Sample Code

# Creating custom audio effects

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

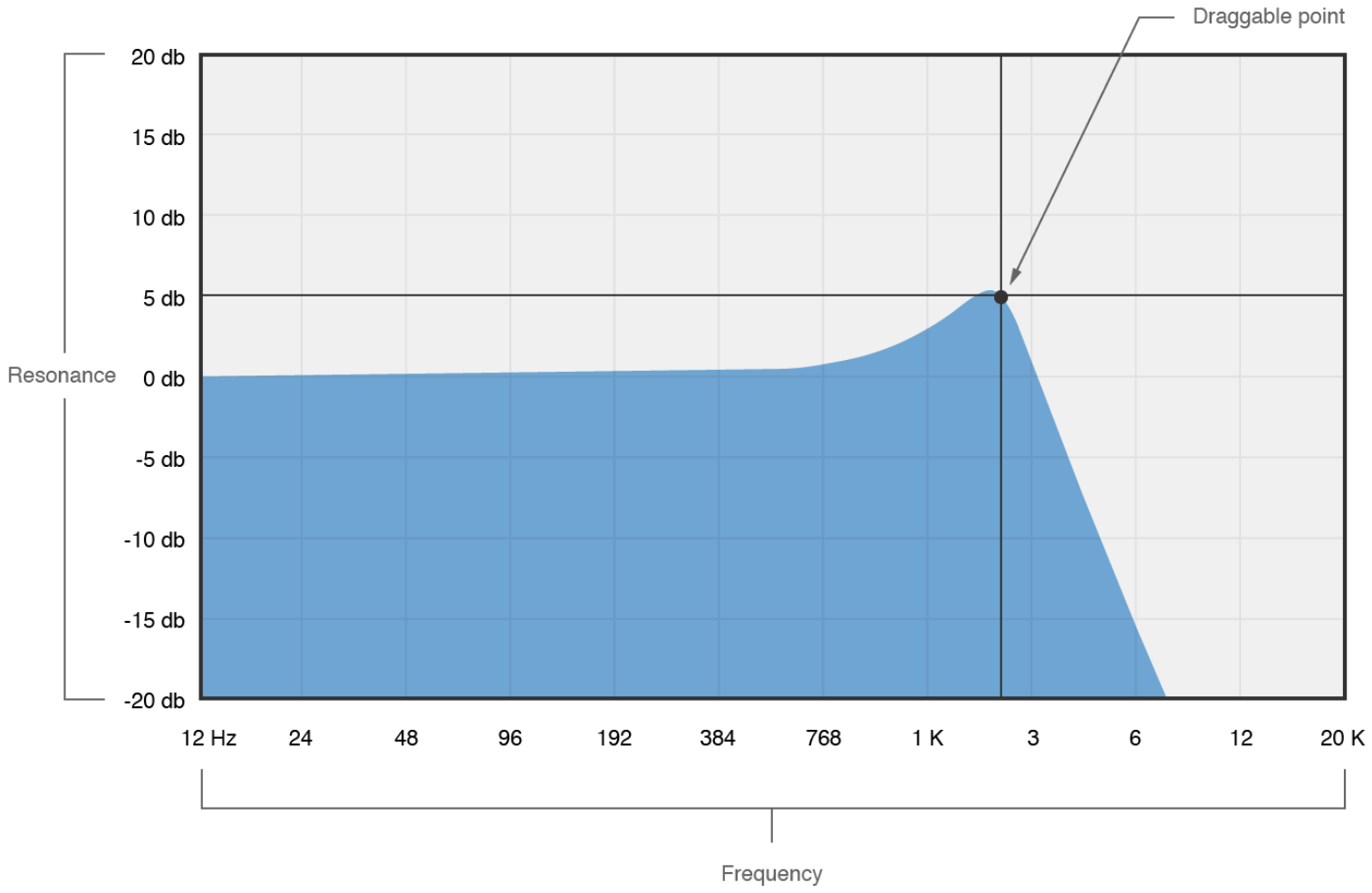
Download

iOS 17.0+ | iPadOS 17.0+ | Mac Catalyst 17.0+ | macOS 14.0+ | Xcode 15.4+

## Overview

This sample app shows you how to create a custom audio effect plug-in using the latest Audio Unit standard (AUv3). The AUv3 standard builds on the App Extensions model, which means you deliver your plug-in as an extension that's contained in an app distributed through the App Store or your own store.

The sample Audio Unit is a low-pass filter that allows frequencies at or below the cutoff frequency to pass through to the output. It attenuates frequencies above this point. It also lets you change the filter's resonance, which boosts or attenuates a narrow band of frequencies around the cutoff point. You set these values by moving the draggable point around the plug-in's user interface, as shown in the figure below.




The project has targets for both iOS and macOS. Each platform's main app target has two supporting targets: `AUv3FilterExtension`, which contains the plug-in packaged as an Audio Unit extension, and `AUv3FilterFramework`, which bundles the plug-in's code and resources.

> **Note**
>
> See Incorporating Audio Effects and Instruments for details on how you can use this Audio Unit extension in a host app.

## Create a Custom Audio Effect Plug-In

The extension itself contains two primary pieces: an Audio Unit proper and a factory object that creates it.

The sample app's Audio Unit is `AUv3FilterDemo`. This is a Swift class that subclasses AUAudioUnit and defines the plug-in's interface, including key features like its parameters, presets, and I/O busses. A class called `FilterDSPKernel` provides the plug-in's digital signal processing (DSP) logic, and is written in C++ to ensure real-time safety. Because Swift can't talk directly to C++, the sample project also includes an Objective-C++ adapter class called `FilterDSPKernelAdapter` to act as an intermediary.

`AUv3FilterDemoViewController` is the Audio Unit's main view controller. It adopts the AUAudioUnitFactory protocol and is responsible for creating new instances of your plug-in. You implement the protocol's createAudioUnit(with:) factory method to return a new instance of `AUv3FilterDemo` when a host app requests it.

```
extension AUv3FilterDemoViewController: AUAudioUnitFactory {
    public func createAudioUnit(with componentDescription: AudioComponentDescriptior
        audioUnit = try AUv3FilterDemo(componentDescription: componentDescription, c
        return audioUnit!
    }
}
```

**Important**

To ensure glitch-free performance, your plug-in's audio processing must occur in a real-time safe context. Don't allocate memory, perform file I/O, take locks, or interact with the Swift or Objective-C runtimes when rendering audio.

## Add Custom Parameters to Your Audio Unit

In most Audio Units, you'll provide one or more parameters to configure the audio processing. Your Audio Unit arranges its parameters into a tree structure, provided by an instance of AUParameterTree. This object represents the root node of the plug-in's tree of parameters and parameter groupings.

`AUv3FilterDemo` has parameters to control the filter's cutoff frequency and resonance. You create its parameters using a factory method on `AUParameterTree`.

```
private enum AUv3FilterParam: AUParameterAddress {
    case cutoff, resonance
}

/// The parameter to control the cutoff frequency (12 Hz – 20 kHz).
var cutoffParam: AUParameter = {
    let parameter =
        AUParameterTree.createParameter(withIdentifier: "cutoff",
                                        name: "Cutoff",
                                        address: AUv3FilterParam.cutoff.rawValue,
                                        min: 12.0,
                                        max: 20_000.0,
                                        unit: .hertz,
```

```
                                        unitName: nil,
                                        flags: [.flag_IsReadable,
                                                .flag_IsWritable,
                                                .flag_CanRamp],
                                        valueStrings: nil,
                                        dependentParameters: nil)
    // Set default value
    parameter.value = 0.0

    return parameter
}()

/// The parameter to control the cutoff frequency's resonance (+/-20 dB).
var resonanceParam: AUParameter = {
    let parameter =
        AUParameterTree.createParameter(withIdentifier: "resonance",
                                        name: "Resonance",
                                        address: AUv3FilterParam.resonance.rawValue,
                                        min: -20.0,
                                        max: 20.0,
                                        unit: .decibels,
                                        unitName: nil,
                                        flags: [.flag_IsReadable,
                                                .flag_IsWritable,
                                                .flag_CanRamp],
                                        valueStrings: nil,
                                        dependentParameters: nil)
    // Set the default value.
    parameter.value = 20_000.0

    return parameter
}()
```

The cutoff parameter defines a frequency range between 12 Hz and 20 kHz, and the resonance parameter defines a decibel range between -20 dB and 20 dB. Each parameter is readable and writeable, and also supports ramping, which means you can modify its value over time.

You arrange the parameters into a tree by creating an `AUParameterTree` instance and setting them as the tree's children.

```
// Create the audio unit's tree of parameters.
parameterTree = AUParameterTree.createTree(withChildren: [cutoffParam,
                                                          resonanceParam])
```

Next, you bind handlers to the parameter tree's readable and writeable values by installing closures for its implementorValueObserver, implementorValueProvider, and implementorStringFromValueCallback properties. These closures delegate to the filter adapter instance, which in turn communicates with the underlying DSP logic.

```
// A closure for observing all externally generated parameter value changes.
parameterTree.implementorValueObserver = { param, value in
    kernelAdapter.setParameter(param, value: value)
}

// A closure for returning state of the requested parameter.
parameterTree.implementorValueProvider = { param in
    return kernelAdapter.value(for: param)
}

// A closure for returning the string representation of the requested parameter valu
parameterTree.implementorStringFromValueCallback = { param, value in
    switch param.address {
    case AUv3FilterParam.cutoff.rawValue:
        return String(format: "%.f", value ?? param.value)
    case AUv3FilterParam.resonance.rawValue:
        return String(format: "%.2f", value ?? param.value)
    default:
        return "?"
    }
}
```

## Connect the Parameters to Your User Interface

The sample app's iOS and macOS targets each provide a platform-specific user interface. You use a shared view controller called `AUv3FilterDemoViewController` to coordinate the communication between the user interface and the Audio Unit. Connect your user interface to the Audio Unit's parameters in the `connectViewToAU()` method.

```
private func connectViewToAU() {
    guard needsConnection, let paramTree = audioUnit?.parameterTree else { return }
```

```
    // Find the cutoff and resonance parameters in the parameter tree.
    guard let cutoff = paramTree.value(forKey: "cutoff") as? AUParameter,
        let resonance = paramTree.value(forKey: "resonance") as? AUParameter else {
            fatalError("Required AU parameters not found.")
    }

    // Set the instance variables.
    cutoffParameter = cutoff
    resonanceParameter = resonance

    // Observe major state changes like a user selecting a user preset.
    observer = audioUnit?.observe(\.allParameterValues) { object, change in
        DispatchQueue.main.async {
            self.updateUI()
        }
    }

    // Observe value changes to the cutoff and resonance parameters.
    parameterObserverToken =
        paramTree.token(byAddingParameterObserver: { [weak self] address, value in
            guard let self = self else { return }

            // An arbitrary queue is calling this closure. Ensure
            // all UI updates dispatch back to the main thread.
            if [cutoff.address, resonance.address].contains(address) {
                DispatchQueue.main.async {
                    self.updateUI()
                }
            }
        })

    // Indicate the view and the audio unit have a connection.
    needsConnection = false

    // Sync the UI with the parameter state.
    updateUI()
}
```

As shown above, in the `connectViewToAU()` method, you find the Audio Unit's parameter tree and retrieve its cutoff and resonance parameters. You also add an observer closure to update the user interface as the plug-in's parameter values change.

## Add Factory Presets

Most audio plug-ins provide a collection of preset values known as *factory presets*. A factory preset is a preconfigured arrangement of the plug-in's parameter values that provide a useful starting point for further customization. A host app presents these presets in its user interface so the user can select them.

The following code example shows how to define the factory presets and their associated values.

```
public override var factoryPresets: [AUAudioUnitPreset] {
    return [
        AUAudioUnitPreset(number: 0, name: "Prominent"),
        AUAudioUnitPreset(number: 1, name: "Bright"),
        AUAudioUnitPreset(number: 2, name: "Warm")
    ]
}

private let factoryPresetValues:[(cutoff: AUValue, resonance: AUValue)] = [
    (2500.0, 5.0),    // "Prominent"
    (14_000.0, 12.0), // "Bright"
    (384.0, -3.0)     // "Warm"
]
```

## Support User Presets

Factory presets provide a useful starting point for further user customization, but users also want the ability to save their changes and create their own custom presets. `AUAudioUnit` provides built-in support for user presets. To enable this support in your Audio Unit, override the supportsUserPresets property to return `true`.

```
/// Indicates that this audio unit supports persisting user presets.
public override var supportsUserPresets: Bool {
    return true
}
```

Opting in to support for user presets automatically enables your Audio Unit to load, save, and delete user presets. The default implementation of the userPresets, saveUserPreset(), and deleteUserPreset() API reads from and writes to an internal store, but you're free to override this property and methods if you want to directly manage the persistence behavior. For example, you can override the default behavior to persist user presets to an iCloud container or some other remote location.

# Select Factory and User Presets

A host app selects a factory or user preset by setting the plug-in's `currentPreset` property. You override this property and take the appropriate action depending on the preset type selected. If the user selected a factory preset (a preset `number` greater than `0`), look up its associated values and set the parameter values accordingly. If the user selected a user preset (a preset `number` less than `0`), restore the preset's parameter state by calling the presetState(for:) method and setting the returned data as the fullStateForDocument property.

```
private var _currentPreset: AUAudioUnitPreset?

/// The currently selected preset.
public override var currentPreset: AUAudioUnitPreset? {
    get { return _currentPreset }
    set {
        // If the newValue is nil, return.
        guard let preset = newValue else {
            _currentPreset = nil
            return
        }

        // Factory presets need to always have a number >= 0.
        if preset.number >= 0 {
            let values = factoryPresetValues[preset.number]
            parameters!.setParameterValues(cutoff: values.cutoff, resonance: values.
            _currentPreset = preset
        }
        // User presets are always negative.
        else {
            // Attempt to restore the archived state for this user preset.
            do {
                fullStateForDocument = try presetState(for: preset)
                // Set the currentPreset after successfully restoring the state.
                _currentPreset = preset
            } catch {
                print("Unable to restore set for preset \(preset.name)")
            }
        }
    }
}
```

# Package Your Plug-In to Run In-Process

Like all App Extensions, AUv3 plug-ins run *out-of-process* by default, which means the extension runs in a separate process from the host app, and all communication between the two occurs over interprocess communication (IPC). This model provides increased security and stability for the host app. For example, if an AUv3 plug-in crashes, the host app won't crash. However, the IPC communication adds a small amount of overhead to each render cycle, which may be unacceptable depending on the needs of a given application. In macOS only, you can package your plug-in to run *in-process*, which eliminates the IPC communication as your Audio Unit runs as part of the host's process.

Running an in-process plug-in requires an agreement between the host and the Audio Unit. The host requests in-process instantiation by passing the .loadInProcess option during the plug-in's creation, and you need to package your Audio Unit as described and shown below.

Your extension's main binary can't be dynamically loaded into another app, which means all executable code needs to reside in a separate framework bundle. However, the extension target still needs to contain at least one source file for the extension binary to be created, properly loaded, and linked with the framework bundle. To ensure the extension is created, add some unused placeholder code in your extension target, like that found in `AUv3FilterExtension.swift`.

```
import AUv3FilterFramework

func placeholder() {
    // This placeholder function ensures the extension loads correctly.
}
```

The macOS sample packages all of the Audio Unit's code into the `AUv3FilterFramework` target. You indicate that the extension's code exists in a separate bundle by adding an `AudioComponentBundle` extension attribute to the target's `Info.plist` file.

```
<key>NSExtension</key>
<dict>
    <key>NSExtensionAttributes</key>
    <dict>
        <key>AudioComponentBundle</key>
        <string>com.example.apple-samplecode.AUv3FilterFramework</string>
        ...
    </dict>
    ...
</dict>
```

If you're using a xib or Storyboard for your user interface, override your view controller's init(nibName:bundle:) initializer and pass the framework bundle to the superclass initializer. This ensures your user interface properly loads when the system requests your Audio Unit extension.

```
public override init(nibName: NSNib.Name?, bundle: Bundle?) {
    // Pass a reference to the owning framework bundle.
    super.init(nibName: nibName, bundle: Bundle(for: type(of: self)))
}
```

Finally, in the extension's `Info.plist` file, set the Audio Unit's factory object, `AUv3FilterDemoViewController`, as the extension's principal class.

```
<key>NSExtension</key>
<dict>
    <key>NSExtensionPrincipalClass</key>
    <string>AUv3FilterFramework.AUv3FilterDemoViewController</string>
    ...
</dict>
```

> **Note**
>
> See Incorporating Audio Effects and Instruments for a host app you can use to load your plug-in both in-process and out-of-process.

# See Also

## Effects

- Audio Units

  The data type for a plug-in component that provides audio processing or audio data generation.