Sample Code

Encoding and decoding audio

Convert audio formats to efficiently manage data and quality.

Download

macOS 12.0+ | Xcode 13.3+

Overview

This sample shows how to encode and decode audio by using Audio Toolbox. If it's encoding, the sample outputs the audio using the Advanced Audio Coding (AAC) audio format. AAC is a compressed, lossy audio format that maintains audio quality while using less storage space. When the sample decodes audio, the output audio format is represented as pulse code modulation (PCM). PCM is both an uncompressed and lossless audio format, resulting in a larger file format.

The sample also explores how to set up data structures that describe audio formats, read and write packets to and from audio files, retrieve and set magic cookies, and loop through the input callback that signals the end of a buffer stream.

Create input and output descriptions

Whether it's encoding or decoding audio, the sample uses the structure `AudioStreamBasicDescription` to describe the input and output audio data formats. When encoding audio, the sample checks whether the input audio is in PCM format. If the input is not PCM, the sample fails because a single audio converter instance can't transcode (decode one non-PCM format and encode to another non-PCM format).

```
// Open the input file and get its data format.
auto inputFile = AudioFile::Open(argv[2]);
AudioStreamBasicDescription inputDescription;
inputFile.GetProperty(
```

```
    kAudioFilePropertyDataFormat,
    sizeof(inputDescription),
    &inputDescription);

// If encoding, make sure the input data is PCM.
if (encode && (inputDescription.mFormatID != kAudioFormatLinearPCM)) {
    std::cerr << "The input file data format is not PCM" << std::endl;
    return EXIT_FAILURE;
}
```

The sample checks whether the input uses packet descriptions. Packet descriptions accompany packets if the audio format indicates a variable number of bytes per packet, or frames per packet. Function calls and callbacks may produce or consume packet descriptions. Provide a buffer even if the client doesn't need them or if it's only processing a single packet.

```
const bool inputUsesPacketDescriptions =
    (inputDescription.mBytesPerPacket == 0 || inputDescription.mFramesPerPacket == 0
```

The sample initializes the output description with the input sample rate and channels per frame. Once created, the output configuration depends on whether the sample is encoding or decoding the audio.

```
// Create the output file as PCM or AAC of the same sampling rate and number of char
// the input.
AudioStreamBasicDescription outputDescription{
    .mSampleRate = inputDescription.mSampleRate,
    .mChannelsPerFrame = inputDescription.mChannelsPerFrame,
};
AudioFileTypeID outputFileType;
if (encode) {
    outputFileType = kAudioFileM4AType;
    outputDescription.mFormatID = kAudioFormatMPEG4AAC;
    outputDescription.mFormatFlags = kAudioFormatFlagsAreAllClear;
    outputDescription.mFramesPerPacket = 1024;
} else {
    outputFileType = kAudioFileWAVEType;
    outputDescription.mFormatID = kAudioFormatLinearPCM;
    outputDescription.mFormatFlags = kAudioFormatFlagIsFloat | kAudioFormatFlagIsPac
    outputDescription.mBytesPerPacket = 4 * inputDescription.mChannelsPerFrame;
    outputDescription.mFramesPerPacket = 1;
    outputDescription.mBytesPerFrame = 4 * inputDescription.mChannelsPerFrame;
```

```
    outputDescription.mBitsPerChannel = 32;
}
```

With the output specification, the sample creates the audio converter object that it uses for encoding or decoding the audio.

Provide a magic cookie

Some audio formats have a magic cookie that the converter requires to decompress audio data. A *magic cookie* refers to information in an audio file that describes its data format. An encoder produces the magic cookie, and it's included in the same file or stream as the audio packets, in a header, or a container file format through a box.

When decoding audio, the sample checks if the format of the audio data that the framework is converting has a magic cookie. If the audio data format has a magic cookie, the sample adds the information to the audio converter instance before converting the audio.

```
std::optional<size_t> magicCookieSize =
    inputFile.GetPropertySize(kAudioFilePropertyMagicCookieData);
if (magicCookieSize.has_value()) {
    std::cout << "The magic cookie is "
              << *magicCookieSize
              << " bytes in size."
              << std::endl;

    // Get the magic cookie from the input file.
    std::vector<uint8_t> magicCookie(*magicCookieSize);
    inputFile.GetProperty(
        kAudioFilePropertyMagicCookieData,
        magicCookie.size(),
        magicCookie.data());

    // Provide the magic cookie to the decoder, via the AudioConverter.
    audioConverter.SetProperty(kAudioConverterDecompressionMagicCookie,
        magicCookie.size(),
        magicCookie.data());
} else {
    std::cout << "There is no magic cookie." << std::endl;
}
```

When encoding an audio file, the sample adds the magic cookie back to the file — after converting the audio — because it could change during the encoding process. Get the magic cookie by using

the property `kAudioConverterCompressionMagicCookie` and then add it to the output file.

Convert the audio

After the sample creates the output description, and sets the magic cookie if it's decoding, it converts the audio by looping through the available data. The loop begins by trying to decode the available packets.

```
// Try to handle more packets depending on whether the sample is encoding or decodir
UInt32 numPackets = packetsPerLoop;
AudioBufferList abl{ 1, {
                            outputDescription.mChannelsPerFrame, // mNumberChannels
                            (UInt32)packetBuffer.size(),         // mDataByteSize
                            packetBuffer.data()                  // mData
                        } };
audioConverter.FillComplexBuffer(InputContext::InputDataProc,
                                 &inputContext,
                                 numPackets,
                                 abl,
                                 encode ? packetDescriptions.data() : NULL);
```

When looping through the available packets, the sample reads packets from the input file through an input callback that wraps `AudioFileReadPacketData(_:_:_:_:_:_:)`. It's important that `ioNumberDataPackets` and `mDataByteSize` in the buffers remain consistent. If one changes, the other must match. If the audio format calls for packet descriptions, they should also be consistent with `ioNumberDataPackets` and `mDataByteSize`.

```
// Read packets from the file into the buffer, along with packet descriptions, if ar
try {
    self.mInputFile.ReadPackets(ioData->mBuffers[0].mDataByteSize,
                                self.mInputUsesPacketDescriptions ? self.mPacketDesc
                                *ioNumberDataPackets,
                                ioData->mBuffers[0].mData);
} catch (AudioToolboxError err) {
    std::cerr << "Encountered an error while reading packets: " << err.what() << std
    return err.status;
}
```

If the sample receives output packets, it writes them to an output file in the project build folder.

```
// If there are output packets, write them to the output file.
if (numPackets > 0) {
    outputFile.WritePackets(abl.mBuffers[0].mDataByteSize,
                            encode ? packetDescriptions.data() : NULL,
                            numPackets,
                            abl.mBuffers[0].mData);
}
```

The loop stops when there aren't enough packets to satisfy the input request. There are two additional cases to consider when running out of data.

First, there's no data currently available from the input stream, but data remains to convert. For example, when streaming input data in real time, the converter may not reach the end of the stream because it's waiting to receive the next input packet. In this case, set `ioNumberDataPackets` to zero and return a custom error code. The error propagates to the call `AudioConverterFillComplexBuffer(_:_:_:_:_:)`, which makes it distinguishable from other errors in the conversion process, such as errors that indicate there's no data currently available. The caller receives any remaining data the converter processes.

Second, the conversion reaches the end of the stream. In this case, set `ioNumberDataPackets` and `mDataByteSize` to zero and return `noErr`. The audio converter may call the input procedure a few times, so return zero and `noErr`.

See Also

Performing Conversions

`func AudioConverterConvertBuffer(AudioConverterRef, UInt32, UnsafeRawPointer, UnsafeMutablePointer<UInt32>, UnsafeMutableRawPointer) -> OSStatus`

Converts audio data from one linear PCM format to another.

`func AudioConverterFillComplexBuffer(AudioConverterRef, AudioConverterComplexInputDataProc, UnsafeMutableRawPointer?, UnsafeMutablePointer<UInt32>, UnsafeMutablePointer<AudioBufferList>, UnsafeMutablePointer<AudioStreamPacketDescription>?) -> OSStatus`

Converts audio data supplied by a callback function, supporting non-interleaved and packetized formats.

```
func AudioConverterConvertComplexBuffer(AudioConverterRef, UInt32,
UnsafePointer<AudioBufferList>, UnsafeMutablePointer<AudioBufferList>)
-> OSStatus
```

Converts audio data from one linear PCM format to another, where both use the same sample rate.