Android* Low-Latency Audio on x86-based Mobile Devices

Objective

This document explains how Android* low-latency audio is implemented on x86 devices starting with the Intel® Atom™ processor-based (codenamed Bay Trail) platform. You can use this guide to aid your investigation of low-latency audio development methods on Intel® devices with low-latency Android build (4.4.4).

Note: Android M Release audio is still under investigation.

Introduction

Android has long been unsuccessful at producing a low-latency audio solution for applications that are focused on sound creation. High latencies negatively impact music creation, game, DJ, and karaoke apps. User interactions on these applications create sound, and end users find the resulting delay in the audible signal to be too high, thus negatively impacting their user experience.

Latency is the delay resulting from when an audio signal is created to when it is played back by some interaction. Round-Trip Latency, or RTL, is the time delay from when an input action by a system or user prompts for a signal and the time it takes to generate an outbound signal.

Users experience audio latency in Android applications when they touch the object to generate sound and the sound is delayed for some time before it is output to the speaker. On most ARM*- and x86-based devices, audio RTL can be measured as low as 300ms and as high as 600ms, mostly in applications developed using the Android method for audio found here: Design for Reduced Latency. These ranges are not acceptable by the user base. The desired latencies have to be well below 100ms, and in most cases under 20ms is the most desired RTL. What also has to be taken into account is the overall latency generated by Android in touch-based music applications which is the total of Touch Latency, Audio Processing Latency, and Buffer Queuing.

This document will only focus on reducing Audio Latency and not total latency; however, it does account for the bulk of the total latency.
The Android Design for Audio

Android’s audio Hardware Abstraction Layer (HAL) connects the higher level, audio-specific framework APIs in android.media to the underlying audio driver and hardware.

You can see how the audio framework is diagramed here: https://source.android.com/devices/audio/index.html

OpenSL ES*

Android specifies the use of OpenSL ES APIs to develop the most robust method for efficiently processing round-trip audio. Though it is not the best option for low-latency audio support, it is the recommended option. This is primarily due to the buffer queuing mechanism OpenSL utilizes, making it more efficient within the Android Media Framework. Since the implementation is native code, it may deliver better performance because native code is not subject to Java* or Dalvik VM overheads. We assume that this is the way forward for audio development on Android. As specified in the Android Native Development Kit (NDK) documents for Open SL, Android releases will continue to improve upon Open SL implementations.

This document will examine the use of the OpenSL ES API through the NDK. As an introduction to OpenSL, examine the three layers that make up the code base for Android Audio using OpenSL.

- Top-level application programming environment is the Android SDK, which is Java based.
- Lower-level programming environment, called the NDK, allows developers to write C or C++ code that can be used in the application via the Java Native Interface (JNI).
- OpenSL ES API, which has been implemented since Android 2.3 and is built into the NDK.

OpenSL operates, like several other APIs, by employing a callback mechanism. In OpenSL the callback can only be used to notify the application that a new buffer can be queued (for playback or for recording). In other APIs, the callback also handles pointers to the audio buffers that are to be filled or consumed. But in OpenSL, by choice, the API can be implemented so the callbacks operate as a signaling mechanism to keep all the processing in your audio processing thread. This would include queuing the required buffers after the assigned signals are received.

Google recommends using a method within OpenSL called the Sched_FIFO policy. The Sched_FIFO policy is based on a ring or circular buffer technique.

Sched_FIFO Policy

Since Android is based on Linux*, Android institutes the Linux CFS scheduler. The CFS may allocate CPU resources in unexpected ways. For example, it may take the CPU away from a thread with numerically low niceness onto a thread with a numerically high niceness. In the case of audio, this can result in buffer timing issues.

The primary solution is to avoid CFS for high-performance audio threads and use the SCHED_FIFO scheduling policy instead of the SCHED_NORMAL (also called SCHED_OTHER) scheduling policy implemented by CFS.
Scheduling latency

Scheduling latency is the time between when a thread becomes ready to run, and when the resulting context switch completes so that the thread actually runs on a CPU. The shorter the latency the better, and anything over two milliseconds causes problems for audio. Long scheduling latency is most likely to occur during mode transitions, such as bringing up or shutting down a CPU, switching between a security kernel and the normal kernel, switching from full-power to low-power mode, or adjusting the CPU clock frequency and voltage.

A Circular Buffer Interface

The first thing to do to test that the buffer is implemented correctly is to prepare a circular buffer interface that the code can use. You need four functions for this: 1) create a circular buffer, 2) Write to it, 3) Read from it, 4) Destroy it.

Code example:

```
circular_buffer* create_circular_buffer(int bytes);
int read_circular_buffer_bytes(circular_buffer *p, char *out, int bytes);
int write_circular_buffer_bytes(circular_buffer *p, const char *in, int bytes);
void free_circular_buffer (circular_buffer *p);
```

The intended effect is that the read operation will only read the number of requested bytes up to what has been written in the buffer already. The write function will only write the bytes for which there is space in the buffer. They will return a count of read/written bytes, which can be anything from zero to the requested number.

The consumer thread (the audio I/O callback in the case of playback, or the audio processing thread in the case of recording) reads from the circular buffer and then does something with the audio. At the same time, asynchronously, the supplier thread is filling the circular buffer, stopping only if it gets filled up. With an appropriate circular buffer size, the two threads will cooperate seamlessly.

Audio I/O

Using the interface as created in the example before, audio I/O functions can be written to use OpenSL callbacks. An example of an input stream I/O function is:

```
// this callback handler is called every time a buffer finishes recording
void bqRecorderCallback(SLAndroidSimpleBufferQueueItf bq, void *context)
{
    OPENSL_STREAM *p = (OPENSL_STREAM *) context;
    int bytes = p->inBufSamples*sizeof(short);
    write_circular_buffer_bytes(p->inrb, (char *) p->recBuffer, bytes);
    (*p->recorderBufferQueue)->Enqueue(p->recorderBufferQueue,p->recBuffer,bytes);
}

// gets a buffer of size samples from the device
int android_AudioIn(OPENSL_STREAM *p,float *buffer,int size){
    short *inBuffer;
    int i, bytes = size*sizeof(short);
    bytes = read_circular_buffer_bytes(p->inrb, (char *)p->inputBuffer,bytes);
    size = bytes/sizeof(short);
    for(i=0; i < size; i++){
        buffer[i] = (float) p->inputBuffer[i]*CONVMYFLT;
    }
    if(p->outchannels == 0) p->time += (double) size/(p->sr*p->inchannels);
    return size;
}
In the callback function (lines 2-8), which is called every time a new full buffer (recBuffer) is ready, all of the data is written into the circular buffer. Then the recBuffer is ready to be queued again (line 7). The audio processing function, lines 10 to 21, tries to read the requested number of bytes (line 14) into inputBuffer and then copies that number of samples to the output (converting it into float samples). The function reports the number of copied samples.

Output Function:

```c
// puts a buffer of size samples to the device
int android_AudioOut(OPENSL_STREAM *p, float *buffer, int size){
  short *outBuffer, *inBuffer;
  int i, bytes = size*sizeof(short);
  if(p == NULL || p->outBufSamples == 0) return 0;
  for(i=0; i < size; i++){
    p->outputBuffer[i] = (short) (buffer[i]*CONV16BIT);
  }
  bytes = write_circular_buffer_bytes(p->outrb, (char *) p->outputBuffer,bytes);
  p->time += (double) size/(p->sr*p->outchannels);
  return bytes/sizeof(short);
}
```

The audio processing function (lines 2-13) takes in a certain number of float samples, converts them to shorts, and then writes the full outputBuffer into the circular buffer, reporting the number of samples written. The OpenSL callback (lines 16-22) reads all of the samples and queues them.

For this to work properly, the number of samples read from the input needs to be passed along with the buffer to the output. Below is the processing loop code that loops the input back into the output:

```c
while(on)
  samps = android_AudioIn(p,inbuffer,VECSAMPS_MONO);
  for(i = 0, j=0; i < samps; i++, j+=2)
    outbuffer[j] = outbuffer[j+1] = inbuffer[i];
  android_AudioOut(p,outbuffer,samps*2);
```

In this snippet, lines 5-6 loop over the read samples and copy them to the output channels. It is a stereo-out/mono-in setup, and for this reason the input samples are copied into two consecutive output buffer locations. Now that the queuing is happening in the OpenSL threads, in order to start the callback mechanism, we need to queue a buffer for recording and another for playback after we start audio on the device. This will ensure the callback is issued when buffers need to be replaced.

This is an example of how to implement an audio I/O track thread for processing through OpenSL. Each implementation is going to be unique and will require modifications to the HAL and ALSA driver to get the most from the OpenSL implementation.
x86 Design for Android Audio

OpenSL implementations do not guarantee a low-latency path to the Android “fast mixer” for all devices with a desirable rate of delay (under 40ms). However, with the modifications to the Media Server, HAL, and the ALSA driver, different devices can achieve varying success in low-latency audio. While conducting research on what is required to drive latencies down on Android, Intel has implemented a low-latency audio solution on the Dell* Venue 8 7460 tablet.

The result of the experiment is a hybrid media processing engine in which the input processing thread is managed by a separate low-latency input server that processes the raw audio and then passes it to the Android-implemented media server that still uses the “fast mixer” thread. Both the input and output servers use the scheduling in the OpenSL Sched_FIFO policy.

Figure 1. Implementation Diagram.
The result of this modification is a very satisfactory 45-ms RTL. This implementation is part of the Intel Atom SoC and tablet design used for this effort. This test is conducted on an Intel Software Development Platform and is available through the Intel Partner Software Development Program.

The implementation of OpenSL and the SCHED_FIFO policy exhibits efficient processing of the round-trip, real-time audio on the above-specified hardware platform and is not available for all devices. Testing any application using the examples given in this document must be conducted on specific devices and can be made available to partner software developers.

Summary

This article discussed how to use OpenSL to create a callback and buffer queue in an application that will adhere to the Android audio development methods. It also demonstrated the efforts put forth by Intel to provide one option for low-latency audio signaling using the modified Media Framework. To conduct this experiment and test the low latency path, developers must follow the Android development design for audio using OpenSL and an Intel Software Development Platform on Android Kit Kat 4.4.2 or higher.

Contributors:
Eric Serre, Intel Corporation
Victor Lazzarini

Notices

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.
The products described in this document may contain design defects or errors known as errata which
may cause the product to deviate from published specifications. Current characterized errata are
available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing
your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel
literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm.

Intel, the Intel logo, and Intel Atom are trademarks of Intel Corporation in the U.S. and other countries.

* Other names and brands may be claimed as the property of others.

Copyright © 2014 Intel Corporation. All rights reserved