doi:10.3969/j.issn.1672-4348.2019.04.011

Android 原生 Audio Effect 框架分析与改进

庄伟达[1],何兴理[1],林金阳[1,2]

(1.福建工程学院 信息科学与工程学院,福建 福州 350118;
2.福建工程学院 微电子技术研究中心,福建 福州 350118)

摘要:基于 Android 系统底层源码,深入分析 Android 系统原生 Audio Effect 框架,针对该框架仅可支持 16 bit 音频数据进行处理的问题,采用先将系统输入的 32 bit 音频数据移位,然后再进行强制位数转换的方法,在增加原生 Audio Effect 框架对于 32 bit 音频数据处理支持的同时将由于量化位深转换带来的失真控制在 0.000 76%左右。实验结果表明,系统可根据用户要求对 32 bit 的音频数据进行相应的处理,丰富了原生框架对于不同品质音频数据处理的支持。

关键词:Android 平台;多媒体功能;Audio Effect 框架;移位;位数转换
中图分类号:TP37 **文献标志码**:A **文章编号**:1672-4348(2019)04-0371-06

Analysis and improvement of Android native Audio Effect framework

ZHUANG Weida[1], HE Xingli[1], LIN Jinyang[1,2]

(1.School of Information Science and Engineering, Fujian University of Technology, Fuzhou 350118, China;
2.Research Center for Microelectronics Technology, Fujian University of Technology, Fuzhou 350118, China)

Abstract: Based on the underlying source code of Android system, an in-depth analysis was conducted into its native Audio Effect framework. In order to solve the problem that the framework can only support 16-bit audio data processing, the 32-bit audio data input by the system was first shifted, and then the forced digit conversion was performed. The method is to control the distortion caused by the convertion of the quantization bit-depth to around 0.000 76% while increasing the support of the native Audio Effect framework for 32-bit audio data processing. Experimental results show that the system can process 32-bit audio data according to user requirements, which enriches the support of native framework for processing audio data of different qualities.

Keywords: Android platform; multimedia function; Audio Effect framework; shift; digit conversion

音频系统是 Android 多媒体功能中重要的组成部分。已公开的关于音效处理的研究文献大多是单独研究音效算法[1-2]或是结合 DSP 芯片开发[3-5],Android 系统音频相关的文献目前多数集中在编解码方面,音效处理方面的较为欠缺[6-7]。Android 系统源代码体系庞大,现有分析 Android 系统源代码以及底层开发类的书籍资料多是从整个系统的宏观角度出发,具体功能实现分析方面的书籍资料较少。本研究分析 Android 音频系统的源码发现,Android 对于音效处理的支持还存在仅支持 16 bit 音频数据处理的问题。其数字音频系统分辨率取决于音频数据采样的量化位深,对于有限的量化技术而言,将连续的模拟信号完整表示出来几乎不可能,故量化值与实际值之间便会产生误差,称为量化误差[8]。理想情况下,16 bit 音源量化下,1 个量化间隔存在的误差大约存

收稿日期:2019-06-07
基金项目:福建省自然科学基金资助项目(2017J05097)
第一作者简介:庄伟达(1995-),男,福建泉州人,硕士研究生,研究方向:Android 底层开发,音频处理等。
通信作者:林金阳(1981-),男,福建莆田人,副教授,博士,研究方向:光电显示系统。

在 0.001 53% 的失真,24 bit 下大约为 0.000 03%,而 32 bit 存在的量化误差则更小[9]。许多音乐发烧友更倾心于 32 bit 音源,仅支持 16 bit 音频数据的处理已经无法满足大众对音质的追求,所以需对 Android 音频系统进一步完善,使其对 32 bit 的音源提供支持[10]。

音频数据采用 PCM 编码,一般采用自然二进制码或格雷码[11]。一般的二进制码的编码规则在将 32 bit 数据进行强制位数转换为 16 bit 数据时,保留下来的为低位的数据,但高位的数据较之更为重要。本文采取先对输入数据进行移位操作,将高位数据移至低位后,再进行强制位数转换,保留高位数据,丢弃低位数据。另外,对于音频数据来说,输入为 32 bit 的音频数据,输出时也应为 32 bit 的数据,否则输出的音频数据会存在一定的误差[8]。当数据处理完成后还原为 32 bit 进行输出可将量化位深转换带来的误差控制在约为 0.000 76%左右[9]。

1 Android 原生 Audio Effect 框架分析

在 Android 源代码中,打开日志输出功能的方法为#define LOG_TAG "XX"以及#define LOG_NDEBUG 0 这两条语句,打开后在适当的地方调用 ALOGV()函数即可将指定的内容在运行日志中输出。由于 Android 的音频系统较为复杂庞大,故改进 Android 原生 Audio Effect 框架必需对原生 Audio Effect 框架进行深入分析。通过日志输出文件可得知,Android 原生 Audio Effect 框架主要分为三个部分,即音效创建、命令执行及音效处理。

1.1 音效创建

Android 系统中,使用原生 Audio Effect 框架必须先进行初始化,定义必要的参数,音效创建便是做此用途。在框架中,音效创建为 EffectCreate 部分。其主要调用框架中的两个初始化操作,分别为 LvmGlobalBundle_init 以及 LvmBundle_init 两部分。具体是先调用 LvmGlobalBundle_init 进行全局初始化,这部分主要是初始化全局内存,再通过 LvmBundle_init 进行框架初始化,这部分主要是使用默认配置初始化引擎,创建禁用所有效果的初始化实例。从此部分代码的默认配置中能够看出,原生框架仅支持 16 bit 音频数据的处理,如表 1 中第 5 以及第 13 行加粗代码所示,inputCfg.format 以及 outputCfg.format 都为 AUDIO_FORMAT_PCM_16_BIT。当然,EffectCreate 部分还有其他如匹配音效模式(重低音、环绕声、均衡器等)来开启对应的音效功能。

表 1 LvmBundle_init 部分代码

Tab.1 Part of the LvmBundle_init code

行号	源码
1	int LvmBundle_init(EffectContext * pContext){
2	int status;
3	pContext->config.inputCfg.accessMode = EFFECT_BUFFER_ACCESS_READ;
4	pContext->config.inputCfg.channels = AUDIO_CHANNEL_OUT_STEREO;
5	**pContext->config.inputCfg.formaT = AUDIO_FORMAT_PCM_16_BIT;**
6	pContext->config.inputCfg.samplingRate = 44100;
7	pContext->config.inputCfg.bufferProvider.getBuffer = NULL;
8	pContext->config.inputCfg.bufferProvider.releaseBuffer = NULL;
9	pContext->config.inputCfg.bufferProvider.cookie = NULL;
10	pContext->config.inputCfg.mask = EFFECT_CONFIG_ALL;
11	pContext->config.outputCfg.accessMode = EFFECT_BUFFER_ACCESS_ACCUMULATE;
12	pContext->config.outputCfg.channels = AUDIO_CHANNEL_OUT_STEREO;
13	**pContext->config.outputCfg.formaT = AUDIO_FORMAT_PCM_16_BIT,**

1.2 命令执行

在原生 Audio Effect 框架中,命令执行部分为 Effect_command,它包含框架中除了数据处理部分之外其他调用。Effect_command 部分中包括了音效的生效、各类参数的获取和设置、以及音效的释放等等部分,其中各类参数的获取和设置是通过在 Effect_command 中调用框架里的 Equalizer_getParameter 以及 Equalizer_setParameter 部分,可获取框架中所需的中心频率、频段、增益以及预设

值等参数。从输出日志可见,这部分一直处于线程中,均衡器每改变一次的频段增益,都会重新执行一次 Equalizer_getParameter 以及 Equalizer_setParameter 来使用户设置的均衡器增益生效。图 1 为 Effect_command 部分主要的跳转过程。

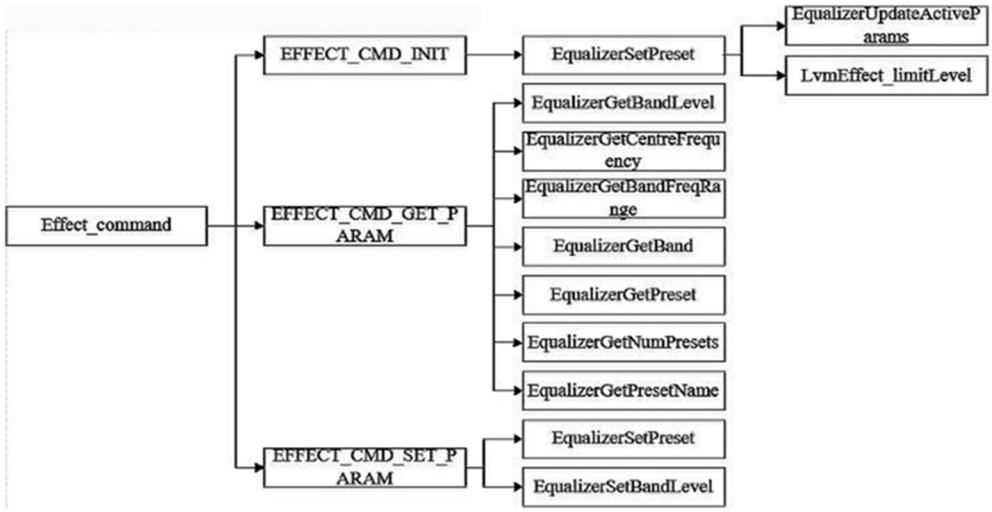

图 1　Effect_command 主要跳转流程

Fig.1　Major jump flow of Effect_command

1.3　音效处理

当框架初始化以及各类参数配置完成后,系统便会根据用户选择的模式对音频数据进行相应的处理了。原生 Audio Effect 框架中音效处理部分是在 Effect_process 中,这部分主要是对输入的音频数据根据用户需要进行相应计算,即对音频数据进行处理并输出。通过阅读相关代码可知,Effect_process 部分会调用框架里的 LvmBundle_process 部分,这个部分也可以看出 Audio Effect 框架仅支持 16 bit 数据的处理,如表 2 中第 1 以及第 2 行代码所示。在 LvmBundle_process 中继续调用外部的 LVM_process,一层层往下调用进行,直到获取到对应的数据处理的具体实现方法,并对输入的音频数据进行处理。这也是 Android 系统代码结构的一大特点,封装成多层,这样有助于减少耦合,便于后期的代码维护。图 2 为音效处理部分的主要代码跳转情况示意图。

表 2　LvmBundle_process 部分代码

Tab.2　Part of the LvmBundle_process code

行号	源码
1	int LvmBundle_process(LVM_INT16　*pIn,
2	LVM_INT16　*pOut,
3	int frameCount,
4	EffectContext　*pContext){
5	LVM_ControlParams_t ActiveParams;
6	LVM_ReturnStatus_en LvmStatus = LVM_SUCCESS;
7	LVM_INT16 *pOutTmp;}

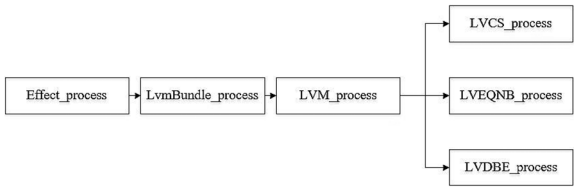

图 2　音效处理部分主要代码跳转情况

Fig.2　Major code jumps in the audio processing section

2　Android 原生 Audio Effect 框架改进方案设计

2.1　改进 Audio Effect 框架代码实现

由于 32 bit 音频数据在精度上对于 16 bit 音频数据会高出许多,故将 32 bit 转换为 16 bit 处理完成后,还应将处理结果还原为 32 bit 音频数据进行输出。在音频数据编码中,每一个二进制数对应一个量化电平,将它们依序排列,得到由二

进制脉冲串组成的数字信息流[2]。图 3 为二进制数示意图,其中 d 只能为 1 或 0,且每一位取值可相同。根据二进制与十进制的转换公式(1)可知,显然高位的数据在输出时相对于低位的数据来说更为重要,若不加处理直接进行强制的数据位数转换,易造成处理所得音频数据出现严重的失真现象。

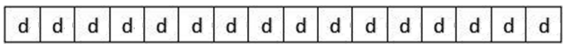

图 3 二进制数示意图

Fig.3 Binary number diagram

$$(N)_{10} = d_n * 2^{n-1} + d_{n-1} * 2^{n-2} + \cdots + d_2 * 2^1 + d_1 * 2^0 \quad (1)$$

式中,N 为二进制转换为十进制的结果,d_n、d_{n-1}、…、d_2、d_1 为二进制各位的系数,n 即为位数,位数从右至左依次为 1、2、…、$(n-1)$、n。

为保留音频数据大部分信息,减轻音频数据由于位数转换造成的失真现象,采取先将音频数据进行移位操作,保留音频数据的高位数据,丢弃低位数据,移位操作完成后再进行位数的强制位数转换操作。经过强制位数转换操作后,32 bit 音频数据便转换为 16 bit 音频数据,此时数据即可通过 Android 原生 Audio Effect 框架处理。该移位操作关键的代码如表 3 所示。

表 3 移位操作关键代码

Tab.3 Key code of the shift operation

行号	代码
1	temp = *src >>16;
2	if (temp > 0x00007FFF)
3	{
4	*dst = 0x7FFF;
5	}
6	else if (temp < -0x00008000)
7	{
8	*dst = - 0x8000;
9	}
10	else
11	{
12	*dst = (LVM_INT16)temp;
13	}

上述代码中,temp 为临时数据,src 为原始数据,dst 为转换后的数据。32 bit 音频数据转换为 16 bit 音频数据,保留高位数据,故需进行右移操作,由于是将 32 bit 音频数据转换为 16 bit 音频数据,故右移位数为 16 位,保留高 16 位数据,且 16 位 int 型的数据可表示范围为 - 32 768 ~ 32 767,故移位后还需注意该数值是否超出 16 位的数据范围,如若超出,应在保留尽可能多的数据的前提下,将其取值为 16 位 int 型数据可表示范围内的数值,上述代码中的两个 if 语句其功能便是判断移位后数据是否超出 16 位可表示的范围,若超出则根据保留尽可能多的数据的前提进行取值;若未超出,便直接强制位数转换。

音频数据在位数转换完成后,经过 Audio Effect 框架处理,处理结束后还需将其转换为 32 bit 的音频数据再进行输出,否则所听到的音乐将会存在极大程度的失真。具体的实现方法与上述将 32 bit 数据转换为 16 bit 数据方法类似,也可通过移位操作实现。将 16 bit 的音频数据转换为 32 bit 的音频数据采用的移位方向为左移,先将处理后的数据强制转换为 32 位 int 型,再将此时低位的 16 位数据通过左移操作移至 32 位中的高位。将移位后的数据输出便完成 32 bit 音频数据音效处理过程。

2.2 改进 Audio Effect 框架 Android 底层实现

由前面对 Android 原生 Audio Effect 框架的分析可知,在原生 Audio Effect 框架中,音频数据输入后,对其进行相应处理的是在框架中的 Effect_process 部分,且该部分的核心处理入口为 LvmBundle_Process,因为原生 Audio Effect 框架中通过 LvmBundle_Process 关联到外部数据处理的实现方法。而该处理部分要求待处理的音频数据必须为 16 bit 音频数据,故 32 bit 音频数据转换为 16 bit 音频数据的操作应在 Effect_process 部分中的 LvmBundle_Process 之前,而输出时还原为 32 bit 音频数据的操作应在 LvmBundle_Process 之后。

在前面代码实现完成后,将两个移位代码实现用 C 语言编写好相关代码后,分别存储为两个.c 文件,并将其在 Android 源代码中引入。引入方法为将 16 bit 音频数据转为 32 bit 音频数据的代码文件以及 32 bit 音频数据转为 16 bit 音频数据的代码文件放入 frameworks/av/media/libeffects/lvm/lib/common/src 中,在 frameworks/av/media/libeffects/

lvm/lib 中的 Android.mk 文件中的 LOCAL_SRC_FILES 这一项中添加两个代码文件索引,并在 frameworks/av/media/libeffects/lvm/lib/Common/lib/VectorArithmetic.h 中添加两个函数的声明,否则在编译时会出错。而在使用时,仅需在原生 Audio Effect 框架中调用这两个函数即可,调用时需注意对应的输入输出数据需配置正确。

之所以将本文所提两个实现方法单独存储为两个代码文件而不在原生 Audio Effect 框架中将这两个具体实现方法直接写入,是学习 Android 源代码中的编程思想,即将该具体的实现方法放在框架之外,使用时只需在相应位置调用该实现方法即可,减少了每一部分的代码量,也便于后期对代码进行维护。

3　实验测试及结果分析

在完成相应的代码编写工作并编译通过后,将编译生成的.so 文件通过 adb 工具 push 到机器中的相应位置,并重启机器使其生效。在将编译生成的.so 文件 push 到机器时,需要使用到四条 adb 命令,分别为 adb root、adb remount、adb push 以及 adb reboot。其中 adb root 为获取机器的开发权限,只有获取了开发权限才可对机器内文件进行修改; adb remount 为将机器重新挂载; adb push 为将指定文件 push 到机器中的指定位置; adb reboot 即为重启机器命令,重启后才会将本次所做更改生效。图 4 为使用 adb 将其中一个.so 文件 push 到机器中并使其生效的过程图。

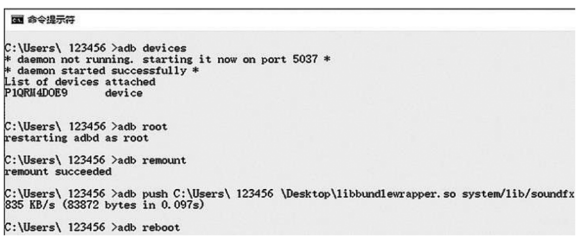

图 4　改进方案实现

Fig.4　Implementation of the improved solution

本设计是在搭载有 Android5.1 系统的音乐播放器中进行测试,完成相应开发工作后,使用该播放器播放 32 bit 的音频数据,并打开均衡器设置均衡器增益,从日志输出中可以看出系统在播放音乐时使用的为原生 Audio Effect 框架处理,即为图 5 中的 Bundle 输出。其次从频率分析图、频率分析数据以及波形图中可以明显看出,均衡器的设置显然已经产生相应的效果,说明原生 Audio Effect 框架已可处理 32 bit 音频数据。

```
V/Bundle  (  153):  EqualizerGetBandLevel -> -6
V/AudioEffect( 1666): setParameter: param: 2, param2: 4
```

图 5　日志输出

Fig.5　Log output

图 6 为系统处理前后的频率分析图,其中横坐标 f 为频率,纵坐标 m 为振幅强度的量级大小。从图 6 可以看出设置均衡器增益前的频率分析图于设置均衡器增益后的频率分析图已有所差异,由于整段频率分析图较大,本文仅截取其中一部分。从表 4 的频率分析数据也可看出,音频数据已根据用户操作发生相应的变化,说明系统已能够对用户所要求的处理做出相应的动作。

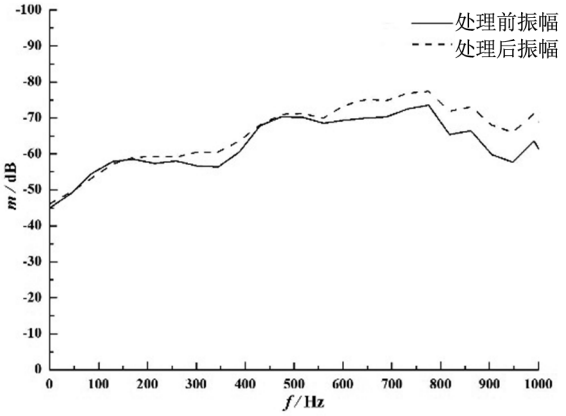

图 6　处理前后频率分析图

Fig.6　Frequency analysis chart before and after processing

表 4　处理前后部分频率分析数据

Tab.4　Partial frequency analysis data before and after processing

频率/Hz	处理前/dB	处理后/dB
86.13	−57.38	−57.18
129.20	−61.58	−62.56
344.53	−68.91	−72.59
646.00	−71.60	−80.03
990.53	−59.19	−68.33
2 024.12	−60.00	−69.87
4 005.18	−59.56	−69.21
6 287.70	−52.12	−59.09
9 991.41	−83.26	−89.20
16 020.70	−102.61	−97.80

4 结论

本文通过对 Android 原生 Audio Effect 框架部分源码进行详细分析,针对 Android 原生 Audio Effect 框架仅支持 16 bit 音频数据处理的问题,采用对输入数据先进行移位操作再进行强制位数转换的方法,达到使 Android 原生 Audio Effect 框架在支持 16 bit 音频数据以及 32 bit 音频数据的处理的同时尽可能控制音频系统失真程度,丰富 Android 系统对于不同品质音乐的支持。且本文采用底层开发方案,改动 Android 原生 Audio Effect 框架部分源码,虽在实现过程中较为繁琐,但对于后期移植开发来说,在很大程度减小了开发工作量,且在实现上,基本达到预期目标,具有一定的应用前景。

参考文献:

[1] RAMO J, VALIMAKI V. Optimizing a high-order graphic equalizer for audio processing[J]. IEEE Signal Processing Letters, 2014, 21(3): 301-305.
[2] 吴礼仲. 音频均衡器算法研究与实现[D]. 西安: 西安电子科技大学, 2010.
[3] JIANG J. Audio processing with channel filtering using DSP techniques[C]//2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC). New York, USA: IEEE, 2018: 545-550.
[4] 陈三强. 基于 DSP 的数字音效系统设计[D]. 湘潭: 湘潭大学, 2016.
[5] 朱增友. 基于 DSP 的综合音效处理器的研究与设计[D]. 西安: 西安工程大学, 2012.
[6] 郝健. 基于 Android 平台的均衡器算法研究[D]. 大连: 大连理工大学, 2011.
[7] 冯启朋, 杨飞然, 杨军. 基于 Android 平台的音效系统设计与实现[J]. 网络新媒体技术, 2016, 5(4): 16-22.
[8] 沈清越. 音频工作中采样率与量化位深的选择[J]. 音响技术, 2013(2): 54-55, 60.
[9] 尤毅. 浅析数字音频系统失真[J]. 音响技术, 2011(3): 53-58.
[10] 刘栋. 数字音频中合理精度转换的重要性[J]. 乐府新声(沈阳音乐学院学报), 2018, 36(3): 101-108.
[11] 谢明. 数字音频技术及应用[M]. 北京: 机械工业出版社, 2017: 99.

(责任编辑: 方素华)