Ch. 1 Overview of Core Audio

- engine behind sound on mac or iOS
  - C API
  - exposed in Objective-C, C++, and Swift
- high-level when compared to general audio programming environments
- low-level from an application developer standpoint
- very different from Cocoa and other C-based APIs (Quartz, Core Foundation)
- when not to use
  - simple file playback
  - use AppKit and NSSound or AVAudioPlayer from AV Foundation
  - can also use QTKit on mac

The Core Audio Frameworks

- two groups
  - audio engines - audio stream processing
  - helper APIs - facilitate audio IO to and from these frameworks
- mac and iOS audio engine APIs
  - Audio Units
    - Core Audio passes audio buffers to graphs of Audio Units which do work on the buffers
  - Audio Queues
    - abstraction on top of audio units that simplifies playback and recording
    - uses callbacks
  - OpenAL
    - industry standard 3D audio API (designed to resemble OpenGL)
- helper APIs
  - Audio File Services
    - abstracts access to various file types
  - Audio File Stream Services
    - network and other stream sources of audio
  - Audio Converter Services
    - format conversion
  - Extended Audio File Services
    - allows simultaneous reading and writing and conversion
  - Core MIDI
    - MIDI access

        - Audio Session Services (iOS-only)
            - coordinates use of resources with system on iOS

Core Audio Conventions

- C function calls
    - brings a number of difficulties that modern programmers may not be equipped to deal with
- does mimic object orientation within C in many ways
- Core Foundation is the main C framework and underlies Foundation
    - many of the Objective-C components of Core Foundation gain their functionality by calling C
- Core Audio is very similar in design to Core Foundation

Your First Core Audio Application

- uses Objective-C in Xcode
- see code for details
- notes
    - need to convert from C strings to NSString/CFStringRef
        - can specify UTF-8
        - can use stringByExpandingTildeInPath
    - must convert file paths to NSURL to use in audio file APIs
    - use local variable reference of type AudioFileID type to refer to audio file objects
    - use OSStatus to signal failure or success through return types
    - should check on all Core Audio calls
    - noErr (0) signals success
    - calls AudioFileOpenURL to open file
    - in Objective-C, NSURL can be toll-free cast to CFURLRef
    - to get metadata, ask for kAudioFilePropertyInfoDictionary
        - requires advance allocation of memory for returned metadata
    - to get size for allocation, call AudioFileGetPropertyInfo with AudioFileID
    - refer to docs and headers for info on the various results of getting properties
    - call AudioFileGetProperty
    - Core Foundation does not offer autorelease so retain/release counts must be managed manually
    - AudioFileID has its own clean-up call, AudioFileClose
    - many Core Audio objects have custom clean-up calls, be aware of these
    - check return codes on Core Audio clean-up calls

Running the Example

- requires linking in frameworks in Xcode
- requires passing command-line arguments
- NOTE: the metadata results will vary wildly depending on files/file types/systems
- NOTE: Core Audio requires an adherence to a large number of implicit contracts between various types of calls
    - BE PATIENT AND WORK WITH THE CODE

Core Audio Properties

- preparing and calling property getters and setters in Core Audio is an essential process
- properties
    - key-value pairs
    - keys are enumerated integers
    - values are types defined by APIs
    - values that are retrieved or set are dependent on the specific property
- for AudioFileGetProperty, the keys are 32-bit integers that can be represented as four character codes
    - NOTE: use single-quotes to represent char literals in C
- many Core Audio types have related *GetProperty() functions
- function signature adds information based on naming conventions of function parameters
    - in - only used for input to the function
    - out - only used for output from the function
    - io - used for input and output from a function

Ch. 2 The Story of Sound

Making Waves

Digital Audio

DIY Samples

- need to set
    - sample rate
    - length of audio sample to create

    - filename/path of output file
    - file format
    - frequency of audio signal (as command-line arg)
    - generate full filename from file format
- need to populate info in AudioStreamBasicDescription struct
    - channels
    - format
    - bitrate
    - etc
- initialize remaining fields of AudioStreamBasicDescription to 0 (using memset in C)
    - Core Audio populates certain fields depending on format
- set info in AudioStreamBasicDescription
    - single channel
    - PCM
    - 44100
    - 16-bit
    - 2 bytes per frame (one channel * 2 bytes of sample data)
    - endianness (for PCM)
    - numeric format of samples (kAudioFormatFlagIsSignedInteger)
    - packed (sample uses all bits available in each byte) vs unpacked
- packets are only useful for VBR formats
    - bytesPerFrame and bytesPerPacket are equal for CBR formats
- AIFF only handles big endian PCM
- numeric format and packed vs unpacked are flags that must be bitwise or'ed together
- create AudioFileID for audio file using AudioFileCreateWithURL
    - toll free cast between NSURL and CFURLRef
- write samples in loop
    - number of samples to write per loop
    - number of samples in each wave
- must be careful to handle need for swapping endianness
    - macs are on Intel little-endian CPUs
    - ARM on iPhone is little-endian
    - use CFSwapInt16HostToBig()
- use AudioFileWriteBytes() to write (for CBR formats only)
    - AudioFileID
    - caching flag
    - offset in output to write to
    - number of bytes being written
- must use AudioFileWritePackets() for VBR formats

- shows how to rewrite for square, triangle and sawtooth waves
- iOS 4 only handles integers (not floats) for PCM

Buffers

- the time it takes for the audio hardware to produce or consume a packet of audio data is much less than the time it takes to move the packets in and out of memory
- this slowdown is called the von Neumann bottleneck
- use buffers to move data between sources and sinks
- buffering adds complexity and delays audio hardware responsiveness
  - latency
- buffers and latency require a balancing act
  - big buffers cause higher latency
  - lower latency can cause premature buffer drain which results in silence (dropouts) or noise

Audio Formats

- audio data formats and file formats are distinct considerations
  - mp3
  - AIFF handles big-endian PCM
  - WAV handles little-endian PCM
  - MP4 can handle AAC, PCM, and AC3
  - .caf (Core Audio Format) is most content agnostic format
    - any audio format supported by Core Audio
    - mp3
    - AAC
    - Apple Lossless
    - etc
    - good for internal format for an application
    - improves performance

Ch. 3 Audio Processing with Core Audio

Audio Data Formats

- Core Audio views audio data as streams of packets
- AudioStreamBasicDescription is critical to most Core Audio programs

```
struct AudioStreamBasicDescription {
    Float64 mSampleRate;
    UInt32 mFormatID;
    UInt32 mFormatFlags;
    UInt32 mBytesPerPacket;
    UInt32 mFramesPerPacket;
    UInt32 mBytesPerFrame;
    UInt32 mChannelsPerFrame;
    UInt32 mBitsPerChannel;
    UInt32 mReserved;
};
typedef struct AudioStreamBasicDescription AudioStreamBasicDescription;
```

- mSampleRate is defined to be the number of samples per channel per second of uncompressed data
    - makes it equal to the number of frames per second, which means can multiply it by mFramesPerPacket to determine packet length in seconds
- mFormatID is a four-character code that serves as the name of the format
    - without naming a format, the struct is meaningless, so this value must be defined
    - default types Core Audio supports are defined in CoreAudioTypes.h
    - contains the four character codes for the default formats and constants that can be used
    - for example, the kAudioFormatLinearPCM format constant used previously chapter is the four-character code 'lpcm'
- mFormatFlags is the format's subtype, or preference panel
    - UInt32 bit field in which set or check various 1-bit flags
    - setting flags answers questions left open by formats that support non-interleaved data, multiple sample types, or variable data structures
    - the only way to know how to interpret this value is to look it up the documentation — search for "AudioStreamBasicDescription Flags" to find the defined flags
    - formats that do not have flags can set this value to 0
- mBytesPerPacket sets details of the structure of the format's data
    - amount/size of data
    - NOTE: VBR formats, by definition, cannot answer this question
    - these formats set this value to 0 and use an AudioStreamPacketDescription
- mFramesPerPacket subdivides the raw bytes of the packet into some number of frames
    - only for compressed formats - uncompressed formats always set this to 1
    - Core Audio can also support formats with a variable frame rate
    - these formats set this value to 0 and use AudioStreamPacketDescriptions

- mBytesPerFrame establishes the size of a single frame
    - the total number of digits used to represent each moment in time
    - when a frame does not contain a sample per channel, as with compression, this value is set to 0
- mChannelsPerFrame subdivides the frame into channels, regardless of compression
    - this value cannot be 0 because an audio stream with no channels is, by definition, empty
- mBitsPerChannel is the sample's bit depth
    - as with bytes per frame, it represents the actual structure of the data
    - compressed formats whose frames do not contain a sample per channel set this value to 0
- mReserved member is for data alignment purposes, padding the structure to an even multiple of 8
    - it must always be 0
- the specifics of a particular ASBD are specific to the format being described, i.e. they are an implementation detail
    - do not compare specific fields between formats (unless pertinent)
- in many cases, Core Audio automatically populates the fields of an ASBD
- for compressed data, whose packet structure cannot be adequately derived from its measurements, each packet must be accompanied by an AudioStreamPacketDescription

```
struct AudioStreamPacketDescription {
    SInt64 mStartOffset;
    UInt32 mVariableFramesInPacket;
    UInt32 mDataByteSize;
};
typedef struct AudioStreamPacketDescription AudioStreamPacketDescription;
```

- mStartOffset represents the packet's location, relative to other packets in the buffer, in bytes
    - important because packets of different sizes cannot use an implied x-axis value as in a linear PCM format
    - cannot find a sample by doing Offset = Time × Sample rate × Frame size + Channel number

- mVariableFramesInPacket represents the number of frames in the packet, but only if the packets use a variable frame rate
  - if the AudioStreamBasicDescription has a value for mFramesPerPacket, mVariableFramesInPacket should be 0, and vice versa
- mDataByteSize contains the packet's actual size, in bytes
- many helper API's use additional magic cookies to describe audio data
  - Audio File Services
  - Audio File Stream Services
  - Audio Conversion Services
  - Audio Queue Services
  - more
- cookie
  - opaque block of data with contents specific to format being encoded
- when opening file or network stream of compressed data
  - check for magic cookie property
  - if present, read in as block of untyped data and pass to Core Audio without modifying or checking

Example: Figuring out Formats

- to use the property called kAudioFileGlobalInfo_AvailableStreamDescriptionsForFormat pass Core Audio a structure called AudioFileTypeAndFormatID
  - this structure has two members, a file type and a data format
  - can set both with Core Audio constants found in the documentation or the AudioFile.h and AudioFormat.h headers
- use an AIFF file with PCM to start
- prepare an OSStatus to receive result codes from your Core Audio calls
  - also prepare a UInt32 to hold the size of the info - have to negotiate before actually retrieving info
- getting a global info property requires a query in advance for the size of the property and to store the size in a pointer to a UInt32
- global info calls take a specifier, which acts like an argument to the property call and depends on the property requested
  - in the case of kAudioFileGlobalInfo_AvailableStreamDescriptionsForFormat provide the AudioFileTypeAndFormatID.

- AudioFileGetGlobalInfoSize() calls return amount of data to be received when when the global property is retrieved
  - malloc memory to hold the property
- call AudioFileGetGlobalInfo() to get the kAudioFileGlobalInfo_AvailableStreamDescriptionsForFormat
  - kAudioFileGlobalInfo_AvailableStreamDescriptionsForFormat
  - pass in the AudioFileTypeAndFormatID and the size of the buffer and a pointer to the buffer
- docs specify that the property call provides an array of AudioStreamBasicDescriptions,
  - can determine length of the array by dividing the data size by the size of an ASBD
  - enables a for loop to investigate the ASBDs
- docs stated that the three ASBD fields that get filled in are mFormatID, mFormatFlags, and mBitsPerChannel
  - useful to log the format ID
  - to make it legible, convert it out of the four-character code numeric format and into a readable four-character string
  - do this with an endian swap because the UInt32 representation will reorder the bits from their original pseudostring representation
  - to pretty print the mFormatId's endian-swapped representation can use the format string %4.4s to force NSLog (or printf) to treat the pointer as an array of 8-bit characters that is exactly four characters long
  - mFormatFlags and mBitsPerChannel members are a bit field and numeric value, so print them as ints
- free() the malloc()' d memory to hold the ASBD array when done with it
- book provides additional information about results of code for various formats here

Canonical Formats

- AAC good for audio fidelity (particularly for music) at high compression rates
- for maximum fidelity can use Apple Lossless which reproduces its source audio perfectly
- the iLBC (Internet Low-Bandwidth Codec) is optimized for speech over potentially unreliable Internet connections
  - makes it well suited for VoIP or in-game chat purposes
- none of above formats are appropriate for editing
- editing better served by the low CPU overhead and losslessness of PCM
  - must have enough RAM and disk space
- any value not specifically set in ASBD defaults to canonical format value
- each platform has two canonical formats
  - AudioSampleType
    - used for I/O situations

- AudioUnitSampleType
    - introduced in Snow Leopard and iOS
    - used in audio units and for digital signal processing
- on Mac OS X
  - AudioSampleType is 32-bit float
  - AudioUnitSampleType is also a 32-bit float
    - channels must be noninterleaved
- can find the mFormatFlags definitions in the CoreAudioTypes.h header file in the CoreAudio.framework

```
kAudioFormatFlagsCanonical = kAudioFormatFlagIsFloat
                                 | kAudioFormatFlagsNativeEndian
                                 | kAudioFormatFlagIsPacked
kAudioFormatFlagsAudioUnitCanonical = kAudioFormatFlagIsFloat
                                          | kAudioFormatFlagsNativeEndian
                                          | kAudioFormatFlagIsPacked
                                          | kAudioFormatFlagIsNonInterleaved
```

- on iOS
  - AudioSampleType is 16-bit integer
  - AudioUnitSampleType is an 8.24-bit fixed-point number
    - 8 bits to the left of the radix point and 24 bits to the right

```
kAudioFormatFlagsCanonical = kAudioFormatFlagIsSignedInteger
                                | kAudioFormatFlagsNativeEndian
                                | kAudioFormatFlagIsPacked
kAudioFormatFlagsAudioUnitCanonical = kAudioFormatFlagIsSignedInteger
                                          | kAudioFormatFlagsNativeEndian
                                          | kAudioFormatFlagIsPacked
                                          | kAudioFormatFlagIsNonInterleaved
                                          | (kAudioUnitSampleFractionBits << kLine
```

- when supplying samples directly to the audio engines (Audio Units and OpenAL) directly, it’s advantageous to use the canonical formats
  - saves some data conversions and allows CPU cycles to be used elsewhere

Processing Audio with Audio Units

- core Audio does most of its work at the Audio Units level
  - Audio Queue and OpenAL engines are implemented atop audio units
- each audio unit processes a buffer of samples in some specific way
  - for example, one captures audio from a mic
  - next one downstream may perform an effect on those samples
  - next one may mix it with another source

- analogous to the patch architecture common in the sound industry
- Audio units are connected by audio streams
  - the way audio equipment is patched together with cables
  - audio units have elements, which may have input scope or output scope to indicate that they either accept or produce data
- to connect two units, you set a property connecting an input element of one unit to the output of another
- types of audio units
  - effect units
    - perform digital signal processing (change the audio data in some way)
    - analogous to hardware effects boxes and outboard signal processors
  - instrument units
    - generate audio data representing musical notes, typically from MIDI input, which can itself come from a musical instrument or a software synthesizer
  - generator units
    - also generate audio data, but not from a midi source
    - some units simply generate a signal programmatically, whereas others load data from a network stream or audio file
  - i/ o units
    - provide interfaces to input or output hardware, such as a microphone or a speaker
    - these are typically implemented on the hardware abstraction layer (HAL)
      - intermediate layer between Core Audio and I/O Kit and the drivers
  - converter units
    - reformat audio data back and forth between the canonical formats and other formats
    - these can also merge and split streams, and alter timing and pitch
  - mixer units
    - combine audio tracks
    - there are also splitter units that provide multiple outputs from a single input
  - panner units
    - use stereo mixing to create panning effects
  - offline effect units
    - perform operations on audio data that cannot be done in real time.

The Pull Model

- framework dictates when audio will be provided to be processed rather than programmer
- in a chain of audio units, the last one (probably the I/O unit that will send audio to the speakers or headphones) pulls from the units connected to its input elements
  - each unit call the units upstream from it
- callback = proc in Core Audio

- read proc - callback that receives data
- write proc - callback that produces data
- set a callback property on the audio unit

Ch. 4 Recording

- Audio Queue Services is the highest level playback and recording API in Core Audio
- conveniences
  - unlike with OpenAL and Audio Units, can use encoded formats such as AAC and MP3 with audio queues
  - by default, Audio Queues call back on their own thread, which isolates application programmer from some timing challenges with Audio Units

All About Audio Queues

- simplifies audio playback and recording
  - reduces complexity of audio data management and codecs
  - reduces complexity of underlying hardware interaction
- audio queue is a software interface to piece of hardware
  - transducer (speaker or microphone)
- queue is filled with data and passed to callback which sends audio data somewhere else and places buffer at the back of the queue
- see Figure 4.1 for graphic

Building a Recorder

- audio queue usage is more akin to building a recorder than using a recorder
- provide a callback function that will be called to provide application with buffers of audio captured from the input device (in the recording case) or to fill a buffer for playback
- AudioQueueNewInput() takes the following
  - format to record to
  - structure representing a callback function
  - pointer to "user data," which is provided to the callback
  - Core Foundation run loop to use for the callbacks
  - Core Foundation run loop "mode" for callbacks
  - "Flags" that must be set to 0
  - pointer to receive a newly created AudioQueueRef

A CheckError() Function

- see modified check_error() function

Creating and Using the Audio Queue

- create a user info struct for recording Audio Queue callbacks
    - similar to creating a class to encapsulate application logic in C++, etc
- create custom struct and ASBD in main function
- set format of ASBD for Audio Queue
    - mFormatID - kAudioFormatMPEG4AAC
    - mChannelsPerFrame - 2
- add convenience function to determine correct sample rate for selected input device
- for formats other than PCM, fill in properties of ASBD that can be filled in and allow Core Audio to handle the rest
- check error on output value
- use AudioQueueNewInput() to create the audio queue
- after creating the queue, can retrieve a more complete ASBD with AudioQueueGetProperty and kAudioConverterCurrentOutputStreamDescription
- use info to create audio file to write captured data to with CFURLCreateWithFileSystemPath ()
- queue returns magic cookie (AAC is a format that uses magic cookies)
    - retrieve from audio queue and set on audio file
    - add convenience function to handle this
- with PCM can calculate buffer sizes to use but cannot with a compressed format
    - 44100 samples/ second × 2 channels × 2 bytes/channel × 1 seconds
        - requires 176400 bytes to hold a second of 16-bit stereo PCM at 44.1 KHz
- create convenience function to retrieve buffer size from audio queue
- common practice in Core Audio to use 3 buffers
    - one is filled
    - one is drained
    - one is in buffer queue as a spare to account for potential lag
- allocate and enqueue buffers in set up
    - AudioQueueAllocateBuffer()
    - AudioQueueEnqueueBuffer()
- start the audio queue with AudioQueueStart()
- handle user stop
- call AudioQueueStop() to stop audio queue
- call magic cookie convenience function for output to file

- clean up audio queue and audio file with AudioQueueDispose() and AudioFileClose()

Utility Functions for the Audio Queue

- get default device sample rate
    - set up
        - AudioDeviceID
        - AudioObjectPropertyAddress
            - mSelector - kAudioHardwarePropertyDefaultInputDevice
            - mScope - kAudioObjectPropertyScopeGlobal
            - mElement - 0
    - call AudioHardwareServiceGetPropertyData()
    - get the returned input device's sample rate using the property address and calling AudioHardwareServiceGetPropertyData
- copying magic cookie from audio queue to audio file
    - get property size of kAudioConverterCompressionMagicCookie with AudioQueueGetPropertySize
    - get property (kAudioQueueProperty_MagicCookie) with AudioQueueGetProperty()
    - set property (kAudioFilePropertyMagicCookieData) with AudioFileSetProperty() on the audio file
    - if size is 0, no cookie exists and can exit, otherwise need to allocate a byte buffer to hold the cookie
- computing recording buffer size for an ASBD
    - first retrieve number of frames (one sample for every channel) in each buffer
        - get this by multiplying the sample rate by the buffer duration
        - if the ASBD already has an mBytesPerFrame value, as in the case for constant bit rate formats such as PCM, can trivially get the needed byte count by multiplying mBytesPerFrame by the frame count.
    - if the ASBD already has an mBytesPerFrame value, must work at the packet level
        - easy case for this is a constant packet size, indicated by a nonzero mBytesPerPacket
        - for hard case, get the audio queue property kAudioConverterPropertyMaximumOutputPacketSize, which gives an upper bound
        - in both cases, maxPacketSize will have been retrieved as a result
    - ASBD might provide a mFramesPerPacket value
        - in this case, to determine number of packets, divide the frame count by mFramesPerPacket to get a packet count (packets).
        - otherwise, assume the worst case of one frame per packet
    - with a frames-per-packet value (which is forced to nonzero for safety) and a maximum size per packet, can multiply the two to get a maximum buffer size

The Recording Audio Queue Callback

- called every time the queue fills one of the buffers with freshly captured audio data
- callback function performs data processing with the data
- start by casting user info void * back to proper user info type
- for AudioFileWritePackets(), need
  - a file (in the user data struct)
  - a boolean indicating whether to cache the data written
    - false in this case
  - the size of the data buffer to write retrieved from the inBuffer parameter's mAudioDataByteSize
  - packet descriptions, provided by the callback's inPacketDesc parameter
  - an index to which packet in the file to write, which is a running count tracked in recorder's recordPacket field
  - number of packets to write, provided by the callback's inNumPackets parameter
  - pointer to the audio data, which is the inBuffer mAudioData pointer
- re-enqueue used buffers

Ch. 5 Playback

- playback requires another form of audio queue

Defining the Playback Application

- create a playback audio queue with AudioQueueNewOutput()
  - AudioStreamBasicDescription
    - describing the audio format being provided to the queue
  - AudioQueueOutputCallback
    - function pointer to a callback function you will write
  - user data pointer to provide to the callback
  - Core Foundation run loop on which to call the callback
  - Core Foundation run loop mode for the callback
  - unused flags parameter that must always be 0
  - pointer to receive the created AudioQueueRef
- function pointer to the callback
  - user data pointer
  - queue that is performing the callback
  - AudioQueueBufferRef to fill with data

Setting Up a File-Playing Audio Queue

- callback gets void * to user data struct
  - reads from audio file and passes packets to the queue
- types in user info struct
  - AudioFileID playbackFile
  - SInt64 packetPosition
  - UInt32 numPacketsToRead
  - Boolean isDone
- define var for the file to read (CFSTR)
- use CFURLCreateWithFileSystemPath()
- set up adn retrieve ASBD using AudioFileGetProperty() and kAudioFilePropertyDataFormat
- create new Audio Queue for output using AudioQueueNewOutput()
- packet - collection of frames which in turn are a collection of samples
- setting up the playback buffers
  - requires working with packets
  - for VBR (mp3 and AAC), use an array of AudioStreamPacketDescriptions to provide a map of the contents of the audio buffer
  - call CalculateBytesForTime() to calculate the playback buffer size and number of packets to read
  - be sure to allocate proper size for packet descriptions array using malloc
  - create convenience function to handle the magic cookie
- see code example for allocating and enqueueing the playback buffers
- call AudioQueueStart() to start the queue playback and then CFRunLoopRunInMode() with kCFRunLoopDefaultMode
  - add delay to ensure queue plays out buffered audio
- call AudioQueueStop(), AudioQueueDispose() and AudioFileClose() to clean-up

Playback Utility Functions

- handling the magic cookie
  - see previous chapter and code example
- calculating buffer size and expected packet count
  - get the maximum packet size for the file's encoding type, which is available from the audio file property kAudioFilePropertyPacketSizeUpperBound
  - set up two constants as fail-safe values
    - maxBufferSize of 64 KB
    - minBufferSize of 16 KB

    - if the ASBD tells how many frames are in a packet
        - calculate how many packets elapse in the given number of seconds and multiply that by the maximum packet size to get a sufficiently large buffer
    - if the ASBD does not tell how many frames are in a packet (no mFramesPerPacket value)
        - select an arbitrarily "large enough" value, which is the greater of maxBufferSize and maxPacketSize
    - in the worst case, at least the buffer will be large enough to hold one packet
    - second if-else applies boundary checks
    - if the calculated buffer size (outBufferSize) is larger than both the maxBufferSize and the maxPacketSize, clamp it to the maxBufferSize
        - also check to see that it's not smaller than the minBufferSize
    - with an outBufferSize calculated
        - divide it by the maxPacketSize to figure out how many packets can be safely read from the file on each callback

The Playback Audio Queue Callback

- cast back user info void *
- use AudioFileReadPackets()
    - file to read from
    - cache flag
    - pointer to receive the number of bytes actually read
    - pointer to a buffer to hold packet descriptions
    - index of the first packet you want to read
    - pointer to a maximum number of packets to read (which will be replaced by the number of packets actually read when the function returns)
    - pointer to a buffer to receive the audio data
- use AudioFileReadPackets() to read packets
- use AudioFileEnqueueBuffer() to enqueue packets
- use AudioQueueStop() when the end of file is reached

Features and Limits of Queue-Based Playback

- level-metering
    - kAudioQueueProperty_EnableLevelMetering
    - kAudioQueueProperty_CurrentLevelMeter
    - kAudioQueueProperty_CurrentLevelMeterDB

- properties vs. parameters
    - properties are used for set-up and initialization of an audio object and have values of any type
    - parameters represent values that may be of interest to end user and may change during use
        - always floating point numbers
        - kAudioQueueParam_Volume for instance
- benefits
    - quiet handling of decompression from encoded formats such as AAC to PCM stream
    - eliminates threading concerns (callbacks are by default performed by one of queue's internal threads)
- drawbacks
    - each buffer adds latency (.5s for each in this example, 1.5s total)
    - user will not hear read sample until all preceding samples are played out
    - using smaller buffers leads to more callback calls, thus more call overhead, etc
    - inherent latency to queues

Ch. 6 Conversion

The afconvert Utility

- use `--help` to see info detailed here

Using Audio Converter Services

- prefixed with AudioConverter (can use for documentation lookup)
- setup user info struct
    - AudioStreamBasicDescription inputFormat
    - AudioStreamBasicDescription outputFormat
    - AudioFileID inputFile
    - AudioFileID outputFile
    - UInt64 inputFilePacketIndex
    - UInt64 inputFilePacketCount
    - UInt32 inputFilePacketMaxSize;

Setting Up Files for Conversion

- define file path
- create user info struct

- open file with CFURLCreateWithFileSystemPath()
- call AudioFileOpenURL() and check error
- call CFRelease() on input file URL
- get ASBD from input audio file using AudioFileGetProperty()
- get packet count and maximum packet size properties from the input audio file
- define output ASBD
    - mSampleRate - 44100.0
    - mFormatID - kAudioFormatLinearPCM
    - mFormatFlags - kAudioFormatFlagIsBigEndian | kAudioFormatFlagIsSignedInteger | kAudioFormatFlagIsPacked
    - mBytesPerPacket - 4
    - mFramesPerPacket - 1
    - mBytesPerFrame - 4
    - mChannelsPerFrame - 2
    - mBitsPerChannel - 16
- create output file with CFURLCreateWithFileSystemPath()
- call the custom conversion function
- close the files with AudioFileClose()
- conversion function
    - create AudioConverterRef using AudioConverterNew()
    - determining the size of a packet buffers array and packets-per-buffer count for variable bit rate data
        - see code
    - determine packets per buffer for CBR data using output buffer size / size per packet
    - allocate memory for the audio conversion buffer
    - loop to convert and write data
    - prepare an AudioBufferList to receive converted data
    - call AudioConverterFillComplexBuffer()
    - write packets of converted data to output file using AudioFileWritePackets()
    - call AudioConverterDispose() to clean-up
- callback function
    - cast user data struct
    - zero audio buffers

    - determine number of packets that can be read from input
    - allocate a buffer to fill and convert
    - read packets into the conversion buffer
    - update the source file position and AudioBuffer members with the results of the read

Converting with Extended Audio File Services

- extended services combine Audio FIle Services and Audio Converter Services into a simplified API
- example follows same general outline as previous example
- user info struct
    - AudioStreamBasicDescription outputFormat
    - ExtAudioFileRef inputFile
    - AudioFileID outputFile
- call ExtAudioFileOpenURL() to open input file
- create ASBD for output format and create audio file
- set the client data format property on the input file using ExtAudioFileSetProperty()
- call the conversion function and close the file with ExtAudioFileDispose() and AudioFileClose ()
- to read and convert for extended audio files
    - determine the size of the output buffer and packets-per-buffer count
    - allocate a buffer for receiving data from an extended audio file
    - run the read-convert-write loop after setting up an AudioBuffer
        - use ExtAudioFileRead() to read
        - terminate if no frames are read
        - use AudioFileWritePackets() to write
        - advance the write position

Ch. 7 Audio Units: Generators, Effects, and Rendering

Where the Magic Happens

- Audio Units is lowest level of audio functionality that application developers need
- enables functionality that is impractical or impossible with the higher-level APIs
- increases difficulty of use and potential for serious error

How Audio Units Work

- works on streams of audio
- set up units, create connections, tell them to start processing
- analogous to the setup and flow of audio hardware
- types of audio units
    - generator units
        - create a stream of audio from some source, such as files, the network, or memory
    - instrument units
        - similar to generator units, produce a stream of synthesized audio from MIDI data
    - mixer units
        - combine multiple streams into one or more streams. the mix can be performed in two dimensions (for stereo panning) or in a simulated three-dimensional sound field
    - effect units
        - perform some sort of digital signal processing on a stream, usually producing an audible effect such as a reverb, a pitch change, noise filtering, and so on
    - converter units
        - perform transformations that are generally not meant to deliver user-audible effects
        - includes units to convert between different flavors of pcm (to change sample rate or bit depth, for example), adjust playback speed, and so on
    - output units
        - interface with audio input and/ or output hardware, enabling you to capture audio from input devices and play it out to output devices
        - name is misleading because these are potentially input/output units
- works with a pull model - software objects that need data pull samples from other objects using callbacks

Sizing Up the Audio Units

- type, subtype, and manufacturer are in AUComponent.h

Audio Unit Subtypes for Generator Units (Type kAudioUnitType_Generator)

- kAudioUnitSubType_ScheduledSoundPlayer
    - schedules audio to be played at a specific time
- kAudioUnitSubType_AudioFilePlayer
    - plays audio from a file
- kAudioUnitSubType_NetReceive
    - receives network audio from a corresponding kAudioUnitSubType_NetSend unit on another host or in another application

Audio Unit Subtypes for Instrument Units (Type kAudioUnitType_MusicDevice)

- kAudioUnitSubType_DLSSynth
  - multi-timbral music synthesizer that accepts MIDI commands
  - works with DLS or SoundFont formats

Audio Unit Subtypes for Mixer Units (Type kAudioUnitType_Mixer)

- kAudioUnitSubType_MultiChannelMixer
  - mixes any number of single or multi-channel input buses to one output bus
- kAudioUnitSubType_StereoMixer
  - mixes any number of mono or stereo input buses to one stereo output
- kAudioUnitSubType_3DMixer
  - mixes any number of mono or stereo input buses
  - mono inputs can be panned with 3D parameters
  - output is one bus of 2 to 8 channels
- kAudioUnitSubType_MatrixMixer
  - mixes any number of input and output buses, with any number of channels per bus
  - supports highly configurable mapping of inputs and outputs

Audio Unit Subtypes for Panner Units (Type kAudioUnitType_Panner)

- kAudioUnitSubType_SphericalHeadPanner
  - uses "spherical head" model to produce stereo output
- kAudioUnitSubType_VectorPanner
  - uses pan between adjacent channels in 3D space to create surround output
- kAudioUnitSubType_SoundFilePanner
  - uses "sound field" model to produce stereo output
- kAudioUnitSubType_HRTFPanner
  - uses head-related transfer function to produce stereo output

Audio Unit Subtypes for Effect Units (Type kAudioUnitType_Effect)

- kAudioUnitSubType_Delay
  - digital delay effect
- kAudioUnitSubType_LowPassFilter
- kAudioUnitSubType_HighPassFilter
- kAudioUnitSubType_BandPassFilter
- kAudioUnitSubType_HighShelfFilter
- kAudioUnitSubType_LowShelfFilter
- kAudioUnitSubType_ParametricEQ
- kAudioUnitSubType_GraphicEQ
- kAudioUnitSubType_PeakLimiter

- kAudioUnitSubType_DynamicsProcessor
- kAudioUnitSubType_MultiBandCompressor
- kAudioUnitSubType_MatrixReverb
- kAudioUnitSubType_SampleDelay
- kAudioUnitSubType_Pitch
- kAudioUnitSubType_AUFilter
    - adjusts gain for five bands of frequencies
- kAudioUnitSubType_NetSend
    - sends audio across network or between applications
- kAudioUnitSubType_Distortion
- kAudioUnitSubType_RogerBeep
    - produces a beep, similar to sound of walkie-talkie release, when the input level drops below a certain threshold for a certain amount of time

Audio Unit Subtypes for Converter Units (Type kAudioUnitType_FormatConverter)

- kAudioUnitSubType_AUConverter
    - uses audio converter services to perform LPCM conversions
- kAudioUnitSubType_Varispeed
    - changes playback speed, pitch-shifts audio as a result
    - faster = higher pitch
- kAudioUnitSubType_TimePitch
    - similar to varispeed
    - allows for independent control of playback speed and pitch
    - can play faster without changing pitch
- kAudioUnitSubType_DeferredRenderer
    - pulls input from a thread other than the caller
- kAudioUnitSubType_Splitter
    - splits one input bus into two identical output buses
- kAudioUnitSubType_Merger
    - merges two input buses into one output bus

Audio Unit Subtypes for Output Units (Type kAudioUnitType_Output)

- kAudioUnitSubType_GenericOutput
    - output unit not tied to audio hardware
    - can be used to perform software rendering of audio in connected units
- kAudioUnitSubType_SystemOutput
    - output to the device used for alerts and other UI sounds
- kAudioUnitSubType_DefaultOutput
    - output to the device selected in System Preferences -> Sound

- kAudioUnitSubType_HALOutput
  - input from and output to any supported audio device