Fraunhofer
IIS

FRAUNHOFER INSTITUTE FOR INTEGRATED CIRCUITS IIS

APPLICATION BULLETIN

AAC-ELD based Audio Communication on iOS
A Developer's Guide

V2.3 - 08.08.2012

ABSTRACT

This document is a developer's guide for accessing the AAC-ELD codec included in iOS from third party audio communication applications. It shows how developers can create their own innovative applications and services using the same high quality codec as in FaceTime.

Source code examples are used to illustrate the required processing steps and Application Programming Interface calls. The complete source code is available together with this paper and comprises a working demo application, which reads audio frames from the microphone and plays them back at low delay after encoding and subsequent decoding. The discussed iOS APIs include full-duplex audio Input/Output using the *Remote I/O AudioUnit* and AAC-ELD encoding/decoding using the *AudioConverter* API. The initialization of components and underlying concepts, such as the usage of callback functions and object properties are discussed.

The scope of the example application is limited for simplicity and does not cover transmission over IP or other advanced features such as error concealment or jitter buffer management. The integration of all 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 the 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 AAC Enhanced Low Delay (AAC-ELD) audio codec, which provides high audio quality at low delay and low bit rates [1]. What makes this codec particularly interesting for iOS developers is that it is accessible through the iOS Application Programming Interface (API). Hence, besides FaceTime, other application can make use of this codec.

Using the AAC-ELD codec and real-time audio Input/Output (I/O) is not as straight forward as using other APIs in iOS. This is mainly because the required functionality is accessible only by lower-level APIs. Consequently, their use is not as convenient as using the very elegant and high-level Cocoa interfaces which makes working with these APIs difficult at times. Moreover, the documentation describing the low-level audio API is tailored towards experienced developers. For example, the code samples in the iOS SDK targeting audio coding mainly cover simple audio file conversion for which additional convenience functions exist within the API. Audio over IP on the other hand requires a frame-based processing in order to achieve low delay. This application bulletin offers support for developers by collecting the required information for frame-based audio processing, encoding and decoding in a single place and explaining the underlying concepts of the Core Audio services that are provided by the iOS Software Development Kit (SDK) [2]. An example application is used to illustrate the required processing steps and API calls. The complete source code is available together with this document and allows building a working application with real-time audio I/O and AAC-ELD encoding/decoding.

The basic components of a real time VoIP application are depicted in Fig. 1. The core parts of the application are two processing chains. Firstly, there's the recording, encoding and packing of the audio data that should be transmitted. Simultaneously, unpacking, decoding and playback of received data packets have to be executed. These processing chains are usually termed the *sender* and the *receiver* and are duplicated in the local and remote client [3]. If hands-free operation is desired, an important part of the sender is the Echo Control (EC), which is a pre-processing step before encoding. In the receiver, Jitter Buffer Management (JBM) is an important component, which tries to compensate variations in network delay through buffering while keeping the overall 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. In Section 3 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.

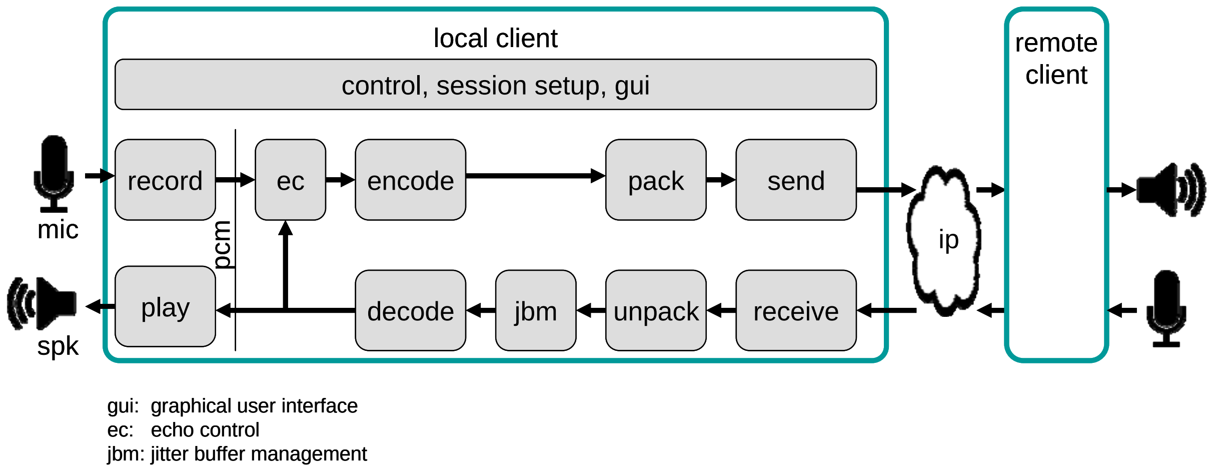

Figure 1: Basic components of a VoIP application

2 IMPLEMENTATION ON iOS

In the following sections we describe the interfaces of the iOS SDK that are required in order to implement high-quality, low-latency audio recording, AAC-ELD encoding, AAC-ELD decoding and playback on iOS based devices. It is assumed that the reader is already familiar with the basic concepts of iOS application programming and the C or C++ programming language. Since the interfaces and APIs that are presented are C-based, no knowledge of Objective-C will be required in order to understand the code samples. Please note that the provided example is 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 iOS platform and its APIs, the reader is referred to [4].

However, the code samples presented during the course of the text canbe used in a real world application. Their scope is limited to audio I/O (recording, playback) and coding (encoding, decoding) as depicted in Fig. 2. The resulting demo application can be used as starting point for a full-featured Audio-Over-IP application or as a reference for extending existing projects.

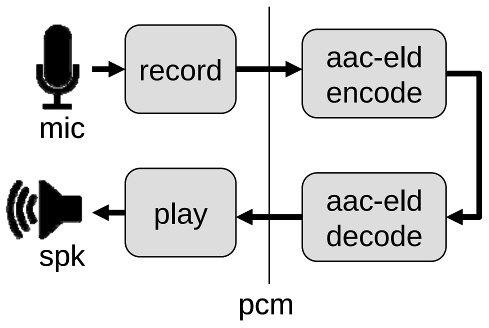

Figure 2: Reduced scope of demo application

Fig. 3 shows the basic iOS components and API calls which are required to implement the demo application. Each component is detailed below and the API calls are explained along with example code. The purpose of Fig. 3 is to provide a high-level overview as guidance throughout the remaining text. The important iOS components are illustrated on the left side of Fig. 3. and include the *AudioUnit* for real-time audio I/O and the *AudioConverter* for encoding and decoding of AAC-ELD. A throughout description of these components is given in Sections 2.2 through 2.5.

A critical concept to understand when using the Core Audio APIs is that of callback functions. These are functions for which the API provides the prototype declaration, such that input- and output parameters as well as return types are pre-defined. However, the developer has to provide the actual implementation for the required functionality on behalf of the iOS component. After registering the callback function by passing a function pointer to the iOS component, it will be called as needed by the component.

The main processing loop in the demo application is defined by the `audioUnitRenderCallback()` function, which is called by the *AudioUnit* whenever new audio samples need to be provided for play-out. When using full-duplex audio I/O, this callback is also the place where audio samples will be acquired from the capture device. This enables simultaneous recording and playback of audio samples. Moreover, in the sample application AAC-ELD encoding and decoding is performed within that function. As a consequence, two additional callback functions are triggered by the *AudioConverter* for AAC-ELD encoding and decoding. Those callback functions are called `encodeProc()` and `decodeProc()`. Their main purpose is to set pointers in order to transfer data to and from the AAC-ELD encoder and decoder within the two respective *AudioConverter* instances,. The details of this interaction will become more clear during the discussion of the code examples further below.

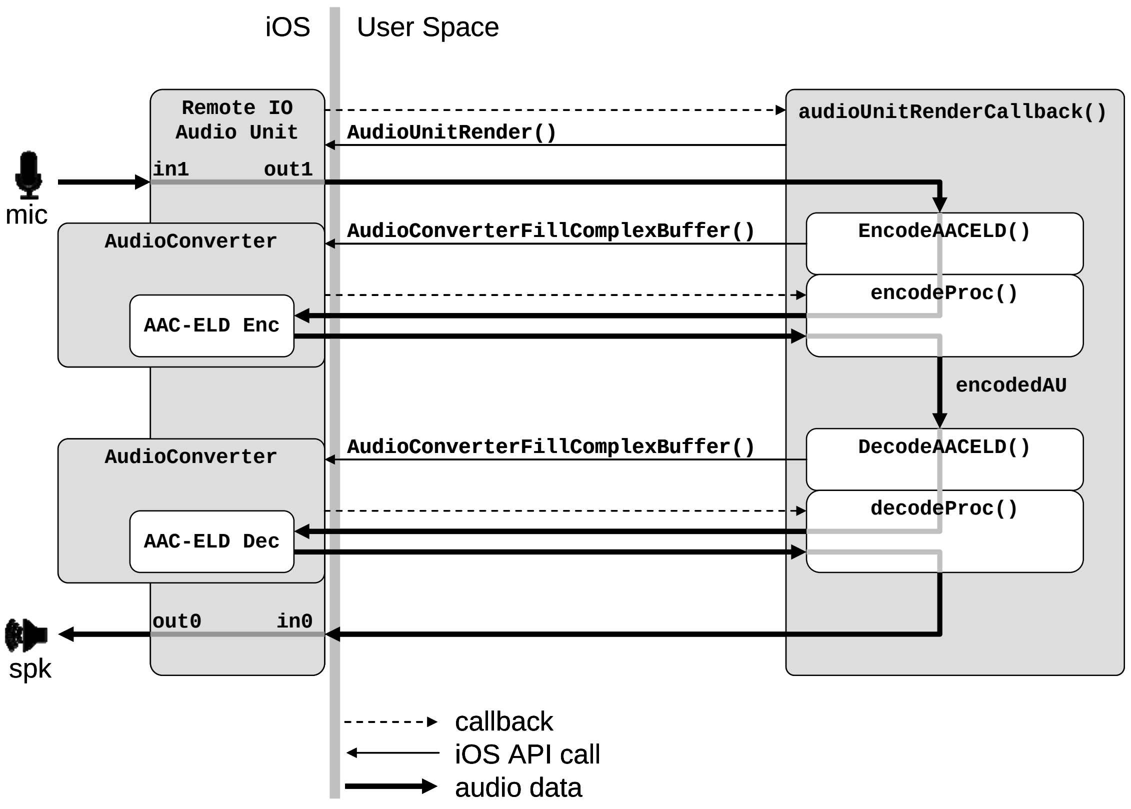

Figure 3: Outline of iOS components and signal flow in the demo application

2.1 AUDIO SESSION INITIALIZATION

Before the actual signal flow can become effective as illustrated in Fig. 3, the iOS components need to be initialized correctly. In preparation for the following code examples, we define a few global variables in Listing 1, which are used in order to increase the readability of the remaining code snippets. The meaning of these variables will be discussed in the corresponding sections.

```
AudioBuffer               g_inputBuffer;
AudioBuffer               g_outputBuffer;
AudioComponentInstance    g_audioUnit;
AudioUnitElement          g_outputBus       = 0;
AudioUnitElement          g_inputBus        = 1;
UInt32                    g_outChannels     = 2;
UInt32                    g_inChannels      = 1;
UInt32                    g_frameSize       = 512;
UInt32                    g_inputByteSize   = 0;
UInt32                    g_outputByteSize  = 0;
unsigned int              g_initialized     = 0;
AACELDEncoder            *g_encoder         = NULL;
AACELDDecoder            *g_decoder         = NULL;
MagicCookie               g_cookie;
```

Listing 1: Global Variables

As can be seen from Listing 1, an input and an output *AudioBuffer* are needed. Moreover, the number of input channels and the number of output channels are defined. Additionally, we assume a fixed frame size, i.e. the number of individual audio samples per channel in one audio frame (or packet) being 512. This applies for the whole course of the following discussion. Note that the types `AACELDEncoder` and `AACELDDecoder` are user defined data types that encapsulate the required state variables and configuration for the encoding and decoding process. Essentially, the required codec state consists of the number of input and output channels, the frame size, the sampling rate and the codec bitrate. Individual elements of these types, as well as the `MagicCookie` type will be discussed in Sections 2.4 and 2.5, respectively.

The general behavior of audio-enabled iOS applications can be managed with the *AudioSession* API. Every iOS application gets a singleton audio session upon launch time. By configuring the properties of the audio session and activating it, the application communicates its desired audio use to the operating system. Notably, the most important steps for an audio communication application are to configure the associated audio session to support simultaneous audio input and output as well as installing callback functions that react to interruption handling (e.g. when the application needs to terminate) and route changes (e.g. when a headset is plugged in). Another essential configuration requirement is to set the preferred hardware I/O buffer duration. While big buffers relax real time requirements, the buffers should be set as small as possible for real-time audio communication applications in order to reduce the overall I/O latency.

Lines 20 – 39 of Listing 2 show how to configure a corresponding audio session for an audio communication iOS application. For this and the following examples, we use an audio sampling rate of 44.1kHz and a frame size of 512 (`g_frameSize`), i.e. our application is processing 512 audio samples per channel at a time. Hence, the preferred I/O buffer time within the audio session should be 512 / 44100 seconds, corresponding to 11.6 milliseconds. Note, however, that setting the preferred hardware I/O buffer duration is just a hint to the operating system and might be changed by the *AudioUnit*. In our simple case, the *AudioUnit* is just able to fulfill the desired request but this does not need to be the case with all configurations. As a consequence, any implementation should check the actual number of samples received within the audio callback, especially if the implementation requires a specific number of samples in order to work correctly. Moreover, during the rest of the examples we also assume a one-channel input and a two-channel output. Though the integrated microphone can only produce mono input, we discuss the case of stereo output to highlight this special capability of AAC-ELD. When the internal microphone is used, the mono signal will be duplicated to the left and right stereo channel[1]. Note that while in lines 21 and 31 of Listing 2 we provide an interruption listener as well as a route change listener, their implementation is empty for the sake of simplicity. In a real world application, you would provide an appropriate implementation reacting to events such as forced termination or plugging in a headset.

The code example in Listing 2 also introduces the important concept of properties that is used throughout the audio APIs in iOS. The objects within the audio frameworks are configured by setting their properties accordingly. A property is essentially a key-value pair. The key is usually an enumeration constant defined by the API, such as `kAudioSessionProperty_AudioCategory`, whereas the value is of a particular data type appropriate for the properties' purposes, e.g. a `void*`, a `Float32`, an `UInt32` and so on. Detailed information on what value type a specific property expects can be found in the respective property's documentation. For the remainder of this paper, the keys and the value types should become clear from the provided code examples. Having set-up the current audio session for the application, the next step is to enable real time, low-latency audio input and output on the device.

[1] This is true for iOS versions >= 4.3 only. In previous iOS versions, the missing channel is filled with zeros.

```
void InitAudioUnit()
{
  /* Calculate the required input and output buffer sizes */
  g_inputByteSize  = g_frameSize * g_inChannels  * sizeof(AudioSampleType);
  g_outputByteSize = g_frameSize * g_outChannels * sizeof(AudioSampleType);

  /* Initialize the I/O buffers */
  g_inputBuffer.mNumberChannels = g_inChannels;
  g_inputBuffer.mDataByteSize   = g_inputByteSize;

  g_inputBuffer.mData           = malloc(sizeof(unsigned char)*g_inputByteSize);
  memset(g_inputBuffer.mData, 0, g_inputByteSize);

  g_outputBuffer.mNumberChannels = g_outChannels;
  g_outputBuffer.mDataByteSize   = g_outputByteSize;
  g_outputBuffer.mData           = malloc(sizeof(unsigned char)*g_outputByteSize);
  memset(g_outputBuffer.mData, 0, g_outputByteSize);
  g_initialized = 1;

  /* Initialize the audio session */
  AudioSessionInitialize(NULL, NULL, interruptionListener, NULL);
  /* Activate the audio session */
  AudioSessionSetActive(TRUE);

  /* Enable recording for full-duplex I/O */
  UInt32 audioCategory = kAudioSessionCategory_PlayAndRecord;
  AudioSessionSetProperty(kAudioSessionProperty_AudioCategory,
                          sizeof(audioCategory),
                          &audioCategory);
  /* Set the route change listener */
  AudioSessionAddPropertyListener(kAudioSessionProperty_AudioRouteChange,
                                  routeChangeListener,
                                  NULL);

  /* Set the preferred buffer time */
  Float32 preferredBufferTime = 512.0 / 44100.0;
  AudioSessionSetProperty(kAudioSessionProperty_PreferredHardwareIOBufferDuration,
                          sizeof(preferredBufferTime),
                          &preferredBufferTime);

  /* Setup the audio component for I/O */
  AudioComponentDescription componentDesc;
  memset(&componentDesc, 0, sizeof(componentDesc));

  componentDesc.componentType         = kAudioUnitType_Output;
  componentDesc.componentSubType      = kAudioUnitSubType_RemoteIO;
  componentDesc.componentManufacturer = kAudioUnitManufacturer_Apple;

  /* Find and create the audio component */
  AudioComponent auComponent = AudioComponentFindNext(NULL, &componentDesc);
  AudioComponentInstanceNew(auComponent, &g_audioUnit);

  /* Enable the audio input */
  UInt32 enableAudioInput = 1;
  AudioUnitSetProperty(g_audioUnit,
                       kAudioOutputUnitProperty_EnableIO,
                       kAudioUnitScope_Input,
                       g_inputBus,
                       &enableAudioInput,
                       sizeof(enableAudioInput));

  /* Setup the render callback */
  AURenderCallbackStruct renderCallbackInfo;
  renderCallbackInfo.inputProc       = audioUnitRenderCallback;
  renderCallbackInfo.inputProcRefCon = NULL;
  AudioUnitSetProperty(g_audioUnit, kAudioUnitProperty_SetRenderCallback,
                       kAudioUnitScope_Input, g_outputBus,
                       &renderCallbackInfo, sizeof(renderCallbackInfo));

  /* Set the input and output audio stream formats */
  AudioStreamBasicDescription audioFormat;
  audioFormat.mSampleRate       = 44100;
  audioFormat.mFormatID         = kAudioFormatLinearPCM;
  audioFormat.mFormatFlags      = kAudioFormatFlagIsSignedInteger | kAudioFormatFlagIsPacked;
  audioFormat.mFramesPerPacket  = 1;
  audioFormat.mBitsPerChannel   = 8 * sizeof(AudioSampleType);
  audioFormat.mChannelsPerFrame = g_inChannels;
  audioFormat.mBytesPerFrame    = audioFormat.mChannelsPerFrame * sizeof(AudioSampleType);
  audioFormat.mBytesPerPacket   = audioFormat.mBytesPerFrame;

  AudioUnitSetProperty(g_audioUnit,
                       kAudioUnitProperty_StreamFormat,
                       kAudioUnitScope_Output,
                       g_inputBus,
                       &audioFormat,
                       sizeof(audioFormat));

  audioFormat.mChannelsPerFrame = g_outChannels;
  audioFormat.mBytesPerFrame    = audioFormat.mChannelsPerFrame * sizeof(AudioSampleType);
  audioFormat.mBytesPerPacket   = audioFormat.mBytesPerFrame;

  AudioUnitSetProperty(g_audioUnit,
                       kAudioUnitProperty_StreamFormat, kAudioUnitScope_Input,
                       g_outputBus, &audioFormat,
                       sizeof(audioFormat));

  /* Initialize the ELD codec */
  InitAACELD();
}
```

Listing 2: Initialization of the AudioUnit

2.2 LOW-LATENCY AND FULL-DUPLEX AUDIO I/O

When implementing a real time VoIP application, it is required to set up low-latency access to the audio I/O hardware of the device. Furthermore, full-duplex, i.e. simultaneous recording and playback of audio, is required. Within the iOS SDK, Apple provides this access to the hardware by the means of so called *AudioUnits* [6]. AudioUnits are audio processing plug-ins that can be dynamically loaded and used by the application. The major advantage of using AudioUnits over the more sophisticated higher-level APIs provided by the SDK is that AudioUnits constitute the lowest programming layer of the iOS audio stack. Thereby, they provide access to a real time priority thread associated with their render callback methods. Thus, the software developer gains maximum responsiveness and a minimum processing delay, which is essential for a high quality audio communication application. As a drawback, however, AudioUnits are also more complicated to use and require a deeper understanding than the higher-level iOS audio APIs. In the remainder of this section, only the AudioUnits designed for recording and playback will be discussed.

iOS provides three AudioUnits for audio I/O. The *Remote I/O* unit is the most commonly used I/O, as it directly connects to the input and output of the audio hardware. Accordingly, the Remote I/O unit offers low-latency access to the incoming and outgoing audio samples. As an add-on to the Remote I/O unit, the *Voice-Processing I/O* unit is also available providing gain control and echo cancelation specifically for the use in VoIP applications. The third output I/O unit is termed *Generic Output*, which is intended for offline audio processing and not suited for real time audio I/O. For the purpose of this discussion, we will focus on the *Remote I/O* unit.

The general workflow for audio processing using AudioUnits is as follows [6]:

1. Obtain a reference to the libraries that implement specific units
2. Instantiate an AudioUnit
3. Interconnect AudioUnits and/or attach render callback functions
4. Start the audio processing workflow

AudioUnits are components within the iOS audio framework. As such, a reference to the library that implements a specific AudioUnit can be acquired by filling in the appropriate fields of an `AudioComponentDescription` structure and querying the system for the component library by a call to the API function `AudioComponentFindNext()`. As the name of the function suggests, there can be more than one component for a single description. Once a reference to the audio component is found and returned by the system, the audio unit is instantiated by calling `AudioComponentInstanceNew()`. This is illustrated in lines 50 - 51 of Listing 2. Here, the resulting reference to the audio unit is stored in a global variable `g_audioUnit` of type `AudioComponentInstance`. In order to understand the following code, it is necessary to describe the internal structure of I/O AudioUnits in more detail.

As shown in Figure 3, an I/O AudioUnit consists of two *elements*, *Element 1* and *Element 0*. Each element is divided into two *scopes*: the *input scope* and the *output scope*. For the I/O unit, the *input scope* of *Element 1* is directly connected to the recording hardware, whereas the *output scope* of *Element 0* is directly connected to the speakers. An application that will record audio samples, process them and play them out is required to connect in between these two elements. Hence, the processing unit of the application will be connected to the *output scope of Element 1* and to the *input scope of Element 0*. In terms of the AudioUnit API, the scopes are addressed by the use of the predefined constants `kAudioUnitScope_Input` and `kAudioUnitScope_Output`. The elements, which are also termed *buses*, are simply addressed by specifying the integer constants 0 and 1, respectively. These constants are defined in the global variables `g_inputBus` and `g_outputBus` in Listing 1.

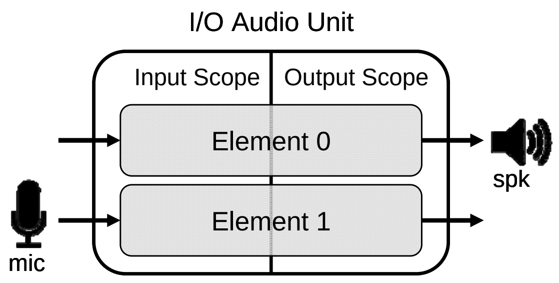

Figure 3: Anatomy of the I/O Audio Unit.

When using a Remote I/O audio unit, audio input is disabled by default. It can be enabled by setting the property `kAudioOutputUnitProperty_EnableIO` on the input scope of *Element 1* (i.e. the input bus) to 1 (i.e. true) with the function `AudioUnitSetProperty()`. This is shown in lines 54 – 60 of Listing 2. As can be seen from the example, the property mechanism for audio units is exactly the same as that for the audio session object. The next step is to set a render callback function to the output bus of the input element in order to get access to the sample values that are recorded by the microphone. Thereby, the user application code is connected to the I/O unit. The render callback function, which will be discussed later, is attached to the I/O unit as a property setting. The relevant structure `AURenderCallbackStruct` (Listing 2, Lines 63 – 68) consists of two elements: A pointer to a user defined callback function and a pointer to user specified data that is passed to the callback function when invoked by the audio unit.

The final step in configuring the I/O audio unit is to specify the expected sample format at the input and to define the provided sample format for the respective output buses. This is done by setting the fields of a structure of type `AudioStreamBasicDescription`. In the presented example, the input and the output of the I/O unit will be linear PCM (`kAudioFormatLinearPCM`) with two channels and a sampling rate of 44.1kHz. Lines 71 – 79 of Listing 2 show the single elements of the structure. For linear PCM data, there is also a convenience macro defined in the API that will fill out the required fields of the structure. This will be shown in Sections 2.4 and 2.5, when configuring the encoders and decoders for AAC-ELD. The resulting stream descriptions are then attached to the input bus of the output scope and to the output bus of the input scope of the audio unit. Note that there is no need to take care of the internal sampling rates or stream formats that are used by the device hardware, as audio units are able to perform dynamic resampling at their respective inputs and outputs.

2.3 AUDIO PROCESSING WITHIN THE RENDER CALLBACK

The render callback function is where the main processing of the VoIP application takes place. For each audio frame the callback function is executed once. Recorded audio samples are received from the I/O AudioUnit and at the same time the processed samples are passed back for play out. In between, the recorded samples are encoded and subsequently decoded again. In a real-world VoIP application, the encoded audio samples would be packetized into IP packets (typically using UDP/RTP) and sent over the network. At the same time, incoming packets would be received and depacketized before decoding. Packetization and IP transmission are omitted in the provided demo application, as the focus of this paper is on audio I/O and coding.

As shown in Listing 3, the render callback function obtains the recorded input samples from the AudioUnits input bus by a call to the `AudioUnitRender()` function. The acquired audio samples are then copied into the global buffer `g_inputBuffer` of type `AudioBuffer` [2]. This audio buffer is subsequently encoded using the AAC-ELD codec by a call to `EncodeAACELD()`. Both, this function and the corresponding `EncodedAudioBuffer` structure are not part of the iOS API. Their implementation is discussed in the next section. Subsequently, the just encoded buffer is decoded into a global output buffer `g_outputBuffer` by a call to `DecodeAACELD()`. This is also a user-defined function and will be discussed in Section 2.5. Finally, the decoded audio data is copied into the playback buffer (`ioData`). This buffer is provided by the system upon performing a call to the render callback function [6]. In order to start the audio unit processing workflow, the audio unit has to be initialized by a call to `AudioUnitInitialize()`, followed by a call to `AudioUnitStart()`.

```
/* The render call back that is called by the audio system */
static OSStatus audioUnitRenderCallback(void                        *inRefCon,
                                        AudioUnitRenderActionFlags  *ioActionFlags,
                                        const AudioTimeStamp        *inTimeStamp,
                                        UInt32                       inBusNumber,
                                        UInt32                       inNumberOfFrames,
                                        AudioBufferList             *ioData)
{

  /* Get the input samples */
  AudioUnitRender(g_audioUnit,
                  ioActionFlags,
                  inTimeStamp,
                  g_inputBus,
                  inNumberOfFrames,
                  ioData);

  /* Copy to global input buffer */
  memcpy(g_inputBuffer.mData, ioData->mBuffers[0].mData, g_inputBuffer.mDataByteSize);

  /* Encode with AudioConverter */
  EncodedAudioBuffer encodedAU;
  EncodeAACELD(g_encoder, &g_inputBuffer, &encodedAU);

  /* Decode with AudioConverter */
  g_outputBuffer.mDataByteSize = g_outputByteSize;
  DecodeAACELD(g_decoder, &encodedAU, &g_outputBuffer);

  /* Copy output samples to Audio Units' IO buffer */
  ioData->mBuffers[0].mNumberChannels = g_outputBuffer.mNumberChannels;
  ioData->mBuffers[0].mDataByteSize   = g_outputBuffer.mDataByteSize;
  memcpy(ioData->mBuffers[0].mData, g_outputBuffer.mData, g_outputBuffer.mDataByteSize);

  return noErr;
}
```

Listing 3: The Audio Unit Render Callback

2.4 AAC-ELD ENCODING

CoreAudio provides a general means of audio format conversion through the AudioConverter Services API [2]. The available conversions can be as simple as sampling rate and channel conversions as well as more complex operations such as encoding and decoding PCM to various other formats. The AudioConverter API uses `AudioStreamBasicDescription` structures in order to define the source and destination audio formats. By setting a source and a destination format, a specific audio converter can be created by calling `AudioConverterNew()`. For the demo application discussed in this paper, the individual state variables that are required to initialize and use the AudioConverter are aggregated into a user-defined structure named `AACELDEncoder`. Single elements of this structure are discussed during the course of the text, for implementation details the reader is referred to the provided source code.

Since our intention is to encode from linear PCM (LPCM) to AAC-ELD, the source data format is set to LPCM by using the convenience macro `FillOutASBDForLPCM` (Line 11, Listing 4) which sets the appropriate `AudioStreamBasicDescription` structure fields for LPCM data. When encoding to AAC-ELD, the destination data format must be set to `kAudioFormatMPEG4AAC_ELD`[1]. Since the user generally does not know the internal format description that is used by the specific implementation of the encoder, the system has to fill in the details of the structure. Therefore, only the format id, the sampling rate and the number of channels have to be set for describing the desired encoder configuration in the

[1] With the introduction of iOS 5.0, the Apple implementation also supports AAC-ELD with SBR [1] by specifying the constant `kAudioFormatMPEG4AAC_ELD_SBR`.

AudioStreamBasicDescription for the destination format. Then the details can be filled in by a call to AudioFormatGetProperty() with a property id of kAudioFormatProperty_FormatInfo (Line 28, Listing 4). If the system finds an encoder that can handle the desired format, the resulting AudioStreamBasicDescription structure can be used for the creation of an audio converter.

Once the audio converter has been created, the desired output bitrate can be set by a call to AudioConverterSetProperty() as shown in line 45 – 51 of Listing 4. Typical bitrates for AAC-ELD are 32-64 kbps for mono or 64-128 kbps for stereo (but may be lower as well as higher, if required). With bitrates of 48 kbps and lower, it is highly recommended to use AAC-ELD with SBR enabled (kAudioFormatMPEG4AAC_ELD_SBR). Subsequently, the maximum output packet size is queried in lines 54ff. of Listing 4. This is required in order to allocate sufficient buffer memory for the generated output packets. Finally, it is necessary to obtain the so-called encoder *magic cookie* for later decoding purposes. Within the context of CoreAudio, the *magic cookie* is an opaque set of metadata that is attached to an encoded or compressed block of audio data. The metadata contains information that is required by the decoder in order to properly decode the data. The *magic cookie* associated with the encoder object is obtained by firstly querying the audio converter for the cookie size. Then, sufficient memory is allocated and the cookie data is retrieved by calling the function AudioConverterGetProperty() and providing kAudioConverterCompressionMagicCookie as the requested property type and the allocated buffer (Lines 76 – 83, Listing 4). For those familiar with the terms and nomenclature of MPEG audio codecs, it is worth noting that the *magic cookie* contains the Audio Specific Config (ASC), which would be also needed for correct signaling when using the Session Description Protocol (SDP). In the demo code, the cookie data and its size have been aggregated into the user-defined structure MagicCookie, which allows for passing the cookie data to the decoder easily.

The subsequent encoding of the PCM sample packets is straightforward as shown in Listing 5. The user calls the EncodeAACELD function, providing the input audio buffer. First, a reference to the encoding buffer is stored. Additionally, an array of type AudioStreamPacketDescription and an AudioBufferList are required for audio data conversion. These are initialized with respect to the input data and the expected output data. In the example code, one input packet (i.e. one frame containing 512 LPCM samples per channel) is provided, and one encoded output packet is requested. Thus, only one AudioStreamPacketDescription element is created and only one output AudioBuffer is expected (Lines 52 and 55 – 59 of Listing 5, respectively). The output buffer is filled by a call to AudioConverterFillComplexBuffer(). This starts the encoding process.

The encoder will request source data packets from the user by calling a user-defined callback function. This callback function (encodeProc) is called whenever the encoder is ready to process an audio packet. In the demo implementation, the maximum possible number of output packets is calculated (which is always one in this case). The ioNumberDataPackets variable is set accordingly, if the number of requested packets exceed the number of maximum packets. Subsequently, the provided ioData parameter is adjusted to point to the input LPCM samples that were provided to the original encoder call. Since all input LPCM samples are sent to the encoder upon the time of the call, the remaining number of bytes to encode (stored in the encoder member variable bytesToEncode) is set to zero. However, it is important to mention that the AudioConverter could call the encodeProc again. If this is the case and the variable bytesToEncode is zero, a value other than noErr has to be returned in order to keep the encoder state working. If bytesToEncode is zero and thus ioNumberDataPackets would be set to zero and the return value is noErr this would signal to the encoder end of audio processing (see also Apple Technical Q&A QA1317 [9]). The parameter outDataPacketDescription can be set to NULL, as the PCM input packet format does not use a specific packet description [2].

When the call to AudioConverterFillComplexBuffer() returns, the encoded audio packet can be found in the data member of the output buffer list, which was previously stored in the member variable encoderBuffer of the AACELDEncoder structure. Additionally, the size of the encoded buffer is stored in the mDataByteSize member of the outPacketDesc variable, which is generated by the audio encoder. The variables are then passed on to the caller of the EncodeAACELD function by using the user-defined structure EncodedAudioBuffer. After that, the encoding of the audio frame is complete and the compressed bitstream data for one LPCM frame, also called *Access Unit* (AU), is to be found in the data structure `encodedAU`.

```
int InitAACELDEncoder(AACELDEncoder *encoder, EncoderProperties props, MagicCookie *outCookie)
{
  /* Copy the provided encoder properties */
  encoder->inChannels   = props.inChannels;
  encoder->outChannels  = props.outChannels;
  encoder->samplingRate = props.samplingRate;
  encoder->frameSize    = props.frameSize;
  encoder->bitrate      = props.bitrate;
  /* Convenience macro to fill out the ASBD structure.
     Available only when __cplusplus is defined! */
  FillOutASBDForLPCM(encoder->sourceFormat,
                     encoder->samplingRate,
                     encoder->inChannels,
                     8*sizeof(AudioSampleType),
                     8*sizeof(AudioSampleType),
                     false,
                     false);

  /* Set the format parameters for AAC-ELD encoding. */
  encoder->destinationFormat.mFormatID         = kAudioFormatMPEG4AAC_ELD;
  encoder->destinationFormat.mChannelsPerFrame = encoder->outChannels;
  encoder->destinationFormat.mSampleRate       = encoder->samplingRate;

  /* Get the size of the formatinfo structure */
  UInt32 dataSize = sizeof(encoder->destinationFormat);

  /* Request the propertie from CoreAudio */
  AudioFormatGetProperty(kAudioFormatProperty_FormatInfo,
                         0,
                         NULL,
                         &dataSize,
                         &(encoder->destinationFormat));

  /* Create a new audio converter */
  AudioConverterNew(&(encoder->sourceFormat),
                    &(encoder->destinationFormat),
                    &(encoder->audioConverter));

  if (!encoder->audioConverter)
  {
    return -1;
  }

  /* Try to set the desired output bitrate */
  UInt32 outputBitrate = encoder->bitrate;
  dataSize = sizeof(outputBitrate);

  AudioConverterSetProperty(encoder->audioConverter,
                            kAudioConverterEncodeBitRate,
                            dataSize,
                            &outputBitrate);

  /* Query the maximum possible output packet size */
  if (encoder->destinationFormat.mBytesPerPacket == 0)
  {
    UInt32 maxOutputSizePerPacket = 0;
    dataSize = sizeof(maxOutputSizePerPacket);
    AudioConverterGetProperty(encoder->audioConverter,
                              kAudioConverterPropertyMaximumOutputPacketSize,
                              &dataSize,
                              &maxOutputSizePerPacket);
    encoder->maxOutputPacketSize = maxOutputSizePerPacket;
  }
  else
  {
    encoder->maxOutputPacketSize = encoder->destinationFormat.mBytesPerPacket;
  }

  /* Fetch the Magic Cookie from the ELD implementation */
  UInt32 cookieSize = 0;
  AudioConverterGetPropertyInfo(encoder->audioConverter,
                                kAudioConverterCompressionMagicCookie,
                                &cookieSize,
                                NULL);

  char* cookie = (char*)malloc(cookieSize*sizeof(char));
  AudioConverterGetProperty(encoder->audioConverter,
                            kAudioConverterCompressionMagicCookie,
                            &cookieSize,
                            cookie);

  outCookie->data     = cookie;
  outCookie->byteSize = cookieSize;

  /* Prepare the temporary AU buffer for encoding */
  encoder->encoderBuffer = malloc(encoder->maxOutputPacketSize);

  return 0;
}
```

Listing 4: Initialization of the AudioConverter for AAC-ELD Encoding

```
static OSStatus encodeProc(AudioConverterRef inAudioConverter,
                           UInt32 *ioNumberDataPackets,
                           AudioBufferList *ioData,
                           AudioStreamPacketDescription **outDataPacketDescription,
                           void *inUserData)
{
  /* Get the current encoder state from the inUserData parameter */
  AACELDEncoder *encoder = (AACELDEncoder*) inUserData;

  /* Compute the maximum number of output packets */
  UInt32 maxPackets = encoder->bytesToEncode / encoder->sourceFormat.mBytesPerPacket;

  if (*ioNumberDataPackets > maxPackets)
  {
    /* If requested number of packets is bigger, adjust */
    *ioNumberDataPackets = maxPackets;
  }

  /* Check to make sure we have only one audio buffer */
  if (ioData->mNumberBuffers != 1)
  {
    return 1;
  }

  /* Set the data to be encoded */
  ioData->mBuffers[0].mDataByteSize    = encoder->currentSampleBuffer->mDataByteSize;
  ioData->mBuffers[0].mData            = encoder->currentSampleBuffer->mData;
  ioData->mBuffers[0].mNumberChannels = encoder->currentSampleBuffer->mNumberChannels;

  if (outDataPacketDescription)
  {
    *outDataPacketDescription = NULL;
  }

  if (encoder->bytesToEncode == 0)
  {
    // We are currently out of data but want to keep on processing
    // See Apple Technical Q&A QA1317
    return 1;
  }

  encoder->bytesToEncode = 0;

  return noErr;
}


int EncodeAACELD(AACELDEncoder *encoder, AudioBuffer *inSamples, EncodedAudioBuffer *outData)
{
  /* Clear the encoder buffer */
  memset(encoder->encoderBuffer, 0, sizeof(encoder->maxOutputPacketSize));

  /* Keep a reference to the samples that should be encoded */
  encoder->currentSampleBuffer = inSamples;
  encoder->bytesToEncode       = inSamples->mDataByteSize;

  UInt32 numOutputDataPackets = 1;

  AudioStreamPacketDescription outPacketDesc[1];

  /* Create the output buffer list */
  AudioBufferList outBufferList;
  outBufferList.mNumberBuffers = 1;
  outBufferList.mBuffers[0].mNumberChannels = encoder->outChannels;
  outBufferList.mBuffers[0].mDataByteSize    = encoder->maxOutputPacketSize;
  outBufferList.mBuffers[0].mData            = encoder->encoderBuffer;

  /* Start the encoding process */
  OSStatus status = AudioConverterFillComplexBuffer(encoder->audioConverter,
                                                    encodeProc,
                                                    encoder,
                                                    &numOutputDataPackets,
                                                    &outBufferList,
                                                    outPacketDesc);

  if (status != noErr)
  {
    return -1;
  }

  /* Set the ouput data */
  outData->mChannels      = encoder->outChannels;
  outData->data           = encoder->encoderBuffer;
  outData->mDataBytesSize = outPacketDesc[0].mDataByteSize;

  return 0;
}
```

Listing 5: AAC-ELD Encoding using the AudioConverter API

2.5 AAC-ELD DECODING

Decoding of AAC-ELD access units using an AudioConverter works analogously to the encoding process. This time, however, the source stream description's format is `kAudioFormatMPEG4AAC_ELD` whereas the destination stream format is LPCM. As shown in Listing 6, the initialization is straightforward. The maximum output packet size could be fixed at 2048 bytes for the demo case (i.e. 2 bytes times 2 channels times 512 samples) but is queried from the *AudioConverter* state for correctness of the implementation. Finally, the *magic cookie* previously obtained from the encoder is provided to the decoder (as `kAudioConverterDecompressionMagicCookie`) for initialization and correct operation.

```c
int InitAACELDDecoder(AACELDDecoder* decoder, DecoderProperties props, const MagicCookie *cookie)
{
  /* Copy the provided decoder properties */
  decoder->inChannels   = props.inChannels;
  decoder->outChannels  = props.outChannels;
  decoder->samplingRate = props.samplingRate;
  decoder->frameSize    = props.frameSize;

  /* We will decode to LPCM */
  FillOutASBDForLPCM(decoder->destinationFormat,
                     decoder->samplingRate,
                     decoder->outChannels,
                     8*sizeof(AudioSampleType),
                     8*sizeof(AudioSampleType),
                     false,
                     false);

  /* from AAC-ELD, having the same sampling rate, but possibly a different channel configuration */
  decoder->sourceFormat.mFormatID         = kAudioFormatMPEG4AAC_ELD;
  decoder->sourceFormat.mChannelsPerFrame = decoder->inChannels;
  decoder->sourceFormat.mSampleRate       = decoder->samplingRate;

  /* Get the rest of the format info */
  UInt32 dataSize = sizeof(decoder->sourceFormat);
  AudioFormatGetProperty(kAudioFormatProperty_FormatInfo,
                         0,
                         NULL,
                         &dataSize,
                         &(decoder->sourceFormat));

  /* Create a new AudioConverter instance for the conversion AAC-ELD -> LPCM */
  AudioConverterNew(&(decoder->sourceFormat),
                    &(decoder->destinationFormat),
                    &(decoder->audioConverter));

  if (!decoder->audioConverter)
  {
    return -1;
  }

  /* Check for variable output packet size */
  if (decoder->destinationFormat.mBytesPerPacket == 0)
  {
    UInt32 maxOutputSizePerPacket = 0;
    dataSize = sizeof(maxOutputSizePerPacket);
    AudioConverterGetProperty(decoder->audioConverter,
                              kAudioConverterPropertyMaximumOutputPacketSize,
                              &dataSize,
                              &maxOutputSizePerPacket);
    decoder->maxOutputPacketSize = maxOutputSizePerPacket;
  }
  else
  {
    decoder->maxOutputPacketSize = decoder->destinationFormat.mBytesPerPacket;
  }

  /* Set the corresponding encoder cookie */
  AudioConverterSetProperty(decoder->audioConverter,
                            kAudioConverterDecompressionMagicCookie,
                            cookie->byteSize,
                            cookie->data);

  return 0;
}
```

Listing 6: Initialization of the AudioConverter for AAC-ELD Decoding

The decoding process, as shown in Listing 7, is very similar to the encoding process. However, since there are always 512 samples processed at a time when encoding, there are also 512 single LPCM samples being decoded from one AAC-ELD input packet. In order to have sufficient storage for the output samples the allocation of an array of 512 elements of type `AudioStreamPacketDescription` is required for holding the samples that will be generated by the `AudioConverterFillComplexBuffer()` function call. This time the output buffer provided as an element of the `AudioBufferList` is directly associated with the `outSamples` structure.

The major difference between the encoding and decoding callback functions is that in the callback that is used for AAC-ELD to LPCM decoding, a packet description structure has to be provided for proper processing of the input data. The packet description structure is necessary since multiple AAC-ELD AU's of different byte sizes could be provided as an input to the decoder via a single call. Setting the required fields, especially the start offset and the packet size is straightforward for the demo case as only one input packet containing the encoded data is present (Lines 28ff., Listing 7). Note that the name `outDataPacketDescription` can be confusing, as what is actually described from the user perspective is the *input* packet, but the variable is for output purposes. The naming was chosen to be consistent with the iOS SDK examples.

```
static OSStatus decodeProc(AudioConverterRef inAudioConverter,
                           UInt32 *ioNumberDataPackets,
                           AudioBufferList *ioData,
                           AudioStreamPacketDescription **outDataPacketDescription,
                           void *inUserData)
{
  /* Get the current decoder state from the inUserData parameter */
  AACELDDecoder *decoder = (AACELDDecoder*)inUserData;

  /* Compute the maximum number of output packets */
  UInt32 maxPackets = decoder->bytesToDecode / decoder->maxOutputPacketSize;

  if (*ioNumberDataPackets > maxPackets)
  {
    /* If requested number of packets is bigger, adjust */
    *ioNumberDataPackets = maxPackets;
  }

  /* If there is data to be decoded, set it accordingly */
  if (decoder->bytesToDecode)
  {
    ioData->mBuffers[0].mData           = decoder->decodeBuffer;
    ioData->mBuffers[0].mDataByteSize   = decoder->bytesToDecode;
    ioData->mBuffers[0].mNumberChannels = decoder->inChannels;
  }

  /* And set the packet description */
  if (outDataPacketDescription)
  {
    decoder->packetDesc[0].mStartOffset            = 0;
    decoder->packetDesc[0].mVariableFramesInPacket = 0;
    decoder->packetDesc[0].mDataByteSize           = decoder->bytesToDecode;

    (*outDataPacketDescription) = decoder->packetDesc;
  }

  if (decoder->bytesToDecode == 0)
  {
    // We are currently out of data but want to keep on processing
    // See Apple Technical Q&A QA1317
    return 1;
  }

  decoder->bytesToDecode = 0;

  return noErr;
}

int DecodeAACELD(AACELDDecoder* decoder, EncodedAudioBuffer *inData, AudioBuffer *outSamples)
{
  OSStatus status = noErr;

  /* Keep a reference to the samples that should be decoded */
  decoder->decodeBuffer  = inData->data;
  decoder->bytesToDecode = inData->mDataBytesSize;

  UInt32 outBufferMaxSizeBytes = decoder->frameSize * decoder->outChannels * sizeof(AudioSampleType);

  assert(outSamples->mDataByteSize <= outBufferMaxSizeBytes);

  UInt32 numOutputDataPackets = outBufferMaxSizeBytes / decoder->maxOutputPacketSize;

  /* Output packet stream are 512 LPCM samples */
  AudioStreamPacketDescription outputPacketDesc[512];

  /* Create the output buffer list */
  AudioBufferList outBufferList;
  outBufferList.mNumberBuffers = 1;
  outBufferList.mBuffers[0].mNumberChannels = decoder->outChannels;
  outBufferList.mBuffers[0].mDataByteSize   = outSamples->mDataByteSize;
  outBufferList.mBuffers[0].mData           = outSamples->mData;

  /* Start the decoding process */
  status = AudioConverterFillComplexBuffer(decoder->audioConverter,
                                           decodeProc,
                                           decoder,
                                           &numOutputDataPackets,
                                           &outBufferList,
                                           outputPacketDesc);

  if (noErr != status)
  {
    return -1;
  }

  return 0;
}
```

Listing 7: AAC-ELD Decoding using the AudioConverter API

2.6 INTEROPERABILITY TO ANDROID 4.1

With OS version 4.1 (Jelly Bean) the AAC-ELD codec is also available in Android [10]. Integration of the AAC-ELD codec is required by vendors in order to fulfill the Android 4.1 compatibility description [11]. The public Android AAC-ELD implementation is interoperable to the implementation on iOS devices and the Android AAC-ELD decoder supports all modes available from the iOS encoder. However, the Java MediaCodec API of the Android 4.1 AAC-ELD encoder has several limitations with respect to the accessible AAC-ELD codec parameters. 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 limited configurations.

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

- AAC-ELD without SBR (`kAudioFormatMPEG4AAC_ELD`)
  - Sampling rate 22.05kHz
  - Frame length 512
  - Bitrate: 32.729 kbps and higher

- AAC-ELD without SBR (`kAudioFormatMPEG4AAC_ELD`)
  - Sampling rate 44.1 kHz
  - Frame length 512
  - Bitrate: 65.459 kbps and higher

2.7 DEMO APPLICATION SOURCE CODE

The Listings 1 - 7 are provided for the explanation of the concepts of iOS CoreAudio with specific source code examples at hand. They are extracted from a working application that performs real time audio I/O and coding with AAC-ELD. In order to keep the text size appropriate, not every detail of the code can be discussed. However, the complete source is available with this paper 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 iOS device. By changing the parameters (e.g. the bitrate or from stereo to mono operation) 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 contains a ready to use Xcode 4.3 project file for building the sample application. 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.

3 AUDIO COMMUNICATION ENGINE

As has been demonstrated in this paper, iOS provides the basic components for implementing high-quality audio communication based on AAC-ELD. Especially because of the native support in iOS, 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 native iOS 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 iOS API can become challenging when real time constraints require the usage of lower-level functions and services. Correct initialization of *AudioUnits* and *AudioConverters* together with callback functions requires a certain expertise in system development and MPEG audio coding. Though the API usage can be worked out with sufficient patience (and with the help of developer guides such as this one), the software development process can still be time consuming and cumbersome. This is all the more true when multi-threading becomes necessary in order to handle real time issues on the audio interface as well as on the network interface. In summary, the task of integrating all the required components using lower-level APIs can be demanding even if they are readily available.

For companies and application developers for whom time to market is key, Fraunhofer IIS has therefore developed the Audio Communication Engine (ACE) and ported it to iOS. 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. For example, it can easily be combined with a SIP-client or other session setup protocols to allow for maximum freedom in service offering. Though the ACE uses the same high quality audio codec as FaceTime it does not assure interoperability.

For further information on the ACE, please visit the Fraunhofer IIS home page [7,8].

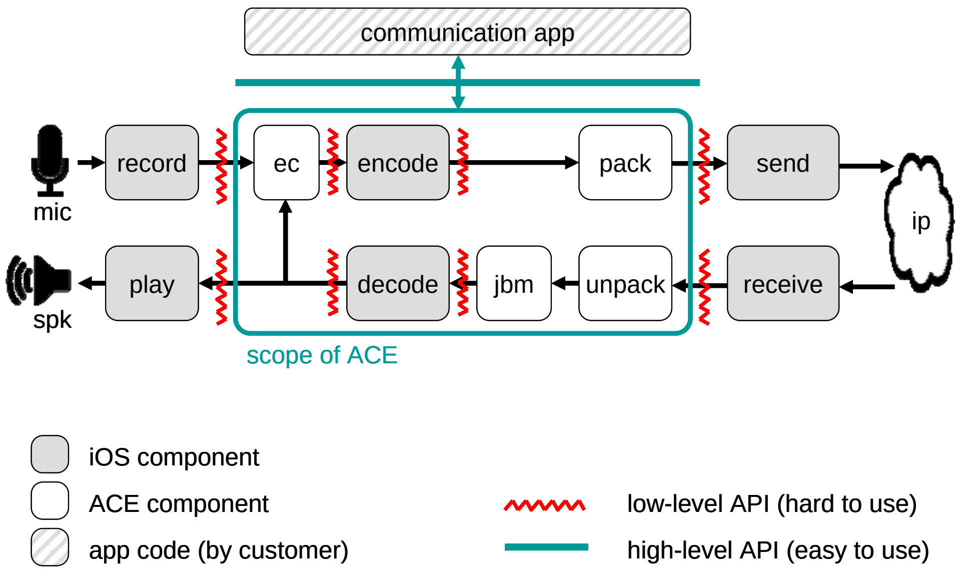

Figure 4: Scope and benefit of using the ACE for building a complete VoIP app on iOS

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] Apple Inc. (2008, Nov.) Core Audio Overview. [Online]. http://developer.apple.com/library/ios/#documentation/MusicAudio/Conceptual/CoreAudioOverview/Introduction/Introduction.html#//apple_ref/doc/uid/TP40003577

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

[4] Apple Inc. (2011, July) iOS Dev Center. [Online]. http://developer.apple.com/library/ios/navigation/index.html

[5] Apple Inc. (2010, Nov.) Audio Session Programming Guide. [Online]. http://developer.apple.com/library/ios/#documentation/Audio/Conceptual/AudioSessionProgrammingGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40007875

[6] Apple Inc. (2010, Sep.) Audio Unit Hosting Guide. [Online]. http://developer.apple.com/library/ios/#documentation/MusicAudio/Conceptual/AudioUnitHostingGuide_iOS/Introduction/Introduction.html#//apple_ref/doc/uid/TP40009492

[7] 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

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

[9] Apple Inc. (2011, Jul.) Apple Technical Q&A QA1317 [Online]. http://developer.apple.com/library/mac/#qa/qa1317/_index.html#//apple_ref/doc/uid/DTS10002349

[10] Google Inc. (2012) Android Developer Website [Online]. http://developer.android.com/index.html

[11] Google Inc. (2012), Android 4.1 Compatibility Definition [Online]. http://source.android.com/compatibility/4.1/android-4.1-cdd.pdf

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.