Sample Code

Incorporating Audio Effects and Instruments

Add custom audio processing and MIDI instruments to your app by hosting Audio Unit (AU) plug-ins.

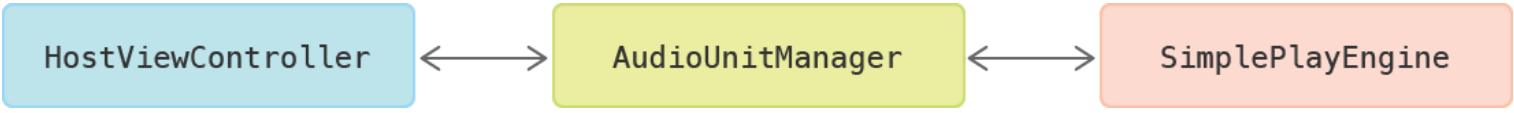
Download

iOS 13.0+ | iPadOS 13.0+ | Mac Catalyst 13.0+ | macOS 10.15+ | Xcode 16.0+

Overview

This sample app shows you how to use AU plug-ins in your iOS and macOS apps. You find and instantiate plug-ins, incorporate their user interfaces into your app's interface, and work with their presets.

The sample app has targets for iOS and macOS. Both versions have three primary classes.



- `HostViewController` and its associated Storyboard provide the user interface.
- `AudioUnitManager` manages the interactions with the effect and instrument plug-ins.
- `SimplePlayEngine` uses AVAudioEngine to play back audio samples and MIDI data.

Find Audio Units

You find Audio Units that are registered with the host system by creating an `AudioComponentDescription` defining your search criteria. The sample app searches for component types, either audio effects (`kAudioUnitType_Effect`) or MIDI instruments (`kAudioUnitType`

`_MusicDevice`). You can also pass values for the other fields of `AudioComponentDescription` or pass `0` as a wildcard matching all values. Get the shared instance of AVAudioUnitComponentManager and call its components(matching:) method to find the components matching your search criteria.

```
// Make a component description matching any Audio Unit.
let description = AudioComponentDescription(componentType: 0,
                                            componentSubType: 0,
                                            componentManufacturer: 0,
                                            componentFlags: 0,
                                            componentFlagsMask: 0)

let components = AVAudioUnitComponentManager.shared().components(matching: descripti
```

This method returns an array of AVAudioUnitComponent objects matching the component description, or an empty array if it found no matches. You can access a component's properties to determine its capabilities and find identifying values, such as its name and manufacturer, for display in your user interface.

Instantiate Audio Units

When the user selects an Audio Unit in the user interface, your app needs to find the component and instantiate it.

iOS supports third-party plug-ins built using the latest Audio Unit standard (AUv3), which is based on the App Extensions model. Like all App Extensions in iOS, AUv3 plug-ins run *out-of-process*, which means they run in a dedicated process outside your app, and communication with the extension is done over interprocess communication (IPC).

You instantiate an AU by calling the instantiate(with:options:completionHandler:) method, passing it the component description. This method asynchronously returns the instantiated `AVAudioUnit` or an `Error` if the process failed. You must avoid blocking your application's main thread when instantiating an Audio Unit.

```
// Instantiate the Audio Unit
AVAudioUnit.instantiate(with: description) { avAudioUnit, error in
    // Use Audio Unit or handle error
}
```

In macOS, AUv3 plug-ins also default to running out-of-process. Running an Audio Unit this way is safer and more secure, because a misbehaving plug-in can't corrupt or crash your app. However, the interprocess communication required of this model adds some small but potentially significant

overhead. This can be problematic in professional audio environments where multiple Audio Units are used, especially when rendering at small audio I/O buffer sizes. To resolve this problem, AU authors can package their plug-ins to be run *in-process*. In macOS only, you can load an appropriately packaged plug-in in-process by passing that instantiation option to the `instantiate` method, as shown below.

```
let options: AudioComponentInstantiationOptions = .loadInProcess

// Instantiate the Audio Unit
AVAudioUnit.instantiate(with: description, options: options) { avAudioUnit, error in
    // Use Audio Unit or handle error
}
```

> **Note**
>
> iOS and macOS support using existing AUv2 plug-ins. iOS supports only those provided by the operating system, but macOS supports third-party AUv2 plug-ins as well. In both platforms, these plug-ins are *always* run as part of the host app's process.

Present an Audio Unit's Custom View

A plug-in can provide a custom user interface to control its parameters. You get the custom view by asking the plug-in for its view controller, which returns an instance of AUViewController, or `nil` if it doesn't provide a custom view. You add the view controller's view to your user interface using the appropriate approach for your platform.

```
func loadAudioUnitViewController(completion: @escaping (ViewController?) -> Void) {
    if let avAudioUnit = avAudioUnit {
        // Call our AVAudioUnit extension to request the ViewController
        // We will obtain a generic view if the plugin does not provide a custom vie
        avAudioUnit.requestViewController(completion: completion)
    } else {
        completion(nil)
    }
}
```

Select Alternative View Configurations

All AU plug-ins can provide a custom user interface, but AUv3 plug-ins may also provide alternative views. A host app can support multiple view configurations. For example, an iOS app may provide compact and expanded views and switch between them depending on the device size or orientation. You define one or more supported view configurations using the AUAudioUnitViewConfiguration class.

```
private var currentViewConfigurationIndex = 1

/// View configurations supported by the host app
private var viewConfigurations: [AUAudioUnitViewConfiguration] = {
    let compact = AUAudioUnitViewConfiguration(width: 400, height: 100, hostHasContr
    let expanded = AUAudioUnitViewConfiguration(width: 800, height: 500, hostHasCont
    return [compact, expanded]
}()
```

> **Note**
>
> The view configuration object's hostHasController property indicates whether the host app should show its control surface for the view configuration. The host app should respect this setting and update its user interface accordingly.

The host can ask the plug-in which, if any, custom view configurations it supports.

```
let supportedConfigurations = audioUnit.supportedViewConfigurations(viewConfiguratio
```

When the host switches between its supported configurations, it can ask the Audio Unit to do the same. The sample app defines two configurations and attempts to toggle between them.

```
/// Toggles the current view mode (compact or expanded)
func toggleViewMode() {
    guard let audioUnit = audioUnit else { return }
    currentViewConfigurationIndex = currentViewConfigurationIndex == 0 ? 1 : 0
    audioUnit.select(viewConfigurations[currentViewConfigurationIndex])
}
```

Load Factory Presets

A plug-in author can optionally provide one or more presets that define specific configurations of the plug-in's parameter values. You access an AUAudioUnit object's presets through its

`factoryPresets` property, which returns an array of `AUAudioUnitPreset` instances, or an empty array if it defines none.

```
/// Gets the audio unit's factory presets.
public var factoryPresets: [Preset] {
    guard let presets = audioUnit?.factoryPresets else { return [] }
    return presets.map { Preset(preset: $0) }
}
```

The sample app uses a simple wrapper type called `Preset` to pass to the user interface tier. The view controller uses these objects to build the app's preset selection interface.

Manage User Presets

A plug-in may also support *user presets*, which are user-configured parameter settings. You query the Audio Unit's `supportsUserPresets` property to determine if it supports saving user presets.

```
var supportsUserPresets: Bool {
    return audioUnit?.supportsUserPresets ?? false
}
```

If a plug-in supports user presets, you can get the currently saved presets by querying its `userPresets` property.

```
/// Gets the audio unit's user presets.
public var userPresets: [Preset] {
    guard let presets = audioUnit?.userPresets else { return [] }
    return presets.map { Preset(preset: $0) }.reversed()
}
```

To be notified of changes to the Audio Unit's user presets, you add a key-value observer to the `userPresets` property. By observing changes to this property, you'll get callbacks as presets are added or deleted.

```
// Add key-value observer to the userPresets property.
observation = audioUnit?.observe(\.userPresets) { _, _ in
    // User presets changed. Update the user interface.
}
```

To create a new user preset, first create an instance of `AUAudioUnitPreset` and give it a user-defined name and a negative `number` value (user presets require a negative value for this property). Then call the `saveUserPreset(_:)` method, which persists the parameter state so the Audio Unit can recall it later.

```
let preset = AUAudioUnitPreset()
preset.name = "A Custom Preset"
preset.number = -1

// Save the preset's parameter state.
do {
    try audioUnit.saveUserPreset(preset)
} catch {
    // Handle the error.
}
```

If the user decides to delete this or another user preset, you call `deleteUserPreset(_:)` to remove it.

Select Factory and User Presets

To select a factory or user preset, set it as the Audio Unit's `currentPreset` property. This restores the plug-in's parameter state to the values stored with the specified preset.

```
/// Get or set the audio unit's current preset.
public var currentPreset: Preset? {
    get {
        guard let preset = audioUnit?.currentPreset else { return nil }
        return Preset(preset: preset)
    }
    set {
        audioUnit?.currentPreset = newValue?.audioUnitPreset
    }
}
```

See Also

Audio Units

Creating an audio unit extension

Build an extension by using an Xcode template.

{} Creating custom audio effects

Add custom audio-effect processing to apps like Logic Pro X and GarageBand by creating Audio Unit (AU) plug-ins.

Debugging Out-of-Process Audio Units on Apple Silicon

Connect to out-of-process audio units using the Xcode debugger.

`class AUAudioUnit`

A class that defines a host's interface to an audio unit.

`class AUAudioUnitBus`

A class that defines an input or output connection point on an audio unit.

`class AUAudioUnitBusArray`

A class that defines a container for an audio unit's input or output busses.

`class AUAudioUnitPreset`

A class that describes an interface for custom parameter settings provided by the audio unit developer.

`class AUAudioUnitV2Bridge`

A class that wraps a version 2 audio unit as version 3 audio unit.

`func AudioUnitExtensionCopyComponentList(CFString) -> Unmanaged<CFArray>?`

Returns the component registrations for a given audio unit extension.

`func AudioUnitExtensionSetComponentList(CFString, CFArray?) -> OSStatus`

Allows the implementor of an audio unit extension to dynamically modify the list of component registrations for the extension.

`protocol AUAudioUnitFactory`

An object that creates a version 3 audio unit.