What’s New in Audio
Session 501

Akshatha Nagesh, AudioEngine-eer
Béla Balázs, Audio Artisan
Torrey Holbrook Walker, Audio/MIDI Black Ops
Audio Stack

Delivering an Exceptional Audio Experience
AVAudioEngine
AVAudioSession
watchOS
AUAudioUnit
Other Enhancements
Inter-Device Audio Mode (IDAM)
AVAudioEngine
Recap

Powerful, feature-rich, Objective-C / Swift API set

Simplifies realtime audio, easier to use

Supports

- Playback, recording, processing, mixing
- 3D spatialization

AVAudioEngine in Practice  
WWDC14

What’s New in Core Audio  
WWDC15
Sample Engine Setup

Karaoke

- InputNode
- EffectNode (EQ)
- PlayerNode (Backing Track)
- PlayerNode (Sound Effects)
- NodeTapBlock (Analyze)
- MixerNode
- OutputNode
Sample Engine Setup
Karaoke

InputNode -> EffectNode (EQ) -> MixerNode -> OutputNode

NodeTapBlock (Analyze) -> PlayerNode (Backing Track) -> MixerNode

PlayerNode (Sound Effects) -> MixerNode
Sample Engine Setup
Karaoke

- **InputNode**
- **EffectNode** (EQ)
- **PlayerNode** (Backing Track)
- **MixNode**
- **OutputNode**

Additional blocks:
- **NodeTapBlock** (Analyze)
- **PlayerNode** (Sound Effects)
Sample Engine Setup

Karaoke

- InputNode
- EffectNode (EQ)
  - PlayerNode (Backing Track)
- NodeTapBlock (Analyze)
  - PlayerNode (Sound Effects)
- MixerNode
- OutputNode
Sample Engine Setup

Karaoke

Diagram:
- **InputNode**
- **NodeTapBlock (Analyze)**
- **EffectNode (EQ)**
- **PlayerNode (Backing Track)**
- **PlayerNode (Sound Effects)**
- **MixerNode**
- **OutputNode**

Connections:
- InputNode to NodeTapBlock (Analyze)
- NodeTapBlock (Analyze) to EffectNode (EQ)
- EffectNode (EQ) to PlayerNode (Backing Track)
- PlayerNode (Backing Track) to PlayerNode (Sound Effects)
- PlayerNode (Sound Effects) to MixerNode
- MixerNode to OutputNode
- OutputNode to Headphones
Sample Engine Setup

Karaoke

- InputNode
- EffectNode (EQ)
- PlayerNode (Backing Track)
- PlayerNode (Sound Effects)
- NodeTapBlock (Analyze)
- MixerNode
- OutputNode
What's New

AVAudioEngine
• Manual rendering
• Auto shutdown

AVAudioPlayerNode
• Completion callbacks
What’s New

AVAudioEngine
  • Manual rendering
  • Auto shutdown

AVAudioPlayerNode
  • Completion callbacks
Sample Engine Setup

- InputNode
- EffectNode
- PlayerNode
- MixerNode
- OutputNode
- NodeTapBlock (Analyze)
Sample Engine Setup
Manual rendering

- InputNode
- NodeTapBlock (Analyze)
- EffectNode
- PlayerNode
- MixerNode
- OutputNode
Sample Engine Setup

Manual rendering

- InputNode
- EffectNode
- PlayerNode
- NodeTapBlock (Analyze)
- MixerNode
- OutputNode

Application
Sample Engine Setup

Manual rendering

InputNode → EffectNode → MixerNode → OutputNode

NodeTapBlock (Analyze) → PlayerNode → MixerNode

Application
Manual Rendering

Engine is not connected to any audio device

Renders in response to requests from the client

Modes
- Offline
- Realtime
Offline Manual Rendering

Engine and nodes operate under no deadlines or realtime constraints

A node may choose to:

- Use a more expensive signal processing algorithm
- Block on render thread for more data if needed
  - For example, player node may wait until its worker thread reads the data from disk
Offline Manual Rendering
Example
Offline Manual Rendering
Example
Offline Manual Rendering

Example
Offline Manual Rendering

Applications

Post-processing of audio files, for example, apply reverb, effects etc.

Mixing of audio files

Offline audio processing using CPU intensive (higher quality) algorithms

Tuning, debugging or testing the engine setup
Demo

AVAudioEngine - Offline Manual Rendering
Realtime Manual Rendering

The engine and nodes:

- Operate under realtime constraints
- Do not make any blocking calls like blocking on a mutex, calling libdispatch etc., on the render thread
  - A node may drop the data if it is not ready to be rendered in time
Realtime Manual Rendering

Applications

Processing audio in an AUAudioUnit’s internalRenderBlock

Processing audio data in a movie/video during streaming/playback
Realtime Manual Rendering

Example
Realtime Manual Rendering

Example

Audio from an Input Movie Stream  Application  Audio into an Output Movie Stream

TV
Realtime Manual Rendering

Example

Realtime Context

InputNode → EffectNode → OutputNode

Audio from an Input Movie Stream → Application → Audio into an Output Movie Stream

TV
Realtime Manual Rendering
Code example

Application

Realtime Context

InputNode -> EffectNode -> OutputNode
do {
    let engine = AVAudioEngine() // by default engine will render to/from the audio device
    // make connections, e.g. inputNode -> effectNode -> outputNode

    // switch to manual rendering mode
    engine.stop()
    try engine.enableManualRenderingMode(.realtime, format: outputPCMFormat,
            maximumFrameCount: frameCount) // e.g. 1024 @ 48 kHz = 21.33 ms

    let renderBlock = engine.manualRenderingBlock // cache the render block
// Realtime Manual Rendering, code example

do {
    let engine = AVAudioEngine() // by default engine will render to/from the audio device
    // make connections, e.g. inputNode -> effectNode -> outputNode

    // switch to manual rendering mode
    engine.stop()
    try engine.enableManualRenderingMode(.realtime, format: outputPCMFormat,
                                           maximumFrameCount: frameCount) // e.g. 1024 @ 48 kHz = 21.33 ms

    let renderBlock = engine.manualRenderingBlock // cache the render block
// Realtime Manual Rendering, code example

do {
    let engine = AVAudioEngine() // by default engine will render to/from the audio device
    // make connections, e.g. inputNode -> effectNode -> outputNode

    // switch to manual rendering mode
    engine.stop()
    try engine.enableManualRenderingMode(.realtime, format: outputPCMFormat,
        maximumFrameCount: frameCount) // e.g. 1024 @ 48 kHz = 21.33 ms

    let renderBlock = engine.manualRenderingBlock // cache the render block
do {
    let engine = AVAudioEngine() // by default engine will render to/from the audio device
    // make connections, e.g. inputNode -> effectNode -> outputNode

    // switch to manual rendering mode
    engine.stop()
    try engine.enableManualRenderingMode(.realtime, format: outputPCMFormat,
                                          maximumFrameCount: frameCount) // e.g. 1024 @ 48 kHz = 21.33 ms

    let renderBlock = engine.manualRenderingBlock // cache the render block
// set the block to provide input data to engine

engine.inputNode.setManualRenderingInputPCMFormat(inputPCMFormat) {
    (inputFrameCount) -> UnsafePointer<AudioBufferList>? in
        guard haveData else { return nil }

        // fill and return the input audio buffer list
        return inputBufferList
}

// create output buffer, cache the buffer list

let buffer = AVAudioPCMBuffer(pcmFormat: outputPCMFormat,
                               frameCapacity: engine.manualRenderingMaximumFrameCount)!

buffer.frameLength = buffer.frameCapacity

let outputBufferList = buffer.mutableAudioBufferList

try engine.start()

} catch { // handle errors }
// set the block to provide input data to engine
engine.inputNode.setManualRenderingInputPCMFormat(inputPCMFormat) {
    (inputFrameCount) -> UnsafePointer<AudioBufferList>? in

        guard haveData else { return nil }

        // fill and return the input audio buffer list
        return inputBufferList
    }

// create output buffer, cache the buffer list
let buffer = AVAudioPCMBuffer(pcmFormat: outputPCMFormat,
                               frameCapacity: engine.manualRenderingMaximumFrameCount)!
buffer.frameLength = buffer.frameCapacity
let outputBufferList = buffer.mutableAudioBufferList
try engine.start()
} catch { // handle errors }
// set the block to provide input data to engine
engine.inputNode.setManualRenderingInputPCMFormat(inputPCMFormat) {
    (inputFrameCount) -> UnsafePointer<AudioBufferList>? in
        guard haveData else { return nil }

        // fill and return the input audio buffer list
        return inputBufferList
}

    // create output buffer, cache the buffer list
    let buffer = AVAudioPCMBuffer(pcmFormat: outputPCMFormat,
        frameCapacity: engine.manualRenderingMaximumFrameCount)!
    buffer.frameLength = buffer.frameCapacity
    let outputBufferList = buffer.mutableAudioBufferList
    try engine.start()
} catch { // handle errors }
// set the block to provide input data to engine
engine.inputNode.setManualRenderingInputPCMFormat(inputPCMFormat) {
    (inputFrameCount) -> UnsafePointer<AudioBufferList>? in
    guard haveData else { return nil }

    // fill and return the input audio buffer list
    return inputBufferList
}

// create output buffer, cache the buffer list
let buffer = AVAudioPCMBuffer(pcmFormat: outputPCMFormat,
                               frameCapacity: engine.manualRenderingMaximumFrameCount)!

buffer.frameLength = buffer.frameCapacity
let outputBufferList = buffer.mutableAudioBufferList

try engine.start()
} catch { // handle errors }
// set the block to provide input data to engine
engine.inputNode.setManualRenderingInputPCMFormat(inputPCMFormat) {
    (inputFrameCount) -> UnsafePointer<AudioBufferList>? in
        guard haveData else { return nil }

        // fill and return the input audio buffer list
        return inputBufferList
}

// create output buffer, cache the buffer list
let buffer = AVAudioPCMBuffer(pcmFormat: outputPCMFormat,
                               frameCapacity: engine.manualRenderingMaximumFrameCount)!
buffer.frameLength = buffer.frameCapacity
let outputBufferList = buffer.mutableAudioBufferList
try engine.start()
} catch { // handle errors }
// to render from realtime context
OSStatus outputError = noErr;
const auto status = renderBlock(framesToRender, outputBufferList, &outputError);
switch (status) {
    case AVAudioEngineManualRenderingStatusSuccess:
        handleProcessedOutput(outputBufferList); // data rendered successfully
        break;

    case AVAudioEngineManualRenderingStatusInsufficientDataFromInputNode:
        handleProcessedOutput(outputBufferList); // input node did not provide data,
        // but other sources may have rendered
        break;
    ..
    default:
        break;
}
/ to render from realtime context

```c
OSStatus outputError = noErr;
const auto status = renderBlock(framesToRender, outputBufferList, &outputError);
switch (status) {
    case AVAudioEngineManualRenderingStatusSuccess:
        handleProcessedOutput(outputBufferList); // data rendered successfully
        break;
    case AVAudioEngineManualRenderingStatusInsufficientDataFromInputNode:
        handleProcessedOutput(outputBufferList); // input node did not provide data,
        break; // but other sources may have rendered
    ..
    default:
        break;
}
```
// to render from realtime context
OSStatus outputError = noErr;
const auto status = renderBlock(framesToRender, outputBufferList, &outputError);
switch (status) {
    case AVAudioEngineManualRenderingStatusSuccess:
        handleProcessedOutput(outputBufferList); // data rendered successfully
        break;
    case AVAudioEngineManualRenderingStatusInsufficientDataFromInputNode:
        handleProcessedOutput(outputBufferList); // input node did not provide data,
        // but other sources may have rendered
        break;
    ..
    default:
        break;
}
// to render from realtime context
OSStatus outputError = noErr;
const auto status = renderBlock(framesToRender, outputBufferList, &outputError);
switch (status) {
    case AVAudioEngineManualRenderingStatusSuccess:
        handleProcessedOutput(outputBufferList); // data rendered successfully
        break;

    case AVAudioEngineManualRenderingStatusInsufficientDataFromInputNode:
        handleProcessedOutput(outputBufferList); // input node did not provide data,
        // but other sources may have rendered
        break;

    default:
        break;
}
Manual Rendering

Render calls

Offline
• Can use either ObjC/Swift render method or the block based render call

Realtime
• Must use the block based render call
What’s New

AVAudioEngine
• Manual rendering
• Auto shutdown

AVAudioPlayerNode
• Completion callbacks
Auto Shutdown

Hardware is stopped if running idle for a certain duration, started dynamically when needed

Safety net for conserving power

Enforced behavior on watchOS, optional on other platforms

```swift
isAutoShutdownEnabled
```
What’s New

AVAudioEngine
• Manual rendering
• Auto shutdown

AVAudioPlayerNode
• Completion callbacks
Completion Callbacks

Existing buffer/file completion handlers called when the data has been consumed

New completion handler and callback types

AVAudioPlayerNodeCompletionCallbackType
  .dataConsumed
  .dataRendered
  .dataPlayedBack
Completion Callbacks

.dataConsumed

- Data has been consumed, same as the existing completion handlers
- The buffer can be recycled, more data can be scheduled

.dataRendered

- Data has been output by the player
- Useful in manual rendering mode
- Does not account for any downstream signal processing latency
Completion Callbacks

- Buffer/file has finished playing
- Applicable only when the engine is rendering to an audio device
- Accounts for both (small) downstream signal processing latency and (possibly significant) audio playback device latency
Completion Callbacks

- Buffer/file has finished playing
- Applicable only when the engine is rendering to an audio device
- Accounts for both (small) downstream signal processing latency and (possibly significant) audio playback device latency
Completion Callbacks

- Buffer/file has finished playing
- Applicable only when the engine is rendering to an audio device
- Accounts for both (small) downstream signal processing latency and (possibly significant) audio playback device latency

```swift
player.scheduleFile(file, at: nil, completionCallbackType: .dataPlayedBack) {
    (callbackType) in
    // file has finished playing from listener’s perspective
    // notify to stop the engine and update UI
}
```
AVAudioEngine
Summary

AVAudioEngine
• Manual rendering
• Auto shutdown

AVAudioPlayerNode
• Completion callbacks

Deprecation coming soon (2018)
• AUGraph
AVAudioSession
AirPlay 2 Support

AirPlay 2 - new technology on iOS, tvOS and macOS
• Multi-room audio with AirPlay 2 capable devices

Long-form audio applications
• Content - music, podcasts etc.
• Separate, shared audio route to AirPlay 2 devices
• New AVAudioSession API for an application to identify itself as long-form
Audio Routing (iOS) - Current Behavior

Music and phone call

Session Arbitration and Mixing

AirPlay

System Audio
Audio Routing (iOS) - Current Behavior

Music and phone call

Session Arbitration and Mixing

AirPlay
System Audio
Audio Routing (iOS) - Current Behavior

Music and phone call

Interrupted

Session Arbitration and Mixing

AirPlay

System Audio
Audio Routing (iOS) - Current Behavior

Music and phone call

Session Arbitration and Mixing

Interrupted

AirPlay

System Audio
Audio Routing (iOS) - Current Behavior

Music and phone call

Interrupted

Session Arbitration and Mixing

AirPlay

System Audio
Audio Routing (iOS) - Current Behavior
Music and phone call

Session Arbitration and Mixing

- AirPlay
- System Audio
Long-form Audio Routing (iOS)
Music and phone call coexistence

NEW
Long-form Audio Routing (iOS)
Music and phone call coexistence

Long-form audio route

Session Arbitration

Long-form Audio (AirPlay 2)

System Audio
Long-form Audio Routing (iOS)
Music and phone call coexistence

Long-form audio route

Session Arbitration

Long-form Audio (AirPlay 2)

System Audio
Long-form Audio Routing (iOS)
Music and phone call coexistence

Long-form audio route
- Session Arbitration
- Long-form Audio (AirPlay 2)

System audio route
- Session Arbitration and Mixing
- System Audio
Long-form Audio Routing (iOS and tvOS)

Applications that use the long-form audio route

- Session Arbitration
- Long-form Audio (AirPlay 2)
Long-form Audio Routing (iOS and tvOS)

Applications that use the long-form audio route
- Session Arbitration
- Long-form Audio (AirPlay 2)

Applications that use the system audio route
- Session Arbitration and Mixing
- System Audio
let mySession = AVAudioSession.sharedInstance()

do {
    try mySession.setCategory(AVAudioSessionCategoryPlayback,
        mode: AVAudioSessionModeDefault,
        routeSharingPolicy: .longForm)
} catch {
    // handle errors
}
let mySession = AVAudioSession.sharedInstance()

do {
    try mySession.setCategory(AVAudioSessionCategoryPlayback,
        mode: AVAudioSessionModeDefault,
        routeSharingPolicy: .longForm)
}
catch {
    // handle errors
}
let mySession = AVAudioSession.sharedInstance()
do {
    try mySession.setCategory(AVAudioSessionCategoryPlayback,
        mode: AVAudioSessionModeDefault,
        routeSharingPolicy: .longForm)
} catch {
    // handle errors
}
let mySession = AVAudioSession.sharedInstance()

do {
    try mySession.setCategory(AVAudioSessionCategoryPlayback, 
        mode: AVAudioSessionModeDefault, 
        routeSharingPolicy: .longForm)
} catch {
    // handle errors
}
Long-form Audio Routing (macOS)

Applications that use the long-form audio route

Arbitration

Long-form Audio (AirPlay 2)
Applications that use the long-form audio route

Arbitration

Long-form Audio (AirPlay 2)

Applications that use the system audio route

Mixing

Default Device
let mySession = AVAudioSession.sharedInstance()
do {
    try mySession.setRouteSharingPolicy(.longForm)
} catch {
    // handle errors
}
let mySession = AVAudioSession.sharedInstance()

do {
    try mySession.setRouteSharingPolicy(.longForm)
} catch {
    // handle errors
}
let mySession = AVAudioSession.sharedInstance()

do {
    try mySession.setRouteSharingPolicy(.longForm)
} catch {
    // handle errors
}
Enhancements in watchOS
watchOS 4.0
Playback and recording

Playback
• AVAudioPlayer (watchOS 3.1 SDK)

Recording
• AVAudioInputNode (AVAudioEngine)
• AVAudioRecorder
• AVAudioSession recording permissions

Formats supported
• AAC-LC, AAC-ELD, HE-AAC, HE-AACv2, MP3 (decoding only), Opus
watchOS 4.0
Recording policies

Recording can start only in foreground

Recording allowed to continue in the background (red microphone icon displayed)

Recording in background is CPU limited
AVAudioEngine
AVAudioSession
watchOS
AUAudioUnit
Other Enhancements
Inter-Device Audio Mode (IDAM)
AUAudioUnit
AU View Configuration

Host applications decide how to display UI for AUs

Current limitations

• No standard view sizes defined
• AUs are supposed to adapt to any view size chosen by host
AU Preferred View Configuration

Host

Audio Unit Extension
AU Preferred View Configuration

Host

Array of all possible view configurations

Audio Unit Extension

supportedViewConfigurations (availableViewConfigurations)
AU Preferred View Configuration

Host

- Array of all possible view configurations
- IndexSet of supported view configurations

Audio Unit Extension

- supportedViewConfigurations (availableViewConfigurations)
AU Preferred View Configuration

Host:
- Array of all possible view configurations
- IndexSet of supported view configurations
- Chosen view configuration

Audio Unit Extension:
- supportedViewConfigurations (availableViewConfigurations)
- select(viewConfiguration)

NEW
AU Preferred View Configuration
Code example - AU extension

Host
- Array of all possible view configurations
- IndexSet of supported view configurations
- Chosen view configuration

Audio Unit Extension
- supportedViewConfigurations (availableViewConfigurations)
- select(viewConfiguration)
override public func supportedViewConfigurations(_ availableViewConfigurations: [AUAudioUnitViewConfiguration]) -> IndexSet {
    var result = NSMutableIndexSet()
    for (index, config) in availableViewConfigurations.enumerated() {
        // check if the config (width, height, hostHasController) is supported
        // a config of 0x0 (default full size) must always be supported
        if isConfigurationSupported(config) {
            result.add(index)
        }
    }
    return result as IndexSet
}
//AU Preferred View Configuration
Code example - AU extension

```swift
override public func supportedViewConfigurations(_ availableViewConfigurations: [AUAudioUnitViewConfiguration]) -> IndexSet {
    var result = NSMutableIndexSet()
    for (index, config) in availableViewConfigurations.enumerated() {
        // check if the config (width, height, hostHasController) is supported
        // a config of 0x0 (default full size) must always be supported
        if isConfigurationSupported(config) {
            result.add(index)
        }
    }
    return result as IndexSet
}
```
override public func supportedViewConfigurations(_: availableViewConfigurations: [AUAudioUnitViewConfiguration]) -> IndexSet {
    var result = NSMutableIndexSet()
    for (index, config) in availableViewConfigurations.enumerated() {
        // check if the config (width, height, hostHasController) is supported
        // a config of 0x0 (default full size) must always be supported
        if isConfigurationSupported(config) {
            result.add(index)
        }
    }
    return result as IndexSet
}
```swift
override public func supportedViewConfigurations(_ availableViewConfigurations: [AUAudioUnitViewConfiguration]) -> IndexSet {
    var result = NSMutableIndexSet()
    for (index, config) in availableViewConfigurations.enumerated() {
        // check if the config (width, height, hostHasController) is supported
        // a config of 0x0 (default full size) must always be supported
        if isConfigurationSupported(config) {
            result.add(index)
        }
    }

    return result as IndexSet
}
```
//AU Preferred View Configuration
Code example - AU extension

```swift
override public func supportedViewConfigurations(_ availableViewConfigurations: [AUAudioUnitViewConfiguration]) -> IndexSet {
    var result = NSMutableIndexSet()
    for (index, config) in availableViewConfigurations.enumerated() {
        // check if the config (width, height, hostHasController) is supported
        // a config of 0x0 (default full size) must always be supported
        if isConfigurationSupported(config) {
            result.add(index)
        }
    }
    return result as IndexSet
}
```
//AU Preferred View Configuration
Code example - AU extension

```swift
override public func select(_ viewConfiguration: AUAudioUnitViewConfiguration) {
    // configuration selected by host, used by view controller to re-arrange its view
    self.currentViewConfiguration = viewConfiguration
    self.viewController?.selectViewConfig(self.currentViewConfiguration)
}
```
//AU Preferred View Configuration
Code example - AU extension

```swift
override public func select(_ viewConfiguration: AUAudioUnitViewConfiguration) {
    // configuration selected by host, used by view controller to re-arrange its view
    self.currentViewConfiguration = viewConfiguration
    self.viewController?.selectViewConfig(self.currentViewConfiguration)
}
```
//AU Preferred View Configuration
Code example - AU extension

```swift
override public func select(_ viewConfiguration: AUAudioUnitViewConfiguration) {
    // configuration selected by host, used by view controller to re-arrange its view
    self.currentViewConfiguration = viewConfiguration
    self.viewController?.selectViewConfig(self.currentViewConfiguration)
}
```
AU Preferred View Configuration

Code example - host application

**Host**
- Array of all possible view configurations
- IndexSet of supported view configurations
- Chosen view configuration

**Audio Unit Extension**
- supportedViewConfigurations (availableViewConfigurations)
- select(viewConfiguration)
//AU Preferred View Configuration
Code example - host application

```swift
var smallConfigActive: Bool = false // true if the small view is the currently active one
@IBAction func toggleViewModes(_ sender: AnyObject?) {
    guard audioUnit = self.engine.audioUnit else { return }
    let largeConfig = AUAudioUnitViewConfiguration(width: 600, height: 400,
                                                      hostHasController: false)
    let smallConfig = AUAudioUnitViewConfiguration(width: 300, height: 200,
                                                      hostHasController: true)
    let supportedIndices = audioUnit.supportedViewConfigurations([smallConfig, largeConfig])
    if supportedIndices.count == 2 {
        audioUnit.select(self.smallConfigActive ? largeConfig : smallConfig)
        self.smallConfigActive = !self.smallConfigActive
    }
}
```
var smallConfigActive: Bool = false // true if the small view is the currently active one

@IBAction func toggleViewModes(_ sender: AnyObject?) {
    guard audioUnit = self.engine.audioUnit else { return }

    let largeConfig = AUAudioUnitViewConfiguration(width: 600, height: 400, hostHasController: false)
    let smallConfig = AUAudioUnitViewConfiguration(width: 300, height: 200, hostHasController: true)

    let supportedIndices = audioUnit.supportedViewConfigurations([smallConfig, largeConfig])
    if supportedIndices.count == 2 {
        audioUnit.select(self.smallConfigActive ? largeConfig : smallConfig)
        self.smallConfigActive = !self.smallConfigActive
    }
}
var smallConfigActive: Bool = false // true if the small view is the currently active one

@IBAction func toggleViewModes(_ sender: AnyObject?) {
    guard audioUnit = self.engine.audioUnit else { return }
    let largeConfig = AUAudioUnitViewConfiguration(width: 600, height: 400, hostHasController: false)
    let smallConfig = AUAudioUnitViewConfiguration(width: 300, height: 200, hostHasController: true)
    let supportedIndices = audioUnit.supportedViewConfigurations([[smallConfig, largeConfig]])
    if supportedIndices.count == 2 {
        audioUnit.select(self.smallConfigActive ? largeConfig : smallConfig)
        self.smallConfigActive = !self.smallConfigActive
    }
/AU Preferred View Configuration

Code example - host application

```swift
var smallConfigActive: Bool = false // true if the small view is the currently active one

@IBAction func toggleViewModes(_ sender: AnyObject?) {
    guard audioUnit = self.engine.audioUnit else { return }
    let largeConfig = AUAudioUnitViewConfiguration(width: 600, height: 400,
                                                   hostHasController: false)
    let smallConfig = AUAudioUnitViewConfiguration(width: 300, height: 200,
                                                   hostHasController: true)

    let supportedIndices = audioUnit.supportedViewConfigurations([[smallConfig, largeConfig]])
    if supportedIndices.count == 2 {
        audioUnit.select(self.smallConfigActive ? largeConfig : smallConfig)
        self.smallConfigActive = !self.smallConfigActive
    }
}
```
AU MIDI Output

AU can emit MIDI output synchronized with its audio output

Host sets a block on the AU to be called every render cycle

Host can record/edit both MIDI performance and audio output from the AU

*MIDIOutputNames
MIDIOutputEventBlock
Other Updates

Entitlement

• AU extension host applications linked against iOS 11 SDK and later will need 'inter-app-audio' entitlement

AU short name

*audioUnitShortName
Demo
AUAudioUnit

Béla Balázs, Audio Artisan
Other Enhancements
Audio Formats

FLAC (Free Lossless Audio Codec)
• Codec, file, and streaming support
• Content distribution, streaming applications

Opus
• Codec support
• File I/O using .caf container
• VOIP applications
Spatial Audio Formats

B-Format

Audio stream is regular PCM

File container .caf

B-format: W,X,Y,Z

• 1st order ambisonics

kAudioChannelLayoutTag_Ambisonic_B_Format
Spatial Audio Formats

Higher Order Ambisonics

N order ambisonics (N is 1..254)

- SN3D normalized streams
  kAudioChannelLayoutTag_HOA_ACN_SN3D

- N3D normalized streams
  kAudioChannelLayoutTag_HOA_ACN_N3D

ACN (Ambisonic Channel Number) Channels

- kAudioChannelLabel_HOA_ACN_0..65024

AudioFormat support for converting

- Between B-format, ACN_SN3D, ACN_N3D
- From ambisonics to arbitrary speaker layout

NEW
Spatial Mixer
Head-Related Transfer Function (HRTF)

AUSpatialMixer - kSpatializationAlgorithm_HRTFHQ

AVAudioEnvironmentNode - AVAudio3DMixingRenderingAlgorithmHRTFHQ

Features

• Better frequency response
• Better localization of sources in a 3D space
AVAudioEngine
AVAudioSession
watchOS
AUAudioUnit
Other Enhancements
Inter-Device Audio Mode (IDAM)
Inter-Device Audio Mode (IDAM)
Inter-Device Audio Mode

Record audio digitally via Lightning-to-USB cable

USB 2.0 audio class-compliant implementation

Available since El Capitan and iOS 9
IDAM
Inter-Device Audio Mode
IDAM

Inter-Device Audio
IDAM
Inter-Device Audio + MIDI
Inter-Device Audio + MIDI

Send and receive MIDI via Lightning-to-USB cable
Class-compliant USB MIDI implementation
Requires iOS 11 and macOS El Capitan or later
Auto-enabled in IDAM configuration
Inter-Device Audio + MIDI

Device can charge and sync in IDAM configuration

Photo import and tethering are temporarily disabled

Audio device aggregation is ok

Use your iOS devices as a MIDI controllers, destinations, or both
Demo

MIDI using IDAM Configuration
Summary

AVAudioEngine - manual rendering

AVAudioSession - Airplay 2 support

watchOS - recording

AUAudioUnit - preferred view size, MIDI output

Other enhancements - audio formats (FLAC, Opus, HOA)

Inter-Device Audio and MIDI
More Information

<table>
<thead>
<tr>
<th>Session</th>
<th>Location</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introducing MusicKit</td>
<td>Grand Ballroom B</td>
<td>Tuesday 3:10PM</td>
</tr>
<tr>
<td>What's New in watchOS</td>
<td>Hall 2</td>
<td>Wednesday 9:00AM</td>
</tr>
<tr>
<td>Introducing AirPlay 2</td>
<td>Executive Ballroom</td>
<td>Thursday 4:10PM</td>
</tr>
<tr>
<td>Labs</td>
<td>Technology Lab</td>
<td>Time</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Audio Lab</td>
<td>Technology Lab F</td>
<td>Tue 4:10PM–6:00PM</td>
</tr>
<tr>
<td>Airplay Lab</td>
<td>Technology Lab A</td>
<td>Wed 11:00AM–1:00PM</td>
</tr>
<tr>
<td>Audio Lab</td>
<td>Technology Lab G</td>
<td>Thu 1:00PM–3:00PM</td>
</tr>
<tr>
<td>Airplay Lab</td>
<td>Technology Lab A</td>
<td>Fri 9:00AM–11:00AM</td>
</tr>
</tbody>
</table>