

# Swift 2: AVFoundation to play audio or MIDI

## Swift AVFoundation

There are many ways to play sound in iOS. Core Audio has been around for a while and it is very powerful. It is a C API, so using it from Objective-C and Swift is possible, but awkward. Apple has been moving towards a higher level API with AVFoundation. Here I will summarize how to use AVFoundation for several common audio tasks.

N.B. Some of these examples use new capabilities of iOS 8.

This is a newer version of [this Swift 1 blog post](#) .

Playing an Audio file

Audio Session

Playing a MIDI file

Audio Engine

Playing MIDI Notes

Summary

Resources

## Playing an Audio file

Let's start by loading an audio file with an **AVAudioPlayer** instance. There

are several audio formats that the player will grok. I had trouble with a few MP3 files that played in iTunes or VLC, but caused a cryptic exception in the player. So, check your source audio files first.

If you want other formats, your Mac has a converter named **afconvert** . See the man page.

```
afconvert -f caff -d LEI16 foo.mp3 foo.caf
```

```
afconvert -f caff -d LEI16foo.mp3 foo.caf
```

Let's go step by step.

Get the file URL.

```
guard let fileURL = NSBundle.mainBundle().URLForResource("modem",  
    print("could not read sound file")  
    return  
}
```

```
guardlet fileURL = NSBundle.mainBundle().URLForResource("modem",  
    print("could not read sound file")  
    return  
}
```

Create the player. You will need to make the player an instance variable. If you just use a local variable, it will be popped off the stack before you hear anything!

```
do {  
    try self.avPlayer = AVAudioPlayer(contentsOfURL: fileURL)  
} catch {  
    print("could not create AVAudioPlayer \(error)")  
    return  
}
```

```
do {  
    try self.avPlayer = AVAudioPlayer(contentsOfURL: fileURL)  
} catch {  
    print("could not create AVAudioPlayer \(error)")  
    return  
}
```

You can provide the player a hint for how to parse the audio data. There are several [constants](#) for file type UTIs you can use. For our MP3 file, we'll use **AVFileTypeMPEGLayer3**.

```
self.avPlayer = AVAudioPlayer(contentsOfURL: fileURL, fileTypeH
```

```
self.avPlayer = AVAudioPlayer(contentsOfURL: fileURL, fileTypeH
```

Now configure the player. **prepareToPlay()** "pre-rolls" the audio file to reduce start up delays when you finally call **play()**.

You can set the player's delegate to track status.

```
avPlayer.delegate = self  
avPlayer.prepareToPlay()
```

```
avPlayer.volume = 1.0
```

```
avPlayer.delegate = self
avPlayer.prepareToPlay()
avPlayer.volume = 1.0
```

To set the delegate you have to make a class implement the player delegate protocol. My class has the clever name "Sound". The delegate protocol requires the NSObjectProtocol, so Sound is a subclass of NSObject.

```
// MARK: AVAudioPlayerDelegate
extension Sound : AVAudioPlayerDelegate {
    func audioPlayerDidFinishPlaying(player: AVAudioPlayer, suc
        print("finished playing \(flag)")
    }

    func audioPlayerDecodeErrorDidOccur(player: AVAudioPlayer,
        if let e = error {
            print("\(e.localizedDescription)")
        }
    }
}
```

```
// MARK: AVAudioPlayerDelegate
extension Sound :AVAudioPlayerDelegate {
    func audioPlayerDidFinishPlaying(player: AVAudioPlayer, suc
        print("finished playing \(flag)")
    }

    func audioPlayerDecodeErrorDidOccur(player: AVAudioPlayer,
        if let e = error {
```

```
        print("\(e.localizedDescription)")
    }
}
```

---

Finally, the transport controls that can be called from an action.

```
func stopAVPlayer() {
    if avPlayer.playing {
        avPlayer.stop()
    }
}
```

```
func toggleAVPlayer() {
    if avPlayer.playing {
        avPlayer.pause()
    } else {
        avPlayer.play()
    }
}
```

---

```
func stopAVPlayer() {
    if avPlayer.playing {
        avPlayer.stop()
    }
}
```

```
func toggleAVPlayer() {
    if avPlayer.playing {
        avPlayer.pause()
    } else {
        avPlayer.play()
    }
}
```

```
}  
}
```

## Audio Session


The Audio Session singleton is an intermediary between your app and the media daemon. Your app and all other apps (should) make requests to the shared session. Since we are playing an audio file, we should tell the session that is our intention by requesting that its category be **AVAudioSessionCategoryPlayback**, and then make the session active. You should do this in the code above right before you call `play()` on the player.

Setting a session for playback.

```
func setSessionPlayback() {  
    let audioSession = AVAudioSession.sharedInstance()  
    do {  
        try audioSession.setCategory(AVAudioSessionCategoryPlayback,  
                                     withOptions: AVAudioSessionCategoryOptions.MixWithOthers)  
        try audioSession.setActive(true)  
    } catch {  
        print("couldn't set category \(error)")  
    }  
}
```

```
func setSessionPlayback() {  
    let audioSession = AVAudioSession.sharedInstance()  
    do {  
        try audioSession.setCategory(AVAudioSessionCategoryPlayback,  
                                     withOptions: AVAudioSessionCategoryOptions.MixWithOthers)
```

```
        try audioSession.setActive(true)
    } catch {
        print("couldn't set category \(error)")
    }
}
```



[Go to Table of Contents](#)

## Playing a MIDI file

You use **AVMIDIPlayer** to play standard MIDI files. Loading the player is similar to loading the **AVAudioPlayer**. You need to load a soundbank from a Soundfont or DLS file. The player also has a pre-roll **prepareToPlay()** function.

I'm not interested in copyright infringement, so I have not included either a DLS or SF2 file. So do a web search for a GM SoundFont2 file. They are loaded in the same manner. I've tried the MuseCore SoundFont and it sounds ok. There is probably a General MIDI DLS on your OSX system already:

/System/Library/Components/CoreAudio.component/Contents/Resources/gs\_instruments.dls. Copy this to the project bundle if you want to try it.


```
func createAVMIDIPlayerFromMIDIFileDLS() {

    guard let midiFileURL = NSBundle.mainBundle().URLForResource(
        fatalError("\(sibeliusGMajor.mid\" file not found.\"")
    }

    guard let bankURL = NSBundle.mainBundle().URLForResource("g
        fatalError("\(gs_instruments.dls\" file not found.\"")
    }
```

```
do {
    try self.midiPlayer = AVMIDIPlayer(contentsOfURL: midiF
    print("created midi player with sound bank url \(bankUR
} catch let error as NSError {
    print("Error \(error.localizedDescription)")
}

self.midiPlayer.prepareToPlay()
}
```




```
func createAVMIDIPlayerFromMIDIFileDLS() {

    guardlet midiFileURL = NSBundle.mainBundle().URLForResource
        fatalError("\(sibeliusGMajor.mid\" file not found."
    }

    guardlet bankURL = NSBundle.mainBundle().URLForResource("gs.
        fatalError("\(gs_instruments.dls\" file not found.")
    }

    do {
        try self.midiPlayer = AVMIDIPlayer(contentsOfURL: midiF
        print("created midi player with sound bank url \(bankUR
    } catch let error as NSError {
        print("Error \(error.localizedDescription)")
    }

    self.midiPlayer.prepareToPlay()
}
```



[Go to Table of Contents](#)



# Audio Engine

iOS 8 introduces a new audio engine which seems to be the successor to Core Audio's **AUGraph** and friends. See my article on using these classes in Swift.

The new **AVAudioEngine** class is the analog to **AUGraph**. You create **AudioNode** instances and attach them to the engine. Then you start the engine to initiate data flow.

Here is an engine that has a player node attached to it. The player node is attached to the engine's mixer. These are instance variables.

```
engine = AVAudioEngine()
playerNode = AVAudioPlayerNode()
engine.attachNode(playerNode)
mixer = engine.mainMixerNode
engine.connect(playerNode, to: mixer, format: mixer.outputFormat)
```

```
engine = AVAudioEngine()
playerNode = AVAudioPlayerNode()
engine.attachNode(playerNode)
mixer = engine.mainMixerNode
engine.connect(playerNode, to: mixer, format: mixer.outputFormat)
```

Then you need to start the engine.

```
func startEngine() {

    if engine.running {
        print("audio engine already started")
    }
}
```

```
        return
    }

    do {
        try engine.start()
        print("audio engine started")
    } catch {
        print("oops \(error)")
        print("could not start audio engine")
    }
}
```

---

```
func startEngine() {

    if engine.running {
        print("audio engine already started")
        return
    }

    do {
        try engine.start()
        print("audio engine started")
    } catch {
        print("oops \(error)")
        print("could not start audio engine")
    }
}
```

---

Cool. Silence.

Let's give it something to play. It can be an audio file, or as we'll see, a MIDI file or a computed buffer.

In this example we create an **AVAudioFile** instance from an MP3 file, and tell the `playerNode` to play it.

First, load an audio file. Or load an audio file into a buffer.

```
var buffer:AVAudioPCMBuffer!
```

```
func readFileIntoBuffer() {

    guard let fileURL = NSBundle.mainBundle().URLForResource("m
        print("could not read sound file")
        return
    }


    do {
        let file = try AVAudioFile(forReading: fileURL)
        buffer = AVAudioPCMBuffer(PCMFormat: file.processingFor
        try file.readIntoBuffer(buffer)
    } catch {
        print("could not create AVAudioPCMBuffer \(error)")
        return
    }
}
```

```
var buffer:AVAudioPCMBuffer!
```

```
func readFileIntoBuffer() {

    guard let fileURL = NSBundle.mainBundle().URLForResource("mo
        print("could not read sound file")
        return
    }
}
```


```
do {
    let file = try AVAudioFile(forReading: fileURL)
    buffer = AVAudioPCMBuffer(PCMFormat: file.processingFormat)
    try file.readIntoBuffer(buffer)
} catch {
    print("could not create AVAudioPCMBuffer \(error)")
    return
}
}
```



Now we hand the buffer to the player node by “scheduling” it, then playing it.

```
func finishedPlaying() {
    print("finished playing")
    playerNode.stop()
}

func togglePlayer() {
    if playerNode.playing {
        playerNode.stop()
    } else {
        setSessionPlayback()
        startEngine()
        playerNode.scheduleBuffer(buffer, completionHandler: nil)
        playerNode.play()
    }
}
```



```
func finishedPlaying() {
    print("finished playing")
    playerNode.stop()
}
```

```
}

func togglePlayer() {
    if playerNode.playing {
        playerNode.stop()
    } else {
        setSessionPlayback()
        startEngine()
        playerNode.scheduleBuffer(buffer, completionHandler: {
            playerNode.play()
        })
    }
}
```



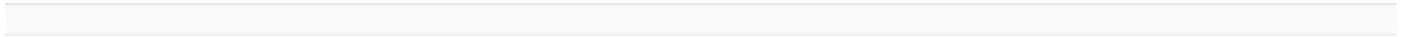
There are quite a few variations on `scheduleBuffer`. Have fun trying them out.

[Go to Table of Contents](#)

## Playing MIDI Notes

How about triggering MIDI notes/events based on UI events? You need an instance of **AVAudioUnitMIDISynthesizer** among your nodes. There is one concrete subclass named **AVAudioUnitSampler**. Create a sampler and attach it to the engine.

```
engine = AVAudioEngine()
sampler = AVAudioUnitSampler()
engine.attachNode(sampler)
engine.connect(sampler, to: engine.mainMixerNode, format: nil)
```



```
engine = AVAudioEngine()
sampler = AVAudioUnitSampler()
```

```
engine.attachNode(sampler)
engine.connect(sampler, to: engine.mainMixerNode, format: nil)
```

In your UI's action function, load the appropriate instrument into the sampler. The program parameter is a General MIDI instrument number. You might want to set up constants. Soundbanks have banks of sound. You need to specify which bank to use with the bankMSB and bankLSB. I use Core Audio constants here to choose the "melodic" bank and not the "percussion" bank.

```
// instance variables
let melodicBank = UInt8(kAUSampler_DefaultMelodicBankMSB)
let defaultBankLSB = UInt8(kAUSampler_DefaultBankLSB)
let gmMarimba = UInt8(12)
let gmHarpsichord = UInt8(6)

func loadPatch(gmpatch:UInt8, channel:UInt8 = 0) {

    guard let soundbank =
        NSBundle.mainBundle().URLForResource("GeneralUser GS Mu
    else {
        print("could not read sound font")
        return
    }

    do {
        try sampler.loadSoundBankInstrumentAtURL(soundbank, pro
            bankMSB: melodicBank, bankLSB: defaultBankLSB)

    } catch let error as NSError {
        print("\(error.localizedDescription)")
        return
    }
}
```

```

        self.sampler.sendProgramChange(gmpatch, bankMSB: melodicBank)
    }

```

```

// instance variables
let melodicBank = UInt8(kAUSampler_DefaultMelodicBankMSB)
let defaultBankLSB = UInt8(kAUSampler_DefaultBankLSB)
let gmMarimba = UInt8(12)
let gmHarpsichord = UInt8(6)

func loadPatch(gmpatch:UInt8, channel:UInt8 = 0) {

    guardlet soundbank =
        NSBundle.mainBundle().URLForResource("GeneralUser GS Mu
        else {
            print("could not read sound font")
            return
        }

    do {
        try sampler.loadSoundBankInstrumentAtURL(soundbank, program:
            bankMSB: melodicBank, bankLSB: defaultBankLSB)

    } catch let error as NSError {
        print("\(error.localizedDescription)")
        return
    }

    self.sampler.sendProgramChange(gmpatch, bankMSB: melodicBank)
}

```

Then send a MIDI program change by calling our load function. After that, you can send **startNote** and **stopNote** messages to the sampler. You need

to match the parameters for each start and stop message.

```
loadPatch(gmHarpsichord)
// play middle C, mezzo forte on MIDI channel 0
self.sampler.startNote(60, withVelocity: 64, onChannel: 0)
...
// in another action
self.sampler.stopNote(60, onChannel: 0)
```

---

```
loadPatch(gmHarpsichord)
// play middle C, mezzo forte on MIDI channel 0
self.sampler.startNote(60, withVelocity: 64, onChannel: 0)
...
// in another action
self.sampler.stopNote(60, onChannel: 0)
```

---

[Go to Table of Contents](#)

## Summary

This is a good start I hope. There are other things I'll cover soon, such as generating and processing the audio buffer data.

## Resources