VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

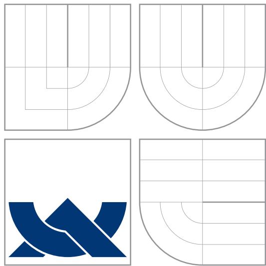
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SIMULATION OF GUITAR SOUND EFFECTS ON MOBILE DEVICE WITH ANDROID

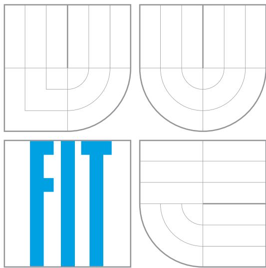
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — TOMÁŠ MÉSZÁROS
AUTHOR

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SIMULACE KYTAROVÝCH ZVUKOVÝCH EFEKTŮ NA MOBILNÍM ZAŘÍZENÍ ANDROID

SIMULATION OF GUITAR SOUND EFFECTS ON MOBILE DEVICE WITH ANDROID

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — TOMÁŠ MÉSZÁROS
AUTHOR

VEDOUCÍ PRÁCE — Ing. VÍTĚZSLAV BERAN, Ph.D.
SUPERVISOR

BRNO 2013

Abstrakt

Hlavním cílem této práce je zkoumání možnosti real-time zpracování zvuku v operačním systému Android a vytvořit prototyp aplikace, která překonává vyskytující se obtíže tohoto úkolu. Obsahuje přehled o problému a návrh možného řešení. Implementované části systému jsou popsány podrobněji v jednotlivých kapitolách. Na závěr jsou shrnuty dosavadní výsledky výzkumu a návrhu aplikace.

Abstract

The primary goal of this thesis is to discover the possibilities of real-time audio processing on the Android operating system and to create a prototype application which overcomes the occurring difficulties of this task. It includes an overview of the problem and the design of a possible solution. The currently implemented components are described in detail. As an afterword, a conclusion is made about the results obtained trough the research and the design process.

Klíčová slova

Android, modelování zvukových efektů, real-time, zvuková karta, ALSA, Linux, USB On-The-Go, Virtual Studio technology, VST pluginy, LADSPA, hostování zvukových pluginů

Keywords

Android, sound effects modelling, real-time, sound card, ALSA, Linux, USB On-The-Go, Virtual Studio Technology, VST plugin, LADSPA, signal processing, audio plugin hosting

Citace

Tomáš Mészáros: Simulation of Guitar Sound Effects on Mobile Device with Android, bakalářská práce, Brno, FIT VUT v Brně, 2013

Simulation of Guitar Sound Effects on Mobile Device with Android

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Vítězslava Berana, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Tomáš Mészáros
May 12, 2013

Poděkování

Děkuji svému vedoucímu bakalářské práce panu Ing. Vítězslavu Beranovi, Ph.D, který podporoval nápad této práce a poskytl odbornou pomoc při řešení. Dále bych poděkoval panu Bc. Sándorovi Ruhásovi za pomoc při měření výsledků a poskytování měřicích prostředků.

© Tomáš Mészáros, 2013.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Contents

1 Introduction **2**

2 Theoretical Overview of Sound Processing on Android **3**
- 2.1 Audio effects in a musical context 3
- 2.2 Virtualization of analogue effect blocks - pedal-boards 4
- 2.3 Reusable audio filters - Plug-ins 6
- 2.4 The Android sound architecture 7
- 2.5 Connecting Android devices with external peripherals 9

3 Conceptional Design of an Android Based Virtual Pedal-board **11**
- 3.1 Decomposition of the application 11
- 3.2 Designing a low latency audio subsystem 12
- 3.3 Designing a plug-in hosting service 15
- 3.4 Front-end for the plug-in hosting service 19

4 Realization Details **21**
- 4.1 Loading the USB-audio driver into the Android kernel 21
- 4.2 The low latency subsystem engine 22
- 4.3 Plug-in hosting service - implementation details 25
- 4.4 Discussion of the implemented user interface 27

5 Conclusion **28**

A Development Environment **30**

B Contents of the provided CD **31**

Chapter 1

Introduction

Today's music industry is crawling with products emulating sound effects which can be applied to many instruments. These solutions (usually called *multieffects*) are mainly distributed as an embedded device with its own processing unit and controlling interface. The market of mobile devices is growing quickly and the processing power of these devices like tablets and mobile phones is becoming suitable for real-time processing of an audio signal. For this reason an idea was born to actually use these devices for such purposes.

An application for a mobile device that can process the signal from a real instrument like an electric guitar, bass or microphone would be cost efficient and very useful for musicians, allowing them to use it as a personal practising or recording tool. Additionally, it could replace some parts of their equipment for live performance as well.

Examples of such applications are available from Apple for iPads and iPhones as their operating system provides a good support for connecting sound devices with very low latency. On the other hand, no such software is available for Android because of the lack of this support by the operating system's API. This fact is disappointing and seams to be solvable through the Linux kernel which is used as the core of the OS.

The aim of this work is to demonstrate the ability of the Android OS to run such programs and serve as a proof that official low latency audio support would be appreciated and useful. The main goal is to implement a subsystem that can utilize existing audio plug-in filters created using different technologies (these will be discussed later in this document). Although the provided solution for real-time sound processing will be described in detail, it cannot replace a generally applicable solution provided in the future releases of the official Android SDK.

Chapter 2

Theoretical Overview of Sound Processing on Android

This chapter provides a generic introduction of the main notions used in the thesis. Gathers all the necessary information from the initial research gravitating towards the Android OS and tries to filter the results through the viewpoint of this platform.

2.1 Audio effects in a musical context

From technical perspective, an audio effect can be thought as a processing block in an electric signal chain. It can do whatever is needed by a musician from a simple hall effect to a revolutionary sounding instrument that does not remind the original input at all. However, the result is still musical and pleasing to the ears of listeners.

These effects can be built as an analogue circuit or simulated by an algorithm. Each approach has its advantages and withdraws. An analogue effect can be more authentic, it can have a more detailed and richer sound. The simulation on the other hand has many practical benefits. It will have significantly lower price and some other positive side effects like noise reduction or the smaller size of the resulting devices. There are ongoing debates about which solution is better. The accuracy of a model highly depends on the processing power of the hosting device and the tendency of increasing computing performance implicates a growing popularity of digital audio effects across many musicians today.

This work does not discuss the various algorithms and techniques of digital audio processing rather it uses existing software modules that are open-source and are free to be used. The reasons of this choice are the following:

- good quality audio effects are hard to write
- additional measurements of the real (analogue) effect could be unavoidable
- existing free — or even open-source — models are available on the internet.

Audio effects are commonly categorized based on the musical qualities they are providing. These categories are named in [3] as :

Signal Conditioners All gain-based and EQ-based effects are included. Their primary function is not to change the nature of the sound but mainly to increase its gain or the amplitude of specific frequencies.

e. g.: preamps, equalizers, distortions, compressors, wah-wah, ...

Modulation and Time-Based Effects Time-based effects, by definition, combine the original signal with a time-manipulated version.

e. g.: choruses/flangers, pitch-shifters, delays, ...

Ambient Processors These effects usually simulate some kind of environment.

e. g.: reverb and delay[1]

Other Effects A big number of other effects are known and used in practise that do not really fit in any of these categories.

e. g.: octavers, noise-gates, synthesizers, harmonizers, ...

From the perspective of simulation the most difficult task is to simulate a specific amplifier. Taking an electric guitar sound as a basis, there are legendary amplifiers by many manufacturers, that — for some reason — gained big popularity across musicians. A common property of these amplifiers is the tube driven power stage. Simulating the behaviour of this electronic component is not trivial. Nevertheless, the real essence of a guitar sound is bound to the amplifier to which the guitar is connected.

2.2 Virtualization of analogue effect blocks - pedal-boards

The traditional way of colouring an instrument's sound is to create a so called *pedalboard* (Figure 2.1). This can contain all kinds of stompboxes[2] which are analogue processing blocks from a technical view. The whole pedalboard can be virtualized by simulating these blocks. The modelling does not have to stop at this point. The impulse response of a guitar amplifier together with the characteristics of the loudspeakers can also be modelled.

Figure 2.1: An example of a pedalboard for live performance. Source: [7]

A *software effect processor* is a digital signal processing (DSP) application which can model the signal path of a guitar equipment from the dry input of the instrument to the

[1] Delay can be thought as both an ambient and a time-based effect. Its realization is evidently connected with time-based operations but the result is often an emulation of an echo sound.

[2] A box containing one or a few effect units traditionally achieved with an analogue circuit but digital implementations are now common alternatives

loudspeakers including the amplifier and the stompbox effects. A popular application of this type is perhaps the software package called *Guitar Rig* developed by the company *Native Instruments*.

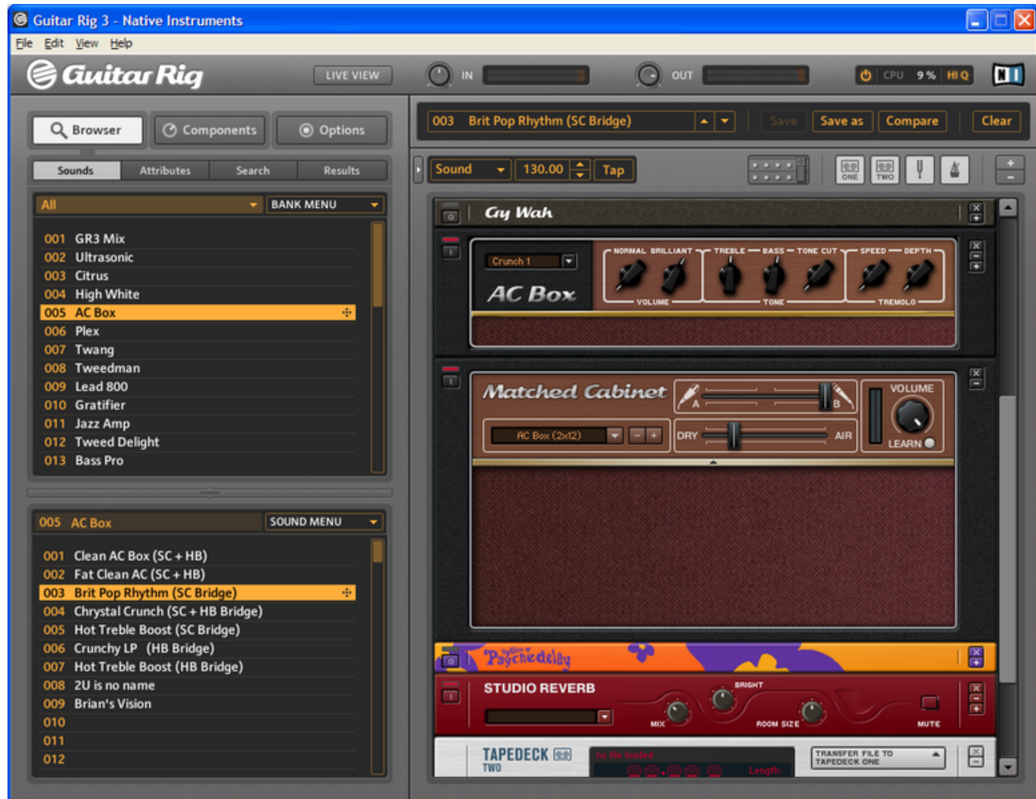


Figure 2.2: Preview of the *Guitar Rig 3* application's UI. Source: Wikipedia

The requirements for user interaction with such software are dependent on the forms of usage. A good approach could be the visualization of a real pedalboard as the average musician is used to them. A tablet with a big touch-screen is a good candidate for being a hosting device in situations like practising or quick recording. A home/professional studio recording software has to enable more detailed editing of parameters and fits better to a desktop environment. Last but not least, the sound quality and accuracy of the effect models have to be comparable with their analogue alternatives.

One of the biggest requirements in real-time processing is the extremely low latency between the input and the output interface. According to [6] latency is the delay between an action and its effect. For example, an action could be triggered by pressing a button and the reaction would be to hear a note. The elapsed time is the overall latency of the system and has 3 major sources:

Physical Distance from the loudspeakers.

Hardware Latencies of hardware components, ADC and DAC conversions, hardware buffering.

Software Mainly caused by software buffering, processing algorithms and the platform itself.

The maximal time of human-perceptible audio latency is assumed between $\mathbf{20 - 30\,ms}$ [4]. The lowest value from this interval has to be taken as an upper limit. The easiest way of reducing software delays is by using short buffers and minimizing platform specific overhead — by choosing the right API for instance. This topic is essential and has to be examined from different angles. The goal is a soft real-time application which have to respond to the input signal without noticeable delays.

2.3 Reusable audio filters - Plug-ins

The simulation of an audio effect is a computer program written in an appropriate language — usually C or C++. To make these simulations reusable they have to be packaged in some format which can be used by a set of applications. In an embedded device there is no need for such packaging but when it comes to professional audio editing programs or desktop based multi-effect applications this modularity is highly required. The usual way of distributing audio effect models is in the form of shared or dynamic libraries[3] mostly referred to as *plug-ins*.

A plug-in is not an application and cannot be used without a hosting software. It can be thought as a black box that receives a stream of audio samples, processes this stream and sends it out to its output [9]. The concept is very similar to the stomboxes mentioned in chapter 2.2 from the analogue world.

Plug-in creation can be done with a specific API or framework that defines the interface of the shared library. Frameworks do exist for different platforms with different approaches but their interfaces can be unified with a moderate effort.

A lot of audio editing applications are using some form of plug-ins to allow the extension of available effects. A quick overview of the best known[4] open-source and proprietary formats is provided here.

Steinberg's VST Plug-in framework

A proprietary plug-in framework from the company Steinberg written in C++. Available for Windows, Linux and Mac OS but compiling it for Android is possible as well.

LADSPA Plug-in API

Linux Audio Developer's Simple Plug-in API is an open-source framework for writing audio effects for GNU/Linux operating systems or generally for any operating system. As a framework, it is really simple and easy to understand. The documentation is captured in the single header file of this API.

DSSI

Stands for *Disposable Soft Synth Interface*. Very similar to the previous one. Sometimes referred to as LADSPA for instruments.

VAMP

Very sophisticated C++ API. Unlike the previous technologies, this framework is specialized for feature extraction and audio analysis.

Other frameworks that are worth mentioning but they are either closed source or incompatible with the Android architecture:

[3] Standard OS dependent binary file, .dll in Windows, .so in Linux...

[4] Plug-in formats are publicly available and a generic overview is available on Wikipedia

- *Real-Time AudioSuite (RTAS)*
- *Apple Computer's Audio Units*

Using the plug-ins requires a hosting application that takes the inputs and outputs of the plug-in and connects them with other plug-ins or with an audio interface. Also manages effect parameters and controls the signal chain in its entirety. The user interface may belong to the host as well and it can incorporate many plug-in formats depending on its specifications.

Writing a very simple realization requires the following steps [2]:

1. Loading and initializing the plug-in
2. Set up the callback methods and other connections with the plug-in
3. Provide audio samples to the plug-in for processing

The software *Guitar Rig* mentioned in chapter 2.2 is actually a plug-in hosting application that is compatible with a variety of plug-in formats.

2.4 The Android sound architecture

To implement a low latency audio (LLA hereafter) system it is very important to be familiar with the limitations and capabilities of the target platform. The support for LLA in Android SDK is under development[5] and there is no well documented or easy way to get it working.

Android is based on the Linux kernel. This means that drivers and hardware specific code is all bound to the monolithic approach of this well known operating system. Manufacturers can branch the kernel source tree and write their own drivers for their devices. The audio section follows this concept without any exception. The need for abstracting these drivers into a single interface — HAL[6], to use the right term — is immediate.

The Linux kernel has already included a type of HAL for its sound devices. It is called **ALSA**, an acronym for *Advanced Linux Sound Architecture* which is a stable and popular base for many Linux distributions to control sound devices [8]. Even so, Android have to deal with a bigger confusion in drivers and hardware layers. A new level has been implemented as a solution, that unifies ALSA drivers with other manufacturer specific drivers. As a consequence, an Android based device is not forced to use ALSA as a framework for its audio drivers.

There is one more important aspect that has to be introduced to the reader to understand the problem behind real-time applications in Android. This is the well known *Dalvik Virtual Machine*. A Java VM reimplemented and optimized by Google. It has many advantages in resolving hardware incompatibilities for the wide variety of Android devices. The side-effect is evidently a dropdown in performance. The overhead caused by Dalvik significantly decreases the ability of Android to host real-time applications.

The Android platform provides a development framework to aid the creation and publication of new software [1]. These tools and the features they provide is a very important design aspect. As mentioned before, the support for real-time applications is absent from this framework in its present state and a brief introduction of this API is useful as it will be mentioned many times from now on.

[5] development news at: http://developer.android.com/about/versions/jelly-bean.html
[6] Hardware Abstraction Layer

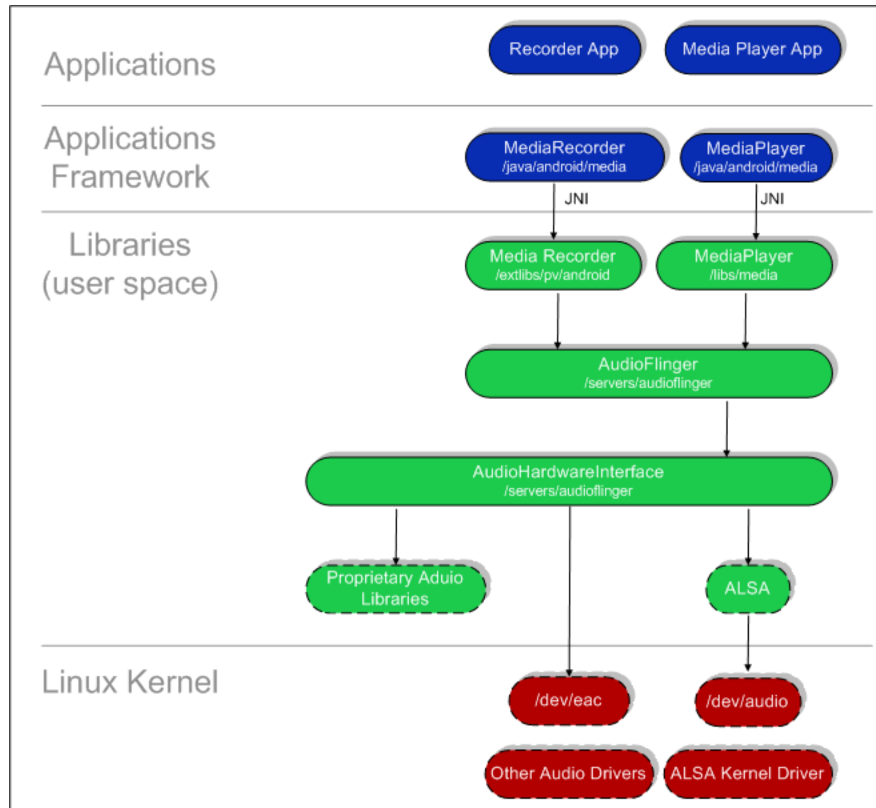


Figure 2.3: The Android audio subsystem. http://www.netmite.com/android/mydroid/development/pdk/docs/audio_sub_system.html

The SDK is based on the Java language. The official documentation [1] states that the primary way of building applications is through this framework. It provides a standard set of classes that are present in the Java API and additional classes associated with the Android OS. A native tool-chain called NDK is provided as an optional part of the SDK[7]. It enables to write part of the application in a native language such as C or C++. Some functions or libraries can be compiled to native binaries or shared libraries and a specific interface — the *Java Native Interface (JNI[8])* — is responsible for connecting these native functions with the Java world.

ALSA is simply the core of Linux audio. It has the control over hardware specific functions in the kernelspace and can be used as a framework for writing audio applications with the provided userspace library. The project has its own official implementation the *ALSA Library API* (referred to as the `alsa-lib` package) [8]. This is a low level library with a robust functionality. In fact, Android uses `tinyalsa`[9], its own tiny version of this library to provide the very basic features for upper layers. Using `alsa-lib` in Android applications is generally not considered a good practise and in the SDK (nor the NDK) it is not even available. Other library worth mentioning is the `salsa-lib` library package. This is a lighter implementation of `alsa-lib` with limited features targeted to embedded systems. The interface is source level compatible with the official alternative.

[7] Available at http://developer.android.com/tools/sdk/ndk/index.html
[8] Java Native Interface – Calling C/C++ functions from Java and accessing Java objects from C++
[9] Available from the official Android source code repositories.

In order to implement a stable audio pipe with the lowest possible latency, it is important to be familiar with the transfer methods of a PCM stream[10]. This includes the playback and capture procedure which are handled by ALSA as I/O operations. The audio devices are files in the `/dev` directory just like any other peripheral. At some point during the transfer an I/O system call is involved and the overlying API has to obey all the rules of such operation. ALSA distinguishes three basic transfer modes [5]:

1. **Blocking read/write:** An I/O operation will block until the required amount of samples is not available.

2. **Non-blocking read/write:** An I/O operation will never block and explicit waiting is used via a *poll* or *select* system call.

3. **Asynchronous transfer:** The process is signalled periodically and resolves the transfer in a signal handler method. *This kind of transfer is* ***not part*** *of the safe operation subset in ALSA.*[11]

ALSA uses the term *period* to mark a time or data block that is periodically transferred while capturing or playing audio samples. Also defines a *frame* as a collection of samples for multiple channels corresponding to a given point in time. Samples are collected in frames with an *interleaved* or *non-interleaved* organization. The Figure 2.4 is an example PCM data layout.

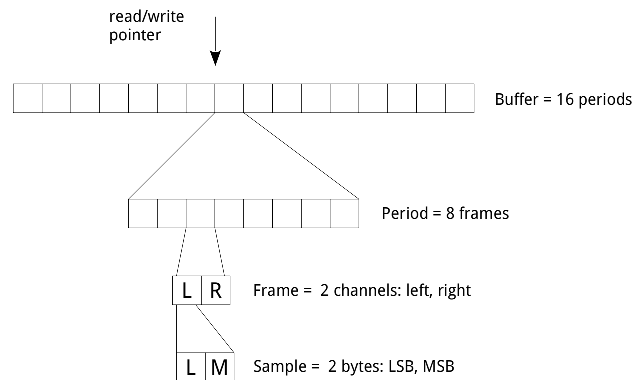


Figure 2.4: PCM data model [13]

2.5 Connecting Android devices with external peripherals

In order to connect anything with a mobile device it has to be equipped with a set of ports which allows interaction with other devices. A wireless solution appears to be a better choice to avoid the use of cables and to maintain the „*mobility*“ of mobile devices. The Android OS has support for peripherals like a keyboard or a mouse thanks to the mentioned Linux kernel which has a decent collection of drivers for external devices. Beside the wireless methods a connection can also be made via the USB port. The primary purpose of an USB connector on a mobile device is to make connection with a PC and for charging the battery.

[10] Pulse-Code Modulation - digitally sampled audio signal

[11] According to the unofficial ALSA wiki page http://alsa.opensrc.org/HowTo_Asynchronous_Playback

In these situations the PC has the role of a USB host and the mobile or tablet acts as a USB device [10]. A mobile device can take the role of a USB host only if it supports the *On-the-Go*[12] extension of the USB specification[13]. This requires a USB OTG controller and a Micro-AB USB plug installed on the device.

This type of connection is incredibly useful and surprisingly a poor documented area of mobile devices. Manufacturers do not give a big attention to indicate the state of this technology in their devices and the easiest way to find out whether it is enabled on a particular phone or tablet is to actually connect it with an external peripheral.

Android's approach to support external devices is the *Android Open Accessory (AOA)* protocol[14]. The connected device has to be implemented according to the specifications described in AOA. The protocol enables the connection without the OTG functionality because the external device acts as a host and the Android-powered device acts in the USB accessory role. Audio support is introduced in AOA 2.0, but no generic sound card support is available for now.

[12] Abbreviated as USB OTG.

[13] http://www.usb.org/developers/onthego/USB_OTG_and_EH_2-0.pdf

[14] http://source.android.com/tech/accessories/index.html

Chapter 3

Conceptional Design of an Android Based Virtual Pedal-board

This chapter can be thought as a detailed conceptual and implementation plan for constructing the final application. Describes the problems and difficulties then chooses an available solution for all of them. Includes UML diagrams and explanations of implementation specific decisions.

3.1 Decomposition of the application

The subject of this thesis is to create an application following the principals introduced in section 2.2. The signal of the instrument is captured by a suitable audio interface and the digitalized audio stream is forwarded to the tablet/mobile via a Micro-USB port. The sound is then processed by the application using open-source plug-ins and sent back to an output interface. The gateway of the guitar signal to the device could be an external USB sound card suitable for real-time processing.

Figure 3.1: Illustration of the initial idea

The first and most important task is to connect the instrument and solve or find a work-

around for latency problems. This procedure is hosted in a separate module. Secondly, a multi-threaded plug-in hosting environment is required to utilize audio filters with variable format or technology. Finally, the application has to be controlled by the user through a visual interface. The Figure 3.2 shows these modules in a bottom-up layout. From the aspect of this thesis the most challenging are the first two modules and they have priority for implementation. The user interface is opened for future enhancements.

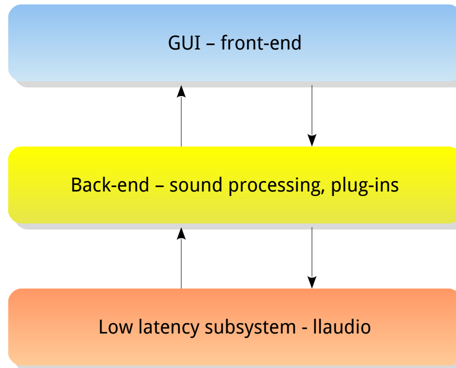


Figure 3.2: Design stages

In the newest versions of Android OS, the Open Accessory protocol enables the sending of audio streams to an external device that meets the AOA specifications. The problem is that none of the generic USB sound interfaces are built with this functionality in mind. The solution is the *USB-audio* kernel module used in most Linux distributions as a generic ALSA USB sound card driver. Unfortunately, the majority of Android-powered devices have their kernel compiled without this module. As a workaround, the kernel version in a particular device can be easily obtained and the module can be compiled and loaded at runtime. The specific how-to is described in the chapter 4.1. An average user can find this procedure really complex, but at least it can be automated and there is no need for changing the firmware of a particular device.

3.2 Designing a low latency audio subsystem

The application *has* to respond to the input signal in the *20 ms* latency limit declared in chapter 2.2. This is the most essential requirement and the following section is intended to outline a usable solution.

After the external sound card has been made available to ALSA, a library is needed for interfacing with the audio device. The `alsa-lib`[1] is the official API for programming audio applications in ALSA, but as mentioned in 2.4 there are other — more lightweight versions. Compilation of the `alsa-lib` package with the NDK is not an option. The package uses the shared memory header files which are deprecated in the NDK tool-chain. Android features the *Binder* [12] mechanism instead for Java applications and drops support for a few *System V IPC*[2] facilities. Sockets and pipes are still usable though.

The NDK includes the *OpenSL ES* library. From all the supported and integrated alternatives this API offers the lowest latency [11].

[1]In some distributions available as the *libasound* package.
[2]Standard Inter Process Communication methods used in UNIX-like systems

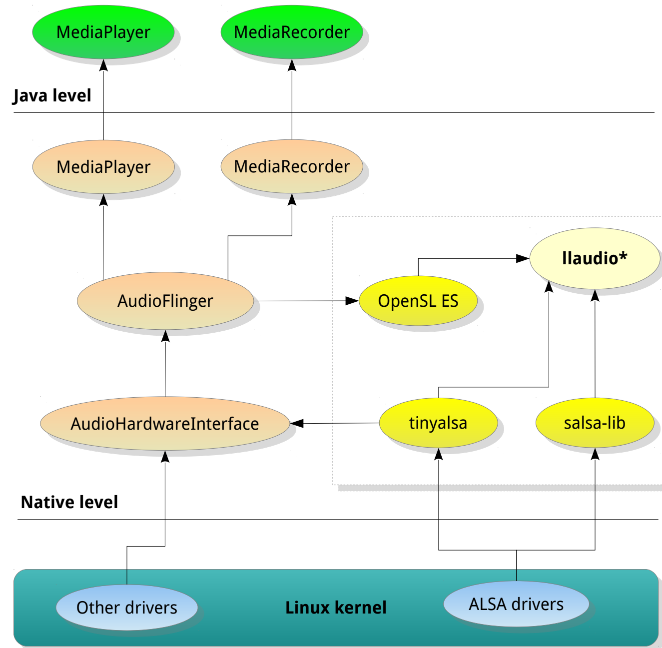


Figure 3.3: The libraries and their relationship

The output latency has gone trough a major optimization and a lot of applications benefit from these improvements (starting with the Jelly Bean family of the Android OS), but the input is still very slow thus the provided API is not a good option to start with. The best solution for minimizing the software latency seems to be the `tinyalsa` library, since it already represents the link between ALSA and Android[3]. As a fall-back option, there is `salsa-lib` which can be compiled for Android as well.

Taking in account that a suitable improvement of the OpenSL ES library might get released in the future, my final decision has been to make my own *low latency audio library* which uses `tinyalsa` as the first possible driver and to make sure that other drivers or *engines* (like the OpenSL ES) can replace `tinyalsa` after the arrival of an official latency free API. This new layer corresponds to the red rectangle on Figure 3.3 and it will be mentioned as the `llaudio` library from now on. It aims to provide a wrapper interface for any kind of low latency audio library and to help the final application to be independent on the underlying subsystem. This approach enables to build a prototype with the `tinyalsa` library and to change it when the support for LLA will be available or if this API would not met the appropriate requirements.

The Figure 3.4 shows the basic shape of the `llaudio` library in the form of a class diagram concentrating mainly on the interfaces and leaving the implementation classes to other sections. This is an important part of the work. The approximate position of this layer in the Android audio stack is illustrated on the Figure 3.3.

[3] Only in the source tree of the Android OS. The SDK does not include `tinyalsa` as an API.

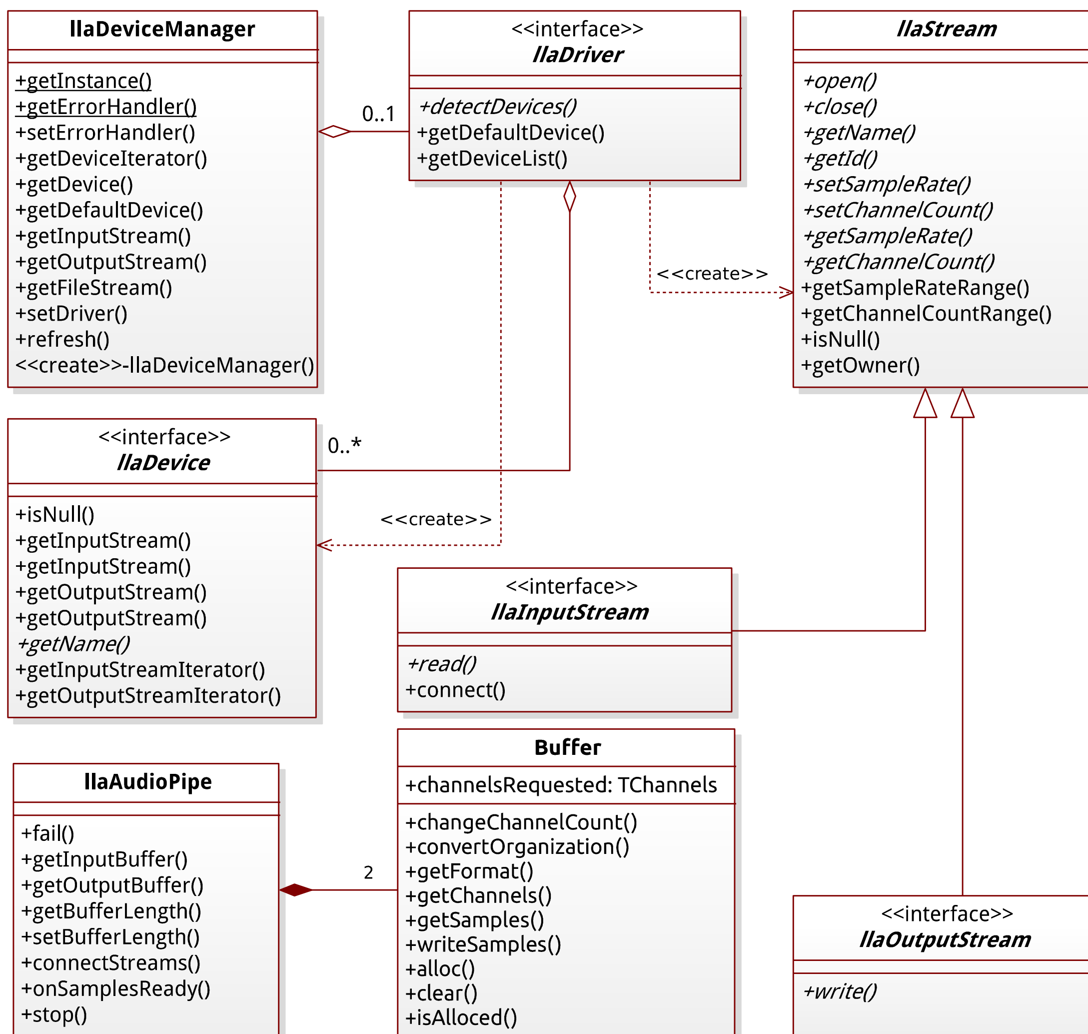


Figure 3.4: Class diagram of the low latency audio layer

The library can be instantiated with the class `llaDeviceManager` which has a singleton character. This object can provide a list of available sound devices gathered from a particular driver. The set of polymorphic base classes `llaDriver`, `llaDevice`, `llaStream`, `llaInputStream`, and `llaOutputStream` are meant to be implemented in specific subclasses which are allowed to use whatever library or API is preferred like the mentioned *tinyalsa*, *salsa-lib*, or *OpenSL ES*. This kind of separate implementation is considered as a *driver or engine* of the *llaudio* library. All objects of type `llaDevice` represent a real hardware based or logical sound device with its name as a primary key for identification. The device has a set of input and a set of output audio streams. These are encapsulated in the `llaInputStream` and `llaOutputStream` classes. Device objects can be obtained from the device manager and the streams are provided by devices. Alternatively, an audio stream is accessible directly from the device manager with the `getInputStream()` and `getOutputStream()` methods by specifying the name of the device and the ID of the stream.

Allocation and memory issues are completely handled in the library and there is no need for deleting objects like devices or streams obtained from the library instance. These

structures are handled by reference to access the polymorphic feature of C++ objects. Explicit clean-up operation is required by the singleton `llaDeviceManager` object only.

The `llaAudioPipe` class is a pipe-like facility to interchange audio data between input and output streams. Samples can be written into or read from it. The class is composed of two `Buffer` type objects (for input and output). The actual samples are stored in internal buffers and the `getSamples()` method of the `Buffer` class delivers them in a unified format handling conversions and memory allocations. The `connectStreams()` method creates a flow of audio data from an input to an output stream, doing this with the lowest possible latency that the engine can deliver. This method could be useless if the `read()` and the `write()` methods of stream objects operate with sufficient delays.

The library also introduces a callback mechanism. The overridden `onSamplesReady()` method of a pipe is the point where the actual processing of audio samples takes place and called right before the samples are written to the output.

It is worth to spend a few words on the internal data structures holding all the device and stream objects in this library. With a careful approach the efficiency and code size can be highly optimised allowing an easy integration into embedded platforms other than Android or any ARM based board with an audio chip. The library has to use a predefined internal interface for its sequence types and a default implementation can be provided by the STL library. Considering a hypothetical situation where there is no need to handle multiple sound devices, the use of such sequences would represent a big redundancy in the binary size and a degradation of execution speed. In such cases the STL sequences can be replaced with dummy but fast and efficient containers for the particular audio chip.

3.3 Designing a plug-in hosting service

This part corresponds to the yellow rectangle on Figure 3.2. The module has the responsibility for managing plug-ins and the processing pipeline. Also dispatches commands from the user interface. Beside these essential features, it has to provide a database of sound filters (derived from various plug-ins), the ability to save and load the configuration of the effect chain as a *„preset“* and to store them in a bank or logical directory.

The are a few options for creating the service. It can be written in Java by providing the latency sensitive functions through native methods (through the JNI interface). This approach is problematic due to the fact that the majority of audio plug-ins are written in C/C++. To easily integrate the plug-ins, a better solution can be a completely native module with a well defined interface that is controllable form Java classes.

The situation gets a bit more complex with the requirement for root access to the system resources. Only the *root* user or the *audio* group can open a sound device in the Android environment (which is at this level identical to Linux). With an engine of `llaudio` being a library provided by the NDK there would be no need for such privileges, but with the compiled and statically linked `tinyalsa` or `salsa-lib` an audio device in the `/dev/snd` directory can be opened only as a privileged user. The straightforward way to compile the module as a shared library and load to the application in a Java class seems to be compromised with this issue. An important fact is that Android runs all the applications with a unique Linux *UID* and privileged native methods do not succeed if they are called from a shared library linked to the Java application. However, root privileges can be assigned to any executable with the *su* command after the rooting process.

My solution is to compile the plug-in hosting module together with the `llaudio` library as a separate executable file which behaves as a service in the Android system. The front-

end can be treated as a „*client*“ meanwhile other clients can connect as well (illustrated on Figure 3.7). The outlined architectures are compared on Figure 3.5.

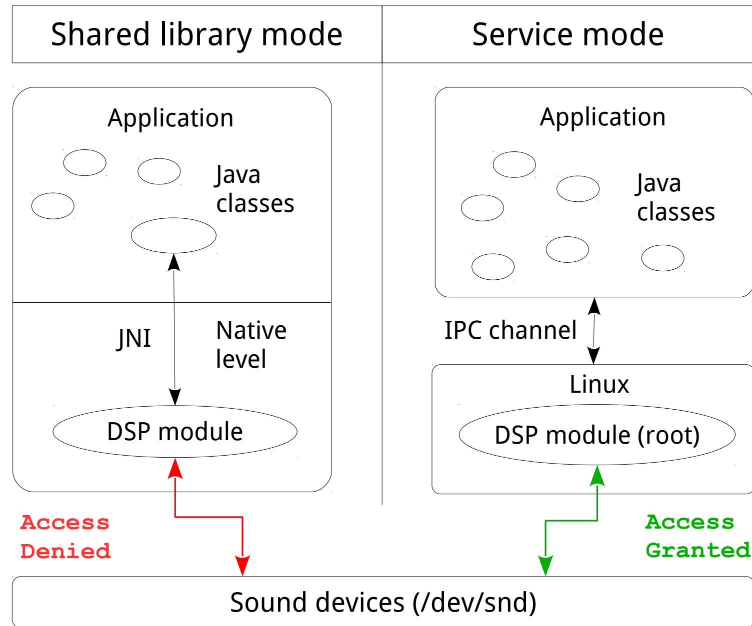


Figure 3.5: Architecture comparison

The Figure 3.6 visualizes the model of the DSP module and the relationships between the main interfaces. The centre of the module is the `DspServer` class which controls the service and manages user commands. The signature of this class can be considered as an API to the system. On instantiation it has no clients that could control the processing. One or more `ClientConnector` type objects have to be specified to provide the messages (captured in the `Message` class) to the server by implementing the receiving and sending methods. The specific mechanism for obtaining messages is completely in responsibility of the `ClientConnectors`. The source of incoming data can be a remote process with an IPC connection or a separate thread connected by a special interface (e.g. JNI in Android). A `ClientConnector` object also utilizes his own execution thread to watch the data input.

Each incoming message is a subclass of the `InboundMessage` base and it carries a command for the server in the overridden `instruct()` method. The server has a special blocking queue to store the messages and to be able to block if no user interaction has been made. A call to the `popMessage()` method blocks the server until a new message is not pushed into the queue. When the message is pulled out, its instruction is executed on the server, and the result is broadcasted to all clients. For example, if the user switches a *preset* with the MIDI controller, the result can be seen on the UI of the mobile device. The benefit of this multi-threaded approach is that the sound processing can run practically undisturbed in its own execution thread with an associated real time scheduling policy to ensure a smooth and continuous experience for the users. The response time for UI events is evidently less crucial than the stability of the processing.

The `SoundEffect` class is a *wrapper* around audio filters. A specialization can provide a bridge to LADSPA, VST, or generally any type of audio plug-in, or can be a simple native filter. The processing is leaved to the `DspProcess` class which takes a `ProcessingGraph`

type object at instantiation. This is an interface to a net of filters connected together. The scheme is not relevant from the perspective of the DSP process. It simply calls the `traverse()` method of the graph and runs all the filters on the prepared audio data. The order of the filters and their connection is up to the implementation of the processing graph.

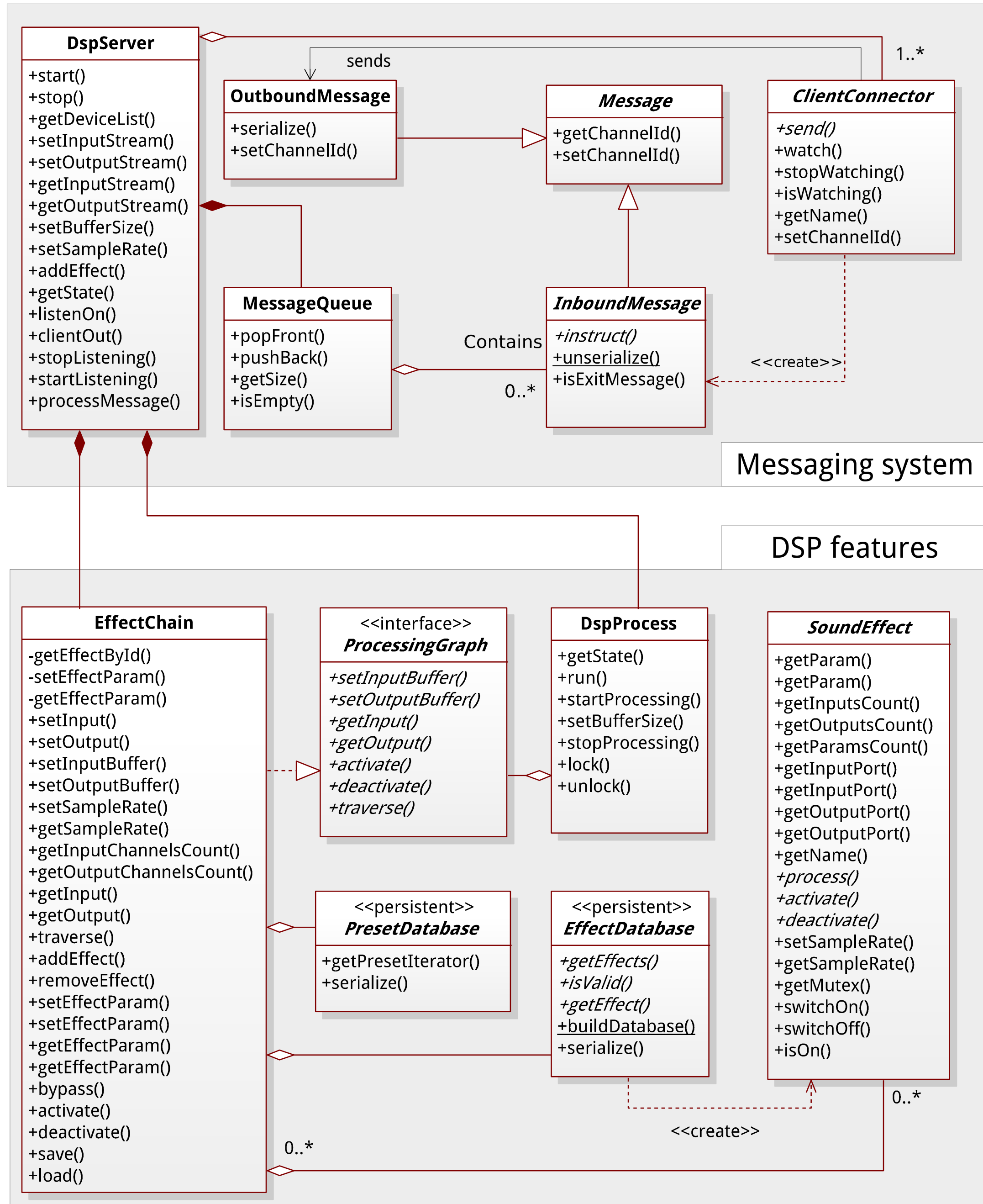


Figure 3.6: OOP design of the DSP service

The `EffectChain` class is a realization of the `ProcessingGraph` interface that models the traditional guitar signal path where the effects are connected in series and the signal is created in a fashion where each block processes the previous block's output. The

`EffectChain` is composed of a mono input and a stereo output node which are `SoundEffect` objects as well. Any number of effects can be inserted between these two built-in nodes if the number of input and output ports are compatible with the surrounding nodes. An additional mixer-node can be inserted in the case of incompatibility. All `SoundEffect` objects are supplied by a *semaphore* to ensure that a parameter of the effect will not change if samples are being processed by it.

Two classes are declared on Figure 3.6 whose state can be recovered from a persistent storage. Namely the `EffectDatabase` and the `PresetDatabase` classes are involved. Although the service uses open-source audio plug-ins as the main source of effects, some kind of meta information is required to integrate them in the context of a virtual pedal-board. This could be done with additional meta-files (e.g. XML or JSON) describing a sound processing block by its name, description, and other properties. Also declaring a category for each effect from a predefined set of effect types like distortion, modulation, ambient processor, or any other (discussed in chapter 2.1) and assigning the shared library file and program identifier to the effect. After loading this database, a `SoundEffect` object can be instantiated by supplying the unique name of the effect to the database, thus behaving like an *abstract factory* of audio filters. The `PresetDatabase` class is a storage structure to store the state of the `EffectChain` object. This saved image is a predefined tone that can be recalled any time. Musicians sometimes refer to them as *patches*. On Android the storage can be the external SD card or the internal flash memory of the device. All applications have a default working directory with the location provided through the Java API.

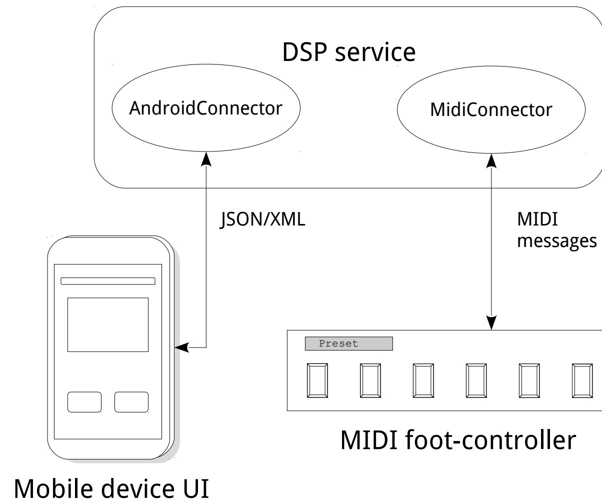


Figure 3.7: DSP service with clients

The controlling algorithm is shown on the Figure 3.8 with a main thread that receives and evaluates messages coming from the `ClientConnector` threads. If a start message is received, the processing thread is started. This event is symbolized by setting the state of the `DspProcess` object to ***`Running`***. As the opposite operation a stop message terminates the sound processing thread. All other messages are doing some kind of modification in the state of the processing. They can add new effect blocks, change a parameter of an effect, change the buffer size, sample rate or other parameter. The `ClientConnector's` *watching* state represents a running thread collecting UI commands from whatever source they come

from. One `ClientConnector` is explicitly required for a running service and others can be specified optionally. The *Message Received* and *Command Received* nodes are waiting points where the execution blocks until no data is obtained.

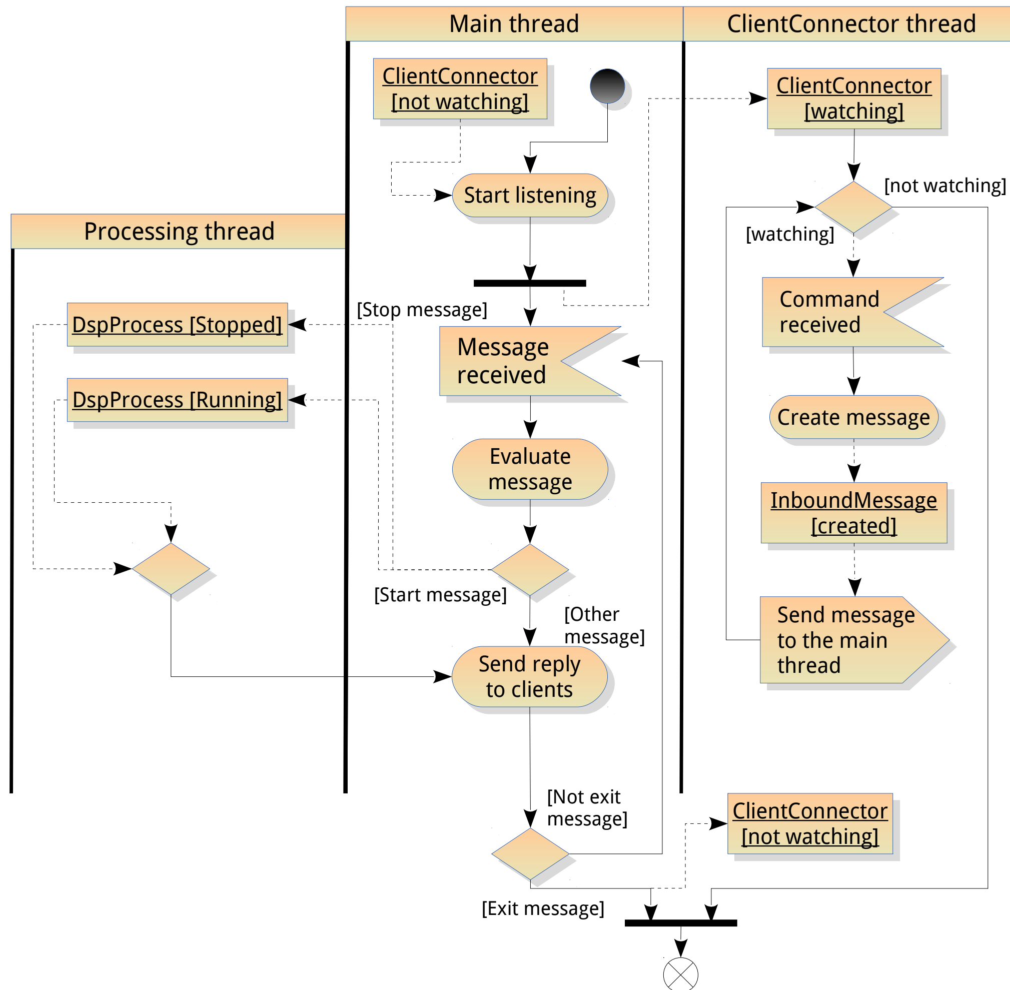


Figure 3.8: Activity diagram of the DSP service

3.4 Front-end for the plug-in hosting service

This short section aims to provide a more detailed description for connecting a user interface with the DSP module, especially on the Android OS.

The previous section introduced the `ClientConnector` class that intends to connect a client with the service. It is important to notice that an object of this type can be a client itself or it can also be only a bridge providing the connection. Talking about this work as an Android application, it has to be installable through an *apk* package without any change in the device firmware.

The previous chapter defines two methods for integrating the DSP module into the application (3.5):

1. Compile the module as a shared library.

2. Compile the module as a service and provide it as an *asset*[4] in the *apk* package. The server executable is then installed and executed on the start-up of the application.

The recommended and safe approach is the first one with the JNI interface. If a suitable API would be provided by the NDK to implement a low latency audio pipe, then no further analysis would be necessary. However, the issues described in chapter 2.4 are forcing the design to the second alternative, at least for a period of a prototype application. This implies a communication protocol and a suitable IPC channel. Using a pipe with a character based protocol seems to be a quick solution to connect an Android *Activity* with the DSP service. Using a socket has more potential in terms of a client-server architecture, but for the prototype application it could be a bit over-complicated.

The Figure 3.9 shows a rough idea about the UI and a vision of how it might look like in a smart-phone. This part of the work is not implemented and it is opened to future development. On this imaginary interface, the main components are the audio effect models stacked in a bottom-up direction. The Output *„effect“* is a mixer node controlling the overall output gain. Each model has a bypass button, and the UI contains a master bypass button inspired by traditional multi-effect processors. An additional *Start/Stop* toggle helps to keep the sound processing running while the application is not visible.

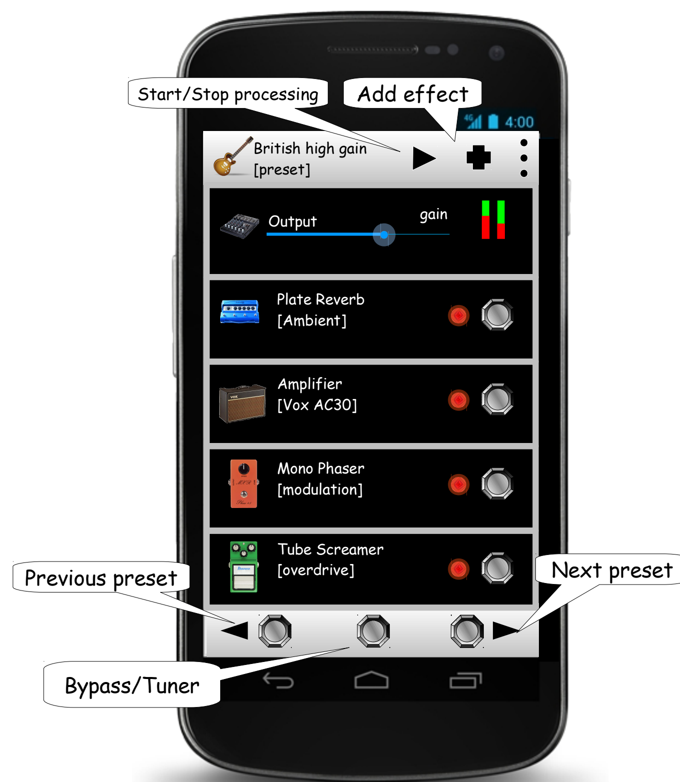


Figure 3.9: UI preview on a smart-phone

[4]Assets are additional files in the application package (*apk*) for general purpose.

Chapter 4

Realization Details

The previous chapter presented a conceptual description of the main components. Specific details were leaved to this chapter to be a reference or a step-by-step guide for various issues. The work has two *„project"* trees. One for the Java code implementing a basic user interface and one for the DSP service with the `llaudio` library written in C++. As a useful information and to maintain the trustworthiness of this thesis, a detailed description of the development tools and the reference mobile device is provided in appendix A.

4.1 Loading the USB-audio driver into the Android kernel

Although the concept is really simple, the mentioned workaround in chapter 3.1 is not trivial. First of all, the device has to be rooted. This simply means that it has to be unlocked to access root privileged commands and service routines.

To make a compatible kernel module the very first step is to obtain the specific kernel version and build number of the particular device (appendix A). The kernel source code can be downloaded via the public git repository of the Android codebase[1]. The steps which led to a compiled kernel module are the following:

1. Cloning the git repository. For the reference device this is:

   ```
   $ git clone https://android.googlesource.com/kernel/omap.git
   ```

2. Searching for the matching revision based on the kernel signature with the `gitk` tool.

3. Compiling the kernel modules using:

   ```
   $ make tuna_defconfig
   $ make menuconfig
   $ make modules
   ```

   After typing `menuconfig`, a terminal based GUI helps to set up the build. The appropriate options have to be set to compile the USB sound card drivers as a kernel module. When the compile process finishes, the kernel modules[2] are available in the directory `$(KERNEL_SRC_DIR)/sound/usb`.

[1] Compilation of an Android kernel: http://source.android.com/source/building-kernels.html.

[2] `snd-hwdep.ko, snd-rawmidi.ko, snd-usbmidi.ko, snd-usb-audio.ko`

4. Load the modules using the `insmod <module.ko>` command in an Android shell.

As a side note the gcc tool-chain provided with the NDK is not suitable for kernel compilation. Instead, an *arm-linux-eabi* version of gcc has to be used. For the reference device the tool-chain was downloaded from the git repository of the Android source tree.

4.2 The low latency subsystem engine

The following section discusses the implementation of the audio path that has a latency below the *20 ms* limit defined in chapter 2.2. The first engine was based on the `tinyalsa` API. The sound quality and latency was sufficient with the i386 desktop Linux build. On the other hand, the Android build was not functional due to unknown sound quality issues with the reference configuration (see appendix A). The most likely causes are incorrect PCM parameters that could not be changed by the API. Instead of tracking down errors in the implementation, my decision was to rewrite the engine with the `salsa-lib` library. The `tinyalsa` engine is therefore not discussed further, though its creation exhausted some fragment of the available time frame.

The `salsa-lib` package is a free, light-weight (with limited functionality), and source level compatible alternative for *libasound*, the official ALSA user-space API. Main targets are embedded or resource restricted platforms. It can provide an interface for opening, writing and reading PCM streams and for controlling their parameters.

As mentioned in chapter 3.2 the engine is made of a few implementation classes. The Figure 4.1 shows how they are connected with the `llaudio` model.

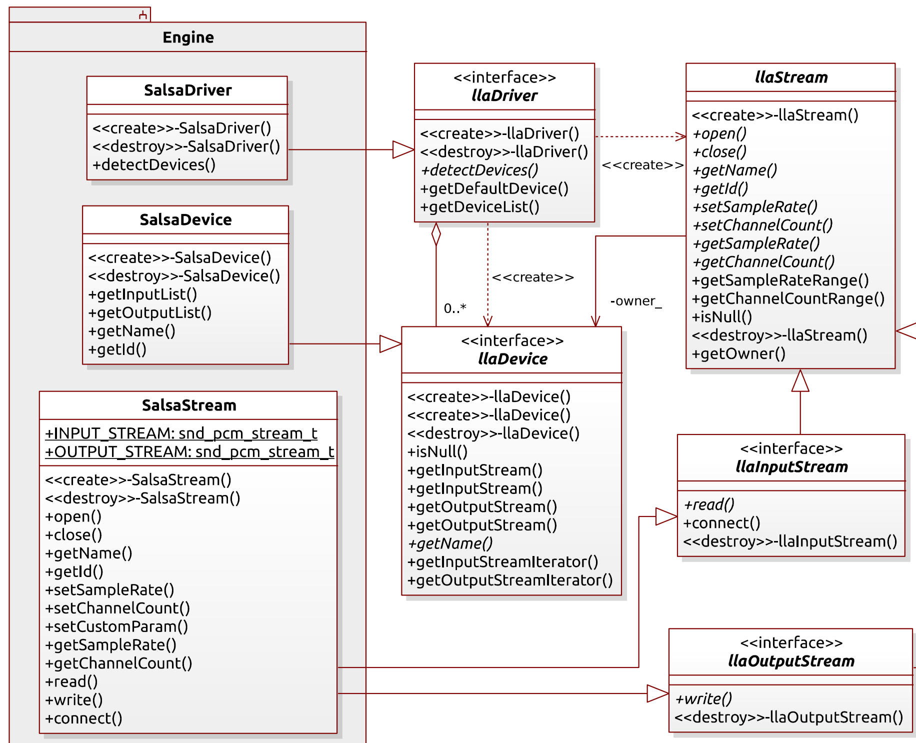


Figure 4.1: The `salsa-lib` engine class diagram

The most interesting aspect of the engine is to achieve the desired latency. The goal is to minimize buffer sizes while still maintain the stability of the stream. The `connect()` method of the `SalsaStream` class implements the low latency audio pipe with variable buffer size. Various transfer methods are described in section 2.4. The implementation uses a non-blocking *read* and a standard blocking *write* transfer in the latest version as this solution appeared to be the most stable. Simply, a very short period of the instrument's sound is recorded and simultaneously the previous block is under processing or played back, if it is already processed. The write operation (playback) blocks only if there is no room for new samples in the hardware ring-buffer of the device. The Figure 4.2 tries to visualise the algorithm with a simple flowchart and a discrete time diagram that shows which operations are done simultaneously.

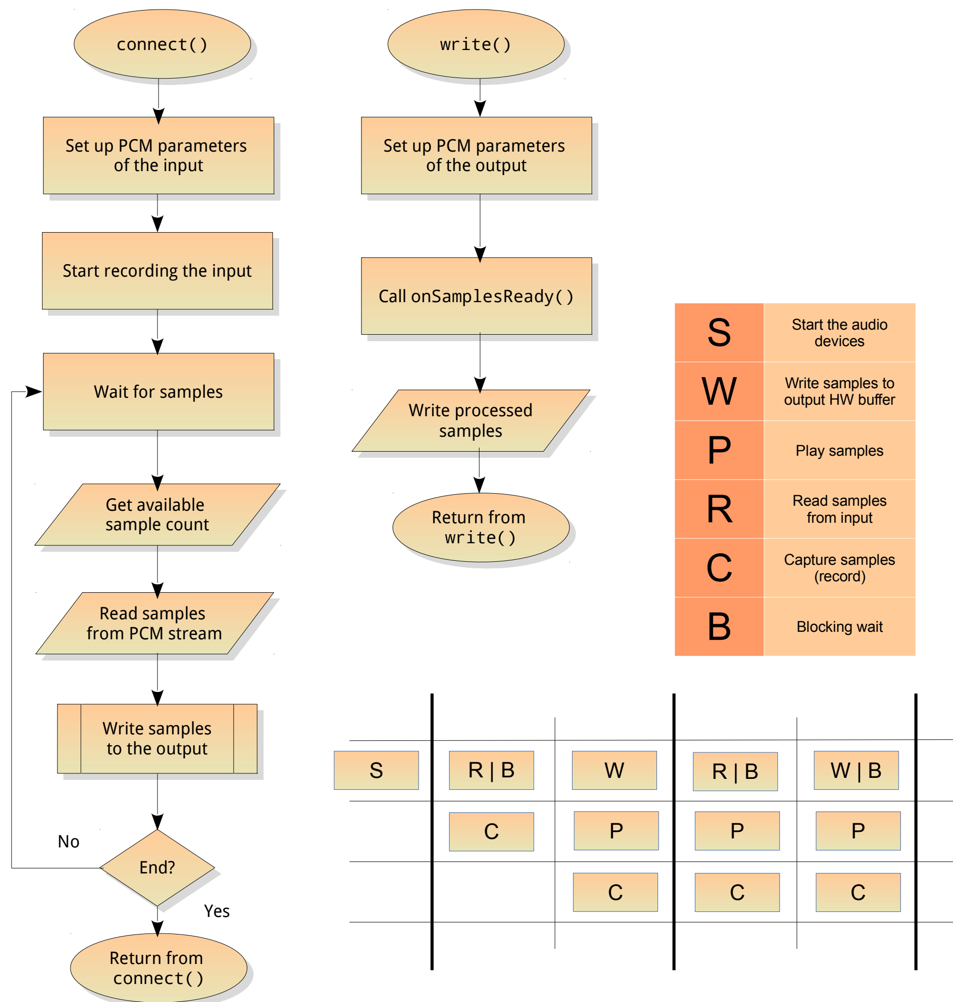


Figure 4.2: Low latency audio pipe - implementation concept

No doubt, that this configuration can be further optimized. Perhaps a different ordering of I/O and blocking wait operations can lead to even smaller latencies and more stable audio flow. Figure 4.3 shows the measurement results of this particular realization.

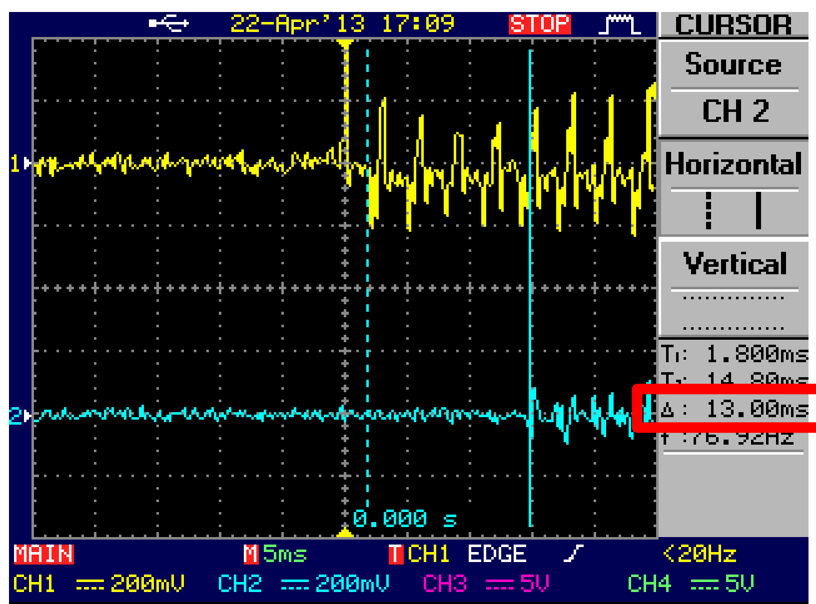


Figure 4.3: measurement results

The measurement was taken with a digital oscilloscope (see appendix A). The upper yellow signal is the raw input of the electric guitar captured by the external USB audio interface. The blue signal at the bottom is the processed output from the reference smartphone coming out from the output jack of the external audio card. A phaser and a plate reverb model are involved in the processing. The noise seen on the snapshot can be caused by the galvanic connections with the 3.5 mm Jack connector. The relevant value is the latency shown in the right side of the picture. **13 ms** is far beyond the noticeable value and enables a comfortable playing experience. An embedded multi-effects processor[3] was also measured and it had a latency near **4 - 5 ms** with all effects turned on. We have to take in account that a multi-effects processor has specially optimized hardware and firmware for its purpose. The Android platform is not designed for real-time applications.

The library includes an enhanced error handling mechanism. An object of the polymorphic `llaErrorHandler` class is delivered to the library at instantiation and all error, warning, and debugging messages are sent with the error handler object. The actual output can be redirected by overriding the `rawlog()` method of the error handler. No messages can be delivered through the *stderr* or *stdout* descriptors in the Android environment because they are managed by a special logging system called *Logcat*. The messages from the `llaudio` and the overlying DSP service are sent to this logging facility for the Android build instead of the standard terminal output.

Simplicity of usage was an important goal. A connection of an input with an output stream might look like as the following example:

[3]See the reference audio interface in appendix A.

```
// The library instance
llaDeviceManager& devman = llaDeviceManager::getInstance();

// get the system default audio device and its default streams
llaDevice& device = devlist.getDefaultDevice();

llaOutputStream& ostream = device->getOutputStream();

llaInputStream& istream = device->getInputStream();

// get an audio pipe and connect the stream with it
llaAudioPipe apipe;

apipe.connectStreams(istream, ostream);

// destroy the library instance
devman.destroy();
```

The reader may notice that the `connectStreams()` method will occupy the calling thread and the last line is never executed. The connection of the streams breaks if the `bool stop()` method of the pipe returns true or if one of the streams are broken. For example, the connection would stop immediately by removing the audio interface. The `read()` and `write()` methods of the streams placed in a loop would have similar results. The `connectStreams()` method is intended to provide an optimized audio data flow.

The internal sequence types mentioned in the design period are solved with the `llaContainer` template class. All objects of this type can provide an iterator to be used for iterating through the contained elements. A default realizations with STL sequences are located in the `defaultcontainer.h` header file.

4.3 Plug-in hosting service - implementation details

The DSP module is implemented in C++ to be a native software. It can be built for any architecture with a cross-compile GCC tool-chain. Initially, it supposed to be a shared library connected with the Android UI through the JNI interface. Further investigation of the problem revealed a security issue with opening sound devices in the `/dev` directory. The design was extended with a messaging system and a client-server architecture that is useful if considering multiple UI clients. The interface for the DSP service is now completely up to this messaging mechanism and direct calls to the public `DspServer` methods are not really thread safe. Messages are encoded using the JSON protocol which is simple and very intuitively translates an object state into a serialized form.

A few external open-source libraries are used by the module. The *JsonCpp* library for parsing, building and storing JSON messages, the *LADSPA plugin framework* to work with LADSPA plug-ins, and finally, the many times mentioned *salsa-lib*. The JSON protocol is also used for the effect database as it is already linked to the binary file. The prototype application uses a hard-coded database in the `database.h` header file[4], but a stand-alone JSON file is also usable.

[4] generated from the `effects.json` file

Error handling is done with simple functions behaving differently on various platforms. The Android build uses the *logcat* terminal for error messages. The utilized `llaudio` library is instantiated with an `llaErrorHandler` object that uses the logging output of the DSP service.

The utilities for concurrency and multi-threading are captured on the Figure 4.4 by the abstract class `Thread, Mutex`, and the interface `Runnable`. This approach helps to avoid any dependency on a particular platform and makes easier to track future errors of concurrent algorithms. An object of type `Thread` is a descriptor for an execution thread and it can take a `Runnable` object to run it in this thread. Critical sections can be realized with `Mutex` objects by placing the code between the `lock()` and the `unlock()` method.

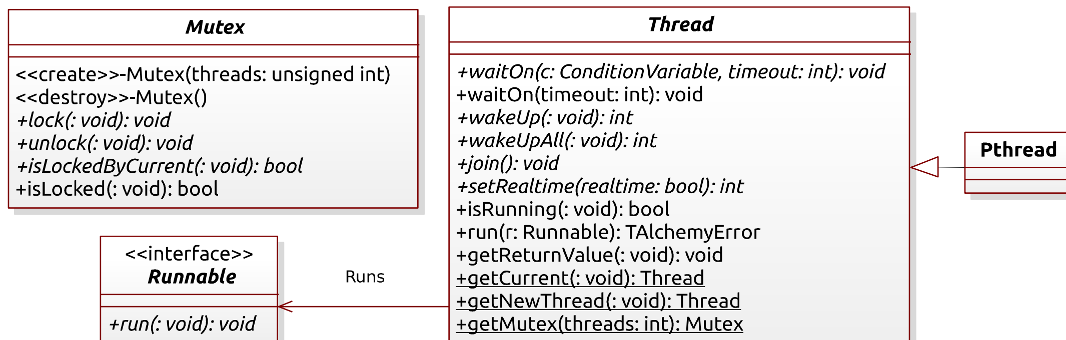


Figure 4.4: Classes for concurrency

The signal processing runs in a separate execution thread with the highest possible priority and real-time scheduling, synchronized with the controlling parent thread. It is not fortunate to change an effect parameter while that effect is running. Thread-safeness is generally not quarantined for all plug-ins. This implies the use of semaphores. Concurrency is managed with the *Pthread* library which has a *mutex* structure for such purposes. An uncontended mutex can be locked and unlocked with a couple of machine instructions in modern Linux systems maintaining critical sections suitable for soft real-time applications. Memory allocations are avoided in the processing pipeline.

The architecture of this module has many potentials in terms of extendibility. As an example, a `DspProcess` object with an appropriate `ProcessingGraph` can process an audio pipe while another pair of these objects can process a different pipe. Practical meaning is that a musician can have an electric guitar and a microphone for vocals. These are two different signals requiring different effect chains. In some interpretation, the code is a framework to make virtual pedal-boards for many platforms. This extendibility is a natural consequence of object oriented design.

The most promising filter collection was the open-source and freely available *C* Audio plug-in Suite*[5]. This is an LADSPA plug-in library with the most essential electric guitar effects and utilities. Includes a phaser, flanger, chorus, compressor, distortion, reverb, delay, and many amplifier and cabinet models. This collection is enough to compose a traditional pedal-board in a virtualized form. Regrettably, the collection has many bugs and the amplifier models (which are perhaps the most important models) are crashing randomly in

[5] Abbreviated as CAPS

the latest release. Other filters, like the reverb, chorus, and phaser are working correctly and have a very authentic sound. A lot of plug-ins are available in LADSPA and other formats waiting to be utilized in the future. Main candidates are the *Guitarix* filters[6], and the *ScorchCrafter*[7] VST amplifier model.

The NDK offers a script for obtaining stand-alone tool-chains and the provided CD contains a reference `arm-android-eabi-gcc` compiler with the standard C and C++ libraries found in the NDK. The source code can be compiled with a traditional `make` command using the provided makefiles and no other tool are required.

4.4 Discussion of the implemented user interface

A very simple UI has been created to control the DSP service. The processing can be started and stopped, the list of available sound devices can be obtained and the buffer size is adjustable. Much more important is the synchronization of these two components and their communication. The UI client has a few classes implemented in Java for this task. The Android code has the traditional project structure described in [1]. It contains an `Application` class specialization, the `AlchemyConnector` that is a bridge to the DSP service. UI events generate messages and these are sent to the server with a passive time-out secured waiting for the reply. If no reply is received, the service has probably crashed. The `AlchemyServer` is a Java interface declaration for the service. This interface is realized by the `AlchemyConnector` class. The `AlchemyMessage` class encapsulates the JSON messages.

The `assets` folder of the *apk* package contains the executable binary file for the DSP service which is installed into the application's data directory. The location is managed by the Android system and obtained with an API call. After the service is installed and started, the UI builds the connection with the service through the standard input and output file descriptors.

Other mobile devices than the reference were not tested although there is nothing to prevent the application to run if the requirements are met. The kernel module for USB-audio functionality can be provided in the `assets` folder and loaded on start-up if a distributable version is considered.

[6]available at http://guitarix.sourceforge.net

[7]available at http://scorchcrafter.destructavator.com

Chapter 5

Conclusion

The main goal of this work was to take the sound of an instrument, process it with a mobile device, and to hear a nice guitar sound on the output without significant delays. The solution has to run on Android, the most popular mobile platform available at the moment that is designed for stability, reliability and not for real-time applications. The thesis provided an overall image for the reader to become familiar with the key notions and actors of the problematic. Additionally, an available solution was declared for each obstacle that occurred during the design period.

After this overview, an ambivalent summary can be made considering the big number of difficulties that the Android platform brings to the field of real-time or at least soft real-time applications and audio oriented software. A few workarounds had to be invented, with each one decreasing the deployability of this idea as an Android application. However, even a working prototype is really valuable, since no official support is released from the main contributors of the platform.

This work has no resources to solve problems like the general low latency USB sound interface support as it would require a special AOA compliant hardware design and additional software support to be compatible with the majority of Android devices. Solving the latency issues was the highest priority task as it represents a major problem of the platform. The creation of a usable and pleasing graphical interface is more of a graphical and time consuming work as an engineering problem. Nevertheless, a complex UI can be built without barriers with the created tools.

The result of this work has accomplished the main goals of the project, guitar sound is created without noticeable latencies. It is very hard to find a commercial application for Android that is able to do this task, probably none is available yet. On the other hand, these applications will be available as soon as a suitable API will enable such usage on a wide range of devices. This involves the standardization of hardware requirements, and a software optimization in the official SDK. Thanks to the versatile architecture, the created software will be still usable, and doing so without requiring any kind of rooting procedure.

Bibliography

[1] Android. Get the android SDK. http://developer.android.com/sdk/index.html.

[2] Teragon Audio. How to make your own VST host. http://teragonaudio.com/article/How-to-make-your-own-VST-host.html, 2012.

[3] Jon Chappell. A guide to guitar effects [online]. http://www.harmonycentral.com/docs/DOC-1351.

[4] Ph.D Ing. Jiři Schimmel. Studiová a hudební elektronika. Technical report, FEKT VUT v Brně, Purkynova 118, 612 00 Brno, 2012. ISBN-978-80-214-4452-2.

[5] Jaroslav Kysela, Abramo Bagnara, Takashi Iwai, and Frank van de Pol. Alsa project - the c library reference. http://www.alsa-project.org/alsa-doc/alsa-lib/pcm.html, [cit. 2013-05-05].

[6] Remi Lorriaux. Real-time audio on embedded linux [online]. http://elinux.org/images/8/82/Elc2011_lorriaux.pdf.

[7] Ed Mitchell. How to build a guitar pedalboard. http://www.musicradar.com/tuition/guitars/how-to-build-a-guitar-pedalboard-553855, 2013-01-20 [cit. 2013-05-06]. Total Guitar.

[8] Dave Phillips. A user's guide to alsa. http://www.linuxjournal.com/node/8234/print, 2005-06-30 [cit. 2013-05-06]. Linux journal.

[9] W. Pirkle. *Designing Audio Effect Plug-Ins in C++: With Digital Audio Signal Processing Theory*. Taylor & Francis, 2012. ISBN-9780240825151.

[10] Maxim Integrated Products. Usb on-the-go basics. http://pdfserv.maximintegrated.com/en/an/AN1822.pdf, 2002-12-20 [cit. 2013-05-06].

[11] Sylvain Ratabouil. *Android NDK Beginner's Guide*. Packt Publishing, January 2012. ISBN-978-1-84969-152-9.

[12] Thorsten Schreiber. Android binder – android interprocess communication. Master's thesis, Ruhr-Universität Bochum, Oct 2011.

[13] Jeff Tranter. Introduction to sound programming with alsa. http://www.linuxjournal.com/node/6735/print, 2004-09-30 [cit. 2013-05-06].

Appendix A

Development Environment

Development operating system:	`Ubuntu Linux 12.10, kernel: 3.6.3-030603-generic`	
Reference Android device:	`Samsung Galaxy Nexus`	
	Variant:	`Maguro 16GB I9250XXLF1`
	HW version:	`9`
	Serial number:	`0A3C27F41501700B`
	Android version:	`Jelly Bean 4.2.1`
	Build number:	`JOP40D.I9250XWMA2`
	Kernel version:	`3.0.31-gd5a18e0 android-build@vpbs1`
		`Fri Nov 2 11:02:59 PDT 2012`
SDK tools revision:	`20.0.3`	
NDK revision:	`r8b`	
Development IDE:	`Eclipse 3.8.0`	
Measurement tools:	`GW INSTEK GDS-2104 - Oscilloscope`	
Reference audio interface:	`Digitech RP250`	
	Firmware version:	`1.6`

Appendix B

Contents of the provided CD

- The source code with a doxygen generated class reference
- The LaTeX source text for this document
- The reference Android GCC tool-chain
- Poster
- Video