Fraunhofer
IIS

FRAUNHOFER INSTITUTE FOR INTEGRATED CIRCUITS IIS

APPLICATION BULLETIN

AAC-ELD based Audio Communication on Android

V2.8 - 25.07.2014

ABSTRACT

This document is a developer's guide that shows how the AAC-ELD codec included in the Android operating system can be accessed from third party audio communication applications. Thereby, it illustrates how developers can create their own innovative applications and services using the high quality audio codec that is also used by Apple's FaceTime.

This text describes the required processing steps and application programming interface calls needed to implement a real-time audio processing application. This includes the presentation of Android's `AudioRecord` and `AudioTrack` classes that are used to create a generic sound card interface for audio recording and playback as well as a discussion of how-to implement an AAC-ELD encoder and decoder using the `MediaCodec` API. These building blocks are then used to create an application that can record, encode, decode and play back audio frames. Finally, it is shown how the created encoder and decoder classes can be used from native code and libraries using the Java Native Interface APIs. In addition to the source code examples presented within the text, the complete source code of the demo application is available together with this document as a ready to use project.

The scope of the example application is limited for simplicity and does thus not cover transmission over IP or other advanced features such as error concealment or jitter buffer management. The development and integration of these components into a complete Voice over Internet Protocol application is a challenging task but can be simplified through the Fraunhofer Audio Communication Engine, which is described briefly at the end of this document.

1 INTRODUCTION

With the introduction of FaceTime, Apple started a new era of Voice over IP (VoIP) communication. An increasing number of people now use video conferencing on their iPhone, iPad, and Mac devices and enjoy excellent audio and video quality while chatting with friends and family members. One of the enabling components of FaceTime is the MPEG-4 AAC Enhanced Low Delay (AAC-ELD) audio codec, which provides high audio quality at low delay and low bit rates [1]. Particularly interesting for Android developers is the recent inclusion of a fully featured AAC-ELD encoder and decoder by Google into their Android operating system starting with version 4.1 (Jelly Bean). Moreover, OEMs are also required to integrate the audio codec in order to fulfill the Google compatibility definitions [2]. Thereby, it is basically guaranteed that every new Android-compatible device is capable of using AAC-ELD for audio encoding and decoding. This creates an extensive user base for new high-quality communication software on Android devices.

The basic components of a real-time VoIP application are depicted in Fig. 1. At its core there are two audio processing chains running in parallel. Audio data that will be sent over the network is recorded, encoded with an appropriate audio codec such as AAC-ELD and packetized into an RTP packet prior to transmission. Simultaneously, incoming RTP packets are depacketized, decoded, and played back via speakers or headphones. These two processing chains are usually termed the *sender* and the *receiver* and are present at both ends of the communication, i.e. in the local as well as in the remote client [3].

If hands-free operation is desired, Echo Control (EC) as a pre-processing step becomes another important part of the *sender*, while in the *receiver* Jitter Buffer Management (JBM) is an invaluable component that tries to compensate variations in network delay through buffering while keeping the overall playback delay low. Finally, Session Setup, e.g. via the Extensible Messaging and Presence Protocol (XMPP) or the Session Initiation Protocol (SIP), Graphical User Interface (GUI) and other control functionalities are required for a complete system. Because this application bulletin cannot cover all the above-mentioned components and their interaction, we will focus on a simple subset for the main part of this paper in Section 2 and 3. In Section 4 and 5 interoperability to iOS is discussed and information about the demo source code is provided. Finally, in Section 6 we will return to some of the more advanced features in VoIP and point to our SDK that can greatly reduce the development time for VoIP applications employing AAC-ELD.

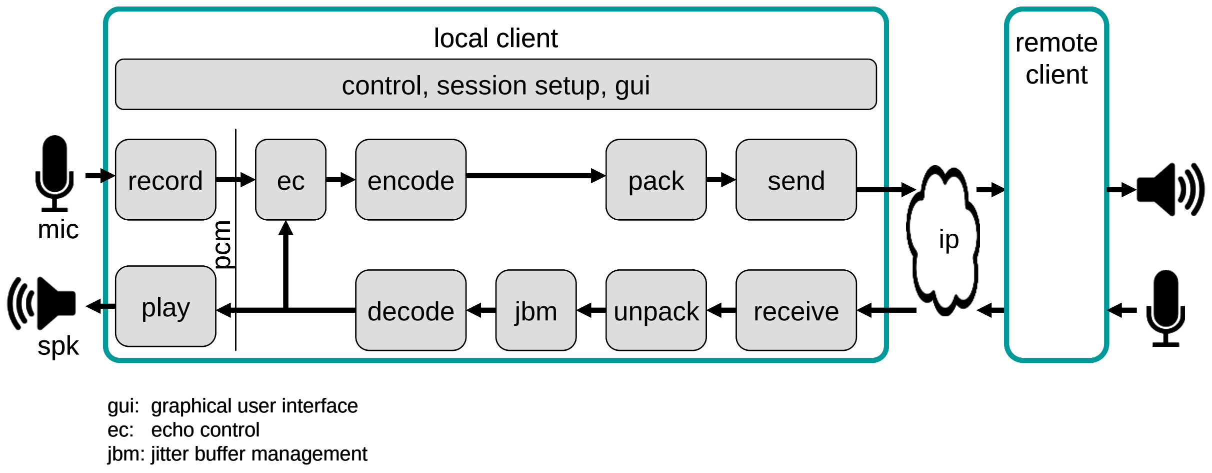

Figure 1: Basic components of a VoIP application.

2 AAC-ELD BASED AUDIO COMMUNICATION ON ANDROID

In the following sections we describe the interfaces of the Android SDK that are required in order to implement high-quality audio recording, AAC-ELD encoding, AAC-ELD decoding and playback on Android based devices. It is assumed that the reader is already familiar with the basic concepts of Android application development and the Java programming language. For section 3, familiarity with the C and the C++ programming languages is assumed. Please note that the provided code samples are tuned for simplicity. Therefore, only a minimum of error checking and abstraction is employed and thus the samples will not fulfill the requirements of production quality code. For in-depth information about the Android platform and its APIs, the reader is referred to [4].

Despite the reduced scope and limitations of the example application, the provided source code can be used as starting point for a full-featured Audio-Over-IP application or as a reference for extending existing projects. The discussion in this application bulletin, however, is limited to audio I/O (recording, playback) and audio coding (encoding, decoding) as depicted in Fig. 2.

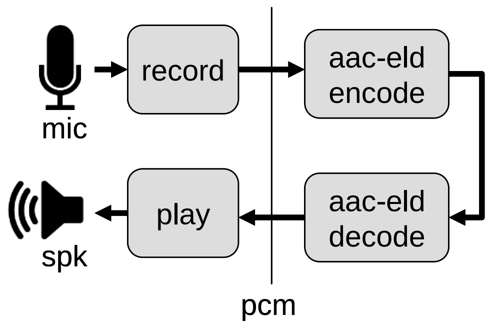

Figure 2: Reduced scope of the demo application.

2.1 AUDIO PROCESSING FOR COMMUNICATION

As explained previously, audio processing for real-time communication applications happens within two individual parts of the application running in parallel. In the *sender* part of the application, audio is recorded and subsequently encoded to a compressed format, while in the *receiver* part of the application incoming packets are decoded and played out. The audio samples for recording and playback are usually in a 16-bit Linear-PCM (LPCM) format. Depending on the number of channels used, a sample can consist of one or more indivdiual values (e.g. one value for mono, two values for stereo and so on). Audio APIs usually deliver a buffer containing a certain number of audio samples at a time. A collection of such associated audio samples is called a *frame*. The exact number of audio samples contained within a frame obtained from the audio hardware is usually implementation dependent but can sometimes be adjusted by an API call. For low-latency communication applications, it is desirable to have the audio hardware return the smallest number of samples possible, i.e. to operate with the lowest possible delay. This is also reflected by the fact that AAC-ELD is designed to operate on frames containing 480 or 512 samples (i.e. the *framesize is 480 or 512*) for a single processing step, which corresponds to about 10 or 11 miliseconds of audio data at a sampling rate of 44.1 kHz. As we will see, the frame sizes delivered by the `AudioRecord` recording API are usually larger due to internal buffering and thus should be split into multiple chunks of the ELD framesize before processing. Note that this internal buffering, of course, adds additional delay to the audio stream.

Although the AAC-ELD encoder that is included in the Android platform supports framesizes of 480 and 512 for encoding, with the current Java API only a fixed framesize of 512 can be used. More configuration options of the encoder may become available as the Java API evolves.

2.2 AUDIO RECORDING AND PLAYBACK: THE SOUNDCARD CLASS

In order to access the sound card for real time audio processing within a Java Android application, the classes `AudioRecord` and `AudioTrack` can be used for recording and playback. For simultaneous recording and playback, the provided demo implementation aggregates individual instances of these classes into a class named `SoundCard` (Listing 1). The `SoundCard` class implements two inner classes named `Player` and `Recorder` that represent two individual threads of processing and use an `AudioTrack` instance and an `AudioRecord` instance to implement their behavior, respectively. Audio samples are delivered by the `Player` instance to a client of the `SoundCard` class by implementing the `ISoundCardHandler` interface. The implementation of these classes is discussed in the next Section.

```java
// Interface that has to be implemented by clients of the SoundCard class
interface ISoundCardHandler {
  public boolean process(byte[] inputBuffer, byte[] outputBuffer);
}

// Simple sound card abstraction using constant parameters for simplicity
public class SoundCard {
  public static final int    sampleRate           = 44100;
  public static final int    nChannels            = 1;
  public static final int    bytesPerSample       = 2; // 16 bit PCM

  private int                 bufferSize           = 0;
  private ISoundCardHandler   soundCardHandler;
  private Recorder            recorder;
  private Player              player;


  // Method will return minimum audio buffer size for recording scaled up to the
  // nearest multiple of 512
  static int recorderBufferSize() { /* ... */ }

  // Method will return minimum audio buffer size for playback scaled up to the
  // nearest multiple of 512
  static int playoutBufferSize() { /* ... */ }

  // Simple audio buffer holding bufferSize samples
  private class AudioBuffer {
    byte[] data = new byte[bufferSize];
  }

  // Audio recorder running as a thread
  private class Recorder extends Thread { /* ... */ }
  private class Player extends Thread { /* ... */ }

  SoundCard(ISoundCardHandler soundCardHandler) {
    this.soundCardHandler = soundCardHandler;
    int recordBufferSize  = recorderBufferSize();
    int playoutBufferSize = playoutBufferSize();

    // Determine audio buffer sizes
    bufferSize            = Math.max(recordBufferSize, playoutBufferSize);
  }

  public void startCallback() {
    // Create and start recorder and player instances
    recorder = new Recorder();
    player   = new Player();
    recorder.start();
    player.start();
  }

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

Listing 1: The `SoundCard` class.

In order to keep the implementation as simple as possible, the audio configuration that is used is fixed. The audio sampling rate is set to 44.1 kHz, which is one of the standard sampling rates that must be supported by any Android device [10] and the channel configuration is mono (i.e. one audio channel). In the `SoundCard` constructor, the minimum buffer sizes that are required for recording and playback are determined using the `AudioTrack.getMinBufferSize()` and `AudioRecord.getMinBufferSize()` API calls providing the fixed audio configuration as parameters. As the AAC-ELD implementation in Android requires a frame size of 512, the obtained minimum buffer sizes are scaled up to a multiple of 512 samples. This is done within the static `SoundCard` class methods `recorderBufferSize()` and `playoutBufferSize()`. The maximum of both buffer sizes is then used to create AudioFrame instances, which is a simple abstraction for native `byte[]` buffers capable of holding a single audio frame at a time. The `startCallback()` method then simply creates a Recorder and a Player instance and starts their respective threads for processing.

2.3 AUDIO RECORDING

Implementing the `Recorder` class is quite straightforward (Listing 2). Internally, an `AudioRecord` instance configured with the fixed audio parameters is used for obtaining recorded audio samples from the hardware. In order to enable synchronization between the recording thread and the player thread, the `Recorder` instance uses an `ArrayBlockingQueue` of `AudioBuffer` instances to store its samples.

```
// Audio recorder running as a thread
private class Recorder extends Thread {
  // New AudioRecord instance using fixed parameters
  private final AudioRecord record  = new
                            AudioRecord(MediaRecorder.AudioSource.VOICE_COMMUNICATION,
                                sampleRate, AudioFormat.CHANNEL_IN_MONO,
                                AudioFormat.ENCODING_PCM_16BIT, bufferSize);

  // Frames will hold successfully recorded audio frames
  private final ArrayBlockingQueue<AudioBuffer> frames
                                                = new ArrayBlockingQueue<AudioBuffer>(1);
    // FramesPool holds a pool of already allocated audio buffer memory
  private final LinkedBlockingQueue<AudioBuffer> framesPool
                                                = new LinkedBlockingQueue<AudioBuffer>();

  public Recorder() {
    // create some audio buffers and pool them
    for (int i = 0; i != 3; ++i)
      framesPool.add(new AudioBuffer());
  }

  @Override
  // Run the recorder thread
  public void run() {
   record.startRecording();

    while (!isInterrupted()) {
     // check if we have already an empty frame in the pool
      AudioBuffer frame = framesPool.poll();
      if (frame == null)
        frame = new AudioBuffer(); // if not, create one
      record.read(frame.data, 0, frame.data.length); // record samples

      try {
        // add recorded frame to the ArrayBlockingQueue
        if (!frames.offer(frame, 20, TimeUnit.MILLISECONDS)) {
          framesPool.add(frame); // if failed, drop samples
        }
      }
      catch (InterruptedException e) {
        interrupt();
      }
    }

    record.stop();
    record.release();
   }
  }
```

Listing 2: Implementation of the Recorder class.

When running, the recording thread continuously reads audio samples from the audio hardware using the `read()` method of the `AudioRecord` instance. The obtained audio data is then stored in the array queue to be read from the corresponding player thread. In order to avoid constant reallocation and deallocation of AudioFrame instances, the Recorder instance additionally keeps a `LinkedBlockingQueue` of `AudioFrame` instances to store unused but already allocated `AudioFrame` buffers.

2.4 AUDIO PLAYBACK

The implementation of the `Player` class is shown in Listing 3. For playback, the `AudioTrack` class is used. When creating an instance of this class, it should be configured with the stream type `AudioManager.STREAM_VOICE_CALL` to indicate phone call stream usage. Additionally, the play out mode must be set to `AudioTrack.MODE_STREAM` in order to indicate to the system that the audio data is not provided fully before playback start, but rather is constantly added to the stream during processing. Once the player thread is started, it continuously polls the `ArrayBlockingQueue` created in the `Recorder` instance for recorded audio frames. After a recorded `AudioFrame` has been obtained, the process() method of the client class (i.e. the class implementing the ISoundCardHandler interface) is called, passing the recorded frame as argument for processing. When this call returns, the client should have filled the outputBuffer byte buffer with the samples that should be played back. These samples are then played out with a call to the write() method of the AudioTrack instance.

```
private class Player extends Thread {
  // New AudioTrack instance using fixed parameters
  private final AudioTrack player
          = new AudioTrack(AudioManager.STREAM_VOICE_CALL, sampleRate,
                         AudioFormat.CHANNEL_OUT_MONO,
                         AudioFormat.ENCODING_PCM_16BIT,
                         bufferSize, AudioTrack.MODE_STREAM);

  // Buffer where output samples will be stored by the client
  private final byte[] outputBuffer = new byte[bufferSize];

  public Player() {}

  @Override
  public void run() {
    player.play();

    while (!isInterrupted()) {
      AudioBuffer inputSamples;
      try {
        // Try to get recorded samples from the ArrayBlockingQueue
        inputSamples = recorder.frames.take();
      } catch (InterruptedException e) {
        interrupt();
        continue;
      }

      // Provide recorded samples to the client and receive output samples
      if (!soundCardHandler.process(inputSamples.data, outputBuffer))
        interrupt();
      // Free the input sample frame
      recorder.framesPool.add(inputSamples);

      // Playback audio
      player.write(outputBuffer, 0, outputBuffer.length);
    }

    player.stop();
    player.release();
    }
  }
```

Listing 3: Implementation of the Player class.

The process() method that has to be implemented by the client that is using the SoundCard class is where the audio samples are encoded to AAC-ELD and subsequently decoded to LPCM again. Before the implementation of this method is shown, we need to discuss the implementation of the AAC-ELD encoder and decoder classes, respectively.

2.5 AAC-ELD ENCODING AND DECODING

Knowing how to access the AAC-ELD encoder and decoder is one of the crucial parts when developing an audio communication application. In Android, AAC-ELD coding is accessible through instances of the MediaCodec class that is contained within the android.media package. Based on this class, we create 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 re-use of the encoder and decoder classes when accessing them via the JNI interface later. The class definitions of the AacEldEncoder and AacEldDecoder classes are given in Listing 4. As can be seen, basically only two 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. Additionally, a close() method is provided for resource cleanup.

```
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() { /* ... */  }
}
```

Listing 4: Class definitions of the AacEldEncoder and AacEldDecoder classes.

Despite their different purposes, the encoder and the decoder instances are configured in a quite similar fashion. In Summary, the steps that must be performed to successfully configure `MediaCodec` instances for AAC-ELD encoding or decoding are:

1. Create an instance of the class `android.media.MediaFormat` that will describe the audio format by specifying a 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 contained within an *audio specific configuration* (ASC) that is generated by a standard conforming AAC-ELD encoder [6]. Therefore, these values can be set to 0 during decoder configuration.
2. In order to configure additional parameters of the audio encoder, one can use the method `setInteger(key, value)` of the just created `MediaFormat` instance. The individual configuration parameters are specified as key-value pairs [5]. For enabling 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`. For initialization of the decoder, neither the profile level key nor the bitrate key have to be set as these parameters are also contained 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*).
3. Subsequently, 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. **Note**: 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!

4. Configure the codec with the previously created `MediaFormat` instance by calling the method `configure()` on the codec instance. In order to configure a `MediaCodec` instance for encoding, additionally the constant `MediaCodec.CONFIGURE_FLAG_ENCODE` has to be passed as the last parameter to the `configure()` call.
5. 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 using the `MediaCodec` API is buffer based and works on internal audio buffers provided by the encoder wrapper. Therefore, an input buffer index has to be requested first by calling the method `dequeueInputBuffer(timeout)` on the encoder instance. In the demo application, a timeout value of 0 is used, meaning that the call will return immediately in order to avoid thread blocking. If a valid buffer is returned (index value >= 0), that buffer can be accessed by calling `getInputBuffers()` and indexing the returned array of `ByteBuffer` objects. The LPCM data that should be encoded can now 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.

In order 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 (i.e. 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 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 contains an ASC.

The complete implementation of the `configure()` and `encode()` methods of the `AacEldEncoder` class is given in Listing 5. 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, i.e. there might be additional delay before output buffers are available. Since the presented implementation requires that data be returned immediately, however, a very simple form of output buffer polling is implemented. Thereby, the `dequeueOutputBuffer()` method is called repeatedly until a valid output buffer index is obtained. In order to prevent application blocking, the number of tries is limited before a `null` object is returned to signal an error.

```
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;
    }
  }
}
```

Listing 5: Configuration and encoding with the AacEldEncoder class.

Decoding of encoded AUs into LPCM frames is done accordingly, as the same API is used for this purpose. The implementation is shown in Listing 7. 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.

```
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() { /* ... */  }
}
```

Listing 7: Configuration and decoding with the AacEldDecoder class.

2.6 BRINGING IT ALL TOGETHER: THE MAIN ACTIVITY

The `MainActivity` class is the entry point of the presented demo and thus the place where the `SoundCard` and the `AacEldEncoder` and `AacEldDecoder` classes are combined in order to make up for a working sample application (Listing 7). Since a basic familiarity of the reader with Android application programming is assumed, only the details relevant to this application bulletin are discussed.

```java
// The main activity for the demo application
public class MainActivity extends Activity implements ISoundCardHandler, View.OnClickListener {
    // The simple sound card implementation
    private SoundCard            m_soundCard;

    // The Java interface to the encoder and the decoder
    private AacEldEncoder        m_encoder;
    private AacEldDecoder        m_decoder;

    // For demo purposes, we use a fixed sampling rate, a mono channel configuration
    // and an encoder bitrate of 65 kbps
    static private final int     encoderBitrate   = 65000;
    static private final int     frameSize        = 512;

    /* ... */

    @Override
    public void onClick(View v) {
        /* ... */
        m_encoder = new AacEldEncoder();
        m_decoder = new AacEldDecoder();
        m_decoder.configure(
        m_encoder.configure(SoundCard.sampleRate,
                            SoundCard.nChannels,
                            MainActivity.encoderBitrate));

        m_soundCard = new SoundCard(this);
        m_soundCard.startCallback();
        /* ... */
    }

    // Processes audio frames of 512 samples each
    private void processFrame(byte[] inFrame, byte[] outFrame) {

        byte[] outAU  = m_encoder.encode(inFrame);
        byte[] tmpOut = m_decoder.decode(outAU);

        // Ensure that a valid sample buffer is returned even
        // if decoding fails
        if (tmpOut == null) java.util.Arrays.fill(outFrame, (byte) 0);
        else System.arraycopy(tmpOut, 0, outFrame, 0, outFrame.length)
    }

    @Override
    public boolean process(byte[] inputBuffer, byte[] outputBuffer) {

      // We need to split the input buffer into framesize-sized frames
      // because ELD requires frames containing 512 samples per packet
      if (inputBuffer.length != outputBuffer.length || inputBuffer.length % frameSize != 0)
        throw new RuntimeException("Error! Cannot handle bogus sized audio buffers in this demo!");

      final int nChunks
           = inputBuffer.length / (frameSize*SoundCard.nChannels*SoundCard.bytesPerSample);

      byte[] inBuf  = new byte[frameSize*SoundCard.nChannels*SoundCard.bytesPerSample];
      byte[] outBuf = new byte[frameSize*SoundCard.nChannels*SoundCard.bytesPerSample];

      for (int i = 0; i < nChunks; ++i) {
        System.arraycopy(inputBuffer,
                         i*frameSize*SoundCard.nChannels*SoundCard.bytesPerSample,
                         inBuf, 0, inBuf.length);

        processFrame(inBuf, outBuf);

        System.arraycopy(outBuf, 0,
                         outputBuffer,
                         i*frameSize*SoundCard.nChannels*SoundCard.bytesPerSample, outBuf.length);
      }
      return true;
    }
}
```

Listing 8: Parts of the `MainActivity` class.

As can be seen from Listing 8, the `MainActivity` class aggregates the `SoundCard` and the `AacEldEncoder` and `AacEldDecoder` classes. Moreover, it implements the `ISoundCardHandler` interface which means that the `process()` method for audio processing is implemented here. The very simple GUI to this application shows a single button and touches on this button will be handled within the `MainActivities onClick()` method. Here, new instances of the encoder and decoder classes are created. The encoder is initialized with the current sampling rate, the number of channels and the encoder bitrate, which is set to 65 kilobits per second (kbps) in the demo. Subsequently, the decoder is initialized with the return value of the encoders configure() method, i.e. the ASC created by the encoder. Finally, an instance of the `SoundCard` class is created and the `startCallback()` method is invoked. As a result, audio processing starts and the `process()` method will be called by the `SoundCard` object whenever there is audio data available. As mentioned earlier, the audio buffers delivered by the hardware are usually larger than the framesize required for AAC-ELD encoding and decoding. Therefore, they are split into individual frames of the required framesize each (here 512 samples per frame) before processing. These frames are then passed to the method `processFrame()`, where they are encoded and subsequently decoded again. Note that the decoder will require up to two AUs before it can start the actual decoding of audio data (this is called *priming*). Therefore it is ensured that the processFrame method returns valid audio samples even when the decoder returns no data (silence).

3 AAC-ELD CODING USING THE JAVA NATIVE INTERFACE

When implementing a fully featured VoIP application, it is often desirable to perform most of the required audio processing in native code and resort back to the Java environment only for providing the user interface. While accessing the device sound card from native code is quite easily accomplished 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 [11, 13]. As a consequence, native applications or application libraries that want to use the AAC-ELD codec built into the Android OS 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 [14]. In the following, we will show how the encoder and decoder implementations from the previous sections can be accessed from native C++ code running within an Android application environment. Thereby, the discussion will focus on how to call Java from native code and thus it is assumed that the reader is already familiar with using JNI for calling native code from Java.

3.1 NATIVE IMPLEMENTATION PREREQUISITES

In order to modify the presented example such that the audio encoding and decoding is performed in C++ code, a new Java class called `NativeAudioProcessor` with methods `init()`, `process()` and `close()` is introduced (Listing 9). This class acts as an entry point to the native implementation that is presented next and its implementation is shown in Listing 10. Having defined this class, the `processFrame()` method of the `MainActivity` class (Listing 8) will be modified to call the `process()` method of an `NativeAudioProcessor` instance, which in turn will call the process method of a C++ class named `AudioProcessor` that is meant as a placeholder representing any other native audio processing library that requires AAC-ELD coding (Listing 11).

During the `init()` method of the `NativeAudioProcessor` instance, an object of type `AudioProcessor` is created. Additionally, memory buffers that can hold a single audio frame for input and output audio are allocated. Because these buffers are required to exist across multiple calls to `process()`, pointers to the native buffers are stored as the private class fields `InputAudioFrameHandle` and `OutputAudioFrameHandle`. Likewise, a pointer to the `AudioProcessor` instance is stored in a field of the `NativeAudioProcessor` instance in order to ensure that it will be accessible during the lifetime of the applications processing loop. Note that because Java does not know pointer types, it is required to store the associated pointer values in a Java data type that is most likely to be able to hold the value of a pointer. Hence, the data type `long` is used for this purpose, as the `long` type should be capable of holding 64 bit values on current platforms. For audio data processing, the `process()` method then only copies the data passed as Java `byte[]` buffers via JNI to the previously allocated `std::vector` buffers and calls the `process()` method of the `AudioProcessor` class.

```
public class NativeAudioProcessor {

  // Load required native libraries when
  // the first instance of this class is
  // created
  static {
    System.loadLibrary("gnustl_shared");
    System.loadLibrary("android_aaceld");
  }

  // Private variables storing pointers
  // to native object instances
  private long AudioProcessorHandle;
  private long InputAudioFrameHandle;
  private long OutputAudioFrameHandle;

  // The native methods that take care of interacting with the native
  // audio processing part of the demo project
  native boolean init(int sampleRate,
                      int nChannels,
                         int bitrate,
                      int framesize,
                      int noOfBytesPerSample);
  native boolean process(byte[] inSamples, byte[] outSamples);
  native boolean close();
}
```

Listing 9: Class NativeAudioProcessor.

```
static jfieldID AUDIO_PROCESSOR_FIELDID    = NULL;
static jfieldID AUDIO_INPUT_FRAME_FIELDID  = NULL;
static jfieldID AUDIO_OUTPUT_FRAME_FIELDID = NULL;

JNIEXPORT jboolean JNICALL Java_de_fraunhofer_iis_android_1aaceld_NativeAudioProcessor_init
  (JNIEnv *env, jobject obj, jint sampleRate, jint nChannels,
   jint bitrate, jint framesize, jint noOfBytesPerSample) {

  // Get field IDs that will store handles to the AudioProcessor, in and out frame classes
  jclass nativeAudioProcessorClass = env->GetObjectClass(obj);
  AUDIO_PROCESSOR_FIELDID = env->GetFieldID(nativeAudioProcessorClass, "AudioProcessorHandle", "J");
  AUDIO_INPUT_FRAME_FIELDID = env->GetFieldID(nativeAudioProcessorClass, "InputAudioFrameHandle", "J");
  AUDIO_OUTPUT_FRAME_FIELDID = env->GetFieldID(nativeAudioProcessorClass, "OutputAudioFrameHandle", "J");

  JavaVM *vm = NULL;
  env->GetJavaVM(&vm);

  AudioProcessor *p = new AudioProcessor(sampleRate, nChannels, bitrate, framesize, noOfBytesPerSample, (void*)vm);

  const int audioFrameByteSize         = framesize * nChannels * noOfBytesPerSample;
  std::vector<unsigned char> *inFrame  = new std::vector<unsigned char>(audioFrameByteSize, 0);
  std::vector<unsigned char> *outFrame = new std::vector<unsigned char>(audioFrameByteSize, 0);

  env->SetLongField(obj, AUDIO_PROCESSOR_FIELDID, (jlong)p);
  env->SetLongField(obj, AUDIO_INPUT_FRAME_FIELDID, (jlong)inFrame);
  env->SetLongField(obj, AUDIO_OUTPUT_FRAME_FIELDID, (jlong)outFrame);
}

JNIEXPORT jboolean JNICALL Java_de_fraunhofer_iis_android_1aaceld_NativeAudioProcessor_process
  (JNIEnv *env, jobject obj, jbyteArray inSamples, jbyteArray outSamples) {

  AudioProcessor *p    = (AudioProcessor*)env->GetLongField(obj, AUDIO_PROCESSOR_FIELDID);
  std::vector<unsigned char> *inFrame  = (std::vector<unsigned char>*)env->GetLongField(obj,
AUDIO_INPUT_FRAME_FIELDID);
  std::vector<unsigned char> *outFrame = (std::vector<unsigned char>*)env->GetLongField(obj,
AUDIO_OUTPUT_FRAME_FIELDID);

  jbyte *inBytes = env->GetByteArrayElements(inSamples, 0);
  memcpy(&(*inFrame)[0], inBytes, inFrame->size());
  env->ReleaseByteArrayElements(inSamples, inBytes, 0);

    p->process(*inFrame, *outFrame);

  jbyte *outBytes = env->GetByteArrayElements(outSamples, 0);
  memcpy(outBytes, &(*outFrame)[0], outFrame->size());
  env->ReleaseByteArrayElements(outSamples, outBytes, 0);

  return true;
}

JNIEXPORT jboolean JNICALL Java_de_fraunhofer_iis_android_1aaceld_NativeAudioProcessor_close
  (JNIEnv *env, jobject obj) {

  AudioProcessor *p    = (AudioProcessor*)env->GetLongField(obj, AUDIO_PROCESSOR_FIELDID);
  std::vector<unsigned char> *inFrame  = (std::vector<unsigned char>*)env->GetLongField(obj,
AUDIO_INPUT_FRAME_FIELDID);
  std::vector<unsigned char> *outFrame = (std::vector<unsigned char>*)env->GetLongField(obj,
AUDIO_OUTPUT_FRAME_FIELDID);
  delete p;
  delete inFrame;
  delete outFrame;
  env->SetLongField(obj, AUDIO_PROCESSOR_FIELDID, (jlong)0);
  env->SetLongField(obj, AUDIO_INPUT_FRAME_FIELDID, (jlong)0);
  env->SetLongField(obj, AUDIO_OUTPUT_FRAME_FIELDID, (jlong)0);

  return true;
}
```

Listing 10: C++ implementation of the NativeAudioProcessor class.

The implementation of the native `AudioProcessor` class shown in Listing 11 is very straightforward. It contains two yet to be defined classes `AacEldEncoder` and `AacEldDecoder` that will interface with their Java counterparts from Section 2 in order to perform audio coding. Besides that, the `process()` method is as simple as its Java implementation: encode the input samples to an AU and subsequently decode that AU for playback. The only notable difference is that the constructor of the `AudioProcessor` class as well as that of the `AacEldEncoder` and `AacEldDecoder` classes receive an additional parameter named `jvmHandle` of type `void*`.

```
#ifndef __AUDIOPROCESSOR_H__
#define __AUDIOPROCESSOR_H__

#include <vector>

#include "AacEldEncoder.h"
#include "AacEldDecoder.h"

class AudioProcessor {
public:
  AudioProcessor(int sampleRate,
                 int nChannels,
                 int bitrate,
                 int framesize,
                 int noOfBytesPerSample,
                 void *jvmHandle);
  ~AudioProcessor();

   void process(std::vector<unsigned char>& inSamples, std::vector<unsigned char>& outSamples);

private:

  AacEldEncoder m_encoder;
  AacEldDecoder m_decoder;

  unsigned int sampleRate;
  unsigned int nChannels;
  unsigned int bitrate;
  unsigned int frameSize;
  unsigned int noOfBytesPerSample;
};

#endif /* __AUDIOPROCESSOR_H__ */

// Implementation

#include "AudioProcessor.h"
#include <stdlib.h>

AudioProcessor::AudioProcessor(int sampleRate,
                                   int nChannels,
                               int bitrate,
                               int framesize,
                               int noOfBytesPerSample,
                               void *jvmHandle)
  : m_encoder(jvmHandle), // <- needs a handle to the JavaVM
    m_decoder(jvmHandle), // <- needs a handle to the JavaVM
    sampleRate(sampleRate),
    nChannels(nChannels),
    bitrate(bitrate),
    frameSize(framesize),
    noOfBytesPerSample(noOfBytesPerSample) {

  std::vector<unsigned char> asc;
  m_encoder.configure(sampleRate, nChannels, bitrate, asc);

  // Use the ASC to initialize the AAC-ELD decoder
  m_decoder.configure(asc);
}

AudioProcessor::~AudioProcessor() {
  m_encoder.close();
  m_decoder.close();
}

void AudioProcessor::process(std::vector<unsigned char>& inSamples,
                             std::vector<unsigned char>& outSamples) {

  // 1. Encode the input samples to an encoded access unit (AU)
  std::vector<unsigned char> encodedAU;
  m_encoder.encode(inSamples, encodedAU);

  // 2. Decode the just encoded AU to PCM samples again for playback
  outSamples.resize(frameSize*nChannels*noOfBytesPerSample);
  m_decoder.decode(encodedAU, outSamples);
}
```

Listing 11: C++ class AudioProcessor declaration and definition.

This parameter is actually of the type `JavaVM*` as defined by the JNI API but cast to a `void*` in order to enforce a loose coupling between the native implementation and the JNI, which is often desirable. However, note that this parameter is absolutely required, because later on it will be used to obtain an associated `JNIEnv` pointer that is needed in order to perform JNI calls. We cannot use the `JNIEnv` pointer that is passed to one of the methods of the `NativeAudioProcessor` instance directly, because that pointer is valid only within the context of the current JNI call (e.g. when `NativeAudioProcessor.init()` returns, the associated `JNIEnv*` would become invalid).

Moreover, even though the JNI specification allows to initialize and create a new `JavaVM` instance through the JNI API, the Android implementation allows at most one `JavaVM` instance per process. As a consequence, it is necessary to obtain a pointer to the `JavaVM` from a valid `JNIEnv*` during the `init()` method of the `NativeAudioProcessor` instance and propagate that pointer to all objects that need it later on (marked bold in Listing 10).

3.2 ACCESSING THE AAC-ELD ENCODER AND DECODER FROM C++

In order to access the Java `AacEldEncoder` and `AacEldDecoder` classes from native code, we define a similar interface in C++ that can be used for encoding and decoding of audio data (Listing 12). Note that these classes make use of the "pointer-to-implementation"-idiom in an effort to hide the JNI part of the implementation from the user. Because the implementation of the C++ `AacEldEncoder` and `AacEldDecoder` classes are almost identical, in the following only the implementation of the `AacEldEncoder` will be discussed.

```
#include <vector>

// 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_;
};
```

Listing 12: C++ declaration of the AacEldEncoder and AacEldDecoder classes.

Before the detailed implementation is shown, we also introduce a small utility class named `JniEnvGuard` in Listing 13. As explained in the previous section, the C++ encoder and decoder implementation receives a pointer to a `JavaVM` type in order 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 thread and cannot be shared between individual threads, because the `JNIEnv` is also used for thread-local storage [12]. 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 thus result in an error and at worst crash your program. Moreover, the Android OS will not detach native threads automatically when they are terminating and thus any application attaching a JNIEnv to its current native thread is required to detach itself before the main thread terminates.

```
#ifndef __JNI_ENV_GUARD_H__
#define __JNI_ENV_GUARD_H__

#include <jni.h>

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 <stdexcept>

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

Listing 13: The JniEnvGuard class.

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, i.e. the JniEnvGuard class implements the RAII-idiom for JNIEnv thread attachment and detachment.

Finally, the implementation of the C++ AacEldEncoder is shown in Listings 14 and 15. The most important method of the implementation is initJni(). In this method, the connection between the Java AacEldEncoder implementation and the C++ implementation is set up. Firstly, the AacEldEncoder Java class is looked up by a call to FindClass(). The argument to FindClass() is a string containing 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".

```
#include <jni.h>
#include <android/log.h>

#include <stdlib.h>

#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, "<init>", "()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;
}
```

Listing 14: Initialization of the AacEldEncoder C++ 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;
}
```

Listing 15: Implementation of the AacEldEncoder C++ class.

Note, that 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 [12]. 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 the class object has been found by JNI, 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. Note that it is required to create a new global reference, because otherwise the Java garbage collector would collect the newly created object.

After this initial setup has taken place, 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 as illustrated in Listing 15.

4 INTEROPBERABILITY TO IOS

Since iOS version 4.0, the AAC-ELD codec is also available on Apple's iOS platform [7]. The Android AAC-ELD implementation is compatible with and interoperable to the implementation on 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 (i.e. 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
  - Sampling rate 22.05kHz
  - Frame length 512
  - Bitrate: 32.729 kbps and higher

- AAC-ELD without SBR
  - Sampling rate 44.1 kHz
  - Frame length 512
  - Bitrate: 65.459 kbps and higher

The Java API on Android version 4.2 supports access to extended configuration settings, e.g. lower bitrates, but not yet the activation of the *Spectral Band Replication* tool (SBR) and the selection of different framesizes. It is expected that during the evolution of the MediaCodec API more encoder configuration settings will become available (e.g. activation of SBR, framelength of 480, etc.) with future Android operating system updates.

5 DEMO APPLICATION SOURCE CODE

The listings used during the text are extracted from a working application that performs real-time audio I/O and coding with AAC-ELD on Android using the Java SDK and the Java Native Interface APIs. In order to keep the text size appropriate, not every detail of the code could be discussed. However, the complete source is available together with this document and can be downloaded from the same source. The implementation should be simple enough to be understandable without excessive effort. After working through the source code on paper and understanding the underlying concepts, it is recommended to open the project and run it on an Android device. By changing the audio processing parameters (e.g. the bitrate) a developer should soon feel familiar with the code and be prepared to start own projects using AAC-ELD for audio communication.

The demo source package can be imported into an Eclipse workspace as an Android project for building the sample application. The Android Native Development Tools package for Eclipse as well as the Android NDK are required. Note that because the demo application requires the Android NDK, it currently cannot be build using Android Studio. The application itself is just a bare bones demonstration of the concepts that are presented in this paper and the program code is optimized for readability and instructional purposes. Consequently, error checking and abstraction is greatly reduced in favor of understandability. Additionally, most parameters that should be dynamic within a real world application are hardcoded and assumed to be constant for the same purpose. Please refer to the README.txt file from the package for further information about the demo project.

6 AUDIO COMMUNICATION ENGINE

As has been demonstrated in this paper, the Android operating system starting with version 4.1 provides the basic components for implementing high-quality audio communication based on AAC-ELD. Especially because of the native support in Android, using AAC-ELD for communication applications becomes very attractive from a business and developer perspective. However, it should be clear that having a high-quality audio codec in place is just the basic foundation of a full VoIP system. Referring to the introduction and Fig.1, it becomes obvious that several other components have to be implemented and various other problems are to be addressed. Besides echo control and IP/UDP/RTP packetization, this also includes the handling of lost and delayed packets in the Jitter Buffer Management. With realistic network conditions, IP packets may get lost or may undergo a variable delay. The VoIP system must react to this by implementing error concealment and Time Scale Modification (TSM) for adapting the play out time during a call. These operations are not supported by the Android API and need to be implemented on top of the provided services. From this paper it should also become clear that the correct usage of the Android Java API could become challenging when real-time constraints and multiple threads of processing come into play. Therefore, the software development process of a fully enabled VoIP application can still be time consuming and cumbersome.

As time-to-market is key for many companies and application developers, Fraunhofer IIS has developed the Audio Communication Engine (ACE) and ported it to Android. The ACE provides the missing components for building a VoIP application based on AAC-ELD and offers the complete functionality through a higher-level API that is easy to use. As illustrated in Fig. 4, the ACE covers all components of the main audio processing chain but leaves the user interface and session setup to the application developer. Additionally, as a C++ library, the ACE allows for an easy integration of existing components and facilitates cross-platform development of audio communication applications on a wide range of devices and operating systems. For example, it can easily be combined with a SIP-client or other session setup protocols to allow for maximum freedom in service offering.

For further information on the ACE, please visit the Fraunhofer IIS home page [8,9].

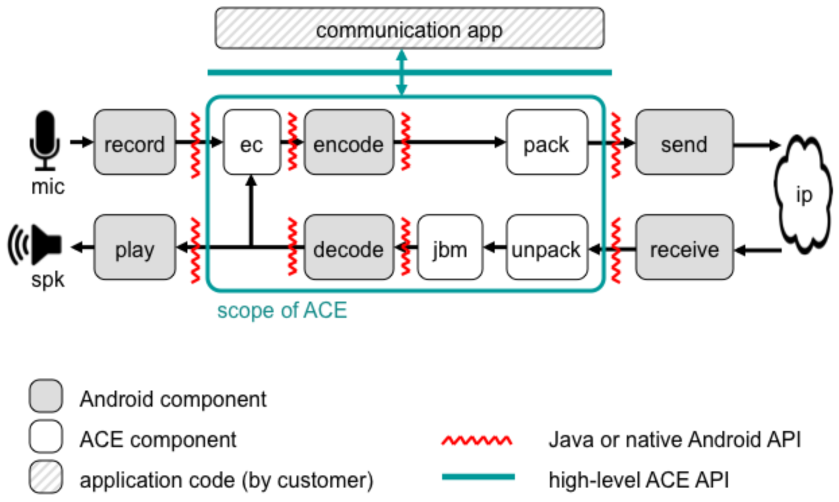

Figure 4: Scope and benefit of using the ACE for building a complete VoIP app on Android

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 [Online]. http://source.android.com/compatibility/4.1/android-4.1-cdd.pdf

[3] Colin Perkins, RTP - Audio and Video for the Internet, 1st ed. Boston, MA: Addison-Wesley, 2003.

[4] Google Inc. (2013) Android Developer Website [Online]. http://developer.android.com/index.html

[5] Google Inc. (2013) Android Developer Website MediaFormat [Online].

http://developer.android.com/reference/android/media/MediaFormat.html

[6] Fraunhofer IIS. (2012) Application Bulletin - AAC Transport Formats. [Online]. http://www.iis.fraunhofer.de/content/dam/iis/en/dokumente/AMM/AAC_Transport_FormatsR1.0-1.pdf

[7] Fraunhofer IIS. (2012) Application Bulletin - AAC-ELD based Audio Communication on iOS, A Developers Guide. [Online]. http://www.full-hd-voice.com/content/dam/fullhdvoice/documents/iOS-ACE-AP-v2.pdf

[8] Fraunhofer IIS. (2010, Jan.) Fraunhofer IIS Audio Communication Engine - Raising the Bar in Communication Quality. [Online]. http://www.iis.fraunhofer.de/bf/amm/download/whitepapers/WP_Audio_Communication_Engine.pdf

[9] Fraunhofer IIS. (2010, Jan.) Fraunhofer IIS Audio Communication Engine [Online]. http://www.iis.fraunhofer.de/en/bf/amm/produkte/kommunikation/ace/index.js

[10] Google Inc. (2013) Android Supported Media Formats [Online]. http://developer.android.com/guide/appendix/media-formats.html

[11] Khronos Group. (2013) OpenSL ES [Online]. http://www.khronos.org/opensles/

[12] Google Inc. (2013) Android JNI Tips [Online]. http://developer.android.com/training/articles/perf-jni.html

[13] Google Inc. (2013) Android NDK [Online]. http://developer.android.com/tools/sdk/ndk/index.html

[14] Oracle. (2011) Java Native Interface [Online]. http://docs.oracle.com/javase/6/docs/technotes/guides/jni/

ABOUT FRAUNHOFER IIS

The Audio and Media Technologies division of Fraunhofer IIS has been an authority in its field for more than 25 years, starting with the creation of mp3 and co-development of AAC formats. Today, there are more than 10 billion licensed products worldwide with Fraunhofer's media technologies, and over one billion new products added every year. Besides the global successes mp3 and AAC, the Fraunhofer technologies that improve consumers' audio experiences include Cingo® (spatial VR audio), Symphoria® (automotive 3D audio), xHE-AAC (adaptive streaming and digital radio), the 3GPP EVS VoLTE codec (crystal clear telephone calls), and the interactive and immersive MPEG-H TV Audio System.

With the test plan for the Digital Cinema Initiative and the recognized software suite easyDCP, Fraunhofer IIS significantly pushed the digitization of cinema. The most recent technological achievement for moving pictures is Realception®, a tool for light-field data processing.

Fraunhofer IIS, based in Erlangen, Germany, is one of 69 divisions of Fraunhofer-Gesellschaft, Europe's largest application-oriented research organization.

For more information, contact amm-info@iis.fraunhofer.de, or visit www.iis.fraunhofer.de/amm.