Efficient Integration of Audio Effects in Android 2.3

Thesis submitted in partial fulfillment of the requirements for
The award of degree of

Master of Engineering
In
Computer Science and Engineering

Submitted By
Kailash Pathak
(Roll No. 800932010)

Under the supervision of:
Dr. V.P. Singh
Assistant Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004
June 2011

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, "*Efficient Integration of Audio Effects in Android 2.3*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Computer Science and Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Dr. V .P. Singh* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

Kailash Pathak

Roll. No. 800932010

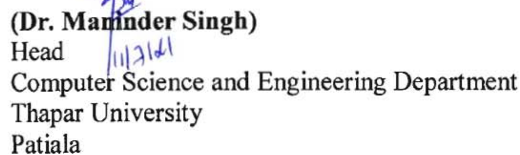
This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Dr. V. P. Singh

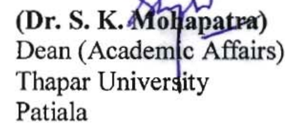
Assistant Professor

CSED, Thapar Universiy
Patiala

Countersigned by



(Dr. Maninder Singh)
Head
Computer Science and Engineering Department
Thapar University
Patiala



(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

Acknowledgement

No volume of words is enough to express my gratitude towards my thesis supervisor Dr. V.P. Singh. Whose guidance, wisdom and invaluable help have aided me in the completion of thesis. He has helped me to explore numerous topics related to the thesis in an organized and methodical manner and provided me with valuable insights in to various technologies.

I am thankful to Dr. Maninder Singh Head of CSE department, for extending his support. I am also thankful to Mr. Karun Verma, P.G. Coordinator, for the motivation and inspiration during the thesis work.

I would also like to thank the staff members and my colleagues who were always there at The need of the hour and provided with all the help and facilities, which I required for the completion of my thesis work.

Kailash Pathak
(Roll.No. 800932010)

Abstract

Android is a new mobile platform for mobile development, It has a rich set of applications. Android multimedia architecture is improving day by day. Multimedia applications are become very important part for the success of any mobile platform. Every day new idea is researched to improve the quality of audio/video for mobile platform. A part from quality, battery power management is also important. Battery power is always on the top of the requirement stack. In current android multimedia architecture, audio effects are handling with Advance RISC Processor (ARM), which is core processor of mobile devices. In this thesis research work, a new framework for the audio routing has been proposed. The change in audio routing stack is proposed to save the battery power. Audio effects are integrated with the help of digital signal processor instea of ARM processor.

New algorithm for the Bass booster and Virtualizer Audio Effects are implemented with respect of proposed framework. The power consumption of these audio effects are tested in the current and proposed android audio multimedia .The power consumption in proposed framework is less as compare to the current framework .

Table of content

Certificate ………………………………………………………………………………… i
Acknowledgement... ……………………………………………………………………… ii
Abstract…………………………………………………………………………………… iii
Table of contents………………………………………………………………………... iv
List of figures……………………………………………………………………… ... vii
1. **Introduction** ……………………………………………………………………… 1
1.1 What is android? 1
1.2 Android Architecture ……………………………………………………… 2
1.3Applications ……………………………………………………………… 3
1.4 Application Framework …………………………………………………… 3
1.5 Libraries ………………………………………………………………… 3
1.6 Android Runtime ………………………………………………………… 5
1.7 Linux Kernel …………………………………………………………… 5
1.8 Overview of Thesis ……………………………………………………… 6
2. **Literature Review ……………………………………………………………… 7**
2.1 Digital Audio Effects ……………………………………………………… 9
2.1.1 Reverb Effect ……………………………………………………………… 9
2.1.2 Bass Boost ………………………………………………………………… 9
2.1.3 Equalizer ………………………………………………………………… 9
2.1.4 Virtualizer ……………………………………………………………… 10
2.2 Android Media Framework ……………………………………………..... 10
2.3 Android Media Layers …………………………………………………… . 12
2.4 Service Manager……………………………………………………………. 13
2.4.1 AddService and GetService………………………………………………… 13
2.5 Service Provider Process……………………………………………………… 14
2.5.1 Client Process……………………………………………………………… . 15
3. Audio Routing in Android Platform ……………………………………… 16

3.1 AudioFlinger…………………………………………………………………. 16
3.1.1 Audio Hardware Interface and Audio Drivers………………………………… 17
3.2 Audio Routing…………………………………………………………….... 17
3.2.1 Audio Format………………………………………………………….….. 18
3.2.2 Compressed Audio…………………………..…………………. 18
3.2.3 Audio Effects…………………………………………………….…… 18
3.2.4 Three -D Gaming /Audio Stream Virtualization……………….……....... 18
3.3 Resource Management……………………………………….……….. 19
3.3.1 Doppler……………………………………………………………..… 19
3.3.2 Recommendations………………………………….……………..… 19
3.4 Routing Manager……………………………………………………. 19
3.4.1 Routing strategies………………………………………………………. 20
3.4.2 Routing work flow……………………………………………..…. 21
3.5 Interfaces…………………………………………………………….… 21
3.5.1 Volume and Mute Control…………………………..…………..…… 22
3.6 Audio Effects……………………………………………….………………. 22
3.7 Software Effects: Reverb and Equalizer…………………….….……… 22
4. **Problem statement**……………………………….……………………… **24**
4.1 Objectives ……………………………………………………………..….. 24
5. **Implementation and Experiment**…………………………......………**25**
5.1 Compressed Audio Framework ……………...………………...…………………… 25
5.2 Proposed Work Flow……………………….………....…………………….. 26
5.3 Audio Post/Pre processing..………...…….………….………...…………...….. 27
5.3.1 Effects plug-ins..……………………….……….………...….... ……. 28
5.3.2 Algorithm of Bass booster audio effects………...………………………….. 30
5.3.3 Algorithm of Virtualizer Audio Effects……………..…………………...… 31

5.4 Comparison of current & proposed approach………………………………..………. 35
6. **Conclusion and Future Scope**…………………………………………………...……. **37**
References……………………………………………………………………… 38
Appendix A……………………………………………………….………………...…..43
List of Publications ……………………..…………………………………………….. 49

Table of Figures

S.No.	Figure Name	Page NO.
1	JNI in Android	2
2	Block Diagram of Android Architecture	2
3	Core Component of Android Media	4
4	Android Runtime	5
5	Android Media Framework	11
6	Binder And JNI in Android Media Layer	12
7	Service Manager Process	14
8	IPC Mechanism Using Binder Driver	15
9	Block Diagram of Current and Propose Approach	26
10	Block Diagram of Current Audio Processing Stack	29
11	Block Diagram of Proposed Audio Processing Stack	29
12	Power Consumption in Unit Time for Bass Booster Effect.	35
13	Power Consumption in Unit Time for Reverb Effect.	36
14	Power Consumption in Unit Time for Virtualizer Effect.	36

CHAPTER 1

Introduction

1.1 What is Android?

Android is a software stack for mobile devices that includes an operating system, middleware and key applications. The android software development kit (SDK) provides the tools and application program interface (API) necessary to begin developing applications on the android platform using the Java programming language. It is an open mobile phone platform that was developed by Google and later, by the Open Handset Alliance. Google defines Android as a "software stack" for mobile phones. Android platform supports the Java native Interface (JNI) mechanism, and its own components library is written in C/C++, some functions interacted with the hardware are achieved through the JNI. Therefore, this approach does not violate the overall architecture of android. Design Goals of the Media Framework include:

- Simplify application development
- Share resources in multi-tasked environment
- Provide a strong security model
- Leave room for future growth

Since the source code of android was released to people, a large community of developers has organized around android. Due to its open-source advantage and powerful application program interfaces (APIs) android has attracted a large number of developers. Android's SDK is based on Java, how to reuse excellence C/C++ open-source projects is a problem. Java Native Interface (JNI) is the interface between the Java code running in a java virtual machine (JVM) and the native code running outside the JVM. It works both ways that is you can use JNI to call native code from your Java programs and to call Java code from your native code. The native code normally resides with in a library (.so file) and is typically written in C/C++.

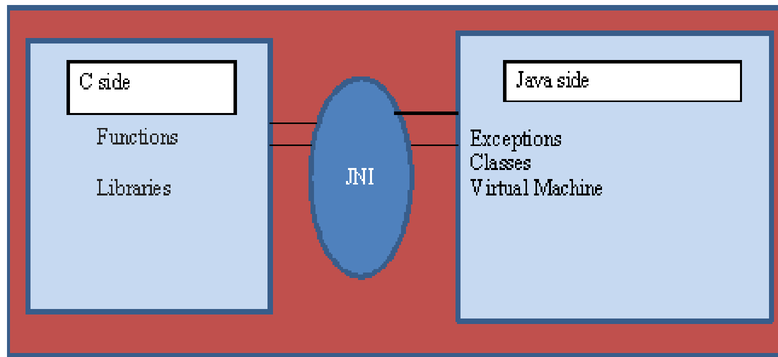


Figure 1. JNI in Android

1.2 Android Architecture

The following diagram shows the major components of the android operating system. Each section is described in more detail

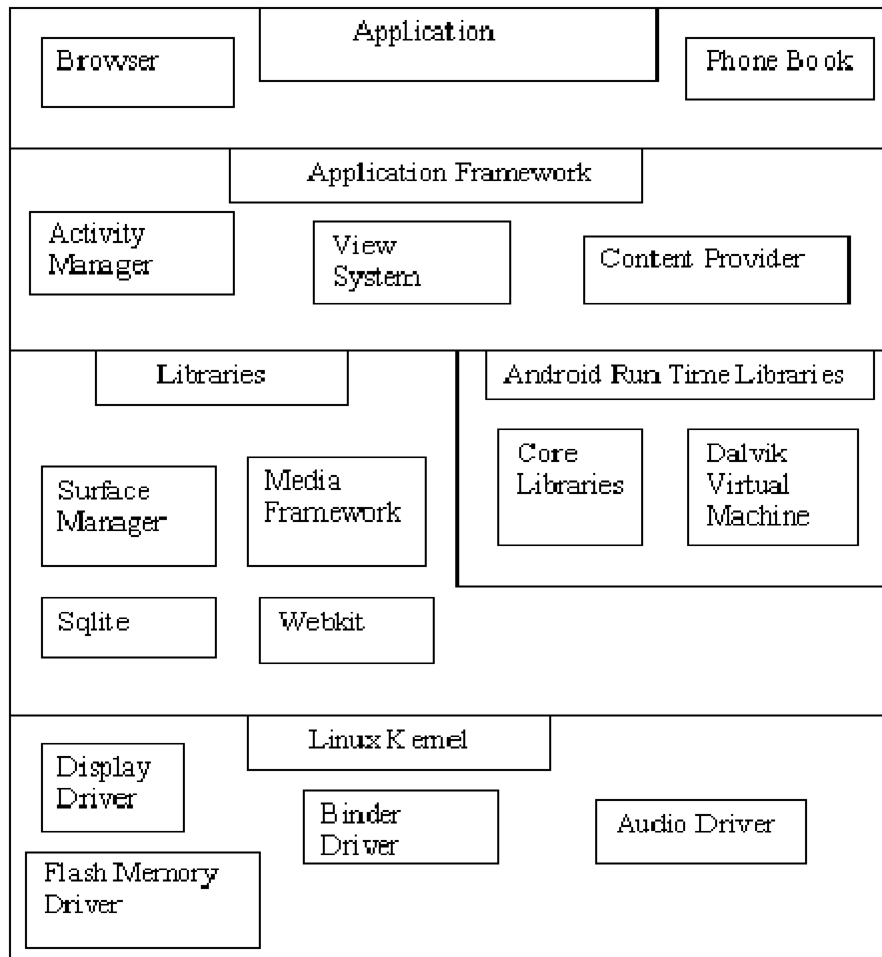


Figure 2. Block Diagram of Android Architecture

1.3 Applications

Android has a rich set of core applications including an email client, SMS program, calendar, maps, browser, Contacts, and others. All applications are written using the Java programming language.

1.4 Application Framework

Developers have full access to the same framework APIs used by the core applications. The application architecture is designed to simplify the reuse of components, any application can publish its capabilities and any other application may then make use of those capabilities (subject to security constraints enforced by the framework). This mechanism allows components to be replaced by the user.

Underlying all applications is a set of services and systems, including:

(a) A rich and extensible set of Views that can be used to build an application (i.e. lists, grids, text boxes, buttons, and even a embeddable web browser)

(b) Content Providers that enable applications to access data from other applications (such as Contacts), or to share their own data

(c) A Resource Manager, providing access to non-code resources such as localized strings, graphics, and layout files

(d) A Notification Manager that enables all applications to display custom alerts in the status bar

(e) An Activity Manager that manages the lifecycle of applications and provides a common navigation back stack

1.5 Libraries

Android includes a set of C/C++ libraries used by various components of the android system. These capabilities are exposed to developers through the android application framework. Some of the core components of android media are listed below

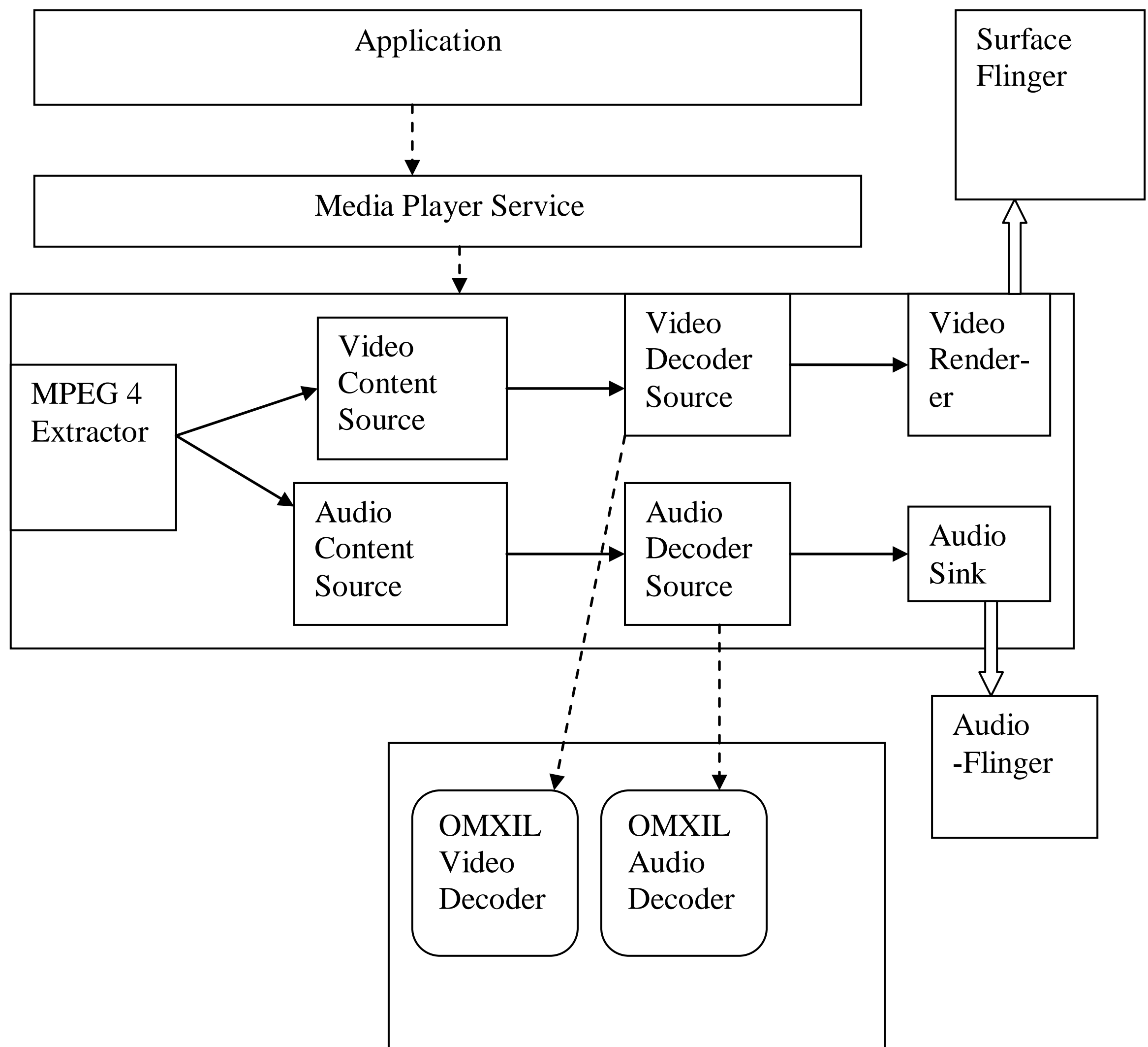


Figure 3. Core Component of Android Media.

(a) System C library – A Berkeley Software Distribution (BSD) derived implementation of the standard C system library (libc), tuned for embedded Linux-based devices.

(b) MPEG4 Extractor- MPEG4 Extractor is available in android.

(c) Surface Manager - Manages access to the display subsystem and seamlessly composites 2D and 3D graphic layers from multiple applications.

(d) AudioFlinger-AudioFlinger manages output/input streams, audio tracks/audio records and their corresponding playback/recording threads. AudioFlinger create new threads for every output stream.

(e) SGL (Skia Graphics Library) - Skia is a complete 2D graphic library for drawing Text, Geometries, and Images.

(f) **3D libraries** - An implementation based on OpenGL ES 1.0 APIs, the libraries use either hardware 3D acceleration (where available) or the included, highly optimized 3D software rasterizer.

(g) **FreeType** - bitmap and vector font rendering.

(h) **SQLite** - A powerful and lightweight relational database engine available to all applications.

1.6 Android Runtime

The android run time libraries include two parts. One provides most of the functionality available in the core libraries of the java programming language.

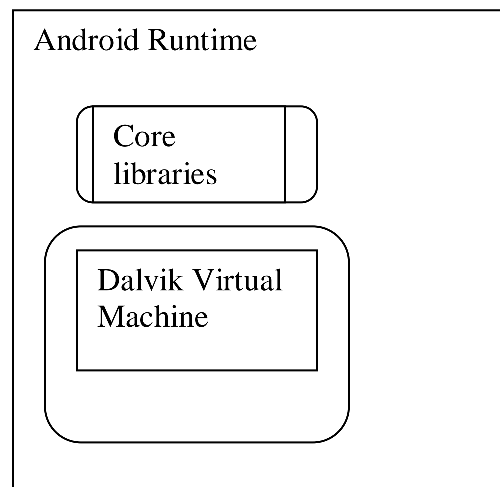


Figure 4. Android Runtime

The other one is android core libraries such as: android.os, android.net, android. Media, etc. Every android application runs in its own process given by the operating system (OS), and owns instance of the Dalvik virtual machine. Dalvik is a register based java virtual machine. The Dalvik VM is executing files in the .dex (Dalvik executable) format which was optimized for minimal cup-and –memory.

1.7 Linux Kernel

Android relies on Linux version 2.6 for core system services such as security, memory management, process management, network stack, and driver model. The kernel also acts as an abstraction layer between the hardware and the rest of the software stack.

1.8 Overview of Thesis

The rest of the thesis is organized in the following order:

Chapter 2 – Literature review section gives a brief survey of research going in the area of android multimedia. Android multimedia framework is discussed in detail.

Chapter 3 – Contains the detail study of audio routing in android platform.

Chapter 4 – After going through the literature review, the problem statement has been identified and defined in this chapter. Objectives of thesis research work are discussed.

Chapter 5 – in this chapter proposed solution is discussed in detail. This includes the proposed changes in the android multimedia framework to achieve the thesis's objectives.

Chapter 6 – Conclusion and future work has been discussed here.

Appendix – Here java code is written for implementing the audio effects.

CHAPTER 2

Literature Review

Google has released an open source platform for mobile devices. Android has new power management framework for power saving. Java uses Advance RISC processor (ARM) and Digital signal processor (DSP) framework to improve the performance.

Recognizing the important role played by embedded software in determining system power consumption, researchers have started to investigate techniques for software power analysis and power-efficient software design. Vivek Tiwari et al. discussed the first systematic attempt to model the power cost in embedded system. A power analysis technique is developed that has been applied to two commercial microprocessors-Intel 486DX2 and Fujitsu SPARClite 934. This can help in verifying if a design meets its specified power constraints. Further, it can also be useful to search the design space in software power optimization. Examples with power reduction of up to 40%, obtained by rewriting code using the information provided by the instruction level power model, illustrate the potential of this idea [3].

Cheng- Ta et al. introduced new approach for estimating power dissipation in a high performance microprocessor chip. The first step of this approach was to identify a characteristic set, which is sufficient to capture the microprocessor power and performance behavior. Next, a trace driven architecture simulation was run through the instruction trace to collect the characteristic profile. Heuristic rules were used to transform predefined macro block templates into full functional program after four synthesis phases [7].

Yanbing Li et al. introduced the first comprehensive framework that simultaneously evaluates the tradeoffs of energy dissipations of software and hardware such as caches and main memory. The trade-off between system performance and energy dissipation is also explored [9].

The System on chip (SOC) design paradigm, which enables integration of processors, peripherals, busses, and complex user-defined logic blocks, has fuelled research in co-estimation of hardware and software power consumption.

Luca Benini et al. formulated power management policies heuristically. They discussed a finite-state, abstract system model for power-managed systems based on Markov decision processes. Under this model, the problem of finding policies that optimally tradeoff performance for power can be cast as a stochastic optimization problem and solved exactly and efficiently. The solution is computed in polynomial time by solving a linear optimization problem [11]. Reducing switching activity would significantly reduce power consumption of a processor chip .Ching-Long Su et al. Discussed two novel techniques, Gray code Addressing and cold scheduling, for reducing switching activity on high performance processors. Cold scheduling is a software method which schedules instructions in a way that switching activity is minimized. They proposed an instruction-level power model and a number of simple compiler techniques for reducing power consumption [12].

There has been done enormous research work on application-level power optimization for mobile phones. Matt Calder et al. illustrated the significant energy savings that can be achieved via batch scheduling of recurrent mobile phone applications. Recurrent applications are primarily characterized by repeated execution, mostly as a background process, to perform a task periodically. A secondary characteristic is that these applications are also delay tolerant [24]. For phones with deep sleep modes, there is also a cost in terms of time for the phone to wake up and go to sleep. When these recurrent applications are left to schedule themselves independently of other recurrent applications, they can have a very negative impact on battery life by bringing the phone out of sleep mode an unnecessary number of times. Xiao et al. Explored the power characteristics of 3G and Wi-Fi but focus on mobile YouTube viewing [25]. Haitao Wul et al. discussed power saving mode (PSM) with integrated Wi-Fi Network Interface Card(NIC), in the research work they proposed footprint leveraging the Cellular information such as the overheard cellular tower IDs and signal strength, to guild the Wi-Fi scan. The number of unnecessary scans through the changes and history logs of the mobile user's locations has been studied [29].

For hand-held devices, a battery is the main power source. Therefore, a Central Processing Unit (CPU) with powerful functions is generally not used. However, an RISC Processor, such as ARM, is utilized to deal with control dominated works. And Digital Signal Processor (DSP) with powerful computation functions works with the CPU, which is responsible for general events, signal distribution, and interruption handling.

2.1 Digital Audio Effects

Kyungjin Byun et al. described the implementation of digital audio effect system on Chip (SoC) which integrates the embedded DSP core, audio codec IP, a number of peripheral blocks and various audio effect algorithms. Digital audio effects are used in various audio applications, such as multi-effectors, and mobile audio devices [21]. The main goal of digital audio effects is the modification of the sound characteristic of the input audio signal. There are many audio effect algorithms such as reverb, virtualizer, stereo widening, bass boost etc.

2.1.1 Reverb Effect

A reverb effect is the result of many reflections of sound that occur in a room and on the surrounding walls in a concert hall. From any sound source, there are direct and indirect paths. The sound through an indirect path is reflected, delayed, and attenuated. These reflected waves can again bounce off another wall before arriving at our ears, and so on. The reverb effect is the realization of a series of delayed and attenuated sound Waves [22].

2.1.2 Bass Boost

Bass boost is an audio effect to boost or amplify low frequencies of the sound. It is comparable to a simple equalizer but limited to one band amplification in the low frequency range. To turn off the bass boost, turn the knob to the minimum (counter clock wise/left).

2.1.3 Equalizer

Equalization is an effect that allows the frequency response of an output signal to be controlled. An equalizer produces an equalization effect that boosts or cuts certain Frequency bands to adjust the output audio sound. Filter bank composed of infinite impulse response (IIR)-type filters. Because digital filter processing requires repetitive and intensive computation, these IIR filters are implemented by a dedicated hardware block. The multiband equalizer can be realized using these filters. The filter coefficients for each IIR filter are supplied by the DSP and are calculated in the DSP in advance. The audio input signal is also fed into each filter by the DSP, and the filtered output is taken by the DSP.

2.1.4 Virtualizer

An audio virtualizer is a general name for an effect to specialize audio channels. The exact behaviour of this effect is dependent on the number of audio input channels and the types and number of audio output channels of the device. For example, in the case of a stereo input and stereo headphone output, a stereo widening effect is used when this effect is turned on.

A number of implementation techniques for audio effects are used, such as filters, delay lines, time-segments, and time-frequency representations [21]. Although audio effect algorithms and their realization by software have been extensively studied .Fan Peck et al. discussed the design of an audio processor card for Generating special sound effects. Designed a card that plugs directly into the Industry Standard Architecture (ISA) bus of a personal computer (PC). The card uses a Motorola DSP processor, DSP56001, for audio signal processing. External static RAM (SRAM) modules are used for program and data storage. A codec chip is employed to handle the digital interfacing between the DSP processor and the analog audio world [44]. Few studies have been devoted to the complete system implementation of audio effects. The advantages of the implementation of audio effects using a dedicated hardware are very high throughput and low latency. However, implementing audio effects in hardware is more difficult and time-consuming than in software.

2.2 Android Media Framework

Maoqiang Song et al. explored the Android media framework. Media framework is top of all the media libraries, like OpenCore, Vorbis and Sonivox. So goal of android media framework to work as interface between these libraries and the services provided by these libraries to use by end user [8]. Below figure shows the dependency between libraries of the media framework.

In the figure 5, green components are media libraries, yellow components are android internal libraries, and grey components are external libraries. Light blue class is the java class which is the consumer of the media framework. All the classes (except for android.media.MediaPlayer) all components are built in c/c++. The core of android media framework is composed of libmedia_jni, libmediaplayerservice, libmedia.

Libmedia_jni is fit between java application and native library. It implements the JNI so that, java application use it then it implements the facade pattern.

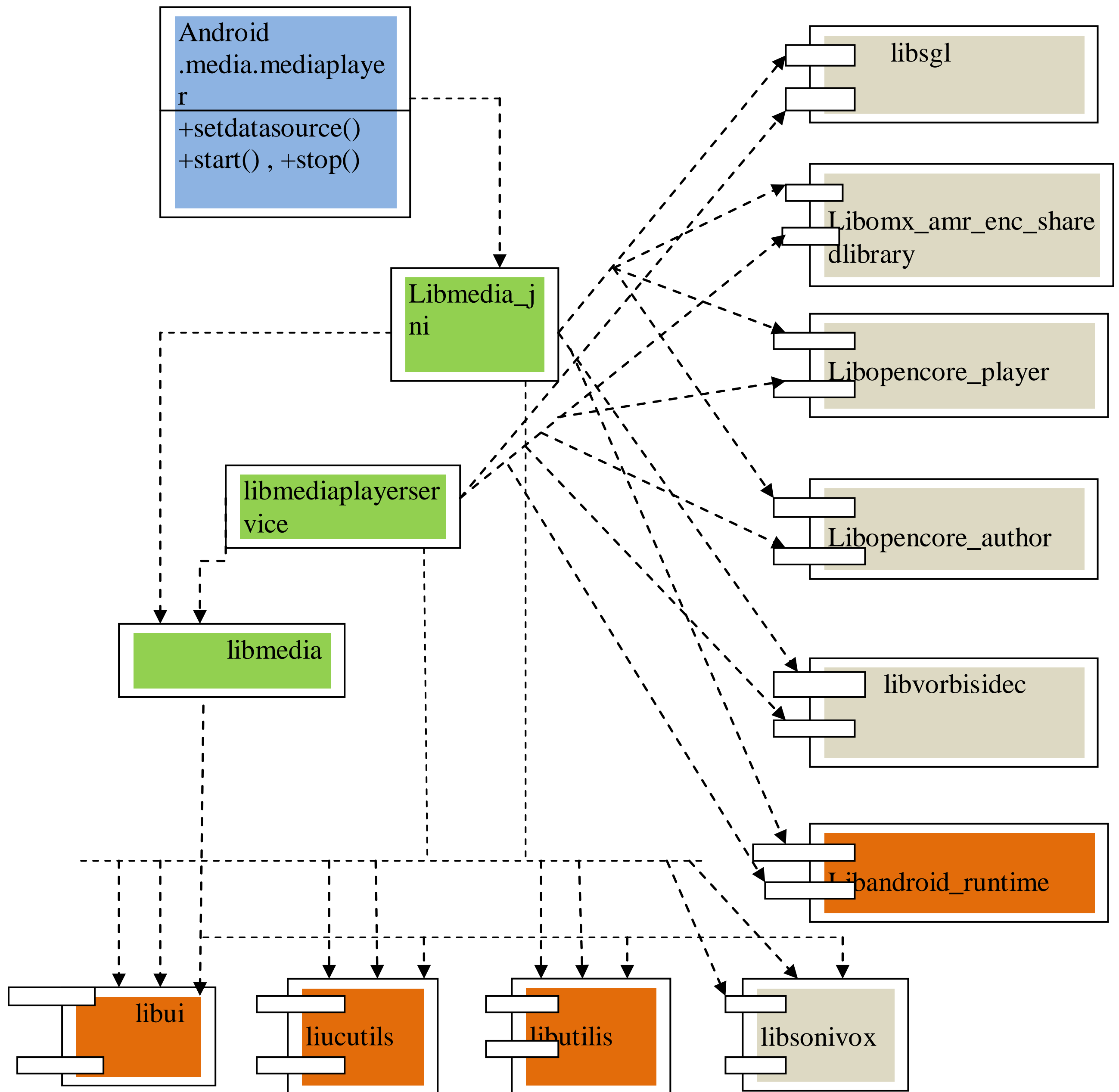


Figure 5. Android Media Framework

.

Libmediaplayerservice implements some of specific players and the media service which will manage player instances.

2.3 Android Media Layers

Maoqiang et al. [8], in this paper they described the android media layers. Media player is an important part of android media framework. It is used to control the playback of audio and video. Take the example of audio system to illustrate the workflow from the top layer applications to the bottom layer hardware output. The work flow is shown in Figure 6.

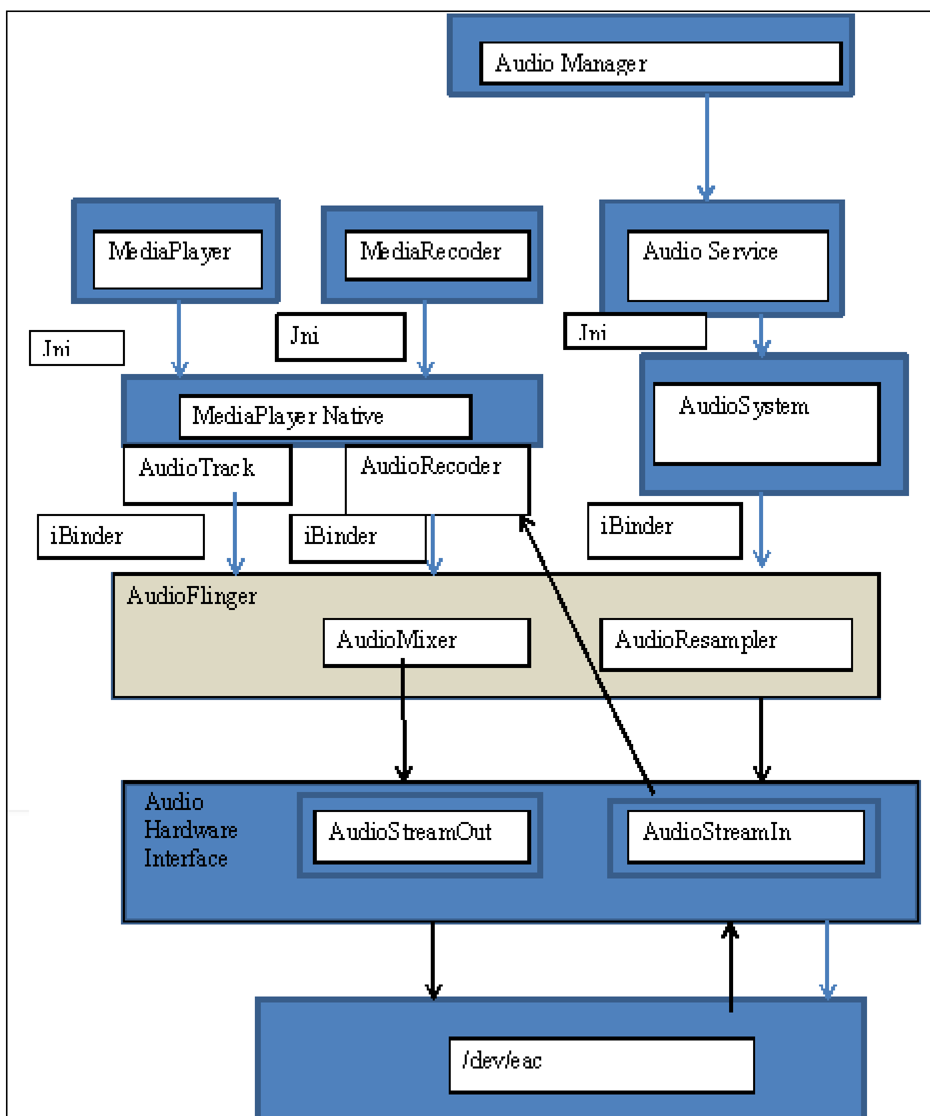


Figure 6. Binder and JNI in Android Media Layer [8].

Java applications use classes MediaPlayer to control the playback of audio/video Methods of MediaPlayer such as play (), setDataSource () are implemented in C/C++.

2.4 Service Manager

Service manager is a special process which manages all the services in the android framework. Upon start this process opens an instance of the binder driver by calling open("/dev/binder"). It then makes a call to ioctl(BINDER_SET_CONTEXT_MGR) to notify the system that it is acting as a service manager [18]. The service manager then enters into a loop waiting for requests from other processes. In the loop the service manager keeps on calling ioctl(BINDER_WRITE_READ) to get requests from other processes. Upon getting a request from a process it sends an appropriate reply to the process by calling ioctl(BINDER_WRITE_READ) .

A process can communicate with the service manager for mainly two reasons:

1) Adding itself as a service in which case it acts as a service provider and

2) Getting a desired service in which case the process acts as a service client. Therefore the service manager normally has to handle mostly two kinds of requests.

2.4.1 AddService and GetService

Each service provider process which interacts with the binder driver has a handle through which the binder driver recognizes it. For service manager process this handle is 0. All the other processes get a non-zero handle [40]. The communication through the binder driver happens by sending data in a particular format called a **Parcel**. A parcel will contain formatted data (along with handle to the process with which it wants to communicate) and along with the parcel a code is sent. The above requests to the service manager (AddService and GetService) are also sent as a parcel with code BC_TRANSACTION.

The control flow of service manager can be nicely summarized throuh the following figure 7.

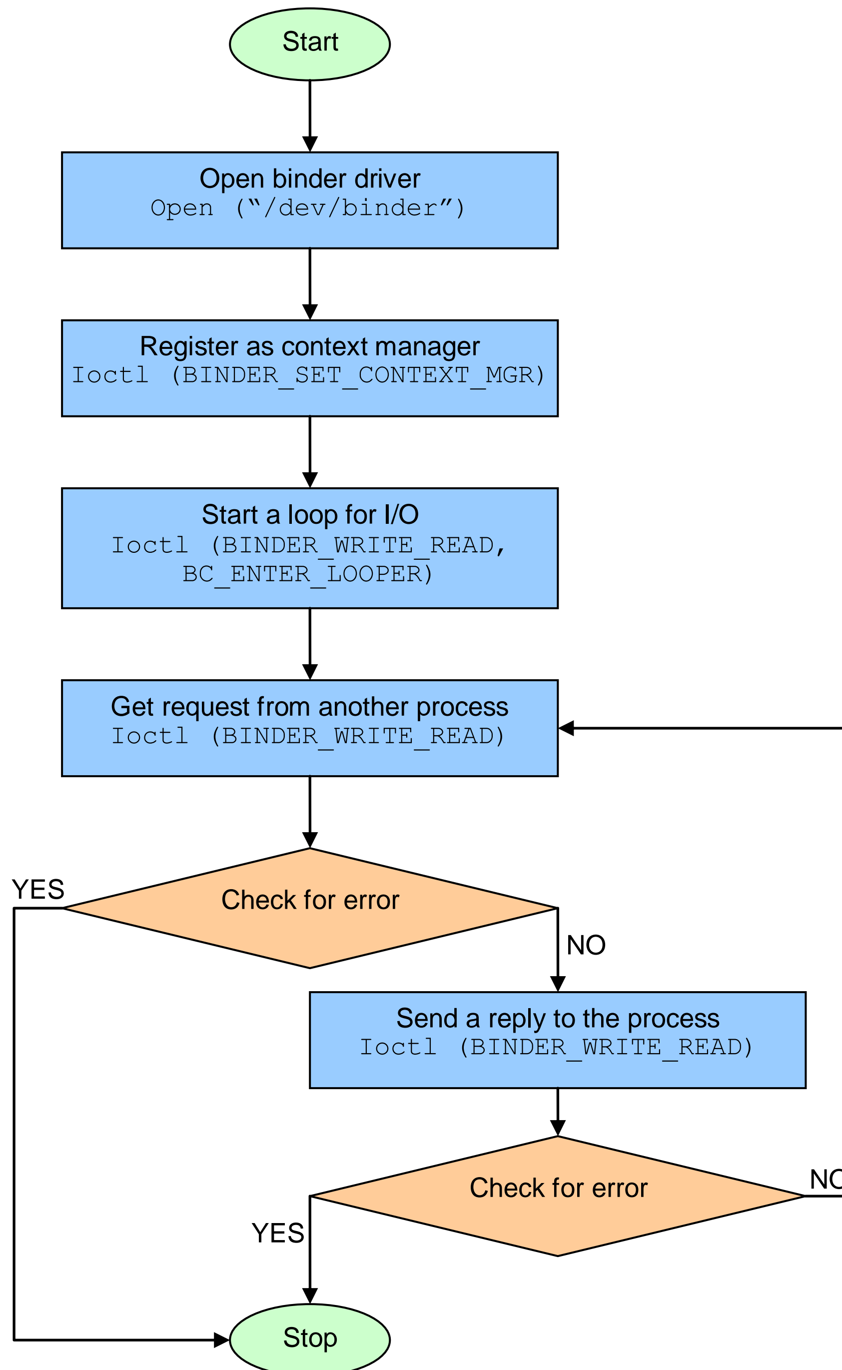


Figure 7. Service Manager Process

2.5 Service Provider Process

A process which wants to provide any service must first register itself with the service manager. This is done by first getting an instance of the service manager process and then sending it an AddService request using a parcel. This time the parcel data contains an address of the service instance (along) in the service provider process space. This way the binder driver in the kernel is notified of the service provider. A

client process can obtain handle to the service by sending a GetService request to the service manager. A process can provide as many services as it wants, to the client each service will be different and the client does not need to know whether a single process is providing all services or not. After instantiating the service, a service process creates a thread to wait for data arriving from the binder driver. This is simply a loop in which it calls ioctl (BINDER_WRITE_READ).

2.5.1 Client Process

A client process wishing to use a service must first ask the service manager for the desired service. This it does by sending a GetService request to the service manager. The service manager returns an appropriate handle associated with that service to the client. The client can now use the handle to use that particular service.

The interaction of client, service provider and service manager can be shown in the following diagram.

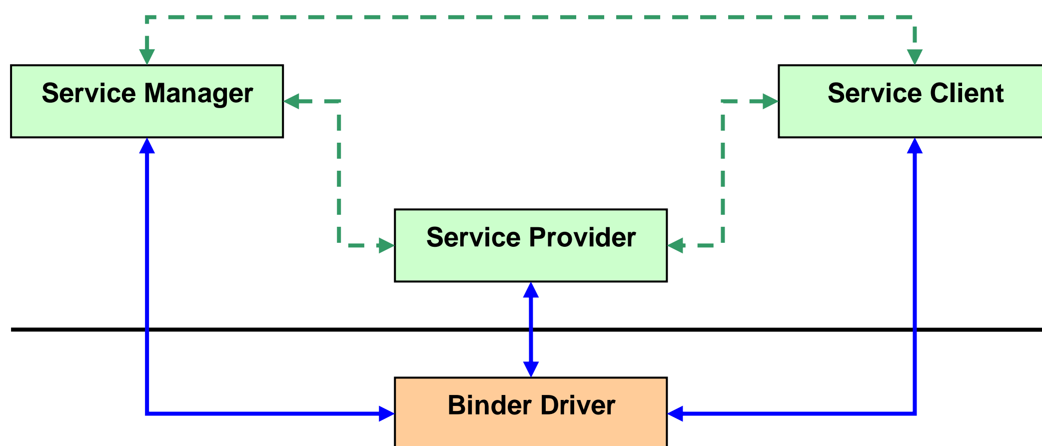


Figure 8. IPC Mechanism Using Binder Driver

To the client it looks as if it is interacting directly with the service manager as well as the service provider whereas all the interaction occurs through the binder driver.

Android has insulated the layer above the binder driver into a set of C++ classes and interfaces (abstract classes with no data members).

CHAPTER 3

Audio Routing in Android Platform

The management of audio routing is scattered among various applications, services and parts of the audio framework. Such as Phone Application, headset Observer, Advanced Bluetooth Audio Distribution Profile Service (BluetoothA2dpService), audio service, audio system, AudioFlinger, audio hardware etc. With regards to routing, the phone can be in one of three modes: NORMAL, RINGING, IN_CALL.

The mode is selected by the phone application via AudioManager setMode ().

To each of these modes corresponds a routing of the audio output to one of the following audio devices:

1. Speaker
2. Earpiece
3. Headset
4. Bluetooth Synchronous Connection Oriented (BT SCO).

Audio (acoustic) post/pre-processing is controlled by audio hardware according to the requested routing. The routing can be modified by some services or phone application (enable/disable BTSCO, enable/disable speaker phone), HeadsetObserver (enable/disable headset). An additional layer of routing is added for A2DP headset. A2DP output is enabled/disabled by BluetoothA2dpService via AudioManager, and handled by audioFlinger. AudioFlinger uses a second mixer thread and hardware interface dedicated to A2DP. Tracks are routed to this output according to their stream type. The hardware output is duplicated to the A2DP output. When playing over certain stream types (RING, ALARM, etc) the routing can also be altered so that the sound is always output at least to the phone speaker. This is handled by audioFlinger.

3.1 AudioFlinger

AudioFlinger is mainly in charge of re-sampling and mixing the audio streams (AudioTracks). It also offers interfaces to control audio output routing.

AudioTracks have a stream type attribute that is used to control common volume settings and some routing decisions. The stream types are:

1. MUSIC
2. RINGTONE
3. NOTIFICATION
4. ALARM SYSTEM
5. VOICE_CALL
6. BLUETOOTH_SCO

It supports only two hardware interfaces, each one with a single audio output stream, opened and configured at init (android booting) time. One output is for blue tooth A2DP devices, the other one for all other audio devices attached to the platform.

Hardware and A2DP outputs can be active simultaneously the output to which a given track is sent is selected based on the stream type. The AD2P output is automatically disabled when hardware output is routed to Bluetooth SCO. The routing of hardware output is forced speaker for certain stream types (RINGTONE, NOTIFICATION, and ALARM). Only one audio input is supported without resampling. The input sampling rate is selected when opening the input.

3.1.1 Audio Hardware Interface and Audio Drivers

The output can be routed to only one device (hardware) at a time except speaker and wired headset. Output post processing is applied according to the audio routing:

1. Compression, equalization (EQ), infinite impulse response (IIR) (Speaker correction)
2. Input pre-processing is applied with different parameters according to the type of Bluetooth device connected
3. Automatic Gain Control (AGC), COMPRESSOR, EQ, IIR

3.2 Audio Routing

Audio routing supports multiple hardware outputs and inputs simultaneously. When playing an audio file, the AudioFlinger will call the policy manager's StartOutput method. The policy manager has few configuration methods:

SetForceUse - Gives priority to a specific device for the audio output.

When Speaker phone mode is selected during a voice call, setForceUse would be called with FOR_COMMUNICATION, FORCE_SPEAKER.

SetDeviceConnectionState - Marks an audio device as available or unavailable. This is used for accessories that can be disconnected (e.g. BT SCO, A2DP).

SetPhoneState - Called by the phone application, and tells the policy manager whether the current state is IN_CALL, RINGTONE or NORMAL (idle). This method takes care of doing the necessary rerouting by calling setOutputDevice,

3.2.1 Audio Format

Audio routing supports hardware input/output dynamic reconfiguration. In audio dynamic reconfiguration is the ability to add or remove resources (like sampling rate, buffer size).

3.2.2 Compressed Audio

Android multimedia framework has compressed audio APIs for processing compressed audio. Applications can send compressed audio directly to hardware for enable some kind of hardware tunnelling mode. When A2DP headset is connected or low power mode is selected the switching to/from hardware tunnelling to software/hardware decode mode is performed. Hardware tunnelling do faster decoding than real-time decoding. Audio framework Feeds a large pulse coded modulation (PCM) buffers to allow more frequent power collapse and power savings. It provides the control of volume and audio post processing to the hardware tunnelling.

3.2.3 Audio Effects

These effects provide a software implementation of environmental reverberation and graphic equalizer effects. It allows insertion of audio effect plug-ins on each track and replacement of default SRC algorithms. It also provides APIs to get/set hardware acoustic parameters for pre/post audio processing.

3.2.4 Three -D Gaming / Audio Stream Virtualization

Specialization, Three-D tracks differ from regular tracks in that:

1. They require a Three-D panner which can be integrated in the main mixer.
2. A simple mapping of the Three-D egocentric coordinates to the traditional 2-Channel pan. Here Three-D panning and mixing with other tracks can happen in the same mixer stage.

3.3 Resource Management

Mechanisms for querying whether 3-D voices are available or are required. A software fall back may or may not be available when the maximum number of hardware accelerated 3-D voices has been reached. In OpenSL ES Spatialization a player must request the 3-D-panning-related interfaces during the creation of the object. These interfaces are SL 3-D GroupingItf, SL3DLocationItf, SL3DSourceItf, SL3DLocationItf and SL3DSourceItf.

3.3.1 Doppler

The SL3D Doppler interface is dynamic (i.e. can be requested anytime during the life cycle of the object) though, implying that any track be able to use its SRC for Doppler (target SR can be continuously adjusted).

3.3.2 Recommendations

Next section lists some high level proposal for a new audio routing and mixing. The goal is to preserve as much as possible the existing audio framework implementation in order to limit the risk for regressions and reduce development time. To proposed modifications are primarily in the domains of audio routing, compressed audio, effects and audio inputs.

3.4 Routing Manager

As stated above, routing management in current implementation is scattered in different layers and modules of the media framework, making it hard to understand and maintain. The proposal is to regroup all the routing intelligence under a single routing manager. Other applications or services won't send direct routing requests anymore to AudioManager but instead will send events (e.g. indicating a state change or a device connection) that will be processed by the routing manager and transformed to a routing request send to audio hardware layer. Routing manager gives the control of the physical audio source device to the application in virtual mode, which is not directly visible to the application. The routing strategy becomes more complex implementation of different layers or modules.

The routing manager will be implemented in the native code; it must interact directly with other native classes (AudioTrack, AudioFlinger) and facilitate the future development of native applications with the native development kit (NDK).

3.4.1 Routing Strategies

A routing strategy is a number of rules associated to a "virtual" audio output that completely defines the routing decision for output under a given set of variables. An important point is that a given audio stream type (MUSIC, RING...etc.) can only be part of a single routing strategy. Thus the routing of all audio tracks pertaining to this stream type will be identical under a given set of conditions and they will be handled by the same software mixer and virtual output.

A virtual audio output is an output of one of the software mixers in the AudioFlinger and can be routed to one or more physical audio device.

Media strategy: OutM

Main use: Media "hifi" output

Covers: STREAM_MUSIC (STREAM_MEDIA)

Characteristics:

• No stream duplication,

• Always outputs to output devices that support high(er) sample rates (> 22 kHz).

Variables:

• (global) [affects routing] media is private / media is public (optional new setting, "media is public" by default)

• (global) [affects volume] "in-call => mute all media": yes / no

• (global) [affects volume] silent mode on / silent mode off

Order of selection by availability:

• Media is public: audio effects digital / BT A2DP (speaker / headphones / generic) / wired headphones / speaker

• Media is private: BT A2DP headphones / wired headphones / ear piece.

Telephony Strategy: OutP

Main use: phone output.

Covers: STREAM_VOICE_CALL, STREAM_BLUETOOTH_SCO

Characteristics:

• No stream duplication

• Potential routing to low-sampling (<22 kHz) rate devices.

•Variables:

• (application) use BT / use speaker / use neither speaker nor BT

• (global) [affects routing] "in call" / "off call" (set by application, e.g. Phone application sets "in call") order of selection by test in the following order:

• Use BT: BT phone car kit / BT headset / wired headset / ear piece.
• Use speaker: BT phone car kit / speaker.
• Use neither speaker nor BT: wired headset / ear piece.

3.4.2 Routing Work Flow

Routing modifications take place when a change in some variables governing a given strategy occurs:

a) A removable device is connected/disconnect. For instance playing music and a wired headset is connected: reroute OutM output from speaker to headset.
b) The phone state changes: Enable the simultaneous output of a ring tone to both speakers an headset by opening a second output routed to headset if the phone state changes to RINGING.
c) The user selects phone speaker from the in call screen: The outP is routed to speaker Phone Routing decision for each track follows the applicable strategy.

Routing decisions can yield to:

a) Opening/closing a secondary hardware output: A default output is always active but if the use case requires a second output, a new one is opened.
b) Reroute an existing output to a different device or set of devices
c) Route a track directly to a hardware output. This can happen if the audio format is compressed or if explicitly requested when opening the track.

3.5 Interfaces

Some methods of The JAVA application interface (AudioManager) will deprecate. This concerns methods allowing direct control of routing like SetAudioMode (), SetBluetoothScoOn(), SetWiredHeadset(). The routing manager will instead listen to intents broadcast by Phone Application, headset Observer or Bluetooth Service and decide of appropriate routing. The AudioFlinger interface will be modified to allow the routing manager to open and close audio outputs, set the routing for a particular output and attach a particular AudioTrack to a given mixer.

The AudioHardwareInterface will allow specifying the routing for each output independently. Available devices will be exported at the audio hardware interface.

3.5.1 Volume and Mute Control

Volume and mute control requests received by AudioFlinger for a given stream type are forwarded to all mixer threads. Software mixer threads apply this volume at the mixing stage for each track belonging to this stream type. For the hardware tunnelling thread, the request is simply forwarded down to the hardware audio output by the setVolume () method and handled by the hardware decoder or mixer.

3.6 Audio Effects

The audio effects are addressed here in three section covering:

a) The effects implemented in software and applied to an audio output mix.
b) The post processing implemented in hardware and applied on a particular path or route.
c) The effects implemented in software and applied to a single track.

3.7 Software Effects: Reverb and Equalizer

The software effects targeted for Android are an environmental reverberation and a graphic equalizer as described in OpenSL ES specification. The implementation will reuse as much as possible the existing audio framework concepts, classes and Binder interfaces. It is also recommended to reuse audio effects implementations already present in some android libraries (e.g. Sonivox EAS).

The implementation can be broken down into three major components:

a) Effect engine library: contains the actual audio effect process engines.
b) Effect control interfaces: a set of binder interfaces enabling the instantiation and control of any effect type by the applications.
c) Effect management in AudioFlinger: controls the effect module instantiation, its control.

By a given client (only one has control at a time) and its insertion in the audio path. The effect engine library is meant to propose a set of audio effect modules that can be used in various Android modules. For instance, a reverb engine can be used in the AudioFlinger as global effect device but also in the Sonivox EAS applied only to Musical Instrument Digital Interface (MIDI) synthesis. These libraries are dynamically loadable and allow platform integrators to implements their own version of these effects, provided the API definition is respected. A set of binder interfaces

and associated control classes must be created so that applications can instantiate and control affects modules in the media server process (AudioFlinger).The effect control interfaces consist in a new method added to IAudioFlinger to create a new effect module and two new interfaces, IEffect used by applications to control the effect module activity and parameters and IEffectClient used by AudioFlinger to notify applications when effect module control is lost/recovered. The application will select to which virtual output the effect must be attached. The AudioFlinger will manage the instantiation, sharing of the effect module between applications and insertion of effect modules in the audio path. It will make sure that only one application has the control of the effect module by applying priorities and also make sure, in case of disappearance of the controlling application, that the control is passed over to another application or the effect module is removed.

CHAPTER 4

Problem Statement

As noticed during the literature survey. Google just released Android 2.3 (Gingerbread) fastest version of android yet, focusing on Gaming, media and brining new forms of communication into android platform. In gingerbread concurrent garbage collection which minimizes application pauses, much smoother animations and responsiveness. Developers have direct access to Input and sensors Events Audio, Open GL ES and Assets, Support for Gyroscope for motion processing in games.

Gingerbread has support for VP8 and WebM that are new open video standards, advance audio coding (AAC) and adaptive multi-rate (AMR) support is also added in the latest android version 2.3. Gingerbread has support for audio effects that were missing in Android version 2.2 Froyo.

This thesis includes study of existing audio framework in android and suggesting changes/ enhancements to hardware accelerate audio post processing also called audio effects. To improve battery life in android mobile devices by hardware accelerating audio effects.

In this research work, a new idea of audio routing is proposed for making the audio routing more efficient with the respect of power. Experimental result is compared with the current android audio routing framework.

4.1 Objectives

The objectives are to design the new audio framework, which have a new audio routing to make the long battery (efficient with respect of power) life. To achieve this following steps are involved:

- Understand the current audio routing.
- How to use in build android media libraries?
- Design a new framework.
- Implementation of various audio effects in JAVA.
- Compare the both framework with respect of power.

CHAPTER 5

Implementation and Experiment

5.1 Compressed Audio Framework

Power efficient rendering of compressed audio via hardware tunnelling is a key feature missing from current media framework implementation. The implementation of this feature is not trivial because of the following issues:

a) The behaviour depends on actual hardware capabilities There must be some means to retrieve those capabilities and strategies defined for both cases where hardware tunnelling is supported or not.
b) The behaviour depends on current phone state, it may not be possible to use hardware tunnelling if the user interface (UI) is active and precise progress information must be displayed.
c) Software mixer has volume and mute controls, which are apply as per the incoming stream type.
d) The proposed solution must enable the use of hardware tunnelling for content types that are not handled by packet video (PV) opencore.

The proposal is based on the addition of support for compressed audio formats to AudioTracks. An AudioTrack opened with a compressed format will be handled by a dedicated mixer thread in AudioFlinger connected directly to a compressed audio output at the AudioHardwareInterface. This mixer thread will coexist with other mixer threads handling the mixing of PCM AudioTracks to PCM output streams at the AudioHardwareInterface. The change implies that compressed format is also supported at the AudioHardwareInterface level. The AudioHardware and audio driver implementations must provide means to forward compressed audio to audio DSP at the same time as providing usual PCM output for streams handled by the software mixers.

If the platform implements an OpenMax integration layer (IL) framework to manage hardware accelerated resources or hardware mixers, the audio output can be wrapped in a source component. MediaPlayerService and opencore are required to

automatically select playback over hardware tunnelling. It is nicely summarized throuh the following figure 9.

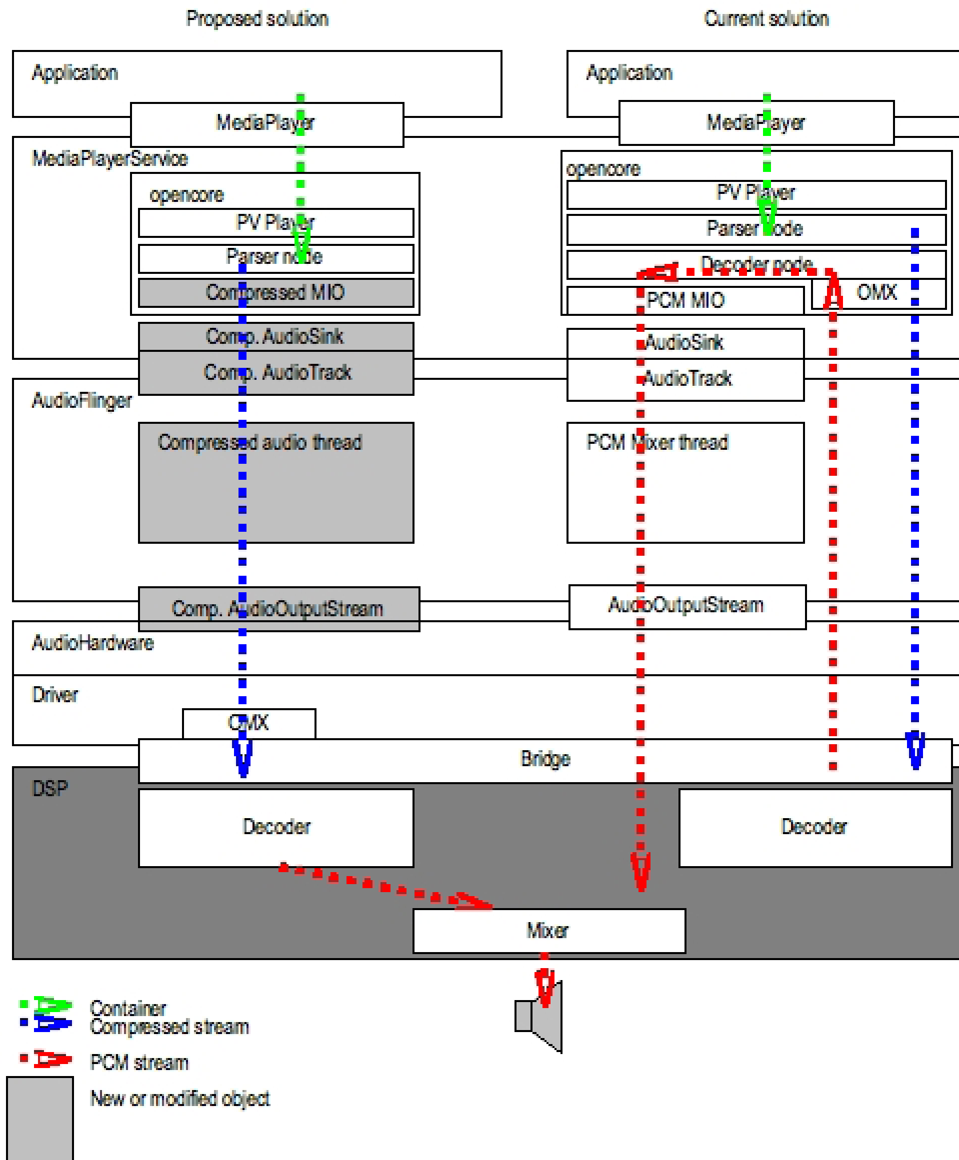


Figure 9. Block Diagram of Current and Propose Approach

5.2 Proposed Work Flow

When opening a compressed file or stream, the media player will check if the current phone state enables the use of hardware tunnelling.

1. If hardware tunnelling is enabled:

Check if the hardware provides a decoder for this compressed format

a) If hardware decompression is supported:

▪ No decoder node is created

▪ A mixer thread is created that will handle both data and volume control for this stream.

▪ the compressed AudioTrack and AudioSink are created and the playback takes place over the compressed sink.

b) If hardware decompression is not supported:

▪ A decoder node is created: audio is decompressed by software or hardware accelerated decoder node.

▪ A PCM AudioSink (and AudioTrack) is opened and playback takes place via the software mixer.

2. If hardware tunnelling is not enabled

A PCM AudioSink (and AudioTrack) is opened and playback takes place via the software mixer. Audio is decompressed by software or hardware accelerated decoder node.

5.3 Audio Post/Pre-processing

The post/pre processes include audio processing applied to the audio output to improve the rendering on a given device (Compression, Filtering) or on the input to improve signal quality (Noise Suppression, AGC). Their availability, implementation and control interfaces are highly platform dependent.

Some applications may leave the default pre/post processing proposed by the audio hardware on each output or input device but others might want to be able to disable or reconfigure those effects. For instance, the Voice Search application may need to implement is own noise suppression on the audio input.

Giving access to the post/pre-process control to applications raises a number of issues:

a) What happens if an application crashes or does not care to reset a specific post process to its default setting?

b) How to handle concurrent conflicting request from several applications sharing the same post effect?

c) How to expose the post processes available on a given platform with all their parameters in a manner that is efficient and usable by applications?

The AudioFlinger will provide a method for an application to create a control interface on post process modules on a particular output/device and then use this interface to enable/disable the post process or change parameters.

The AudioFlinger will keep track of client processes, handle conflicts with a priority mechanism. The AudioHardwareInterface will provide methods to query the post process modules and their parameters. The parameters should be of a sufficiently abstracted level (e.g. a gain in dB, a frequency range in Hz.) rather than a low level (e.g. coefficient values) so that an application can use the interface without knowledge of the actual post process implementation.

The audio hardware will still have the latitude to activate some post process by default when the output is routed to certain devices. It is the audio hardware implementation responsibility to accept or reject the addition of a given post process and manage conflicts with default ones.

5.3.1 Effects Plug-ins

The AudioFlinger mixer will allow the insertion of a process module before the SRC/mixer process. A possible implementation is based on a Dynamically Loaded library that contains a set of process functions. These functions have a standardized API (init, process, get/set) compatible with current API of AudioMixer. When a request is made to attach such a process to an AudioTrack, the library is loaded and parsed for the appropriate process.

A similar customization mechanism can also be used to replace the default resample implementation. The effects plug-in library and the AudioFlinger will provide methods to query the list of available plug-in effects. As per the current framework audioflinger pass the required parameters to the Effect factory and audio effects specific libraries (.SO) are loaded from the ARM processor. Audio Network Manager is audio policy manager and audio stream input/output, which pass parameters for audio. The current process is shown in figure 10.

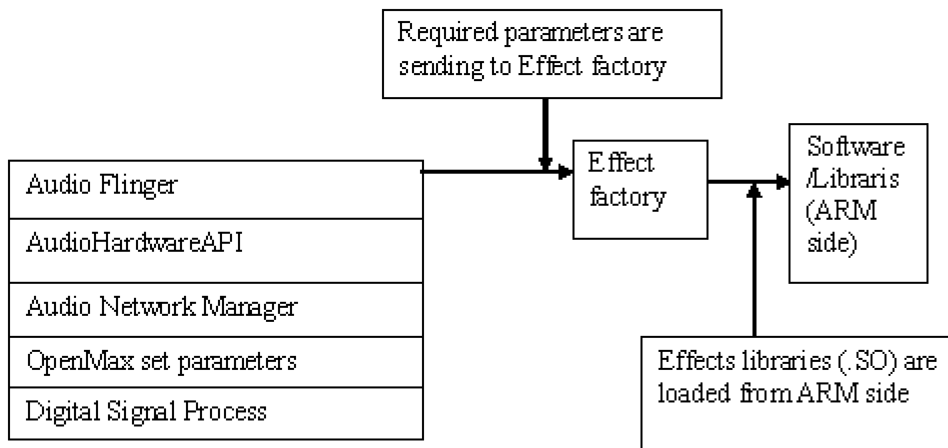


Figure 10. Block diagram of current audio processing stack

ARM processor is the core processor for Mobile devices, which takes more power as compare to the secondary processor (Digital signal processor). As per the proposed media framework AudioFlinger will pass the required parameter to the AudioHardwareInterface. And this AudioHardwareInterface take care the required parameter. From Audio Network Manager, audio policy manager will be enabling and initialize the stream input/output methods. Further audio device manager implements the initialize methods and audio effects libraries are loaded from the Digital signal processor which are present in (.elf) format. The proposed audio processing is shown in the figure 11.

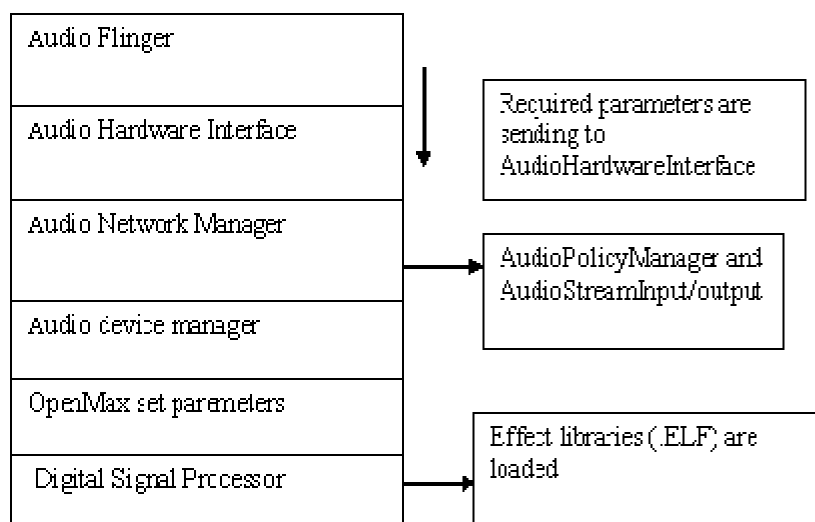


Figure 11. Block Diagram of Propose Audio Processing Stack

5.3.2 Algorithm of Bass Boost Audio Effects

The algorithm is mention below, which shows integration of Bass boost audio effects.

START:

Plays the sound file designated by the given path, and applies a bass boost effect of the specified strength, where strength is a integer value between 0 and 1000.

Step1: Realizing the SL Engine in synchronous mode.

If this fails call ExitOnError (Boolean);

Step 2: Define function TestBassBoostPathFromFD (sl_object, path, bass_boost strength) objects this application uses one player and an output mix, set the source of audio data to play. Apply data sinks for the audio player set the Interfaces for the audio player.

Step 3: Get the SL Engine Interface which is implicit, and create output mix object which is used by the player. And set the Output Mix object in synchronous mode,

Step 4: Configuration of the player, Set arrays required [] and iidArray[] for required interfaces with default TRUE

(SLPlayItf is implicit)

Step 5: Setup the data source structure for the URI

```
    locator_Field_descriptor.locator_Type =
SL_DATALOCATOR_ANDROID_Field_Descriptor;
    Int field_descriptor = open (path, O_RDONLY);
  If (field_descriptor == -1) {
     ExitOnError (SL_RESULT_RESOURCE_ERROR);
  }
```

Step 6: Create the audio player, to play the file and if any error occurs call ExitOnError (error result);

Realize the player in synchronous mode. Get the SLPlayItf, SLPrefetchStatusItf and SLBassBoostItf interfaces for the player.

Step 7: Start the data prefetching by setting the player to the paused state, wait until there's data to play ;

Step 8: Get duration

```
SL_millisecond durationInMsec = SL_TIME_UNKNOWN;
result = (*playItf) ->GetDuration (playItf, &durationInMsec);
if (durationInMsec == SL_TIME_UNKNOWN) {
```

durationInMsec = 5000;

Start playback ;

Step 9 : Configure BassBoost :

```
SLboolean strengthSupported = SL_BOOLEAN_FALSE;
result = (*bbItf)->IsStrengthSupported(bbItf, &strengthSupported);
if (SL_BOOLEAN_FALSE == strengthSupported) {
fprintf(stdout, "BassBoost strength is not supported on this platform. Too
bad!\n");
} else {
fprintf(stdout, "BassBoost strength is supported, setting strength to %d\n",
boostStrength);
result = (*bbItf)->SetStrength(bbItf, boostStrength);
}
SLpermille strength = 0;
result = (*bbItf)->GetRoundedStrength(bbItf, &strength);
fprintf(stdout, "Rounded strength of boost = %d\n", strength);
```

Step 10: Make sure player is stopped, Destroy the player object, Destroy Output Mix object.

END.

5.3.3 Algorithm of Virtualizer Audio Effects

Algorithm is for audio virtualizer an effect is given below.

Start:

#define MAX_NUMBER_INTERFACES 3

#define TIME_S_BETWEEN_VIRT_ON_OFF 3

Exits the application if an error is encountered

#define ExitOnError(x) ExitOnErrorFunc(x,__LINE__);

Variable declaration: SLresult result, SLObjectItf sl;

AudioPlayer with SLDataLocator_AndroidFD source/OutputMix sink Plays the sound file designated by the given path, and applies a virtualization effect of the specified strength, where strength is an integer value between 0 and 1000.

Step 1: Realizing the SL Engine in synchronous mode,

if it get fails call ExitOnError () ,

Intentionally not checking that argv [2], the virtualizer strength, is between 0 and 1000.

Step 2: call TestVirtualizerPathFrom (SLObjectItf sl, const char* path, int16_t virtStrength)

SLresult result;

SLEngineItf EngineItf;

Objects this application uses: one player and an ouput mix, and give source of audio data to play.

Step 3: Data sinks for the audio player

SLDataSink audioSink;

SLDataLocator_OutputMix locator_outputmix;

Play and PrefetchStatus interfaces for the audio player.

Step 4: Get the SL Engine Interface which is implicit

Initialize arrays required [] and iidArray []

for (int i=0; i < MAX_NUMBER_INTERFACES ; i++) {

required[i] = SL_BOOLEAN_FALSE;

iidArray[i] = SL_IID_NULL;

Step 5: Configuration of the output mix, Create Output Mix object to be used by the Player Realize the Output Mix object in synchronous mode. Setup the data Sink structure.

Step 6: Set arrays required [] and iidArray [] for SLPrefetchStatusItf interfaces

SLPlayItf is implicit.

required [0] = SL_BOOLEAN_TRUE;

iidArray [0] = SL_IID_PREFETCHSTATUS;

required [1] = SL_BOOLEAN_TRUE;

iidArray [1] = SL_IID_VIRTUALIZER;

Step 7: Setup the data source structure for the URI

locator_Field_description.locator_Type =

SL_DATALOCATOR_ANDROID_Field_Description;

int field_descriptin = open(path, O_RDONLY);

if (field_description == -1)

ExitOnError(SL_RESULT_RESOURCE_ERROR);

else

locator_Uri.locatorType = SL_DATALOCATOR_URI;

locator_Uri.URI = (SLchar *) path;

Step 8 : Create the audio player, if it fails call ExitOnError ();

Step 9: Realize the player in synchronous mode and Get the SLPlayItf, SLPrefetchStatusItf and SLAndroidStreamTypeItf interfaces for the player.

Step 10: Playback and test, Start the data prefetching by setting the player To the paused state.Wait until there's data to play.

Get duration

```
SLmillisecond durationInMsec = SL_TIME_UNKNOWN;
result = (*playItf)->GetDuration(playItf, &durationInMsec);
if (durationInMsec == SL_TIME_UNKNOWN) {
durationInMsec = 5000;
```

Step 11: Start playback

```
fprintf(stdout, "Starting to play\n");
result = (*playItf)->SetPlayState(playItf, SL_PLAYSTATE_PLAYING );
```

Step 12: Configure Virtualizer

```
SLboolean strengthSupported = SL_BOOLEAN_FALSE;
result = (*virtItf)->IsStrengthSupported(virtItf, &strengthSupported);
ExitOnError(result);
if (SL_BOOLEAN_FALSE == strengthSupported) {
fprintf(stdout, "Virtualizer strength is not supported on this platform. Too
bad!\n");
}
else
{
fprintf(stdout, "Virtualizer strength is supported, setting strength to %d\n",
virtStrength);
result = (*virtItf)->SetStrength(virtItf, virtStrength);
    }
```

Step 13: Switch Virtualizer on/off every TIME_S_BETWEEN_VIRT_ON_OFF seconds.

```
SLboolean enabled = SL_BOOLEAN_TRUE;
result = (*virtItf)->SetEnabled(virtItf, enabled);
for (unsigned int j=0;
    j< (durationInMsec/ (1000*TIME_S_BETWEEN_VIRT_ON_OFF));
     j++) {
```

```
usleep(TIME_S_BETWEEN_VIRT_ON_OFF * 1000 * 1000);
result = (*virtItf)->IsEnabled(virtItf, &enabled);
enabled = (enabled == SL_BOOLEAN_TRUE? SL_BOOLEAN_FALSE;
SL_BOOLEAN_TRUE);
result = (*virtItf)->SetEnabled(virtItf, enabled);
if (SL_BOOLEAN_TRUE == enabled) {
fprintf(stdout, "Virtualizer on\n");
            }
            else {
            fprintf(stdout, "Virtualizer off\n");
              }
            ExitOnError(result);
            }
```

Step 14: Make sure player is stopped, if it fails call ExitOnError(result);

Step 15: Destroy the player object.

```
(*player)->Destroy (player);
Destroy Output Mix object.
(*outputMix)->Destroy (outputMix);
```

END;

5.4 Comparison of current & proposed approach

Every approach required the specification of the battery for the power measurement. Measure of charge (Power) is the Coulomb; a single electron has 1.602 e – 19 Coulombs of charge. One Ampere flowing in a wire for one second will use one Coulomb of Charge.

$$Q= I*T$$

Where, Q= charge in Coulombs.

I= current in Ampere.

T = time in second.

Small batteries (mobile's batteries) are come up with mill ampere- hour's unit. It is explained with the help of 15oo mAh (Milliampere hours), if two or three application is running on device. The power they take is 100 mA (milliampere)

Then total battery life is

1500 mAh/100mA = 15 hours;

If the consumed power is reduce by proposed framework and let it now takes 80 mA.

Then we have.

1500 mAh/80mA = 18.75 hours battery life.

Exact calculation of battery life is depends up on the specification.

In the current android multimedia framework, audio effects are integrated with the help of Arm's libraries (.SO). The results of the changes in Power after enabling the bass booster audio effect in vanilla android 2.3 and proposed android audio framework are shown in the figures 12 – 14.

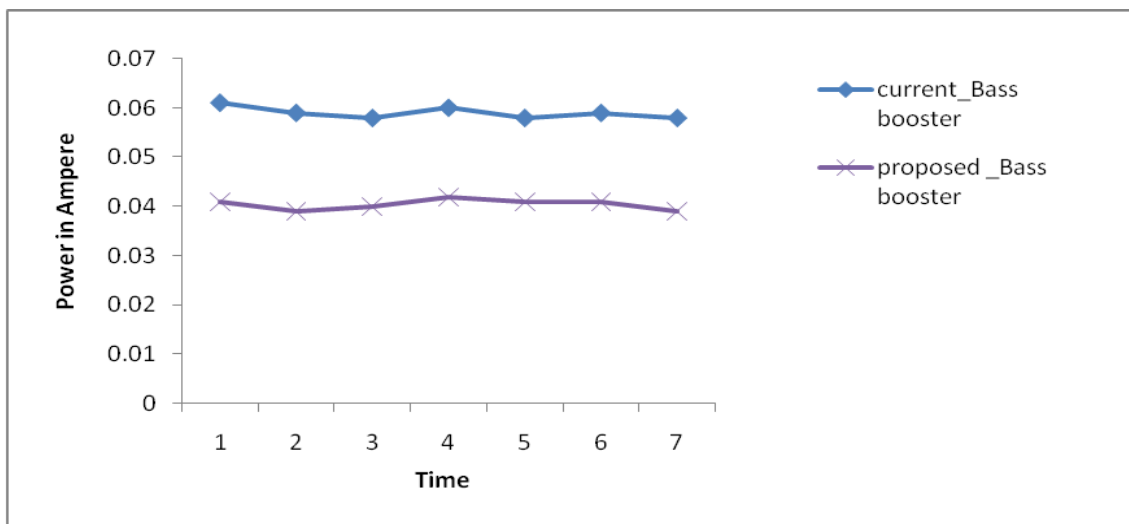


Figure 12. Power Consumption in Unit Time for Bass Booster Effect.

In the graph (figure 12), Power is continuously changing with respect of a time unit (experimental time). Next graph shows the power consumption after enabling the reverb audio effect in vanilla android 2.3 and proposed android audio framework.

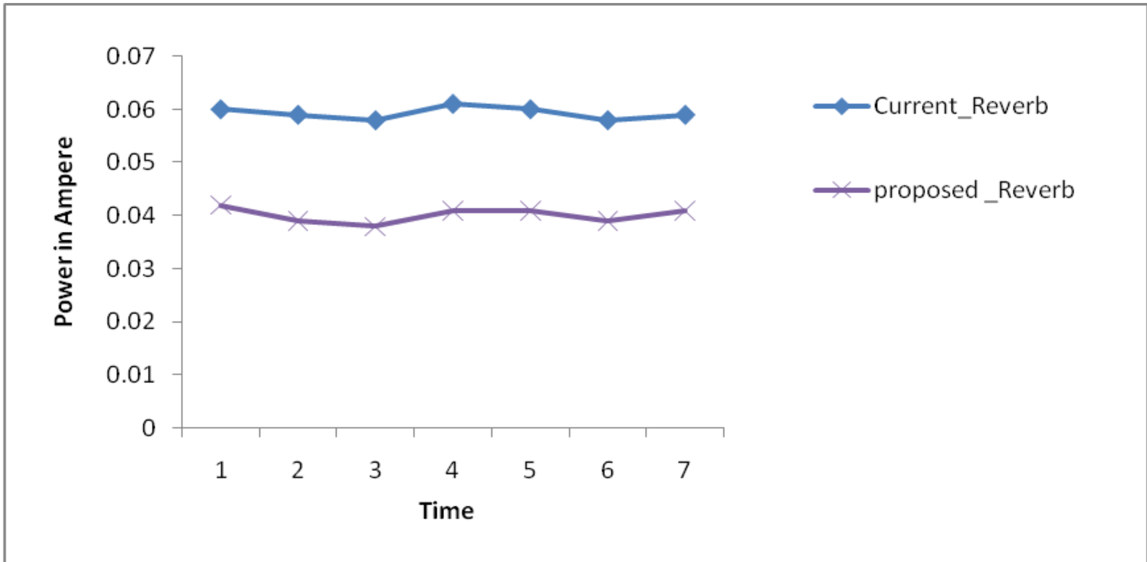


Figure 13. Power Consumption in Unit Time for Reverb Effect.

Next graph shows the power consumption after enabling virtualizer audio effect in vanilla android 2.3 and proposed android audio framework.

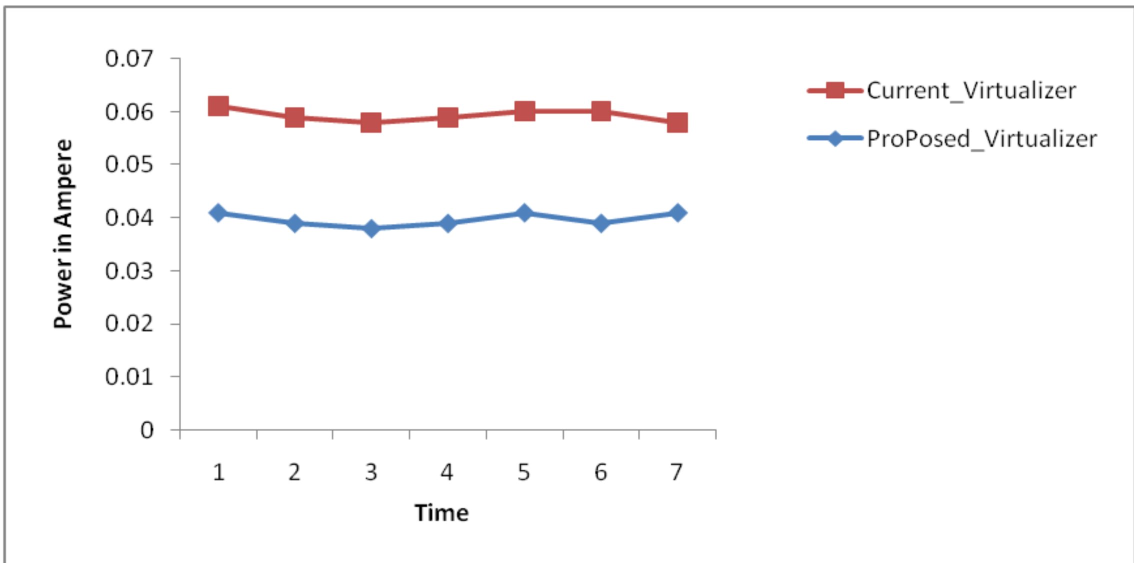


Figure 14. Power Consumption in Unit Time for Virtualizer Effect.

.

CHAPTER 6

Conclusion and Future Scope

It is found to be possible to accelerate audio effects and it also provides power gain hence improved battery life, although the whole solution is need continuous integration with ongoing releases of android.

As per the proposed framework we have saved approx 20% power. This is the main objective to design the framework. By using this framework user can accesses the pointer of low level coding (Device driver) or hardware level coding.

In proposed framework, audio processing is done on DSP so most of the time ARM Processor will remain ideal, when playback is on.

In future we can extend this framework for passing the compressed audio to DSP And save more battery. This framework can be also extended for achieving the direct rendering of audio effects in the audio hardware.

References

[1]. Xiangling Fu, Xiangxiang Wu, Maoqiang Song, Mian Chen ,"*Research on Audio/Video Codec Based on Android*" Wireless Communications Networking and Mobile Computing (WiCOM), 2010 6th International Conference at Chengdu , 23-25 Sept. 2010, pp 1 – 4.

[2]. Bian Wu, Wang, A.I., Hartvoll Ruud, A.Wan Zhen Zhang "*Extending Google Android's Application as an Educational Tool*", Digital Game and Intelligent Toy Enhanced Learning (DIGITEL), 2010 Third IEEE International Conference at Kaohsiung, 12-16 April 2010, pp. 23 – 30.

[3]. V. Tiwari, S. Malik, and A. Wolfe, "*Power analysis of embedded software: A first Step towards software power minimization*," Proc. Int. Conf. Computer-Aided Design, Nov. 1994.

[4]. T.kumar kundu, Prof. Kolin Paul "*improving Android Performance and Energy Efficiency"* 1063-9667/11, 2011 IEEE, DOI 10, 1109/vlsi D .2011.63, IEEE Computer Society 2011, pp. 256-261.

[5]. T. Sato,Y. Ootaguro, M. Nagamatsu, and H.Tago, *"Evaluation of architecture-Level power estimation for CMOS RISC processors,*" Proc. Int. Symp. Low Power Electronics, Oct. 1995, pp 44-45.

[6]. Kuzmanovic, N., Maruna, T., Savic, M., Miljkovic, G., Isailovic, D *"Google's Android as an application environment for DTV decoder system*" Consumer Electronics (ISCE), 2010 IEEE 14th International Symposium,7-10 June 2010, pp 1 –5.

[7]. C.T. Hsieh, M. Pedram, G. Mehta, and F.Rastgar, *"Profile-driven program Synthesis for evaluation of system power dissipation,"* Proc. Design Automation Conf. June 1997, pp 576-581.

[8]. Maoqiang Song, Wenkuo Xiong, Xiangling Fu *"Research on Architecture of Multimedia and Its Design Based on Android"* Internet Technology and Applications, 2010 International Conference at Wuhan 20-22 Aug. 2010, pp 1 – 4.

[9]. Y. Li and J. Henkel, "*A framework for estimating and minimizing energy Dissipation of embedded HW/SW systems"* in Proc. Design Automation Conf,June 1998, pp. 188–193.

[10]. Open Handset Alliance, Open Handset Alliance Announces 14 New Members, http://www.openhandsetalliance.com/press_120908.html.

[11]. L. Benini, A. Bogliolo, G.A. Paleologo and G. De Micheli, *"Policy Optimizatio For dynamic power management",* IEEE Trans. on Computer-Aided Design, 18(6),1999, pp 813-833.

[12]. C. L. Su, C-Y. Tsui and A. M. Despain, "*Low power architecture design and Compilation techniques for high performance processors*", Proc. IEEE Compcon, 1994, pp 489-498.

[13]. V. Tiwari, S. Malik and A. Wolfe, *"Instruction level power analysis and Optimization of software",* Journal of VLSI Signal Processing, 1996, pp 1-18.

[14]. Android - An Open Handset Alliance Project: http://code.google.com/intl/zh-CN/android/

[15]. Open Handset Alliance, http://www.openhandsetalliance.com/

[16]. http://code.google.com/intl/zh-CN/android/reference/available-resources.html

[17]. http://java.sun.com/j2se/1.5.0/docs/guide/jni/

[18]. www.cs.ucy.ac.cy/~dzeina/courses/epl371/

[19]. Android Developers, http://www.androidin.com/

[20]. http://java.sun.com/j2se/1.5.0/docs/guide/jni/

[21]. Kyungjin Byun, Young-Su Kwon, Seongmo Park, and Nak-Woong Eum "*Digital audio Effect System-on-a-Chip Based on Embedded DSP Core*", ETRI Journal, 31, Number 6, December 2009, pp 732-740.

[22]. http://media.wiley.com/product_data/excerpt/98/04706659/0470665998241.pdf

[23]. T.Austin, E.Larson, and D.Ernst. "*Simple scalar: An infrastructure for computer System modelling",* Computer, 35(2) Feb. 2002, pp 59-67,

[24]. Matt Calder and Mahesh K. Marina *"Batch Scheduling of Recurrent Applications for Energy Savings on Mobile Phones"*, Sensor Mesh and Ad Hoc Communications and Networks (SECON), 2010 7th Annual IEEE Communications Society Conference on, 21-25 June 2010. 978-1-4244-7151-5/10/$26.00 ©2010.

[25]. Y. Xiao, R. Sri Kalyanaraman and A. Yla-Jaaski "*Energy Consumption of Mobile YouTube: Quantitative Measurement and Analysis"*, The Second International Conference on Next Generation Mobile Applications, Services, and Technologies 2008, pp.61-69.

[26]. G.Chen, M. Kandemir, N.Vijaykrishnan and M.irwin *"pennbench:a benchmark suite for embedded java."* , in workload Characterization, 2002 WWC-5. 2002, pp 71-80.

[27]. Kundu T.K, Paul K. *"Improving Android Performance and Energy Efficiency"* VLSI Design (VLSI Design), 24th international conference 22 feb.2011, pp 256 – 261.

[28]. L. Butts and A. Cockburn, *"An Evaluation of Mobile Phone Text Input Methods,"* , Proc. Conf. Research and Practice in Information Technology Series, 3rd Australasian Conf. User Interfaces, 7 Australian Computer Soc., 2002, pp 55 59.

[29]. H. Wu, K. Tan, J. Liu, and Y. Zhang. *"Footprint: Cellular Assisted Wi-Fi AP Discovery on Mobile Phones for Energy Savings." ,* Proc. ACM WiNTECH, 2009.

[30]. Tapas Kumar, Kundu and Kolin Paul *"Android on mobile devices an energy perspective"*, CIT 2010 proceedings of the 2010 10th IEEE international conference on computer and information technology DIO 10.1109/CIT 2010.416, pp 2421-2426.

[31]. Yu-Sheng Lu, Chin-Ho Lee, Hung-Yen Weng, Yueh-Min Huang "*Design and Implementation of digital TV widget for Android on multi-core platform*" , Computer Symposium (ICS), 2010 International, Tainan 16-18 Dec. 2010,pp 576 – 580.

[32]. Ballagas, R., Borchers, J., Rohs, M., Sheridan, J.G.*,"The Smart Phone: A Ubiquitous Input Devic*e" , Pervasive Computing, IEEE, Jan.-March 2006 ,5(1),pp 70-77.

[33]. http://www.3gpp.org/ftp/Specs/html-info/26071.htm

[34]. Audio Engineering Society, http://www.aes.org/e-lib/browse.cfm?elib=14433

[35]. http://www.iqmagazineonline.com/current/pdf/Pg1115Ittiam Integrating.

[36]. Gavalas, D., Economou D. *"Development Platforms for Mobile Applications: Status and Trends "*, Software, IEEE, Jan.-Feb. 2011, (1), pp 78-86.

[37]. A**.** Koller, G. Foster, and M. Wright, *"Java Micro Edition and Adobe Flash Lite for Ar- cade-Style Mobile Phone Game Development: A Comparative Study*", Proc. ACM Annual Conference South African Inst. Computer Scientists and Information Technologists (SAICSIT 08), ACM Press, 2008, pp 131–138**.**

[38]. N. Gramlich, Android Programming, PDF Electronic Book, 2008. Available from: http://androidos.cc/dev/index.php.

[39]. J.F. DiMarzio, Android A Programmer's Guide, Chicago: McGraw-Hill, Jul.2008.

[40]. Developer resources for Google Android: http://developer.android.com/

[41]. N Vun, Y. H. Ooi, *"Implementation of an Android Phone Based Video Streamer"*, Green Computing and Communications (GreenCom), 2010 IEEE/ACM Conference & Conference on Cyber, Physical and Social Computing (CPSCom), Physical and Social Computing, 18-20 Dec. 2010, pp 912 – 915.

[42]. Pushkar Apte,W.R. Bottoms, William Chen & George Scalise "*Good things in small packages*", IEEE Spectrum , March 2011,pp 44-49.

[43]. Tommi takala and Mika katara, Julian harty *"experiences of system-level model-based gui testing of an android application*", Fourth IEEE International Conference on software testing, verification and validation 2010, 978-0-7695-4342- 0/11 $26.00 © 2011 IEEE, DOI 10.1109/icst.2011.11, pp 377- 386.

[44]. Fan Peck Ling, Fung Kah Khuen, Radhakrishnan D."*An audio processor card for special sound effects"*, Circuits and Systems, 2000. Proceedings of the 43rd IEEE Midwest Symposiu, 2, 2000, pp 730.

Appendix A

Java Code for the Various Audio Effects

```
import java.io.IOException;
import android.media.audiofx.*;
import android.media.audiofx.AudioEffect.Descriptor;
import android.os.SystemClock;
import android.util.Log;

public final class TC_AD_AU_000_AudioEffect extends PlayAudioTestCase {
        private final String TAG = "AudioFxTestApp";
        private final MediaPlayer mPlayer = new MediaPlayer();
public void runTest(String... dataSource) throws IllegalArgumentException,
IllegalStateException, IOException {
                mPlayer.setDataSource(dataSource[0]);
                mPlayer.prepare();
                mPlayer.start();
                int sessionId = mPlayer.getAudioSessionId();
                Log.d(TAG, "Playing without any effect applied.");
                SystemClock.sleep(3000);
                // Reverb effect.
        PresetReverb reverb = new PresetReverb(0, sessionId);
                Log.d(TAG, "Attached reverb effect to session " + sessionId);

                reverb.setPreset(PresetReverb.PRESET_MEDIUMHALL);
        Log.d(TAG, "Setting 'PRESET_MEDIUMHALL' applied to reverb   effect.");
                reverb.setEnabled(true);
                Log.d(TAG, "Enabled reverb effect.");
                SystemClock.sleep(10000);
            reverb.setEnabled(false);
                Log.d(TAG, "Disabled reverb effect.");
                SystemClock.sleep(3000);
```

```
reverb.release();
Log.d(TAG, "Released reverb effect.");
SystemClock.sleep(500);
// Bass boost effect.
BassBoost bass = new BassBoost(0, sessionId);
Log.d(TAG, "Attached bass boost effect to session " + sessionId);

bass.setStrength((short)1000);
Log.d(TAG, "Setting 'STRENGTH 1000' applied to bass boost effect.");
bass.setEnabled(true);
Log.d(TAG, "Enabled bass boost effect.");
SystemClock.sleep(10000);

bass.setEnabled(false);
Log.d(TAG, "Disabled bass boost effect.");
SystemClock.sleep(4000);

bass.release();
Log.d(TAG, "Released bass boost effect.");
SystemClock.sleep(500);



// Virtualizer effect.
Virtualizer virtualizer = new Virtualizer(0, sessionId);
Log.d(TAG, "Attached virtualizer effect to session " + sessionId);

virtualizer.setStrength((short)1000);
Log.d(TAG, "Setting 'STRENGTH 1000' applied to virtualizer effect.");
virtualizer.setEnabled(true);
Log.d(TAG, "Enabled virtualizer effect.");
SystemClock.sleep(10000);

virtualizer.setEnabled(false);
Log.d(TAG, "Disabled virtualizer effect.");
```

```
                SystemClock.sleep(5000);

                virtualizer.release();
                Log.d(TAG, "Released virtualizer effect.");
                SystemClock.sleep(500);



                mPlayer.stop();
                Log.d(TAG, "Stopped playback.");

                mPlayer.reset();
                mPlayer.release();
        }
Public void test_Reverb() throws IllegalArgumentException,
IllegalStateException, IOException {
String dataSource = mTargetDestination +
"PCM_WAV_Music_RegalBaroque_169sec_Stereo_48kHz_SInt16.wav";

                mPlayer.setDataSource(dataSource[0]);
                mPlayer.prepare();
                mPlayer.start();
                int sessionId = mPlayer.getAudioSessionId();
                Log.d(TAG, "Playing without any effect applied.");
                SystemClock.sleep(3000);
        // Reverb effect.

                PresetReverb reverb = new PresetReverb(0, sessionId);
                Log.d(TAG, "Attached reverb effect to session " + sessionId);

                reverb.setPreset(PresetReverb.PRESET_MEDIUMHALL);
Log.d(TAG, "Setting 'PRESET_MEDIUMHALL' applied to reverb effect.");

                reverb.setEnabled(true);
                Log.d(TAG, "Enabled reverb effect.");
```

```
            SystemClock.sleep(10000);

            reverb.setEnabled(false);
            Log.d(TAG, "Disabled reverb effect.");
            SystemClock.sleep(3000);

            reverb.release();
            Log.d(TAG, "Released reverb effect.");
            SystemClock.sleep(500);



            mPlayer.stop();
            Log.d(TAG, "Stopped playback.");

            mPlayer.reset();
            mPlayer.release();
        }
Public void test_BassBoost()throws IllegalArgumentException,
IllegalStateException, IOException {
String dataSource = mTargetDestination +
"PCM_WAV_Music_RegalBaroque_169sec_Stereo_48kHz_SInt16.wav";
mPlayer.setDataSource(dataSource[0]);
            mPlayer.prepare();
            mPlayer.start();
            int sessionId = mPlayer.getAudioSessionId();
            Log.d(TAG, "Playing without any effect applied.");
            SystemClock.sleep(3000);
            // Bass boost effect.
            BassBoost bass = new BassBoost(0, sessionId);
            Log.d(TAG, "Attached bass boost effect to session " + sessionId);
            bass.setStrength((short)1000);
            Log.d(TAG, "Setting 'STRENGTH 1000' applied to bass boost
           effect.");
```

```
bass.setEnabled(true);
Log.d(TAG, "Enabled bass boost effect.");
SystemClock.sleep(10000);

bass.setEnabled(false);
Log.d(TAG, "Disabled bass boost effect.");
SystemClock.sleep(4000);
 bass.release();
Log.d(TAG, "Released bass boost effect.");
SystemClock.sleep(500);
mPlayer.stop();
Log.d(TAG, "Stopped playback.");
mPlayer.reset();
mPlayer.release();
}
Public void test_Virtualizer() throws IllegalArgumentException,
 IllegalStateException, IOException {
String dataSource    =  mTargetDestination +
"PCM_WAV_Music_RegalBaroque_169sec_Stereo_48kHz_SInt16.wav";

mPlayer.setDataSource(dataSource[0]);
mPlayer.prepare();
mPlayer.start();
int sessionId = mPlayer.getAudioSessionId();
Log.d(TAG, "Playing without any effect applied.");
SystemClock.sleep(3000);
// Virtualizer effect.
Virtualizer virtualizer = new Virtualizer(0, sessionId);
Log.d(TAG, "Attached virtualizer effect to session " + sessionId);
virtualizer.setStrength((short)1000);

Log.d(TAG, "Setting 'STRENGTH 1000' applied to virtualizer effect.");

virtualizer.setEnabled(true);
```

```
                Log.d(TAG, "Enabled virtualizer effect.");
                SystemClock.sleep(10000);
              virtualizer.setEnabled(false);
                Log.d(TAG, "Disabled virtualizer effect.");
                SystemClock.sleep(5000);
               virtualizer.release();
                Log.d(TAG, "Released virtualizer effect.");
                SystemClock.sleep(500);
               mPlayer.stop();
                Log.d(TAG, "Stopped playback.");

                mPlayer.reset();
                mPlayer.release();
        }
public void test_ListEffects() throws
IllegalArgumentException, IllegalStateException, IOException {
                Descriptor [] effects = AudioEffect.queryEffects();
                for (Descriptor d: effects)
                   {
Log.d(TAG, "Effect: '" + d.name + "', implementor: '" + d.implementor + "', connect
mode: '" + d.connectMode + "', type: '" + d.type + "', uuid: '" + d.uuid + "'.");
                }
        }
}
```

List of Publications

1. Kailash Pathak , V.P.Singh *"Audio routing algorithm for multimedia framework in android 2.3"*, Communicated to Research Journal of computer system engineering, an International journal (ISSN : 2230-8563, e-ISSN : 2230-8571).