Audio Queue Services

Dave Dribin – @ddribin

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

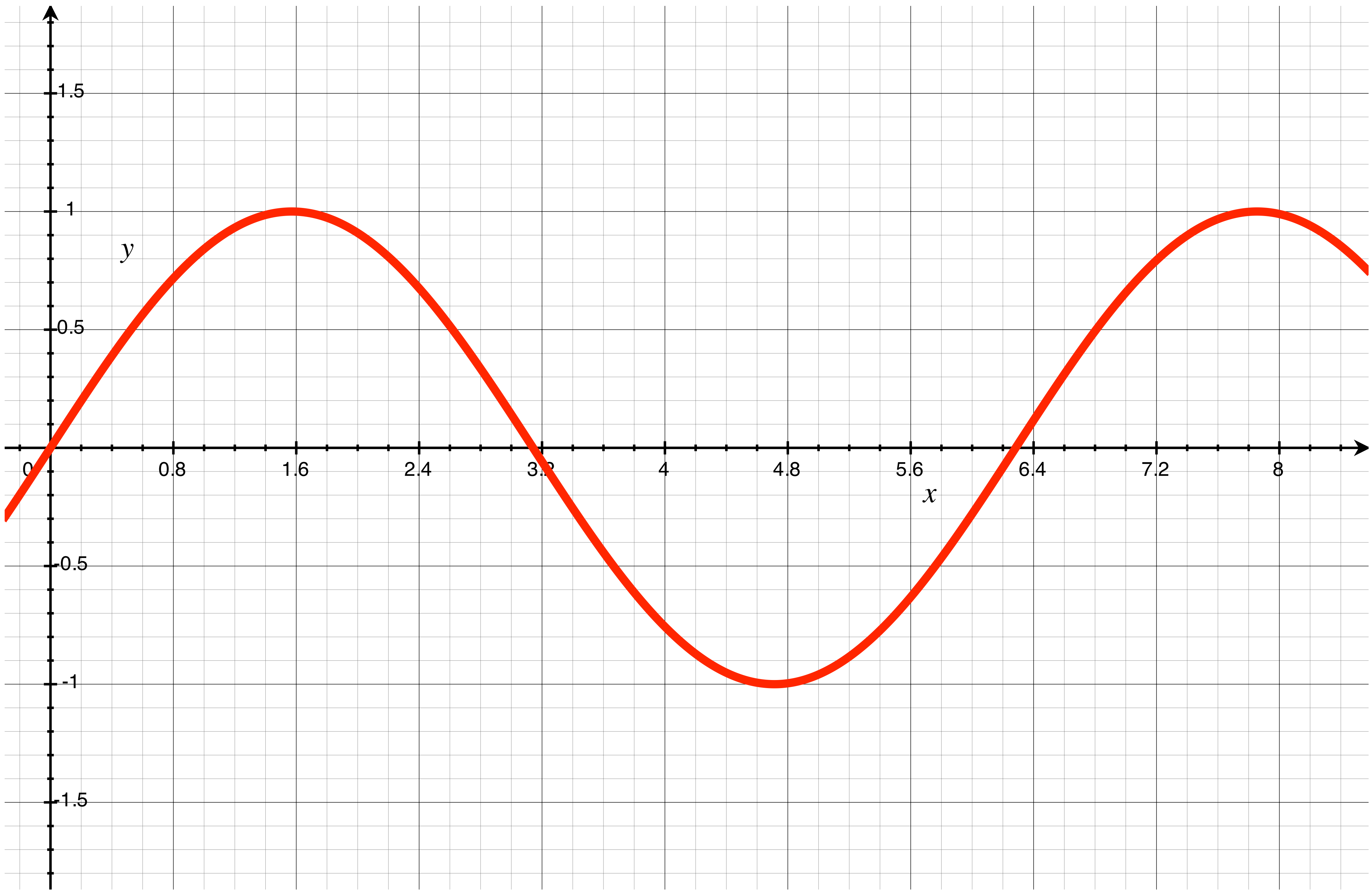
Playing Audio on iPhone OS

- ~~Media Player Framework~~
- ~~AV Foundation Framework~~
- ~~OpenAL Library~~
- Audio Toolbox Framework
- Audio Unit Framework

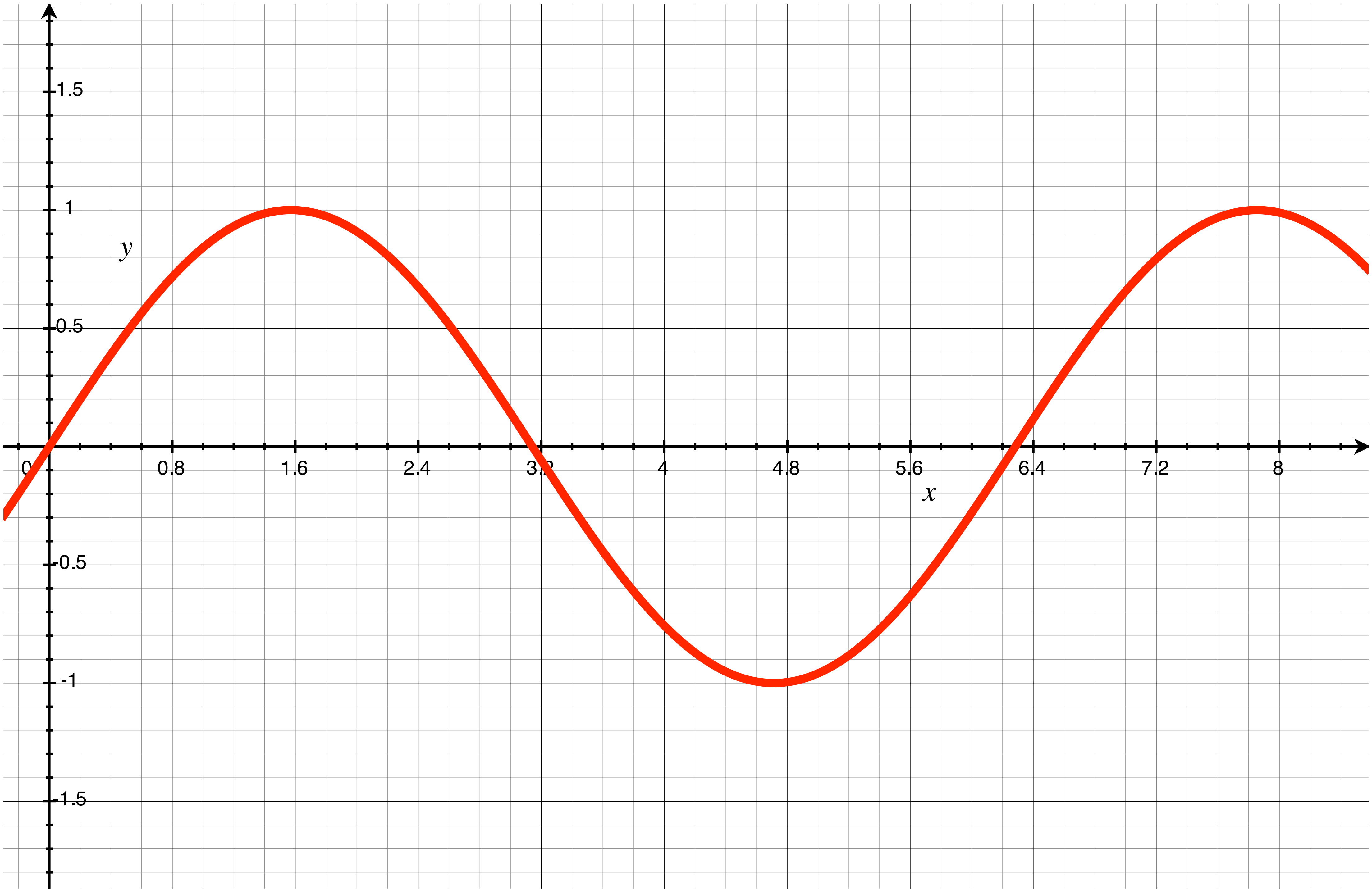
Final Application Demo

Crash Course in Digital Audio

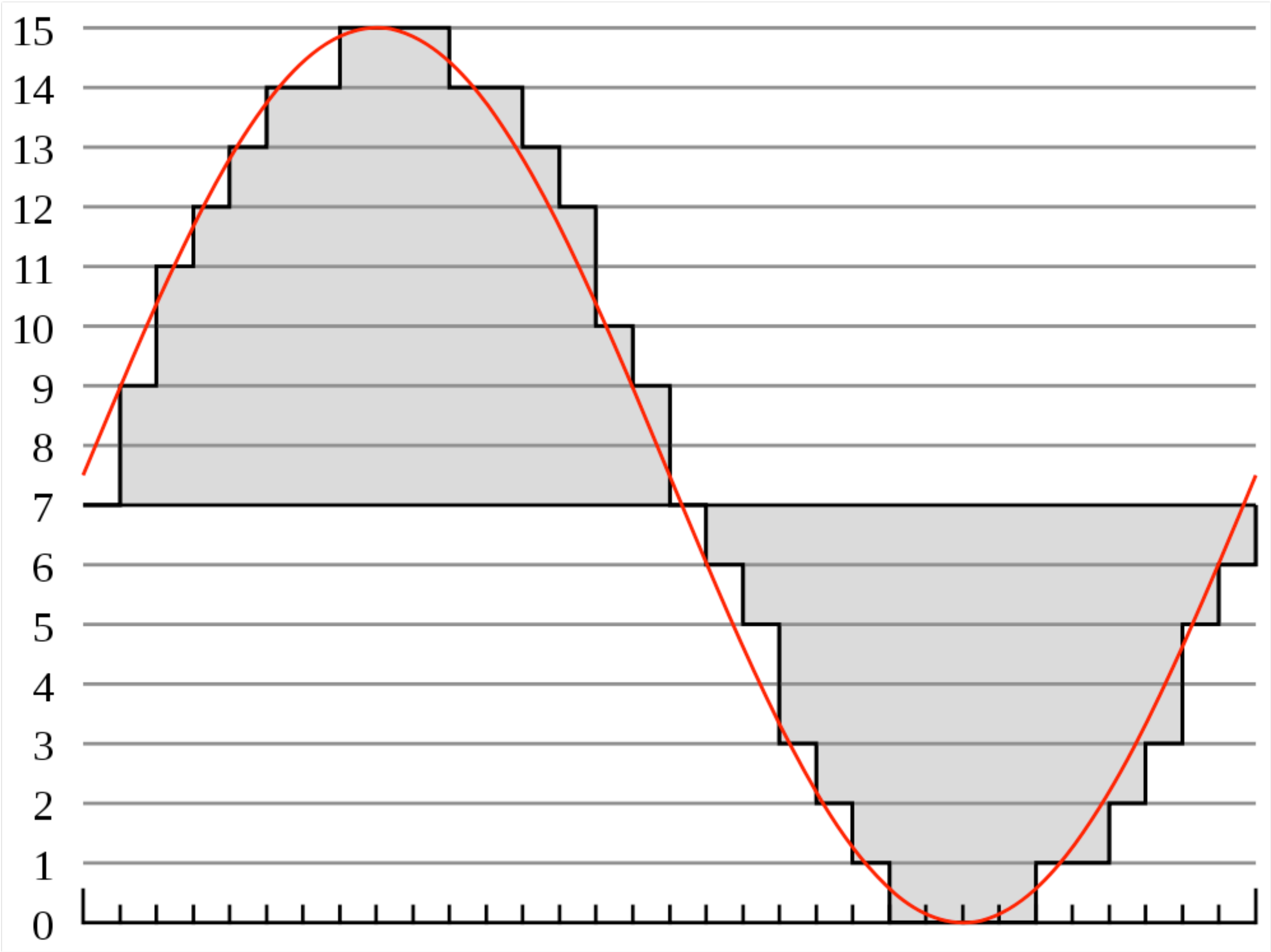
Analog Audio - A.K.A Sound



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

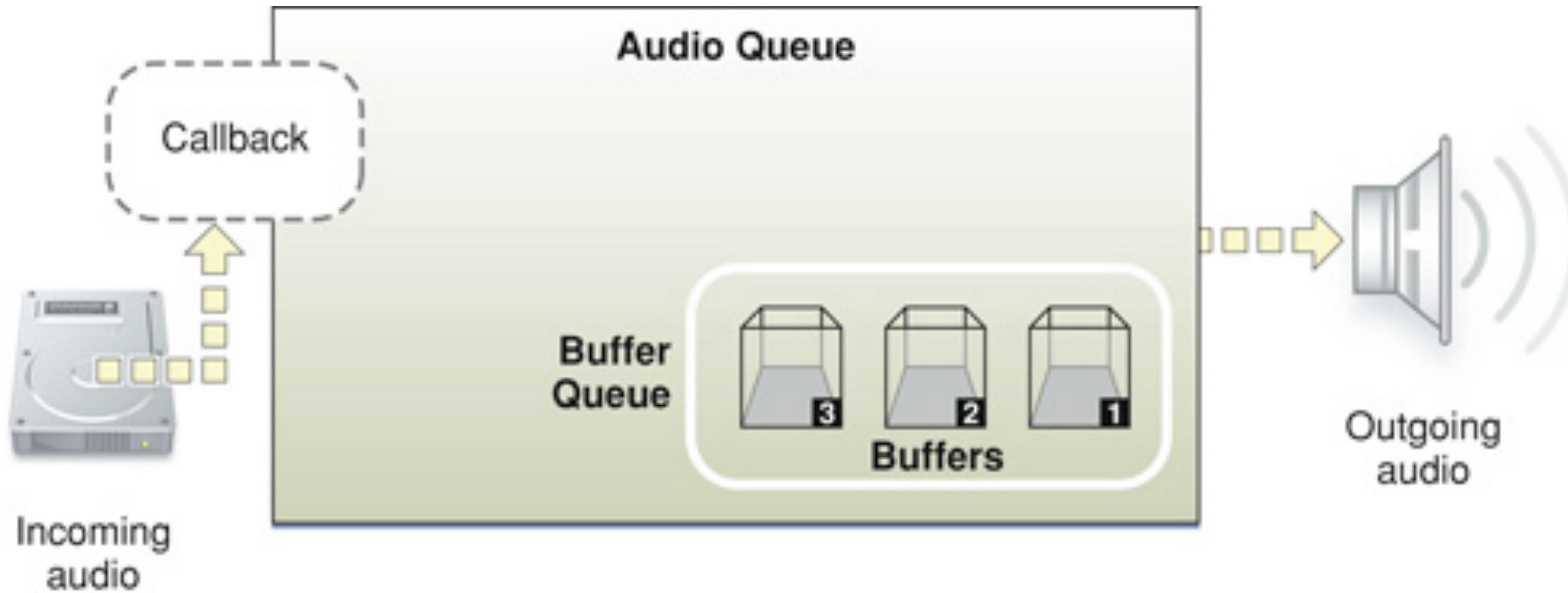
- Lossless
  - WAV, AIFF (1:1)
  - FLAC, Apple Lossless (2:1)
- Lossy
  - MP3, AAC (10:1)

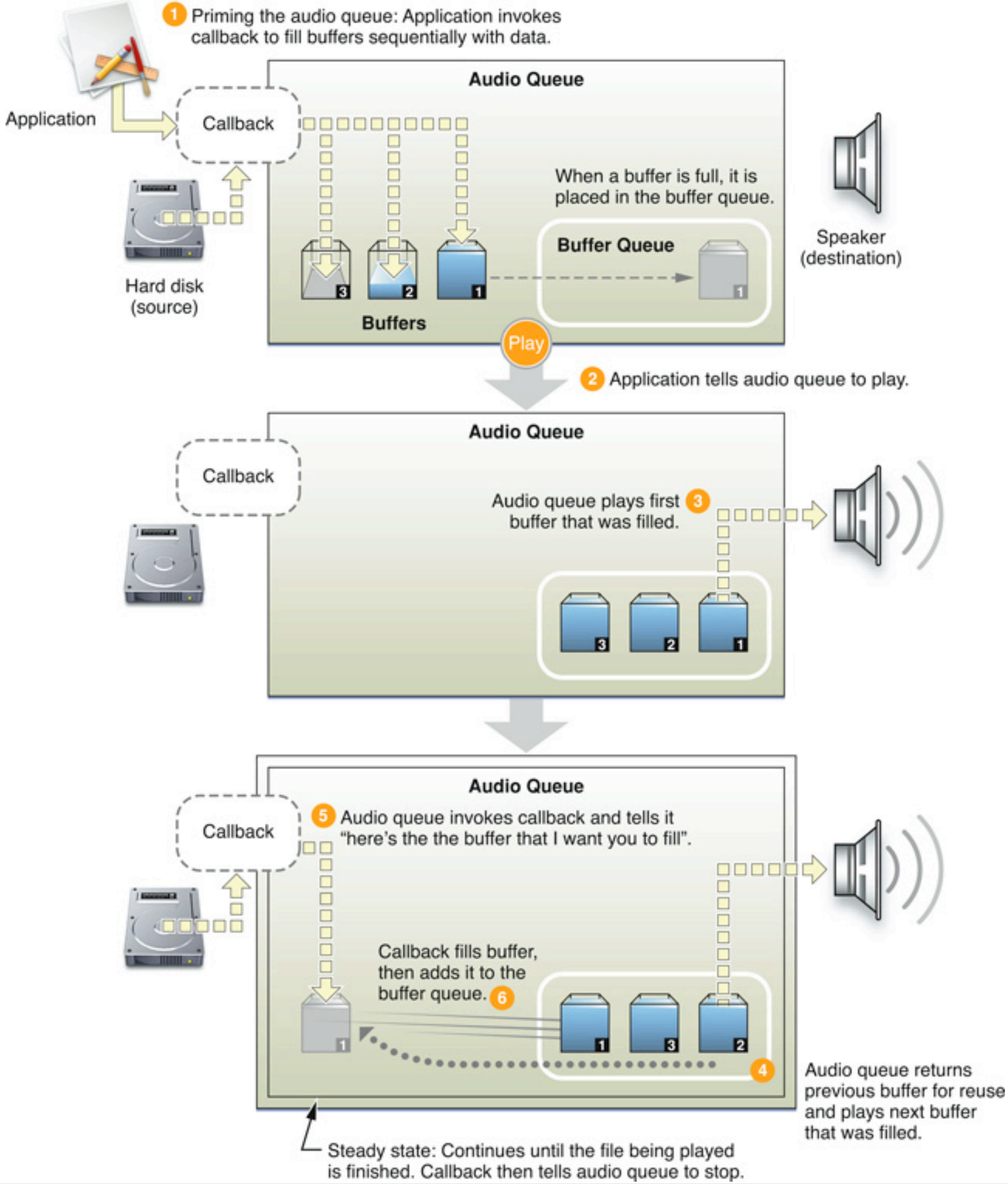
Audio Queue Services “Hello World”

- Play 440 Hz sound
- A440 Project on BitBucket:
    - http://bitbucket.org/ddribin/a440

Demo

What is an Audio Queue





1 Priming the audio queue: Application invokes callback to fill buffers sequentially with data.
Application
Callback
Audio Queue
Hard disk (source)
When a buffer is full, it is placed in the buffer queue.
Buffer Queue
Speaker (destination)
3
2
1
1
Buffers
Play
2 Application tells audio queue to play.
Audio Queue
Callback
Audio queue plays first 3 buffer that was filled.
3
2
1
Audio Queue
Callback
5 Audio queue invokes callback and tells it "here's the the buffer that I want you to fill".
Callback fills buffer, then adds it to the buffer queue. 6
1
1
3
2
4
Audio queue returns previous buffer for reuse and plays next buffer that was filled.
Steady state: Continues until the file being played is finished. Callback then tells audio queue to stop.

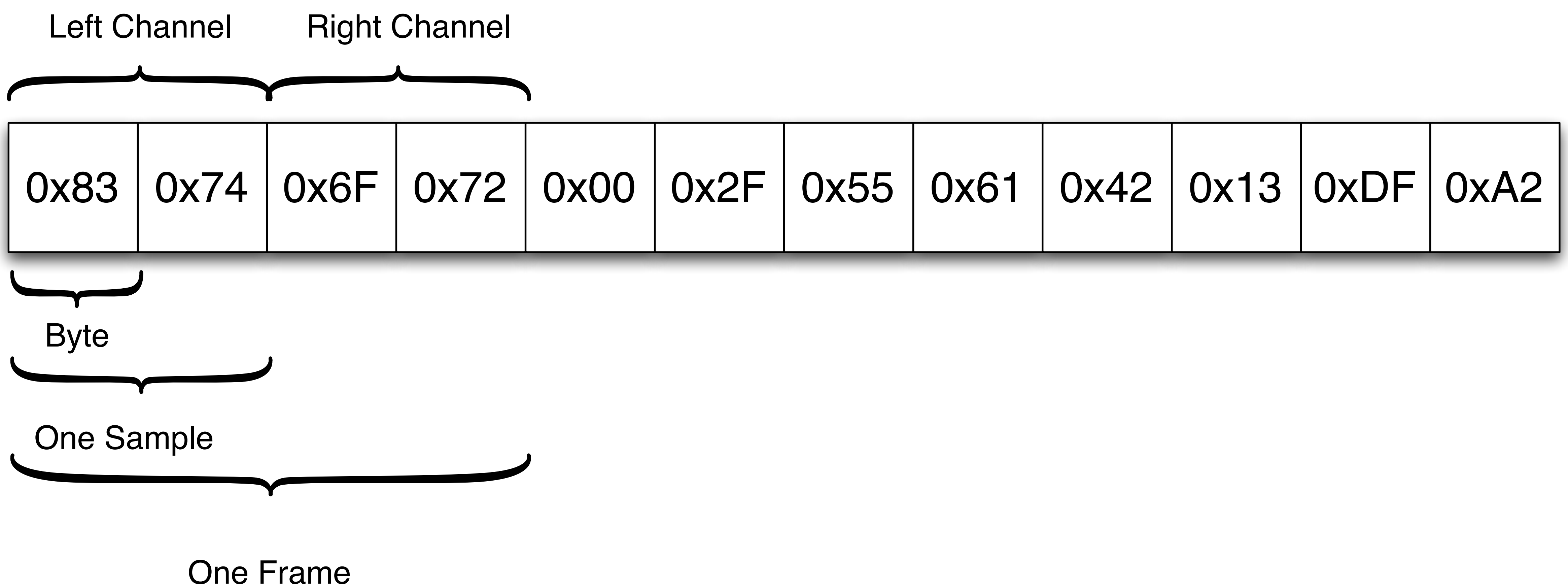
Creating an Audio Queue for Output

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

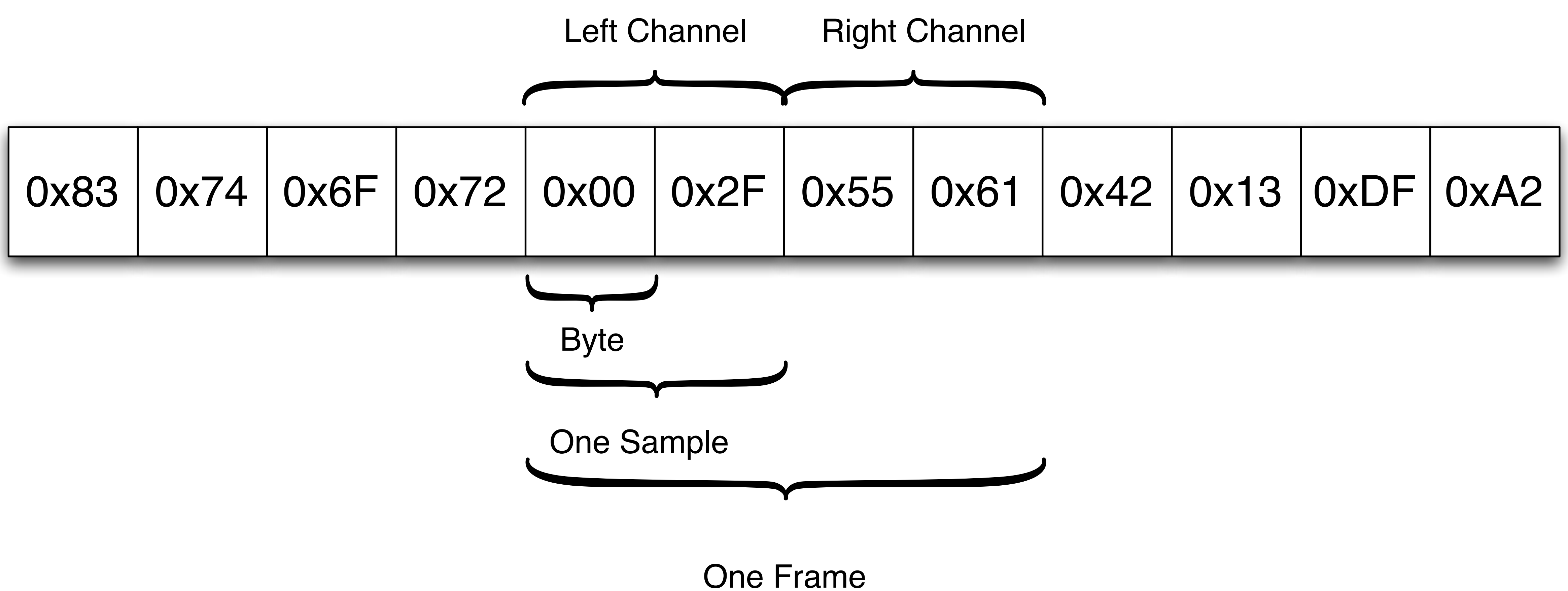
AudioStreamBasicDescription (ASBDs)

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

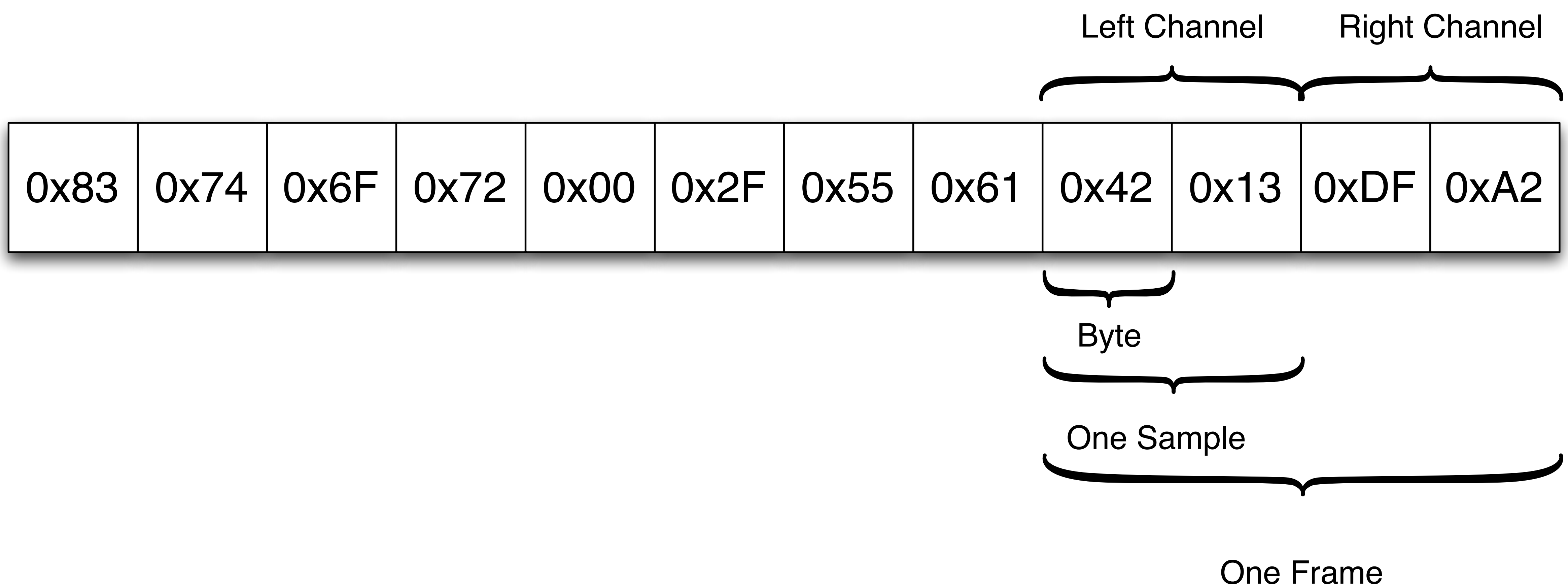
16-bits per Sample, Stereo Byte Stream



16-bits per Sample, Stereo Byte Stream



16-bits per Sample, Stereo Byte Stream



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
```

```objc
- (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;
}
```

```objc
- (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;
}
```

```
- (UInt32)calculateBufferSizeForSeconds:(Float64)seconds;
{
    UInt32 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;
}
```

```c
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

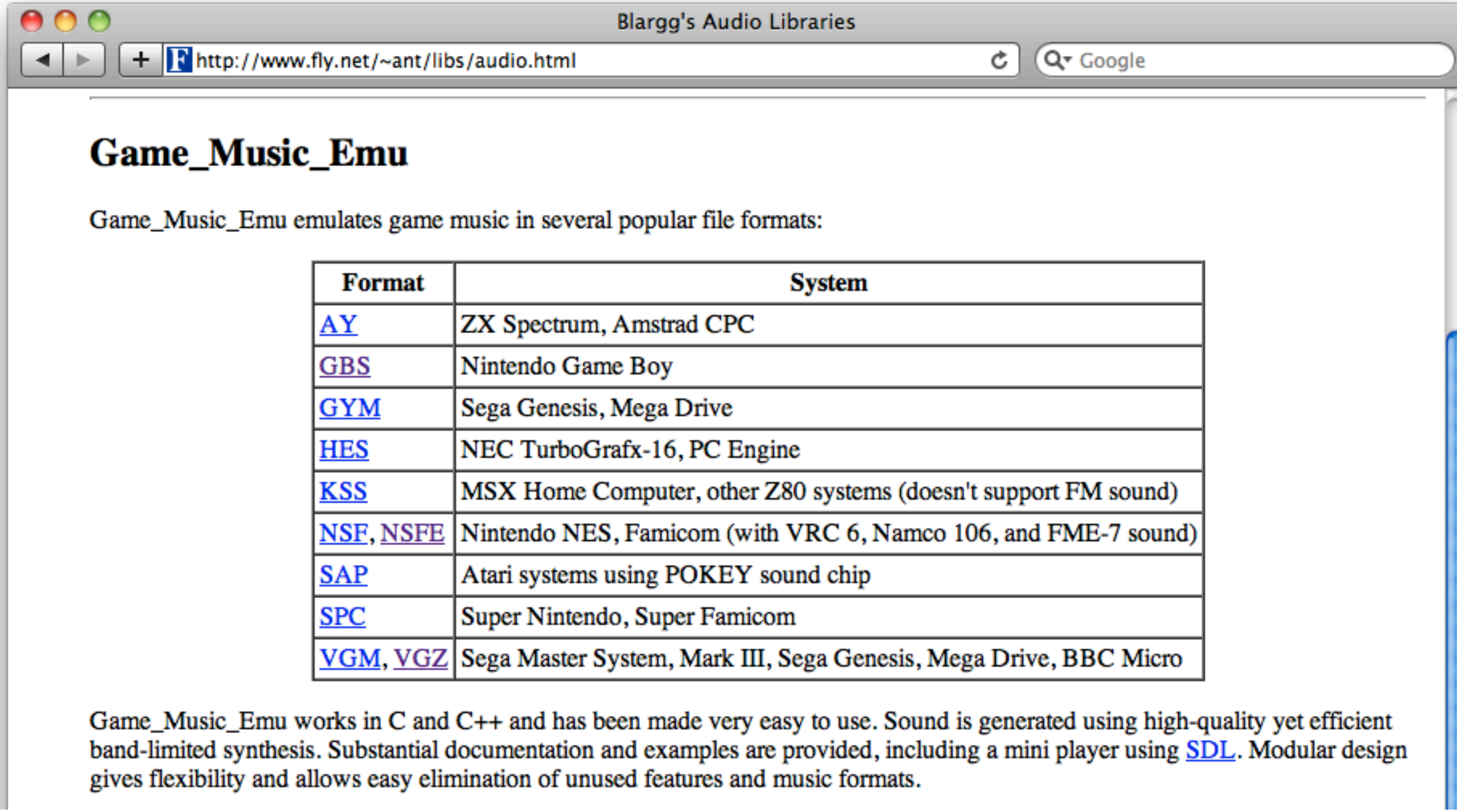
```objc
- (void)setupAudioSession;
{
    [self activateAudioSession];
    [[AVAudioSession sharedInstance] setDelegate:self];
    [self setPlaybackAudioSessionCategory];
}

- (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:

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

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.

```objc
@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->mAudioDataBytesCapacity/2,
                      inBuffer->mAudioData, &error))
{
    NSLog(@"GmeMusicFilePlay error: %@ %@", error,
         [error userInfo]);
    return;
}

inBuffer->mAudioDataByteSize =
    inBuffer->mAudioDataBytesCapacity;
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:

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

- Setup a Property Listener:

```
AudioQueueAddPropertyListener(_queue,
                              kAudioQueueProperty_IsRunning,
                              HandleIsRunningChanged,
                              self);
```

Other Audio Queue Goodies

- Control the volume
- Level metering
- Handle compression without using AudioConverter