Audio Queue Services

Dave Dribin – @ddribin

BitMaki Logomarks

Color: Deep Green

Additional Colors:
- 92 Cyan
- 42 Magenta
- 75 Yellow
- 44 Black
- Pantone 3435 #0c3124
- Bright Green
- 45 Cyan
- 0 Magenta
- 100 Yellow
- 0 Black
- Pantone 376 #8cc919
Playing Audio on iPhone OS
Playing Audio on iPhone OS

- Media Player Framework
Playing Audio on iPhone OS

- Media Player Framework

- AV Foundation Framework
Playing Audio on iPhone OS

- Media Player Framework
- AV Foundation Framework
- OpenAL Library
Playing Audio on iPhone OS

- Media Player Framework
- AV Foundation Framework
- OpenAL Library
- Core Audio Framework
Playing Audio on iPhone OS

• Media Player Framework

• AV Foundation Framework

• OpenAL Library

• Core Audio Framework

• Audio Toolbox Framework
Playing Audio on iPhone OS

- Media Player Framework
- AV Foundation Framework
- OpenAL Library
- Core Audio Framework
  - Audio Toolbox Framework
  - Audio Unit Framework
Playing Audio on iPhone OS

- Media Player Framework
- AV Foundation Framework
- OpenAL Library
- Audio Toolbox Framework
- Audio Unit Framework
Final Application Demo
Crash Course in Digital Audio
Analog Audio - A.K.A Sound
Analog Audio - A.K.A Sound
Digital Audio
Digital Audio Glossary

- Format – Example: Linear Pulse Code Modulation (Linear PCM)

- Sampling Rate – Example: 44,100 Hz

- Sampling Resolution – Example: 16-bit signed integer

- Channels – Example: 2 (for stereo)
CD Audio

- Linear PCM
- 44.1 kHz sampling rate
- 16-bit signed integer
- 2 Channels (Stereo)
Digital Telephony

- μ-law PCM
- 8 kHz Sampling Rate
- 8-bit unsigned integer
- 1 channel (monaural)
Audio Compression

- Lossless
  - WAV, AIFF (1:1)
  - FLAC, Apple Lossless (2:1)

- Lossy
  - MP3, AAC (10:1)
Audio Queue Services “Hello World”

- Play 440 Hz sound

- A440 Project on BitBucket:
  - [http://bitbucket.org/ddribin/a440](http://bitbucket.org/ddribin/a440)
Demo
What is an Audio Queue
1. Priming the audio queue: Application invokes callback to fill buffers sequentially with data.

2. Application tells audio queue to play.

3. Audio queue plays first buffer that was filled.

4. Audio queue returns previous buffer for reuse and plays next buffer that was filled.

5. Audio queue invokes callback and tells it “here’s the buffer that I want you to fill”.

6. Callback fills buffer, then adds it to the buffer queue.

Steady state: Continues until the file being played is finished. Callback then tells audio queue to stop.
Creating an Audio Queue for Output

```c
OSStatus AudioQueueNewOutput(
    const AudioStreamBasicDescription *inFormat,
    AudioQueueOutputCallback inCallbackProc,
    void *inUserData,
    CFRunLoopRef inCallbackRunLoop,
    CFStringRef inCallbackRunLoopMode,
    UInt32 inFlags,
    AudioQueueRef *outAQ);
```
AudioStreamBasicDescription (ASBDs)

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

typedef struct AudioStreamBasicDescription AudioStreamBasicDescription;
16-bits per Sample, Stereo Byte Stream

<table>
<thead>
<tr>
<th>Left Channel</th>
<th>Right Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x83</td>
<td>0x74</td>
</tr>
<tr>
<td>0x6F</td>
<td>0x72</td>
</tr>
<tr>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>0x2F</td>
<td>0x55</td>
</tr>
<tr>
<td>0x61</td>
<td>0x42</td>
</tr>
<tr>
<td>0x13</td>
<td>0xDF</td>
</tr>
<tr>
<td>0xA2</td>
<td></td>
</tr>
</tbody>
</table>

Byte

One Sample

One Frame
16-bits per Sample, Stereo Byte Stream

One Sample

One Frame

Left Channel

Right Channel

Byte

0x83 0x74 0x6F 0x72 0x00 0x2F 0x55 0x61 0x42 0x13 0xDF 0xA2
16-bits per Sample, Stereo Byte Stream

- Left Channel
- Right Channel

Byte

One Sample

One Frame

<table>
<thead>
<tr>
<th>0x83</th>
<th>0x74</th>
<th>0x6F</th>
<th>0x72</th>
<th>0x00</th>
<th>0x2F</th>
<th>0x55</th>
<th>0x61</th>
<th>0x42</th>
<th>0x13</th>
<th>0xDF</th>
<th>0xA2</th>
</tr>
</thead>
</table>
Audio Queue Sine Player

@interface A440AudioQueue : NSObject <A440Player>
{
    AudioQueueRef _queue;
    AudioStreamBasicDescription _dataFormat;
    AudioQueueBufferRef _buffers[3];
    A440SineWaveGenerator _sineWaveGenerator;
    BOOL _shouldBufferDataInCallback;
}

- (BOOL)play:(NSError **)error;
- (BOOL)stop:(NSError **)error;
@end
- (BOOL)play:(NSError **)error {
    NSAssert(_queue == NULL, @"Queue is already setup");

    OSStatus status;

    [self setupDataFormat];
    FAIL_ON_ERR(AudioQueueNewOutput(&_dataFormat,
    HandleOutputBuffer, self,
    CFRunLoopGetCurrent(),
    kCFRunLoopCommonModes, 0, &_queue));

    FAIL_ON_ERR([[self allocateBuffers]]);
    A440SineWaveGeneratorInitWithFrequency(&_sineWaveGenerator, 440.0);

    [self primeBuffers];
    FAIL_ON_ERR(AudioQueueStart(_queue, NULL));
    return YES;

failed:
    // Error handling....
    return NO;
- (void)setupDataFormat;
{
    // 16-bit native endian signed integer, stereo LPCM
    UInt32 formatFlags = (0
                        | kAudioFormatFlagIsPacked
                        | kAudioFormatFlagIsSignedInteger
                        | kAudioFormatFlagsNativeEndian);

    _dataFormat = (AudioStreamBasicDescription){
        .mFormatID = kAudioFormatLinearPCM,
        .mFormatFlags = formatFlags,
        .mSampleRate = SAMPLE_RATE,
        .mBitsPerChannel = 16,
        .mChannelsPerFrame = 2,
        .mBytesPerFrame = 4,
        .mFramesPerPacket = 1,
        .mBytesPerPacket = 4,
    };
}
- (BOOL)play:(NSError **)error{
    NSAssert(_queue == NULL, @"Queue is already setup");

    OSStatus status;

    [self setupDataFormat];
    FAIL_ON_ERR(AudioQueueNewOutput(&_dataFormat,
                                      HandleOutputBuffer,
                                      self,
                                      CFRunLoopGetCurrent(),
                                      kCFRunLoopCommonModes, 0, &_queue));

    FAIL_ON_ERR([self allocateBuffers]);
    A440SineWaveGeneratorInitWithFrequency(&_sineWaveGenerator,
                                             440.0);

    [self primeBuffers];
    FAIL_ON_ERR(AudioQueueStart(_queue, NULL));
    return YES;

failed:
    // Error handling....
    return NO;
}
- (OSStatus)allocateBuffers;
{
    UInt32 bufferSize = [self calculateBufferSizeForSeconds:0.5];

    OSStatus status;
    for (int i = 0; i < kNumberBuffers; ++i) {
        status = AudioQueueAllocateBuffer(_queue, bufferSize, &_buffers[i]);
        if (status != noErr) {
            return status;
        }
    }
    return noErr;
}
- (UInteger)calculateBufferSizeForSeconds:(CGFloat)seconds {
    UInteger bufferSize = (_dataFormat.mSampleRate * _dataFormat.mBytesPerPacket * seconds);
    return bufferSize;
}
- (void)primeBuffers;
{
    _shouldBufferDataInCallback = YES;
    for (int i = 0; i < kNumberBuffers; ++i) {
        HandleOutputBuffer(self, _queue, _buffers[i]);
    }
}
- (BOOL)play:(NSError **)error {
    NSAssert(_queue == NULL, @"Queue is already setup");

    OSStatus status;

    [self setupDataFormat];
    FAIL_ON_ERR(AudioQueueNewOutput(&_dataFormat,
                                       HandleOutputBuffer,
                                       self,
                                       CFRunLoopGetCurrent(),
                                       kCFRunLoopCommonModes,
                                       0, &_queue));

    FAIL_ON_ERR([[self allocateBuffers]]);
    A440SineWaveGeneratorInitWithFrequency(&_sineWaveGenerator,
                                             440.0);

    [self primeBuffers];
    FAIL_ON_ERR(AudioQueueStart(_queue, NULL));
    return YES;

    failed:
    // Error handling....
    return NO;
}
static void HandleOutputBuffer(void * inUserData, AudioQueueRef inAQ, AudioQueueBufferRef inBuffer)
{
    A440AudioQueue * self = inUserData;

    if (!self->_shouldBufferDataInCallback) {
        return;
    }

    int16_t * sample = inBuffer->mAudioData;
    UInt32 numberOfFrames = (inBuffer->mAudioDataBytesCapacity /
                            self->_dataFormat.mBytesPerFrame);

    for (UInt32 i = 0; i < numberOfFrames; i++) {
        FillFrame(self, sample);
        sample += self->_dataFormat.mChannelsPerFrame;
    }

    inBuffer->mAudioDataByteSize = (numberOfFrames *
                                    self->_dataFormat.mBytesPerFrame);

    OSStatus result;
    result = AudioQueueEnqueueBuffer(self->_queue, inBuffer, 0, NULL);
    if (result != noErr) {
        NSLog(@"AudioQueueEnqueueBuffer error: %d", result);
    }
}
static void FillFrame(A440AudioQueue * self, int16_t * sample) {
    A440SineWaveGenerator * generator = &self->_sineWaveGenerator;
    int16_t sampleValue =
        A440SineWaveGeneratorNextSample(generator);
    // Divide by four to keep the volume away from the max
    sampleValue /= 4;

    // Fill two channels
    sample[0] = sampleValue;
    sample[1] = sampleValue;
}
typedef struct
{
    float currentPhase;
    float phaseIncrement;
} A440SineWaveGenerator;

const Float64 SAMPLE_RATE = 44100.0;

void A440SineWaveGeneratorInitWithFrequency(
    A440SineWaveGenerator * self,
    double frequency)
{
    // Given:
    //   frequency in cycles per second
    //   2*PI radians per sine wave cycle
    //   sample rate in samples per second
    //
    // Then:
    //   cycles   radians    seconds    radians
    //   ------- * ------- * ------- = -------
    //   second    cycle     sample     sample
    self->currentPhase = 0.0;
    self->phaseIncrement = frequency * 2*M_PI / SAMPLE_RATE;
}
int16_t A440SineWaveGeneratorNextSample(A440SineWaveGenerator * self) {
    int16_t sample = INT16_MAX * sinf(self->currentPhase);

    self->currentPhase += self->phaseIncrement;
    // Keep the value between 0 and 2*M_PI
    while (self->currentPhase > 2*M_PI) {
        self->currentPhase -= 2*M_PI;
    }

    return sample;
}
- (BOOL)stop:(NSError **)error;
{
    NSAssert(_queue != NULL, @"Queue is not setup");

    OSStatus status;
    _shouldBufferDataInCallback = NO;
    FAIL_ON_ERR(AudioQueueStop(_queue, YES));
    FAIL_ON_ERR(AudioQueueDispose(_queue, YES));
    _queue = NULL;
    return YES;
}

failed:
    // Error handling...
    return NO;
Audio Session

• Activation and Deactivation

• Session Categories

• Interruption Callbacks

• Two APIs:
  
  • AVAudioSession - Objective-C

  • Audio Session Services - C
- (void)setupAudioSession;
{
    [self activateAudioSession];
    [[AVAudioSession sharedInstance] setDelegate:self];
    [self setPlaybackAudioSessionCategory];
}

- (void)beginInterruption;
{
    _playOnEndInterruption = self.isPlaying;
    [self stop];
}

- (void)endInterruption;
{
    [self activateAudioSession];
    if (_playOnEndInterruption) {
        [self play];
    }
}
Chiptune Player
Nintendo Sound Format (NSF)
Nintendo Sound Format (NSF)

- Similar to MIDI
  - Two square waves
  - One triangle wave
  - One noise channel
  - One (primitive) sample channel
- Need to emulate a 6502
Game_Music_Emu

Game_Music_Emu emulates game music in several popular file formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>AY</td>
<td>ZX Spectrum, Amstrad CPC</td>
</tr>
<tr>
<td>GBS</td>
<td>Nintendo Game Boy</td>
</tr>
<tr>
<td>GYM</td>
<td>Sega Genesis, Mega Drive</td>
</tr>
<tr>
<td>HES</td>
<td>NEC TurboGrafx-16, PC Engine</td>
</tr>
<tr>
<td>KSS</td>
<td>MSX Home Computer, other Z80 systems (doesn't support FM sound)</td>
</tr>
<tr>
<td>NSF, NSFE</td>
<td>Nintendo NES, Famicom (with VRC 6, Namco 106, and FME-7 sound)</td>
</tr>
<tr>
<td>SAP</td>
<td>Atari systems using POKEY sound chip</td>
</tr>
<tr>
<td>SPC</td>
<td>Super Nintendo, Super Famicom</td>
</tr>
<tr>
<td>VGM, VGZ</td>
<td>Sega Master System, Mark III, Sega Genesis, Mega Drive, BBC Micro</td>
</tr>
</tbody>
</table>

Game_Music_Emu works in C and C++ and has been made very easy to use. Sound is generated using high-quality yet efficient band-limited synthesis. Substantial documentation and examples are provided, including a mini player using SDL. Modular design gives flexibility and allows easy elimination of unused features and music formats.
@interface GmeMusicFile : NSObject
{
    MusicEmu * _emu;
}

+ (id)musicFileAtPath:(NSString *)path
    sampleRate:(long)sampleRate error:(NSError **)error;
+ (id)musicFileAtPath:(NSString *)path error:(NSError **)error;

- (long)sampleRate;
- (int)numberOfTracks;
- (TrackInfo *)infoForTrack:(int)track;

- (BOOL)playTrack:(int)track error:(NSError **)error;
- (BOOL)trackEnded;

BOOL GmeMusicFilePlay(GmeMusicFile * file, long count,
                        short * samples, NSError ** error);

@end
static void HandleOutputBuffer(void * inUserData,
    AudioQueueRef inAQ,
    AudioQueueBufferRef inBuffer)
{
    MusicPlayerAudioQueueOutput * self =
        (MusicPlayerAudioQueueOutput *) inUserData;

    if (!self->_shouldBufferDataInCallback) {
        return;
    }

    GmeMusicFile * musicFile = self->_musicFile;
    if (musicFile == nil) {
        NSLog(@"No music file");
        return;
    }
NSError * error = nil;
if (!GmeMusicFilePlay(musicFile, 
    inBuffer->mAUDIO_DATA_BYTES_CAPACITY/2, 
    inBuffer->mAUDIO_DATA, &error))
{
    NSLog(@"GmeMusicFilePlay error: %@ %@", error, 
        [error userInfo]);
    return;
}

inBuffer->mAUDIO_DATA_BYTE_SIZE = 
    inBuffer->mAUDIO_DATA_BYTES_CAPACITY;
OSStatus result = AudioQueueEnqueueBuffer(self->_queue, 
                                            inBuffer, 0, NULL);
if (result != noErr) {
    NSLog(@"AudioQueueEnqueueBuffer error: %d", result);
}
Asynchronous stop means all queued buffers still get played.

if ([musicFile trackEnded]) {
    self->_shouldBufferDataInCallback = NO;
    self->_stoppedDueToTrackEnding = YES;
    AudioQueueStop(self->_queue, NO);
}
Handling File Ending

- Asynchronous Stop:

  ```c
  AudioQueueStop(_queue, NO);
  ```

- Setup a Property Listener:

  ```c
  AudioQueueAddPropertyListener(_queue,
  kAudioQueueProperty_IsRunning,
  HandleIsRunningChanged,
  self);
  ```
Other Audio Queue Goodies

• Control the volume

• Level metering

• Handle compression without using AudioConverter