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

Digital Multi-Channel Audio Workstation for Live Performance

Master thesis

Study programme:	N2612 – Electrical engineering and informatics
Study branch:	1802T007 – Information technology
Author:	**Bc. Daniel Louda**
Supervisor:	doc. Ing. Zbyněk Koldovský, Ph.D.
Consultants:	Ing. Jiří Málek, Ph.D
	Fernando Perdigão

Liberec 2019

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Zadání diplomové práce

Digitální vícekanálová audio stanice pro živá vystoupení

Jméno a příjmení:	**Bc. Daniel Louda**
Osobní číslo:	M16000177
Studijní program:	N2612 Elektrotechnika a informatika
Studijní obor:	Informační technologie
Zadávající katedra:	Ústav informačních technologií a elektroniky
Akademický rok:	**2017/2018**

Zásady pro vypracování:

1. Seznamte se s knihovnami PortAudio a Qt pro C++. Zvolte si vhodné vývojové nástroje pro jazyk C++ a vhodné API (např. ASIO).
2. Vytvořte aplikaci pro míchání audio signálů v reálném čase s následujícími možnostmi: -možnost vícestopého záznamu/přehrávání -import/export WAV souborů -přidávání/ubírani mono/stereo stop -nastavení pevného nebo plovoucího tempa, ve druhém případě řízeného vybraným audio vstupem -výstup synchronizace tempa na MIDI a metronomu do vybrané audio stopy -možnost vložení VST pluginu do signálové cesty před výstupem -správa souborů a playlistů s rychlým načítáním
3. Upravte grafické uživatelské prostředí aplikace tak, aby bylo vhodné pro ovládání při živých vystoupení. Ověřte aplikaci v reálném provozu.
4. Text práce zpracujte v anglickém jazyce.

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

Seznam odborné literatury:

[1] H. L. Van Trees, Optimum Array Processing: Part IV of Detection, Estimation, and Modulation Theory, John Wiley & Sons, Inc.,2002.

[2] U. Zolzer, DAFX: Digital Audio Effects, John Wiley & Sons, 2002.

Vedoucí práce:	doc. Ing. Zbyněk Koldovský, Ph.D. Ústav informačních technologií a elektroniky
Konzultanti práce:	Ing. Jiří Málek, Ph.D. Ústav informačních technologií a elektroniky
	Fernando Perdigao Universidade de Coimbra, Portugalsko
Datum zadání práce:	19. října 2017
Předpokládaný termín odevzdání:	14. května 2018

L. S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

prof. Ing. Ondřej Novák, CSc.
vedoucí ústavu

V Liberci 19. října 2017

Declaration

30. 4. 2019 Bc. Daniel Louda

Digital Multi-Channel Audio Workstation for Live Performance

Abstract

Whenever there is complex audio playback or recording necessary, audio workstations have been used to fit the needs of musicians and sound engineers. In this thesis, research and development of a digital audio workstation (DAW) is described. A software DAW is an application for personal computers that uses internal or external sound card to obtain, process and output audio in real-time, allowing application to be used during live performances where low latency output is mandatory. Application is implemented in C++ using Qt graphical framework and PortAudio audio I/O library. It allows users to create custom routing between sound card input and output channels as well as audio files. Several features like creating and saving of a playlist settings or displaying graph of a signal in time were implemented. Interface for signal processing units was designed and four processing units were created that can be applied on any input selected by the user. These units are: Volume Processing Unit to control the volume of the signal, VST Processing Unit that allows usage of open standard for application independent DSP blocks called VST Plug-ins, Tempo Processing Unit that implements state-of-the-art algorithm for detecting tempo of a song using spectral flux and MIDI Process Unit for sending pre-defined MIDI message triggered by musical onset to selected MIDI device.

Keywords: digital audio workstation, music processing, onset detection

Digitální vícekanálová audio stanice pro živá vystoupení

Abstrakt

Digitální audio stanice (DAW) jsou využívány muzikanty a zvukovými inženýry, kdykoliv je vyžadováno komplexní přehrávání nebo nahrávání zvuku. Vývoj DAW specializované pro živá vystoupené je popsán v této práci. Softwarová DAW pro živá vystoupení je aplikace pro osobní počítače, která využívá interní nebo externí zvukovou kartu k přijímání a odesílání signálu s minimální latencí. Signál může být dle nastavení aplikace zpracován a vyhodnocován. Aplikace je implementovaná v programovacím jazyce C++ s použitím grafického frameworku Qt a knihovny PortAudio pro práci se zvukovými vstupy a výstupy. Umožňuje uživatelům vytvoření a uložení vlastních pravidel pro přesměrování a slučování vstupů a výstupů zvukové karty a zvukových souborů. Jsou implementovány funkce jako zobrazení grafu signálu v čase nebo vytváření playlistů. Bylo navrhnuto rozhraní pro zpracování signálu a implementovány čtyři jednotky, které je možné aplikovat na libovolný vstup. Tyto jednotky jsou: Volume Processing Unit pro úpravu hlasitosti signálu, VST Processing Unit, která umožňuje využití otevřeného standardu pro nezávisle DSP bloky nazvané VST Plug-in, Tempo Processing Unit, která implementuje moderní algoritmus pro detekci tempa na základě analýzy spektrální diference a MIDI Process Unit pro odesílání předdefinovaných MIDI zpráv při detekci první doby.

Klíčová slova: digitální audio stanice, zpracování hudby, detekce tempa

Acknowledgements

I would like to express enormous thanks to all the people that made this paper possible. You know who you are.

Contents

List of abbreviations

DSP	Digital Signal Processing
DAW	Digital Audio Workstation
PA	PortAudio
CPU	central processing unit
GPU	graphic processing unit
STFT	short-time Fourier transform
FFT	fast Fourier transform
VST	Virtual Studio Technology
SDK	software development kit
API	application programming interface
MIDI	Musical Instrument Digital Interface
I/O	input/output
OS	operating system
ASIO	Audio Stream Input/Output
ALSA	Advanced Linux Sound Architecture
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
MS	Microsoft
GNU	GNU's Not Unix
GCC	GNU Compiler Collection
MSYS	Minimal SYStem
MSVC	Microsoft Visual C++
bpm	beats per minute
CLI	Command Line Interface

1 Introduction

Whenever there is complex sound recording, editing or producing necessary, people have been using audio workstations to perform such tasks. It took almost 50 years from the first reel-to-reel recording to the creation of the first digital audio workstation (DAW) in the 1970s. This was made possible by making computers smaller, more capable and affordable. As this trend continued, DAWs were becoming more available for the sound engineers. Higher computational power allowed bigger possibilities for the features that manufactures implemented and thus transitioning from the bulky, simple and expensive first attempts to the state-of-the-art devices we call digital audio workstations today.

As a DAW could be considered a software application, integrated standalone unit or complex architecture of devices connected and controlled by a central computer. Personal computers and software implementations allowed for the wide spread of digital audio workstations and made them affordable choices for both professional and non-professional usage while maintaining high functionality.

The goal of this thesis is to create an open-source application for musicians or audio engineers that allows real-time sound card input to output routing and to implement some of the features traditionally used by digital audio workstations to enhance user experience while using this application during live performances.

In this paper, an implementation of an application in C++ targeting operating system Microsoft Windows is described. The application is created with an emphasis on live performances and implements a set of features. Proposed digital audio workstation features graphical user interface created in Qt framework, playlist creation, ability to record input, custom signal routing settings that can be saved and loaded from an XML file, ability to use independent audio processing blocks called VST Plug-ins, ability to respond to musical onset with MIDI messages and tempo detection. Suitable libraries are selected for the real-time input to output routing, fast Fourier transform and audio file import and export.

In the theoretical part of the paper, topics necessary for the development are covered. First, state-of-the-art algorithms for onset detection and tempo analysis from the audio signal are described. After that, a brief introduction to Virtual Studio Technology (VST) and MIDI follows. In the practical part of the thesis, the development of the proposed application is described together with the environment and libraries best fitting the needs of a DAW.

1.1 Available Alternatives

In the field of software Digital Audio Workstations there are several options to choose from. Two notable ones are described below.

1.1.1 Ableton Live

Ableton Live is a DAW for MS Windows and MacOS that includes features for both music production and live performances. The basic musical building blocks of Ableton Live are called clips. A clip is a piece of musical material: a melody, a drum pattern, a bassline or a complete song. Ableton Live allows users to record and alter clips, and to create larger musical structures from them: songs, scores, remixes, DJ sets or stage show. Ableton Live operates in two different modes or views that can hold clips – Session view or Arrangement view. The Arrangement is a layout of clips along a musical timeline; the Session is a real-time-oriented "launching base" for clips [2].

The first version of Ableton Live was released in 2001 and Ableton Live is still under active development with the latest version 10 being released in February 2018. Ableton Live has a proprietary license and is currently being sold in three editions – Intro, Standard and Suite for 79, 349 and 599 EUR respectively[1]. These editions differ in available sound banks, a number of software instruments and a number of audio and MIDI effects.

1.1.2 Steinberg Cubase

Steinberg Cubase is a DAW developed by Steinberg that was originally created for the Atari ST computer in 1989, making it one of the oldest software digital audio workstation that is still being developed. In the 1999 Steinberg introduced virtual instrument interface for software synthesizers known as VSTi. This allowed third-party developers to create and sell virtual instruments for Cubase and contributed to its popularity. Latest version 9.5 is available on Microsoft Windows and macOS. Steinberg Cubase is used for recording, mixing and editing of music. Three editions are currently available - Pro, Artist, and Elements being sold for 99.99, 309 and 559 EUR[2]. All editions feature 64-bit floating-point audio engine with up to 192 kHz but differ at a number of physical inputs and outputs available, a number of MIDI tracks available for recording and mixing and a number of instrument tracks and VST instruments slots [18].

[1]According to https://www.ableton.com/en/shop/live/

[2]According to https://www.steinberg.net/en/shop/cubase.html

2 Fundamentals on Musical Tempo and Beat Analysis

In music terminology, beat is the regularly occurring pattern of rhythmic stresses in music [10]. It is intuitively defined as periodic pulses or rhythm that people tap along while listening to a musical piece. These pulses are defined by their phase and period. Tempo refers to the rate of the pulses and thus setting the speed or pace of a musical piece. It is usually measured in beats per minute or bpm for short, meaning how many of these periodic pulses we perceive in a minute [10]. In modern songs, the tempo is usually constant throughout the song, although it is not a necessity. In classical music, on the other hand, the tempo is more often defined by Italian terms like Allegro or Largo, giving a rough estimate of the speed of the musical piece. The tempo in classical music is often changing during the song giving the artist another possibility of expressing ideas. Extraction of tempo from a musical piece is an important area in music processing and while it is simple and straightforward for humans to determine the tempo, more complex algorithms are needed for the automated tempo extraction and beat tracking.

2.1 State of the Art

The start of the new beat is usually accompanied by a change of the song in some form. In most of the tempo extraction methods first step is to determine these changes. This step is called onset detection and results in a so-called **novelty curve** that represents level of change in the musical piece as a function of time[3]. Later on novelty curve is analyzed to extract tempo and beat information from the musical piece. Several methods of estimating onsets of the signal have been proposed. For example, in "Music Onset Detection Based on Resonator Time Frequency Image" by Zhou, Mattavelli, and Zoia[21], "A Comparison of Sound Onset Detection Algorithms with Emphasis on Psychoacoustically Motivated Detection Functions" by Collins[4] or "Extracting Predominant Local Pulse Information From Music Recordings" by Grosche and Muller[9]. The next step of analyzing onsets to be able to determine tempo and beat positions of a musical piece has been described for example in "Beat Tracking by Dynamic Programming" by P. W. Ellis[13], "OBTAIN" by Mottaghi et al.[11] or in *Fundamentals of Music Processing* by Müller[12]. Some of the possibilities of onset detection are described in this chapter, together with two methods for extracting tempo information using the detected onsets.

2.2 Onset Detection

The objective of onset detection is to find out the starting time of notes or other events in music. In order to detect onsets we also need to define related terms which are **attack**, **transient** and **decay** as seen on Figure 2.1 with piano note on the left and idealized amplitude envelope on the right. Attack of the note is the part where the sound of the note builds up. Transient may be described as a noise-like sound component of short duration and high amplitude typically occurring at the beginning of a musical tone or a more general sound event [12]. Onset, unlike attack transient or decay, refers to an exact point in time rather than an interval thus giving us good candidates for detecting beats. To determine this point in time changes in the signal are analyzed. In some cases, like playing a note on the piano or percussive instruments, we can determine the onset of the note by the sudden change in signal amplitude envelope. On the other hand, some instruments like violin have a far more subtle increase in signals amplitude making the onset detection harder. An example can be seen on Figure 2.2 where waveform and amplitude envelope of a piano and violin is shown with attack (A), decay (D), sustain (S) and release(R) marked. Because of this phenomenon, several approaches of onset detection are described in the following chapters.

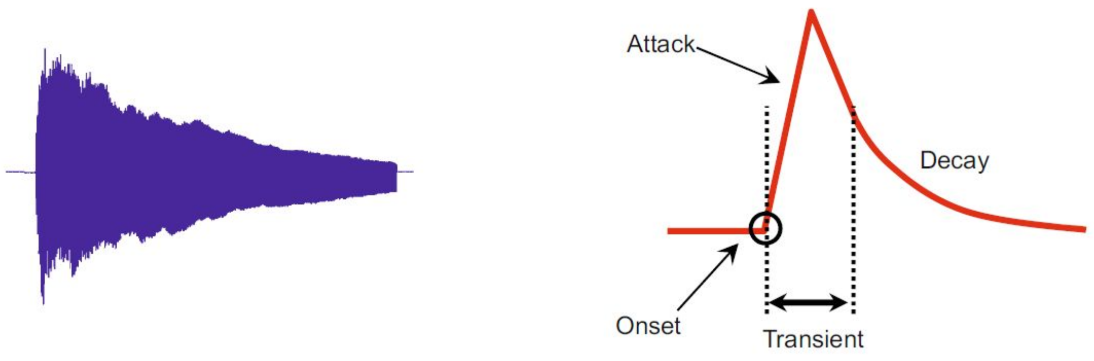


Figure 2.1: Illustration of attack, transient, onset and decay of a note.[12]

2.2.1 Energy-Based Novelty

This method is utilizing the fact that playing a note is often connected with a change in signals energy. To demonstrate this waveform of a piano note can be observed on Figure 2.2. To extract an information about changes in energy in a given time frame of the discrete signal x a discrete window function w is defined which is applied on x to determine local sections. Window function w is a bell-shaped function centered at time zero. $w(m)$ for $m \in [-M : M]$ consists of nonzero samples of w for some $M \in \mathbb{N}$. The local energy of x in a time frame n ($n \in \mathbb{Z}$) with regard to the w

is defined by the following equation.

$$E_w^x(n) = \sum_{m=-M}^{M} |x(n+m)w(m)|^2 \tag{2.1}$$

In order to detect changes in local energy discrete derivative is calculated. This is done by subtracting subsequent samples of the local energy function E_w^x. The result will yield the information about changes in local energy. Halfwave rectification is used because for the detection of onsets only positive values have relevant information (increases in signals energy). Halfwave rectification for $r \in \mathbb{R}$ is defined as

$$|r|_{\geq 0} = \frac{r + |r|}{2}. \tag{2.2}$$

From this point, energy-based novelty function for $n \in \mathbb{Z}$ is obtained.

$$\Delta_{Energy}(n) = |E_w^x(n+1) - E_w^x(n)|_{\geq 0} \tag{2.3}$$

Applying the knowledge that the human ear perceives sound in a logarithmic way leads to further enhancing the results obtained from (2.3). This means that even weaker energy signal parts could change our way of perceiving sound and rhythm. To take this into account switching into logarithmic decibel scale or logarithmic compression is advised. After applying this, following equations are obtained.

$$\Delta_{Energy}^{Log}(n) = |log(E_w^x(n+1)) - log(E_w^x(n))|_{\geq 0} \tag{2.4}$$

or

$$\Delta_{Energy}^{Log}(n) = |log(\frac{E_w^x(n+1)}{E_w^x(n)})|_{\geq 0} \tag{2.5}$$

demonstrating that human perception of differences in energy can be seen as a **ratio** between subsequent local energies rather than their **subtraction**.

Further enhancement of this method is possible by decomposing the signal into several subbands and applying energy-based novelty function separately on each of them and combining the results afterwards. Subband frequencies could be set by applying knowledge about the used instrument and their corresponding frequency ranges and thus increasing the robustness of this algorithm.

2.2.2 Spectral-Based Novelty

In some cases, using energy-based novelty for detecting note onsets does not deliver meaningful results. Such cases could be when onsets are masked by other sound events played at the same time or when sustain phase of a note played with vibrato on musical instruments is having a bigger change in local energy than attack phase of the note making it much harder to detect. Note onsets are typically accompanied not only with a change in signals energy as shown in 2.2 but also with a change in frequency. In this section, it is described how to detect onsets by observing changes in a frequency spectrum. To be able to analyze the discrete signal x in

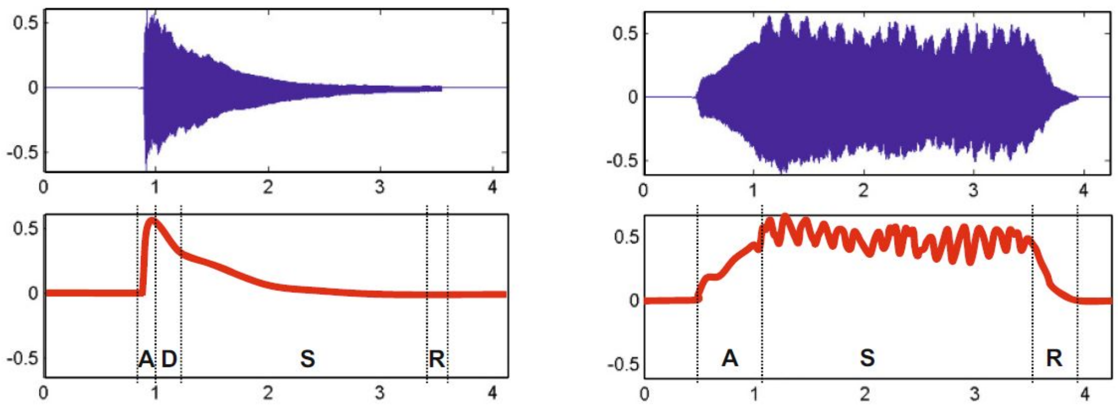


Figure 2.2: Waveform and amplitude envelope of piano (left) and violin (right) playing the same note. [12]

time-frequency domain, signal is transformed using the discrete short-time Fourier transform $\mathcal{X}$ defined as

$$\mathcal{X}[m,k] = \sum_{n=-\infty}^{\infty} x[n]w[n-m]e^{-2\pi jkn/N} \tag{2.6}$$

with $m \in \mathbb{Z}$ and $k \in [0 : N-1]$ where N is the length of DFT, for example 1024.

The next step is to compute the difference between subsequent spectral vectors and therefore obtaining the **spectral-based novelty function** or in other words **spectral flux**.

The first step is to use logarithmic compression to enhance weak spectral components. Doing this will take into account the logarithmic nature of sound intensity and balances out the dynamic range of the signal. Using logarithmic compression on magnitude spectrogram $|\mathcal{X}|$ gives

$$\mathcal{Y} = log(1 + \gamma|\mathcal{X}|) \tag{2.7}$$

for a given constant $\gamma \geq 1$. By selecting bigger γ, weaker spectral components are enhanced that might contain relevant information for detection onsets. On the other hand, it could lead to amplifying non-relevant noise-like features.

After that, the first temporal derivative of logarithmically compressed magnitude spectrogram is computed. The equation for temporal discrete derivative follows.

$$\Delta_{Spectral}^{Log}(n) = \sum_{k=0}^{N/2} \mathcal{Y}[n+1,k] - \mathcal{Y}[n,k] \tag{2.8}$$

Same as in energy-based novelty, negative values are disregarded since we are only interested in positive ones to detect onsets. Using the halfwave rectifier defined in (2.2) gives us the following equation.

$$\Delta_{Spectral}^{Log}(n) = \sum_{k=0}^{N/2} |\mathcal{Y}[n+1,k] - \mathcal{Y}[n,k]|_{\geq 0} \tag{2.9}$$

The result can be further enhanced by subtracting the local average. Local average function $\mu(n)$ with window length $2M+1$ in a time n is defined as

$$\mu(n) = \frac{1}{2M+1} \sum_{-M}^{M} \Delta_{Spectral}^{Log}(n+m). \tag{2.10}$$

Local average is now subtracted from the spectral flux and halfwave rectified resulting in the following equation for $n \in \mathbb{Z}$.

$$\bar{\Delta}_{Spectral}^{Log}(n) = |\Delta_{Spectral}^{Log}(n) - \mu(n)|_{\geq 0} \tag{2.11}$$

First 40 seconds of a rock song and related waveform and spectral-based novelty curve can be seen in Figure 2.3. Peaks in novelty curve represent changes in the frequency spectrum and are good candidates for estimating beat positions. The height of the peaks represents how much the signal changed and can be understood as a confidence in the beat position.

2.2.3 Phase-Based Novelty

In this section onset detection method is described similar to the spectral-based method. Information is obtained from the short-time Fourier transform, but this time, in contrast to the one described earlier, phase of the signal is analyzed to detect onsets of musical events. This is possible because of fact that notes in the transient phase have an unstable phase as appose to the stationary tones that have stable phase.

Consider $\mathcal{X}(n,k) \in \mathbb{C}$ a complex-valued Fourier coefficient of the given frequency index $k \in [0:K]$ in the time frame $n \in \mathbb{Z}$, where $K = N/2$ is the frequency index corresponding to the Nyquist frequency. Using polar coordinates,

$$\mathcal{X}(n,k) = |\mathcal{X}(n,k)| \cdot e^{2\pi j \varphi(n,k)} \tag{2.12}$$

equation is obtained, where $\varphi(n,k) \in [0:1]$ describes phases. Assume that $|\mathcal{X}(n,k)|$ is large and signal x is locally stationary around time frame n. By observing the phase in subsequent time frames one can deduct that if the differences in phase in subsequent time frames are increasing in a way that is linear in a hop size H of STFT. Meaning the differences in this region are approximately constant.

$$\varphi(n,k) - \varphi(n-1,k) \approx \varphi(n-1,k) - \varphi(n-2,k) \tag{2.13}$$

At this point, a first-order difference of phases is calculated with

$$\varphi\prime(n,k) = \varphi(n,k) - \varphi(n-1,k), \tag{2.14}$$

making the second-order difference

$$\varphi\prime\prime(n,k) = \varphi\prime(n,k) - \varphi\prime(n-1,k). \tag{2.15}$$

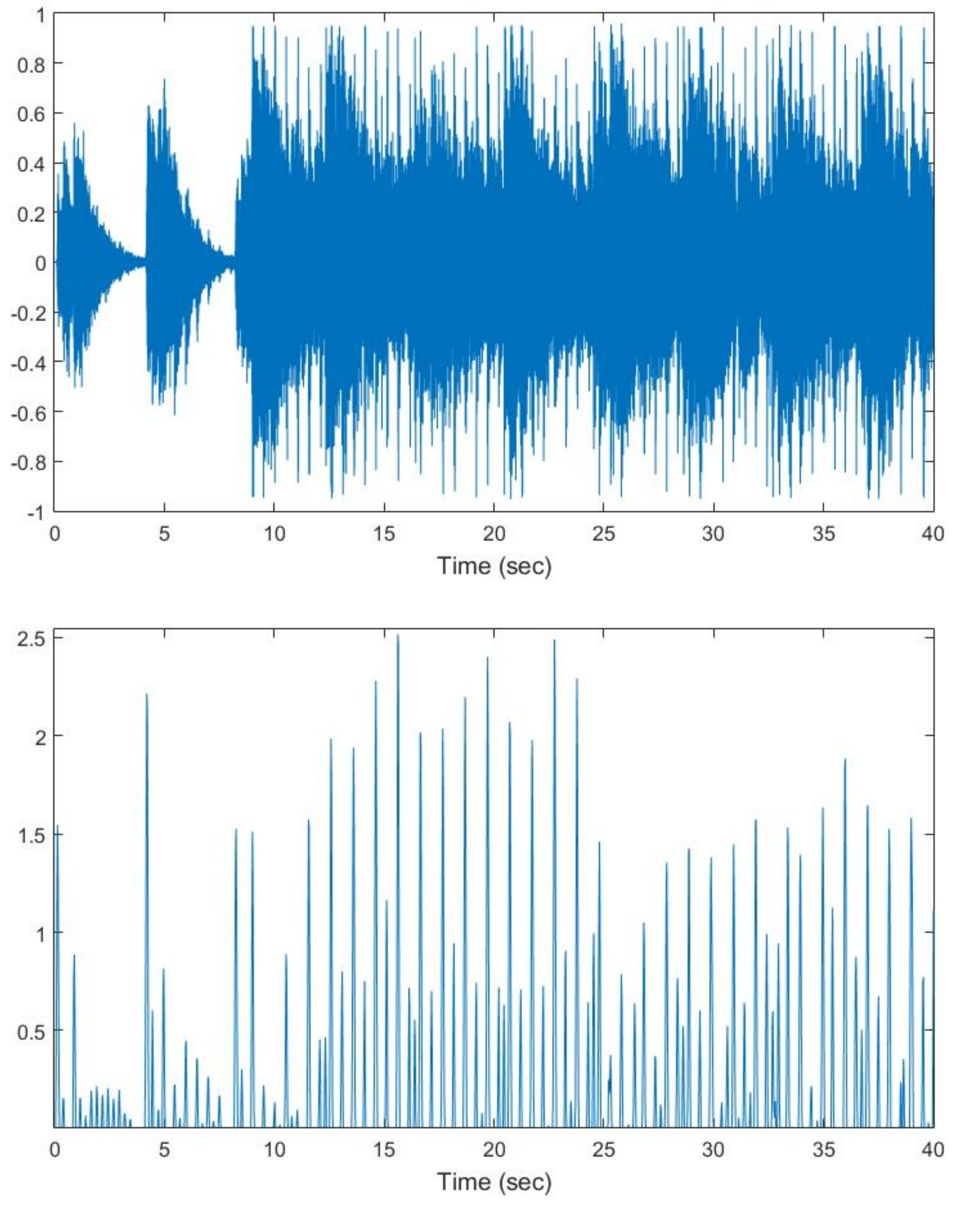


Figure 2.3: Waveform (top) and spectral-based novelty curve (bottom) of a song.

During the steady regions of x one obtain $\varphi\prime\prime(n,k) \approx 0$ on the other hand during a transient phase of note the phase is behaving in an unpredictable manner. This information is used to obtain phase-based novelty function which is defined as the following equation.

$$\Delta_{Phase}(n) = \sum_{k=0}^{K} |\varphi\prime\prime(n,k)| \tag{2.16}$$

2.2.4 Complex-Domain Novelty

In this onset detection method phase information is weighted with the magnitude of the spectral coefficient to increase the robustness of the onset detection. This is done because if $|\mathcal{X}(n,k)|^2$ is small, phase $\varphi(n,k)$ may not behave as expected because of the small noise-like variations from the signal x that could happen even in steady regions of x. Using the fact that phase and magnitude differences in the subsequent frames are approximately constant in the steady regions a steady-state estimate $\hat{\mathcal{X}}$ could be obtained from the following equation.

$$\hat{\mathcal{X}}(n+1,k) = |\mathcal{X}(n,k)| \cdot e^{2\pi i(\varphi(n,k)+\varphi\prime(n,k))} \tag{2.17}$$

$\hat{\mathcal{X}}(n+1,k)$ is the estimation of how the signal is **expected** to behave in the following time index $n+1$. By subtracting the **actual** Fourier coefficient $\mathcal{X}(n+1,k)$ from our estimate $\hat{\mathcal{X}}(n+1,k)$ a $\mathcal{X}'(n+1,k)$ is obtained that reflects the difference between the expectation and the reality.

$$\mathcal{X}'(n+1,k) = |\hat{\mathcal{X}}(n+1,k) - \mathcal{X}(n+1,k)| \tag{2.18}$$

At this point, it is possible to detect how much the signal changed, large values of $\mathcal{X}'(n,k)$ suggesting that a beat onset took place around time frame n. Since we are only interested in note onsets only positive magnitudes are selected. This step is described as

$$\mathcal{X}^+(n,k) = \begin{cases} \mathcal{X}'(n,k) & \text{for } |\mathcal{X}(n,k)| > |\mathcal{X}(n-1,k)| \\ 0 & \text{for } |\mathcal{X}(n,k)| \leq |\mathcal{X}(n-1,k)| \end{cases}. \tag{2.19}$$

Summing the values of $\mathcal{X}^+(n,k)$ across all frequency coefficients gives us the **complex-domain novelty function** $\Delta_{Complex}$.

$$\Delta_{Complex}(n) = \sum_{k=0}^{K} \mathcal{X}^+(n,k) \tag{2.20}$$

2.3 Tempo Analysis

In this section it is described how to take advantage of novelty functions that were previously defined to analyze tempo of a song. Using the knowledge that the beat positions correspond with previously detected onsets and that the beats are at least in a short time period equally spaced it is possible to determine the tempo of a musical piece at a given time. Taking into account that tempo does not have to be constant throughout the musical piece, two methods are described that estimate the tempo of a signal. To visualize tempo information in a musical piece **tempogram** is introduced to showcase changes of a tempo in a signal.

2.3.1 Tempogram

Similar to the spectrogram, tempogram represents time-frequency information about the signal. Frequency is in this case represented by tempo τ measured in beats per minute (BPM) satisfying the formula for frequency $f = 1/T$ given in Hz, where T is the length of the period.

$$\tau = f \cdot 60 \tag{2.21}$$

For example, spikes in novelty function spaced with a 0.25 second period T results in a frequency of $f = 4Hz$ or 240 BPM. A Discrete tempogram $\mathcal{T}(n, \tau)$ is defined as a function of time frame t and tempo τ that demonstrates how well signal around time t corresponds with the tempo τ.

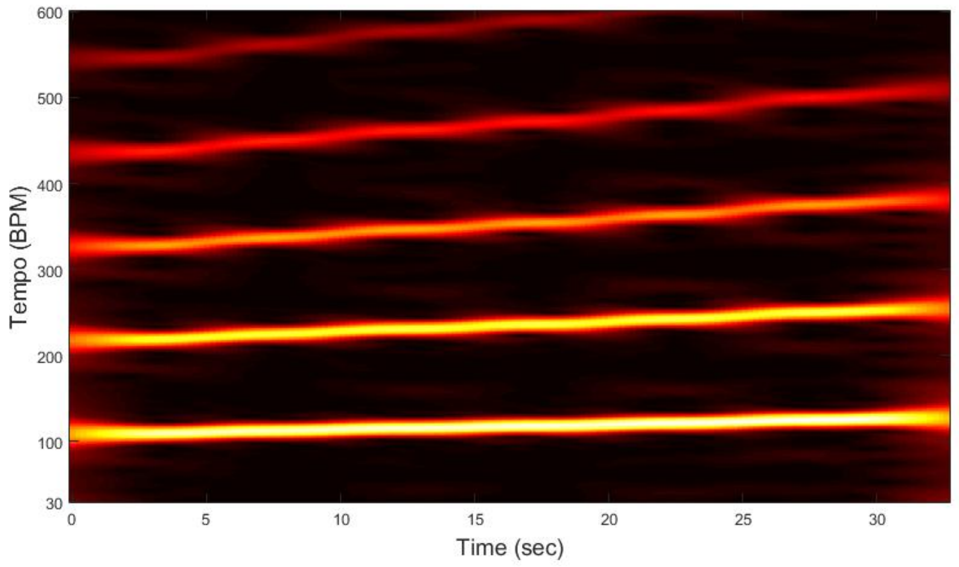


Figure 2.4: Tempogram of a metronome with increasing tempo.

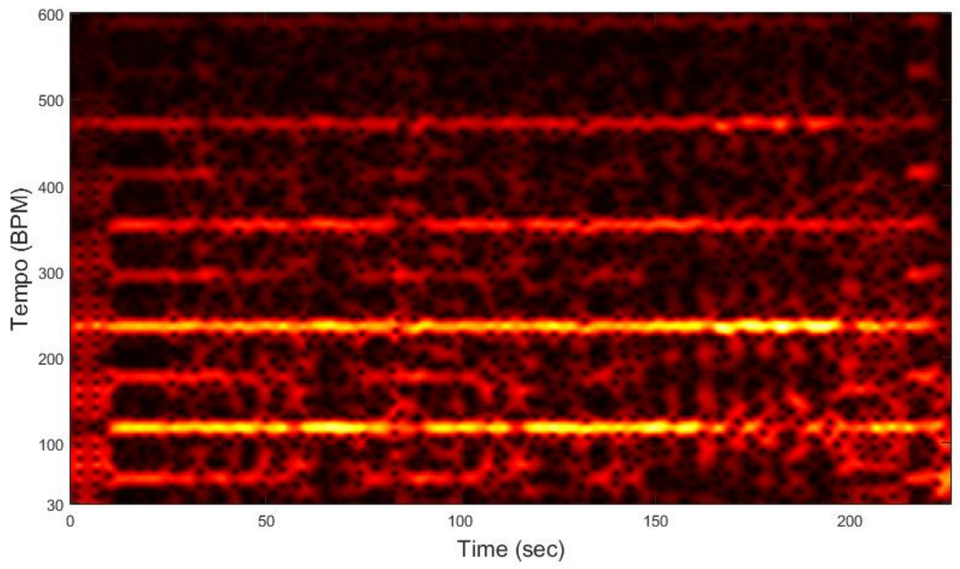


Figure 2.5: Tempogram of a song from Figure 2.3 with roughly constant tempo.

2.3.2 Fourier Tempogram

In this method, Fourier transformation is applied on a novelty function to obtain a tempogram. Using a finite length window function w centered at $n = 0$, for example, Hann window. For the given frequency $f \in \mathbb{R}_{\geq 0}$ and time frame $n \in \mathbb{Z}$ Fourier coefficient is defined as

$$\mathcal{F}(n, \omega) = \sum_{m \in \mathbb{Z}} \Delta(m) w(m-n) e^{-2\pi i \omega m}. \tag{2.22}$$

To obtain Fourier tempogram in beats per minute rather than a Hz equation below is used.

$$\mathcal{T}^F(n, \tau) = |\mathcal{F}(n, \tau)/60| \tag{2.23}$$

Because humans are only capable of perceiving tempo in some frequencies, Θ can only consist of tempi between 30 BPM and 600 BPM. (This information could be used to lower the computational power of the algorithm). Size of the window also plays an important role, bigger the windowing function more precise values we can obtain about the tempo, on the other hand, making it harder to detect sudden changes in tempo. On contrary, shorter windowing function will give us a lower resolution of the detected tempo but better detection of the tempo change. In practice windowing function lengths of 4-12 seconds are suitable.

Example of Fourier tempogram can be seen in 2.5. It demonstrates that the algorithm is not only detecting the original tempo but also tempo harmonics – integer multiples of the original tempo. Tempo harmonics can be negated by making the range of possible tempi lower.

2.3.3 Autocorrelation Tempogram

Another method to extract tempo information from the novelty curve is using autocorrelation. Autocorrelation is a tool used to find repeating patters in the signal by comparing the original signal with a shifted version of itself. It is defined as

$$R_{xx}(\ell) = \sum_{m \in \mathbb{Z}} x(m)x(m+\ell). \tag{2.24}$$

For a discrete, real-valued signal x at lag $\ell \in \mathbb{Z}$. As in the case of Fourier tempogram, window function of a finite length centered around $n = 0$ is created. By applying window function w on a novelty function Δ a windowed novelty function $\Delta_{w,n}$ is obtained

$$\Delta_{w,n} = \Delta(m)w(m-n) \tag{2.25}$$

for $m \in \mathbb{Z}$. By combining equations 2.24 and 2.25 a short-time autocorrelation is obtained.

$$\mathcal{A}(n,\ell) = \sum_{m \in \mathbb{Z}} \Delta(m)w(m-n)\Delta(m+\ell)w(m-n+\ell) \tag{2.26}$$

This will result in a time-lag representation of the musical piece. It is possible to convert time-lag representation to tempo representation by knowing the time resolution of the novelty function. If one time frame of novelty function corresponds to the r seconds, then a time-lag of ℓ corresponds to the $r \cdot \ell$ seconds. Using the 2.21 formula following equation is obtained and used to convert Y-axis to tempo representation.

$$\tau = \frac{60}{r \cdot \ell} \text{ BPM} \tag{2.27}$$

Autocorrelation tempogram of a song with tempo representation of Y-axis can be seen on Figure 2.6. This tempogram apart from the original tempo also detects the tempo subharmonics – integer divisors of the tempo.

2.4 Conclusion

Different approaches for detecting onsets yield different results and are suitable for different applications. The rate of successfully detected onsets depends on factors like a genre of a song or used instruments. Comparison of different approaches for detection onsets can be seen in Figure 2.7 that shows 5 seconds of violin and piano recording. Spectral difference corresponds to the spectral-based novelty described in this chapter. Other possible methods such as wavelet regularity modulus or negative log-likelihood are not covered in this paper and are described in “A Tutorial on Onset Detection in Music Signals” by Bello et al.

The extraction of tempo information is possible using methods described in this chapter. As seen in Figures 2.6 and 2.5 both approaches have different properties. Fourier tempogram method is not only detecting tempo of the musical piece but also tempo harmonics and autocorrelation method is also detecting tempo subharmonics.

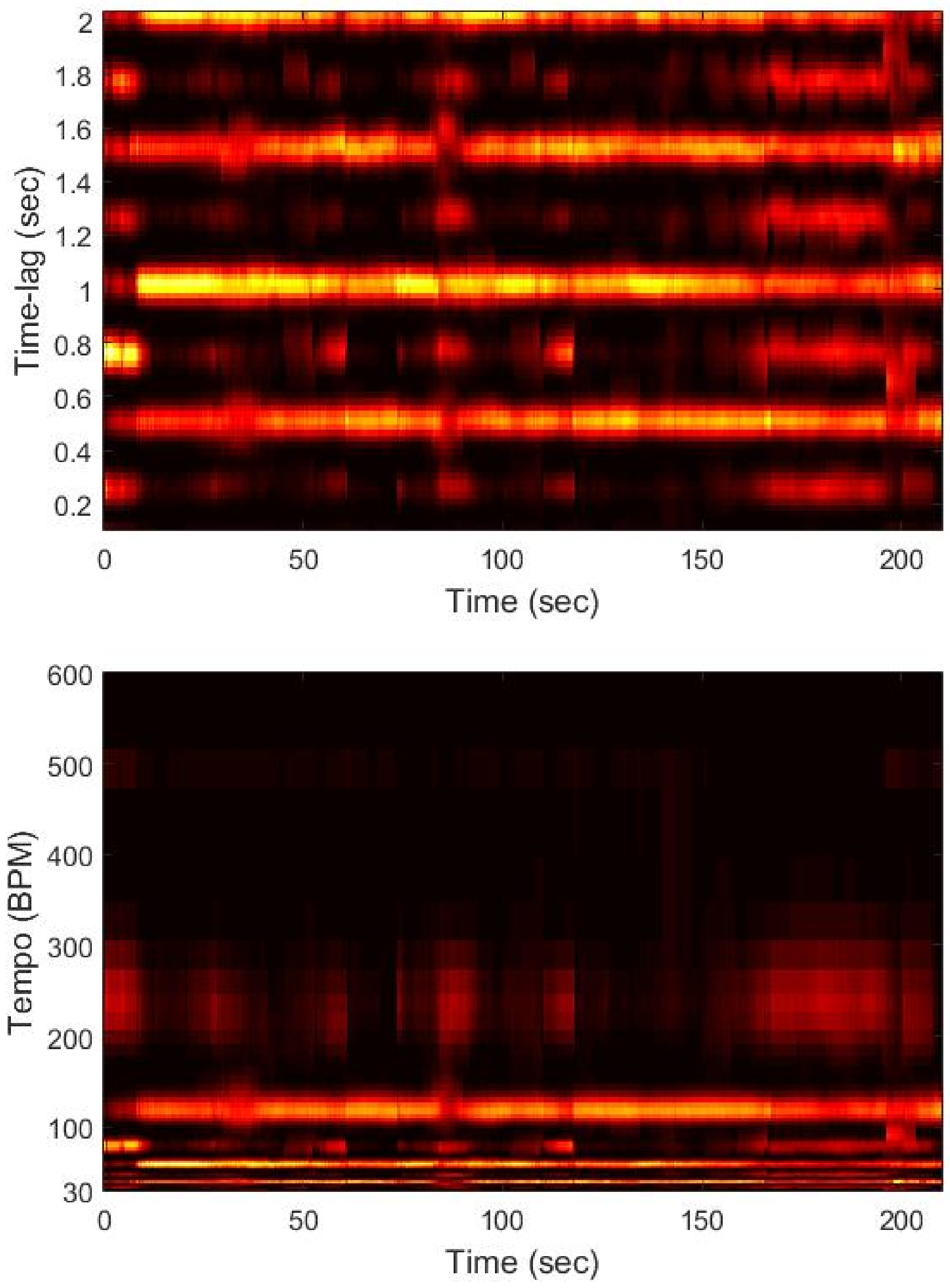


Figure 2.6: Autocorrelation tempogram of a song with time-lag (top) and tempo (bottom) representation of Y-axis.

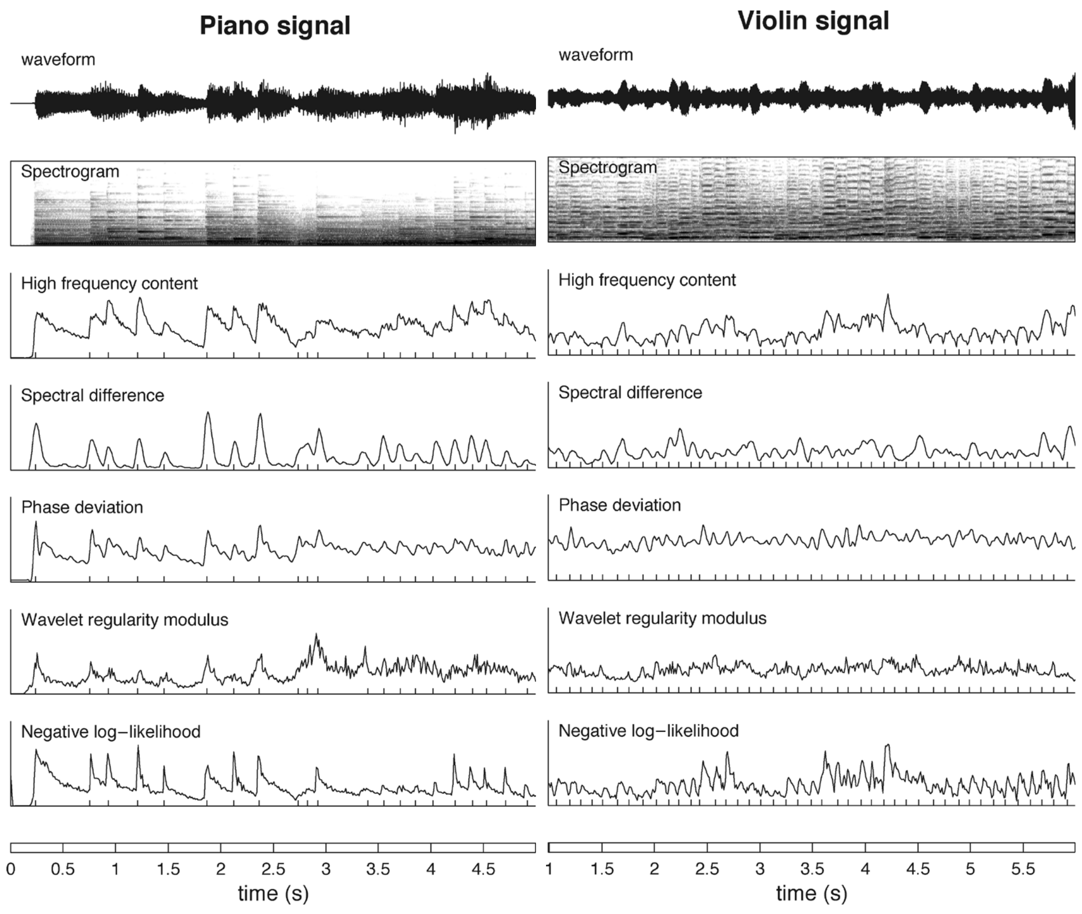


Figure 2.7: Comparision of different approaches for detecting onsets [3].

Spectral-based novelty algorithm and Fourier tempogram is implemented in the application for tempo analysis of audio input because of the detection accuracy when dealing with instruments with longer attack phase of the note as opposed to the energy-based novelty. On the other hand, energy-based novelty is implemented in the application to detect onsets when fast computation and low delay is necessary.

3 VST

Virtual Studio Technology (VST) is a software interface for creating audio plug-ins and their integration in audio processing applications. Developed by Steinberg in 1996, VST became standard on the field of audio plug-ins. Since the first version VST undertook major changes throughout the years. One of the bigger changes was an introduction of VST version 2.0 in 1999, featuring the ability for VST plug-ins to receive MIDI signals. This change made possible for creating virtual instruments. VST 3.0 came out in 2008 with additional features added like audio inputs for VST instruments. Note that versions 2.x and 3.x are not compatible since VST SDK was changed in a major way to allow the new features and to remove some limitations of the older version 2. VST is still under active development by Steinberg with new features being added every version. To this date, the latest version is 3.6.9 and it is the one described in this section [17][20].

3.1 Basic Conception

A VST Plug-in is typically an audio effect or virtual instrument that can be used across applications and operating systems. It does not work as a standalone and requires a hosting application called VST Host, usually in a form of a digital audio workstation. For the VST Host, plug-in is like a black box with defined capabilities based on which interfaces plug-in implements.

3.2 VST Plug-in

VST 3 audio effect or instrument consists of two parts **processing** part and **edit controller** part. Each with corresponding interfaces, processing part implements `IAudioProcessor` and `IComponent` interfaces and edit controller part implements the `IEditController` interface. The processing part handles signal processing and edit controller part handles the graphical user interface and communication between user and plug-in in a form of parameter changes. This change introduced in version 3 allows for the complete separation of the two components enabling them to run in different contexts or even on different computers. Making possible for the GUI component to be updated with much lower frequency than the processing component.

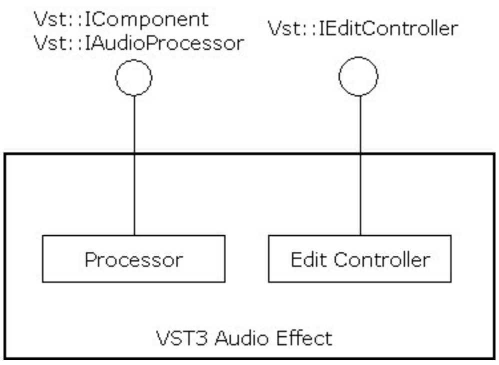


Figure 3.1: VST 3 Plug-in [17].

3.2.1 Processor

This part of VST Plug-in that handles processing of audio is split into two interfaces named `IComponent` and `IAudioProcessor`. Reason for this is to have one generic processing interface `IComponent` and one audio specific interface `IAudioProcessor` to allow for the non-audio plug-ins in the future versions of VST. To be able to use the processor it has to be configured first while being inactive. To do this `ProcessSetup` struct is used to pass all the parameters to the processing unit. These parameters can not be changed while the processor is active. After that, it is possible for the host to change the number of channels in an audio bus if that does not happen the default values of the plug-in are used. Next the processor has to be activated using the `IComponent::setActive(true)` and `IAudioProcessor::setProcessing(true)`. At this point, the component is ready to process audio data. This is done in the following way. Processing of audio is done in a `IAudioProcessor::process` method. All the data mandatory for the processing are passed to the method by the `ProcessData` structure. List of some of the attributes of `ProcessData` with brief descriptions follows.

- `numSamples` – number of samples to process
- `numInputs` – number of audio input busses
- `numOutputs` – number of audio output busses
- `inputs` – buffers of input busses
- `outputs` – buffers of output busses
- `inputParameterChanges` – incoming parameter changes for this block
- `outputParameterChanges` – outgoing parameter changes for this block

When the processing is over VST Host must call `IAudioProcessor::setProcessing(false)` after the last process call. Detailed workflow of the processing part from the VST documentation can be seen in Figure 3.2.

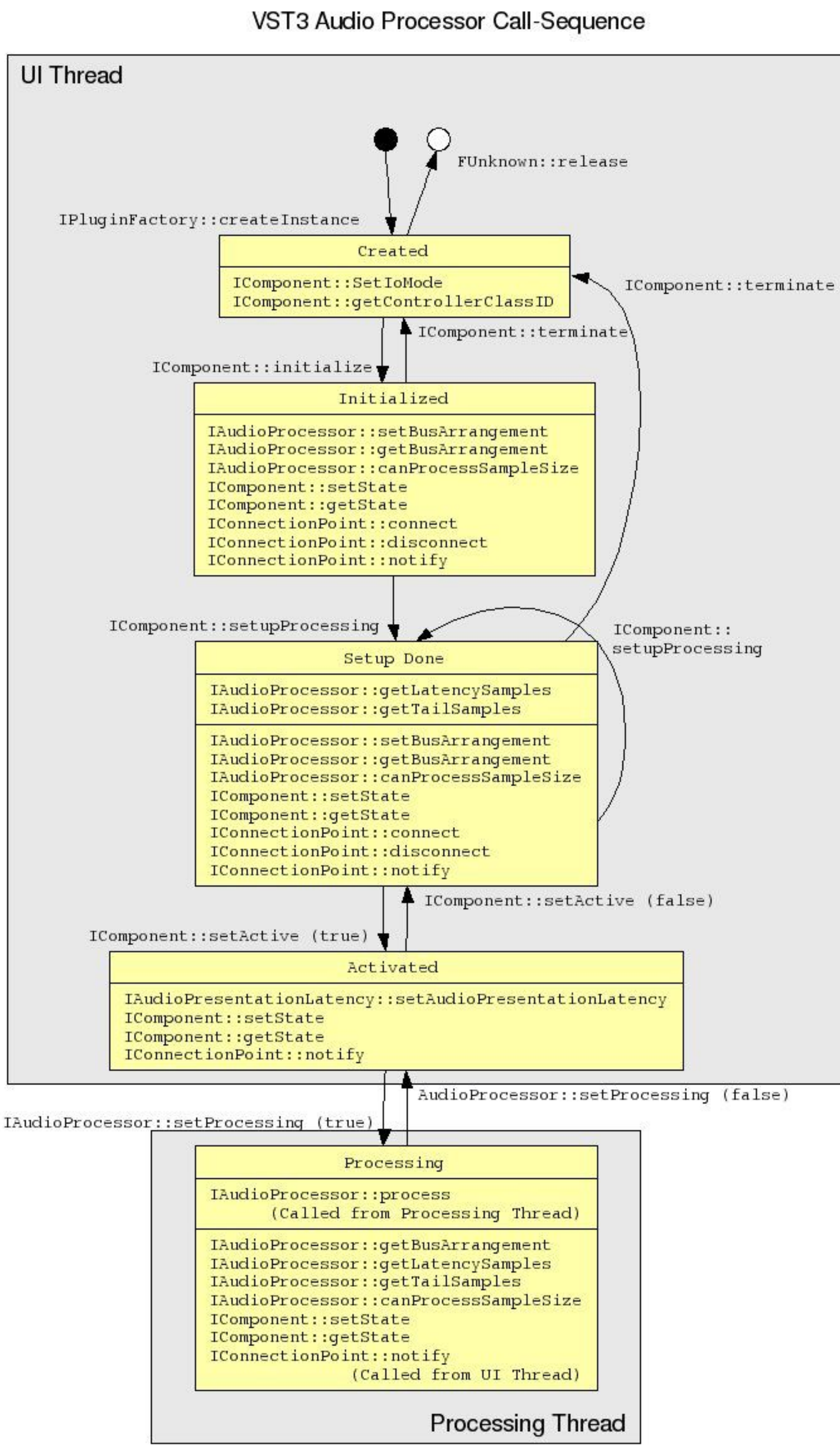

VST3 Audio Processor Call-Sequence
UI Thread
FUnknown::release
IPluginFactory::createInstance
Created
IComponent::SetIoMode
IComponent::getControllerClassID
IComponent::terminate
IComponent::terminate
IComponent::initialize
Initialized
IAudioProcessor::setBusArrangement
IAudioProcessor::getBusArrangement
IAudioProcessor::canProcessSampleSize
IComponent::setState
IComponent::getState
IConnectionPoint::connect
IConnectionPoint::disconnect
IConnectionPoint::notify
IComponent::setupProcessing
IComponent::setupProcessing
Setup Done
IAudioProcessor::getLatencySamples
IAudioProcessor::getTailSamples
IAudioProcessor::setBusArrangement
IAudioProcessor::getBusArrangement
IAudioProcessor::canProcessSampleSize
IComponent::setState
IComponent::getState
IConnectionPoint::connect
IConnectionPoint::disconnect
IConnectionPoint::notify
IComponent::setActive (false)
IComponent::setActive (true)
Activated
IAudioPresentationLatency::setAudioPresentationLatency
IComponent::setState
IComponent::getState
IConnectionPoint::notify
AudioProcessor::setProcessing (false)
IAudioProcessor::setProcessing (true)
Processing
IAudioProcessor::process
(Called from Processing Thread)
IAudioProcessor::getBusArrangement
IAudioProcessor::getLatencySamples
IAudioProcessor::getTailSamples
IAudioProcessor::canProcessSampleSize
IComponent::setState
IComponent::getState
IConnectionPoint::notify
(Called from UI Thread)
Processing Thread

Figure 3.2: VST 3 Process Workflow [17].

3.2.2 Edit Controller

Edit Controller is part of the VST Plug-in that implements `IEditController` interface and handles graphical user interface. It is also responsible for the parameter management. Since the processing component and the edit controller component does not communicate with each other directly, passing the parameters rely on host implementation of `IComponentHandler` interface. The job of the edit controller is to transfer changes of parameters that were triggered by the user action in GUI of the plug-in to the `IComponentHandler` and from that point, it is on VST Host to pass it to the processing component.

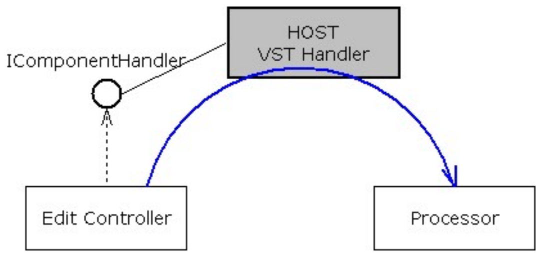


Figure 3.3: VST 3 Standard Communication [17].

3.2.3 Private Communication

It is possible for the two components of VST Plug-in to communicate with each other privately by using `IMessage` interface. This communication is unknown to the VST Host. To allow communication between processor and edit controller component hosting application has to establish a connection between them using the `IConnectionPoint`. Diagram of the communication can be seen in Figure 3.4. From the processor point of view, this communication should be performed outside the process method to not break the real-time audio processing. Separate timer to handle the messaging is advised.

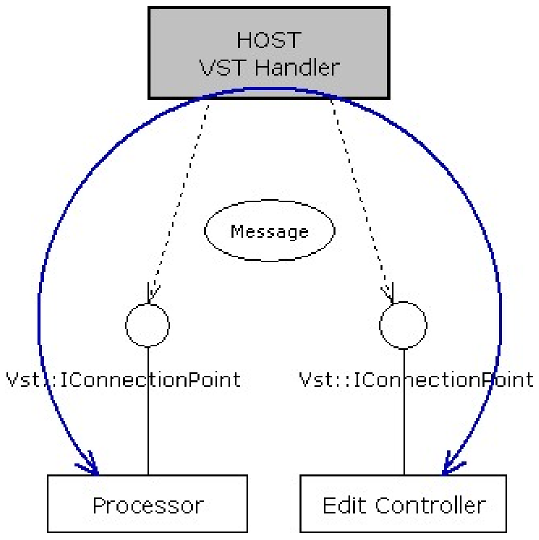


Figure 3.4: VST 3 Private Communication [17].

4 MIDI

MIDI (Musical Instrument Digital Interface) is a technical standard defined originally by the Midi 1.0 specification in 1982 that describes hardware and software for the exchange of musical event messages between digital musical instruments. A wide variety of MIDI instruments is available such as keyboards, sequencers or computers (software instruments) or even lighting instruments. Rather than transporting sampled signal digital messages are sent through the interface that carries information about music such as note's notation, pitch or velocity. MIDI is also used for the synchronization of instruments. To keep all the instruments updated about the current tempo and beat position in time.

4.1 MIDI Interface

MIDI interface consists of three ports – IN, OUT and THRU. IN is the input for of the MIDI devices and allows the device to receive MIDI messages. OUT is used to send MIDI messages to other devices and THRU is used to daisy-chain MIDI devices together. According to Andrew Swift (1997), "At first glance it may seem unnecessary to have the THRU port, since you could simply chain devices together using the output from the OUT port. The digital information is sent serially at a rate of 3000 bytes per second, by switching a current of approximately 5mA on and off. If the OUT port was used this current may be too small to drive any of the devices." [19].

4.2 Windows OS Support

From Windows 2000 onward, the operating system has built-in support for handling MIDI devices and exposes a C language interface for establishing a connection, sending messages, etc. by including the appropriate header file in the application source code. This, as a preferred way for handling MIDI on Windows OS, is how the MIDI message exchange is implemented in the application.

5 Development

The application is developed for personal computers with operating system Microsoft Windows although programming language and used libraries open possibilities for later ports to different operating systems. The program is written in C/C++ with Qt graphical framework and PortAudio (PA) audio I/O library. Since the goal of this work is to develop a DAW, in this chapter objectives are reorganized as a functional, non-functional and graphical requirements for better understanding the implementation process and later on described in a more detailed manner.

5.1 Requirements Analysis

The following requirements were created through the analysis of goals of this paper.

5.1.1 Functional Requirements

Functional requirements of application are organized and described in Table 5.1.

5.1.2 Graphical Requirements

Requirements that are based on user interaction with application. Requirements for controlling program using the graphical user interface (GUI) or displaying information about state of a program are listed in a Table 5.2

5.1.3 Non-Functional Requirements

The only non-functional requirement si to achieve as low latency as possible for the selected device.

5.2 Software Development Model

The development of this application is using the **incremental build model**, which applies the waterfall model incrementally. One iteration of the waterfall model is called increment and it adds new functionality from the requirements. Deployment of software is done at the end of each increment. This helps to evaluate the status of software by examining the feedback from the end-users. Because it deploys partially-functioning software it also avoids long development time.

Number	Name	Description
1	Input/Output Routing	Ability to route selected inputs to selected outputs.
2	Sound Card Output	Ability to use sound card output channels as output.
3	WAV File Input	Ability to use audio files (.WAV) as inputs.
4	VST Plugin	Ability to apply VST Plug-in on selected input.
5	Recording Output	Ability to use file (WAV.) as an output.
6	Playlist Creation	Ability to store information about settings for each song.
7	Add Song to Playlist	Ability to add songs to playlist.
8	Remove Song from Playlist	Ability to remove songs from playlist.
9	Add Line to Song	Ability to add lines to song.
10	Configure Device	Ability to select only used channels in device and assign them custom names. Use these when selecting line's inputs and outputs.
11	Song Control	Ability to control songs. Includes: **Next song**, **Stop song**, **Previous song**, **Play song**.
12	Playlist Save	Ability to save playlist into file (.xml).
13	Playlist Load	Ability to load playlist from file (.xml).
14	Configure Device Save	Ability to save configure device information into file (.xml).
15	Configure Device Load	Ability to load configure device information from file (.xml).

Table 5.1: Functional requirements

The development of the digital audio workstation consists of increments that are created from a group or a single requirement and are analyzed, designed, implemented and tested. At first, the core of application – audio I/O is implemented, the next increment consists of a graphical user interface. The following increments add functionality from the defined requirements resulting in a finished application when all the requirements are met.

5.3 Environment

Proposed application is targeting operating system Microsoft Windows. Libraries used in this project were selected to fit the needs of the application and to fulfill requirements. The application was developed in programming language C++ under the c++11 standard to allow for the usage of the new features introduced in c++11.

Number	Name	Description
1	Menu Bar	Ability to control program functions through menu bar.
2	Top Bar	Ability to access most frequently used functions of the application via buttons on the main window below Menu Bar.
2.1	Play button	Starts the playback.
2.2	Stop button	Stops the playback.
2.3	Pause button	Pauses the playback.
2.4	Backward button	Scroll the timeline backwards.
2.5	Forward button	Scroll the timeline forward.
3	Display Input to Output Line	Ability to display sound inputs to outputs as a line.
4	Display Graph	Ability to display signal as a 2D graph.
5	Control Line	Ability to use GUI elements to make changes to line settings.
5.1	Select I/O	Button for selecting inputs and outputs.
5.2	Mute	Button for muting the line.
5.3	Fx	Button for processing unit selection.
6	Playlist Creation	Ability to store information about settings for each song.
7	Logging Window	Ability to display messages to user with levels of importance defined as **Debug**, **Info**, **Warning**, **Critical** and **Fatal**.
8	Display Playlist	Ability to display playlist and current song.

Table 5.2: Graphical requirements

5.3.1 Compiler

At the beginning of this project, MinGW was used as a compiler. It is an open source software development environment for MS Windows. It contains among others a GCC compiler as well as collection of GNU utilities known as MSYS. During the development, the application switched to MSVC compiler because of MinGW compatibility issues with VST SDK.

5.4 Libraries

5.4.1 Qt Framework

Qt framework is used to create a graphical user interface to allow simple human-computer interaction needed for live music performances. Qt framework contains an integrated development environment called Qt Creator that is used in this project as well as all the required libraries for software development in Qt. Qt also extends the capabilities of C++ with features and mechanics that might not be present in the language itself (depending on the C++ standard in use) such as *Signals and Slots* for communication between objects, variety of containers or guarded pointers. Base class of all objects in Qt is called `Q_OBJECT`. By deriving from this class it is possible to use the Qt features such as Signals and Slots on custom objects.

Life of a QObject

`QObjects` – classes that inherit the `QObject` are organized as a tree. When creating a new object, a pointer to parent can be passed as a parameter creating a parent-child relationship. In this relationship when a parent is being deleted, it sends a signal to the children causing the deletion of them as well and this process cascades to the grand-children and so on. When a child is being deleted it informs the parent and is removed from its children list. When creating objects on the heap using the keyword `new` objects can be created in any order and Qt mechanics take care that the object is destroyed and memory deallocated exactly once when the object is no longer needed, for example when a parent was destroyed. On the other hand, while creating `Q_OBJECT`s on the stack the C++ standard states that the destructors of the local objects are called in the reversed order of their initialization.

5.4.2 FFTW

FFTW is a C subroutine library used for calculations of discrete Fourier transform. Library can be downloaded from the official website as a pre-compiled package with dynamic libraries for 64-bit Windows operating systems. FFTW is selected because the C-style interface fits the C/C++ nature of the application and because it is the ”Fastest Fourier Transform in the West”[7]. It features multi-dimensional transform, parallel transform and is portable to any platform with the C compiler. Description of this library is not in the scope of this paper, further information about the usage can be found in the official documentation[1].

5.4.3 libsndfile

Libsndfile is an audio library created by Erik de Castro Lopo for reading and writing audio files. The library is written in C and supports a variety of formats such as WAV, FLAC, and OGG. Lightweight C++ wrapper is distributed together with the library as one header include that takes care about things like memory management and proper file opening and closing. In the proposed application libsndfile is used only for creating audio files during recording.

5.4.4 PortAudio

PortAudio is an open source audio library written in C that creates cross-platform API above platform-specific native audio APIs like ASIO or ALSA that are called

[1]Available at `http://www.fftw.org/fftw3_doc/`.

Host APIs. Relationships between HostAPIs and PortAudio can be seen in the diagram in Figure 5.1. This allows PortAudio to be used on a range of sound APIs on various operating systems with as little changes to source code as possible. It is low latency and low-level library which is critical when creating an audio application for live performances.

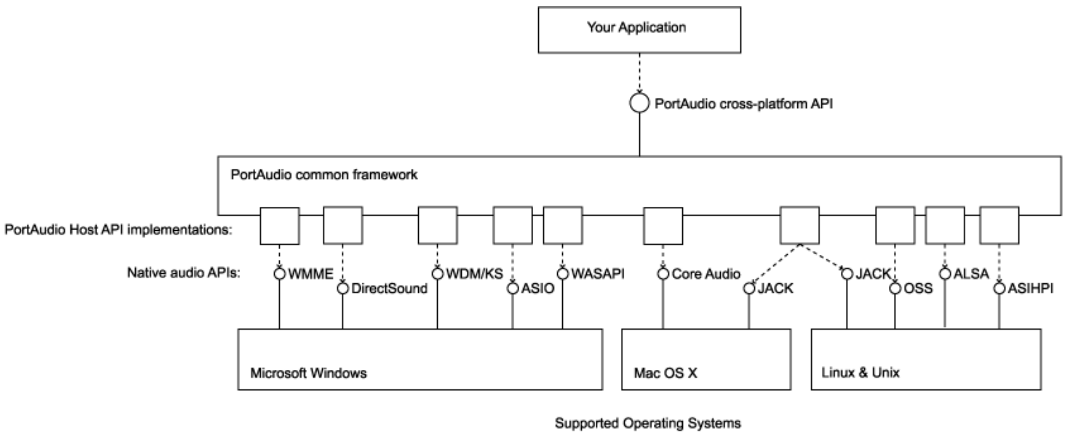


Figure 5.1: Relation between PortAudio API and Host APIs. [15]

PortAudio library processing model works with **Devices** and **Streams**. Devices are hardware audio interfaces like external or internal sound cards accessible on the hosting platform. PortAudio provides utility functions to provide information about devices names and capabilities like supported sample rates and a number of input and output channels. The stream represents an active flow of audio data between application and devices. Streams support both either one of input or output for example in a case of playing an audio file or simultaneous input and output. Stream properties are defined using the `Pa_OpenStream` method parameters. It is possible to define the sample rate of the stream, sample format, buffer size and latency. PortAudio is capable of using 8, 16, 24 and 32-bit integers and 32-bit floating points as a sample format regardless of the Host API. Two possible methods for audio communication between application and stream exists – asynchronous callback that calls user-defined callback function whenever audio data is available or required or synchronous read and write functions.

PortAudio Callback Method

The Callback I/O method is periodically calling a callback function passed in the `Pa_OpenStream` method that implements the `PaStreamCallback` signature. The callback is returning `paContinue` value indicating that stream should continue. By returning `PaComplete` or `PaAbort` values it is possible to deactivate the stream from the callback function. While `PaComplete` cancels the stream after the last buffer has finished playing, `PaAbort` cancels the stream as soon as possible. Because stream callback function is often operating with very high or real-time priority in a special thread or interrupt handler there are some restrictions to the code allowed in the callback to produce glitch-free audio. It is important to avoid operations that can take an unbounded amount of time to execute like memory allocation/deallocation, I/O operations concerning both file system as well as a console I/O, such as `printf` method, mutex operations or OS API function calls.

Read/Write I/O Method

This method, on the other hand, provides a synchronous read/write interface for acquiring and playing audio. This method typically has higher latency then callback method, but it could be used for example from programming languages that do not support asynchronous callbacks. Instead of the callback function, audio data are read by the `Pa_ReadStream` method and written with the `Pa_WriteStream` method. In the case of internal buffers being full, these calls will be ignored making them safe to call in a tight loop. Selection of the method is done via the parameter in `Pa_OpenStream` function call.

Before using PortAudio library it has to be initialized by calling the `Pa_Initialize` function and terminated after usage by calling `Pa_Terminate` function. After successful initialization stream can be opened by calling the `Pa_OpenStream` function. Properties of the stream are set by the following parameters [15].

- `PaStream**` stream
- `const PaStreamParameters *` inputParameters
- `const PaStreamParameters *` outputParameters
- `double` sampleRate
- `unsigned long` framesPerBuffer
- `PaStreamFlags` streamFlags
- `PaStreamCallback*` streamCallback
- `void*` userData

stream is the address of a `PaStream` pointer that will receive a pointer to the opened stream. *inputParameters* and *outputParameters* are structures that define used device, the number of channels, sample format and latency. Optionally they can also set Host API specific data structure containing additional information for the device setup and stream processing. *sampleRate* parameter determines the sample rate of the stream. *framesPerBuffer* defines the number of frames passed to the stream callback function or preferred block granularity for a blocking read/write stream. *streamFlags* are modifiers of the streaming process behavior, for example, disabling clipping of out of range samples or disabling default dithering. Several flags can be combined together using the bitwise or operator (|). *streamCallback* is a pointer to the user-defined function that is responsible for processing and filling input and output buffers. If this parameter is `NULL` stream is opened in a blocking read/write mode. *userData* is a user-defined pointer that is passed to the callback function allowing to access a pointed object in the callback.

Build

To be able to use PortAudio in an application source code was downloaded from the official website and build targetting the demanded audio API, in this case, ASIO. To build the library with the MinGW compiler following commands was used.

```
./configure --with-winapi=asio --with-asiodir=../ASIOSDK2.3
make
```

To build the library with the MSVC compiler, a Visual Studio project file (*.vcx-proj) is distributed with the source code of the library. The project file contains most of the configuration required to compile the library. The project file was opened in Visual Studio 2015 and the library was built. For both cases (MinGW and MSVC) it is required to have the ASIO SDK folder set while compiling the library with ASIO API support. ASIO is a sound card driver protocol created by Steinberg to provide a low-latency interface between application and sound card. ASIO SDK is required to build ASIO support. It is free to download from Steinberg's web-site. This will result in the creation of PortAudio example programs and library file *portaudio_x64.lib* that is linked to the application.

5.5 Implementation

Implementation details of the proposed application are described in this section. It consists of describing the implementation of GUI and implementation of functional features.

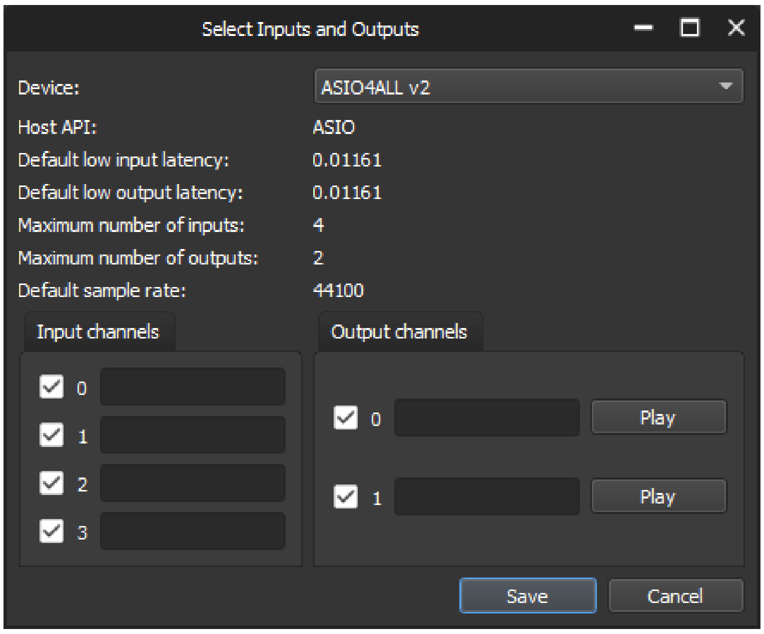
5.5.1 Graphical User Interface

To be able to control the application Qt graphical framework was used to create a suitable GUI. In this section implementation of the graphical part of the application is split into several sections and described in detail. Application window consists of following parts:

Top Menu

Top menu of the application features a set of actions available to the user to control the program using the GUI. The menu consists of the following groups

- File
- Devices


Select Inputs and Outputs

Device: ASIO4ALL v2
Host API: ASIO
Default low input latency: 0.01161
Default low output latency: 0.01161
Maximum number of inputs: 4
Maximum number of outputs: 2
Default sample rate: 44100

Figure 5.2: Configure Devices window.

- Playlist
- Help

File includes a selection to exit the application. **Devices** includes actions connected with sound card settings. First one is configure devices, it triggers a method that displays a list of available sound cards as well as information about the API, latency, number of inputs/outputs and sample rate (Figure 5.2). It also allows a user the select the channels to use and assign custom names to them to ease the future recognition. It is possible to test the outputs using `QPushButtons` assigned to each output channel. This will result in a sine wave played in a selected output channel. Next two items under the devices menu are save devices configuration and load devices configurations, these actions trigger method that loads previously saved configuration from a .xml file or saves the current configuration into the .xml file. **Playlist** menu consists of actions that deal with playlist creation. Namely **New Playlist** that creates a blank playlist, **Load Playlist** – action that loads previously saved playlist, **Save Playlist** – action that saves the current playlist into the .xml file, **Add Song** – Adds a new song into the current playlist and **Add Line** that adds a new `LineWidget` into the song. In the **Help** section it is possible to display information about the application and licence as well as information about the PortAudio version using the **About** item and **About PortAudio** item. All actions from the menu are accessible through the `Alt + [key]` keyboard shortcut, where `[key]` stands for the first letter of the name of the item.

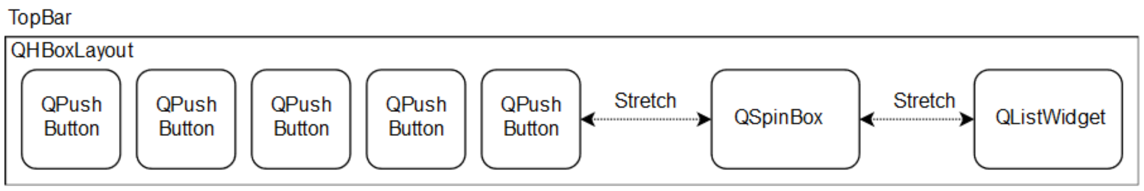


Figure 5.3: TopBar Layout

Top Section

This part serves as the main control area of the application. It derives from the `QWiget` class making it possible to connect it to the rest of the applications using the asynchronous signals and slots mechanism. Deriving from `QWidget` also handles the memory management. When an object derived from `QObject` and is created on the heap using `new` operator i.e.

```
TopBar *topBar = new TopBar(QWidget* parent);
```

it counts the number of references to that object and when the number drops to zero, the object is deleted from memory. This makes implementation easier for the programmer and decreases the chances of memory leaks or dangling pointers.

This class contains five `QPushButton`s with fixed size and each with a custom icon to represent the following actions

- Play
- Stop
- Pause
- Forward
- Backward

When a user presses the button, signal is emitted informing the rest of the application about the action. In the middle of the Top Bar, `QSpinBox` is created to represent tempo of the current song. The area on the right serves to display information about the name of the current playlist using `QLabel` and shows the list of songs in the current playlist using the `QListWidget`. It is possible to shuffle through the songs by clicking on their name. Playlist widget is connected to the rest of the application in a way that it automatically refreshes the list when action is performed that changed the current state of a playlist. Buttons on the right are used to add or remove a song from the playlist. When they are clicked the corresponding signal is emitted. All widgets in the Top Bar are in a `QHBoxLayout` stretched to separate each section.

Middle Section

This is the part visualizing settings for a current song and displaying a graph of a signal in time for each input. It allows a user to define their own routing rules and to apply filters on a signal. Because of variable size of this section the root widget is `QScrollArea` allowing the scrolling behavior in the case that the size of elements inside this widget exceeds its space. Inside the `QScrollArea` there is a dynamic array of `LineWidget`s that refreshes whenever a current song is changed. `LineWidget` is a graphical representation of route from one input to one output. For reasons described earlier it is derived from `QWidget` and serves to apply settings for one line as well as show graph of the signal in time. Parent widget of `LineWidget` is a `QGroupBox` that serves as a container for the widgets inside and enables labeling of the respective `LineWidgest`s to distinguish them from each other. All the widgets inside the `QGroupBox` are sorted using the `QGridLayout`. As seen on Fig. 5.4 `LineWidget` consists of two `QPushButton`s, one opens dialog window (Fig. 5.5) for setting input and output of the line the other one opens a dialog window to select digital signal processing units from an available list. `LineWidget` also has one custom two-state button called `StatePushButton` deriving from `QPushButton` to add the two-state logic to the `QPushButton` using the `QStateMachine`, Qt implementation of a finite state machine. These buttons allows user the select **mute** for the current line. In the middle of `LineWidget` there is a `QTabWidget` a class that provides a stack of tabbed widgets and it is used to display the graphical interface of the currently selected processing units. In the right part, a `QChartView` that included `QChart` was used to display the graph of a signal in time during the playback. Both these components are part of the `QtCharts` module and are not part of the Qt core.

Because of performance issues of these two objects a custom object called `AudioGraphWidget` was later on created for displaying graph of a signal. `AudioGraphWidget` inherits `QWidget` class and overrides the `paintEvent` method. This allows to define a custom look for the derived widget, in this case a graph. Graph is designed to show the n values passed to the widget. n is set during the drawing part based on the width of the widget. To get the basic information about time length of a signal, scale along the x-axis is implemented to display one second and ten second marks in the graph. To display a point in time in the graph, two markers are implemented, visualized as a horizontal line. One is used to display the current point in time where the signal is being processed or played and the second one to allow a user to define a point in time from which the playback should start, typical use case being to start a song from a place other than the beginning. This second marker can be set with a left mouse click. This feature is possible by reimplementing the `mousePressEvent` method of the `QWidget`. Exact time point is calculated from the x-coordinate of the mouse button press and the relative position of the currently displayed signal in the audio file. Values of audio levels and information about the progressing current time line are passed to the object using signals and slots. These signals are connected with the `Queued Connection` option specified, making the slot invoked when control returns to the event loop of the

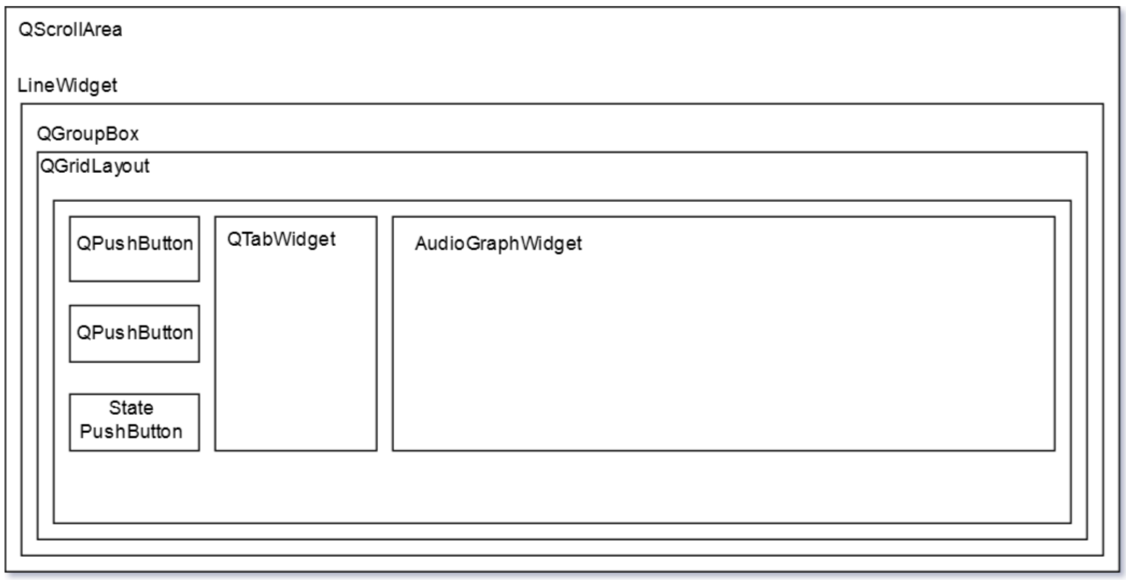


Figure 5.4: Middle section layout.

receiver's thread rather than being performed from the caller's thread.

Bottom section

Bottom section only consists of one custom widget called `LogWindow`. Its purpose is to display information about ongoing events in the application in a form of messages with timestamps and priority levels. `LogWindow` derives from the `QTableWidget` and sorts the messages in a table-like fashion. `LogWindow` is connected to the Qt messaging system `QMessageLogger` in a way that calling a Qt functions like `qDebug`, `qInfo`, `qWarning`, `qCritical`, or `qFatal` anywhere in the application during the lifetime of `MainWindow` will result in a new row in the `LogWindow` containing the message and some additional information. Each row contains information about the **time** of the message, its **level of importance**, **location** of the code that triggered the message including relative path and row number, **context** from which it was called (class and function name) and the **message** itself. This functionality was made possible by creating `LogWindow` as a static object in `MainWindow` class and calling the Qt global directive `qInstallMessageHandler` in the `Main` class of the project with pointer to the method as argument that redirects the messages to the `LogWindow` if it is created, if not standard console output is used to print them.

Global Look

By default, Qt uses the predefined style for the application depending on the operating system and version of Qt. How to change the global look of the application is described in this chapter. Styles for the application are set using the

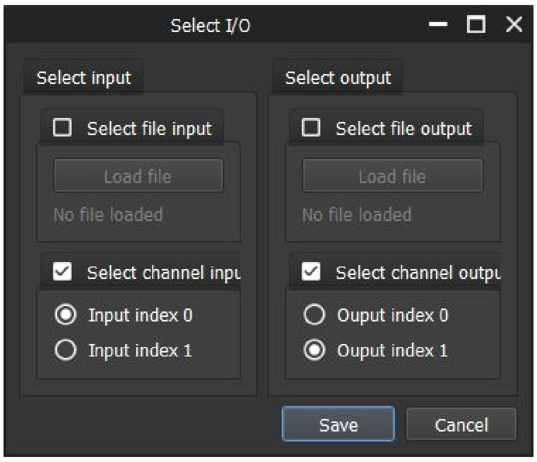


Figure 5.5: Select Line I/O window.

`QApplication::setStyle(QStyle* style)` method, where the `QStyle` is an abstract base class that encapsulates the look and feel of a GUI. In order to implement custom style a class `CustomStyle` is created that inherits `QProxyStyle` which is a convenience class that simplifies dynamically overriding `QStyle` elements. Virtual methods of `QProxyStyle` class `polish(QPalette &palette)` and `polish(QApplication *app)` are implemented. First one sets the colors of the palette while the second one is responsible for loading the custom stylesheet file and setting the stylesheet of the application. Stylesheet file includes the information about the look of Qt widgets. It is similar to the CSS known from HTML.

This results in a change of the application window, but the title bar and frame will remain the same since this is taken care of by the Windows API itself. It is possible to change the style of the frame and title bar of the window by creating a custom top-level `QWidget` that contains the window itself and set its properties using `QWidget::setWindowFlags` to `Qt::FramelessWindowHint` and `Qt::WindowSystemMenuHint` making the top level widget borderless and without frame. This was implemented in a `FramelessWindow` class. A custom top title bar was created using `QToolButtonss` with appropriate logic and look to behave as a **maximalize**, **minimalize** and **close** buttons. `QLabel` is used to display the name of the window.

Application window

Application window is a `FramelessWindow` with `MainWindow` inside. `MainWindow` consists of Top Menu, Top Bar, Middle section and bottom section put together using `QVBoxLayout`. Top Bar and Middle Section are aligned to the top, while `LogWindow` is aligned to the bottom using `QSplitter` making it hideable and resizeable.

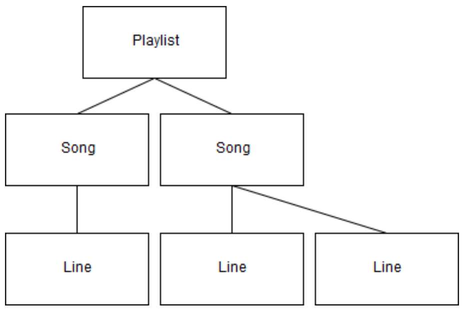


Figure 5.6: Playlist hierarchy.

5.5.2 Features

Implementation of functional requirements is divided into several paragraphs and described in this section.

Routing and Playlist Settings

Application routes desired input to output based on a user selection. Every routing possibility between a set of inputs and set of outputs can be broken down to a set of 1:1 connections between input and output. This connection is called *Line* and its implemented in a class `Line`. It stores information about the input and output of a Line in a form of struct `LineIOInfo`. `LineIOInfo` defines a type of the I/O and information depending on the type, path the to file in the case of audio file or id of a channel in the case of a sound card. `Line` class also contains information about id and used signal processing units as well as a set of functions connected with Line features.

One step above in the playlist hierarchy is a song. It is implemented in a class `Song` and contains a `std::vector<Line*>` representing all the routing settings as well as the name of and tempo of a song.

On top of settings hierarchy is a playlist implemented in a `Playlist` class. It stores songs in a form of `std::vector<Song*>`, playlist name and set of functions to ease working with objects in playlist hierarchy. Example of playlist structure can be seen in Figure 5.6.

Saving and Loading Playlist Settings

To be able to store the current playlist or load previously created playlist in the form of .xml file a `PlaylistXML` class was created that implements this functionality. It takes a pointer to the `Playlist` object as a parameter and either creates a .xml file based on the current playlist using the `QXmlStreamWriter` or loads playlist settings from a .xml file using `QXmlStreamReader` while checking for errors in the file. Example of playlist settings from Figure 5.6 is shown below. Playlist consists of two songs; the first song has one Line connecting input channel on index 0 to output channel on index 0. The second song has two Lines, first one connecting input channel 0 to output channel 1 and second Line has an input file of given path and output on output channel 0.

```
<?xml version="1.0" encoding="UTF-8"?>
<playlist>
    <name>Concert</name>
    <songs>
        <song id="0" name="First␣song" bpm="120">
            <line id="0">
                <input type="Channel">
                    <channel id="0"/>
                </input>
                <output type="Channel">
                    <channel id="0"/>
                </output>
            </line>
        </song>
        <song id="1" name="Second␣song" bpm="120">
            <line id="0">
                <input type="Channel">
                    <channel id="0"/>
                </input>
                <output type="Channel">
                    <channel id="1"/>
                </output>
            </line>
            <line id="1">
                <input type="File">
                    <file path="C:/example.wav"/>
                </input>
                <output type="Channel">
                    <channel id="0"/>
                </output>
            </line>
        </song>
    </songs>
</playlist>
```

Audio Processing

In this paragraph is described implementation of reading audio from multiple channels or audio files, processing the signal and outputting the result on audio channel or file. To be able to process audio from the input channel to the output channel with minimal latency PortAudio library is used. It was compiled from source code (available on the official website) targeting ASIO into a dynamic (.dll) library.

In order to use PortAudio functionality in application, it has to be initialized first. A class called `PaHandler` was created to serve as a C++ wrapper for initializing and terminating PortAudio according to the lifetime of the application. It also contains functions to handle possible errors and getting additional information about available devices and APIs.

Handling audio in the application is divided into following steps that are described below.

1. Obtaining audio
2. Processing audio
3. Outputting audio

Obtaining audio

There are two types of possible audio inputs, sound card input channel or an audio file. To be able to access audio data from the sound card using PortAudio a PortAudio Stream is open and started. This is implemented in the `Wire` class. First part defined in `Wire::open` consists of extracting information about the current song settings and configuration from the `Playlist` object. Then the information is used to set `PaStreamParameters` structure together with sample format and suggested latency. If `PaStreamParameters` passes the compatibility check stream is opened using defined settings. After the successful opening of a stream, it is started using the `Wire::start` method. PortAudio callback method will be trigging from now on whenever audio is acquired. This gives direct access to the input audio buffers.

Audio File Reading

Since PortAudio callback function cannot contain operations like reading from a hard drive as discussed in section 5.4.4 a class `AudioReader` was implemented to solve this problem. In the phase of opening stream `Wire::open` list of `AudioReader` objects are created, one for every distinct input file in the current song. Afterwards, `AudioReader::openFile` function is called that checks the integrity of the file, initializes the file and extracts the sample rate and channel information of the audio file and prepares the object for reading audio data. Next `AudioReader::startFileRead` function is called that allocates memory for a ring buffer based on a sample rate, number of channels and type of format that is used to allocate enough space for given number of seconds of data rounded to the nearest highest power of 2. After the successful initialization of ring buffer a function `AudioReader::fromFileToBuffer` is started in a separate thread and consists of endless loop with logic described on Figure 5.7. This will keep the ring buffers filled with audio data from the files. In the PortAudio callback function these ring buffers are available for reading using the `AudioReader::readBuffer` method and unlike direct I/O access does not violate callback restrictions.

Ring buffer or circular buffer is a data structure of a fixed size that behaves as if memory is contiguous and circular in nature. It is usually used in buffering data streams. It consists of a buffer of fixed size and two pointers: one head pointer that advances when data is added and a tail pointer that advances when data is removed. This is used to transport data between two different execution contexts without requiring the use of any locks.

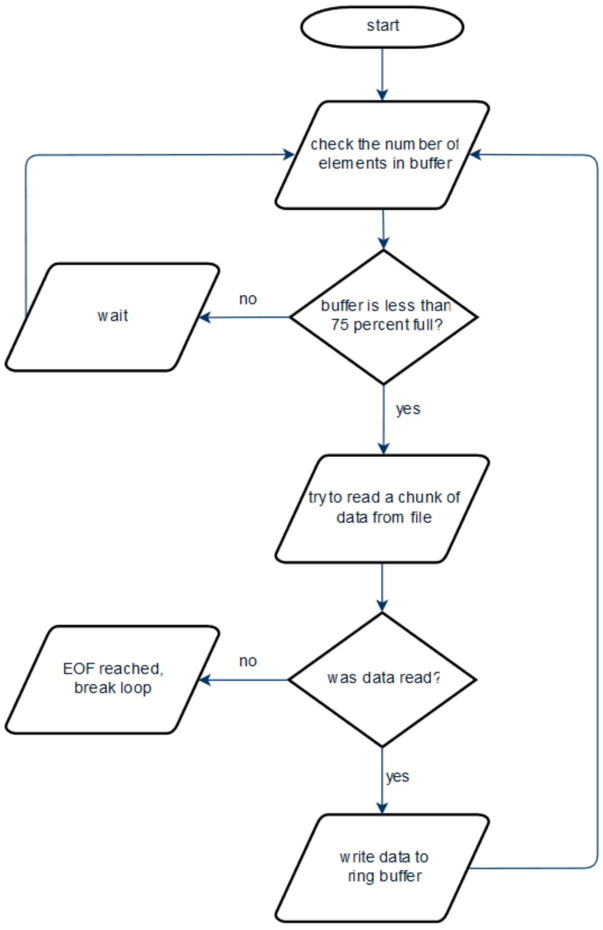

start
check the number of elements in buffer
buffer is less than 75 percent full?
no
wait
yes
try to read a chunk of data from file
was data read?
no
EOF reached, break loop
yes
write data to ring buffer

Figure 5.7: Audio file reading thread workflow diagram.

An alternative approach for reading audio files is implemented to support the GUI feature that shows a graph of a signal in time. An audio file is loaded into memory using the `QAudioDecoder` from the Qt Multimedia module that invokes the callback that fills in buffers whenever data is available. This method enables to set the desired format for the data samples while resampling or converting when necessary. When all the raw data are saved `QAudioDecoder` emits the `finished` signal that is connected to a lambda that calculates the root mean square and passes the data to `AudioGraphWidget` that displays the graph of the audio file.

Processing Audio

This part describes the methods of processing audio in the proposed application. Processing is done in the PortAudio callback function and consists of several steps. First input buffer with a size of `framesPerBuffer` × number of Lines of the current Song is filled with audio data from the sound card input channels or audio files with the defined sample format and sampling rate. Audio data from sound card are available via `const void *input` pointer passed to the PortAudio callback as an argument and audio file data are accessed with the `AudioReader::readBuffer` function. In the next step user selected processing units will run for each Line. After the processing has been done an average value of the buffer is calculated for each Line and added to the array that contains the values for displaying a graph of a signal in `LineWidget`. This is done to lower the number of calls for updating the graph making the application less demanding on CPU and GPU.

Process Unit Interface

To be able to implement new signal processing features to the application in an easy way an interface was created that all the DSP units inherit. It is called `IProcessUnit` and objects that inherits this interface are referred to as **Process Units**. It defines several methods that have to be implemented by the Process Units most importantly `IProcessUnit::init` function that is called in the `Wire::open` method that should contain memory allocation and initialization of the Process Unit or other operations that should not be in the PortAudio callback as discussed in previous chapter. `IProcessUnit::process` function is called in the PortAudio callback with the custom `ProcessData` structure as an argument that contains `AudioData` and `EventData` structures. `AudioData` represent pointers to the array of the used sample format and their length while `EventData` includes a list of non-audio data called `Events` as a general mean of communication between Process Unit and the application. `IProcessUnit::process` method should be restricted to the mathematical calculations on the `ProcessData`. To be able to control Process Unit using a graphical user interface it can inherit `QWidget` class from Qt to create a custom GUI.

A user can select a set of Processing Units to apply on the selected Line using the *Fx* button in the `LineWidget`. This action will fill the `QTabWidget` in `LineWidget` with the selected Process Units GUIs with each tab representing one.

Process Units

Three Process Units were created in this application – Volume Process Unit, VST Process Unit, and Tempo Process Unit. Their implementations are discussed in this chapter.

Volume Process Unit

`VolumeProcessUnit` was created to control to the volume of a signal on a given Line. In the constructor basic parameters are set and GUI objects are created. GUI consists of a label with the name of the Process Unit and `QSlider *slider` with the range set to [0:100] to give user ability to control the volume. `QSlider::volumeChanged` signal is connected using the lambda function to change the `volume` variable as follows

```
    volume = slider->value / 100;
```

In the `VolumeProcessUnit::process` method input audio data are multiplied with the `volume` variable resulting in the decrease of volume on selected Line. Can be seen in Figure 6.1.

VST Process Unit

This Process Unit allows a user to load a VST Plugin and apply it on the selected Line. To be able to load a VST Plugin, a VST Host had to be implemented. VST 3 Plug-ins are distributed as a single .dll file on Microsoft Windows. VST Process Unit takes a path to the VST Plugin as an input selectable through a graphical user interface of the Process Unit. First, the .dll file containing VST Plugin is loaded from which we obtain a pointer to the `PluginFactory`. This pointer is passed to the helper class named `PlugProvider` that inherits `FObject` and `IPlugProvider` classes. `PlugProvider::setupPlugin` method is called that handles the following actions.

1. Creates and initializes Processor part of the Plugin.
2. Creates and initializes Edit Controller part of the Plugin.
3. If both parts are successfully initialized, they are connected using `ConnectionProxy` helper class that implements `IConnectionPoint` interface. This allows communication between the two parts of the Plugin.

In the next step properties such as height, width, resizing policy or window name are extracted from the Edit Controller and new window with Plugin is created by calling WinAPI function `CreateWindowEx`. Afterwards `AudioClient::create` method is called with Processor and Edit Controller pointers as parameters. `AudioClient` class is implementation of `IComponentHandler` and few other interfaces. `AudioClient` works in following steps.

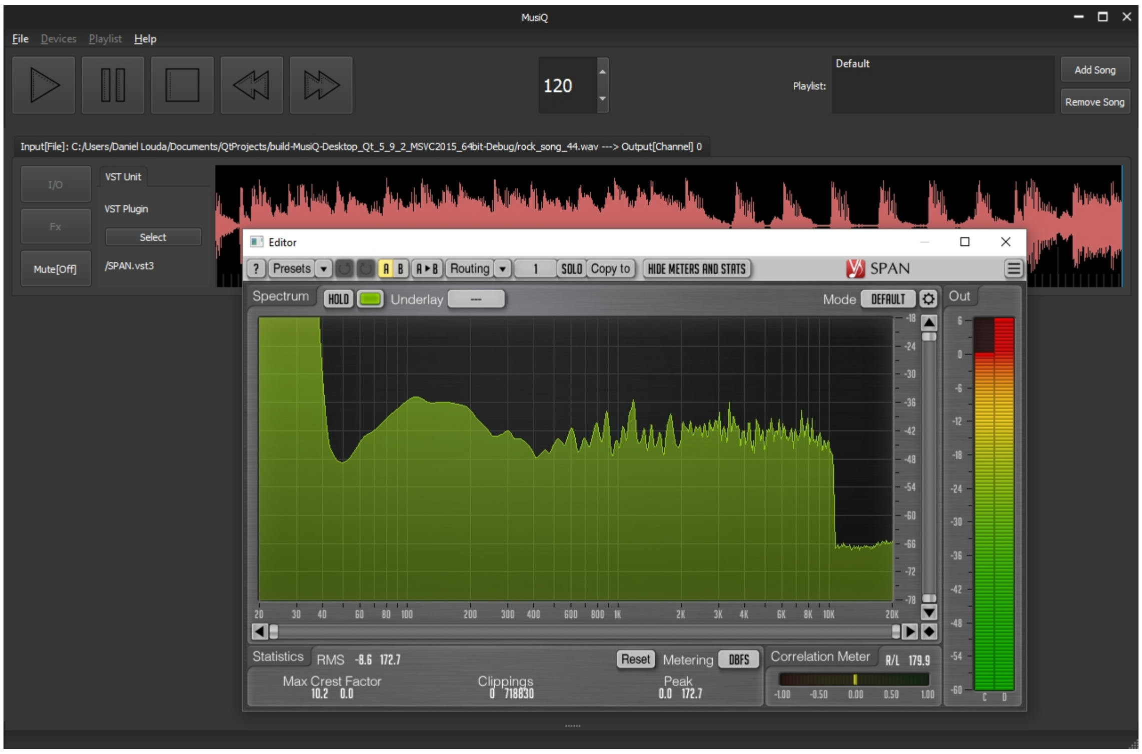


Figure 5.8: Application running a VST Plug-in called SPAN[8].

1. sets itself as a component handler of the Edit Controller part
2. initializes `ProcessData` and `ProcessContext` structures
3. registers input and output audio ports
4. initializes the buffers
5. sets the sampling rate and block size
6. creates `ProcessSetup` structure with defined settings to setup processing
7. activates and starts processing of the plugin

At this point, VST Plugin is ready to process data. Processing is done in three steps. Preprocess step – assign bus buffers and transfers parameter changes from the queue to the `ProcessData`. Process step – calls the Plugin method to process data with `ProcessData` argument. Postprocess – transfer parameter changes from the processor and passes them to the edit controller and unassign bus buffers. Example of application running VST Plug-in called SPAN[2] can be seen on Figure 5.8.

[2]SPAN is a free real-time fft audio spectrum analyzer plugin for professional music production applications. Available at https://www.voxengo.com/product/span/

Tempo Process Unit

For detecting tempo of a song in real-time a spectral-based novelty function described in 2 is implemented from which tempo information is derived using Fourier coefficients. First audio is windowed using appropriate Hann window function and FFT is computed using the FFTW library. Magnitudes of the results are combined together creating spectrogram. In the next step, spectrogram is normalized and split into several frequency bands to increase the robustness of the algorithm. Spectral-based novelty function is computed for each band separately. Obtaining the novelty curve consists of following steps. Appling logarithmic compression to enhance weak spectral components. Smooth differentiator filter f_d of length len is computed by using Hann window w of length len satisfying following equation for $n = [0 : len]$.

$$f_d(n) = \begin{cases} -1 \times w(n) & \text{for } n < len/2 \\ 0 & \text{for } n = len/2 \\ 1 \times w(n) & \text{for } n > len/2 \end{cases} \tag{5.1}$$

len is calculated to be the closest odd number to represent 0.3 seconds given the current sampling rate. By convoluting the band data with differentiator filter smoothed discrete derivative is obtained. Since we are only interested in positive values that correspond to the note onsets negative values are set to zero. Because of convolution result is bigger by $len - 1$, therefore, edge values are removed to fit the original size. In the next step, band is normalized and summed to obtain band novelty curve. Average of band novelty curves are computed thus creating spectral-based novelty curve of an input signal. The result is further enhanced by subtracting the local average. Local average L_{avg} is implemented as convolution of the novelty function $\Delta_{Spectral}$ and averaging filter f_a of length $wlen$ defined as

$$f_a(n) = w(n) / \sum_{m=0}^{wlen} w[m] \tag{5.2}$$

for $n = [0 : wlen]$ where w is a Hann window of length $wlen$.

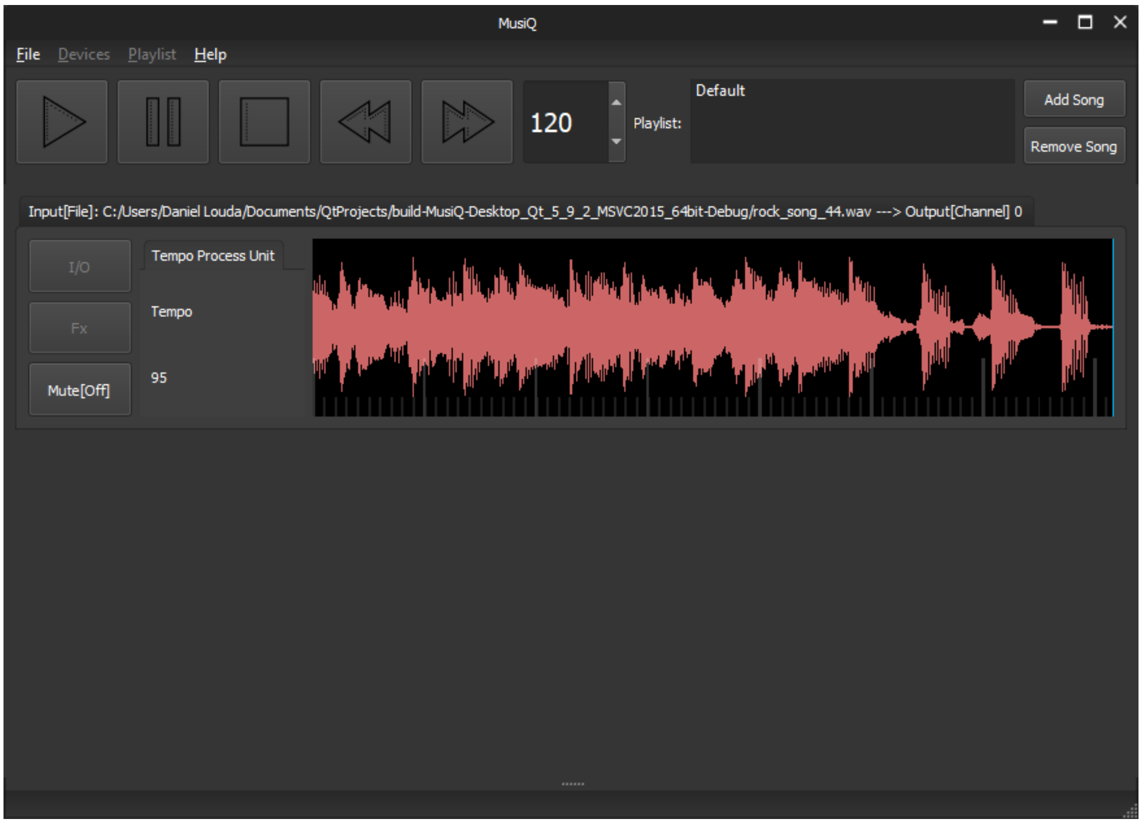
$$L_{avg} = \Delta_{Spectral} * f_a \tag{5.3}$$

The final form of novelty curve $\hat{\Delta}_{Spectral}$ is created by subtracting the local average from the novelty curve and setting negative values to zero for each time frame $n (n \in \mathbb{Z})$.

$$\Delta'_{Spectral}(n) = \Delta_{Spectral}(n) - L_{avg}(n) \tag{5.4}$$

$$\hat{\Delta}_{Spectral}(n) = \begin{cases} \Delta'_{Spectral}(n) & \text{for } \Delta'_{Spectral}(n) > 0 \\ 0 & \text{for } \Delta'_{Spectral}(n) \leq 0 \end{cases} \tag{5.5}$$

In the next step, the novelty curve $\hat{\Delta}_{Spectral}$ is used to determine the tempo of the song. For the given set of tempi in this case [30:600] corresponding frequencies are



Input[File]: C:/Users/Daniel Louda/Documents/QtProjects/build-MusiQ-Desktop_Qt_5_9_2_MSVC2015_64bit-Debug/rock_song_44.wav ---> Output[Channel] 0

Figure 5.9: Application running a Tempo Processing Unit. Tempo estimated as 95 bpm.

obtained using the equation 2.21. For each frequency and time index, sine and cosine functions are computed and multiplied with windowing function of length that corresponds to 8 seconds and signal to obtain a coefficient that represents a level of similarity between the tempo and the novelty function. By finding the index with the highest magnitude of the coefficient the most dominant tempo in the given time frame is found. Alternatively, the autocorrelation method could be used to determine delay with the biggest similarity and from that compute the tempo.

This method can take enough time to break the real-time audio processing of the application. Because of that, computations are performed in a different thread approximately once per second. After the tempo information is extracted, GUI is updated to display the current value. Image of application analyzing the tempo from a song can be seen in Figure 5.9.

MIDI Process Unit

This module is developed to send MIDI messages to the MIDI device during the playback in the event of detecting onset. The latency of MIDI messages after an onset happened should be as low as possible to allow the module to be used to synchronize MIDI devices by transforming a music signal into MIDI events in a real-time fashion.

Onsets are detected on-the-go during playback using the Energy-based Novelty described in Chapter 2. Detection of onsets is possible by the analysis of a discrete derivative of a windowed local energy function. Hann window of length 32 is used. This approach has the benefit of detecting onset more efficiently regarding the CPU usage then the spectral-based novelty because no FFT is being calculated. The second benefit is that this method does not need to evaluate as many samples per invocation resulting in an ability to respond to onsets faster. In this implementation Energy-Based novelty is calculated once per PortAudio callback method that has the buffer size B_s of 512 samples per channel. Using the 44100 Hz sampling rate this results in the detection of onset with delay T_d of approximately 12ms.

$$T_d = \frac{B_s}{f_s} \tag{5.6}$$

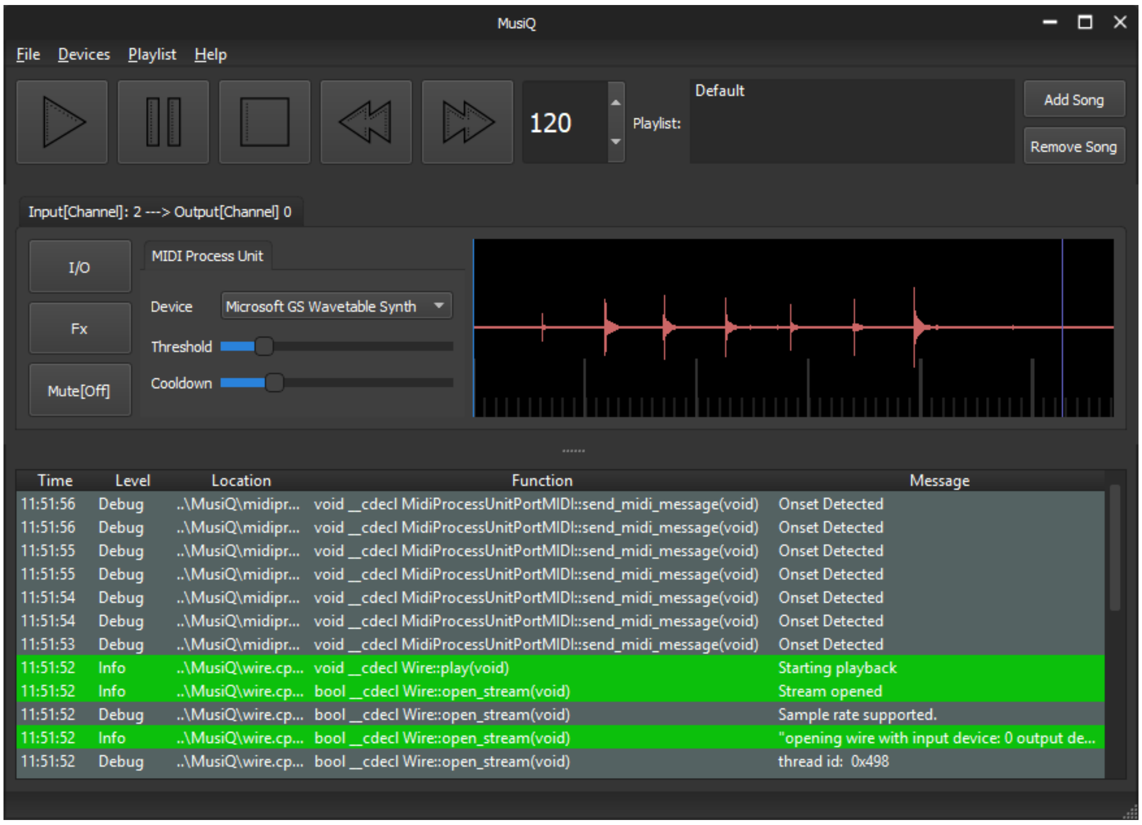
$$T_d \approx 0,0116s \tag{5.7}$$

The downside of this approach is bad performance when detecting onsets of instruments that have a subtle attack of the note. Best results are expected when used with percussion instruments or instruments with a steep attack of the note.

Necessary library calls used in this module are available through the Windows API by including `<Mmsystem.h>`. Functions `midiOutGetNumDevs` and `midiOutGetDevCaps()` are used for listing devices and determine their capabilities, `midiOutShortMsg()` is used to send the MIDI message. To open the device `midiOutOpen()` and `midiOutClose()` to close the device after playback ends. Target MIDI device is selected using `QComboBox` with all the available MIDI devices listed. The module is enhanced by introducing two variable parameters changeable by corresponding `QSlider` in the GUI part of the `ProcessUnit`. One parameter is **cooldown** and controls the time after onset was detected in which another onset cannot occur. The other one is **threshold** to set the limit on the y-axis in Energy-Based Novelty curve above which it is determined as an onset. After the onset is detected a pre-defined MIDI message is sent to the selected device. Image of application running this process unit can be seen on Figure 5.10. Detected onsets are logged in the `LogWindow`.

Outputting Audio

In the next step, audio data are outputted either on soundcard output channels using the pointer to the output buffer in the PortAudio callback method or written down to a file using a `AudioReader::writeToBuffer` function. Hard drive I/O operations are forbidden in the PortAudio callback method, therefore, the following algorithm was implemented to enable writing audio data into a file. During the `Wire::open` method `AudioReader` instance is created for every distinct output file. Afterwards `AudioReader::createFile` method called with arguments that specify the name and format of the file, sample type and sample rate. In the next step `AudioReader::startFileWrite` function is called that initializes



Input[Channel]: 2 ---> Output[Channel] 0

Device: Microsoft GS Wavetable Synth

```
Time      Level   Location              Function                                                   Message
11:51:56  Debug   ..\MusiQ\midipr...    void __cdecl MidiProcessUnitPortMIDI::send_midi_message(void)  Onset Detected
11:51:56  Debug   ..\MusiQ\midipr...    void __cdecl MidiProcessUnitPortMIDI::send_midi_message(void)  Onset Detected
11:51:55  Debug   ..\MusiQ\midipr...    void __cdecl MidiProcessUnitPortMIDI::send_midi_message(void)  Onset Detected
11:51:55  Debug   ..\MusiQ\midipr...    void __cdecl MidiProcessUnitPortMIDI::send_midi_message(void)  Onset Detected
11:51:54  Debug   ..\MusiQ\midipr...    void __cdecl MidiProcessUnitPortMIDI::send_midi_message(void)  Onset Detected
11:51:54  Debug   ..\MusiQ\midipr...    void __cdecl MidiProcessUnitPortMIDI::send_midi_message(void)  Onset Detected
11:51:53  Debug   ..\MusiQ\midipr...    void __cdecl MidiProcessUnitPortMIDI::send_midi_message(void)  Onset Detected
11:51:52  Info    ..\MusiQ\wire.cp...   void __cdecl Wire::play(void)                                  Starting playback
11:51:52  Info    ..\MusiQ\wire.cp...   bool __cdecl Wire::open_stream(void)                           Stream opened
11:51:52  Debug   ..\MusiQ\wire.cp...   bool __cdecl Wire::open_stream(void)                           Sample rate supported.
11:51:52  Info    ..\MusiQ\wire.cp...   bool __cdecl Wire::open_stream(void)                           "opening wire with input device: 0 output de...
11:51:52  Debug   ..\MusiQ\wire.cp...   bool __cdecl Wire::open_stream(void)                           thread id:  0x498
```

Figure 5.10: Application running a MIDI Processing Unit.

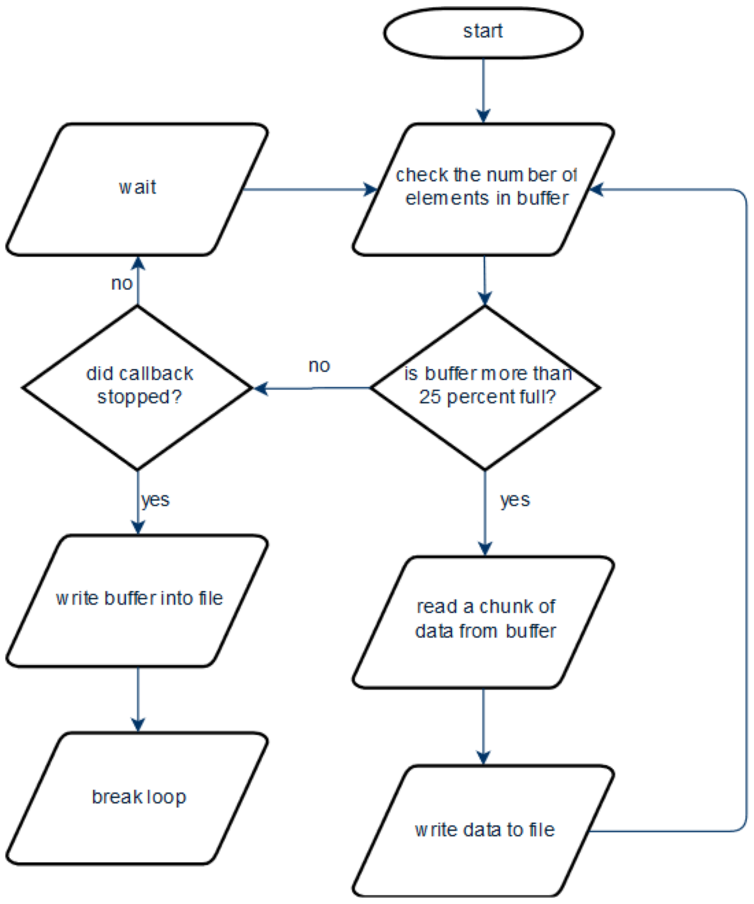

start
wait
check the num ber of elements in buffer
no
did callback stopped?
no
is buffer more than 25 percent full?
yes
yes
write buffer into file
read a chunk of data from buffer
break loop
write data to file

Figure 5.11: Audio file writing thread workflow diagram.

the ring buffer and allocates the memory size of a given number seconds of audio based on the configuration. Next, a separate thread is created to run the `AudioReader::fromBufferToFile` function that contains an endless loop following the logic described in the workflow diagram in Figure 5.11.

Closing of the stream

When the song has finished, the callback function is stopped using the `Wire::close` function that calls the `Pa_CloseStream` method to close the stream. After the stream finishes `Wire::paStreamFinished` method will be triggered as was defined during the opening of the stream. This method will inform output file writers that the callback has finished. Managing the closing of the output audio files is done automatically by destroying the `AudioReader` object.

5.6 Debugging

While encountering misbehavior of the application on a local computer during the development, debugging takes place in the Qt Creator IDE. When using the (nondefault) MSVC compiler appropriate debugger has to be set in the settings. For this project, a CDB debugger from the Windows SDK is used.

For debugging of the application outside of the Qt Creator IDE, possibly on the user's computer, it is useful to generate the Program Debug Database (*.pdb) file while building the application. File generation is enabled with the `\Zi` option of MSVC compiler and contains debugging symbols for the current build of the application. It is used together with the crash dump to obtain information like the source code location or current state of data structures often necessary to identify the cause of the error. WinDbg application from Microsoft was used for a crash dump examination on Windows OS.

5.7 Deployment

To be able to distribute an application across computers a `release` build is created. Unlike `debug` build it does not contain Program Debug Database (*.pdb) files and has more strict memory access policies. To resolve the dependencies on Qt libraries outside the QtCreator IDE a `windeployqt` tool is used. It is a CLI application designed by Qt to automate this process. The following command will copy the Qt-related dependencies as dynamic libraries into the folder with the executable.

```
windeployqt --release <path-to-app-binary>
```

When using MSVC compiler it is important to have the proper environment variables (e.g. `$VCINSTALLDIR`) set for the `windeployqt` to resolve the MSVC-related dependencies.

6 Results and Conclusion

6.1 Libraries

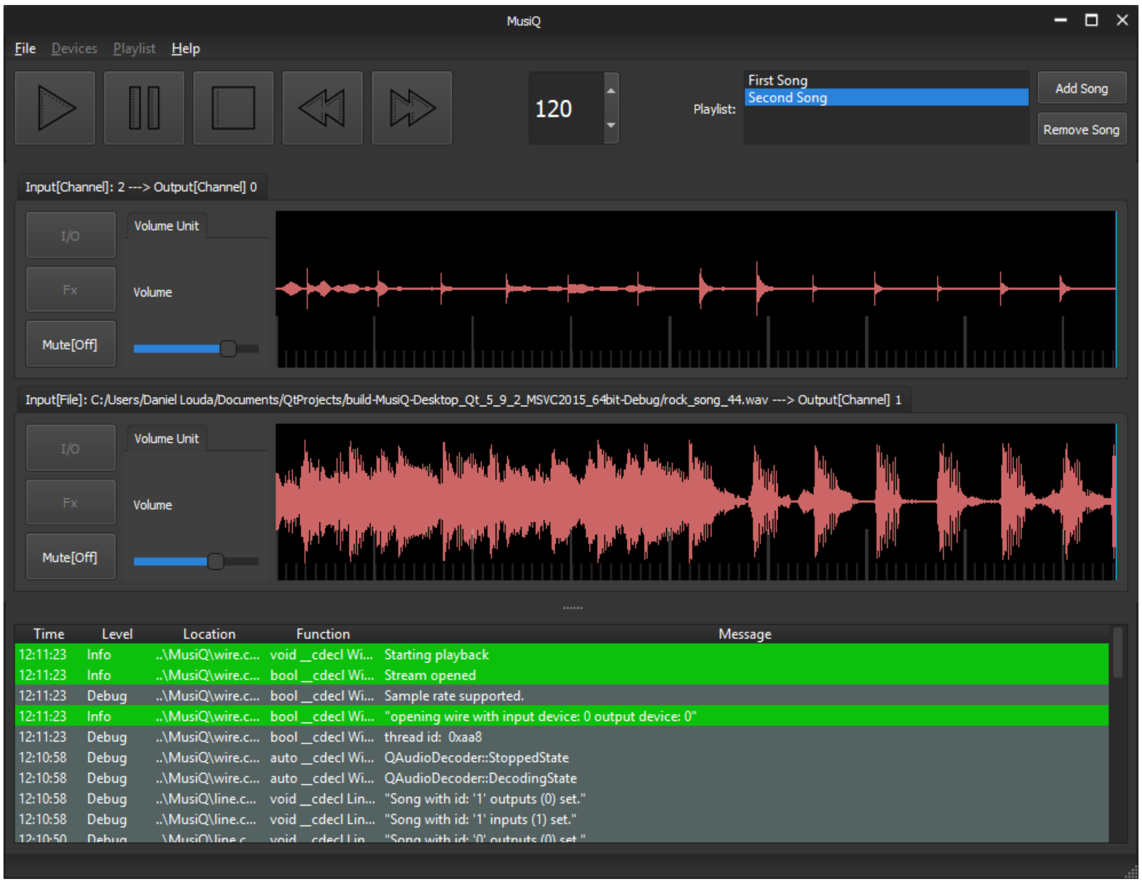
It was possible to fulfill the application requirements using described libraries. Qt framework consists of numerous of both GUI-related and non-related features that can speed up the process of software development. Qt has a modern interface with the addition of support for features from the new C++ standard. FFTW library for fast Fourier transform was working as expected with a reasonable time of FFT calculation and therefore capable of being used in the real-time application. Audio file import and export library libsndfile satisfied the requirements on continuous file writing and format support. It also includes a header only C++ wrapper to isolate the programmer from the C interface and memory management. For the sound card I/O PortAudio library was used because it is capable of low latency routing and supports ASIO. The only downside of using PA is an C-style interface. VST SDK was used to implement the capability of applying VST Plug-in to a selected input. This was the most time-consuming part of the development since creating VST Host for VST version 3 and above is not very well documented and requires good knowledge of how VST works. Apart from VST Host, all other libraries are very well documented often with examples included. Described library stack is more error-prone and has a steeper learning curve than using only features from one framework like Qt or Juce so in a case where a short time to market is essential it is not recommended. On the other hand, this approach offers better modularity and good results.

6.2 Application

A digital audio workstation for operating system Microsoft Windows that specializes in live performances was developed in C++ and tested on an internal sound card and external sound card Creative EMU 0404 USB[1]. Application features real-time multi-channel playback and recording routing on the selected ASIO device with the lowest possible latency supported by the sound card. Usage of audio files as input, displaying a graph of a signal for every input. Ability to create, save and load playlist and configuration settings in a form of a .xml file. Interface for digital signal processing units was designed and four Processing Units were implemented that can

[1]http://www.creative.com/emu/products/product.aspx?pid=15185

be used on the user selected input. Volume Processing Unit to control the volume of the signal. VST Processing Unit that allows the use of the open standard for application independent DSP blocks called VST Plug-ins, Tempo Processing Unit that implements a state-of-the-art method for extracting tempo information from the musical piece using spectral analysis and MIDI Processing Unit that response to detected onsets with MIDI messages. The application is controlled by a customized graphical user interface using Qt framework. Example of the application running settings from Figure 5.6 can be seen in Figure 6.1.



Input[Channel]: 2 ---> Output[Channel] 0

Input[File]: C:/Users/Daniel Louda/Documents/QtProjects/build-MusiQ-Desktop_Qt_5_9_2_MSVC2015_64bit-Debug/rock_song_44.wav ---> Output[Channel] 1

```
Time      Level   Location            Function            Message
12:11:23  Info    ..\MusiQ\wire.c...  void __cdecl Wi...  Starting playback
12:11:23  Info    ..\MusiQ\wire.c...  bool __cdecl Wi...  Stream opened
12:11:23  Debug   ..\MusiQ\wire.c...  bool __cdecl Wi...  Sample rate supported.
12:11:23  Info    ..\MusiQ\wire.c...  bool __cdecl Wi...  "opening wire with input device: 0 output device: 0"
12:11:23  Debug   ..\MusiQ\wire.c...  bool __cdecl Wi...  thread id:  0xaa8
12:10:58  Debug   ..\MusiQ\wire.c...  auto __cdecl Wi...  QAudioDecoder::StoppedState
12:10:58  Debug   ..\MusiQ\wire.c...  auto __cdecl Wi...  QAudioDecoder::DecodingState
12:10:58  Debug   ..\MusiQ\line.c...  void __cdecl Lin...  "Song with id: '1' outputs (0) set."
12:10:58  Debug   ..\MusiQ\line.c...  void __cdecl Lin...  "Song with id: '1' inputs (1) set."
12:10:50  Debug   ..\MusiQ\line.c...  void __cdecl Lin...  "Song with id: '0' outputs (0) set."
```

Figure 6.1: Example of application running settings from Figure 5.6.

6.3 Future Work

The application can be enhanced in the future implementing new Processing Units that could be useful during live performances. For example noise reduction, microphone feedback loop reduction, frequency filtering or pitch correction which are all challenging signal processing problems. Concerning tempo analysis, it can be tested in the future with a public database for comparison with other algorithms or improved by detecting phase of the beat.

List of Figures

List of Tables

Bibliography

[1] *Ableton*. en. URL: https://www.ableton.com/en/shop/live/ (visited on 07/20/2018).

[2] *Ableton Reference Manual Version 10*. en. URL: https://www.ableton.com/en/manual/welcome-to-live/ (visited on 07/20/2018).

[3] J. P. Bello et al. “A Tutorial on Onset Detection in Music Signals”. In: *IEEE Transactions on Speech and Audio Processing* 13.5 (Sept. 2005), pp. 1035–1047. ISSN: 1063-6676. DOI: 10.1109/TSA.2005.851998.

[4] Nick Collins. “A Comparison of Sound Onset Detection Algorithms with Emphasis on Psychoacoustically Motivated Detection Functions”. In: (Jan. 2012).

[5] Eric D. Scheirer. “Tempo and beat analysis of acoustic music”. In: *The Journal of the Acoustical Society of America* 103 (Feb. 1998), pp. 588–601. DOI: 10.1121/1.421129.

[6] Simon Dixon. “Evaluation of the Audio Beat Tracking System BeatRoot”. In: *Journal of New Music Research* 36 (Mar. 2007), pp. 39–50. DOI: 10.1080/09298210701653310.

[7] *FFTW Benchmark*. en. URL: http://www.fftw.org/%20http://www.fftw.org/benchfft/ (visited on 03/29/2019).

[8] *Free Spectrum Analyzer Plugin, FFT, Real-Time [VST, AU, AAX] - SPAN / Voxengo*. URL: https://www.voxengo.com/product/span/ (visited on 07/26/2018).

[9] P. Grosche and M. Muller. “Extracting Predominant Local Pulse Information From Music Recordings”. In: *IEEE Transactions on Audio, Speech, and Language Processing* 19.6 (Aug. 2011), pp. 1688–1701. ISSN: 1558-7916. DOI: 10.1109/TASL.2010.2096216.

[10] Daniel J. Levitin. *This Is Your Brain on Music: The Science of a Human Obsession*. English. Reprint edition. New York, N.Y: Plume/Penguin, Aug. 2007. ISBN: 978-0-452-28852-2.

[11] Ali Mottaghi et al. “OBTAIN: Real-Time Beat Tracking in Audio Signals”. In: *International Journal of Signal Processing Systems* 5 (Apr. 2017). DOI: 10.18178/ijsps.5.4.123-129.

[12] Meinard Müller. *Fundamentals of Music Processing*. Jan. 2015. ISBN: 978-3-319-21944-8. DOI: 10.1007/978-3-319-21945-5.

[13] Daniel P. W. Ellis. “Beat Tracking by Dynamic Programming”. In: *Journal of New Music Research* 36 (Mar. 2007), pp. 51–60. DOI: 10.1080/09298210701653344.

[14] Geoffroy Peeters. “Template-Based Estimation of Time-Varying Tempo”. In: *EURASIP Journal on Advances in Signal Processing* 2007 (Jan. 2007). DOI: 10.1155/2007/67215.

[15] *PortAudio: PortAudio API Overview*. URL: http://portaudio.com/docs/v19-doxydocs/api_overview.html (visited on 07/17/2018).

[16] *Qt Documentation*. URL: http://doc.qt.io/ (visited on 07/17/2018).

[17] Steinberg. *VST 3 SDK Documentation*. July 2018. URL: https://www.steinberg.net/en/company/developers.html.

[18] *Steinberg Cubase*. en. Page Version ID: 849780932. July 2018. URL: https://en.wikipedia.org/w/index.php?title=Steinberg_Cubase&oldid=849780932 (visited on 07/20/2018).

[19] Andrew Swift. “A brief Introduction to MIDI”. In: (May 1997). URL: http://www.doc.ic.ac.uk/~nd/surprise_97/journal/vol1/aps2/.

[20] *Virtual Studio Technology*. en. Page Version ID: 848455033. July 2018. URL: https://en.wikipedia.org/w/index.php?title=Virtual_Studio_Technology&oldid=848455033 (visited on 07/20/2018).

[21] R. Zhou, M. Mattavelli, and G. Zoia. “Music Onset Detection Based on Resonator Time Frequency Image”. In: *IEEE Transactions on Audio, Speech, and Language Processing* 16.8 (Nov. 2008), pp. 1685–1695. ISSN: 1558-7916. DOI: 10.1109/TASL.2008.2002042.