bude držet stav aplikace (*ViewModel*), podle níž se vykreslují ovládací prvky uživatelského rozhraní (*View*). Díky již několikrát zmíněnému Data Bindingu komunikace funguje oboustranně (není-li zadáno jinak nastavením jiného módu, viz.) – zadá-li uživatel do uživatelského rozhraní nějaké údaje, automaticky jsou předány do části ViewModel.

[11] https://upload.wikimedia.org/wikipedia/commons/thumb/b/b4/MVC_Diagram_(Model-View-Controller).svg/220px-MVC_Diagram_(Model-View-Controller).svg.png

Obrázek 20 - Model - View - ViewModel

Model

Slouží pouze pro popis datových struktur, se kterými pracuje ViewModel. Tato vrstva je zodpovědná pouze za data a aplikační logiku.

View

View reprezentuje uživatelské rozhraní v jazyce XAML. Tato vrstva je zodpovědná pouze za vzhled aplikace. Pouze potřebuje vědět, jaké zdroje a struktury dat mu jsou vystaveny z ViewModelu.

ViewModel

ViewModel spojuje třídy Model a View a drží si stav aplikace. Jedná se o nejdůležitější třídu návrhového vzoru MVVM. Jejími základními kameny jsou kolekce *ObservableCollection<T>* a rozhraní *INotifyPropertyChanged*.

Kolekce *ObservableCollection<T>* hlásí, když je přidán či odebrán její prvek. Rozhraní *INotifyPropertyChanged* popisuje událost, která nastane, pokud se změní některá z vlastností ViewModelu.

3.6. Data Binding

Data Binding je technika propojení prvků a elementů s daty. Jedná se o zautomatizovanou možnost zobrazit uživateli data na UI a umožnit jejich editaci bez nutnosti manuálního vytváření událostí v kódu na pozadí (tzv. Code Behind). Tato datová vazba nahrazuje zmíněného správce událostí a vede k zpřehlednění a zjednodušení kódu. Správci událostí se tedy stávají skrytými a zmenšuje se prostor pro chyby v jejich obsluze [31].

Data Binding nabízí mnoho možných využití. Tato práce ale není určena k výuce bindingu a proto zde budou uvedeny pouze stručně základní vlastnosti, které byly využity během návrhu koncové aplikace.

3.6.1. Binding

U datových vazeb existuje „zdroj“ a „cíl“. Zdrojem se rozumí libovolné datové hodnoty a cílem je pak prvek, na který tato data navazujeme.

Datové vazby mohou být vytvořeny mezi mnoha elementy. V základu by se dalo hovořit o čtyřech typech:

- vazby mezi jednotlivými grafickými prvky
- vazby na zdroje dat v podobě databáze
- vazby na zdroje dat v podobě xml souboru
- vazby na strukturu objektů

Z toho vyplývá, že tato technologie se může využít téměř kdekoliv, kde potřebujeme zobrazit, nebo jinak editovat data.

A protože se říká „Lepší ukázka než tisíc slov“, zde jsou uvedeny základní typy použití:

I.

```
<Label Content="{Binding FftLength}" />
```

Prvek *Label* žádá *ViewModel* o hodnotu proměnné jménem *FftLength*. Proměnná by měla být typu String - na této úrovni se však typovost nekontroluje a případné chyby se projeví až při běhu aplikace.

II.

```
<Label Content="{Binding ElementName=ComboBoxDevice,
Path=SelectedIndex}" />
```

Prvek Label tentokrát nekomunikuje s ViewModelem, ale žádá o hodnoty View. Parametr ElementName určuje jméno požadovaného ovládacího prvku, který v tuto chvíli slouží jako zdroj dat. Parametr Path určuje, o jakou vlastnost volaného prvku má Label zájem – v tomto

případě prvek žádá o název vybrané položky ComboBoxu. Vyjádříme-li vlastnost pomocí tečkové notace, získáme zápis ComboBoxDevice.SelectedIndex.

III.

```
<TextBox Text="{Binding FftLength, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}" />
```

Zde žádá *ViewModel* o hodnotu proměnné prozměnu prvek typu *TextBox*. Tato ukázka je zde uvedena z důvodu příkladu definice módu a chování datové vazby.

Tento prvek opět žádá o hodnotu proměnné *FftLength* s tím rozdílem, že skrze *TextBox* lze hodnotu proměnné nejen přijímat, ale také editovat z uživatelského prostředí. To, jakým směrem bude povolena aktualizace hodnot, je dáno parametrem *Mode*. Zároveň lze pomocí parametru *UpdateSourceTrigger* definovat, kdy bude aktualizace provedena.

IV.

```
<RadioButton x:Name="rb_1"  Command="{Binding rb_1}"/>
<ComboBox ItemsSource="{Binding Devices}" IsEnabled="{Binding
ElementName=rb_1, Path=IsChecked}" SelectedItem="{Binding
SelectedDevice, Mode=TwoWay}"/>
```

Toto je další ukázka komunikace mezi prvky formuláře (*View*), tj. pokud je daný RadioButton *„rb_1“* aktivní, ComboBox je povolen *(IsEnabled="{Binding ElementName=rb_1, Path=IsChecked}")*.

V této ukázce se využívá též automatického naplnění listu položek ComboBoxu z kódu – *ItemsSource = "{Binding Devices}"*, kde *CaptureDevice* je pole snímaných zařízení.

Jaká položka byla zvolena, se dává kódu na vědomí bindováním vlastnosti *„SelectedItem“*.

Mode – Módy datových vazeb

Pomocí módu se vyjadřuje, jak bude komunikace mezi zdrojem a cílem probíhat:

- **OneWay:** komunikuje pouze jednosměrně – aktualizuje cíl ze zdroje
- **OneTime:** komunikuje stejně jako OneWay pouze jednosměrně, navíc je limitován pouze jednou aktualizací cíle – další změny se již v cíli neprojeví

- **OneWayToSource:** komunikace je opět jednosměrná, ale probíhá opačným směrem – z cíle se aktualizuje zdroj
- **TwoWay:** komunikace probíhá obousměrně – cíl může aktualizovat zdroj a zdroj muže aktualizovat cíl

V definici Bindingu se to poté zapisuje `Mode=TwoWay`.

3.6.2. Command (Příkaz)

Jedná se o metodu ViewModelu, kterou volá View. Tato metoda se nejčastěji používá ve spojení se stisknutím tlačítka a je náhradou za událost *Button.OnClick()*.

```
<Button Command="{Binding PlayCommand}" />
```

View volá metodu nastavením vlastnosti *Command* u prvku *Button*. Zároveň je možné předat i parametr metody, je-li potřeba – přidáním následující definicí vlastnosti prvku:

```
CommandParameter="{Binding ElementName=FftWindow, Path=SelectedItem}".
```

Zde je vazba na příkaz *PlayCommand*, který reprezentuje třídu *DelegateCommand*, která implikuje rozhraní *ICommand* .

4. Hardwarová realizace

Aplikace byla vyvíjena pro vícevláknové snímání a zpracování dat z pole 64 mikrofonů. Mikrofony byly umístěny na vývojovou desku MicroZed™, která je blíže popsána níže [viz. A.2].

4.1. Vývojová deska MicroZed™

MicroZed™ je vývojová deska založená na Xilinx Zynq®-7000 AP SoC [viz. 4.1.1]. Konstrukce desky umožňuje její využití jednak jako samostatné evaluační desky, anebo v kombinaci s nosnou kartou jako integrovatelný SOM [viz. 4.1.2][34].

Deska je osazena integrovaným obvodem Xilinx Zynq XC7Z010-CLG400, který obsahuje dvou-jádrový procesor s ARM architekturou Cortex-A9 s maximální frekvencí 667 MHz. Procesor je vybaven 32 kB instrukční a datovou cache pamětí úrovně L1 a 512 kB cache pamětí úrovně L2. Souhrn technických údajů je uveden v tabulce [viz. Tabulka 2].

Tabulka 2 - Technické údaje procesoru Xilinx Zynq XC7Z010-CLG400 [34]

	XC7Z010-CLG400
Procesor - jádro	Dual-core ARM® Cortex™-A9 MPCore™ s CoreSight™
Procesor - rozšíření	NEON & Single/Double Precision Floating Point
Maximální frekvence	667 MHz (-1); 733 MHz (-2); 800 MHz (-3)
L1 cache	32 kHz instrukční cache, 32 kHz datová cache
L2 cache	512 kB
On-Chip paměť	256 kB
Podpora externí paměti	DDR3, DDR3L, DDR2, LPDDR2
Podpora externí statické paměti	2x Quad-SPI, NAND, NOR
DMA kanály	8 (4 pro část „Programmable Logic“)
Periférie	2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO, 2x USB 2.0 (OTG), 2x Tri-mode Gigabit Ethernet, 2x SD/SDIO
Propojení výpočetní části s programovatelnou logickou částí	2x AXI 32b Master 2x AXI 32b Slave 4x AXI 64b/32b Memory AXI 64b ACP, 16 Interrupts

4.1.1. AP SoC

AP SoC (angl. *All programmable System on Chip*) je systém, který se skládá z jednoho a více procesoru, řadiče pamětí (externích a interních), sběrnicového propojení a specifických periferií – UART, VGA atd.

Jedná se v podstatě o FPGA na vyšší úrovni. FPGA funkce je stále zachována – čip je pouze rozšířen o procesorový subsystém (procesory IBM, ARM, aj.).

4.1.2. SOM

SOM (angl. *System On Module*) je rozšířením výše zmíněného SoC [viz. 4.1.1] a SiP (angl. *System in Package*). Jedná se o typ SBC (angl. *Single Board Computer* – malý počítač s jednou deskou plošných spojů, jako je např. Raspberry Pi, Arduino, Intel Edison nebo 64bitový AMD Gizmo Board)

4.2. PDM mikrofon

PDM (angl. *Pulse Density Modulation*) mikrofony, někdy nazývány digitální mikrofony, se skládají z následujících částí:

- mikrofonový prvek (obvykle se jedná o elektretové kapsle)
- analogový předzesilovač
- PDM modulátor [viz. 1.5.2]
- logické rozhraní

Analogový signál z mikrofonového prvku je nejprve zesílen, a pak navzorkován a kvantován v PDM modulátoru. Modulátor kombinuje operace kvantování a tvarování šumu. Výstupem je jeden bit o vysoké vzorkovací frekvenci. Díky tvarování šumu je hluk ve zvukovém pásmu relativně nízký, zatímco šum nadzvukového pásma je poměrně vysoký. Logické rozhraní je odpovědné za hlavní hodinový signál a přenášení navzorkovaného bitstreamu [11].

Jedno-bitová data jsou posílány buď na vzestupnou či sestupnou hranu hlavního hodinového signálu. Většina PDM mikrofonů podporuje stereo provoz, kdy jeden mikrofon

posílá data na náběžnou hranu hodinového signálu a druhý mikrofon na hranu sestupnou. Za rozdělení dvou binárních streamů je odpovědný PDM přijímač.

Posílání dat na nástupnou a sestupnou hranu je využito i v tomto projektu, kdy je zapotřebí získávat data až z 64 mikrofonů.

4.3. TCP/IP komunikace

TCP/IP je množinou protokolů vycházející z ISO/OSI. Původním záměrem bylo vytvoření komunikačního protokolu ministerstva obrany USA pro sjednocení počítačové komunikace v rámci ARPANET [16].

Rodina protokolů TCP/IP předpokládá existenci čtyř vrstev:

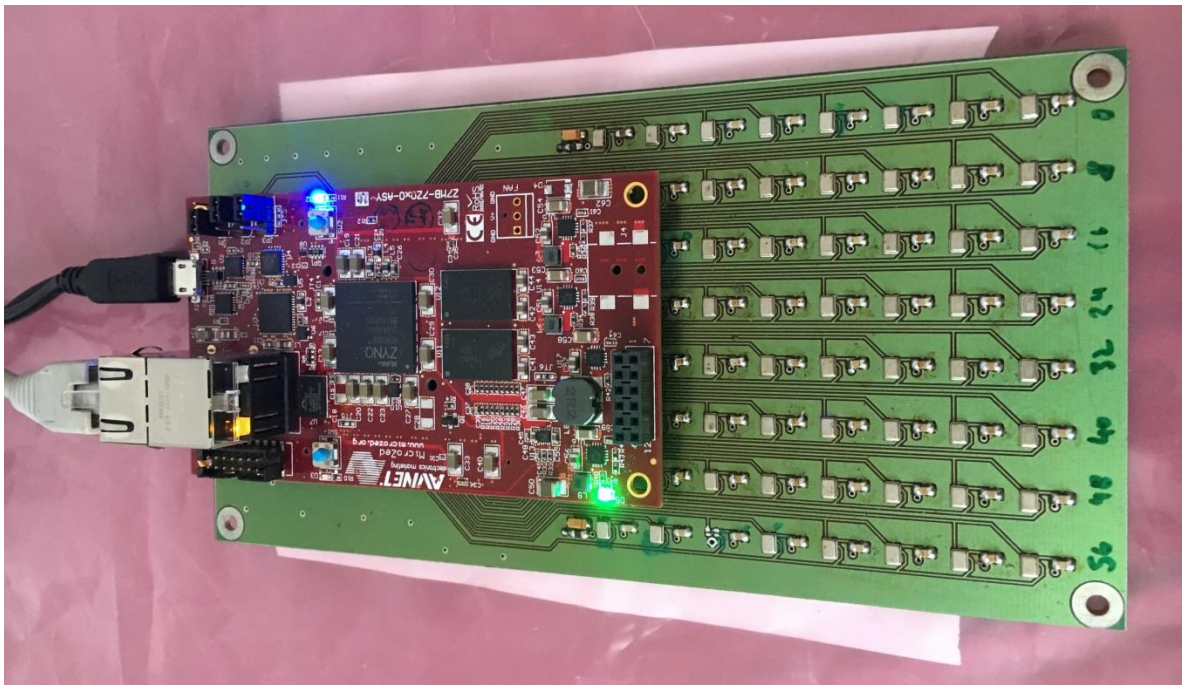
- **aplikační**
  - zajišťuje přenos a srozumitelnost zpráv
  - např. TELNET, FTP, SMTP, WWW, DNS, …
- **transportní**
  - TCP
    - zajišťuje navázání/ukončení spojení a zaručení celistvosti zprávy
  - UDP
    - nezaručuje spolehlivost ale rychlost spojení
- **síťová**
  - IP
    - zajišťuje rychlé doručení dat přes případné uzly
    - pouze částečná detekce chyb (kontrolní součet hlavičky)
  - ICMP
    - přenáší zprávy o chybách a řídící zprávy
  - ARP
    - zabezpečuje pro IP fyzické adresy (MAC) podle logické IP adresy
  - RARP

- Reverse ARP
- zajišťuje logickou adresu k fyzické

- **vrstva síťového rozhraní**
  - zajišťuje přenos rámců (frame) mezi dvěma přímo propojenými počítači

<table>
<tr><th>OSI</th><th>TCP/IP</th><th colspan="8">Aplikace a protokoly</th></tr>
<tr><td>7. aplikační
6. presentační
5. relační</td><td>Aplikační vrstva</td><td>telnet</td><td>FTP</td><td>TFTP</td><td>SMTP</td><td>RIP</td><td>DNS</td><td colspan="2">Ostatní</td></tr>
<tr><td>4. transportní</td><td>Transportní vrstva</td><td colspan="4">TCP</td><td colspan="4">UDP</td></tr>
<tr><td rowspan="2">3. síťová</td><td rowspan="2">Síťová vrstva</td><td colspan="4" rowspan="2">IP</td><td colspan="2">ICMP</td><td colspan="2" rowspan="2"></td></tr>
<tr><td colspan="2">ARP</td><td>RARP</td></tr>
<tr><td>2. linková
1. fyzická</td><td>Vrstva síťového rozhraní</td><td colspan="3">token ring</td><td colspan="3">ethernet</td><td colspan="3">jiné typy protokolů</td></tr>
</table>

Obrázek 21 - Rodina protokolů TCP/IP [16]

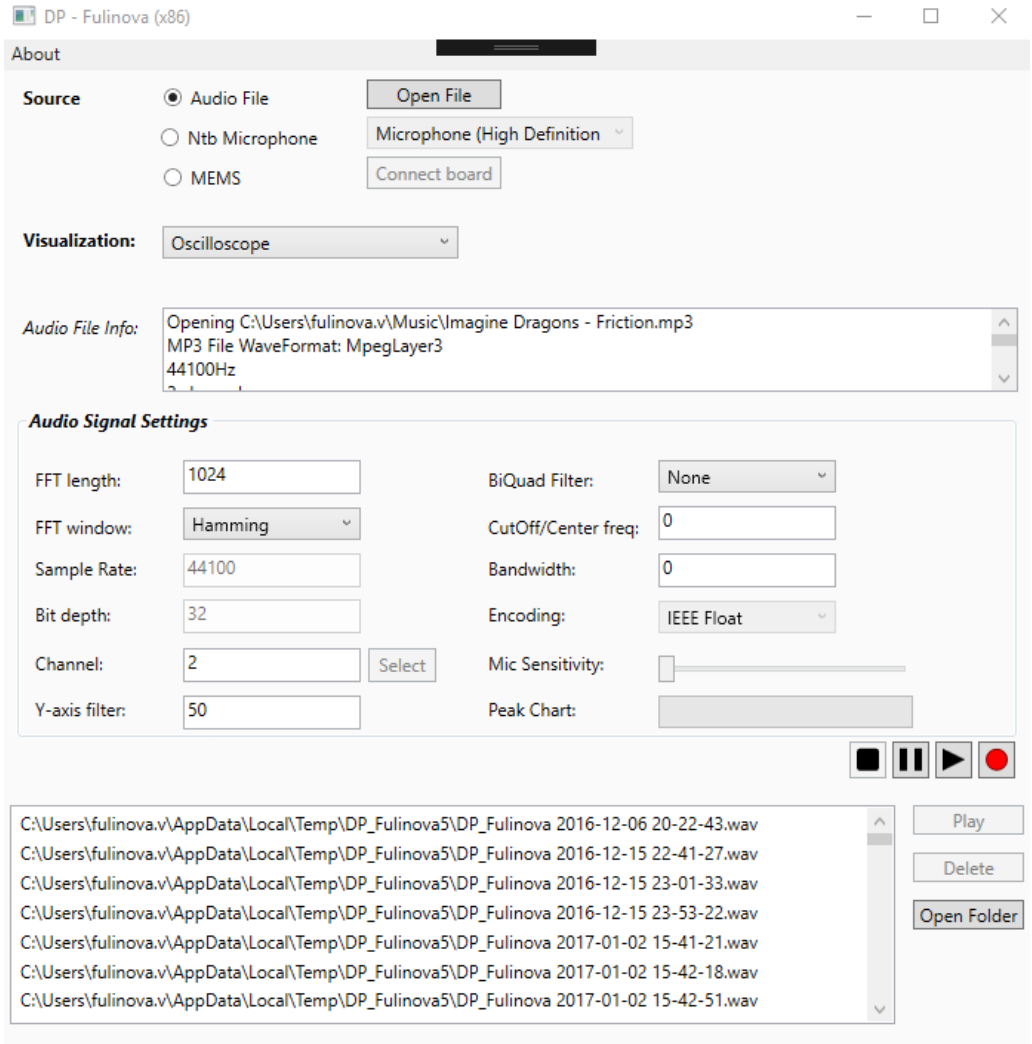


Obrázek 22 - Zapojená vývojová deska MicroZed rozšířená o pole 64 MEMS mikrofonů

5. Návrh aplikace

Aplikace byla navržena v jazyce C# za využití WPF frameworku [viz. 3.1] a jako celek prošla několika úrovněmi návrhu. Nejprve bylo zapotřebí naučit se se samotným zpracováním a vykreslením audio signálu. Proto prvotní fází bylo zpracování audio souboru formátu MP3 či WAV. Dalším krokem bylo zpracování real-time signálu přijímaného z hardwaru/mikrofonu počítače za pomoci WASAPI (angl. *the Windows Audio Session API*). Konečným a hlavním krokem bylo zpracování audio signálu z vývojové desky osazené polem 64 MEMS PDM mikrofonů.

Jednotlivé fáze jsou rozepsány v kapitolách níže.



Obrázek 23 - Základní uživatelské rozhraní aplikace

5.1. Zpracování audio souboru

5.1.1. Knihovna NAudio.dll

Ke zpracování audio signálu byla použita volně dostupná open-source .NET knihovna NAudio.dll, která obsahuje širokou škálu různých možností práce se signálem. Do aplikace se přidá jednoduše – pomocí Nuget balíčku [12].

Knihovna byla využita především z důvodu předzpracovaných audio formátů, čtení a zápisu audio streamu a snímání a zpracování zvukových signálů z počítačových komponent/mikrofonů real-time za využití WASAPI.

Konkrétně byly využity metody tříd:

- WaveStream
- WaveFormat
- WaveFileWriter
- WasapiCapture
- MMDevice
- BiQuadFilter
- FastFourierTransform

5.1.2. WASAPI

WASAPI (angl. *the Windows Audio Session API*) je rozhraní umožňující klientským aplikacím řídit tok audio dat mezi aplikací a koncovým audio zařízením.

Enginem je user-mode audio komponenta, skrz kterou aplikace sdílí přístup ke koncovému audio hardwaru. Audio engine přenáší data mezi koncovým bufferem/zásobníkem a koncovým zařízením.

5.2. Vizualizace signálu

Po přijetí dat je pro následnou analýzu signálu potřeba je zobrazit. Jak již bylo řečeno výše, aplikace prošla několika stupni vývoje – to ovlivnilo i způsob zobrazení

přijatého audio streamu. Jedná se o samostatné komponenty, které se starají o správné zobrazení zpracovaného signálu.

Poté, co si uživatel v aplikaci zvolí zdroj zvukového signálu, způsob vizualizace a případně nastaví další parametry [viz. 5.3], se pošlou tyto informace z View Modelu do třídy *AudioDataCapture.cs*, která se stará o průběžné odebírání dat a ukládá je do zásobníku.

Data ze zásobníku jsou následně předzpracovány ve třídě *SampleAggregator.cs*, která v aplikaci slouží jako výpočetní jednotka, a poté jsou poslány do zvolené komponenty, která je vyobrazí vybraným způsobem.

5.2.1. Spektrální analyzátor

Prvním způsobem zobrazení je spektrální analyzátor, který zobrazuje četnost dat pro určitou frekvenci. Signál je tedy nutné nejprve nechat zpracovat pomocí Fourierovy transformace [viz. 1.3.2].

Výslednou hodnotu může uživatel ovlivnit nastavením vstupních parametrů FT – *FftLenght* (počtem vzorků) a *FftWindow* (filtrovací okénkovací funkcí).

```
public SampleAggregator(ISampleProvider source, int fftSize, int
fftWindow)
{
	fftL = fftSize;
	channels = source.WaveFormat.Channels;
	if (!IsPowerOfTwo(fftLength)) {
	throw new ArgumentException("FFT Length must be a power of two"); }
	this.m = (int)Math.Log(fftLength, 2.0);
	this.fftL = fftSize;
	this.fftBuff = new Complex [fftLength];
	this.fftArgs = new FftEventArgs (fftBuffer);
	this.fftWindow = fftWindow;
	this.source = source;
	. . .
}
public void Add(float v)
{
	try{
	if (PerformFFT && FftCalculated != null) {
		if (this.fftWindow == 0) {
		fftBuff[fftPos].X=(float)(v * FFT.Hamming (fftPos, fftL));
		} else if (this.fftWindow == 1) {
		fftBuff[fftPos].X=(float)(v * FFT.Hann(fftPos, fftL));
```

```
            } else if (this.fftWindow == 2) {
            fftBuff[fftPos].X=(float)(v * FFT.BlackmannHarris (fftPos,
                              fftL));
            }
            fftBuffer[fftPos].Y = 0;
            fftPos++;
            if (fftPos >= fftBuff.Length) {
            fftPos = 0;
            FFT.FFT(true, m, fftBuff);
            FftCalculated(this, fftArgs);}}. . .}
```

Zdrojový kód 1 – Sample aggregaror – Zpracování přijatých dat signálu pomocí FFT

Komponenta *SpectrumAnalyser* dostává již spočtené výsledné hodnoty z třídy *SampleAggregator.cs* a jejím úkolem je jejich vizualizace. Data jsou přijímána díky volání virtuální metody zvolené vizualizace využívající EventHandler, který snímá zpracovaná data.

```
delegate void UpdateDelegate(SpectrumAnalyser sa, NAudio.Dsp.Complex[]
result);
public void OnFftCalculated(NAudio.Dsp.Complex[] result)
{
      if (spectrumAnalyser.Dispatcher.CheckAccess()){
            spectrumAnalyser.Update(result);
      }else{
            Action action = () =>
            {spectrumAnalyser.Update(result);};
      spectrumAnalyser.Dispatcher.BeginInvoke(DispatcherPriority.Normal,
      action);
}}
```

Zdrojový kód 2 – Spectrum analyser visualization

Z *OnFftCalculated()* je volána metoda komponenty *SpectrumAnalyser Update(Complex[] fftResults)*, která získané pole dat (velikost pole je dána vstupním parametrem *FftLength*) dále přepočítá na odpovídající souřadnice a následně aktualizuje současné vykreslení grafu.

```
public void Update(Complex[] fftResults)
{
      if (fftResults.Length / 2 != bins){
            this.bins = fftResults.Length / 2;
            CalculateXScale();
      }
      for (int n = 0; n < fftResults.Length / 2; n += binsPerPoint){
            double yPos = 0;
            for (int b = 0; b < binsPerPoint; b++){
                  yPos += GetYPosLog(fftResults[n + b]);
            }
            AddResult(n / binsPerPoint, yPos / binsPerPoint);}}
```

Zdrojový kód 3 – Spectrum analyser - Update

```
private double GetYPosLog(Complex c)
{
	double magnitudeDb;
	//mag = sqrt(Re^2 + Im^2); magnitude [dB] = 20 * Log(mag)
	magnitudeDb = 20 * Math.Log10(Math.Sqrt(c.X * c.X + c.Y * c.Y));
	double minDB = -90;
	if (magnitudeDb < minDB) magnitudeDb = minDB;
	double percent = magnitudeDb / minDB;
	double yPos = percent * this.ActualHeight;
	return yPos;
}
```

Zdrojový kód 4 – Spectrum analyser - Získání souřadnice Y

Jednotlivé body signálu jsou znázorněny pomocí *Polyline.cs* dědící z abstraktní třídy *Shapes.cs* jež je součástí *FrameworkElement.cs*.

```
private void AddResult(int index, double power)
{
	Point p = new Point(CalculateXPos(index), power);
	if (index >= polyline1.Points.Count)
	{
		polyline1.Points.Add(p);
	}
	else
	{
		polyline1.Points[index] = p;
	}
}
private double CalculateXPos(int bin)
{
	if (bin == 0) return 0;
	return bin * xScale;
}
private void CalculateXScale()
{
	this.xScale = this.ActualWidth / binsPerPoint;
}
```

Zdrojový kód 5 – Spectrum analyser - Přidání bodů do Polyline

5.2.2. Časový průběh signálu

Druhým přidaným způsobem zobrazení průběhu signálu je jeho ztvárnění na časové ose. Na rozdíl od předchozí vizualizace není nutné přepočítávat hodnoty vzorků. Ale protože je graf vykreslován pomocí *Polygon.cs*, jež, stejně jako *Polyline.cs*, dědí z abstraktní třídy *Shapes.cs*, je nutné odchytávat maximální a minimální hodnoty.

Třída *Polygon.cs* znázorňuje data jako dvě protilehlé křivky – prostor mezi nimi je možné vyplnit zvolenou barvou.

```
maxValue = Math.Max(maxValue, value);
minValue = Math.Min(minValue, value);
count++;
if (count >= NotificationCount && NotificationCount > 0)
{
	if (MaximumCalculated != null)
	{
		MaximumCalculated(this, new MaxSampleEventArgs(minValue,
		maxValue, multY));
	}
	Reset();
}
*************************************************************************
Polygon waveForm = new Polygon();
public void AddValue(float maxValue, float minValue, float multY)
{
	this.yScale = this.ActualHeight * (multY / 10);
	int visiblePixels = (int)(ActualWidth / xScale);
	if (visiblePixels > 0)
	{
		CreatePoint(maxValue, minValue);
		if (renderPosition > visiblePixels)
		{
			renderPosition = 0;
		}
		int erasePos = (renderPosition + blankZone)
					% visiblePixels;
		if (erasePos < Points)
		{
			double yPos = SampleToYPosition(0);
			waveForm.Points[erasePos] = new Point(erasePos
							* xScale, yPos);
			veForm.Points[BottomPointIndex(erasePos)] =
			new Point(erasePos * xScale, yPos);
		}
	}
}
```

Zdrojový kód 6 – Time analyser - Vykreslení grafu časového průběhu signálu

5.2.3. Osciloskop

Osciloskop je vizualizace zobrazující časový průběh signálu, ovšem v přiblížené formě, která umožňuje studii tvaru signálu.

Stejně jako vizualizace *SpectrumAnalyser* i v komponentě *Oscilloscope* jsou pro zobrazení využity metody třídy *Polyline.cs*.

```
private Polyline polyline1 = new Polyline { Stroke = Brushes.Blue,
StrokeThickness = 2 };
public void Update(float[] results)
{
```

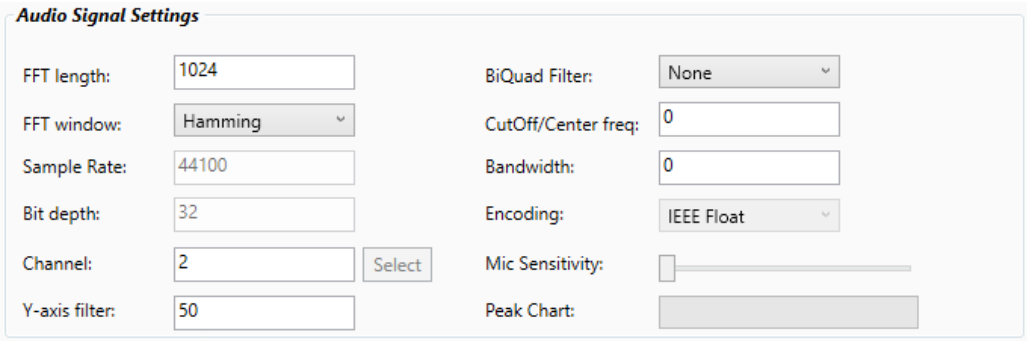
```
        WAVEDATA_SIZE = results.Length / 3;
        double yPos = this.ActualHeight / 2;
        for (int n = 0; n < results.Length; n++)
        {
              if (n < WAVEDATA_SIZE - 1) yPos += GetYPos(results[n + 1]);
              AddResult(n, yPos);
        }
}
private void AddResult(int index, double power)
{
        Point p = new Point(CalculateXPos(index), power);
        if (index >= polyline1.Points.Count)
        {
              polyline1.Points.Add(p);
        }
        else
        {
              polyline1.Points[index] = p;
        }
}
private double CalculateXPos(int x)
{
        if (x == 0) return 0;
        return x + this.ActualWidth - WAVEDATA_SIZE - 100.0f;
}
private double GetYPos(float i, float filterY)
{
        float y = i / filterY * (float)this.ActualHeight;
        return y;
}
```

Zdrojový kód 7 - Oscilloscope

5.3. Nastavení parametrů vstupních dat

Zvolením požadované vizualizace uživatelovy možnosti nekončí. Dle vybraného zdroje audio signálu se mu zpřístupní možnost úpravy parametrů ovlivňujících vstupní data. Pro všechny varianty je povolená možnost editace FFT parametrů a úprava citlivosti zobrazení dat v grafech ‘Osciloskop‘ a ‘Časový průběh signálu’, jež má funkci přiblížení či oddálení zobrazovaných dat.

Protože je pro ovlivnění citlivosti zobrazení dat používáno jen jedno vstupní textové pole, ale každý graf používá jiný algoritmus a vstupní hodnotu používá jiným způsobem, je nutné měnit jeho hodnotu dle typu grafu vizualizace dat. Například osciloskop je citlivější, čím je vstupní hodnota ‘Y-axis filter’ menší, zato vizualizace časového průběhu signálu to má přesně naopak.



Obrázek 24 - Vstupní parametry a BiQuad filtr

5.3.1. Audio soubor

V případě vizualizace signálu audio souboru jsou povoleny pouze parametry týkající se výpočtu FFT, tj. délka FFT a volba okénkovací funkce.

5.3.2. Real-Time WASAPI zpracování

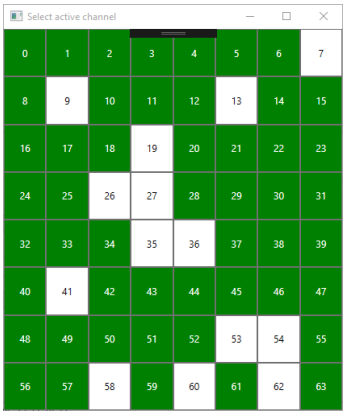
Pro real-time zpracování audio signálu ze zařízení, jež je kompatibilní s WASAPI rozhraním, se zpřístupní i ostatní parametry, poskytující uživateli především editaci vzorkovací frekvence, bitové šířky, citlivosti mikrofonu a typu kódování (PCM či IEEE Float).

5.3.3. Real-Time MEMS zpracování

Pro zbylou variantu zpracování audio signálu z pole MEMS mikrofonů se zpřístupní především možnost editace počtu kanálů.

V momentě volby komunikace s vývojovou deskou osazenou polem MEMS PDM mikrofonů čítající 64 kusů (tj. 64 kanálů) se zpřístupní tlačítko *Select*, jež po kliknutí zobrazí matici s 64 tlačítky představujícími konkrétní kanály. Barva tlačítek představuje jejich stav:

- zelené – aktivní
- šedé – neaktivní



Obrázek 25 - Volba aktivních kanálů pole MEMS PDM mikrofonů

Po kliknutí na zmíněné tlačítko *Select* se odešle na serverovou část desky příkaz, jež nastaví všechny kanály do stavu ‘ a restartuje tím tak předchozí nastavení. Poté se tlačítkům nastaví odpovídající barva, tedy všechna budou zelená.

Po kliknutí na jednotlivá tlačítka představující jeden kus MEMS PDM mikrofonu, se dle jeho aktuálního stavu odešle příkaz na jeho změnu, tj. aktivní deaktivovat a neaktivní aktivovat.

```
void button_Click(object sender, RoutedEventArgs e)
{
	cmds = ClientCommandsNResponses.Ccommands;
	if ((sender as Button).Background == Brushes.White)
	{
		Console.WriteLine(string.Format("{0}. btn activated",
		(sender as Button).Tag));
		var btnTag = (sender as Button).Tag;
		Command cmd =
		cmds.CreateCommand(command_classes_enum.MICSRV_CLASS_SET,
		command_targets_enum.MICSRV_TARGET_PDMMIC,
		(int)commands_pdmmic_enum.MICSRV_COMMAND_PDMMIC_CHANNEL_ENABLE,
		(int)btnTag, 1, callback_test);
		if (cmd != null)
		{
			AC.EnqueueCommand(cmd);
		}
		(sender as Button).Background = Brushes.Green;
		(sender as Button).Foreground = Brushes.White;
	}
	else if ((sender as Button).Background == Brushes.Green)
	{
		Console.WriteLine(string.Format("{0}. btn disactivated",
		(sender as Button).Tag));
		var btnTag = (sender as Button).Tag;
		Command cmd =
		cmds.CreateCommand(command_classes_enum.MICSRV_CLASS_SET,
```

```
            command_targets_enum.MICSRV_TARGET_PDMMIC,
            (int)commands_pdmmic_enum
            .MICSRV_COMMAND_PDMMIC_CHANNEL_ENABLE, (int)btnTag, 0,
            callback_test);
            if (cmd != null)
            {
                  AC.EnqueueCommand(cmd);
            }
            (sender as Button).Background = Brushes.White;
            (sender as Button).Foreground = Brushes.Black;
      }
}
```

Zdrojový kód 8 - Nastavení stavu MEMS PDM mikrofonů na vývojové desce

Pro nastavení povolených kanálů lze zároveň použít textové vstupní pole. Po zadání hodnoty v rozmezí 1 – 64 kanálů bude povolen odpovídající počet prvních *n* kanálů.

5.4. Filtrování signálu

Příchozí data mohou být filtrována lineárním filtrem BiQuad [viz. 1.6.3]. Uživateli jsou nabídnuty různé varianty:

- dolní propust (angl. *Low Pass*)
- horní propust (angl. *High Pass*)
- pásmová propust (angl. *Pass Band*)
- pásmová zádrž (angl. *Notch*)
- All-Pass propust

Jednou z možností nabízených v combo-boxu je „NONE“, tj. žádná. v tom případě se filtr neaplikuje.

Obrázek 26 - Volba BiQuad filtru

Pokud se však uživatel rozhodne přijímaná data filtrovat, musí nastavit i příslušné vstupní parametry filtru – mezní či střední frekvenci a šířku pásma.

Dle nastavení vybraného filtru a vzorkovací frekvence signálu se vypočítají koeficienty sloužící k výpočtu hodnot filtrovaného signálu. Zvolený filtr je identifikován pomocí indexu připadajícímu k hodnotě prvku combo-boxu.

```
public BiQuadFilter bqFilter;
. . .
private void BiQuadFilterSignal(int sampleRate, int biQuadFilterIndex, int
biQuadFreq, int biQuadBandwidth)
{
	switch (biQuadFilterIndex)
	{
		case 0:
			bqFilter = null;
			break;
		case 1:
			bqFilter = BiQuadFilter.LowPassFilter(sampleRate,
				biQuadFreq, biQuadBandwidth);
			break;
		case 2:
			bqFilter = BiQuadFilter.HighPassFilter(sampleRate,
				biQuadFreq, biQuadBandwidth);
			break;
		case 3:
			bqFilter = BiQuadFilter
				.BandPassFilterConstantPeakGain(sampleRate,
				biQuadFreq, biQuadBandwidth);
			break;
		case 4:
			bqFilter = BiQuadFilter
				.BandPassFilterConstantSkirtGain(sampleRate,
				biQuadFreq, biQuadBandwidth);
			break;
		case 5:
			bqFilter = BiQuadFilter.NotchFilter(sampleRate,
				biQuadFreq, biQuadBandwidth);
			break;
		case 6:
			bqFilter = BiQuadFilter.AllPassFilter(sampleRate,
				biQuadFreq, biQuadBandwidth);
			break;
		default:
			bqFilter = null;
			break;
	}
}
. . .
if (bqFilter != null)
{
	sample32 = bqFilter.Transform(sample32);
}
sampleAggregator[i].Add(sample32);
```

Zdrojový kód 9 - Nastavení BiQuad filtru

K aplikaci filtru na příchozí audio signál je použita metoda *Transform()*, která hodnotu vzorku datového typu *float* přepočítá za použití koeficientů filtru.

```
public float Transform(float inSample)
{
	// compute result
	var result = a0 * inSample + a1 * x1 + a2 * x2 - a3 * y1 - a4 * y2;
	// shift x1 to x2, sample to x1
	x2 = x1;
	x1 = inSample;
	// shift y1 to y2, result to y1
	y2 = y1;
	y1 = (float)result;
	return y1;
}
```

Zdrojový kód 10 - Transformace vzorku na filtrovanou hodnotu vzorku

5.4.1. Dolní propust

Dolní propust je označení lineárního filtru, jež nepropouští signál vyšších frekvencí. Omezení je definováno mezní frekvencí. Využívá se zejména v audio technice k filtraci basových tónů o nízké frekvenci.

5.4.2. Horní propust

Horní propust je frekvenční lineární filtr, který na rozdíl od dolní propusti nepropouští signál nižších frekvencí, než je dáno mezní frekvencí.

```
public static BiQuadFilter HighPassFilter(float sampleRate, float
cutoffFrequency, float q)
{
	var filter = new BiQuadFilter();
	filter.SetHighPassFilter(sampleRate, cutoffFrequency, q);
	return filter;
}
public void SetHighPassFilter(float sampleRate, float cutoffFrequency,
float q)
{
	// H(s) = s^2 / (s^2 + s/Q + 1)
	var w0 = 2 * Math.PI * cutoffFrequency / sampleRate;
	var cosw0 = Math.Cos(w0);
	var alpha = Math.Sin(w0) / (2 * q);
	var b0 = (1 + cosw0) / 2;
	var b1 = -(1 + cosw0);
	var b2 = (1 + cosw0) / 2;
	var aa0 = 1 + alpha;
	var aa1 = -2 * cosw0;
```

```
        var aa2 = 1 - alpha;
        SetCoefficients(aa0, aa1, aa2, b0, b1, b2);
}
private void SetCoefficients(double aa0, double aa1, double aa2, double
b0, double b1, double b2)
{
        a0 = b0 / aa0;
        a1 = b1 / aa0;
        a2 = b2 / aa0;
        a3 = aa1 / aa0;
        a4 = aa2 / aa0;
}
```

Zdrojový kód 11 - Výpočet horní propusti [12]

5.4.3. Pásmová propust

Pásmová propust je lineární filtr, který, jak z názvu vyplývá, propouští signál jen určitého frekvenčního pásma. Hranice mohou být určeny buď zadáním spodní a horní mezní frekvence nebo střední frekvencí a šířkou pásma. v aplikaci je využita druhá ze zmíněných variant.

```
public     static     BiQuadFilter     BandPassFilterConstantSkirtGain(float
sampleRate, float centreFrequency, float q)
{
        // H(s) = s / (s^2 + s/Q + 1)  (constant skirt gain, peak gain = Q)
        var w0 = 2 * Math.PI * centreFrequency / sampleRate;
        var cosw0 = Math.Cos(w0);
        var sinw0 = Math.Sin(w0);
        var alpha = sinw0 / (2 * q);
        var b0 = sinw0 / 2; // =   Q*alpha
        var b1 = 0;
        var b2 = -sinw0 / 2; // =  -Q*alpha
        var a0 = 1 + alpha;
        var a1 = -2 * cosw0;
        var a2 = 1 - alpha;
        return new BiQuadFilter(a0, a1, a2, b0, b1, b2);
}
private BiQuadFilter(double a0, double a1, double a2, double b0, double
b1, double b2)
{
        SetCoefficients(a0, a1, a2, b0, b1, b2);
        x1 = x2 = 0;
        y1 = y2 = 0;
}
```

Zdrojový kód 12 - Výpočet pásmové propusti [12]

5.5. Multi-channel

Zvolený způsob vizualizace je stejný pro veškeré snímané kanály. Po kliknutí na tlačítko k jejímu spuštění (*„Play“* či *„Record“*) se zobrazí okno obsahující jednotlivé grafy – data každého kanálu jsou zobrazována odděleně. Aby byl tento způsob vykreslení efektivní, je používán multithreading, neboli vícevláknové zpracování dat.

```
private void VisualizeMEMS()
{
	for (int i = 0; i < channel_count - 1; i++)
	{
		Task.Factory.StartNew(() => computeVisualizations[i]
					.OnDataAvailableMems(strm_parser
					.Dequeue(i, 1024)),
					CancellationToken.None,
					TaskCreationOptions.None,
					TaskScheduler.Default);
	}
}
```

Zdrojový kód 13 - Vícevláknové zpracování a zobrazení dat MEMS PDM mikrofonů pomocí Task.Factory.StartNew()

K vytvoření jednotlivých vláken a přiřazení jim odpovídajících akcí byly nejprve použity metody třídy *Task.cs* dědící ze systémové třídy *Thread.cs*.

Později však došlo ke změně na metody třídy *Parallel.cs*, která také, stejně jako *Task.cs*, spadá do jmenného prostoru *System.Threading.Tasks*. Volání metod této druhé třídy však zjednodušuje vytvoření paralelně běžících akcí v opakujícím se cyklu – uživateli jsou přímo nabídnuty metody *For* či *Foreach*.

```
private void VisualizeMEMS()
{
	MemsChannelCount = ChannelCount;
	Parallel.For(0, MemsChannelCount, i =>
	{
		computeVisualizations[i].OnDataAvailableMems(strm_parser.
		 Dequeue(i, 1024), BiQuadFilterIndex, BiQuadFrequency,
		 BiQuadBandwidth, NegativeZoom);
	});
}
```

Zdrojový kód 14 - Vícevláknové zpracování a zobrazení dat MEMS PDM mikrofonů pomocí Parallel.For()

5.6. Zpracování signálu audio souboru

Jak již bylo výše zmíněno, aplikace prošla několika stupni vývoje. Prvním z nich je zobrazení signálu zvukové nahrávky, k čemuž byly využity metody knihovny *NAudio.dll* [viz. 5.1.1].

Po zvolení zdroje signálu kliknutím na odpovídající radio-button se zpřístupní tlačítko umožňující výběr audio souboru. Data nahrávky i s parametry Fourierovy transformace jsou okamžitě poté předány ke zpracování. Pokud se ještě před spuštěním vizualizace změní nastavení parametrů, dojde k opětovnému odeslání dat ke zpracování.

```
private void OpenFile(string fileName, int fft, int fftWindow)
{
	try
	{
		var inputStream = new AudioFileReader(fileName);
		fileStream = inputStream;
		var aggregator = new SampleAggregator(inputStream, fft,
				fftWindow);
		. . .
	}
	catch (Exception e)
	{
		MessageBox.Show(e.Message, "Problem opening file");
		CloseFile();
	}
}
. . .
public class AudioFileReader : WaveStream, ISampleProvider
{
	. . .
	public AudioFileReader(string fileName)
	{
		lockObject = new object();
		this.fileName = fileName;
		CreateReaderStream(fileName);
		sourceBytesPerSample =
		(readerStream.WaveFormat.BitsPerSample/8)
		* readerStream.WaveFormat.Channels;
		sampleChannel = new SampleChannel(readerStream, false);
		destBytesPerSample = 4 * sampleChannel.WaveFormat.Channels;
		length = SourceToDest(readerStream.Length);
	}
	. . .
}
```

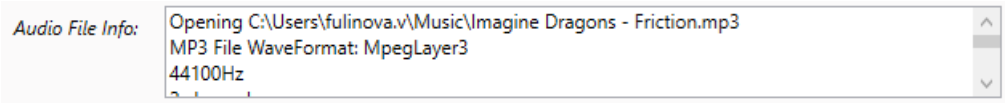
Zdrojový kód 15 - Získání dat zvukové nahrávky

Data popisující signál jsou získána pomocí metod třídy *AudioFileReader.cs* z knihovny *NAudio.dll*, kde jsou jednotlivé informace roztříděny do příslušných proměnných. Dále je audio stream předán metodám třídy *SampleAggregator* k výpočtům hodnot jednotlivých vzorků a jejich následné zobrazení.

Vykreslování vizualizací je spuštěno tlačítkem s ikonou symbolizující přehrávání, tj. ▶.

5.6.1. Výpis vlastností audio souboru

Aplikace nabízí výpis základních vlastností vloženého souboru využitím rozhraní *IAudioFileDescription*. Získané vlastnosti jsou promítnuty do textového pole umístěného mezi volbou zdroje audio signálu a blokem s možnostmi nastavení vstupních parametrů.



Obrázek 27 - Výpis vlastností audio souboru

5.7. Zpracování real-time signálu pomocí WASAPI rozhraní

Druhým stupněm vývoje aplikace bylo zpracování audio signálu v reálném čase za využití rozhraní WASAPI [viz. 5.1.2]. Po kliknutí na odpovídající radio-button se zpřístupní combo-box, který obsahuje list hardwaru připojeného k počítači, jež systém Windows rozpoznal jako audio HW.

```
public IEnumerable<MMDevice> CaptureDev { get; private set; }
var enums = new MMDeviceEnumerator();
CaptureDev = enums.EnumerateAudioEndPoints(DataFlow.Capture,
             DeviceState.Active).ToArray();
```

Zdrojový kód 16 - List audio hardwaru

Jelikož je zvuk snímán v reálném čase, není do spuštění vizualizace nic předpočítáváno. Uživatel má, na rozdíl varianty audio souboru, možnost nastavit kromě parametrů ovlivňujících výpočet FFT i vzorkovací frekvenci, bitovou šířku a formát vzorků signálu (PCM či IEEE Float). Ke každému kanálu se zobrazí jeho vlastní vizualizace.

U snímání zvuku jsou uživateli nabídnuty dvě varianty spuštění vizualizace, tj. pouhé zobrazení signálu tlačítkem *"Play"* nebo i jeho nahrávání ve formátu WAV stisknutím tlačítka *"Record"*. Seznam nahraných souborů je vyobrazen ve spodní části aplikace.

Jak již bylo napsáno v úvodu této kapitoly, pro zachytávání signálu z audio HW v reálném čase je využito rozhraní WASAPI, jež je v knihovně *NAudio.dll* definováno třídou *WasapiCapture.cs*. Díky metodám této třídy lze také určit formát vzorků signálu dle nastavení příslušného vstupního parametru.

```
capture.WaveFormat = SampleTypeIndex == 0 ?
                     WaveFormat.CreateIeeeFloatWaveFormat(sampleRate,
                     channelCount) :
                     new WaveFormat(sampleRate, bitDepth, channelCount);
```

Zdrojový kód 17 - Nastavení formátu vzorků signálu (IEEE Float či PCM)

Po spuštění zvolené vizualizace se pomocí handleru *OnDataAvailable(object sender, WaveInEventArgs e)* odchytávají přijímaná data a posílají ke zpracování do *SampleAggregator*.

5.8. Zpracování dat z pole MEMS PDM mikrofonů

5.8.1. Server firmware

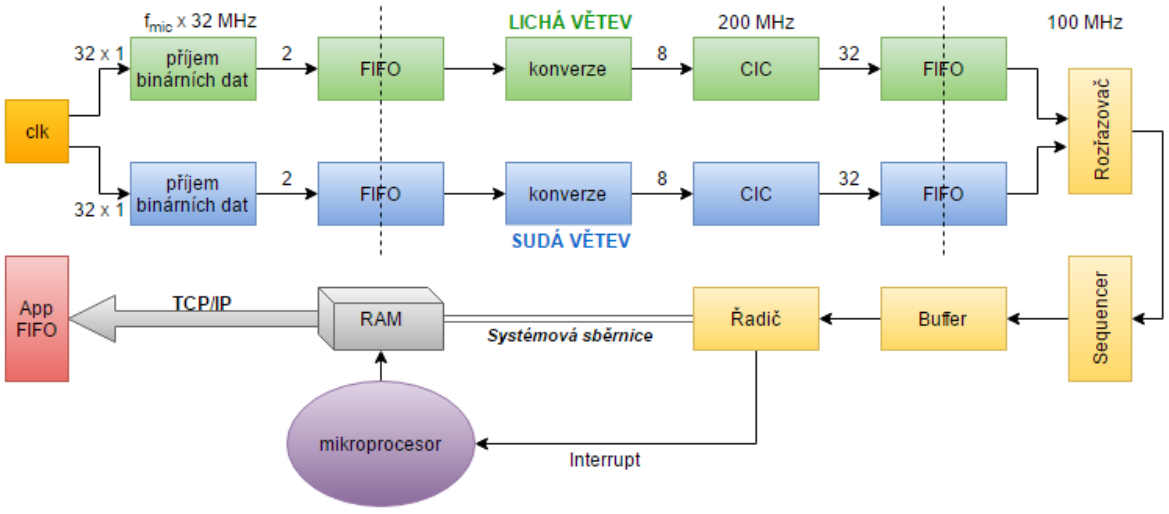
Data jsou z vývojové desky osazené polem MEMS PDM mikrofonů snímána po dvou mikrofonech (kanálech). Využívá se zde zapojení mikrofonů jako stereo mikrofonů. Odebírání dat se střídá na základě hodinového signálu – liché mikrofony jsou snímány na náběžné hraně signálu a sudé na hraně sestupné. Tímto způsobem se sníží potřebná režie na polovinu.

Protože však každé odesílání a přijímání dat pracuje na jiné frekvenci/rychlosti, jsou mezi těmito bloky používány vyrovnávací paměti FIFO.

Po přijetí signálu z mikrofonu se data přefiltrují pomocí programovatelného CIC filtru. Do filtru vstupuje 8 bitový údaj, využívají se z něj však pouze 2 bity. Výstupem je spodních 32 bitů z 62 bitového údaje.

Poté se data ukládají opět do FIFO, odkud jsou posílány do rozřazovače, který střídá lichou a sudou větev. Na základě povolených kanálů data buď zapíše do sequenceru, anebo jen přečte, ale nezapíše.

Údaje jsou následně odesílány po systémové sběrnici do paměti RAM po blocích o velikosti 1MB, odkud jsou přenášeny pomocí protokolu TCP/IP do FIFO paměti klientské strany aplikace.



Obrázek 28 – HW schéma příjmu PDM signálů

5.8.2. Komunikace s vývojovou deskou

S vývojovou deskou se komunikuje pomocí protokolu TCP/IP [viz. 4.3]. Počítač, na kterém běží aplikace, musí aktivovat spojení s deskou. Ta je k zařízení připojena ethernetovým a USB kabelem. Spojení se navazuje pomocí COM portu.

Poté, co je deska osazená polem 64 MEMS PDM mikrofonů určena jako zdroj snímání audio signálu, zpřístupní se tlačítko *"Connect"* pro propojení serverové části s klientskou.

Kliknutím na tlačítko se aktivuje asynchronní klient, který, jak z názvu vyplývá, umožňuje asynchronní komunikaci mezi aplikací a deskou. Na základě IP adresy a portu se vytvoří požadované spojení.

```
public void StartClient(IPAddress RipAddress, int Rport)
{
      // Connect to a remote device.
      try
      {
            // Establish the remote endpoint for the socket.
            IPEndPoint remoteEP = new IPEndPoint(RipAddress, Rport);
            // Create a TCP/IP socket.
            client = new Socket(AddressFamily.InterNetwork,
                  SocketType.Stream, ProtocolType.Tcp);
            // Connect to the remote endpoint.
            client.BeginConnect(remoteEP,
                  new AsyncCallback(connect_callback), client);
```

```
                    connect_done.WaitOne();
                . . .
                current_state = AsynchronoutClientFSMStates.ACFSM_Started;
                OnConnect();
        }
        catch (Exception e)
        {
                Console.WriteLine(e.ToString());
        }
}
```

Zdrojový kód 18 - TCP/IP propojení serverové a klientské části

Stejným způsobem, avšak s jiným portem, se aktivuje i klient určený ke snímání toku dat.

Zmíněný asynchronní klient má na starosti obsluhu příkazů. Jeho inicializací se vytvoří fronta na přijímané úkony a fronta určená na jejich odpovědi ze serveru (tj. z vývojové desky). Aby příkazy bylo možné vykonávat, vytvoří se zároveň instance Singlton třídy *ClientCommandsNResponses.cs* [viz. 5.8.3].

Asynchronní zpracování příkazů funguje na principu stavového automatu – kontroluje tedy stav přijímaných úkolů a jejich odpovědi (zda přenos byl v pořádku či nikoliv) a dle stanovených přechodových funkcí [viz. A.4] je vykoná či pokračuje na vyřízení následujícího příkazu, nebo zachytí chybu přenosu či výkonu daného příkazu.

5.8.3. Vytváření příkazů

Po úspěšném vytvoření spojení mezi serverovou a klientskou částí, je aplikace připravena na nastavení vstupních parametrů [viz. 5.3]. v případě komunikace s vývojovou deskou se pro každý požadavek musí vytvořit příkaz, jehož zpracování probíhá asynchronně.

Ke složení příkazu v korektním tvaru slouží Singlton třída *ClientCommandsNResponses.cs*. Potřebný příkaz je vytvořen pomocí výčtu jejích přetěžovaných funkcí *CreateCommand()*. Dle vložených parametrů funkce je rozeznána požadovaná akce. Po nalezení shody funkce vrátí odpověď typu *Command*.

```
cmds = ClientCommandsNResponses.Ccommands;
Command cmd = cmds.CreateCommand(command_classes_enum.MICSRV_CLASS_GET,
              command_targets_enum.MICSRV_TARGET_PDMMIC,
              (int)commands_pdmmic_enum
              .MICSRV_COMMAND_PDMMIC_CHANNEL_ENABLE, i,
```

```
                callback_channelCheck);
if (cmd != null)
{
        ac.EnqueueCommand(cmd);
}
```

Zdrojový kód 19 - Příklad vytvoření příkazu zjišťující stav mikrofonu

Jednotlivé příkazy jsou ukládány do fronty a postupně odesílány ke zpracování serverem. Na každý dotaz z desky odejde i odpověď, která je taktéž ukládána do fronty určené k dalšímu zpracování.

Jednotlivými příkazy se koriguje celkové chování vývojové desky rozšířené o mikrofonové pole – ať už se jedná o počáteční navázání komunikace, přijímání dat či úpravu zpracování přijímaných dat. Po odeslání dat serverové části je očekávána odpověď – zpracování příkazu buď proběhlo v pořádku, nebo nastala nějaká chyba, jejíž konkrétní znění je dle typu odesláno klientské straně.

Konkrétní podoba ovládacích příkazů je definována řídícím protokolem, který jednotlivé úkony překládá apomocí něhož je s deskou komunikováno.

```
// command classes
classes_dict.Add(command_classes_enum.MICSRV_CLASS_GET, "GET");
classes_dict.Add(command_classes_enum.MICSRV_CLASS_SET, "SET");
classes_dict.Add(command_classes_enum.MICSRV_CLASS_OFF, "OFF");



// command targets
targets_dict.Add(command_targets_enum.MICSRV_TARGET_PDMMIC, "PDMMIC");
targets_dict.Add(command_targets_enum.MICSRV_TARGET_DECIMATOR, "DECIMATOR");
targets_dict.Add(command_targets_enum.MICSRV_TARGET_CLOCK, "CLOCK");
targets_dict.Add(command_targets_enum.MICSRV_TARGET_STREAMER, "STREAMER");
targets_dict.Add(command_targets_enum.MICSRV_TARGET_SYSTEMCFG, "SYSTEMCFG");



// pdmmic commands
pdmmic_commands_dict.Add(commands_pdmmic_enum.MICSRV_COMMAND_PDMMIC_ENABLE,
new command_string_and_behaviour("ENABLE"));
pdmmic_commands_dict.Add(commands_pdmmic_enum.MICSRV_COMMAND_PDMMIC_CHANNEL_E
NABLE, new command_string_and_behaviour("CHANENA"));
pdmmic_commands_dict.Add(commands_pdmmic_enum.MICSRV_COMMAND_PDMMIC_CAPTURE_P
OINT, new command_string_and_behaviour("CAPTUREP"));



// decimator
decimator_commands_dict.Add(commands_decimator_enum.MICSRV_COMMAND_DECIMATOR_
RATE, new command_string_and_behaviour("RATE"));
```

```
// clock
clock_commands_dict.Add(commands_clock_enum.MICSRV_COMMAND_CLOCK_INPUT_FREQ,
new command_string_and_behaviour("INPUTFREQ"));
clock_commands_dict.Add(commands_clock_enum.MICSRV_COMMAND_CLOCK_VCO_NUM, new
command_string_and_behaviour("VCONUM"));
clock_commands_dict.Add(commands_clock_enum.MICSRV_COMMAND_CLOCK_VCO_DEN, new
command_string_and_behaviour("VCODEN"));
clock_commands_dict.Add(commands_clock_enum.MICSRV_COMMAND_CLOCK_OUT_DIV, new
command_string_and_behaviour("VCODIV"));
clock_commands_dict.Add(commands_clock_enum.MICSRV_COMMAND_CLOCK_APPLY, new
command_string_and_behaviour("APPLY"));
clock_commands_dict.Add(commands_clock_enum.MICSRV_COMMAND_CLOCK_LOCKED, new
command_string_and_behaviour("LOCKED"));

// steramer
streamer_commands_dict.Add(commands_streamer_enum.MICSRV_COMMAND_STREAMER_ENA
BLE, new command_string_and_behaviour("ENABLE"));

// system
system_commands_dict.Add(commands_system_enum.MICSRV_COMMAND_SYSTEM_INIT_CONF
IG, new command_string_and_behaviour("INITCFG",
commands_behaviour_enum.MICSRV_COMPLEX_BEHAVIOUR));
system_commands_dict.Add(commands_system_enum.MICSRV_COMMAND_SYSTEM_INIT_IPCO
NFIG, new command_string_and_behaviour("IPCONFIG",
commands_behaviour_enum.MICSRV_COMPLEX_BEHAVIOUR));
system_commands_dict.Add(commands_system_enum.MICSRV_COMMAND_SYSTEM_FIRMWARE,
new command_string_and_behaviour("FIRMWARE",
commands_behaviour_enum.MICSRV_COMPLEX_BEHAVIOUR));
system_commands_dict.Add(commands_system_enum.MICSRV_COMMAND_SYSTEM_VERBOSE,
new command_string_and_behaviour("VERBOSE"));

// responses
response_dict.Add("OK", response_enum.MICSRV_RESPONSE_OK);
response_dict.Add("ER", response_enum.MICSRV_RESPONSE_ERROR);

// error messages
error_codes_dict.Add(error_codes.SYS_ERROR_PDMMIC_CLOCK_FAIL, "PDMMIC CLOCK
general error - NULL pointer, address, etc..");
error_codes_dict.Add(error_codes.SYS_ERROR_PDMMIC_CLOCK_INVALID_FREQUENCY,
"PDMMIC CLOCK invalid frequency setting");
error_codes_dict.Add(error_codes.SYS_ERROR_TCP_OUT_OF_MEMORY, "TCP stack out
of memory");
```

Zdrojový kód 20 - Vytvořen slovník se základními příkazy k ovládání mikrofonového pole

Každý vytvořený příkaz obsahuje callback vracející odpověď serveru na dotaz.

```
private int callback (Response rsp)
{
	if (rsp.Success)
	{
		Console.WriteLine("Command successful - {0}",
		rsp.ResponseParam);
	}else {
		Console.WriteLine("Command failed");
	}
	return 0;
}
```

Zdrojový kód 21 - Příklad callbacku vracející odpověď serveru na příkaz

Veškerá komunikace probíhá síťovým přenosem. Její stav je sledován pomocí tzv. *EventHandlerů*, jež slouží k odchytávání chyb přenosu či jeho dokončení.

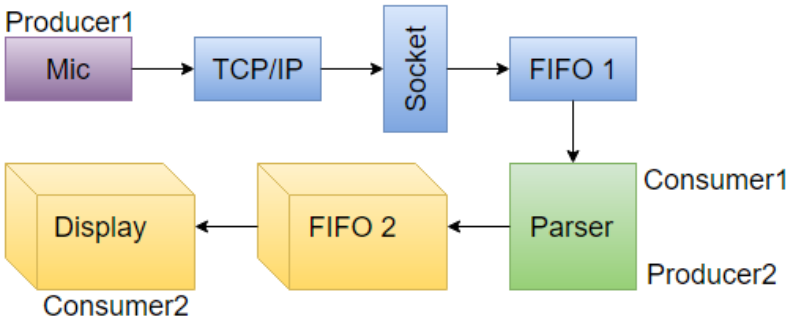
5.8.4. Zpracování audio signálu

Pro spuštění snímání dat z pole mikrofonu má uživatel dvě možnosti – pouhou vizualizaci streamů či jejich nahrávání do souboru formátu WAV. Po nastavení všech požadovaných parametrů ovlivňujících hodnotu dat signálu a tvar jeho vizualizace může uživatel aktivovat přenos dat stisknutím jednoho ze dvou tlačítek – *"Play"* pro prosté zobrazení, nebo *"Record"* pro vizualizaci a nahrávání do souboru audio formátu WAV.

Přenos dat započne po odeslání příkazu na stranu serveru, jež aktivuje snímání audio signálu z povolených mikrofonů. Komunikace mezi vývojovou deskou a aplikací je řešena způsobem *"Producer-Consumer"*. Tento typ je typický pro programování souběžných procesů [17][18]. Funguje na principu:

- *Producer* odesílá data do zásobníku
- *Consumer* odebírá data ze zásobníku

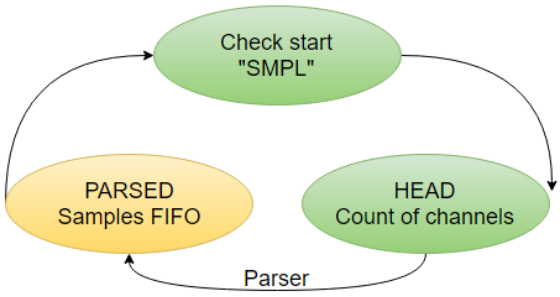
Obvykle má zásobník omezenou velikost. Pokud se zásobník naplní, *Producer* již nemá kam ukládat data a uspí se. Pokud je naopak zásobník prázdný, uspí se *Consumer*. Problém nastává, pokud se uspí oba - může tak nastat nebezpečí uváznutí (tzv. deadlock). Jedním z řešení tohoto problému je tzv. semafor, který hlídá stav zásobníku a řídí chování *Producera* a *Consumera*.



Obrázek 29 - Zpracování audio dat snímaných polem mikrofonů

Použití zmíněného způsobu je vhodné především v případech, kdy se liší rychlosti přenosu na straně *Producera* a *Consumera*.

Data jsou nejprve přijímána v kuse a poté rozdělována dle počtu kanálů. Rozparsovaná data se uloží do fronty, odkud jsou odebírána ke zpracování a následné vizualizaci.



Obrázek 30 - Rozdělení přijatých dat dle počtu kanálů

Na začátku každého streamu je klíčové slovo "SMPL" (ASCII kód: 0x534D504C) k lepšímu ohraničení nového celku dat. Následuje hlavička o velikosti 64 jedniček, tj. sizeof(UInt64) (8 x 11111111B; 8 x 255). Každá jednička reprezentuje aktivní mikrofon, nula naopak neaktivní. Dle stavu mikrofonů se stream rozdělí na jednotlivé vzorky o velikosti sizeof(Int32), které se ukládají do kolekce typu *ConcurrentQueue*.

```
private ConcurrentQueue<Int32>[] output_queues;
. . .
output_queues = new ConcurrentQueue<int>[output_queue_cnt];
for (int i = 0; i<output_queue_cnt; i++)
{
      output_queues[i] = new ConcurrentQueue<int>();
}
. . .
output_queues[output_queue_counter].Enqueue(BitConverter.ToInt32(b_smpl,
0));
```

Zdrojový kód 22 - Ukládání audio streamů jednotlivých mikrofonů

Vyparsovaná data jsou poté odebírána a předávána metodám *SampleAggregator.cs* ke zpracování [viz. 5.5] a následnému vyobrazení dle zvolené vizualizace.

Po stisknutí tlačítka „Stop" se ukončí odebírání dat z mikrofonů a nasbíraná data se uloží do „.wav" audio souboru. Vytvoří se jak 64 kanálový soubor (nebo méně – dle počtu povolených kanálů) a zároveň stejný počet jedno-kanálových audio souborů. Tento způsob

vytváření nahrávek je z důvodu umožnění pozdější vizualizace nahrávky, protože v současné době není podporováno přehrání 64 kanálového „.wav“ audio souboru.

Původně bylo zamýšleno ukládat data průběžně, jako tomu je u ukládání dat ze stereo mikrofonu zabudovaného do počítače. Ale protože to bylo velice nákladné na režii již tak velmi náročné operace jejich vizualizace, bylo od toho upuštěno.

Závěr

V rámci práce byl úspěšně vyvinut program, který vykresluje průběh signálu ze třech různých zdrojů – tj. audio soubor, WASAPI a MicroZed™ vývojová deska osazená polem MEMS PDM mikrofonů o velikosti 64 kusů. Aplikace nabízí tři varianty vizualizace, z nichž jedna zobrazuje frekvenční průběh a zbylé dvě průběh časový. Uživatel má možnost měnit vstupní parametry signálu, které mají vliv na výsledné zobrazení zpracovaných dat.

Všechny body zadání se mi s pomocí mého vedoucího diplomové práce podařily splnit a jsou v práci obsaženy.

Vývoj desktopové GUI aplikace prošel několika fázemi, během nichž jsem se naučila pracovat s .NET frameworkem WPF a lépe porozuměla problematice zpracování přijímaných dat audio signálu – od statických dat audio souboru po data přijímaná v reálném čase.

Pro zpracování WAVE audio signálu a jeho zobrazení bylo využito metod knihovny NAudio.dll. Bohužel má omezenou podporu audio formátů. Není tak možné číst 64 kanálový *„.wav“* audio soubor. Audiosignál je z tohoto důvodu nahráván jak vcelku, tak rozdělen do jednotlivých jedno-kanálových souborů.

Celá aplikace je provázána několika procesy běžícími na vlastním vlákně – snímání dat, parsování datového toku z vývojové desky na jednotlivé kanály... Z procesorového pohledu je nejnáročnější vícevláknové zpracování a vizualizace audio signálu, kdy se pro každý kanál vytváří vlastní vlákno s vlastní vizualizací.

Cílem této práce bylo vytvořit klientskou část aplikace, jež by pomocí protokolu TCP/IP komunikovala s vývojovou deskou a umožnila vizualizaci signálů z povolených kanálů mikrofonového pole. To, jaké kanály jsou povolené, se nastaví v sekci vstupních parametrů. Uživatel definuje stav konkrétních mikrofonů pomocí kliknutí na tlačítko v matici o velikosti 8×8, která představuje dané mikrofonové pole na desce či zadá počet povolených kanálů – v tom případě je povoleno prvních *n* kanálů.

Schéma zapojení mikrofonového pole a firmware vývojové desky byli vytvořeny mým vedoucím diplomové práce, který mi byl velkou oporou během celého vývoje. Společně jsme došli k úspěšnému propojení serverové a klientské strany.

Použitá literatura

[1] McClellan J.H., Schaefer R., Yoder M.A.: DSP First - A Multimedia Approach. Prentice Hall, 1998, ISBN: 978-0132431712

[2] WIRSUM, Siegfried. Abeceda NF techniky. Praha: BEN - technická literatura, 1998. ISBN 80-86056-26-0.

[3] FRERKING, Marvin E. Digital signal processing in communication systems. New York: Van Nostrand Reinhold, c1994. ISBN 0442016166.

[4] KWENTUS, A.Y., ZHONGNONG JIANG a A.N. WILLSON. Application of filter sharpening to cascaded integrator-comb decimation filters. IEEE Transactions on Signal Processing [online]. 45(2), 457-467 [cit. 2016-08-30]. DOI: 10.1109/78.554309. ISSN 1053587x. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=554309

[5] HOGENAUER, E. An economical class of digital filters for decimation and interpolation. IEEE Transactions on Acoustics, Speech, and Signal Processing [online]. 1981, 29(2), 155-162 [cit. 2016-08-30]. DOI: 10.1109/TASSP.1981.1163535. ISSN 0096-3518. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1163535

[6] Charles Petzold, Mistrovství ve WPF, Computer Press, EAN: 978-8025121412

[7] GAMMA, Erich. Design patterns elements of reusable object-oriented software. Reading: Addison-Wesley, c1995. Addison-Wesley professional computing series. ISBN 0-201-63361-2.

[8] SMETANA, Ctirad. Hluk a vibrace: měření a hodnocení. Praha: Sdělovací technika, 1998. ISBN 80-901936-2-5.

[9] THOMPSON, Daniel M. Understanding audio: getting the most out of your project or professional recording studio. Boston, Mass.: Berklee Press, c2005. ISBN 0634009591.

[10] POHLMANN, Ken C. Principles of digital audio. 2nd ed. Indianapolis, Ind., USA: H.W. Sams, 1989. Audio library. ISBN 0672226340.

[11] KITE, Thomas. Understanding PDM Digital Audio [online]. Copyright © 2012 Audio Precision, 6-7 [cit. 2016-07-18]. DOI: 800-231-7350. Dostupné z: http://users.ece.utexas.edu/~bevans/courses/rtdsp/lectures/10_Data_Conversion/AP_Understanding_PDM_Digital_Audio.pdf

[12] NAudio [online]. © 2006-2016 Microsoft [cit. 2016-04-12]. Dostupné z: https://naudio.codeplex.com/

[13] KARAS, Ondřej. Realizace filtrů FIR a IIR v programovacím jazyce C#. In: Programujte.com [online]. 2010 [cit. 2016-08-02]. Dostupné z: http://programujte.com/clanek/2010050400-realizace-filtru-fir-a-iir-v-programovacim-jazyce-c/

[14] VÁVRA, Jan. WPF pro začátečníky: Úvod do WPF a vytvoření projektu. Programujte.com [online]. 2014 [cit. 2016-04-11]. Dostupné z: http://programujte.com/clanek/2014051701-wpf-pro-zacatecniky-1-dil-uvod-do-wpf-a-vytvoreni-projektu/

[15] BRISTOW-JOHNSON, Robert. Cookbook formular for audio EQ biquad filter coefficients. In: Music DSP [online]. 2012 [cit. 2016-08-02]. Dostupné z: http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt

[16] TCP/IP. Katedra technické a informační výchovy: Pedagogická fakulta Masarykovy univerzity [online]. [cit. 2016-05-22]. Dostupné z: http://www.ped.muni.cz/wtech/03_studium/teps/teps-03.pdf

[17] HORDĚJČUK, Vojtěch. Úloha Producer-Consumer (producent-konzument) [online]. © 2008 - 2016 Vojtěch Hordějčuk - Creative Commons BY-NC-SA [cit. 2016-06-03]. Dostupné z: http://voho.cz/wiki/producer-consumer/

[18] DONLIN, Adam. An Analysis of the Producer-Consumer Problem [online]. [cit. 2016-06-03]. Dostupné z: http://www.dcs.ed.ac.uk/home/adamd/essays/ex1.html

[19] Nexys4 DDR™ FPGA Board Reference Manual. Digilent: Beyond theory. 2014, 25-26.

[20] BURDA, Zdeněk. Mikrofony [online]. 1999 [cit. 2016-04-29]. Dostupné z: http://www.zdenda.com/1999/01/mikrofony/

[21] VOJÁČEK, Antonín. MEMS mikrofony: Obecný popis struktury a funkce. Automatizace.hw.cz: rady a poslední noviky z oboru [online]. 2010 [cit. 2016-05-26]. Dostupné z: http://automatizace.hw.cz/mems-mikrofony-obecny-popis-struktury-a-funkce

[22] VOJÁČEK, Antonín. MEMS: Co to je a jak to vypadá? Vyvoj.hw.cz: profesionální elektronika [online]. 2006 [cit. 2016-05-26]. Dostupné z: http://vyvoj.hw.cz/clanek/2006111901

[23] NĚMEC, Miloš. MEMS: Předchůdce nanotechnologií [online]. 2007 [cit. 2016-05-26]. Dostupné z: http://www.milosnemec.cz/clanek.php?id=101

[24] KAŇKA, Jan. Intenzita zvuku – decibel. Akustika staveb [online]. © Copyright Topinfo s.r.o. 2001-2016, všechna práva vyhrazena. [cit. 2016-05-26]. Dostupné z: http://stavba.tzb-info.cz/akustika-staveb/216-intenzita-zvuku-decibel

[25] DAJBYCH, Václav. MVVM: model-view-viewmodel. DotNETportal.cz [online]. 2009 [cit. 2016-03-21]. Dostupné z: http://www.dotnetportal.cz/clanek/4994/MVVM-Model-View-ViewModel

[26] JECHA, Tomáš. Jazyk XAML. DotNETportal.cz [online]. 2012 [cit. 2016-03-25]. Dostupné z: http://www.dotnetportal.cz/clanek/198/Jazyk-XAML

[27] (Diskrétní) Fourierova transformace. Aplikovaná fyzika: Univerzita Palackého v Olomouci. 2003.

[28] Základní pojmy: PPM, CPPM, PWM, PCM a S.BUS. OpenTX [online]. 2015 [cit. 2016-06-11]. Dostupné z: http://www.opentx.cz/index.php/Z%C3%A1kladn%C3%AD_pojmy:_PPM,_CPPM,_PWM,_PCM_a_S.BUS

[29] Modulation and Different Types of Modulation. Electronics hub [online]. 2015 [cit. 2016-06-11]. Dostupné z: http://www.electronicshub.org/modulation-and-different-types-of-modulation/#4_Pulse_Duration_Modulation_PDM_or_Pulse_Width_Modulation_PWM

[30] JIRAVA, Jarda. Používáme MVVM - jednoduchý ViewModel. Xaml.cz [online]. 2010 [cit. 2016-04-11]. Dostupné z: http://xaml.cz/wpf/pouzivame-mvvm-jednoduchy-viewmodel/

[31] JIRAVA, Jarda. DataBinding a DataTemplate. Xaml.cz [online]. 2010 [cit. 2016-04-11]. Dostupné z: http://xaml.cz/wpf/databinding-a-datatemplate/

[32] Okenní aplikace v C# .NET WPF. ITnetwork.cz [online]. [cit. 2016-04-11]. Dostupné z: http://www.itnetwork.cz/csharp/wpf

[33] Visual Studio: Visual Studio Community [online]. Microsoft, © 2016 [cit. 2016-08-30]. Dostupné z: https://www.visualstudio.com/cs-cz/products/visual-studio-community-vs.aspx

[34] MicroZed. ZedBoard.org [online]. [cit. 2016-06-19]. Dostupné z: http://microzed.org/product/microzed

[35] ORSÁG, Petr. Měření základních charakteristik mikrofonů. Brno, 2011. Bakalářská práce. Vysoké učení technické v Brně. Vedoucí práce Havránek Z.

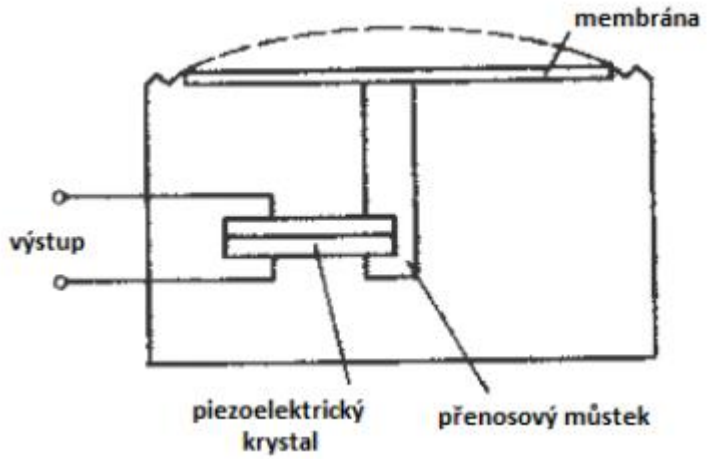
A. Příloha

A.1. Obsah adresářů na přiloženém CD

- Diplomová práce
  - text práce ve formátu *.pdf
  - fulinova_veronika_diplomova_prace_2015_2016.pdf
- Zdrojový kód
  - C# WPF projekt psaný v MS Visual Studiu 2015

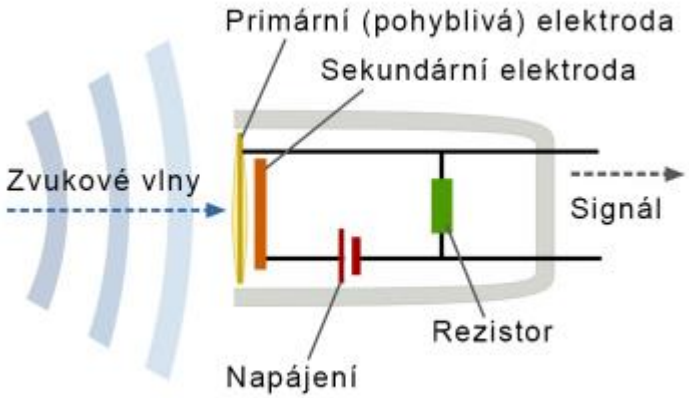
A.2. Nejběžnější typy mikrofonů

A.2.1. Piezoelektrické (krystalové)



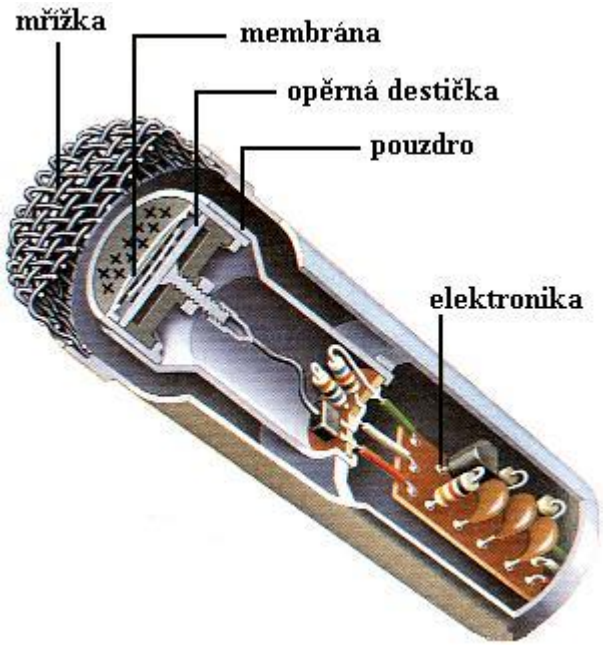
Obrázek 31 - Piezoelektrický mikrofon [35]

A.2.2. Elektrostatické (kondenzátorové/kapacitní)



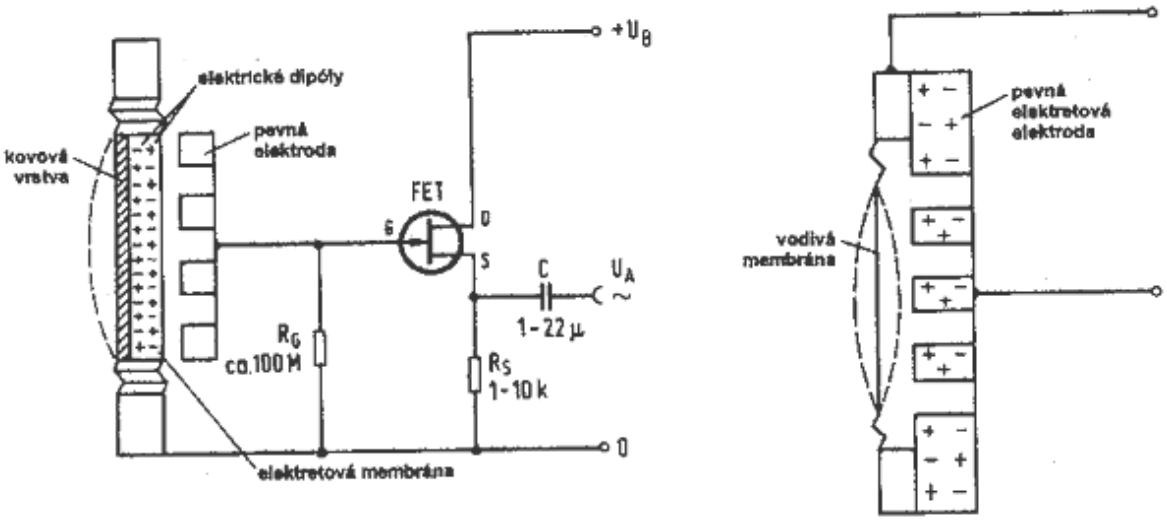
Obrázek 32 - Elektrostatický (kapacitní) mikrofon[12]

[12] http://noel.feld.cvut.cz/vyu/a2b31hpm/images/thumb/8/80/Kondenzatorovy_schema.png/350px-Kondenzatorovy_schema.png



Obrázek 33 - Elektrostatický (kapacitní) mikrofon v praxi[13]

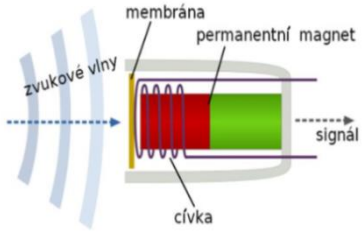
A.2.3. Elektretové



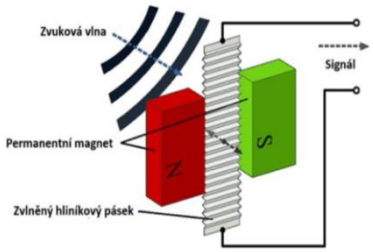
Obrázek 34 - Elektretový mikrofon[14]

[13] http://fyzika.jreichl.com/data/E_elektroakustika_soubory/image027.jpg
[14] http://www.zdenda.com/system/files/elektretmic.gif

A.2.4. Elektrodynamické



Obrázek 35 - Elektrodynamický mikrofon - Cívkové provedení[15]

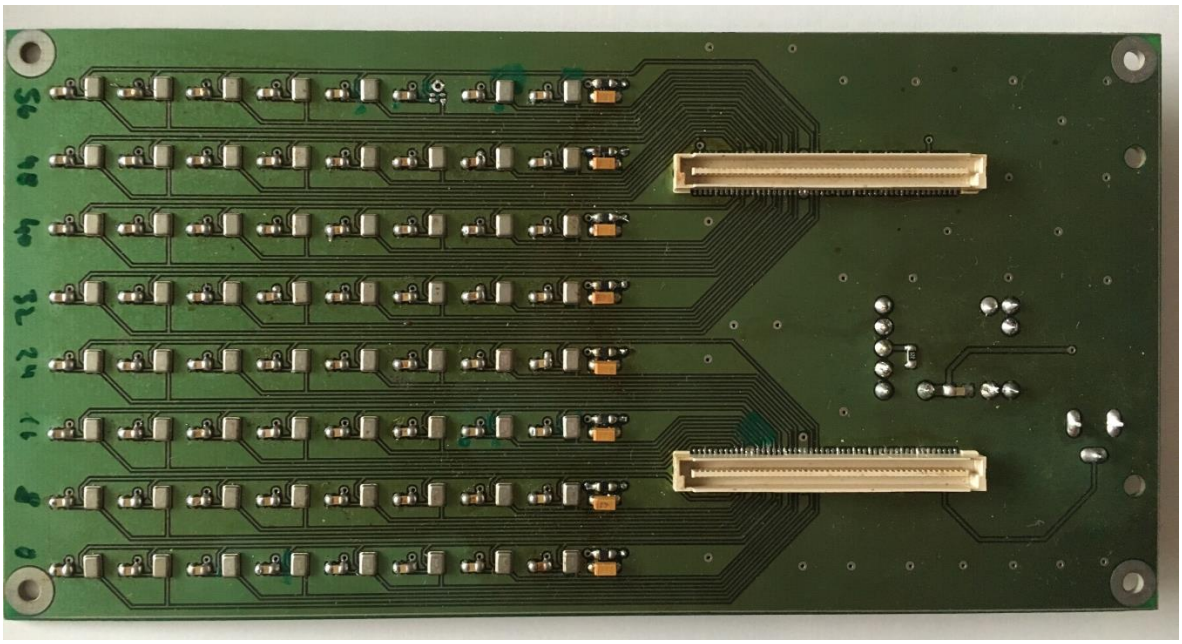


Obrázek 36 - Elektrodynamický mikrofon - páskové provedení[16]

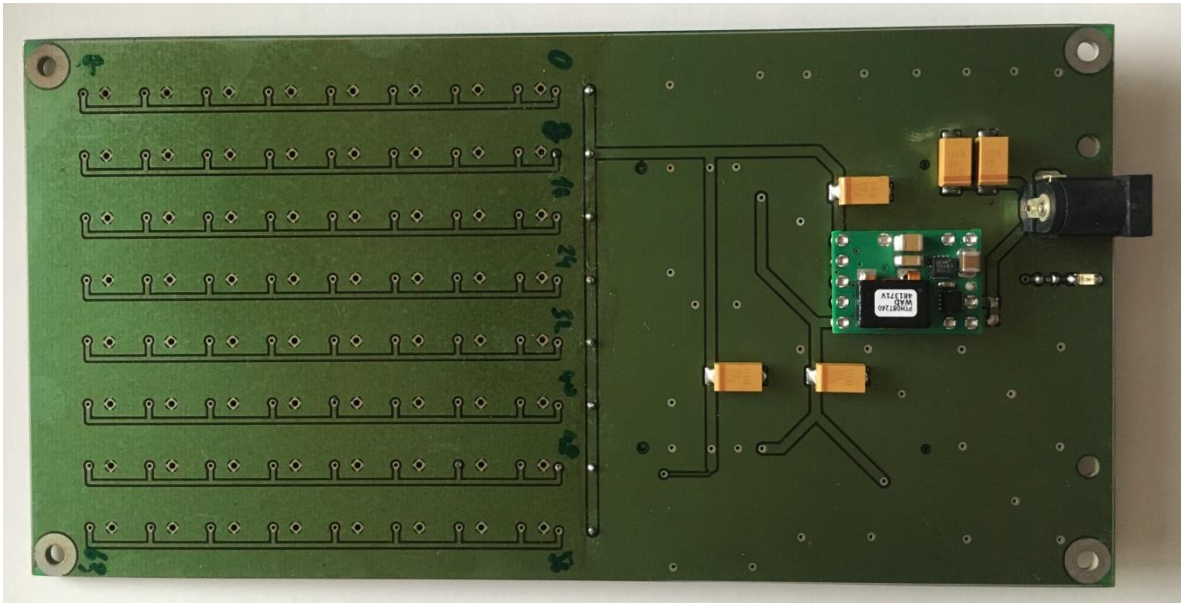
[15] http://www.mff.cuni.cz/verejnost/zpravicky/reproduktor.jpg

[16] https://upload.wikimedia.org/wikipedia/commons/thumb/8/8f/B%C3%A4ndchenmikrofon_SK.svg/220px-B%C3%A4ndchenmikrofon_SK.svg.png

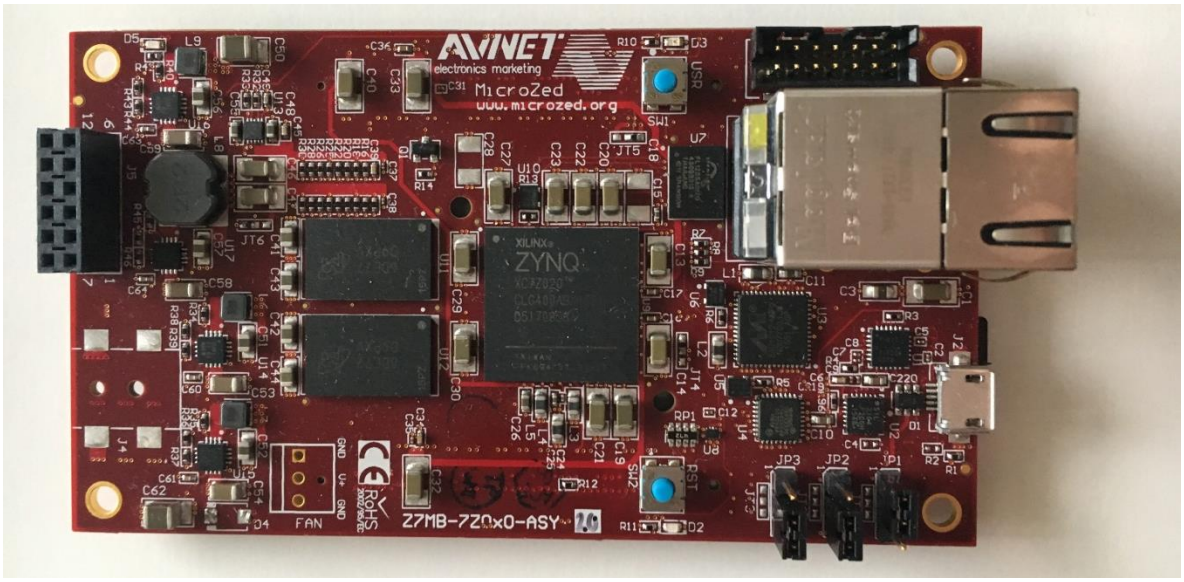
A.3. Vývojová deska



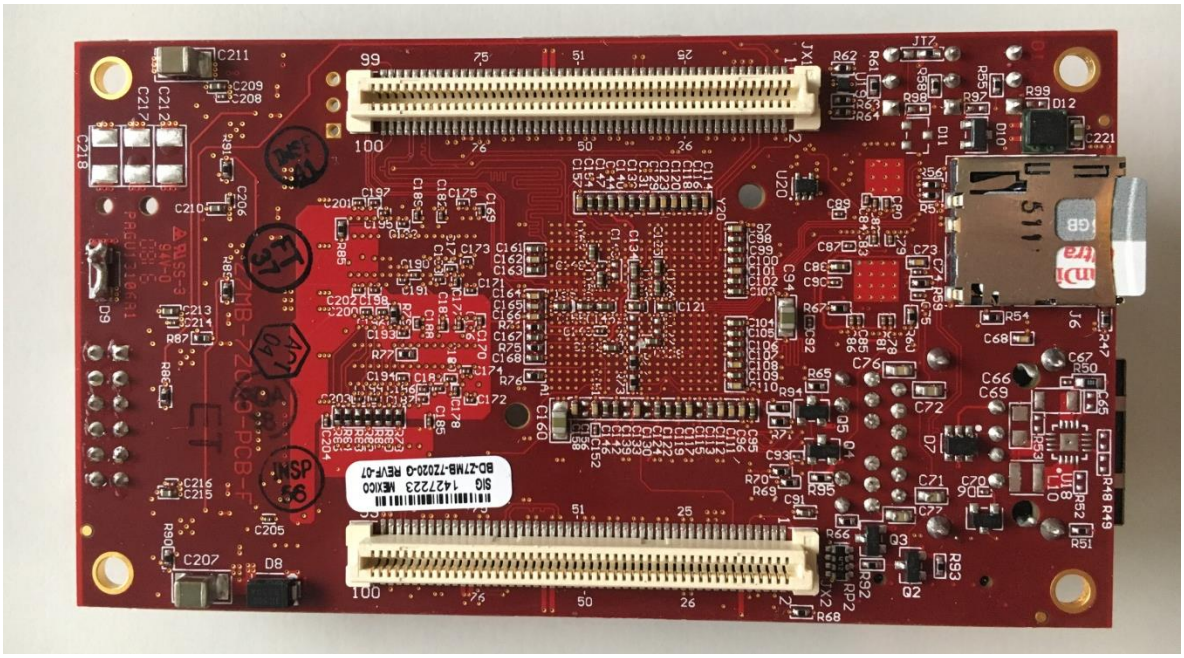
Obrázek 37 - Pole MEMS mikrofonů shora



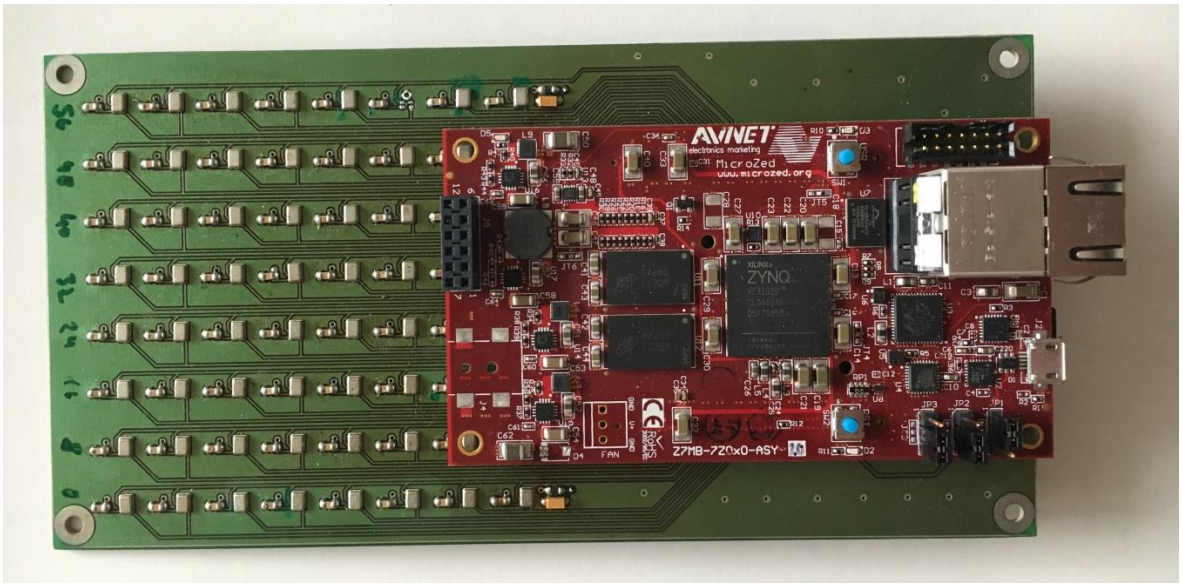
Obrázek 38 - Pole MEMS mikrofonů zdola



Obrázek 39 - Vývojová deska MicroZed shora



Obrázek 40 - Vývojová deska MicroZed zdola



Obrázek 41 - Vývojová deska MicroZed rozšířená o pole MEMS mikrofonů

A.4. Asynchronní klient

```
switch (current_state) {
case AsynchronoutClientFSMStates.ACFSM_NotStarted:
        // klient nebezi
        current_state = AsynchronoutClientFSMStates.ACFSM_NotStarted;
        return;
case AsynchronoutClientFSMStates.ACFSM_Started:
        //pokud jsou ve fronte jiz nejake prikazy
        if (command_queue.Count > 0)
        {
                // mereni casu
                timer.Start();
                // klient bezi
                // vyjmememem prikaz z fronty
                lock (lck) { current_command = command_queue.Dequeue(); }
                // vytvorime retezec prikazu
                byte[] lbuf = Encoding.ASCII.GetBytes(current_command.CommandString);
                // retezcem vytvorime novy prenos
                current_transfer = new TransferState(lbuf.Length, lbuf);
                // dle typu prikazu prejdeme do ruznych vetvi automatu
                switch (current_command.CommandBehaviour)
                {
                        case commands_behaviour_enum.MICSRV_SIMPLE_BEHAVIOUR:
                                // jednoduchy set/get
                                current_state =
AsynchronoutClientFSMStates.ACFSM_SimpleSend_MessageSent;
                                break;
                                case commands_behaviour_enum.MICSRV_SEND_BEHAVIOUR:
                                // posilani dat serveru
                                current_state =
AsynchronoutClientFSMStates.ACFSM_DataSend_MessageSent;
                                break;
                        case commands_behaviour_enum.MICSRV_RECV_BEHAVIOUR:
                                // prijimani dat od serveru
                                current_state =
AsynchronoutClientFSMStates.ACFSM_DataRecv_MessageSent;
                                break;
                        default:
                                break;
                }
                // odeslani prikazu
                send(current_transfer);
        }
        break;
// Simple send/recv branch
case AsynchronoutClientFSMStates.ACFSM_SimpleSend_MessageSent:
        // odeslano OK
        current_state = AsynchronoutClientFSMStates.ACFSM_SimpleSend_ResponseReceived;
        // vytvorime transfer buffer
        current_transfer = new TransferState(128, new byte[128], true);
        // a prijmememe odpoved
        receive(current_transfer);
        break;
case AsynchronoutClientFSMStates.ACFSM_SimpleSend_ResponseReceived:
        // odpoved prijata OK
        // parsing odpovedi
        resp = CommandCreator.ParseResponse(current_command.ID, current_transfer.Buffer,
current_command.CommandBehaviour);
        resp.Callback = current_command.Callback;
        // vlozeni do fifo
        lock (lck) { response_queue.Enqueue(resp); }
        timer.Stop();
        timer.Reset();
        if (resp.Success == false)
        {
                // v pripade ER vyvolame vyjimku
                current_state = AsynchronoutClientFSMStates.ACFSM_ErrorState;
```

```
                OnTransferError();
        }
        else
        {
                // v pripade OK vyvolame handler OnTransferDone
                current_state = AsynchronoutClientFSMStates.ACFSM_Started;
                OnTransferDone();
        }
        break;
// Data send branch
case AsynchronoutClientFSMStates.ACFSM_DataSend_MessageSent:
        // poslani prikazu OK
        current_state = AsynchronoutClientFSMStates.ACFSM_DataSend_ResponseReceived;
        current_transfer = new TransferState(128, new byte[128], true);
        // pockame na potvrzeni
        receive(current_transfer);
        break;
case AsynchronoutClientFSMStates.ACFSM_DataSend_ResponseReceived:
        // potvrzeni prijato
        // parsing odpovedi
        resp = CommandCreator.ParseResponse(current_command.ID, current_transfer.Buffer,
current_command.CommandBehaviour);
        resp.Callback = current_command.Callback;
        if (resp.Success == false)
        {
                // chyba - vyvolame vyjimku
                current_state = AsynchronoutClientFSMStates.ACFSM_ErrorState;
                response_queue.Enqueue(resp);
                OnTransferError();
        }
        else
        {
                // Odpoved OK, posleme samotna data
                current_state = AsynchronoutClientFSMStates.ACFSM_DataSend_DataSent;
                current_transfer = new TransferState(current_command.Data.Length,
current_command.Data);
                send(current_transfer);
        }
        break;
case AsynchronoutClientFSMStates.ACFSM_DataSend_DataSent:
        // data odeslana
        current_transfer = new TransferState(128, new byte[128], true);
        current_state = AsynchronoutClientFSMStates.ACFSM_DataSend_ConfirmationReceived;
        // cekame na odpoved
        receive(current_transfer);
        break;
case AsynchronoutClientFSMStates.ACFSM_DataSend_ConfirmationReceived:
        // odpoved prijata
        // parsing odpovedi
        resp = CommandCreator.ParseResponse(current_command.ID, current_transfer.Buffer,
current_command.CommandBehaviour);
        resp.Callback = current_command.Callback;
        timer.Stop();
        timer.Reset();
        lock (lck) { response_queue.Enqueue(resp); }
        if (resp.Success == false)
        {
                // chyba
                current_state = AsynchronoutClientFSMStates.ACFSM_ErrorState;
                OnTransferError();
        }
        else
        {
                // OK
                current_state = AsynchronoutClientFSMStates.ACFSM_Started;
                OnTransferDone();
        }
        break;
// Data receive branch
```

```
case AsynchronoutClientFSMStates.ACFSM_DataRecv_MessageSent:
        // odeslano OK
        // pripravime se na prijeti odpovedi
        current_state = AsynchronoutClientFSMStates.ACFSM_DataRecv_ResponseReceived;
        current_transfer = new TransferState(128, new byte[128], true);
        receive(current_transfer);
        break;
case AsynchronoutClientFSMStates.ACFSM_DataRecv_ResponseReceived:
        // odpoved prijata
        resp = CommandCreator.ParseResponse(current_command.ID, current_transfer.Buffer,
current_command.CommandBehaviour);
        resp.Callback = current_command.Callback;
        if (resp.Success == false)
        {
                // chyba - obsluha
                current_state = AsynchronoutClientFSMStates.ACFSM_ErrorState;
                response_queue.Enqueue(resp);
                OnTransferError();
        }
        else
        {
                // OK, pripravime buffer a prijmeme data
                current_state = AsynchronoutClientFSMStates.ACFSM_DataRecv_DataReceived;
                current_transfer = new TransferState((int)resp.ResponseParam, new
byte[resp.ResponseParam]);
                receive(current_transfer);
        }
        break;
case AsynchronoutClientFSMStates.ACFSM_DataRecv_DataReceived:
        // data prijata
        timer.Stop();
        timer.Reset();
        // vytvorime "falesnou" odpoved a vlozime ji do fifo
        resp = new Response();
        resp.Callback = current_command.Callback;
        resp.Success = true;
        resp.ID = current_command.ID;
        resp.Data = current_transfer.Buffer;
        lock (lck) { response_queue.Enqueue(resp); }
        current_state = AsynchronoutClientFSMStates.ACFSM_Started;
        OnTransferDone();
        break;
default:
        // V ostatni stavech neni provadena zadna cinnost
        // v pripade ErrorState je nutno externe automat resetovat
        break;
}
```

Zdrojový kód 23 - Přechodová funkce asynchronního klienta