Understanding Full-HD Voice Communication On Android-Based Devices

-Ing. Matthias Teßmann

Fri, 2014-03-14 16:02

Today's cloud and media streaming services allow us to experience high-quality audio and video content everywhere and at any time. The availability of thousands of CD-quality songs in our pocket and HD streaming right from the Internet, all made possible through highly efficient audio coding algorithms such as the Advanced Audio Codec (AAC), leads to one question: What about the phone call? While we have grown accustomed to "HD everywhere," the audio quality of the traditional phone call is still roughly the same as it was 100 years ago. Can't we do better?

Related Articles

- Understanding MPEG Audio Codecs From mp3 To xHE-AAC
- Fraunhofer Video Conferencing Q and A
- The Sounds Of CES 2012

From a technical perspective, sound is little more than continuous-time varying pressure waves, i.e. vibrations of air that our ear absorbs and our brain interprets. As such, sound can be characterized by its frequency (vibrations per second, unit Hz) and its amplitude (observed volume). Higher frequencies correspond to high tones, and lower frequencies correspond to low tones. The difference between the frequency of the highest audible tone and the frequency of the lowest audible tone is called the bandwidth of the audio signal.

Today's Standards

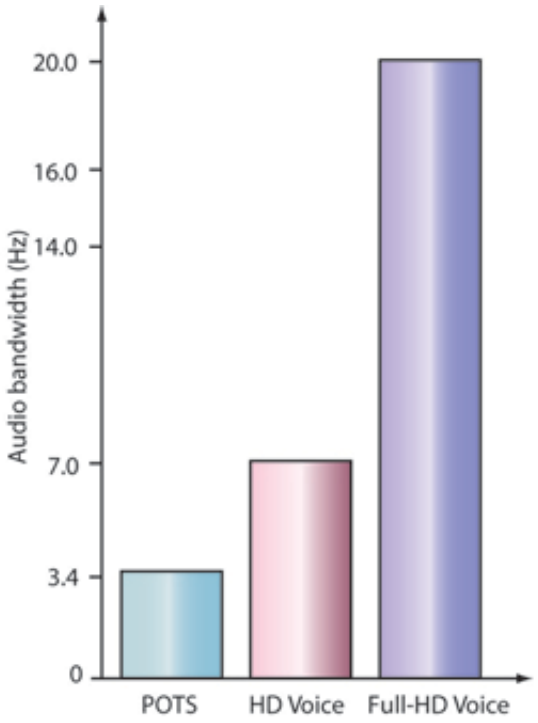
The human ear can hear audio bandwidths of up to 20 kHz-thus, the higher the bandwidth within an audio signal, the higher the perceived audio quality. When an audio signal is recorded digitally, sampling discretizes the continuous signal, where the sampling rate must be at least twice the original frequency to be able to fully reconstruct the original signal.

If we wanted to record and transmit stereo audio data for voice communication with the full audible bandwidth in 16-bit PCM format, we would need a transmission channel bandwidth of 2*16*40,000 = 1,280,000 bits/s or 1280 kbits/s-clearly too much for audio data alone. Due to bitrate restrictions, the audio bandwidth of voice calls is therefore usually limited to about 3.4 kHz, or a sampling rate of 8 kHz leading to a required bitrate of 64 kbits/s when the audio data is stored as an 8-bit value. This is the bandwidth at which voice is just about understandable.

But on traditional mobile networks (GSM), things get even more problematic because the available bit rate for voice calls is only about 13 kbits/s. This led to the development of lossy voice codecs for mobile communication applications that further degrade the audio quality.

Luckily, carrier networks evolved and developments such as LTE promise higher data rates and lower latencies, leading to the development of wide-band communication codecs such as AMR-WB that promise HD Voice quality, with a transmitted bandwidth of up to 7 kHz. While HD Voice is a real quality gain for communication, the quality gain from HD

Voice to Full-HD Voice, i.e. a transmitted audio bandwidth of 14 to 20 kHz, is as big as the gain from the 3.4-kHz plain old telephony service (POTS) to HD Voice *(see the figure)*. Full-HD Voice quality then can be achieved at bitrates comparable to those required by HD Voice codecs.



To use the full audible audio bandwidth for voice-over-IP calls and experience Full-HD Voice communication, use a high-quality audio codec that can encode the audio data with sufficient low coding latencies at low bitrates. The MPEG AAC Enhanced Low Delay (AAC-ELD) audio codec, a member of the AAC family of audio codecs, is a Full-HD Voice capable codec that meets these requirements.1 With AAC-ELD, the full audible audio bandwidth can be encoded at typical bitrates of only 24 to 64 kbits/s. Moreover, the very low coding latency of 15 to 32 ms makes this codec ideal for real-time communication applications.

With the introduction of FaceTime, Apple already has used AAC-ELD to deliver a new communication experience to its users. The number of people who now use video conferencing on their iPhone, iPad, and Mac devices and enjoy excellent audio and video quality is increasing.

What makes AAC-ELD particularly interesting for Android developers is the recent inclusion of a fully featured AAC-ELD encoder and decoder by Google into the Android operating system starting with version 4.[1] (Jelly Bean). Moreover, specific brand implementations or distributions by vendors other than Google are also required to integrate the audio codec to fulfill the Google compatibility definitions.[2] This essentially guarantees that every new Android-compatible device can use AAC-ELD for audio encoding and decoding, creating an extensive user base for new high-quality communication software on Android devices.

Using AAC-ELD On Android

AAC-ELD became a part of Android when Google integrated the Fraunhofer FDK AAC codec library with its 4.1 release (Jelly Bean). As a component of the operating system, the AAC-ELD encoder and decoder are available to application

developers through the MediaCodec class provided by the android.media package.[3]

Based on this class, we created an AacEldEncoder and an AacEldDecoder class that wrap the MediaCodec interface into a straightforward and simple to use encoder and decoder abstraction. Moreover, creating this simple abstraction allows for easy reuse of the encoder and decoder classes when accessing them through the JNI interface for native applications.

Listing 1 provides the class definitions of the AacEldEncoder and AacEldDecoder classes. Only three methods for each class are sufficient to perform AAC-ELD encoding and decoding: configure() and encode() for the encoder as well as decode() for the decoder. A close() method is provided for resource cleanup.

1. Class Definitions For AacEldEncoder and AacEldDecoder

```
public class AacEldEncoder {
  // The encoder instance
  private MediaCodec m_encoder;
  // Encoder output buffer information
  private MediaCodec.BufferInfo m_bufferInfo;
  // Initialization state
  private Boolean m_isInitialized = false;

  // MIME type and encoder name for AAC-ELD encoding
  private final static String MIME_TYPE        = "audio/mp4a-latm";
  private final static String OMX_ENCODER_NAME = "OMX.google.aac.encoder";

  public AacEldEncoder() {}

  public byte[] configure(int sampleRate, int channelCount, int bitrate) { /* ... */  }

  public byte[] encode(byte[] pcmFrame) { /* ... */  }

  public void close() { /* ... */ }
}

public class AacEldDecoder {
  // The MediaCodec instance used as decoder
  private MediaCodec m_decoder;
  // Output buffer information
  private MediaCodec.BufferInfo m_bufferInfo;

  private Boolean m_isInitialized = false;

  // The constant mime-type and decoder name for AAC
  private final static String MIME_TYPE        = "audio/mp4a-latm";
  private final static String OMX_DECODER_NAME = "OMX.google.aac.decoder";

  public AacEldDecoder() {}

  public boolean configure(byte[] asc) { /* ... */ }

  public byte[] decode(byte[] au) { /* ... */  }

  public void close() { /* ... */  }
}
```

Despite their different purposes, the encoder and the decoder instances are configured very similarly. Five individual steps must be performed to successfully configure a MediaCodec instance for AAC-ELD encoding or decoding.

First, create an instance of the class android.media.MediaFormat that will describe the audio format by specifying a Multipurpose Internet Mail Extensions (MIME) type, the sample rate, and the desired number of audio channels. The MIME type for audio coding with AAC and thus also AAC-ELD on Android is "audio/mp4a-latm." Other MIME types will not work for creating an AAC encoder or decoder.

Note that the sampling rate and the number of audio channels are only required for encoder initialization. The corresponding configuration for the decoder is included within an audio-specific configuration (ASC) that should be generated by a standard conforming AAC-ELD encoder.[4] Therefore, these values can be set to 0 during decoder configuration.

Second, to configure additional parameters of the audio encoder, use the method setInteger(key, value) of the just created MediaFormat instance. The individual configuration parameters are specified as key-value pairs.[5] To enable AAC-ELD encoding in the AAC encoder object, the value of the key KEY_AAC_PROFILE must be set to MediaCodecInfo.CodecProfileLevel.AACObjectELD. The desired encoder output bitrate can be specified using the key KEY_BIT_RATE.

To initialize the decoder, neither the profile level key nor the bitrate key has to be set, as these parameters are also included within the ASC provided by the encoder. The ASC can be transferred to the decoders MediaFormat instance by calling the method setByteBuffer(key, buffer) using the string "csd-0" as key (csd = codec specific data).

Third, an encoder and decoder instance can be created by calling the static class method MediaCodec.createByCodecName(name). Currently the name for the AAC encoder is "OMX.google.aac.encoder," and the name for the decoder is "OMX.google.aac.decoder." These names can be obtained by using the getCodecInfoAt() method of the android.media.MediaCodecList class.

Internally, for AAC coding, the Android MediaCodec implementation is a wrapper using the native Fraunhofer FDK AAC codec libraries. The valid codec names may vary, depending on the respective Android OS version. Be sure to check if the name is still valid when deploying your software for a different OS version!

Fourth, configure the codec with the previously created MediaFormat instance by calling the method configure() on the codec instance. To configure a MediaCodec instance for encoding, the constant MediaCodec.CONFIGURE_FLAG_ENCODE has to be passed as the last parameter to the configure() call.

Finally, start the encoder and decoder by calling start() on the respective MediaCodec instance. Once the individual MediaCodec instances for AAC-ELD encoding and decoding are initialized and started, they can be used for encoding and decoding LPCM to AAC-ELD and vice versa.

Encoding LPCM samples to AAC-ELD access units (AUs) using the MediaCodec application programming interface (API) is buffer based and works on audio buffers provided by the Android-internal codec wrapper. Therefore, an input buffer index has to be requested first by calling the method dequeueInputBuffer(timeout) on the encoder instance.

In the demo implementation, a timeout value of 0 is used, meaning the call will return immediately to avoid thread blocking. If a valid buffer is returned (index value >= 0), it can be accessed by calling getInputBuffers() and indexing the returned array of ByteBuffer objects. The LPCM data that should be encoded now can be copied into this buffer. Finally, a call to queueInputBuffer() is used to tell the encoder that the data in the input buffer is available for encoding.

To obtain the encoded AU, an output buffer index has to be requested by calling dequeueOutputBuffer(bufferInfo,

timeout) on the encoder instance. In contrast to the dequeueInputBuffer() method, dequeueOutputBuffer() receives an additional parameter of type MediaCodec.BufferInfo.

The BufferInfo instance is used to describe the returned output buffer with respect to size and type. If an output buffer is available, it can be accessed by indexing the ByteBuffer array returned from a call to getOutputBuffers(). Once the output data like the encoded AU has been copied from the returned buffer, it has to be released for reuse by calling releaseOutputBuffer(bufferIndex, …).

Note that the first buffer obtained from a correctly configured MediaCodec encoder instance is usually the ASC that is required for decoder initialization. The ASC can be distinguished from an encoded AU buffer by checking the flags field of the BufferInfo instance. If the flags field is equal to MediaCodec.BUFFER_FLAG_CODEC_CONFIG, then the returned output buffer includes an ASC.

Listing 2 provides the complete implementation of the configure() and encode() methods of the AacEldEncoder class. Since the ASC will be generated right after codec configuration, that data is already obtained during configuration and returned by the configure() method. Note that the actual encoding (and decoding) of data happens asynchronously. There might be additional delay before output buffers are available.

2. Configuration And Encoding With The AacEldEncoder Class

```
public class AacEldEncoder {

/* ... */

public byte[] configure(int sampleRate, int channelCount, int bitrate) {
    try {
      MediaFormat mediaFormat
        = MediaFormat.createAudioFormat(MIME_TYPE, sampleRate, channelCount);
      mediaFormat.setInteger(MediaFormat.KEY_AAC_PROFILE,
        MediaCodecInfo.CodecProfileLevel.AACObjectELD);
      mediaFormat.setInteger(MediaFormat.KEY_BIT_RATE, bitrate);

      m_encoder = MediaCodec.createByCodecName(OMX_ENCODER_NAME);
      m_encoder.configure(mediaFormat, null, null, MediaCodec.CONFIGURE_FLAG_ENCODE);

      m_encoder.start();

      m_bufferInfo = new MediaCodec.BufferInfo();

      int ascPollCount = 0;
      byte[] aubuf = null;

      while (aubuf == null && ascPollCount < 100) {
        // Try to get the asc
        int encInBufIdx = -1;
        encInBufIdx = m_encoder.dequeueOutputBuffer(m_bufferInfo, 0);
        if (encInBufIdx >= 0) {
          if (m_bufferInfo.flags == MediaCodec.BUFFER_FLAG_CODEC_CONFIG) {
            aubuf = new byte[m_bufferInfo.size];
            m_encoder.getOutputBuffers()[encInBufIdx].get(aubuf, 0, m_bufferInfo.size);
            m_encoder.getOutputBuffers()[encInBufIdx].position(0);
            m_encoder.releaseOutputBuffer(encInBufIdx, false);
          }
        }
        ++ascPollCount;
```

```
      }

      if (aubuf != null)
        m_isInitialized = true;

      return aubuf;
    } catch (Exception e) {
      System.out.println("ERROR configuring the encoder: " + e.getMessage());
      return null;
    }
  }

  public byte[] encode(byte[] pcmFrame) {
    try {
      if (!m_isInitialized)
        return null;

      // When we have a valid PCM frame we enqueue
      // it as an input buffer to the encoder instance
      if (pcmFrame != null) {
        int encInBufIdx = m_encoder.dequeueInputBuffer(0);
        if (encInBufIdx >= 0) {
          m_encoder.getInputBuffers()[encInBufIdx].position(0);
          m_encoder.getInputBuffers()[encInBufIdx].put(pcmFrame, 0, pcmFrame.length);
          m_encoder.queueInputBuffer(encInBufIdx, 0, pcmFrame.length, 0, 0);
        }
      }

      byte[] aubuf = null;
      int aubufPollCnt = 0;

      while (aubuf == null && aubufPollCnt < 100) {
        int encInBufIdx = m_encoder.dequeueOutputBuffer(m_bufferInfo, 0);
        if (encInBufIdx >= 0) {
          aubuf = new byte[m_bufferInfo.size];
          m_encoder.getOutputBuffers()[encInBufIdx].get(aubuf, 0, m_bufferInfo.size);
          m_encoder.getOutputBuffers()[encInBufIdx].position(0);
          m_encoder.releaseOutputBuffer(encInBufIdx, false);
        }
        ++aubufPollCnt;
      }

      return aubuf;

    } catch (Exception e) { // Handle any unexpected encoding issues here
      return null;
    }
  }
}
```

Since the presented implementation requires data to be returned immediately, however, a very simple form of output buffer polling is implemented. The dequeueOutputBuffer() method then is called repeatedly until a valid output buffer index is obtained. To prevent application blocking, the number of tries is limited before a null object is returned to signal an error.

Encoded AUs are decoded into LPCM frames accordingly, as the same API is used for this purpose. Listing 3 shows the implementation. The only notable difference between the two implementations is the configure() method and the way the ASC is passed to the decoder. When audio processing is finished, the encoder and decoder should be stopped by calling stop() and their internal resources should be released with a call to release(). This is done in the close() method.

3. Configuration And Decoding With The AacEldDecoder Class

```
public class AacEldDecoder {

  /* ... */

  public boolean configure(byte[] asc) {
    try {
      MediaFormat mediaFormat = MediaFormat.createAudioFormat(MIME_TYPE, 0, 0);

      ByteBuffer ascBuf       = ByteBuffer.wrap(asc);
      mediaFormat.setByteBuffer("csd-0", ascBuf);

      // Create decoder instance using the decoder name
      m_decoder = MediaCodec.createByCodecName(OMX_DECODER_NAME);
      // Configure the decoder using the previously created MediaFormat instance
      m_decoder.configure(mediaFormat, null, null, 0);

      // Start the decoder
      m_decoder.start();

      // Create object for output buffer information
      m_bufferInfo = new MediaCodec.BufferInfo();
      m_isInitialized = true;

    } catch (Exception e) {
      System.out.println("ERROR configuring the decoder: " + e.getMessage());
      m_isInitialized = false;
    }

    return m_isInitialized;
  }

  public byte[] decode(byte[] au) {
    try {
      if (!m_isInitialized)
        return null;

      if (au != null) {
        int decInBufIdx = m_decoder.dequeueInputBuffer(0);
        if (decInBufIdx >= 0) {
          m_decoder.getInputBuffers()[decInBufIdx].position(0);
          m_decoder.getInputBuffers()[decInBufIdx].put(au, 0, au.length);
          m_decoder.queueInputBuffer(decInBufIdx, 0, au.length, 0, 0);
        }
      }

      byte[] pcmbuf = null;
      int pcmbufPollCnt = 0;

      while (pcmbuf == null && pcmbufPollCnt < 100) {
        int decBufIdx = m_decoder.dequeueOutputBuffer(m_bufferInfo, 0);
        if (decBufIdx >= 0) {
          pcmbuf = new byte[m_bufferInfo.size];
          m_decoder.getOutputBuffers()[decBufIdx].get(pcmbuf, 0, m_bufferInfo.size);
          m_decoder.getOutputBuffers()[decBufIdx].position(0);
          m_decoder.releaseOutputBuffer(decBufIdx, false);
        }
        ++pcmbufPollCnt;
      }
```

```
      return pcmbuf;
    } catch (Exception e) {
      return null;
    }
  }

  public void close() { /* ... */  }
}
```

When implementing a fully featured VoIP application, it is often required that most of the audio data is processed in native code, resorting back to the Java environment to provide the user interface and perform other higher-level tasks. While the device's sound card can be accessed from native code quite easily using the OpenSL APIs provided by the Android NDK, there is no such native API or library for using the audio coding facilities in Android.[6,7] Native applications or application libraries that want to use the AAC-ELD codec built into the Android OS thus will have to use the Java Native Interface (JNI) API, not to call native code from Java, but to call the Java MediaCodec APIs from native code.[8]

Accessing The AAC-ELD Encoder And Decoder

To access the Java AacEldEncoder and AacEldDecoder classes from native code, we define a similar interface in C++ that can be used to encode and decode audio data (Listing 4). These classes use the "pointer-to-implementation" idiom to hide the JNI part of the implementation from the user. Because the implementations of the C++ AacEldEncoder and AacEldDecoder classes are almost identical, only the implementation of the AacEldEncoder will be discussed.

4. AacEldEncoder And AacEldDecoder Class Definitions

```
#include

// Encoder class
class AacEldEncoder {
public:
  AacEldEncoder(void *jvmHandle);
  ~AacEldEncoder();


  bool configure(unsigned int sampleRate,
                 unsigned int nChannels,
                 unsigned int bitrate,
                 std::vector<unsigned char>& asc);

  bool encode(std::vector<unsigned char>& inSamples, std::vector<unsigned char>& outAU);

  void close();

private:
  class AacEldEncImpl;
  AacEldEncImpl *impl_;
};

// Decoder class
class AacEldDecoder {
public:
  AacEldDecoder(void *jvmHandle);
  ~AacEldDecoder();

  bool configure(std::vector<unsigned char>& asc);
```

```
  bool decode(std::vector<unsigned char>& inAU, std::vector<unsigned char>& outSamples);

  void close();

private:
  class AacEldDecImpl;
  AacEldDecImpl *impl_;
};
```

Before the detailed implementation is shown, Listing 5 introduces a small utility class named JniEnvGuard. The C++ encoder and decoder implementation receive a pointer to a JavaVM type to acquire a JNIEnv pointer later that is required to make JNI calls and access the Java runtime. However, JNIEnv handles are valid only within the context of a single (and current) thread and cannot be shared between individual threads, as the JNIEnv also is used for thread-local storage.

5. JniEnvGuard Class Definition

```
#ifndef __JNI_ENV_GUARD_H__
#define __JNI_ENV_GUARD_H__

#include

class JniEnvGuard {
public:
    explicit JniEnvGuard(JavaVM* vm, jint version = JNI_VERSION_1_6);
    ~JniEnvGuard();

    JNIEnv* operator->();
    JNIEnv const* operator->() const;

private:
  JavaVM *vm_;
  JNIEnv *env_;
  bool   threadAttached_;
};

#endif /* __JNI_ENV_GUARD_H__ */

// Implementation

#include "JniEnvGuard.h"

#include

JniEnvGuard::JniEnvGuard(JavaVM* vm, jint version) : vm_(vm), env_(NULL), threadAttached_(false) {
  jint jniResult = vm_->GetEnv(reinterpret_cast(&env_), version);

  if (jniResult == JNI_EDETACHED) { // Detached, attach
    jint rs = vm_->AttachCurrentThread(&env_, NULL);

    if (rs != JNI_OK) {
      throw std::runtime_error("Error attaching current thread to JNI VM");
    }

    threadAttached_ = true;
  } else if (jniResult != JNI_OK) {
    throw std::runtime_error("Error obtaining a reference to JNI environment");
  }
```

```
  if (env_ == NULL) {
    throw std::runtime_error("JNIEnv* invalid");
  }
}

JniEnvGuard::~JniEnvGuard() {
  if (threadAttached_) {
    vm_->DetachCurrentThread();
  }
}

JNIEnv* JniEnvGuard::operator->() {
  return env_;
}

JNIEnv const* JniEnvGuard::operator->() const {
  return env_;
}
```

As a consequence, any JNIEnv handle obtained through a JavaVM needs to be attached to the current thread of execution before it can be used. Failing to attach the JNIEnv to the current thread before making JNI calls will result in an error and at worst crash your program. Moreover, the Android OS will not detach native threads automatically when they are terminating, so any application attaching a JNIEnv to its current native thread is required to detach itself before the main thread terminates.

The JniEnvGuard class ensures that the JNIEnv pointer obtained through a JavaVM pointer is attached to the current thread upon construction time and detached again when the object is destructed. So instead of creating a JNIEnv handle manually by calling GetEnv() in the following implementation, an object of type JniEnvGuard is created that automatically handles thread attachment and detachment as required. The JniEnvGuard class implements the RAII-idiom for JNIEnv thread attachment and detachment.

Finally, Listings 6 and 7 show the implementation of the C++ AacEldEncoder. The most important method of implementation is initJni(). In this method, the connection between the Java AacEldEncoder implementation and the C++ implementation is set up. First, the AacEldEncoder Java class is looked up by a call to FindClass(). The argument to FindClass() is a string including the fully qualified class name used in Java. For the example application, the package de.fraunhofer.iis.aac_eld_encdec contains the AacEldEncoder class, so the argument to FindClass() becomes "de/fraunhofer/iis/aac_eld_encdec/AacEldEncoder."

6. Initialization Of The AacEldEncoder Class

```
#include
#include

#include

#include "JniEnvGuard.h"

#include "AacEldEncoder.h"

class AacEldEncoder::AacEldEncImpl {
public:
  AacEldEncImpl(void *jvmHandle);
  ~AacEldEncImpl();
```

```
  bool configure(unsigned int sampleRate, unsigned int nChannels,
                 unsigned int bitrate, std::vector<unsigned char>& asc);
  bool encode(std::vector<unsigned char>& inSamples, std::vector<unsigned char>& outAU);
  void close();

private:
  JavaVM*   javavm;
  jclass    aacEldEncoderClass;
  jobject   aacEldEncoderInstance;
  jmethodID configureMethodId;
  jmethodID encodeMethodId;
  jmethodID closeMethodId;
  bool      jniInitialized;

  bool      initJni();
};

bool AacEldEncoder::AacEldEncImpl::initJni() {
  JniEnvGuard env(javavm);

  jclass encoderClass = env->FindClass("de/fraunhofer/iis/aac_eld_encdec/AacEldEncoder");

  if (encoderClass && !aacEldEncoderClass) // Store a global reference for this application
    aacEldEncoderClass = reinterpret_cast<jclass>(env->NewGlobalRef(encoderClass));


  if (!encoderClass) { // in case of an error, first check if
    if (aacEldEncoderClass) { // some thread got wild and we messed up the class loader stack
      encoderClass = aacEldEncoderClass; // try cached class if found before
      if(env->ExceptionCheck() == JNI_TRUE) { // and clear the pending exception that FindClass has
already thrown
        env->ExceptionClear();
      }
    } else { // all bets are off - cannot find class
      jthrowable exc = env->ExceptionOccurred();
      if (exc) {
        env->ExceptionDescribe();
        env->ExceptionClear();
    }
      return false;
    }
  }

  jmethodID encoder_ctor   = env->GetMethodID(encoderClass, "", "()V");
  configureMethodId        = env->GetMethodID(encoderClass, "configure", "(III)[B");
  encodeMethodId           = env->GetMethodID(encoderClass, "encode", "([B)[B");
  closeMethodId            = env->GetMethodID(encoderClass, "close", "()V");

  // It is an error if one of these is not found
  if (!encoder_ctor || !configureMethodId || !encodeMethodId || !closeMethodId) {
    return false;
  }

  // If all methods are found, create a new instance of the AacEldEncoder object
  jobject encoderObj              = env->NewObject(encoderClass, encoder_ctor);

  if (!encoderObj) {
    return false;
  }
```

```
  // Finally create a new global reference (otherwise the
  // just created object will be garbage collected as soon
  // as the current JNI call returns to Java)
  aacEldEncoderInstance = env->NewGlobalRef(encoderObj);

  if (!aacEldEncoderInstance) {
    return false;
  }

  jniInitialized = true;

  return true;
}
```

7. Implementation Of The AacEldEncoder Class

```
AacEldEncoder::AacEldEncImpl::AacEldEncImpl(void *jvmHandle)
: javavm((JavaVM*)jvmHandle),
  aacEldEncoderClass(NULL),
  aacEldEncoderInstance(NULL),
  configureMethodId(NULL),
  encodeMethodId(NULL),
  closeMethodId(NULL),
  jniInitialized(false)
{}

AacEldEncoder::AacEldEncImpl::~AacEldEncImpl() {
  if (jniInitialized)
    close();
}

bool AacEldEncoder::AacEldEncImpl::configure(unsigned int sampleRate, unsigned int nChannels,
unsigned int bitrate, std::vector<unsigned char>& asc) {
  if (!jniInitialized)
    if (!initJni())
      return false;

  JniEnvGuard env(javavm);

  jbyteArray resBuf = (jbyteArray) env->CallObjectMethod(aacEldEncoderInstance,
                                                         configureMethodId,
                                                         sampleRate,
                                                         nChannels,
                                                         bitrate);

   if (!resBuf) {
     return false;
   }

   jsize len  = env->GetArrayLength(resBuf);
   jbyte *buf = env->GetByteArrayElements(resBuf, 0);

   asc.clear();
   asc.resize(len);
   memcpy(&asc[0], buf, sizeof(unsigned char)*len);

   env->ReleaseByteArrayElements(resBuf, buf, 0);
   env->DeleteLocalRef(resBuf);

   return true;
```

```
}

bool AacEldEncoder::AacEldEncImpl::encode(std::vector<unsigned char>& inSamples,
std::vector<unsigned char>& outAU) {

  JniEnvGuard env(javavm);

  jbyteArray inArray = env->NewByteArray(inSamples.size());
  jbyte *inBytes     = env->GetByteArrayElements(inArray, 0);

  memcpy(inBytes, &inSamples[0], sizeof(unsigned char)*inSamples.size());
  env->ReleaseByteArrayElements(inArray, inBytes, 0);

  jbyteArray resBuf = (jbyteArray) env->CallObjectMethod(aacEldEncoderInstance, encodeMethodId,
inArray);
  env->DeleteLocalRef(inArray);

  if (!resBuf)
    return false;

  jsize resLen      = env->GetArrayLength(resBuf);
  jbyte *resByteBuf = env->GetByteArrayElements(resBuf, 0);
  outAU.clear();
  outAU.resize(resLen);
  memcpy(&outAU[0], resByteBuf, sizeof(unsigned char)*resLen);

  env->ReleaseByteArrayElements(resBuf, resByteBuf, 0);
  env->DeleteLocalRef(resBuf);

  return true;
}

void AacEldEncoder::AacEldEncImpl::close() {
  JniEnvGuard env(javavm);
  env->CallVoidMethod(aacEldEncoderInstance, closeMethodId);
  env->DeleteGlobalRef(aacEldEncoderInstance);
  aacEldEncoderInstance = NULL;
  jniInitialized = false;
}
```

When calling FindClass() for the first time in an application, it should be called from the main thread. If it is not called from the main thread, the Android class loader might not be able to find the class, even though it exists.[9] As a consequence, it is important to ensure that the encoder (and decoder) is at least once created from the main thread of your application.

If the class has been found, it is stored in the member variable aacEncoderClass, ensuring that subsequent encoder objects can be created also from different threads. After JNI has found the class object, the initialization method tries to obtain handles to the class methods that need to be used for encoding: the class constructor, configure(), encode(), and close(). Therefore, the JNI function GetMethodID() is used and care should be taken to provide the correct Java method signatures to the call (the command line utility javap can be used to display Java class method signatures that should be used in JNI calls).

Next, a new object of type AacEldEncoder (Java) is created by a call to the JNI function NewObject(). If the object creation is successful, a new global reference to the just created object is obtained and stored in the member variable aacEldEncoderInstance. All subsequent object calls are then performed using this object reference. For this object a global

Java reference must be created in C++ by calling NewGlobalRef(). Otherwise, the Java garbage collector would immediately collect the newly created object.

After this initial setup, the individual object methods can simply be called from JNI using the Call*Method() calls provided by the JNI API and wrapping the input and output data accordingly (Listing 7).

Accessing the Android AAC-ELD implementation from native code via JNI will cause the Java garbage collector to be run from time to time. This will also happen in an otherwise pure native Android application and should be considered during performance evaluation and optimization of your application.

Interoperability With iOS

The AAC-ELD codec has been available on the iOS platform since version 4.0.10 The Android AAC-ELD implementation is compatible and interoperable with iOS devices. Notably the Android AAC-ELD decoder supports decoding of all audio streams created by the iOS AAC-ELD encoder as long as the correct ASC is used for configuration.

However, the Java MediaCodec API of the Android 4.1 AAC-ELD encoder has still some limitations with respect to the available AAC-ELD encoder configuration parameters, such as available keys in the MediaFormat class. Therefore, audio streams that are encoded with AAC-ELD using Android devices running version 4.1 of the Jelly Bean operating system may support only a limited set of all configurations.

If the same codec configuration for sending and receiving between Android and iOS devices has to be used, the following modes are recommended on Android 4.1 (besides others):

- AAC-ELD without SBR: 22.05-kHz sampling rate, 512-frame length, 32.729-kbit/s+ bitrate
- AAC-ELD without SBR: 44.1-kHz sampling rate, 512-frame length, 65.459-kbit/s+ bitrate

The Java API on Android version 4.2 supports access to extended configuration settings, such as lower bitrates, but not the activation of the Spectral Band Replication (SBR) tool or the selection of different frame sizes. During the evolution of the MediaCodec API more encoder configuration, settings will become available including SBR activation and 480-frame lengths, with future Android operating system updates.

Conclusion

Application developers can access the AAC-ELD audio encoder and decoder that are built into the Android operating system to create a Full-HD Voice capable communication application. Additionally, the Java-only MediaCodec API can be used from native C/C++ code, which enables integration of the Android AAC-ELD codec into existing native communication frameworks to provide a basis for a wide variety of possible implementations.

Being readily available on over 1.3 billion devices running iOS and Android (4.1 or higher), AAC-ELD becomes the ideal foundation for creating a high-quality Full-HD Voice capable communication application delivering a compelling audio experience to the user. Because of its broad availability and the ease of access to this audio coding technology via mobile operating system APIs, AAC-ELD should be the audio codec of choice for your next communication application.

References

1. Markus Schnell et al., “Enhanced MPEG-4 Low Delay AAC - Low Bitrate High Quality Communication,” in Audio Engineering Society Convention, Vienna, 2007.

2. Google Inc. (2012), Android 4.1 Compatibility Definition
3. Google Inc. (2013), Android Developer Website
4. Fraunhofer IIS (2012), Application Bulletin - AAC Transport Formats.
5. Google Inc. (2013), Android Developer Website MediaFormat
6. Khronos Group (2013), OpenSL ES
7. Google Inc. (2013), Android NDK
8. Oracle (2011), Java Native Interface
9. Google Inc. (2013), Android NDK
10. Fraunhofer IIS (2012), Application Bulletin - AAC-ELD based Audio Communication on iOS, A Developer's Guide.

Source URL: http://electronicdesign.com/embedded/understanding-full-hd-voice-communication-android-based-devices