
Audio Unit Programming Guide



2006-11-07



Apple Computer, Inc.
© 2006 Apple Computer, Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Computer, Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled or Apple-licensed computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, Carbon, Cocoa, Final Cut, Final Cut Pro, Logic, Mac, Mac OS, Macintosh, QuickTime, and Xcode are trademarks of Apple Computer, Inc., registered in the United States and other countries.

eMac, Finder, GarageBand, and Tiger are trademarks of Apple Computer, Inc.

Objective-C is a registered trademark of NeXT Software, Inc.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction [Introduction](#) 9

[Who Should Read This Document?](#) 9
[Organization of This Document](#) 10
[Making Further Progress in Audio Unit Development](#) 10
[Required Tools for Audio Unit Development](#) 11
[See Also](#) 11

Chapter 1 [Audio Unit Development Fundamentals](#) 13

[The Audio Unit Development Cycle](#) 13
[What Is An Audio Unit?](#) 14
 [Audio Unit Programmatic Structure and Life Cycle](#) 14
 [Audio Unit File Structure](#) 16
 [Some Basic Terminology](#) 19
[Audio Units as Plug-Ins](#) 20
 [The Nature of Plug-Ins](#) 20
 [Tutorial: Using an Audio Unit in a Host Application](#) 20
 [The Role of the Core Audio SDK](#) 28
 [Component Manager Requirements for Audio Units](#) 28
 [Plug-in API Requirements for Audio Units](#) 30
[Audio Units as Instances of the Model-View-Controller Design Pattern](#) 32
[Audio Units in Action](#) 32
 [Opening and Closing Audio Units](#) 32
 [Audio Processing Graphs and the Pull Model](#) 34
 [Processing: The Heart of the Matter](#) 36
 [Supporting Parameter Automation](#) 37
[Audio Unit Validation and Testing](#) 38
 [Audio Unit Validation with the auval Tool](#) 38
 [Audio Unit Testing and Host Applications](#) 39

Chapter 2 [The Audio Unit](#) 43

[Audio Unit Architecture](#) 43
 [Audio Unit Scopes](#) 44
 [Audio Unit Elements](#) 45
 [Audio Unit Connections](#) 46
 [Audio Unit Channels](#) 48

Creating an Audio Unit by Subclassing	49
Control Code: Parameters, Factory Presets, and Properties	50
Defining and Using Parameters	50
Factory Presets and Parameter Persistence	51
Defining and Using Properties	51
Synthesis, Processing, and Data Format Conversion Code	55
Signal Processing	55
Music Synthesis	56
Music Effects	56
Data Format Conversion	56
Audio Unit Life Cycle	56
Overview	57
Categories of Programmatic Events	57
Bringing an Audio Unit to Life	58
Property Configuration	58
Audio Unit Initialization and Uninitialization	59
Kernel Instantiation in n-to-n Effect Units	60
Audio Processing Graph Interactions	60
Audio Unit Processing	61
Closing	62

Chapter 3 The Audio Unit View 63

Types of Views	63
Separation of Concerns	63
The Generic View	64
Custom Views	65
View Instantiation and Initialization	66
Parameter and Property Events	68
The Audio Unit Event API	68
Basic Parameter Adjustments	69
Parameter Adjustments with Notification	69
Parameter Gestures	70
Tutorial: Demonstrating Parameter Gestures and Audio Unit Events	70

Chapter 4 A Quick Tour of the Core Audio SDK 83

Obtaining the Core Audio SDK	83
Navigating within the Core Audio SDK	84
The AudioUnits Folder	84
The Documentation Folder	85
The PublicUtility Folder	86
The Services Folder	86

Chapter 5 **Tutorial: Building a Simple Effect Unit with a Generic View** 89

Overview	89
Install the Core Audio Development Kit	90
Specify the Function of Your Audio Unit	92
Design the Parameter Interface	92
Design the Factory Presets	94
Collect Configuration Information for the Audio Unit Bundle	95
Set Your Company Name in Xcode	95
Create and Configure the Project	96
Test the Unmodified Audio Unit	106
Implement the Parameter Interface	114
Name the Parameters and Set Values	115
Edit the Constructor Method	116
Define the Parameters	117
Provide Strings for the Waveform Pop-up Menu	119
Implement the Factory Presets Interface	120
Name the Factory Presets and Give Them Values	121
Add Method Declarations for Factory Presets	122
Set the Default Factory Preset	122
Implement the GetPresets Method	123
Define the Factory Presets	124
Implement Signal Processing	125
DSP Design for the Tremolo Effect	126
Define Member Variables in the Kernel Class Declaration	126
Write the TremoloUnitKernel Constructor Method	128
Override the Process Method	129
Override the Reset Method	133
Implement the Tail Time Property	133
Validate your Completed Audio Unit	133
Test your Completed Audio Unit	135

Chapter 6 **Appendix: Audio Unit Class Hierarchy** 139

Core Audio SDK Audio Unit Class Hierarchy	139
Starting Points for Common Audio Units	140

Document Revision History 141

C O N T E N T S

Figures, Tables, and Listings

Chapter 1 Audio Unit Development Fundamentals 13

- Figure 1-1 A running audio unit, built using the Core Audio SDK 15
- Figure 1-2 An audio unit in the Mac OS X file system 17
- Figure 1-3 An audio unit view in the Mac OS X file system 18
- Figure 1-4 The Apple audio units in the Mac OS X file system 19
- Figure 1-5 Constructing an audio unit version number 30
- Figure 1-6 Multiple instantiation of audio units in AU Lab 34
- Figure 1-7 The pull model in action with two effect units 35

Chapter 2 The Audio Unit 43

- Figure 2-1 Audio unit architecture for an effect unit 43
- Table 2-1 Using a channel information structure 53
- Listing 2-1 Using “scope” in the GetProperty method 44
- Listing 2-2 Audio unit scopes 44
- Listing 2-3 Using “element” in the GetProperty method 45
- Listing 2-4 The audio stream description structure 46
- Listing 2-5 The audio unit connection structure 47
- Listing 2-6 The render callback 47
- Listing 2-7 The AudioUnitRender function 48
- Listing 2-8 The audio buffer structure 48
- Listing 2-9 The audio buffer list structure 49
- Listing 2-10 The GetPropertyInfo method from the SDK’s AUBase class 54
- Listing 2-11 The GetProperty method from the SDK’s AUBase class 54
- Listing 2-12 The Process method from the AUKernelBase class 61

Chapter 3 The Audio Unit View 63

- Table 3-1 User interface items in an audio unit generic view 64
- Listing 3-1 A host application gets a Cocoa custom view from an audio unit 66
- Listing 3-2 The AudioUnitEvent structure 68
- Listing 3-3 The AudioUnitEventType enumeration 69
- Listing 3-4 The SetParameter convenience method 69

Chapter 5 Tutorial: Building a Simple Effect Unit with a Generic View 89

- Figure 5-1 Monaural tremolo 90

Figure 5-2	Confirming that the audio unit templates are installed	91
Table 5-1	Specification of tremolo frequency parameter	92
Table 5-2	Specification of tremolo depth parameter	93
Table 5-3	Specification of tremolo waveform parameter	93
Table 5-4	Specification of Slow & Gentle factory preset	94
Table 5-5	Specification of Fast & Hard factory preset	94
Listing 5-1	Parameter names and values (TremoloUnit.h)	115
Listing 5-2	Setting parameters in the constructor (TremoloUnit.cpp)	116
Listing 5-3	The customized GetParameterInfo method (TremoloUnit.cpp)	117
Listing 5-4	The customized GetParameterValueStrings method (TremoloUnit.cpp)	119
Listing 5-5	Factory preset names and values (TremoloUnit.h)	121
Listing 5-6	Factory preset method declarations (TremoloUnit.h)	122
Listing 5-7	Setting the default factory preset in the constructor (TremoloUnit.cpp)	122
Listing 5-8	Implementing the GetPresets method (TremoloUnit.cpp)	123
Listing 5-9	Defining factory presets in the NewFactoryPresetSet method (TremoloUnit.cpp)	124
Listing 5-10	TremoloUnitKernel member variables (TremoloUnit.h)	126
Listing 5-11	Modifications to the TremoloUnitKernel Constructor (TremoloUnit.cpp)	128
Listing 5-12	The Process method (TremoloUnit.cpp)	130
Listing 5-13	The Reset method (TremoloUnit.cpp)	133
Listing 5-14	Implementing the tail time property (TremoloUnit.h)	133

Chapter 6 **Appendix: Audio Unit Class Hierarchy** 139

Figure 6-1	Core Audio SDK audio unit class hierarchy	140
------------	---	-----

Introduction

This document describes **audio units** and how to create them. Audio units are digital audio plug-ins based on Apple's world class Core Audio technology for Mac OS X.

As a hobbyist or a computer science student, you can design and build your own audio units to make applications like GarageBand do new things with sound.

As a commercial developer, you can create professional quality software components that provide features like filtering, reverb, dynamics processing, and sample-based looping. You can also create simple or elaborate MIDI-based music synthesizers, as well as more technical audio units such as time and pitch shifters and data format converters.

As part of Core Audio and being integral to Mac OS X, audio units offer a development approach for audio plug-ins that excels in terms of performance, robustness, and ease of deployment. With audio units, you also gain by providing a consistent, simple experience for end-users.

Your target market is wide, including performers, DJs, recording and mastering engineers, and anyone who likes to play with sound on their Macintosh.

Note: This first version of *Audio Unit Programming Guide* does not go into depth on some topics important to commercial audio unit developers such as copy protection, parameter automation, and custom views (graphical user interfaces for audio units). Nor does this version provide instruction on developing types of audio units other than the most common type, effect units.

Who Should Read This Document?

To use this document, you should already be familiar with the C programming language. You should be comfortable with using Xcode to create a Mac OS X plug-in as described in the *Xcode 2 User Guide*. For example, you should know about Xcode's various build options, such as ZeroLink, and when to use them. You should also know how and why to include frameworks and files in the linking phase of an Xcode build.

It's very helpful in using this document to have enough familiarity with the C++ programming language to read header and implementation files. It's also helpful to have a basic understanding of the Mac OS X Component Manager as described in *Component Manager for QuickTime*, as well as a grounding in digital audio coding and audio DSP (digital signal processing).

This document does not address the needs of audio unit host application developers, whose code opens, connects, and uses audio units. Nor is this document an audio unit cookbook. It devotes very few pages to DSP or music synthesis techniques, which work essentially the same way in audio units as in other audio software technologies

Organization of This Document

Depending on your needs and your learning style, you can use this document in the following ways.

- If you want to get your hands on building an audio unit right away, go straight to [“Tutorial: Building a Simple Effect Unit with a Generic View”](#) (page 89). As you build the audio unit, you can refer to other sections in this document for conceptual information related to what you’re doing.
- If you prefer to build your knowledge incrementally, starting with a solid conceptual foundation before seeing the technology in action, read the chapters in order.
- If you already have some familiarity with building your own audio units, you may want to go straight to [“The Audio Unit”](#) (page 43) and [“Appendix: Audio Unit Class Hierarchy”](#) (page 139). You might also want to review [“A Quick Tour of the Core Audio SDK”](#) (page 83) to see if the SDK contains some treasures you haven’t been using until now.

This document contains the following chapters:

- “Audio Unit Development Fundamentals”, a bird’s eye view of audio unit development, covering Xcode, the Core Audio SDK, design, development, testing, and deployment
- “The Audio Unit”, design and programming considerations for the part of an audio unit that performs the audio work
- “The Audio Unit View”, a description of the two audio unit view types—generic and custom—as well as an explanation of parameter automation
- “A Quick Tour of the Core Audio SDK”: Taking advantage of the code in the Core Audio SDK is the fastest route to audio unit development
- “Tutorial: Building a Simple Effect Unit with a Generic View”, a tutorial that takes you from zero to a fully functioning effect unit
- “Appendix: Audio Unit Class Hierarchy”, a tour of the audio unit class hierarchy provided by the Core Audio SDK

Making Further Progress in Audio Unit Development

To go forward in developing your own audio units based on what you learn here, you will need:

- The ability to develop plug-ins using the C++ programming language, because the audio unit class hierarchy in the Core Audio SDK uses C++.

- A grounding in audio DSP, including the requisite mathematics. Alternatively, you can work with someone who can provide DSP code for your audio units, along with someone who can straddle the audio unit and math worlds. For optimum quality and performance of an audio unit, DSP code needs to be correctly incorporated into the audio unit scaffolding.
- A grounding in MIDI, if you are developing instrument units.

Required Tools for Audio Unit Development

When you perform a full installation of the current version of Mac OS X, including Xcode Tools, you'll have everything you need on your system for audio unit development. These items are also available free from Apple's developer website, <http://developer.apple.com>:

- **The latest version of Xcode.** The examples in this document use Xcode version 2.4.
- **The latest Mac OS X header files.** The examples in this document use the header files in the 10.4.0 Mac OS SDK, installed with Apple's Xcode Tools.
- **The latest Core Audio development kit.** The examples in this document use Core Audio SDK v1.4.3, installed with Apple's Xcode Tools at this location on your system:
`/Developer/Examples/CoreAudio`
- **At least one audio unit hosting application.** Apple recommends the AU Lab application, installed with Apple's Xcode Tools at this location on your system: `/Developer/Applications/Audio/AU Lab`
- **The audio unit validation command-line tool,** `auval`, Apple's validation tool for audio units, provided with Mac OS X.

See Also

As you learn about developing audio units, you may find the following information and tools helpful:

- The [coreaudio-api mailing list](#), a very active discussion forum hosted by Apple that covers all aspects of audio unit design and development.
- Audio Unit framework API documentation in the [Core Audio SDK](#), available from Apple's [developer website](#).
- Additional audio unit API reference available in the [Core Audio Framework Reference](#) in Apple's Reference Library.
- The [TremoloUnit](#) sample project, which corresponds to the audio unit you build in "[Tutorial: Building a Simple Effect Unit with a Generic View](#)" (page 89).
- *Core Audio Overview*, which surveys all of the features available in Core Audio, and describes where audio units fit in.
- *Bundle Programming Guide*, which describes the file-system packaging mechanism in Mac OS X used for audio units.

- *Component Manager Reference*, which describes the API of the Component Manager, the Mac OS X technology that manages audio units at the system level. To learn more about the Component Manager you can refer to the Apple legacy document [Component Manager](#) from *More Macintosh Toolbox*, and to *Component Manager for QuickTime*.

Audio Unit Development Fundamentals

When you set out to create an audio unit, the power and flexibility of Core Audio’s **Audio Unit framework** give you the ability to go just about anywhere with sound. However, this power and flexibility also mean that there is a lot to learn to get started on the right foot. In this chapter, you get a bird’s-eye view of this leading edge technology, to serve you as you take the first steps toward becoming an audio unit developer.

You begin here with a quick look at the audio unit development cycle. Then, you focus in on what audio units are, and discover the important role of the **Core Audio SDK** in audio unit development. You learn how audio units function as plug-ins in Mac OS X and in concert with the applications that use them. Finally, you get introduced to the *Audio Unit Specification*, and how it defines the **plug-in API** that audio unit developers and application developers both write to.

After reading this chapter you’ll be ready to dig in to the architectural and development details presented in “[The Audio Unit](#)” (page 43).

If you want to get your hands on building an audio unit right away, you can skip this chapter for now and go straight to “[Tutorial: Building a Simple Effect Unit with a Generic View](#)” (page 89). As you build the audio unit, you can refer back to this chapter, and to other sections in this document, for conceptual information related to what you’re doing.

The Audio Unit Development Cycle

Audio unit development typically follows these steps:

1. Design the audio unit: specify the audio unit’s action, programmatic and user interface, and bundle configuration information.
2. Create and configure an appropriate Xcode project.
3. Implement the audio unit including parameters, factory presets, and properties—all described in the next chapter; implement copy protection, if desired; implement the synthesis, DSP, or data format conversion code.
4. Implement a graphical user interface—known as a custom view—if desired. Implement parameter automation support, if desired.
5. Validate and test the audio unit.

6. Deploy the audio unit bundle by packaging it in an installer or by providing installation instructions.

As with any software development, each of these steps typically entails iteration.

The tutorial later in this document, [“Tutorial: Building a Simple Effect Unit with a Generic View”](#) (page 89), leads you through most of these steps.

What Is An Audio Unit?

An **audio unit** (often abbreviated as *AU* in header files and elsewhere) is a Mac OS X plug-in that enhances digital audio applications such as Logic Pro and GarageBand. You can also use audio units to build audio features into your own application. Programmatically, an audio unit is packaged as a **bundle** and configured as a **component** as defined by the Mac OS X Component Manager.

At a deeper level, and depending on your viewpoint, an audio unit is one of two very different things.

From the inside—as seen by an audio unit developer—an audio unit is executable implementation code within a standard plug-in API. The API is standard so that any application designed to work with audio units will know how to use yours. The API is defined by the *Audio Unit Specification*.

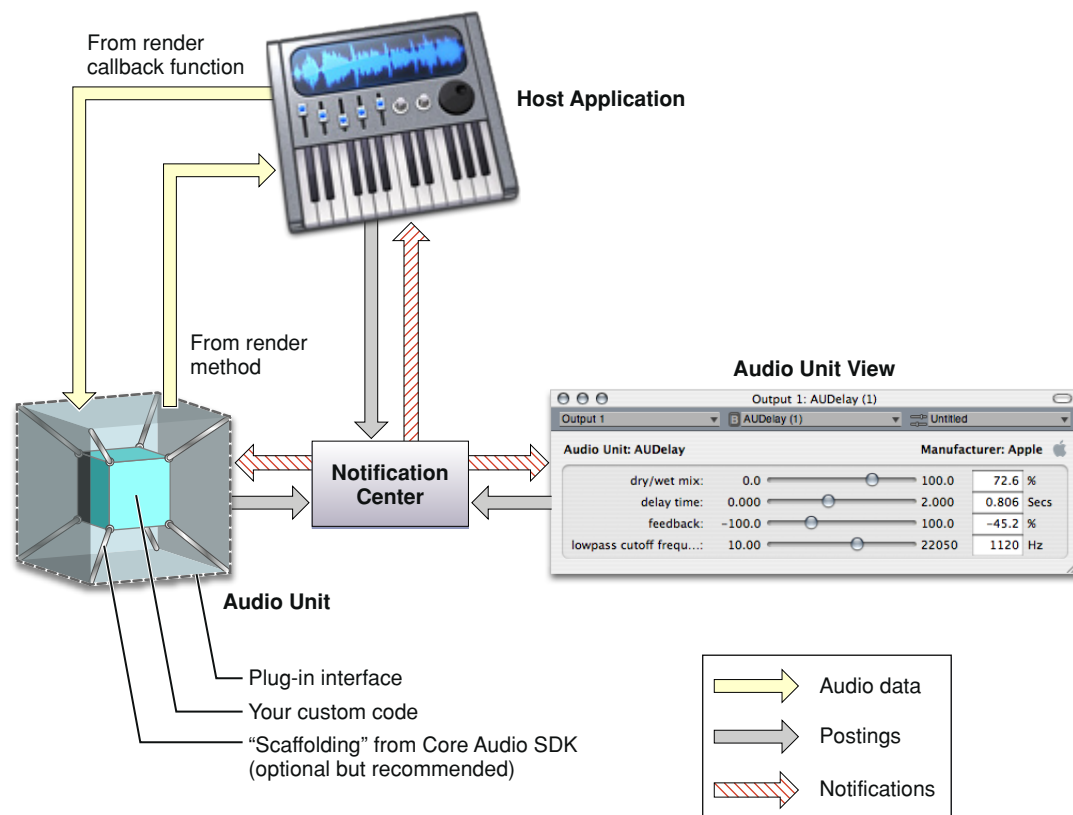
An audio unit developer can add the ability for users or applications to control an audio unit in real time through the audio unit parameter mechanism. Parameters are self-describing; their values and capabilities are visible to applications that use audio units.

From the outside—as seen from an application that uses the audio unit—an audio unit is just its plug-in API. This plug-in API lets applications query an audio unit about its particular features, defined by the audio unit developer as parameters and properties.

Because of this encapsulation, how you implement an audio unit is up to you. The quickest way, the one endorsed by Apple, and the one described in this document, is to subclass the appropriate C++ superclasses of the freely-downloadable Core Audio SDK.

Audio Unit Programmatic Structure and Life Cycle

The following figure represents a running audio unit built with the SDK. The figure shows the audio unit in context with its view and with an application—known as a **host**—that is using the audio unit:

Figure 1-1 A running audio unit, built using the Core Audio SDK

The figure shows two distinct internal parts of an audio unit bundle: the audio unit itself, on the left, and the audio unit view, on the right. The audio unit performs the audio work. The view provides a graphical user interface for the audio unit, and, if you provide it, support for parameter automation. (See [“Supporting Parameter Automation”](#) (page 37).) When you create an audio unit, you normally package both pieces in the same bundle—as you learn to do later—but they are logically separate pieces of code.

The audio unit, its view, and the host application communicate with each other by way of a notification center set up by the host application. This allows all three entities to remain synchronized. The functions for the notification center are part of the Core Audio **Audio Unit Event API**.

When a user first launches a host application, neither the audio unit nor its view is instantiated. In this state, none of the pieces shown in Figure 1-1 are present except for the host application.

The audio unit and its view come into existence, and into play, in one of two ways:

- Typically, a user indicates to a host application that they’d like to use an audio unit. For example, a user could ask the host to apply a reverb effect to a channel of audio.
- For an audio unit that you supply to add a feature to your own application, the application opens the audio unit directly, probably upon application launch.

When the host opens the audio unit, it hooks the audio unit up to the host's audio data chain—represented in the figure by the light yellow (audio data) arrows. This hook up has two parts: providing fresh audio data to the audio unit, and retrieving processed audio data from the audio unit.

- To provide fresh audio data to an audio unit, a host defines a callback function (to be called by the audio unit) that supplies audio data one slice at a time. A **slice** is a number of frames of audio data. A **frame** is one sample of audio data across all channels.
- To retrieve processed audio data from an audio unit, a host invokes an audio unit's render method.

Here is how the audio data flow proceeds between a host application and an audio unit:

1. The host invokes the audio unit's render method, effectively asking the audio unit for a slice of processed audio data
2. The audio unit responds by calling the host's callback function to get a slice of audio data samples to process
3. The audio unit processes the audio data samples and places the result in an output buffer for the host to retrieve
4. The host retrieves the processed data and then again invokes the audio unit's render method

In the depiction of the audio unit in [Figure 1-1](#) (page 15), the outer cube represents the plug-in API. Apple provides the *Audio Unit Specification* that defines the plug-in API for a variety of audio unit types. When you develop your audio unit to this specification, it will work with any host application that also follows the specification.

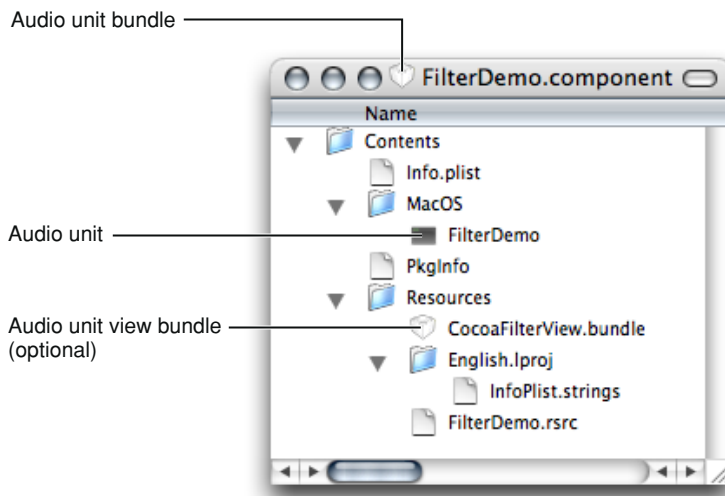
Inside, an audio unit contains programmatic scaffolding to connect the plug-in API to your custom code. When you use the Core Audio SDK to build your audio unit, this scaffolding is supplied in the form of glue code for the Component Manager along with a C++ class hierarchy. [Figure 1-1](#) (page 15) (rather figuratively) represents your custom code as an inner cube within the audio unit, and represents the SDK's classes and glue code as struts connecting the inner cube to the outer cube.

You can build an audio unit without using the Core Audio SDK, but doing so entails a great deal more work. Apple recommends that you use the Core Audio SDK for all but the most specialized audio unit development.

To learn about the internal architecture of an audio unit, read [“Audio Unit Architecture”](#) (page 43) in [“The Audio Unit”](#) (page 43).

Audio Unit File Structure

An audio unit looks like this within the Mac OS X file system:

Figure 1-2 An audio unit in the Mac OS X file system

When you build an audio unit using Xcode and a supplied audio unit template, your Xcode project takes care of packaging all these pieces appropriately.

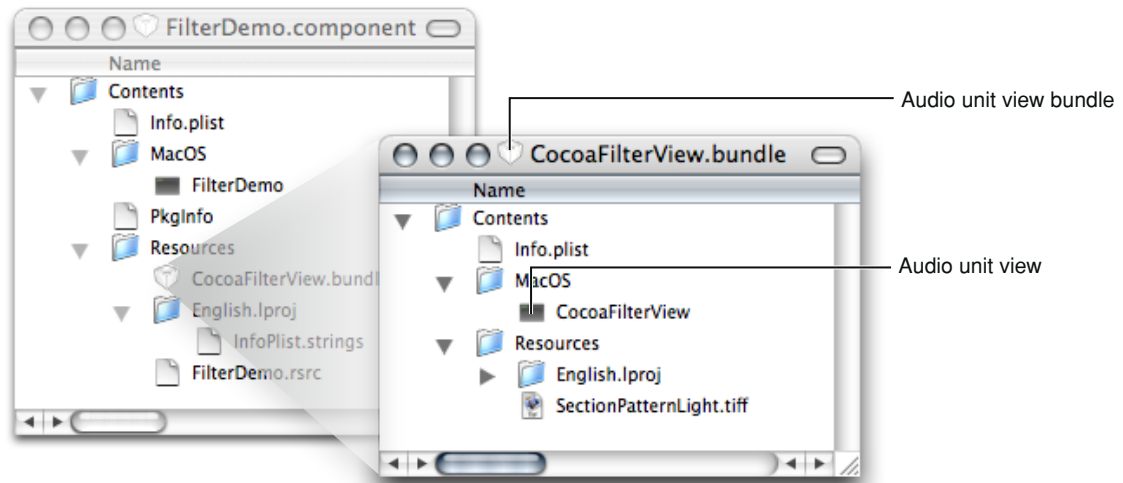
As a component, an audio unit has the following file system characteristics:

- It is a bundle with a `.component` file name extension
- It is a package; users see the bundle as opaque when they view it in the Finder

The information property list (`Info.plist`) file within the bundle's top-level `Contents` folder provides critical information to the system and to host applications that want to use the audio unit. For example, this file provides:

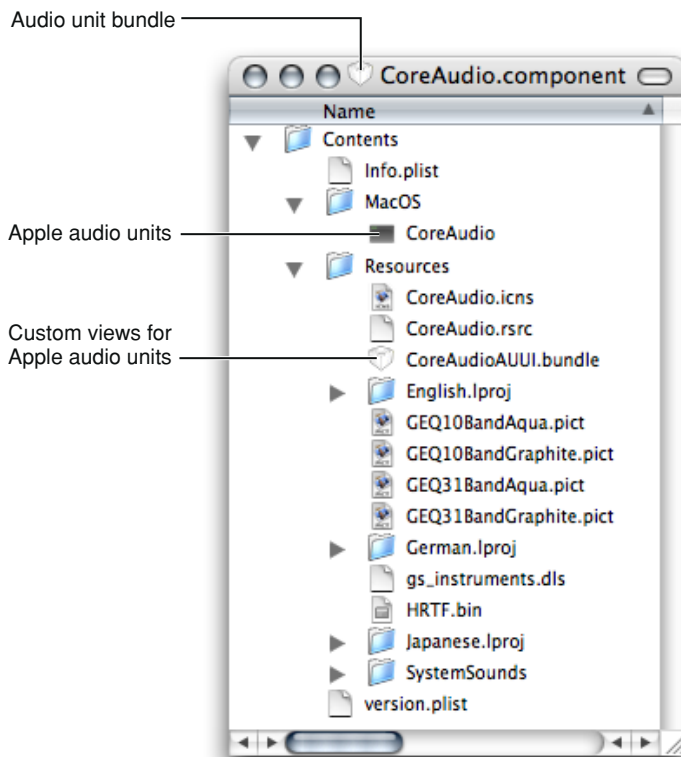
- The unique bundle identifier string in the form of a reverse domain name (or uniform type identifier). For example, for the `FilterDemo` audio unit provided in the Core Audio SDK, this identifier is `com.apple.demo.audiounit.FilterDemo`.
- The name of the file, within the bundle, that is the audio unit proper. This file is within the `MacOS` folder in the bundle.

An audio unit bundle can contain a custom user interface, called a view. The standard location for the view is in the audio unit bundle's `Resources` folder. The audio unit shown in Figure 1-2 includes such a view, packaged as an opaque bundle itself. Looking inside the audio unit view bundle shows the view bundle file structure:

Figure 1-3 An audio unit view in the Mac OS X file system

When a host application opens an audio unit, it can ask the audio unit if it has a custom view. If there is one, the audio unit can respond by providing the path to the view bundle. You can put the view bundle anywhere, including a network location. Typically, however, views are packaged as shown here.

An audio unit bundle typically contains one audio unit, as described in this section. But a single audio unit bundle can contain any number of audio units. For example, Apple packages all of its audio units in one bundle, `System/Library/Components/CoreAudio.component`. The `CoreAudio.component` bundle includes a single file of executable code containing all of the Apple audio units, and another file containing all of the supplied custom views:

Figure 1-4 The Apple audio units in the Mac OS X file system

Some Basic Terminology

To understand this document, it's important to understand the terms "audio unit," "audio unit view," and "audio unit bundle," as well as their relationships to each other.

- "Audio unit" usually refers to the executable code within the `MacOS` folder in the audio unit bundle, as shown in [Figure 1-2](#) (page 17). This is the part that performs the audio work. Sometimes, as in the title of this document, "audio unit" refers in context to the entire audio unit bundle and its contents. In this case, the term "audio unit" corresponds to a user's view of a plug-in in the Mac OS X file system.
- "Audio unit view" refers to the graphical user interface for an audio unit, as described in ["The Audio Unit View"](#) (page 63). As shown in [Figure 1-2](#), the code for a custom view typically lives in its own bundle in the `Resources` folder inside the audio unit bundle. Views are optional, because the `AudioUnit` framework lets a host application create a generic view based on parameter and property code in the audio unit.
- "Audio unit bundle" refers to the file system packaging that contains an audio unit and, optionally, a custom view. When this document uses "audio unit bundle," it is the characteristics of the packaging, such as the file name extension and the `Info.plist` file, that are important. Sometimes, as in the description of where to install audio units, "audio unit bundle" refers to the contents as well as the packaging. In this case, it's analogous to talking about a folder while meaning the folder and its contents.

Audio Units as Plug-Ins

In this section you learn about audio units from the outside in. First you take a look at using an audio unit in Apple's AU Lab host application. From there, you see how an audio unit plays a role as a component in Mac OS X.

The Nature of Plug-Ins

A plug-in is executable code with some special characteristics. As a library rather than a program, a plug-in cannot run by itself. Instead, a plug-in exists to provide features to host applications. For example, an audio unit could provide GarageBand with the ability to add tube-amplifier distortion to an audio signal.

Mac OS X provides two plug-in technologies: Core Foundation's CFPlugin architecture, and the Component Manager. Audio units are Component Manager-based plug-ins. Later in this section you learn about supporting the Component Manager in your audio units.

Host applications can ship with plug-ins, in which case the plug-in's use is transparent to a user. In other cases, a user can acquire a plug-in and explicitly add it to a running application.

What makes plug-ins special relative to other code libraries is their ability to contribute features dynamically to running host applications. You can see this in the AU Lab application, part of the Xcode Tools installation.

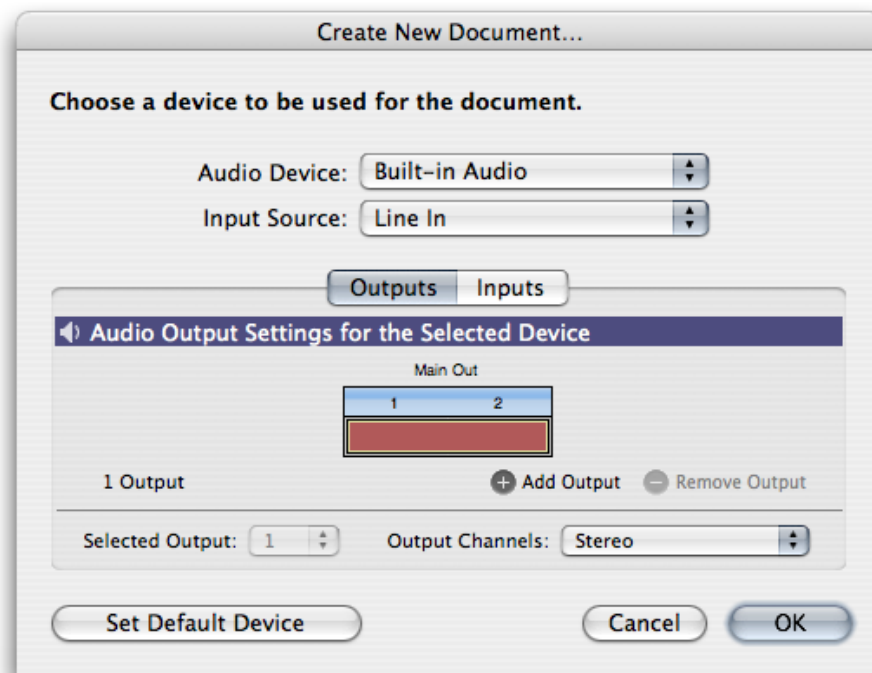
Tutorial: Using an Audio Unit in a Host Application

This mini-tutorial illustrates the dynamic nature of plug-ins by:

- Adding an audio unit to a running host application
- Using the audio unit
- Removing the audio unit from the running host application

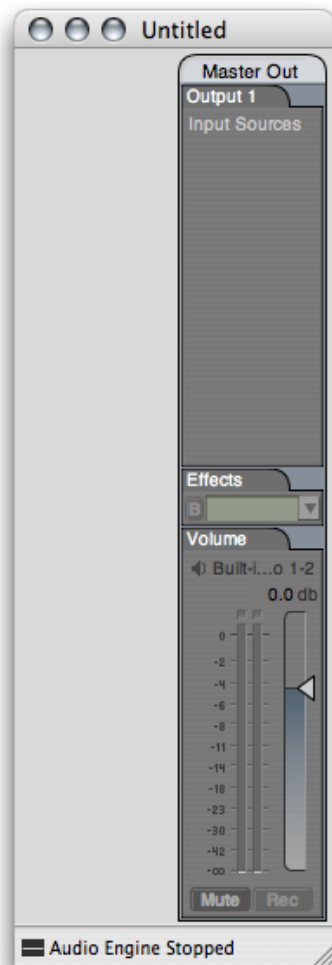
Along the way, this tutorial shows you how to get started with the very useful AU Lab application.

1. Launch the AU Lab audio unit host application (in `/Developer/Applications/Audio/`) and create a new AU Lab document. Unless you've configured AU Lab to use a default document style, the Create New Document window opens. If AU Lab was already running, choose `File > New` to get this window.



Ensure that the configuration matches the settings shown in the figure: Built-In Audio for the Audio Device, Line In for the Input Source, and Stereo for Output Channels. Leave the window's Inputs tab unconfigured; you will specify the input later. Click OK.

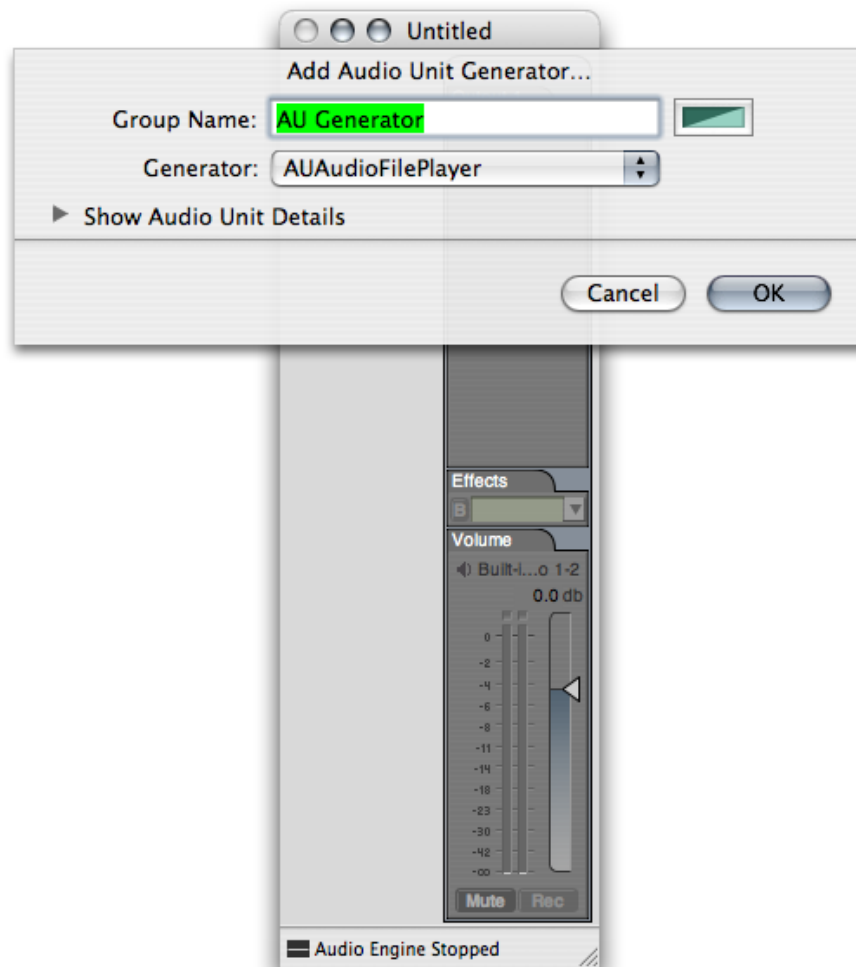
A new AU Lab window opens, showing the output channel you specified.



At this point, AU Lab has already instantiated all of the available audio units on your computer, queried them to find out such things as how each can be used in combination with other audio units, and has then closed them all again.

(More precisely, the Mac OS X Component Manager has invoked the instantiation and closing of the audio units on behalf of AU Lab. [“Component Manager Requirements for Audio Units”](#) (page 28), below, explains this.)

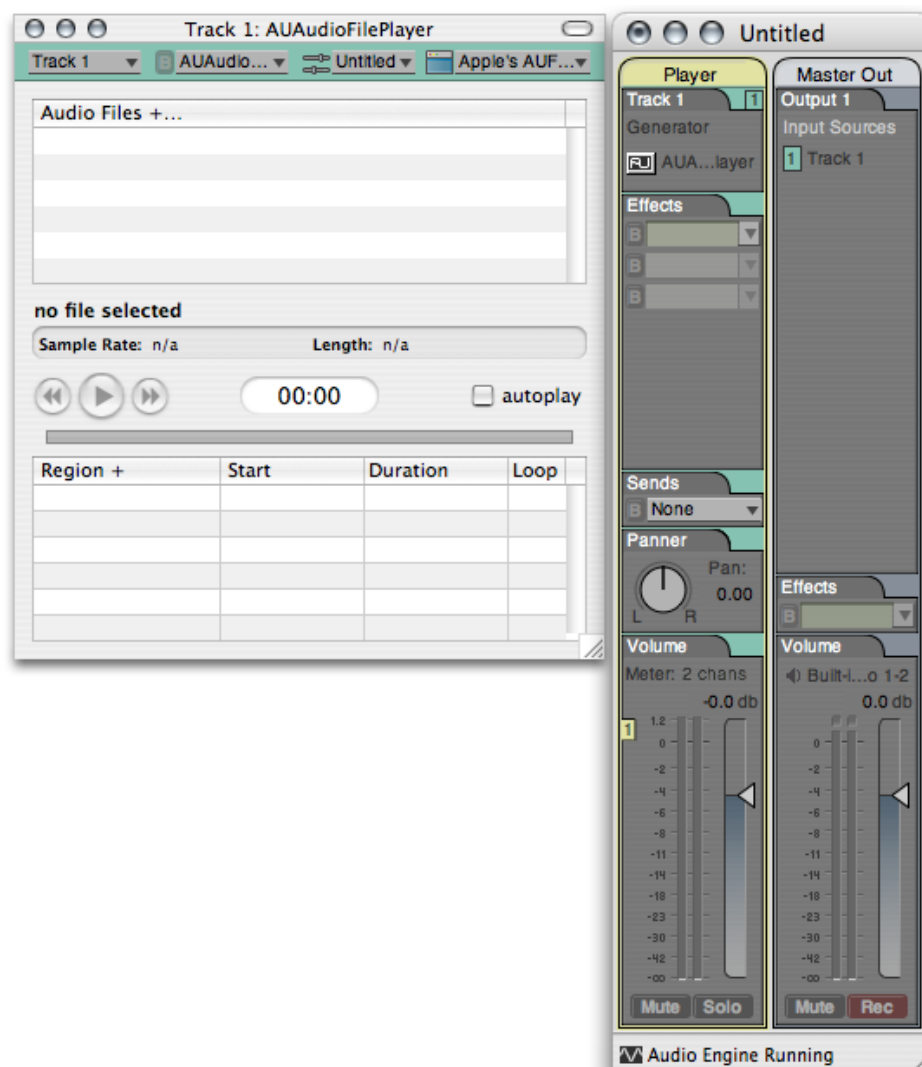
2. In AU Lab, choose Edit > Add Audio Unit Generator. A dialog opens from the AU Lab window to let you specify the generator unit to serve as the audio source.



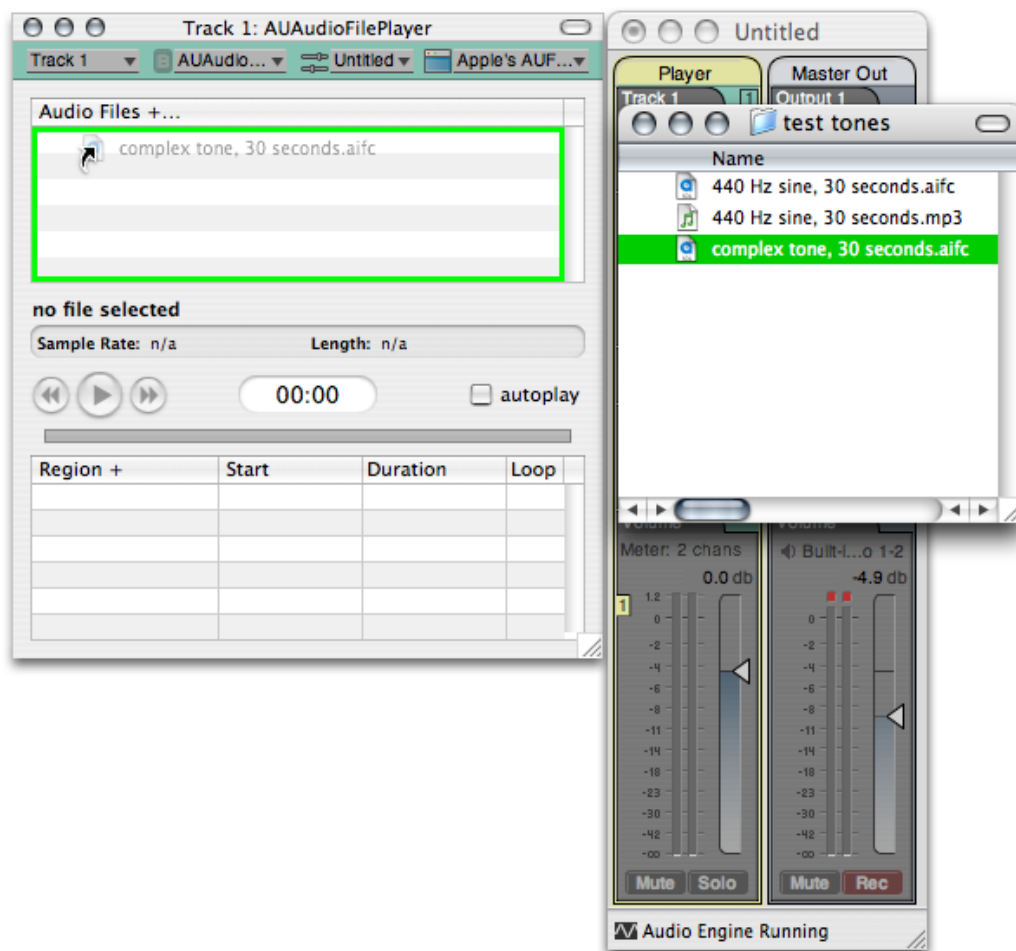
In the dialog, ensure that the AUAudioFilePlayer generator unit is selected in the Generator pop-up. To follow this example, change the Group Name to Player. Click OK.

You can change the group name at any time by double-clicking it in the AU Lab window.

The AU Lab window now shows a stereo input track. In addition, an inspector window has opened for the generator unit. If you close the inspector, you can reopen it by clicking the rectangular "AU" button near the top of the Player track.



3. Add one or more audio files to the Audio Files list in the player inspector window. Do this by dragging audio files from the Finder, as shown in the figure. Putting some audio files in the player inspector window lets you send audio through the AU Lab application, and through an audio unit that you add to the Player track. Just about any audio file will do. For this example, a music file works well.

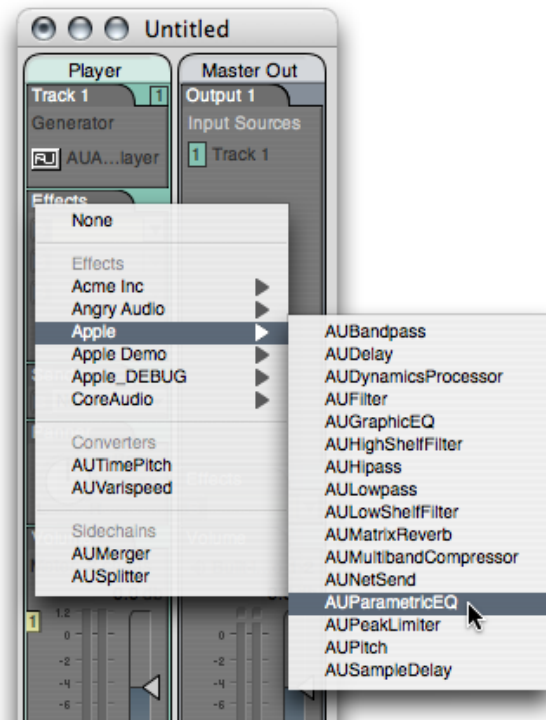


Now AU Lab is configured and ready for you to add an audio unit.

4. To dynamically add an audio unit to the AU Lab host application, click the triangular menu button in the first row of the Effects section in the Player track in AU Lab, as shown in the figure.

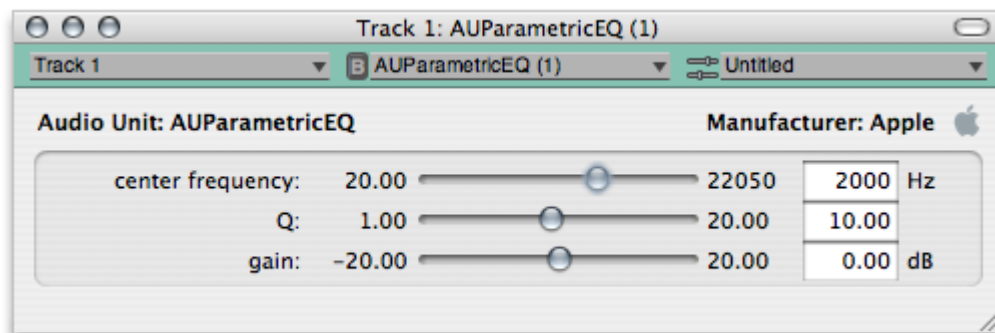


A menu opens, listing all the audio units available on your system, arranged by category and manufacturer. AU Lab gets this list from the Component Manager, which maintains a registry of installed audio units.



Choose an audio unit from the pop-up. To follow this example, choose the AUParametricEQ audio unit from the Apple submenu. (This audio unit, supplied as part of Mac OS X, is a single-band equalizer with controls for center frequency, gain, and Q.)

AU Lab asks the Component Manager to instantiate the audio unit you have chosen. AU Lab then initializes the audio unit. AU Lab also opens the audio unit's Cocoa generic view, which appears as a utility window:

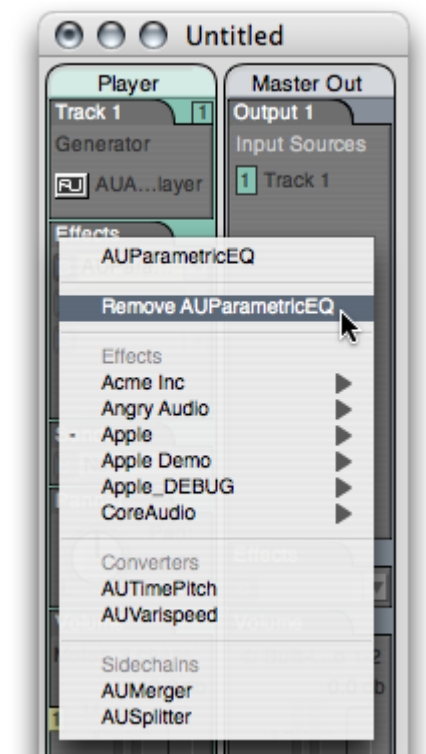


You have now dynamically added the AUParametricEQ audio unit to the running AU Lab host application.

5. To demonstrate the features of the audio unit in AU Lab, click the Play button in the AUAudioFilePlayer inspector to send audio through the audio unit. Vary the sliders in the generic view to hear the audio unit working.
6. To remove the audio unit from the host application, once again click the triangular menu button in the first row of the Effects section in the Player track, as shown in the figure.



From the pop-up menu, choose Remove AUParametricEQ.



The Component Manager closes the audio unit on behalf of AU Lab. You have now dynamically removed the audio unit, and its features, from the running AU Lab host application.

The Role of the Core Audio SDK

When you build an audio unit using the Core Audio SDK, you get Component Manager scaffolding for free. You also get comprehensive support for most of the *Audio Unit Specification*. This lets you concentrate on the more interesting aspects of audio unit development: the audio processing and the user interface.

You create an SDK-based audio unit by subclassing the appropriate classes in the SDK's audio unit C++ class hierarchy. "[Appendix: Audio Unit Class Hierarchy](#)" (page 139) shows this hierarchy.

Host applications communicate with audio units through their plug-in API and by way of the Component Manager. In all, there are six bodies of code that cooperate to support a running audio unit:

- The audio unit bundle. The bundle wraps the audio unit and its view (if you provide a custom view), and provides identification for the audio unit that lets Mac OS X and the Component Manager use the audio unit.
- The audio unit itself. When you build your audio unit with the Core Audio SDK, as recommended, the audio unit inherits from the SDK's class hierarchy.
- The audio unit view.
- The Core Audio API frameworks.
- The Component Manager.
- The host application.

Refer to "[A Quick Tour of the Core Audio SDK](#)" (page 83) if you'd like to learn about the rest of the SDK.

Component Manager Requirements for Audio Units

The Component Manager acts as a go-between for a host application and the audio units it uses—finding, opening, instantiating, and closing audio units on behalf of the host.

For Mac OS X to recognize your audio units, they must meet certain requirements. They must:

- Be packaged as a component, as defined by the Component Manager
- Have a single entry point that the Component Manager recognizes
- Have a resource (`.rsrc`) file that specifies a system wide unique identifier and version string
- Respond to Component Manager calls

Satisfying these requirements from scratch is a significant effort and requires a strong grasp of the Component Manager API. However, the Core Audio SDK insulates you from this. As demonstrated in the chapter "[Tutorial: Building a Simple Effect Unit with a Generic View](#)" (page 89), accommodating the Component Manager requires very little work when you use the SDK.

Audio Unit Installation and Registration

The Mac OS X Component Manager looks for audio units in some specific locations, one of which is reserved for use by Apple.

When you install your audio units during development or deployment, you typically put them in one of the following two locations:

- `~/Library/Audio/Plug-Ins/Components/`
Audio units installed here can be used only by the owner of the home folder
- `/Library/Audio/Plug-Ins/Components/`
Audio units installed here can be used by all users on the computer

It is up to you which of these locations you use or recommend to your users.

The Mac OS X preinstalled audio units go in a location reserved for Apple's use:

`/System/Library/Components/`

The Component Manager maintains a cached registry of the audio units in these locations (along with any other plug-ins it finds in other standard locations). Only registered audio units are available to host applications. The Component Manager refreshes the registry on system boot, on user log-in, and whenever the modification timestamp of one of the three `Components` folders changes.

A host application can explicitly register audio units installed in arbitrary locations by using the Component Manager's `RegisterComponent`, `RegisterComponentResource`, or `RegisterComponentResourceFile` functions. Audio units registered in this way are available only to the host application that invokes the registration. This lets you use audio units to add features to a host application you are developing, without making your audio units available to other hosts.

Audio Unit Identification

Every audio unit on a system must have a unique signature. The *Audio Unit Specification* takes advantage of this to let host applications know the plug-in API for any audio unit, based on its signature. This section describes how this works.

The Component Manager identifies audio units by a triplet of four-character codes:

- The “type” specifies the general type of functionality provided by an audio unit. In so doing, the type also identifies the audio unit's plug-in API. In this way, the type code is programmatically significant. For example, a host application knows that any audio unit of type `'aufx'` (for “audio unit effect”) provides DSP functionality.

The *Audio Unit Specification* specifies the available type codes for audio units, as well as the plug-in API for each audio unit type.

- The “subtype” describes more precisely what an audio unit does, but is not programmatically significant for audio units.

For example, Mac OS X includes an effect unit of subtype `'lpas'`, named to suggest that it provides low-pass filtering. If, for your audio unit, you use one of the subtypes listed in the `AUComponent.h` header file in the Audio Unit framework (such as `'lpas'`), you are suggesting to users of your

audio unit that it behaves like the named subtype. However, host applications make no assumptions about your audio unit based on its subtype. You are free to use any subtype code, including subtypes named with only lowercase letters.

- The “manufacturer code” identifies the developer of an audio unit.

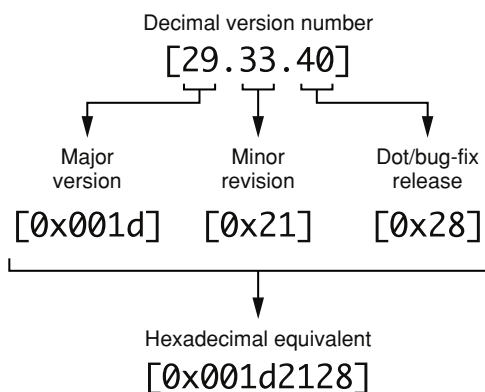
Apple expects each developer to register a manufacturer code, as a “creator code,” on the [Data Type Registration](#) page. Manufacturer codes must contain at least one uppercase character. Once registered, you can use the same manufacturer code for all your audio units.

In addition to these four-character codes, each audio unit must specify a correctly formatted version number. When the Component Manager registers audio units, it picks the most recent version if more than one is present on a system.

As a component, an audio unit identifies its version as an eight-digit hexadecimal number in its resource (.rsrc) file. As you’ll see in [“Tutorial: Building a Simple Effect Unit with a Generic View”](#) (page 89), you specify this information using Xcode.

Here is an example of how to construct a version number. It uses an artificially large number to illustrate the format unambiguously. For a decimal version number of 29.33.40, the hexadecimal equivalent is 0x001d2128. The format works as follows for this number:

Figure 1-5 Constructing an audio unit version number



The four most significant hexadecimal digits represent the major version number. The next two represent the minor version number. The two least significant digits represent the dot release number.

When you release a new version of an audio unit, you must ensure that its version number has a higher value than the previous version—not equal to the previous version, and not lower. Otherwise, users who have a previous version of your audio unit installed won’t be able to use the new version.

Plug-in API Requirements for Audio Units

Audio units are typed, as described above in [“Audio Unit Identification”](#) (page 29). When a host application sees an audio unit’s type, it knows how to communicate with it.

Implementing the plug-in API for any given type of audio unit from scratch is a significant effort. It requires a strong grasp of the Audio Unit and Audio Toolbox framework APIs and of the *Audio Unit Specification*. However, the Core Audio SDK insulates you from much of this as well. Using the SDK, you need to implement only those methods and properties that are relevant to your audio unit. (You learn about the audio unit property mechanism in the next chapter, “The Audio Unit” (page 43).)

The Audio Unit Specification

The *Audio Unit Specification* defines the common interface that audio unit developers and host application developers must support.

Note: The *Audio Unit Specification* document is currently in development. The following header files contain information relevant to the *Audio Unit Specification*: `AUComponent.h`, `AudioUnitProperties.h`, `MusicDevice.h`, and `OutputUnit.h`.

In addition, the following tutorial files contain information relevant to the Audio Unit Specification: `AUPannerUnits.text`, `OfflineRendering.rtf`, and `OfflineAPIAdditions.text`.

The *Audio Unit Specification* describes:

- The various Apple types defined for audio units, as listed in the “AudioUnit component types and subtypes” enumeration in the `AUComponent.h` header file in the Audio Unit framework
- The functional and behavioral requirements for each type of audio unit
- The plug-in API for each type of audio unit, including required and optional properties

You develop your audio units to conform to the *Audio Unit Specification*. You then test this conformance with the `auval` command-line tool, described in the next section.

The *Audio Unit Specification* defines the plug-in API for the following audio unit types:

- Effect units (`'auxf'`), such as volume controls, equalizers, and reverbs, which modify an audio data stream
- Music effect units (`'aumf'`), such as loopers, which combine features of instrument units (such as starting and stopping a sample) with features of effect units
- Offline effect units (`'auol'`), which let you do things with audio that aren’t practical in real time, such as time reversal or look-ahead level normalization
- Instrument units (`'aumu'`), which take MIDI and soundbank data as input and provide audio data as output—letting a user play a virtual instrument
- Generator units (`'augn'`), which programmatically generate an audio data stream or play audio from a file
- Data format converter units (`'aufc'`), which change characteristics of an audio data stream such as bit depth, sample rate, or playback speed
- Mixer units (`'aumx'`), which combine audio data streams
- Panner units (`'aupn'`), which distribute a set of input channels, using a spatialization algorithm, to a set of output channels

Audio Units as Instances of the Model-View-Controller Design Pattern

Apple’s Core Audio team designed the Audio Unit technology around one of the more popular software design patterns, the Model-View-Controller, or MVC. See [The Model-View-Controller Design Pattern](#) for more about this pattern.

Keep the MVC pattern in mind as you build your audio units:

- The audio unit serves as the model, encapsulating all of the knowledge to perform the audio work
- The audio unit’s view serves, naturally, as the view, displaying the audio unit’s current settings and allowing a user to change them
- The Audio Unit Event API, and the code in an audio unit and its view that calls this API, corresponds to the controller, supporting communication between the audio unit, its view, and a host application

Audio Units in Action

Opening and Closing Audio Units

Host applications are responsible—with the help of the Component Manager—for finding, opening, and closing audio units. Audio units, in turn, need to be findable, openable, and closable. Your audio unit gets these attributes when you build it from the Core Audio SDK and use the Xcode audio unit templates.

There is a two-step sequence for an audio unit becoming available for use in a host. These two steps are opening and initializing. Opening an audio unit amounts to instantiating an object of the audio unit’s main class. Initializing amounts to allocating resources so the audio unit is ready to do work.

To be a well-behaved, host-friendly plug-in, your audio unit’s instantiation must be fast and lightweight. Resource intensive startup work for an audio unit goes into the initialization step. For example, an instrument unit that employs a large bank of sample data should load it on initialization, not instantiation.

For more on finding, opening, and closing from your perspective as an audio unit developer, see [“Audio Unit Initialization and Uninitialization”](#) (page 59) and [“Closing”](#) (page 62) in [“The Audio Unit”](#) (page 43).

Adding Copy Protection

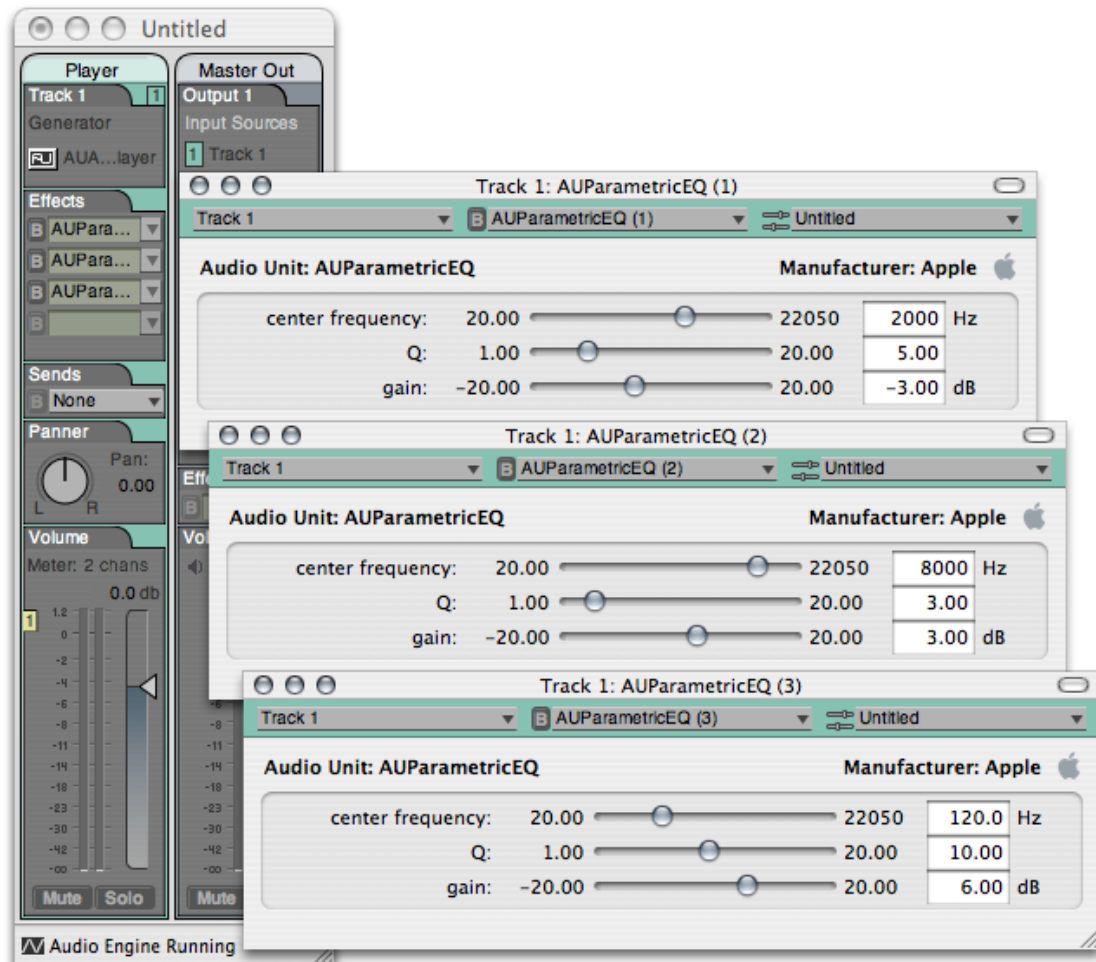
If you choose to add copy protection to your audio unit, it’s especially important to consider the audio unit’s opening sequence. The time for copy protection is during audio unit initialization—not instantiation. Therefore, you put copy protection code into an override of the `Initialize` method from the SDK’s `AUBase` superclass. You do not put copy protection code into an audio unit’s constructor.

Here is a scenario where this matters. Suppose a user doesn't have the required hardware dongle for your (copy protected) audio unit. Perhaps he left it at home when he brought his laptop to a performance. If your audio unit invokes its copy protection on instantiation, this could prevent a host application from opening. If your audio unit invokes its copy protection on initialization, as recommended, the performer could at least use the host application.

Multiple Instantiation

An audio unit can be instantiated any number of times by a host application and by any number of hosts. More precisely, the Component Manager invokes audio unit instantiation on behalf of host applications. The Component Manager infrastructure ensures that each audio unit instance exists and behaves independently.

You can demonstrate multiple instantiation in AU Lab. First add one instance of the AUParametricEQ effect unit to an AU Lab document, as described above in [“Tutorial: Using an Audio Unit in a Host Application”](#) (page 20). Then invoke the pop-up menus in additional rows of the Effects section in the Player track. You can add as many one-band parametric equalizers to the track as you like. Each of these instances of the audio unit behaves independently, as you can see by the varied settings in the figure:

Figure 1-6 Multiple instantiation of audio units in AU Lab

Audio Processing Graphs and the Pull Model

Host applications can connect audio units to each other so that the output of one audio unit feeds the input of the next. Such an interconnected series of audio units is called an **audio processing graph**. In the context of a graph, each connected audio unit is called a **node**.

When you worked through the [“Tutorial: Using an Audio Unit in a Host Application”](#) (page 20) section earlier in this chapter, the AU Lab application constructed an audio processing graph for you. This graph consisted of the AUAudioFilePlayer generator unit, the AUParametricEQ effect unit, and finally (not represented in the user interface of AU Lab) the Apple-supplied AUHAL I/O unit that interfaces with external hardware such as loudspeakers.

[“Audio Processing Graph Connections”](#) (page 47) provides details on how these connections work.

The Audio Processing Graph API, declared in the Audio Toolbox framework, provides interfaces for assisting host applications to create and manage audio processing graphs. When a host application employs this API, it uses an opaque data type called a **graph object** (of type `AUGraph`).

Some applications, such as AU Lab, always use graph objects when interconnecting audio units. Others, like Logic, connect audio units to each other directly. An individual audio unit, however, is not aware whether its connections are managed by a graph object on behalf of a host application, or by a host directly.

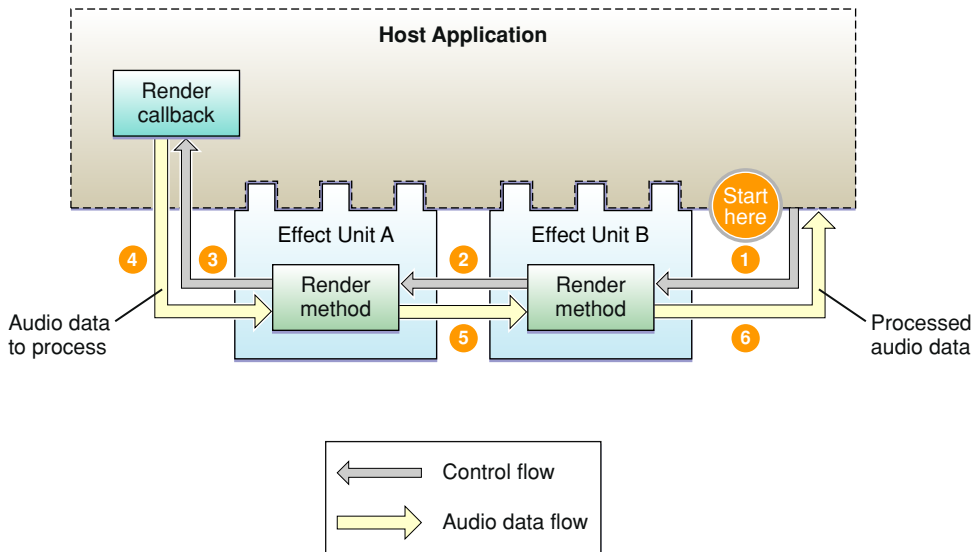
Audio data flow in graphs proceeds from the first (input) to last (output) node, as you'd expect. Control, however, flows from the last node back to the first. In Core Audio, this is called the **pull model**. The host application is in charge of invoking the pull.

You can think of the pull model in terms of a straw in a glass of water. The water in the glass represents fresh audio data waiting to be processed. The straw represents an audio processing graph, or even a single audio unit. Acting as a host application, you begin the flow of audio data by “pulling” (sipping) on the end of the straw. Specifically, a host application initiates the flow of audio data by calling the rendering method of the final node in a graph. Each sip that you take through the straw corresponds to another pull of a slice of audio data frames—another call to the final node's rendering method.

Before audio or control flow can start, the host application performs the work to hook up audio units to each other. In a case such as the one shown in the figure, the host also makes connections to and from the audio processing graph. But hosts do not necessarily feed audio data to graphs that they use. In a case where the first audio unit in a graph is a generator unit, there is no input connection; the generator provides audio data algorithmically or by playing a file.

Figure 1-7 shows the pull model in the particular case of a host application employing two effect units in sequence.

Figure 1-7 The pull model in action with two effect units



Here is how the pull proceeds in Figure 1-7:

1. The host application calls the render method of the final node (effect unit B) in the graph, asking for one slice worth of processed audio data frames.
2. The render method of effect unit B looks in its input buffers for audio data to process, to satisfy the call to render. If there is audio data waiting to be processed, effect unit B uses it. Otherwise, and as shown in the figure, effect unit B (employing a superclass in the SDK's audio unit class

hierarchy) calls the render method of whatever the host has connected to effect unit B's inputs. In this example, effect unit A is connected to B's inputs—so effect unit B pulls on effect unit A, asking for a slice audio data frames.

3. Effect unit A behaves just as effect unit B does. When it needs audio data, it gets it from its input connection, which was also established by the host. The host connected effect unit A's inputs to a render callback in the host. Effect unit A pulls on the host's render callback.
4. The host's render callback supplies the requested audio data frames to effect unit A.
5. Effect unit A processes the slice of data supplied by the host. Effect unit A then supplies the processed audio data frames that were previously requested (in step 2) to effect unit B.
6. Effect unit B processes the slice of data provided by effect unit A. Effect unit B then supplies the processed audio data frames that were originally requested (in step 1) to the host application. This completes one cycle of pull.

Audio units normally do not know whether their inputs and outputs are connected to other audio units, or to host applications, or to something else. Audio units simply respond to rendering calls. Hosts are in charge of establishing connections, and subclasses (for audio units built with the Core Audio SDK) take care of implementing the pull.

As an audio unit developer, you don't need to work directly with audio processing graphs except to ensure that your audio unit plays well with them. You do this, in part, by ensuring that your audio unit passes Apple's validation test, described in [“Audio Unit Validation with the auval Tool”](#) (page 38). You should also perform testing by hooking up your audio unit in various processing graphs using host applications, as described in [“Audio Unit Testing and Host Applications”](#) (page 39).

Processing: The Heart of the Matter

Audio units process audio data, of course. They also need to know how to stop processing gracefully, and how to modify their processing based on user adjustments. This section briefly discusses these things. [“The Audio Unit”](#) (page 43) describes processing in greater detail.

Processing

An audio unit that processes audio data, such as an effect unit, works in terms of rendering cycles. In each rendering cycle, the audio unit:

- Gets a slice of fresh audio data frames to process. It does this by calling the rendering callback function that has been registered in the audio unit.
- Processes the audio data frames.
- Puts the resulting audio data frames into the audio unit's output buffers.

An audio unit does this work at the beck and call of its host application. The host application also sets number of audio data frames per slice. For example, AU Lab uses 512 frames per slice as a default, and you can vary this number from 24 to 4,096. See [“Testing with AU Lab”](#) (page 39).

The programmatic call to render the next slice of frames can arrive from either of two places:

- From the host application itself, in the case of the host using the audio unit directly
- From the downstream neighbor of the audio unit, in the case of the audio unit being part of an audio processing graph

Audio units behave exactly the same way regardless of the calling context—that is, regardless of whether it is a host application or a downstream audio unit asking for audio data.

Resetting

Audio units also need to be able to gracefully stop rendering. For example, an audio unit that implements an IIR filter uses an internal buffer of samples. It uses the values of these buffered samples when applying a frequency curve to the samples it is processing. Say that a user of such an audio unit stops playing an audio file and then starts again at a different point in the file. The audio unit, in this case, must start with an empty processing buffer to avoid inducing artifacts.

When you develop an audio unit's DSP code, you implement a `Reset` method to return the DSP state of the audio unit to what it was when the audio unit was first initialized. Host applications call the `Reset` method as needed.

Adjustments While Rendering

While an audio unit is rendering, a user can adjust the rendering behavior using the audio unit's view. For example, in a parametric filter audio unit, a user can adjust the center frequency. It's also possible for host applications to alter rendering using parameter automation, described in the next section.

Supporting Parameter Automation

Parameters let users adjust audio units. For instance, Apple's low-pass filter audio unit has parameters for cut-off frequency and resonance.

Parameter automation lets users program parameter adjustments along a time line. For example, a user might want to use a low pass filter audio unit to provide an effect like a guitar wah-wah pedal. With parameter automation, the user could record the wah-wah effect and make it part of a musical composition.

Parameter automation relies on three things:

- The ability of an audio unit to change its parameter values programmatically on request from a host application
- The ability of an audio unit to post notifications as its parameter values are changed by a user
- The ability of a host application to support recording and playback of parameter automation data

Some hosts that support parameter automation with audio units are Logic Pro, Ableton Live, and Sagan Metro.

Parameter automation relies on the Audio Unit Event API, declared in the `AudioUnitUtilities.h` header file as part of the Audio Toolbox Framework. This thread-safe API provides a notification mechanism that supports keeping audio units, their views, and host applications in sync.

To support parameter automation in your audio unit, you must create a custom view. You add the automation support to the view's executable code, making use of the Audio Unit Event API to support some or all of the following event types:

- **Parameter gestures**, which include the `kAudioUnitEvent_BeginParameterChangeGesture` and `kAudioUnitEvent_EndParameterChangeGesture` event types
- **Parameter value changes**, identified by the `kAudioUnitEvent_ParameterValueChange` event type
- **Property changes**, identified by the `kAudioUnitEvent_PropertyChange` event type

[“The Audio Unit View”](#) (page 63) gives more information on parameter automation.

In some unusual cases you may need to add support for parameter automation to the audio unit itself. For example, you may create a bandpass filter with adjustable upper and lower corner frequencies. Your audio unit then needs to ensure that the upper frequency is never set below the lower frequency.

Audio Unit Validation and Testing

Audio Unit Validation with the `auval` Tool

Apple strongly recommends validating your audio units using the `auval` command-line tool during development. The `auval` tool (named as a contraction of “audio unit validation”) comes with Mac OS X. It performs a comprehensive suite of tests on:

- An audio unit's plug-in API, as defined by its programmatic type
- An audio unit's basic functionality including such things as which audio data channel configurations are available, time required to instantiate the audio unit, and the ability of the audio unit to render audio

The `auval` tool tests only an audio unit proper. It does not test any of the following:

- Audio unit views
- Audio unit architecture, in terms of using the recommended model-view-controller design pattern for separation of concerns
- Correct use of the Audio Unit Event API
- Quality of DSP, quality of audio generation, or quality of audio data format conversion

The `auval` tool can validate every type of audio unit defined by Apple. When you run it, it outputs a test log and summarizes the results with a “pass” or “fail” indication.

For more information, refer to the `AUValidationReadMe.rtf` file described in [“A Quick Tour of the Core Audio SDK”](#) (page 83), and refer to the `auval` built-in help system. To see `auval` help text, enter the following command at a prompt in the Terminal application:

```
auval -h
```

Audio Unit Testing and Host Applications

When you build to the *Audio Unit Specification*, you’ve done the right thing. Such an audio unit should work with all hosts. But practically speaking, development isn’t complete until you test your audio units in commercial applications. The reasons include:

- Evolution of the Core Audio frameworks and SDK
- Variations across host application versions
- Idiosyncrasies in the implementation of some host applications

As host applications that recognize audio units proliferate, the task of testing your audio unit in all potential hosts becomes more involved.

The situation is somewhat analogous to testing a website in various browsers: your code may perfectly fit the relevant specifications, but nonconformance in one or another browser requires you to compensate.

With this in mind, the following sections provide an overview of host-based audio unit testing.

Testing with AU Lab

AU Lab, the application you used in “Tutorial: Using an Audio Unit in a Host Application”, is the reference audio unit host. It is in active development by Apple’s Core Audio team. They keep it in sync with the `auval` tool, with the Core Audio frameworks and SDK, and with Mac OS X itself. This makes AU Lab the first place to test your audio units.

What You Can Test with AU Lab

Testing your audio unit with AU Lab lets you test:

- Behavior, in terms of being found by a host, displayed in a menu, and opened
- View, both generic and custom
- Audible performance
- Interaction with other audio units when placed in an audio processing graph
- I/O capabilities, such as sidechains and multiple outputs, as well as basic testing of monaural and stereophonic operation

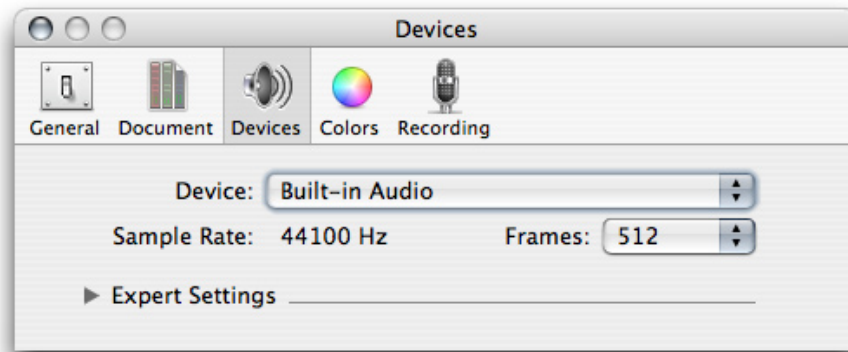
In Mac OS X v10.4 “Tiger,” AU Lab lets you test the following types of audio units:

- Converter units
- Effect units
- Generator units
- Instrument units

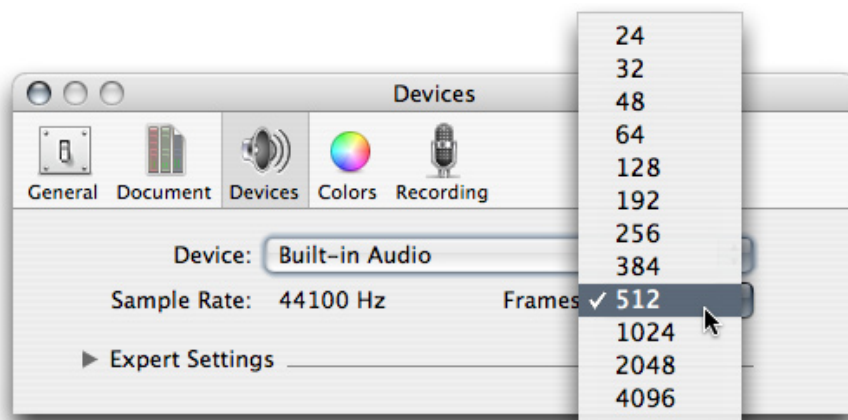
Varying the Host Application's Characteristics

AU Lab lets you control some of its hosting characteristics, which lets you test the behavior of your audio unit under varying conditions. For example, you can change the number of frames of audio data to process in each rendering cycle. You do this using Devices Preferences.

In AU Lab, choose Preferences from the AU Lab menu. Click Devices to show Devices Preferences:



Click the Frames pop-up menu. You can choose the number of frames for your audio unit to process in each rendering cycle:



Click the disclosure triangle for Expert Settings. You can vary the slider to choose the percentage of CPU time to devote to audio processing. This lets you test the behavior of your audio unit under varying load conditions:



Custom Testing of Audio Units

As an audio unit developer, you'll want to stay up to date with the host applications your target market is using. Apple recommends that you test your audio units with, at least, Apple's suite of professional host applications:

- GarageBand
- Logic Pro
- Soundtrack Pro
- Final Cut Pro

There are many third-party and open source applications that support audio units, among them Ableton Live, Amadeus, Audacity, Cubase, Digital Performer, DSP-Quattro, Peak, Rax, and Metro.

The Audio Unit

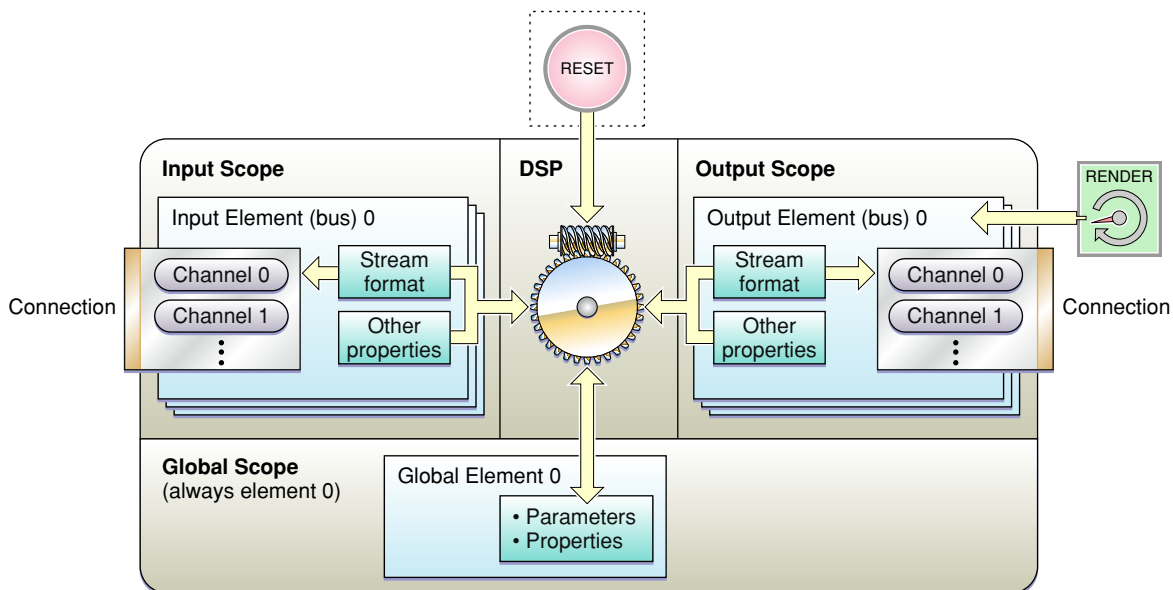
When you develop an audio unit, you begin with the part that performs the audio work. This part exists within the `MacOS` folder inside the audio unit bundle as shown in [Figure 1-2](#) (page 17). You can optionally add a custom user interface, or view, as described in the next chapter, “[The Audio Unit View](#)” (page 63).

In this chapter you learn about the architecture and programmatic elements of an audio unit. You also learn about the steps you take when you create an audio unit.

Audio Unit Architecture

The internal architecture of an audio unit consists of scopes, elements, connections, and channels, all of which serve the audio processing code. Figure 2-1 illustrates these parts as they exist in a typical effect unit. This section describes each of these parts in turn. For discussion on the section marked *DSP* in the figure, representing the audio processing code in an effect unit, see “[Synthesis, Processing, and Data Format Conversion Code](#)” (page 55).

Figure 2-1 Audio unit architecture for an effect unit



Audio Unit Scopes

An audio unit **scope** is a programmatic context. Unlike the general computer science notion of scopes, however, audio unit scopes cannot be nested. Each scope is a discrete context.

You use scopes when writing code that sets or retrieves values of parameters and properties. For example, here is an implementation of a standard `GetProperty` method, as used in the effect unit you build in [“Tutorial: Building a Simple Effect Unit with a Generic View”](#) (page 89):

Listing 2-1 Using “scope” in the `GetProperty` method

```
ComponentResult TremoloUnit::GetProperty (
    AudioUnitPropertyID    inID,
    AudioUnitScope         inScope,    // the host specifies the scope
    AudioUnitElement       inElement,
    void                   *outData
) {
    return AUEffectBase::GetProperty (inID, inScope, inElement, outData);
}
```

When a host application calls this method to retrieve the value of a property, the host specifies the scope in which the property is defined. The implementation of the `GetProperty` method, in turn, can respond to various scopes with code such as this:

```
if (inScope == kAudioUnitScope_Global) {
    // respond to requests targeting the global scope
} else if (inScope == kAudioUnitScope_Input) {
    // respond to requests targeting the input scope
} else {
    // respond to other requests
}
```

There are five scopes defined by Apple in the `AudioUnitProperties.h` header file in the Audio Unit framework:

Listing 2-2 Audio unit scopes

```
enum {
    kAudioUnitScope_Global    = 0,
    kAudioUnitScope_Input    = 1,
    kAudioUnitScope_Output    = 2,
    kAudioUnitScope_Group     = 3,
    kAudioUnitScope_Part      = 4
};
```

The three most important scopes are:

- **Input scope:** The context for audio data coming into an audio unit. Code in an audio unit, a host application, or an audio unit view can address an audio unit’s input scope for such things as the following:
 - ❑ An audio unit defining additional input elements
 - ❑ An audio unit or a host setting an input audio data stream format
 - ❑ An audio unit view setting the various input levels on a mixer audio unit
 - ❑ A host application connecting audio units into an audio processing graph

Host applications also use the input scope when registering a render callback, as described in [“Render Callback Connections”](#) (page 47).

- **Output scope:** The context for audio data leaving an audio unit. The output scope is used for most of the same things as input scope: connections, defining additional output elements, setting an output audio data stream format, and setting output levels in the case of a mixer unit with multiple outputs.

A host application, or a downstream audio unit in an audio processing graph, also addresses the output scope when invoking rendering.

- **Global scope:** The context for audio unit characteristics that apply to the audio unit as a whole. Code within an audio unit addresses its own global scope for setting or getting the values of such properties as:
 - latency,
 - tail time, and
 - supported number(s) of channels.

Host applications can also query the global scope of an audio unit to get these values.

There are two additional audio unit scopes, intended for instrument units, defined in `AudioUnitProperties.h`:

- **Group scope:** A context specific to the rendering of musical notes in instrument units
- **Part scope:** A context specific to managing the various voices of multitimbral instrument units

This version of *Audio Unit Programming Guide* does not discuss group scope or part scope.

Audio Unit Elements

An audio unit **element** is a programmatic context that is nested within a scope. Most commonly, elements come into play in the input and output scopes. Here, they serve as programmatic analogs of the signal buses used in hardware audio devices. Because of this analogy, audio unit developers often refer to elements in the input or output scopes as *buses*; this document follows suit.

As you may have noticed in [Listing 2-1](#) (page 44), hosts specify the element as well as the scope they are targeting when getting or setting properties or parameters. Here is that method again, with the `inElement` parameter highlighted:

Listing 2-3 Using “element” in the GetProperty method

```
ComponentResult TremoloUnit::GetProperty (
    AudioUnitPropertyID    inID,
    AudioUnitScope         inScope,
    AudioUnitElement       inElement, // the host specifies the element here
    void                   *outData
) {
    return AUEffectBase::GetProperty (inID, inScope, inElement, outData);
}
```

Elements are identified by integer numbers and are zero indexed. In the input and output scopes, element numbering must be contiguous. In the typical case, the input and output scopes each have one element, namely element (or bus) 0.

The global scope in an audio unit is unusual in that it always has exactly one element. Therefore, the global scope's single element is always element 0.

A bus (that is, an input or output element) always has exactly one stream format. The stream format specifies a variety of characteristics for the bus, including sample rate and number of channels. Stream format is described by the **audio stream description structure** (`AudioStreamBasicDescription`), declared in the `CoreAudioTypes.h` header file and shown here:

Listing 2-4 The audio stream description structure

```
struct AudioStreamBasicDescription {
    Float64 mSampleRate;           // sample frames per second
    UInt32  mFormatID;             // a four-char code indicating stream type
    UInt32  mFormatFlags;          // flags specific to the stream type
    UInt32  mBytesPerPacket;        // bytes per packet of audio data
    UInt32  mFramesPerPacket;       // frames per packet of audio data
    UInt32  mBytesPerFrame;         // bytes per frame of audio data
    UInt32  mChannelsPerFrame;      // number of channels per frame
    UInt32  mBitsPerChannel;        // bit depth
    UInt32  mReserved;              // padding
};
typedef struct AudioStreamBasicDescription AudioStreamBasicDescription;
```

An audio unit can let a host application get and set the stream formats of its buses using the `kAudioUnitProperty_StreamFormat` property, declared in the `AudioUnitProperties.h` header file. This property's value is an audio stream description structure.

Typically, you will need just a single input bus and a single output bus in an audio unit. When you create an effect unit by subclassing the `AUEffectBase` class, you get one input and one output bus by default. Your audio unit can specify additional buses by overriding the main class's constructor. You would then indicate additional buses using the `kAudioUnitProperty_BusCount` property, or its synonym `kAudioUnitProperty_ElementCount`, both declared in the `AudioUnitProperties.h` header file.

You might find additional buses helpful if you are building an interleaver or deinterleaver audio unit, or an audio unit that contains a primary audio data path as well as a sidechain path for modulation data.

A bus can have exactly one connection, as described next.

Audio Unit Connections

A **connection** is a hand-off point for audio data entering or leaving an audio unit. Fresh audio data samples move through a connection and into an audio unit when the audio unit calls a render callback. Processed audio data samples leave an audio unit when the audio unit's render method gets called. The Core Audio SDK's class hierarchy implements audio data hand-off, working with an audio unit's rendering code.

Hosts establish connections at the granularity of a bus, and not of individual channels. You can see this in [Figure 2-1](#) (page 43). The number of channels in a connection is defined by the stream format, which is set for the bus that contains the connection.

Audio Processing Graph Connections

To connect one audio unit to another, a host application sets a property in the destination audio unit. Specifically, it sets the `kAudioUnitProperty_MakeConnection` property in the input scope of the destination audio unit. When you build your audio units using the Core Audio SDK, this property is implemented for you.

In setting a value for this property, the host specifies the source and destination bus numbers using an **audio unit connection structure** (`AudioUnitConnection`), shown here:

Listing 2-5 The audio unit connection structure

```
typedef struct AudioUnitConnection {
    AudioUnit sourceAudioUnit;    // the audio unit that supplies audio
                                // data to the audio unit whose
                                // connection property is being set
    UInt32 sourceOutputNumber;    // the output bus of the source unit
    UInt32 destInputNumber;      // the input bus of the destination unit
} AudioUnitConnection;
```

The `kAudioUnitProperty_MakeConnection` property and the audio unit connection structure are declared in the `AudioUnitProperties.h` file in the Audio Unit framework.

As an audio unit developer, you must make sure that your audio unit can be connected for it to be valid. You do this by supporting appropriate stream formats. When you create an audio unit by subclassing the classes in the SDK, your audio unit will be connectible. The default, required stream format for audio units is described in [“Commonly Used Properties”](#) (page 52).

[Figure 1-7](#) (page 35) illustrates that the entity upstream from an audio unit can be either another audio unit or a host application. Whichever it is, the upstream entity is typically responsible for setting an audio unit’s input stream format before a connection is established. If an audio unit cannot support the stream format being requested, it returns an error and the connection fails.

Render Callback Connections

A host application can send audio data to an audio unit directly and can retrieve processed data from the audio unit directly. You don’t need to make any changes to your audio unit to support this sort of connection.

To prepare to send data to an audio unit, a host defines a **render callback** (shown in [Figure 1-7](#) (page 35)) and registers it with the audio unit. The signature for the callback is declared in the `AUComponent.h` header file in the Audio Unit framework:

Listing 2-6 The render callback

```
typedef OSStatus (*AURenderCallback)(
    void *inRefCon,
    AudioUnitRenderActionFlags *ioActionFlags,
    const AudioTimeStamp *inTimeStamp,
    UInt32 inBusNumber,
```

The Audio Unit

```

        UInt32                inNumberFrames,
        AudioBufferList       *ioData
    );

```

The host must explicitly set the stream format for the audio unit's input as a prerequisite to making the connection. The audio unit calls the callback in the host when it's ready for more audio data.

In contrast, for an audio processing graph connection, the upstream audio unit supplies the render callback. In a graph, the upstream audio unit also sets the downstream audio unit's input stream format.

A host can retrieve processed audio data from an audio unit directly by calling the `AudioUnitRender` function on the audio unit:

Listing 2-7 The `AudioUnitRender` function

```

extern ComponentResult AudioUnitRender (
    AudioUnit             ci,
    AudioUnitRenderActionFlags *ioActionFlags,
    const AudioTimeStamp  *inTimeStamp,
    UInt32                inOutputBusNumber,
    UInt32                inNumberFrames,
    AudioBufferList       *ioData
);

```

The Core Audio SDK passes this function call into your audio unit as a call to the audio unit's `Render` method.

You can see the similarity between the render callback and `AudioUnitRender` signatures, which reflects their coordinated use in audio processing graph connections. Like the render callback, the `AudioUnitRender` function is declared in the `AUComponent.h` header file in the Audio Unit framework.

For more information on how a host application connects directly to an audio unit, see the [Audio Unit Rendering document \(aurender.html\)](#) in the Core Audio SDK Documentation/AudioUnits/Topics folder.

Audio Unit Channels

An audio unit **channel** is, conceptually, a monaural, noninterleaved path for audio data samples that goes to or from an audio unit's processing code. The Core Audio SDK represents channels as buffers. Each buffer is described by an audio buffer structure (`AudioBuffer`), as declared in the `CoreAudioTypes.h` header file in the Core Audio framework:

Listing 2-8 The audio buffer structure

```

struct AudioBuffer {
    UInt32  mNumberChannels; // number of interleaved channels in the buffer
    UInt32  mDataByteSize;   // size, in bytes, of the buffer
    void    *mData;          // pointer to the buffer
};
typedef struct AudioBuffer AudioBuffer;

```


An audio buffer can hold a single channel, or multiple interleaved channels. However, most types of audio units, including effect units, use only noninterleaved data. These audio units expect the `mNumberChannels` field in the audio buffer structure to equal 1.

Output units and format converter units can accept interleaved channels, represented by an audio buffer with the `mNumberChannels` field set to 2 or greater.

An audio unit manages the set of channels in a bus as an audio buffer list structure (`AudioBufferList`), also defined in `CoreAudioTypes.h`.

Listing 2-9 The audio buffer list structure

```
struct AudioBufferList {
    UInt32      mNumberBuffers; // the number of buffers in the list
    AudioBuffer mBuffers[kVariableLengthArray]; // the list of buffers
};
typedef struct AudioBufferList AudioBufferList;
```

In the common case of building an n -to- n channel effect unit, such as the one you build in “[Tutorial: Building a Simple Effect Unit with a Generic View](#)” (page 89), the audio unit template and superclasses take care of managing channels for you. You create this type of effect unit by subclassing the `AUEffectBase` class in the SDK.

In contrast, when you build an m -to- n channel effect unit (for example, stereo-to-mono effect unit), you must write code to manage channels. In this case, you create your effect unit by subclassing the `AUBase` class. (As with the rest of this document, this consideration applies to version 1.4.3 of the Core Audio SDK, current at the time of publication.)

Creating an Audio Unit by Subclassing

The simplest, and recommended, way to create an audio unit is by subclassing the Core Audio SDK’s C++ superclasses. With minimal effort, this gives you all of the programmatic scaffolding and hooks your audio unit needs to interact with other parts of Core Audio, the Component Manager, and audio unit host applications.

For example, when you build an n -channel to n -channel effect unit using the SDK, you define your audio unit’s main class as a subclass of the `AUEffectBase` superclass. When you build a basic instrument unit (also known as a software-based music synthesizer), you define your audio unit’s main class as a subclass of the SDK `AUInstrumentBase` superclass. “[Appendix: Audio Unit Class Hierarchy](#)” (page 139) describes these classes along with the others in the SDK.

In practice, subclassing usually amounts to one of two things:

- Creating an audio unit Xcode project with a supplied template. In this case, creating the project gives you source files that define custom subclasses of the appropriate superclasses. You modify and extend these files to define the custom features and behavior of your audio unit.
- Making a copy of an audio unit project from the SDK, which already contains custom subclasses. In this case, you may need to strip out code that isn’t relevant to your audio unit, as well as change symbols in the project to properly identify and refer to your audio unit. You then work in the same way you would had you started with an Xcode template.

Control Code: Parameters, Factory Presets, and Properties

Most audio units are user adjustable in real time. For example, a reverb unit might have user settings for initial delay, reverberation density, decay time, and dry/wet mix. Such adjustable settings are called **parameters**. Parameters have floating point or integer values. Floating point parameters typically use a slider interface in the audio unit's view. You can associate names with integer values to provide a menu interface for a parameter, such as to let the user pick tremolo type in a tremolo effect unit. Built-in (developer defined) combinations of parameter settings in an audio unit are called **factory presets**.

All audio units also have characteristics, typically non-time varying and not directly settable by a user, called **properties**. A property is a key/value pair that refines the plug-in API of your audio unit by declaring attributes or behavior. For example, you use the property mechanism to declare such audio unit characteristics as sample latency and audio data stream format. Each property has an associated data type to hold its value. For more on properties, as well as definitions of latency and stream format, see [“Commonly Used Properties”](#) (page 52).

Host applications can query an audio unit about its parameters and about its standard properties, but not about its custom properties. Custom properties are for communication between an audio unit and a custom view designed in concert with the audio unit.

To get parameter information from an audio unit, a host application first gets the value of the audio unit's `kAudioUnitProperty_ParameterList` property, a property provided for you by superclasses in the SDK. This property's value is a list of the defined parameter IDs for the audio unit. The host can then query the `kAudioUnitProperty_ParameterInfo` property for each parameter ID.

Defining and Using Parameters

Specifying parameters means specifying which settings you'd like to offer for control by the user, along with appropriate units and range. For example, an audio unit that provides a tremolo effect might offer a parameter for tremolo rate. You'd probably specify a unit of hertz and might specify a range from 1 to 10.

To define the parameters for an audio unit, you override the `GetParameterInfo` method from the `AUBase` class. You write this method to tell the view how to represent a control for each parameter, and to specify each parameter's default value.

The `GetParameterInfo` method may be called:

- By the audio unit's custom view (if you provide one), or generic view, when the view is drawn on screen
- By a host application that is providing a generic view for the audio unit
- By a host application that is representing the audio unit's parameters on a hardware control surface

To make use of a parameter's current setting (as adjusted by a user) when rendering audio, you call the `GetParameter` method. This method is inherited from the `AUEffectBase` class.

The `GetParameter` method takes a parameter ID as its one argument and returns the parameter's current value. You typically make this call within the audio unit's `Process` method to update the parameter value once for each render cycle. Your rendering code can then use the parameter's current value.

See [“Tutorial: Building a Simple Effect Unit with a Generic View”](#) (page 89) for step-by-step guidance on implementing parameters.

Factory Presets and Parameter Persistence

You specify factory presets to provide convenience and added value for users. The more complex the audio unit, and the greater its number of parameters, the more a user will appreciate factory presets.

For example, the Mac OS X Matrix Reverb unit contains more than a dozen parameters. A user could find setting them in useful combinations daunting. The developers of the Matrix Reverb took this into account and provided a wide range of factory presets with highly descriptive names such as Small Room, Large Hall, and Cathedral.

The `GetPresets` method does for factory presets what the `GetParameterInfo` method does for parameters. You define factory presets by overriding `GetPresets`, and an audio unit's view calls this method to populate the view's factory presets menu.

When a user chooses a factory preset, the view calls the audio unit's `NewFactoryPresetSet` method. You define this method in parallel with the `GetPresets` method. For each preset you offer in the factory presets menu, you include code in the `NewFactoryPresetSet` method to set that preset when the user requests it. For each factory preset, this code consists of a series of `SetParameter` method calls. See [“Tutorial: Building a Simple Effect Unit with a Generic View”](#) (page 89) for step-by-step guidance on implementing factory presets.

Parameter persistence is a feature, provided by a host application, that lets a user save parameter settings from one session to the next. When you develop audio units using the Core Audio SDK, your audio units will automatically support parameter persistence.

Host application developers provide parameter persistence by taking advantage of the SDK's `kAudioUnitProperty_ClassInfo` property. This property uses a `CFPropertyListRef` dictionary to represent the current settings of an audio unit.

Defining and Using Properties

There are more than 100 Apple-defined properties available for audio units. You find their declarations in the `AudioUnitProperties.h` header file in the Audio Unit framework. Each type of audio unit has a set of required properties as described in the *Audio Unit Specification*.

You can get started as an audio unit developer without touching or even being aware of most of these properties. In most cases, superclasses from the Core Audio SDK take care of implementing the required properties for you. And, in many cases, the SDK sets useful values for them.

Yet the more you learn about the rich palette of available audio unit properties, the better you can make your audio units.

Each Apple-defined property has a corresponding data type to represent the property's value. Depending on the property, the data type is a structure, a dictionary, an array, or a floating point number.

For example, the `kAudioUnitProperty_StreamFormat` property, which describes an audio unit's audio data stream format, stores its value in the `AudioStreamBasicDescription` structure. This structure is declared in the `CoreAudioTypes.h` header file in the `CoreAudio` framework.

The `AUBase` superclass provides general getting and setting methods that you can override to implement properties, as described in “Defining Custom Properties” (page 54). These methods, `GetPropertyInfo`, `GetProperty`, and `SetProperty`, work with associated “dispatch” methods in the SDK that you don't call directly. The dispatch methods, such as `DispatchGetPropertyInfo`, provide most of the audio unit property magic for the SDK. You can examine them in the `AUBase.cpp` file in the SDK to see what they do.

The `AUEffectBase` class, `MusicDeviceBase` class, and other subclasses override the property accessor methods with code for properties specific to one type of audio unit. For example, the `AUEffectBase` class handles property calls that are specific to effect units, such as the `kAudioUnitProperty_BypassEffect` property.

Commonly Used Properties

For some commonly used properties, the Core Audio SDK provides specific accessor methods. For example, The `CAStreamBasicDescription` class in the SDK provides methods for managing the `AudioStreamBasicDescription` structure for the `kAudioUnitProperty_StreamFormat` property.

Here are a few of the properties you may need to implement for your audio unit. You implement them when you want to customize your audio unit to vary from the default behavior for the audio unit's type:

- `kAudioUnitProperty_StreamFormat`

Declares the audio data stream format for an audio unit's input or output channels. A host application can set the format for the input and output channels separately. If you don't implement this property to describe additional stream formats, a superclass from the SDK declares that your audio unit supports the default stream format: non-interleaved, 32-bit floating point, native-endian, linear PCM.

- `kAudioUnitProperty_BusCount`

Declares the number of buses (also called elements) in the input or output scope of an audio unit. If you don't implement this property, a superclass from the SDK declares that your audio unit uses a single input and output bus, each with an ID of 0.

- `kAudioUnitProperty_Latency`

Declares the minimum possible time for a sample to proceed from input to output of an audio unit, in seconds. For example, an FFT-based filter must acquire a certain number of samples to fill an FFT window before it can calculate an output sample. An audio unit with a latency as short as two or three samples should implement this property to report its latency.

If the sample latency for your audio unit varies, use this property to report the maximum latency. Alternatively, you can update the `kAudioUnitProperty_Latency` property value when latency changes, and issue a property change notification using the Audio Unit Event API.

If your audio unit's latency is 0 seconds, you don't need to implement this property. Otherwise you should, to let host applications compensate appropriately.

■ `kAudioUnitProperty_TailTime`

Declares the time, beyond an audio unit's latency, for a nominal-level signal to decay to silence at an audio unit's output after it has gone instantaneously to silence at the input. Tail time is significant for audio units performing an effect such as delay or reverberation. Apple recommends that all audio units implement the `kAudioUnitProperty_TailTime` property, even if its value is 0.

If the tail time for your audio unit varies—such as for a variable delay—use this property to report the maximum tail time. Alternatively, you can update the `kAudioUnitProperty_TailTime` property value when tail time changes, and issue a property change notification using the Audio Unit Event API.

■ `kAudioUnitProperty_SupportedNumChannels`

Declares the supported numbers of input and output channels for an audio unit. The value for this property is stored in a channel information structure (`AUChannelInfo`), which is declared in the `AudioUnitProperties.h` header file:

```
typedef struct AUChannelInfo {
    Sint16    inChannels;
    Sint16    outChannels;
} AUChannelInfo;
```

Table 2-1 Using a channel information structure

Field values	Example	Meaning, using example
both fields are -1	<code>inChannels = -1</code> <code>outChannels = -1</code>	This is the default case. Any number of input and output channels, as long as the numbers match
one field is -1, the other field is positive	<code>inChannels = -1</code> <code>outChannels = 2</code>	Any number of input channels, exactly two output channels
both fields have negative values, but the values differ	<code>inChannels = -1</code> <code>outChannels = -2</code>	Any number of input channels, any number of output channels
both fields have positive values	<code>inChannels = 2</code> <code>outChannels = 6</code>	Exactly two input channels, exactly six output channels

If you don't implement this property, a superclass from the SDK declares that your audio unit can use any number of channels provided the number on input matches the number on output.

■ `kAudioUnitProperty_CocoaUI`

Declares where a host application can find the bundle and the main class for a Cocoa-based view for an audio unit. Implement this property if you supply a Cocoa custom view.

The `kAudioUnitProperty_TailTime` property is the most common one you'll need to implement for an effect unit. To do this:

1. Override the `SupportsTail` method from the `AUBase` superclass by adding the following method statement to your audio unit custom class definition:

```
virtual bool SupportsTail () {return true;}
```

2. If your audio unit has a tail time other than 0 seconds, override the `GetTailTime` method from the `AUBase` superclass. For example, if your audio unit produces reverberation with a maximum decay time of 3000 mS, add the following override to your audio unit custom class definition:

```
virtual Float64 GetTailTime() {return 3;}
```

You can find more information on Apple defined properties in the `AudioUnitProperties.h` header file, in the `au_properties.html` page in the `Documentation/AudioUnits/Topics/` folder in the SDK, and in the *Audio Unit Specification*.

Defining Custom Properties

You can define custom audio unit properties for passing information to and from a custom view. For example, the `FilterDemo` project in the Core Audio SDK uses a custom property to communicate the audio unit's frequency response to its view. This allows the view to draw the frequency response as a curve.

To define a custom property when building your audio unit from the SDK, you override the `GetPropertyInfo` and `GetProperty` methods from the `AUBase` class. Your custom view calls these methods when it needs the current value of a property of your audio unit.

You add code to the `GetPropertyInfo` method to return the size of each custom property and a flag indicating whether it is writable. You can also use this method to check that each custom property is being called with an appropriate **scope** and **element**. Listing 2-10 shows this method's signature:

Listing 2-10 The `GetPropertyInfo` method from the SDK's `AUBase` class

```
virtual ComponentResult GetPropertyInfo (
    AudioUnitPropertyID  inID,
    AudioUnitScope       inScope,
    AudioUnitElement     inElement,
    UInt32               &outDataSize,
    Boolean               &outWritable);
```

You add code to the `GetProperty` method to tell the view the current value of each custom property:

Listing 2-11 The `GetProperty` method from the SDK's `AUBase` class

```
virtual ComponentResult GetProperty (
    AudioUnitPropertyID  inID,
    AudioUnitScope       inScope,
    AudioUnitElement     inElement,
    void                 *outData);
```

You would typically structure the `GetPropertyInfo` and `GetProperty` methods as switch statements, with one case per custom property. Look at the `Filter::GetPropertyInfo` and `Filter::GetProperty` methods in the `FilterDemo` project to see an example of how to use these methods.

You override the `SetProperty` method to perform whatever work is required to establish new settings for each custom property.

Each audio unit property must have a unique integer ID. Apple reserves property ID numbers between 0 and 63999. If you use custom properties, specify ID numbers of 64000 or greater.

Synthesis, Processing, and Data Format Conversion Code

Audio units synthesize, process, or transform audio data. You can do anything you want here, according to the desired function of your audio unit. The digital audio code that does this is right at the heart of why you create audio units. Yet such code is largely independent of the plug-in architecture that it lives within. You'd use the same or similar algorithms and data structures for audio units or other audio plug-in architectures. For this reason, this programming guide focuses on creating audio units as containers and interfaces for audio DSP code—not on how to write the DSP code.

At the same time, the way that digital audio code fits into, and interacts with, a plug-in does vary across architectures. This section describes how audio units built with the Core Audio SDK support digital audio code. The chapter [“Tutorial: Building a Simple Effect Unit with a Generic View”](#) (page 89) includes some non-trivial DSP code to help illustrate how it works for effect units.

Signal Processing

To perform DSP, you use an effect unit (of type `'aufx'`), typically built as a subclass of the `AUEffectBase` class. `AUEffectBase` uses a helper class to handle the DSP, `AUKernelBase`, and instantiates one kernel object (`AUKernelBase`) for each audio channel.

Kernel objects are specific to n -to- n channel effect units subclassed from the `AUEffectBase` class. They are not part of other types of audio units.

The `AUEffectBase` class is strictly for building n -to- n channel effect units. If you are building an effect unit that does not employ a direct mapping of input to output channels, you subclass the `AUBase` superclass instead.

As described in [“Processing: The Heart of the Matter”](#) (page 36), there are two primary methods for audio unit DSP code: `Process` and `Reset`. You override the `Process` method to define the DSP for your audio unit. You override the `Reset` method to define the cleanup to perform when a user takes an action to end signal processing, such as moving the playback point in a sound editor window. For example, you ensure with `Reset` that a reverberation decay doesn't interfere with the start of play at a new point in a sound file.

[“Tutorial: Building a Simple Effect Unit with a Generic View”](#) (page 89) provides a step-by-step example of implementing a `Process` method.

While an audio unit is rendering, a user can make realtime adjustments using the audio unit's view. Processing code typically takes into account the current values of parameters and properties that are relevant to the processing. For example, the processing code for a high-pass filter effect unit would perform its calculations based on the current corner frequency as set in the audio unit's view. The processing code gets this value by reading the appropriate parameter, as described in [“Defining and Using Parameters”](#) (page 50).

Audio units built using the classes in the Core Audio SDK work only with constant bit rate (CBR) audio data. When a host application reads variable bit rate (VBR) data, it converts it to a CBR representation, in the form of linear PCM, before sending it to an audio unit.

Music Synthesis

An instrument unit (of type `'aumu'`), in contrast to effect unit, renders audio in terms of notes. It acts as a virtual music synthesizer. An instrument unit employs a bank of sounds and responds to MIDI control data, typically initiated by a keyboard.

You subclass the `AUMonotimbralInstrumentBase` class for most instrument units. This class supports monophonic and polyphonic instrument units that can play one voice (also known as a patch or an instrument sound) at a time. For example, if a user chooses a piano voice, the instrument unit acts like a virtual piano, with every key pressed on a musical keyboard invoking a piano note.

The Core Audio SDK class hierarchy also provides the `AUMultitimbralInstrumentBase` class. This class supports monophonic and polyphonic instrument units that can play more than one voice at a time. For example, you could create a multitimbral instrument unit that would let a user play a virtual bass guitar with their left hand while playing virtual trumpet with their right hand, using a single keyboard.

Music Effects

A music effect unit (of type `'aumf'`) provides DSP, like an effect unit, but also responds to MIDI data, like an instrument unit. You build a music effect unit by subclassing the `AUMIDIEffectBase` superclass from the SDK. For example, you would do this to create an audio unit that provides a filtering effect that is tuned according to the note pressed on a keyboard.

Data Format Conversion

Audio data transformations include such operations as sample rate conversion, sending a signal to multiple destinations, and altering time or pitch. To transform audio data in ways such as there, you build an format converter unit (of type `'aufc'`) as a subclass of the `AUBase` superclass in the SDK.

Audio units are not intended to work with variable bitrate (VBR) data, so audio units are not generally suited for converting to or from lossy compression formats such as MP3. For working with lossy compression formats, use Core Audio's Audio Converter API, declared in the `AudioConverter.h` header file in the Audio Toolbox framework.

Audio Unit Life Cycle

An audio unit is more than its executable code, its view, and its plug-in API. It is a dynamic, responsive entity with a complex life cycle. Here you get a grasp of this life cycle to help you make good design and coding decisions.

Consistent with the rest of this document, this section describes audio unit life cycle in terms of development based on the Core Audio SDK. For example, this section's discussion of object instantiation and initialization refers to SDK subclasses. If you are developing with the Audio Unit framework directly, instead of with the SDK, the audio unit class hierarchy isn't in the picture.

Overview

The life cycle of an audio unit, as for any plug-in, consists of responding to requests. Each method that you override or write from scratch in your audio unit is called by an outside process, among them:

- The Mac OS X Component Manager, acting on behalf of a host application
- A host application itself
- An audio processing graph and, in particular, the downstream audio unit
- The audio unit's view, as manipulated by a user

You don't need to anticipate which process or context is calling your code. To the contrary, you design your audio unit to be agnostic to the calling context.

Audio unit life cycle proceeds through a series of states, which include:

- **Uninstantiated.** In this state, there is no object instance of the audio unit, but the audio unit's class presence on the system is registered by the Component Manager. Host applications use the Component Manager registry to find and open audio units.
- **Instantiated but not initialized.** In this state, host applications can query an audio unit object for its properties and can configure some properties. Users can manipulate the parameters of an instantiated audio unit by way of a view.
- **Initialized.** Host applications can hook up initialized audio units into audio processing graphs. Hosts and graphs can ask initialized audio units to render audio. In addition, some properties can be changed in the initialized state.
- **Uninitialized.** The Audio Unit architecture allows an audio unit to be explicitly uninitialized by a host application. The uninitialization process is not necessarily symmetrical with initialization. For example, an instrument unit can be designed to still have access, in this state, to a MIDI sound bank that it allocated upon initialization.

Categories of Programmatic Events

Audio units respond to two main categories of programmatic events, described in detail later in this chapter:

- *Housekeeping events* that the host application initiates. These include finding, opening, validating, connecting, and closing audio units. For these types of events, an audio unit built from the Core Audio SDK typically relies on code in its supplied superclasses.
- *Operational events* that invoke your custom code. These events, initiated by the host or by your audio unit's view, include initialization, configuration, rendering, resetting, real-time or offline changes to the rendering, uninitialization, reinitialization, and clean-up upon closing. For some simple audio units, some operational events (especially initialization) can also rely on code from SDK superclasses.

Bringing an Audio Unit to Life

Even before instantiation, an audio unit has a sort of ghostly presence in Mac OS X. This is because during user login, the Component Manager builds a list of available audio units. It does this without opening them. Host applications can then find and open audio units using the Component Manager.

Life begins for an audio unit when a host application asks the Component Manager to instantiate it. This typically happens when a host application launches—at which time a host typically instantiates every installed audio unit. Hosts such as AU Lab and Logic Pro do this, for example, to learn about input and output data stream formats as well as the number of inputs and outputs for each installed audio unit. Hosts typically cache this information and then close the audio units.

A host application instantiates a particular audio unit again when a user tells the host that they want to use it by picking it from a menu.

Instantiation results in invocation of the audio unit’s constructor method. To not interfere with the opening speed of host applications, it’s important to keep the constructor method lightweight and fast. The constructor is the place for defining the audio unit’s parameters and setting their initial values. It’s not the place for resource-intensive work.

An n -channel to n -channel effect unit (built from the `AUEffectBase` class) doesn’t instantiate its kernel object or objects until the audio unit is initialized. For this reason, and for this type of audio unit, it’s appropriate to perform resource-intensive work, such as setting up wave tables, during kernel instantiation. For more on this, see “[Kernel Instantiation in \$n\$ -to- \$n\$ Effect Units](#)” (page 60).

Most properties should be implemented and configured in the constructor as well, as described in the next section.

Property Configuration

When possible, an audio unit should configure its properties in its constructor method. However, audio unit properties can be configured at a variety of times and by a variety of entities. Each individual property is usually configured in one of the following ways:

- By the audio unit itself, typically during instantiation
- By the application hosting the audio unit, before or after audio unit initialization
- By the audio unit’s view, as manipulated by a user, when the audio unit is initialized or uninitialized

This variability in configuring audio unit properties derives from the requirements of the various properties, the type of the audio unit, and the needs of the host application.

For some properties, the SDK superclasses define whether configuration can take place while an audio unit is initialized or only when it is uninitialized. For example, a host application cannot change an audio unit’s stream format (using the `kAudioUnitProperty_StreamFormat` property) unless it ensures that the audio unit is uninitialized.

For other properties, such as the `kAudioUnitProperty_SetRenderCallback` property, the audio unit specification prohibits hosts from changing the property on an initialized audio unit but there is no programmatic enforcement against it.

For yet other properties, such as the `kAudioUnitProperty_OfflineRender` property, it is up to the audio unit to determine whether to require uninitialization before changing the property value. If the audio unit can handle the change gracefully while initialized, it can allow it.

The audio unit specification details the configuration requirements for each Apple defined property.

Audio Unit Initialization and Uninitialization

The place for time-intensive and resource-intensive audio unit startup operations is in an audio unit's initialization method. The idea is to postpone as much work in your audio unit as possible until it is about to be used. For example, the AU Lab application doesn't initialize an audio unit installed on the system until the user specifically adds the audio unit to an AU Lab channel. This strategy improves user experience by minimizing untoward delays on host application startup, especially for users who have large numbers of audio units installed.

- An instrument unit acquires a MIDI sound bank for the unit to use when responding to MIDI data
- An effect unit allocates memory buffers for use during rendering
- An effect unit calculates wave tables for use during rendering

Here are some examples of work that's appropriate for initialization time:

Generally speaking, each of these operations should be performed in an override of the `Initialize` method from the `AUBase` class.

If you define an override of the `Initialize` method for an effect unit, begin it with a call to `AUEffectBase::Initialize`. This will ensure that housekeeping tasks, like proper channel setup, are taken care of for your audio unit.

If you are setting up internal buffers for processing, you can find out how large to make them by calling the `AUBase::GetMaxFramesPerSlice` method. This accesses a value that your audio unit's host application defines before it invokes initialization. The actual number of frames per render call can vary. It is set by the host application by using the `inFramesToProcess` parameter of the `AUEffectBase::Process` or `AUBase::DoRender` methods.

Initialization is also the appropriate time to invoke an audio unit's copy protection. Copy protection can include such things as a password challenge or checking for the presence of a hardware dongle.

The audio unit class hierarchy in the Core Audio SDK provides specialized `Initialize` methods for the various types of audio units. Effect units, for example, use the `Initialize` method in the `AUEffectBase` class. This method performs a number of important housekeeping tasks, including:

- Protecting the effect unit against a host application which attempts to connect it up in ways that won't work
- Determining the number of input and output channels supported by the effect unit, as well as the channel configuration to be used for the current initialization.

(Effect units can be designed to support a variable number of input and output channels, and the number used can change from one initialization to the next.)

- Setting up or updating the kernel objects for the effect unit, ensuring they are ready to do their work

In many cases, such as in the effect unit you'll create in [“Tutorial: Building a Simple Effect Unit with a Generic View”](#) (page 89), effect units don't need additional initialization work in the audio unit's class. They can simply use the `Initialize` method from `AUBase` as is, by inheritance. The effect unit you'll build in the tutorial does this.

In the specific case of an effect unit based on the `AUEffectBase` superclass, you can put resource-intensive initialization code into the constructor for the DSP kernel object. This works because kernels are instantiated during effect unit initialization. The example effect unit that you build in [“Tutorial: Building a Simple Effect Unit with a Generic View”](#) (page 89) uses this approach. [“Kernel Instantiation in n-to-n Effect Units”](#) (page 60) describes this part of an effect unit's life cycle.

Once instantiated, a host application can initialize and uninitialize an audio unit repeatedly, as appropriate for what the user wants to do. For example, if a user wants to change sampling rate, the host application can do so without first closing the audio unit. (Some other audio plug-in technologies do not offer this feature.)

Kernel Instantiation in n-to-n Effect Units

In an effect unit built using the `AUEffectBase` superclass—such as the tremolo unit you build in [“Tutorial: Building a Simple Effect Unit with a Generic View”](#) (page 89)—processing takes place in one or more so-called kernel objects. These objects are subclassed from the `AUKernelBase` class, as described in [“Bringing an Audio Unit to Life”](#) (page 58).

Such effect units instantiate one kernel object for each channel being used. Kernel object instantiation takes place during the audio unit initialization, as part of this sequence:

1. An n -channel to n -channel effect unit gets instantiated
2. The effect unit gets initialized
3. During initialization, the effect unit instantiates an appropriate number of kernel objects

This sequence of events makes the kernel object constructor a good place for code that you want invoked during audio unit initialization. For example, the tremolo unit in this document's tutorial builds its tremolo wave tables during kernel instantiation.

Audio Processing Graph Interactions

Audio processing graphs—hookups of multiple audio units—are the most common way that your audio unit will be used. This section describes what happens in a graph.

Interactions of an audio unit with an audio processing graph include:

- Establishing input and output connections
- Breaking input and output connections
- Responding to rendering requests

A host application is responsible for making and breaking connections for an audio processing graph. Performing connection and disconnection takes place by way of setting properties, as discussed earlier in this chapter in [“Audio Processing Graph Connections”](#) (page 47). For an audio unit to be added to or removed from a graph, it must be uninitialized.

Audio data flow in graphs proceeds according to a pull model, as described in [“Audio Processing Graphs and the Pull Model”](#) (page 34).

Audio Unit Processing

Depending on its type, an audio unit does one of the following:

- Processes audio (for example, effect units and music effect units)
- Generates audio from MIDI data (instrument units) or otherwise, such as by reading a file (generator units)
- Transforms audio data (format converter units) such as by changing sample rate, bit depth, encoding scheme, or some other audio data characteristic

In effect units built using the Core Audio SDK, the processing work takes place in a C++ method called `Process`. This method, from the `AUKernelBase` class, is declared in the `AUEffectBase.h` header file in the SDK. In instrument units built using the SDK, the audio generation work takes place in a method called `Render`, defined in the `AUInstrumentBase` class.

In an effect unit, processing starts when the unit receives a call to its `Process` method. This call typically comes from the downstream audio unit in an audio processing graph. As described in [“Audio Processing Graph Interactions”](#) (page 60), the call is the result of a cascade originating from the host application, by way of the graph object, asking the final node in the graph to start.

The processing call that the audio unit receives specifies the input and output buffers as well as the amount of data to process:

Listing 2-12 The `Process` method from the `AUKernelBase` class

```
virtual void Process (
    const Float32    *inSourceP,
    Float32          *inDestP,
    UInt32           inFramesToProcess,
    UInt32           inNumChannels,
    bool &           ioSilence) = 0;
```

For an example implementation of the `Process` method, see [“Tutorial: Building a Simple Effect Unit with a Generic View”](#) (page 89).

Processing is the most computationally expensive part of an audio unit’s life cycle. Within the processing loop, avoid the following actions:

- Mutually exclusive (mutex) resource locking
- Memory or resource allocation

Note: Some Core Foundation calls, such as `CFRetain` and `CFRelease`, employ mutex locks. For this reason, it's best to avoid Core Foundation calls during processing.

Closing

When a host is finished using an audio unit, it should close it by calling the Component Manager's `CloseComponent` function. This function invokes the audio unit's destructor method. Audio units themselves must take care of freeing any resources they have allocated.

If you're using copy protection in your audio unit, you should end it only on object destruction.

The Audio Unit View

Almost every audio unit needs a graphical interface to let the user adjust the audio unit's operation and see what it's doing. In Core Audio terminology, such a graphical interface is called a view. When you understand how views work and how to build them, you can add a great deal of value to the audio units you create.

Types of Views

There are two main types of views:

- **Generic views** provide a functional yet decidedly non-flashy interface. You get a generic view for free. It is built for you by the host application that opens your audio unit, based on parameter and property definitions in your audio unit source files.
- **Custom views** are graphical user interfaces that you design and build. Creating a great custom view may entail more than half of the development time for an audio unit. For your effort you can offer something to your users that is not only more attractive but more functional as well.

You may choose to wait on developing a custom view until after your audio unit is working, or you may forgo the custom view option entirely. If you do create a custom view, you can use Carbon or Cocoa.

Separation of Concerns

From the standpoint of a user, a view *is* an audio unit. From your standpoint as a developer, the situation is a bit more subtle.

You build a view to be logically separate from the audio unit executable code, yet packaged within the same bundle. To achieve this programmatic separation, Apple recommends that you develop your custom views so that they would work running in a separate address space, in a separate process, and on a separate machine from the audio unit executable code. For example, you pass data between the audio unit executable code and its view only by value, never by reference.

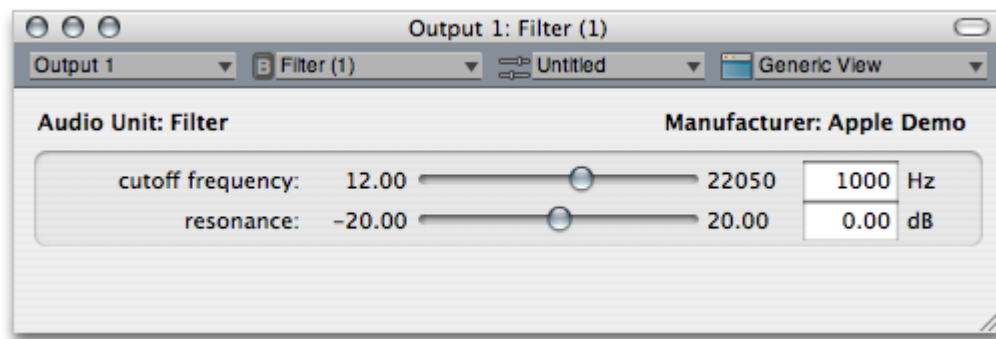
Without impinging on this formal separation, however, it's often convenient to share a header file between an audio unit executable code and its view. You'll see this done in the SDK's FilterDemo project. A shared header file can provide, for example, data types and constants for custom properties that an audio unit's executable code and its view can both use.

Both generic and custom views make use of a notification mechanism to communicate with their associated audio unit executable code, as described in [“Parameter and Property Events”](#) (page 68).

The Generic View

Host applications create a generic view for your audio unit based on your parameter and property definitions. Starting with Mac OS X v10.4, Tiger, host applications can generate Carbon or Cocoa generic views.

Here's the Cocoa generic view for the FilterDemo audio unit, one of the sample projects in the Core Audio SDK:



The audio unit associated with this view has two continuously variable parameters, each represented in the view by a simple slider along with a text field showing the parameter's current value. The generic view for your audio unit will have a similar look.

Table 3-1 describes where each user interface element in a generic view comes from. The “Source of value” column in the table refers to files you see in an audio unit Xcode project built using the “Audio Unit Effect with Cocoa View” template.

Table 3-1 User interface items in an audio unit generic view

User interface item	Example value	Source of value
Audio unit name, at upper left of view utility window	Audio Unit: Filter	NAME defined constant in the <code><className>.r</code> file
Audio unit name, in title bar of view utility window and in pop-up	Filter	NAME defined constant in the <code><className>.r</code> file
Manufacturer name, at upper right of view utility window	Apple Demo	NAME defined constant in the <code><className>.r</code> file

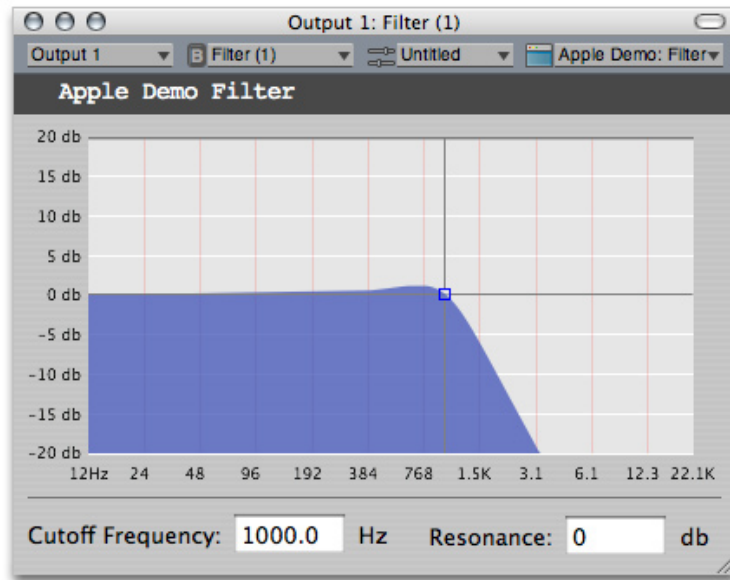
User interface item	Example value	Source of value
Parameter name	cutoff frequency	kParameterOneName CFStringRef string value in the <className>.h file
Parameter maximum value	22050	GetParameterInfo method in the <className>.cpp file
Parameter minimum value	12.00	GetParameterInfo method in the <className>.cpp file
Parameter default value	1000	kDefaultValue_ParamOne float value from the <className>.h file
Measurement units name	Hz	Measurement unit as specified in the GetParameterInfo method in the <className>.cpp file.
Slider	User adjustable. Initially set to indicate the parameter default value.	The generic view mechanism presents a slider for this parameter. This is because the parameter unit of measurement, as defined in the GetParameterInfo method, is “linear gain.”

As shown next, a custom view can provide significantly more value and utility than a generic view.

Custom Views

When a host opens your audio unit, it asks if there’s a custom view available. If so, it can use it as described in [“View Instantiation and Initialization”](#) (page 66).

Here’s the custom view for the same audio unit described in “The Generic View”, namely, the FilterDemo audio unit from the Core Audio SDK:



The primary feature of this custom view is a realtime frequency response curve. This makes the custom view (and, by association, the audio unit) more attractive and far more useful. Instead of seeing just a pair of numbers, a user can now see the frequency response, including how filter resonance and cutoff frequency influence the response. No matter what sort of audio unit you build, you can provide similar benefits to users when you include a custom view with your audio unit.

The advantages of custom views over generic views include:

- The ability to hide unneeded detail, or to provide progressive disclosure of controls
- The ability to provide support for parameter automation
- Choice of user-interface controls, for example knobs, faders, or horizontal sliders
- Much more information for the user through real-time graphs, such as frequency response curves
- A branding opportunity for your company

The SDK's FilterDemo audio unit project is a good example to follow when creating a custom view for your audio unit. See [“Tutorial: Demonstrating Parameter Gestures and Audio Unit Events”](#) (page 70) later in this chapter for more on custom views.

View Instantiation and Initialization

When a host application opens an audio unit, it can query the audio unit as to whether it has a custom view. The host does this with code such as this snippet from the CocoaAUHost project in the SDK:

Listing 3-1 A host application gets a Cocoa custom view from an audio unit

```
if (AudioUnitGetProperty (
    inAU,                                // the audio unit the host is checking
    kAudioUnitProperty_CocoaUI, // the property the host is querying
```

```

        kAudioUnitScope_Global,
        0,
        cocoaViewInfo,
        &dataSize) == noErr) {
    CocoaViewBundlePath =          // the host gets the path to the view bundle
        (NSURL *) cocoaViewInfo -> mCocoaAUVViewBundleLocation;
    factoryClassName =             // the host gets the view's class name
        (NSString *) cocoaViewInfo -> mCocoaAUVViewClass[0];
}

```

If you do not supply a custom view with your audio unit, the host will build a generic view based on your audio unit's parameter and property definitions.

Here is what happens, in terms of presenting a view to the user, when a host opens an audio unit:

1. The host application calls the `GetProperty` method on an audio unit to find out if it has a custom view, as shown in Listing 3-1. If the audio unit provides a Cocoa view, the audio unit should implement the `kAudioUnitProperty_CocoaUI` property. If the audio unit provides a Carbon view, the audio unit should implement the `kAudioUnitProperty_GetUIComponentList` property. The rest of this sequence assumes the use of a Cocoa custom view.
2. The host calls the `GetPropertyInfo` method for the `kAudioUnitProperty_CocoaUI` property to find out how many Cocoa custom views are available. As a short cut, a host can skip the call to `GetPropertyInfo`. In this case, the host would take the first view in the view class array by using code such as shown in the listing above, using an array index of 0: `factoryClassName = (NSString *) cocoaViewInfo -> mCocoaAUVViewClass[0];`. In this case, skip ahead to step 4.
3. The audio unit returns the size of the `AudioUnitCocoaViewInfo` structure as an integer value, indicating how many Cocoa custom views are available. Typically, developers create one view per audio unit.
4. The host examines the value of `cocoaViewInfo` to find out where the view bundle is and what the main view class is for the view (or for the specified view if the audio unit provides more than one).
5. The host loads the view bundle, starting by loading the main view class to instantiate it.

There are some rules about how to structure the main view class for a Cocoa view:

- The view must implement the `AUCocoaUIBase` protocol. This protocol specifies that the view class acts as a factory for views, and returns an `NSView` object using the `uiViewForAudioUnit:withSize:` method. This method tells the view which audio unit owns it, and provides a hint regarding screen size for the view in pixels (using an `NSSize` structure).


```

- (NSView *) uiViewForAudioUnit: (AudioUnit) inAudioUnit
               withSize: (NSSize) inPreferredSize;

```
- If you're using a nib file to construct the view (as opposed to generating the view programmatically), the owner of the nib file is the main (factory) class for the view.

An audio unit's view should work whether the audio unit is simply instantiated or whether it has been initialized. The view should continue to work if the host uninitializes the audio unit. That is, a view should not assume that its audio unit is initialized. This is important enough in practice that the `auval` tool includes a test for retention of parameter values across uninitialization and reinitialization.

Parameter and Property Events

If you provide appropriate code to support parameter changes, your audio unit can be adjusted programmatically or by a user. A user can adjust parameters manually using the audio unit's view. A host application can record these manual changes along with synchronization information, tying the changes to time markers for an audio track. The host can then play back the parameter changes to provide automated control of the audio unit.

Host applications can also provide the ability for a user to indirectly specify parameter manipulation. For example, a host can let a user draw a volume or panning curve along an audio track's waveform representation. The host can then translate such graphical input into parameter automation data.

As an audio unit developer, you support all of these highly useful host-implemented features when you use the appropriate calls for setting parameter values in your audio unit.

The Audio Unit Event API

Core Audio provides a notification mechanism for ensuring that an audio unit, its view, and a host application all stay in sync in terms of parameter adjustments. This mechanism, the Audio Unit Event API, works no matter which of these three entities performs a parameter change. The API is declared in the `AudioUnitUtilities.h` header file in the Audio Toolbox framework.

The Audio Unit Event API defines an `AudioUnitEvent` data type, shown in the following listing:

Listing 3-2 The `AudioUnitEvent` structure

```
typedef struct AudioUnitEvent {
    AudioUnitEventType mEventType;           // 1
    union {
        AudioUnitParameter mParameter;      // 2
        AudioUnitProperty mProperty;         // 3
    } mArgument;
} AudioUnitEvent;
```

Here's how this structure works:

1. Identifies the type of the notification, as defined in the `AudioUnitEventType` enumeration.
2. Identifies the parameter involved in the notification, for notifications that are begin or end gestures or changes to parameters. (See [“Parameter Gestures”](#) (page 70).) The `AudioUnitParameter` data type is used by the Audio Unit Event API and not by the Audio Unit framework, even though it is defined in the Audio Unit framework.
3. Identifies the property involved in the notification, for notifications that are property change notifications.

A corresponding `AudioUnitEventType` enumeration lists the various defined `AudioUnitEvent` event types:

Listing 3-3 The AudioUnitEventType enumeration

```
typedef UInt32 AudioUnitEventType;
```

```
enum {
    kAudioUnitEvent_ParameterValueChange      = 0,
    kAudioUnitEvent_BeginParameterChangeGesture = 1,
    kAudioUnitEvent_EndParameterChangeGesture  = 2,
    kAudioUnitEvent_PropertyChange             = 3
};
```

kAudioUnitEvent_ParameterValueChange

Indicates that the notification describes a change in the value of a parameter

kAudioUnitEvent_BeginParameterChangeGesture

Indicates that the notification describes a parameter “begin” gesture; a parameter value is about to change

kAudioUnitEvent_EndParameterChangeGesture

Indicates that the notification describes a parameter “end” gesture; a parameter value has finished changing

kAudioUnitEvent_PropertyChange

Indicates that the notification describes a change in the value of an audio unit property

Basic Parameter Adjustments

For basic parameter adjustment, Core Audio provides the `AudioUnitSetParameter` function, declared in the `AUComponent.h` header file in the Audio Unit framework. This function simply sets the specified parameter. It does not provide notification to support automation or updating of views.

Parameter Adjustments with Notification

To add notification that a parameter has changed, follow a call to `AudioUnitSetParameter` with a call to the `AUParameterListenerNotify` function from the Audio Unit Event API.

To set a parameter and notify listeners all in one step, use the `AUParameterSet` function, also from the Audio Unit Event API.

In the most common case, namely setting parameters in the global scope for an effect unit, the SDK provides a very convenient wrapper method. This method, `SetParameter`, from the `AUEffectBase` class, takes just two method parameters—the ID of the parameter to be changed and its new value—and not only adjusts the parameter but also notifies listeners of the change.

Listing 3-4 The `SetParameter` convenience method

```
void SetParameter(
    UInt32 paramID,
    Float32 value
);
```

Most of the audio unit examples in the Core Audio SDK use the `SetParameter` method. The audio unit you build in “[Tutorial: Building a Simple Effect Unit with a Generic View](#)” (page 89) does as well.

Parameter Gestures

User-interface events that signal the start or end of a parameter change are called gestures. These events can serve to pass notification among a host application, a view, and an audio unit that a parameter is about to be changed, or has just finished being changed. Like parameter changes, gestures are communicated using the Audio Unit Event API. Specifically, gestures use the `kAudioUnitEvent_BeginParameterChangeGesture` and `kAudioUnitEvent_EndParameterChangeGesture` event types.

When you use the `SetParameter` method from the `AUEffectBase` class in an effect unit, as described above in “Parameter Adjustments with Notification”, the method also takes care of providing begin and end gesture notifications. To provide gesture notification explicitly, you make use of the functions in the Audio Unit Event API.

Tutorial: Demonstrating Parameter Gestures and Audio Unit Events

This mini-tutorial illustrates how gestures and audio unit events work by:

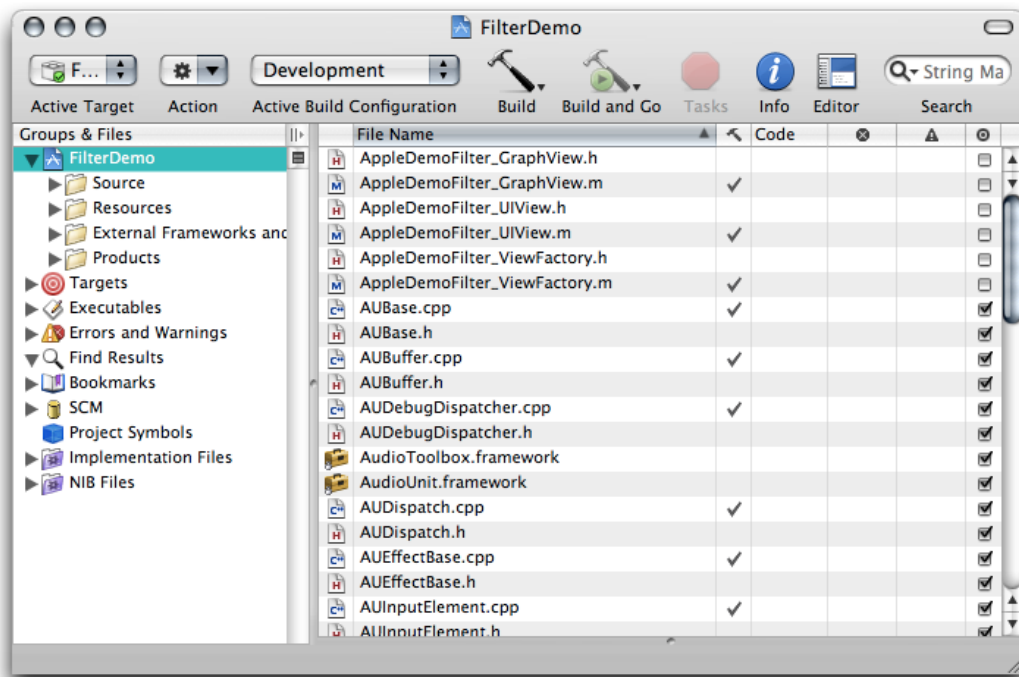
- Instantiating two views for an audio unit
- Performing adjustments in one of the views while observing the effect in the other view

Along the way, this tutorial:

- Introduces you to compiling an audio unit project with Xcode
- Shows how AU Lab can display a generic view and a custom view for the same audio unit
- Shows how a custom property supports communication between an audio unit and a custom view

Before you start, make sure that you’ve installed Xcode and the Core Audio SDK, both of which are part of Apple’s Xcode Tools installation.

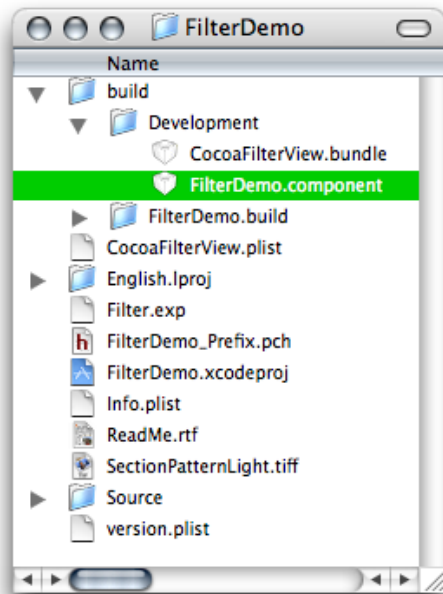
1. Open the `FilterDemo.xcodeproj` Xcode project file in the `Developer/Examples/CoreAudio/AudioUnits/FilterDemo` folder.



2. Click Build to build the audio unit project. (You may see some warnings about “non-virtual destructors.” Ignore these warnings.)

Xcode creates a new `build` folder inside the `FilterDemo` project folder.

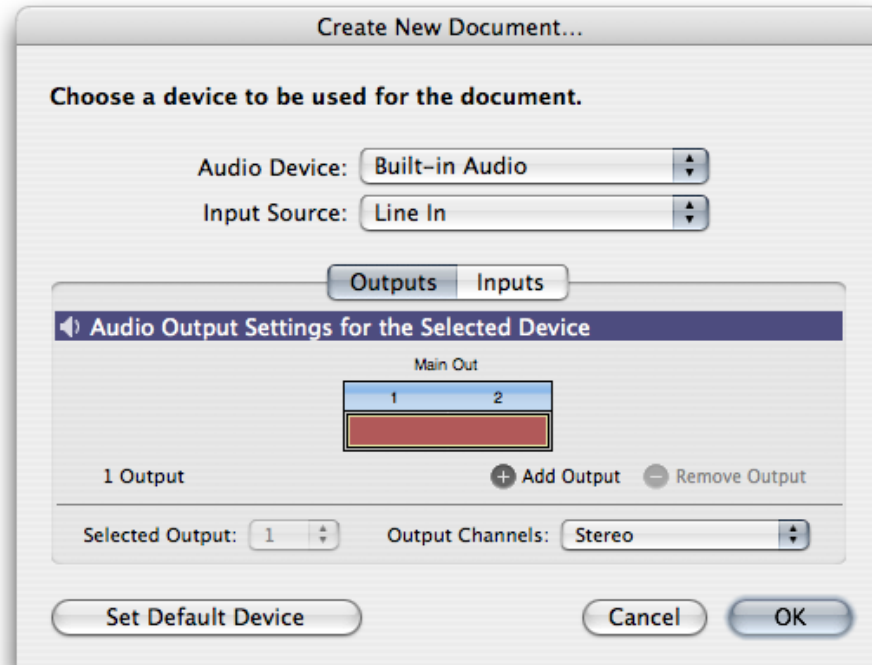
3. Open the `build` folder and look inside the `Development` target folder.



The newly built audio unit bundle is named `FilterDemo.component`, as shown in the figure.

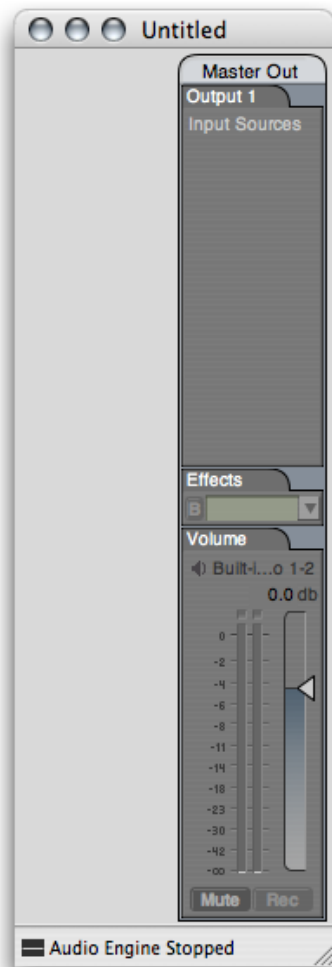
4. Copy the `FilterDemo.component` bundle to the `~/Library/Audio/Plug-Ins/Components` folder. In this location, the newly built audio unit is available to host applications.

5. Launch the AU Lab audio unit host application (in `/Developer/Applications/Audio/`) and create a new AU Lab document. Unless you've configured AU Lab to use a default document style, the Create New Document window opens. If AU Lab was already running, choose `File > New` to get this window.

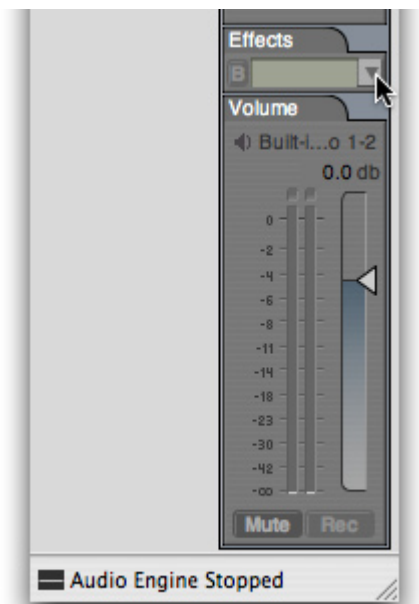


Ensure that the configuration matches the settings shown in the figure: Built-In Audio for the Audio Device, Line In for the Input Source, and Stereo for Output Channels. Leave the window's Inputs tab unconfigured; you will specify the input later. Click OK.

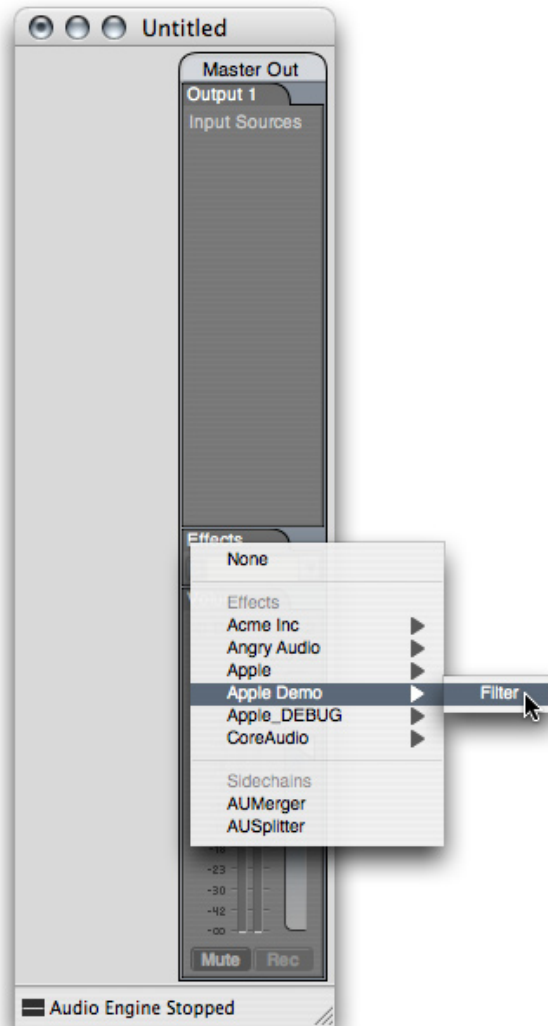
A new AU Lab window opens, showing the output channel you specified.



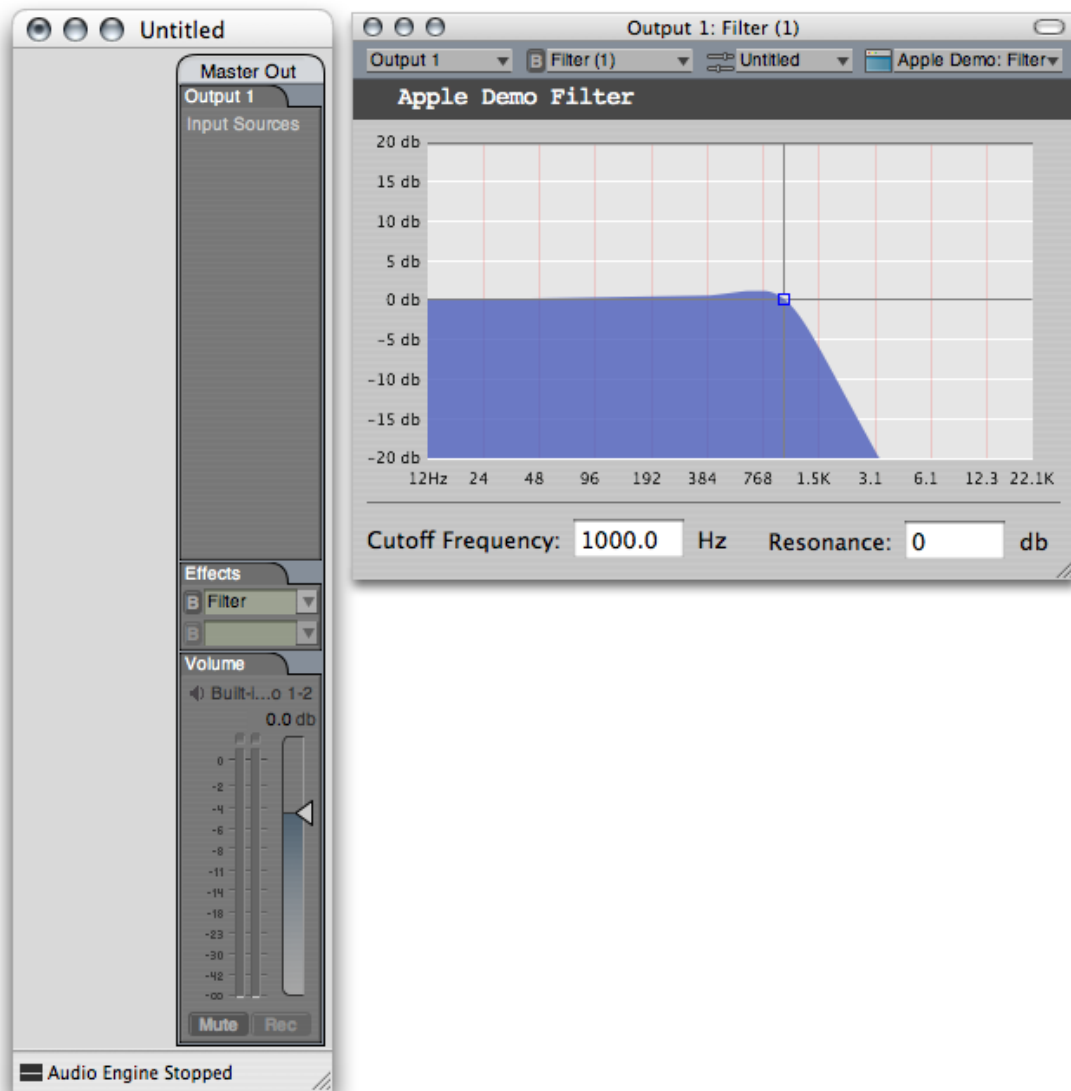
6. Click the triangular menu button in the one row of the Effects section in the Master Out track in AU Lab, as shown in the figure.



In the menu that opens, choose the Filter audio unit from the Apple Demo group:

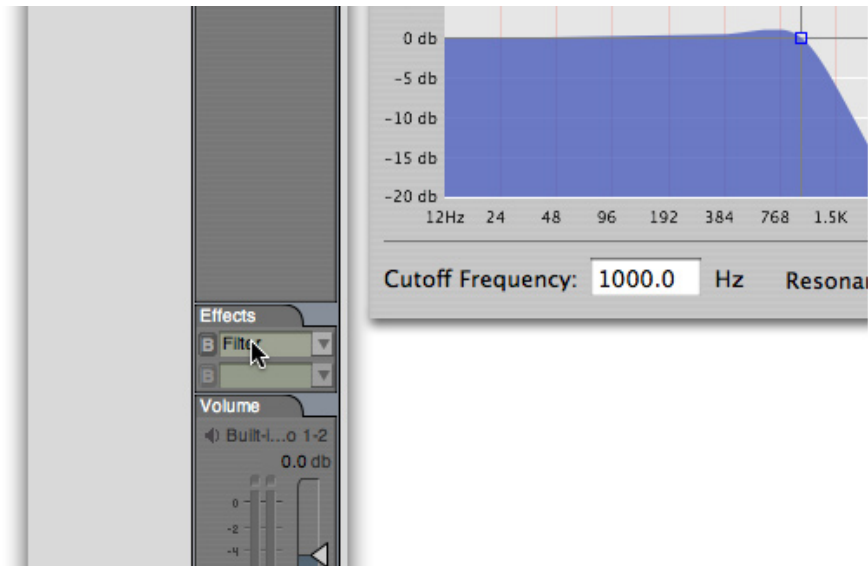


The custom view for the Filter audio unit opens.

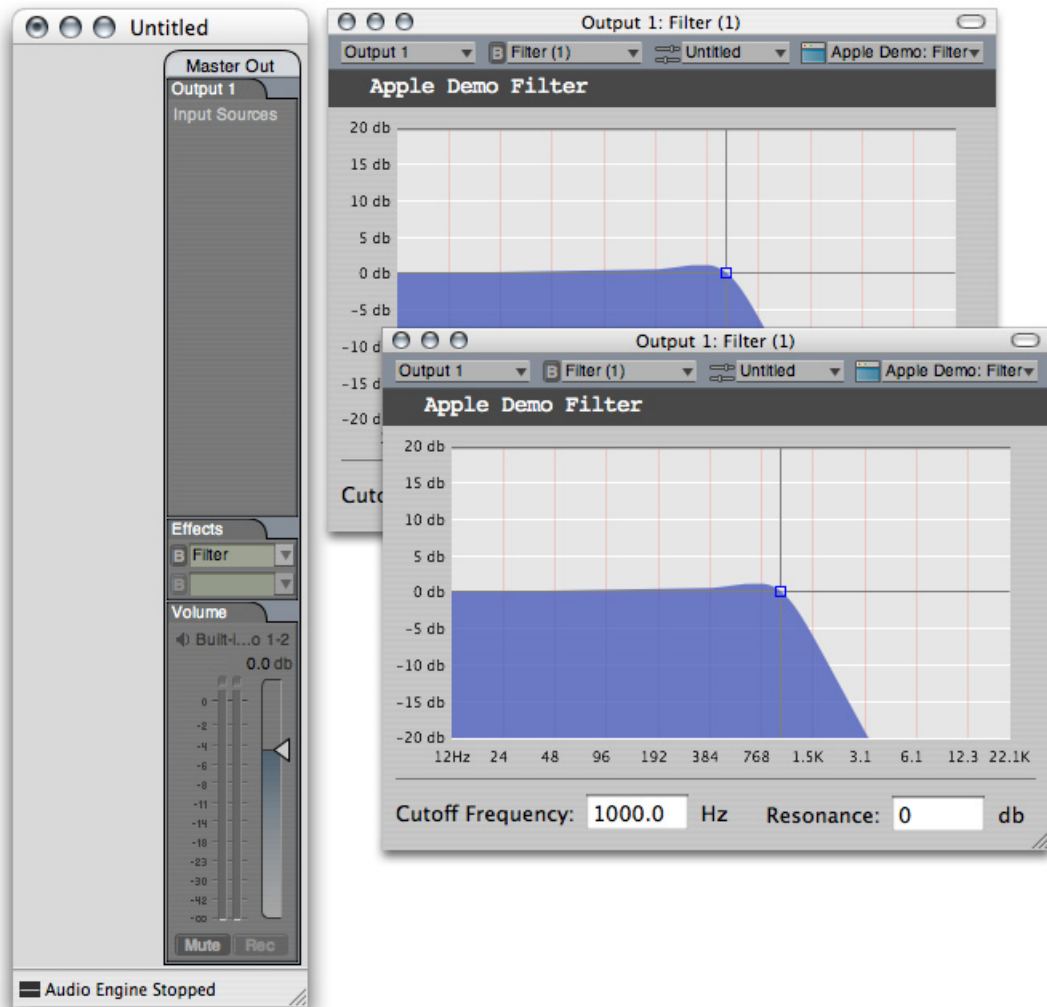


The custom view's frequency response curve is drawn in real time based on the audio unit's actual frequency response. The audio unit makes its frequency response data available to the custom view by declaring a custom property. The audio unit keeps the value of its custom property up to date. The custom view queries the audio unit's custom property to draw the frequency response curve.

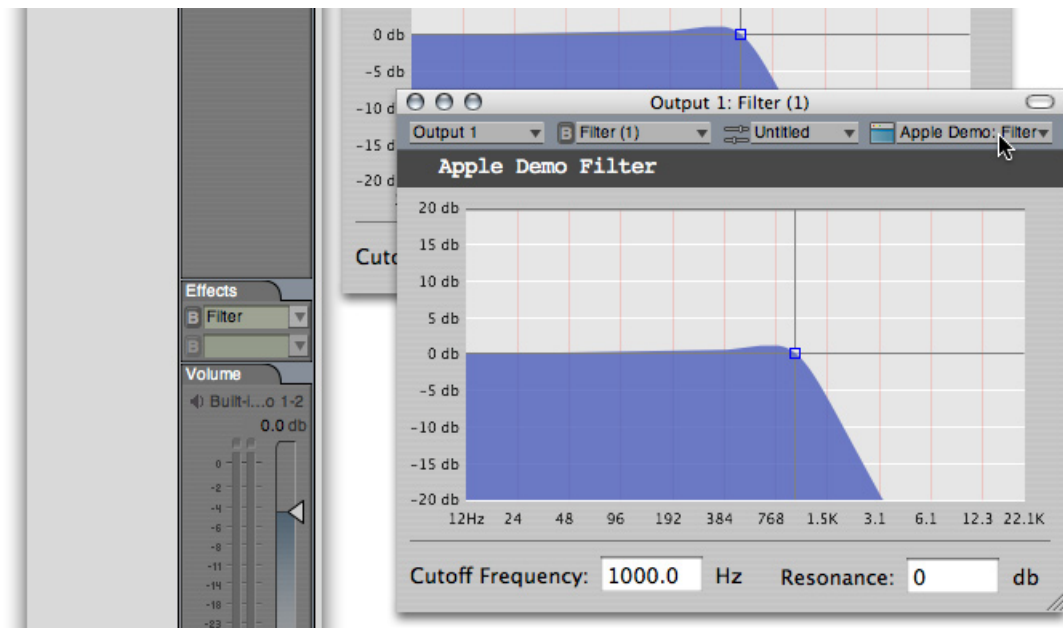
7. Option-click the audio unit name in the Effects row.



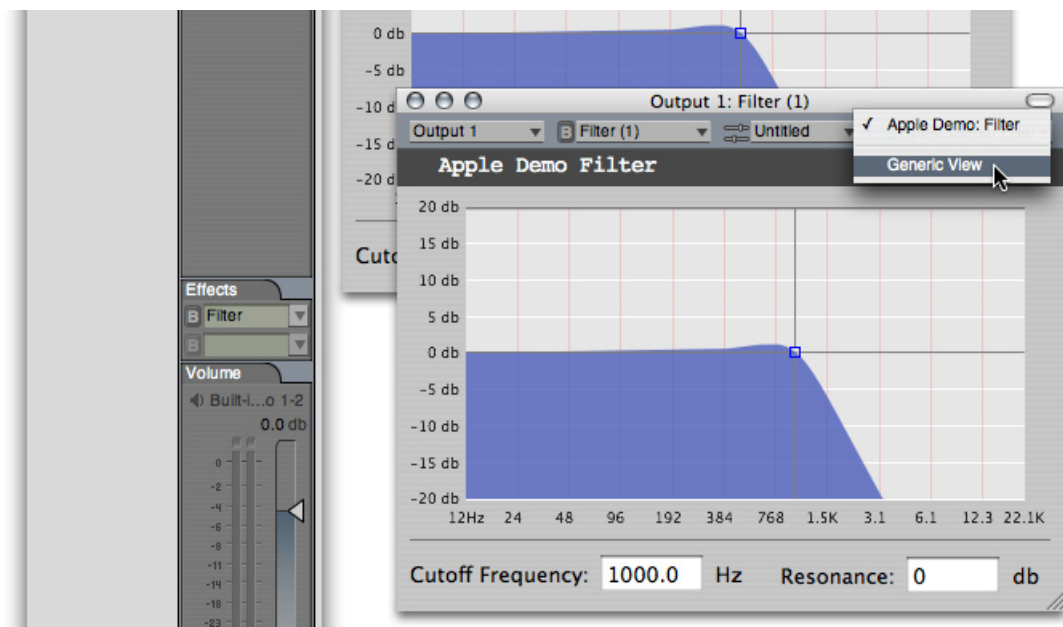
A second instance of a view for the Filter audio unit opens.



8. Click the view type pop-up menu in one instance of the audio unit's view, as shown in the figure:

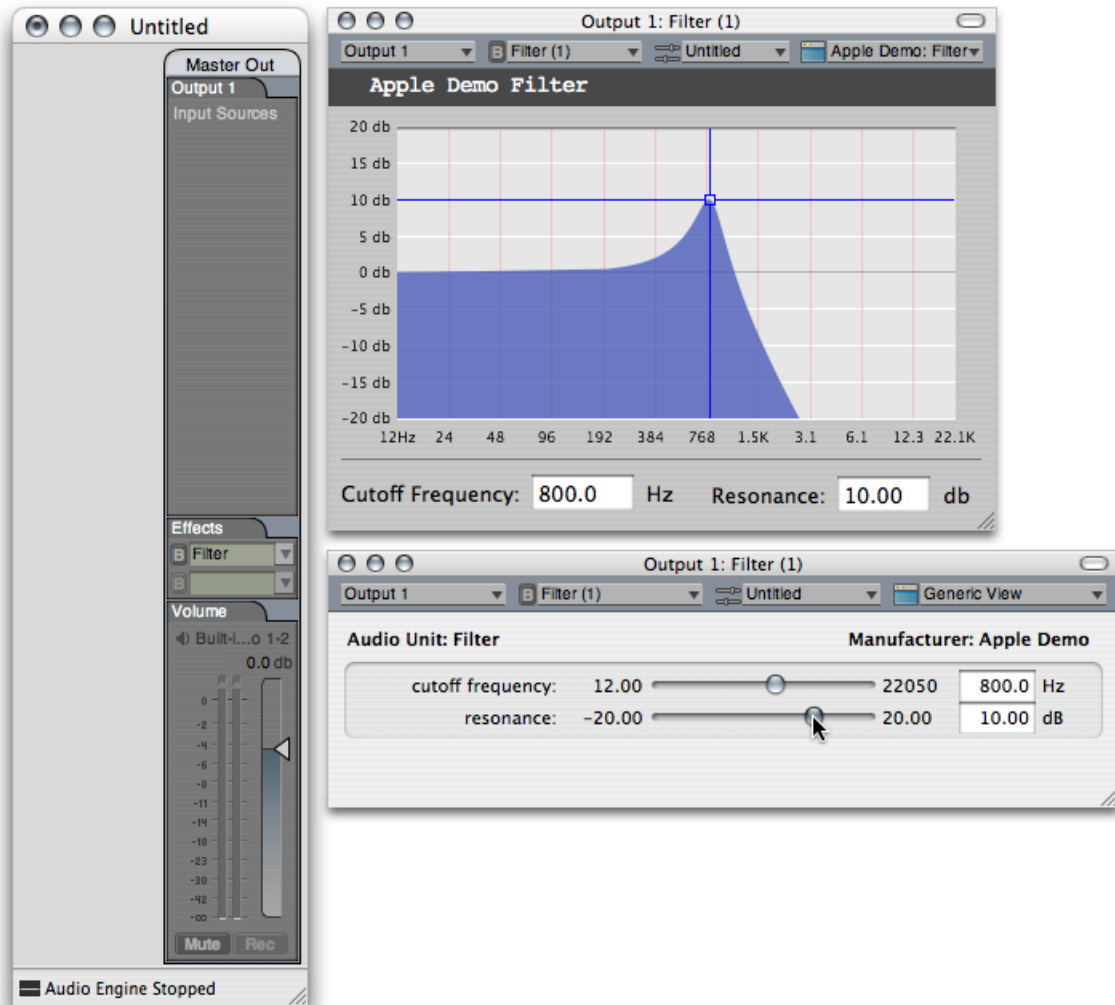


In the menu that opens, choose the Generic View item:



The view changes to the generic view, as shown in the next figure. You are now set up to demonstrate gestures and audio unit events.

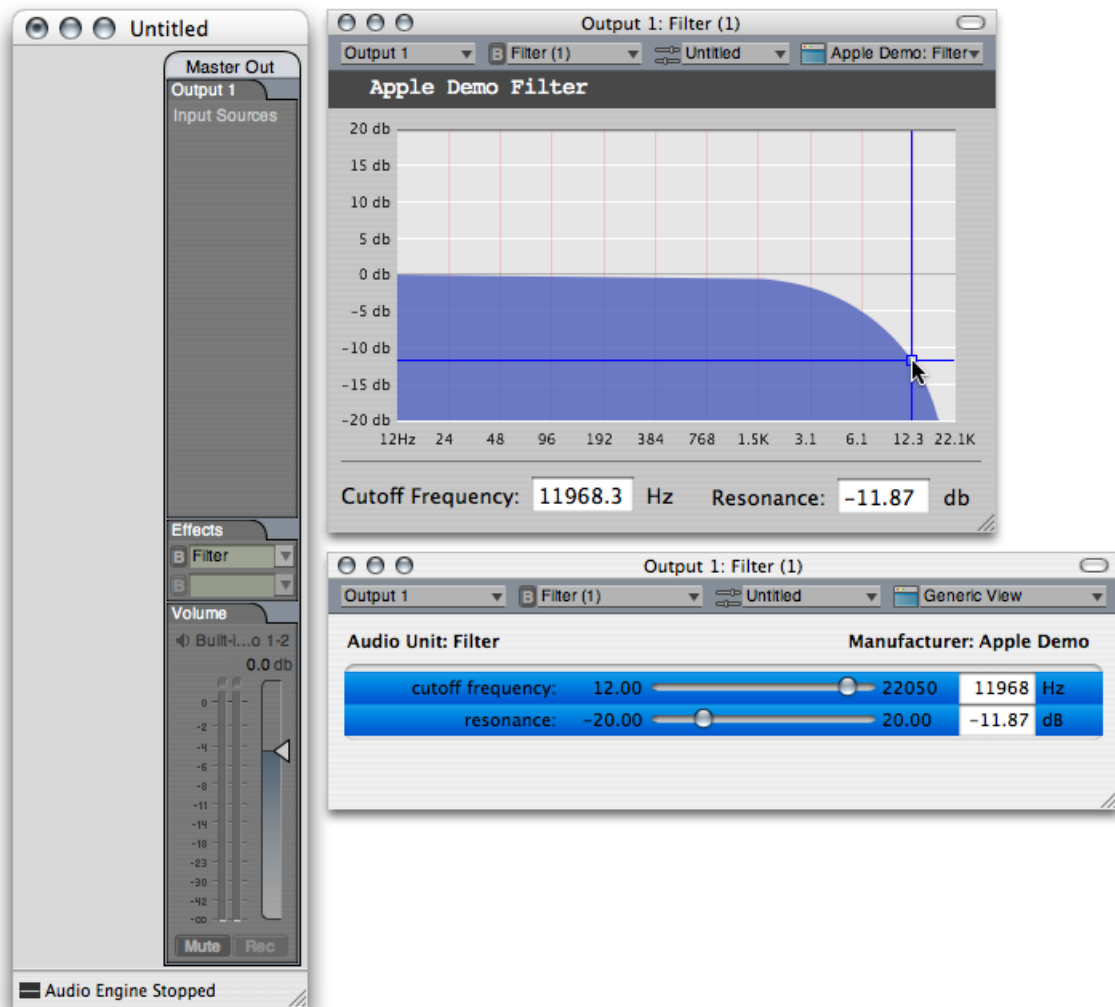
9. Click and hold one of the sliders in the generic view, as shown in the figure. When you click, observe that the crosshairs in the custom view become highlighted in bright blue. They remain highlighted as long as you hold down the mouse button.



As you move the slider in the generic view, frequency response curve in the custom view keeps pace with the new setting.

- The highlighting and un-highlighting of the crosshairs in the custom view, when you click and release on a slider in the generic view, result from gesture events.
- The changes in the frequency response curve in the custom view, as you move a slider in the generic view, result from parameter change events

10. Finally, click and hold at the intersection of the crosshairs in the custom view. When you click, observe that the sliders in the generic view become highlighted. As you move the crosshairs in the custom view, the sliders in the generic view keep pace with the new settings.



This demonstrates that both views, and the audio unit itself, remain in sync by way of audio unit events.

A Quick Tour of the Core Audio SDK

You can build an audio unit from scratch using the Core Audio frameworks directly. However, as described throughout this document, Apple strongly encourages you to begin with the freely downloadable Core Audio software development kit, or SDK. Apple builds all of the audio units that ship in Mac OS X using the SDK.

The SDK offers many advantages to audio unit developers. The SDK:

- Insulates you from almost all the complexity in dealing with the Mac OS X Component Manager
- Greatly simplifies development effort with a rich C++ class hierarchy. In many cases, you can create audio units with a few method overrides. This lets you build full-featured, commercial quality audio units without directly calling any of the APIs in the Core Audio frameworks.
- Provides a straightforward starting point with Xcode audio unit project templates.

Obtaining the Core Audio SDK

Install the most recent Core Audio SDK. It is part of the Xcode Tools installation on the Mac OS X DVD. You can also download it from Apple's developer website at this location:

<http://developer.apple.com/sdk/>

The Core Audio team updates the SDK frequently so be sure that you're using the most current version.

The installer package places the SDK in a folder named `CoreAudio` at the path:

`/Developer/Examples/CoreAudio/` on your startup volume.

Navigating within the Core Audio SDK

Note: This section describes version 1.4.3 of the Core Audio SDK. The Core Audio team updates the SDK frequently. As with all Apple SDKs, use the most current version.

The Core Audio SDK contains materials for working with audio units, audio files, codecs, MIDI, and the HAL. The `ReadMe.rtf` file within the `CoreAudio` folder helps you get oriented. It includes a commented listing of the folders in the SDK, a list of the SDK's own documentation, pointers to related resources from the Core Audio team, and release notes.

For audio unit development, the most important parts of the SDK are in the following folders:

- `AudioUnits`
- `Documentation`
- `PublicUtility`
- `Services`

This section describes each of these folders in turn.

The AudioUnits Folder

The `AudioUnits` folder contains the C++ class hierarchy that you use to build audio units. This is in the `AudioUnits/AUPublic` folder. The audio unit Xcode templates depend on this class hierarchy.

The `AudioUnits/CarbonGenericView` folder contains the source files for the Carbon generic view.

Host developers find the interfaces for Apple's Cocoa generic view not in the Core Audio SDK but in Mac OS X's `CoreAudioKit` framework.

In addition to the `AUPublic` and `CarbonGenericView` source folders, the `AudioUnits` folder contains several complete example Xcode projects for audio units. You can use these projects in a variety of ways:

- Directly using the the audio units built by these projects
- Studying the source code to gain insight into how to design and implement audio units
- Using the projects as starting points for your own audio units

The SDK includes the following example audio unit projects:

- The `DiagnosticAUs` project builds three audio units that you may find useful for troubleshooting and analysis as you're developing your audio units:
 - `AUValidSamples` detects samples passing through it that are out of range or otherwise invalid. You can choose this audio unit from the `Apple_DEBUG` group in an audio unit host such as `AU Lab`.
 - `AUDebugDispatcher` facilitates audio unit debugging. You can choose this audio unit from the `Acme Inc` group in an audio unit host such as `AU Lab`.

- ❑ AUPulseDetector measures latency in host applications
- The FilterDemo project builds a basic resonant low-pass filter with a custom view.
- The MultitapDelayAU project builds a multi-tap delay with a custom view.
- The SampleAUs project builds a pass-through audio unit that includes presets and a variety of parameter types.
- The SinSynth project builds a simple instrument unit that you can use in a host application like GarageBand.

The Documentation Folder

Next, examine the contents of the SDK's `Documentation` folder. This folder contains a great deal of documentation, provided by the Core Audio engineering team, that applies to audio unit development.

HTML Documentation

To view the HTML files in this folder, open the `index.html` file directly within the `Documentation` folder: `file:///Developer/Examples/CoreAudio/Documentation/index.html`. This HTML page links to five sections in the SDK documentation. The four that concern audio unit development are:

- **HAL**
Includes the documentation for the data types and constants used by audio units, under the link “CoreAudioTypes.” The most important structure described here is `AudioStreamBasicDescription`.
- **AU Lab Documentation**
Documentation for the AU Lab application. You can also access this content from the Help menu within AU Lab.
- **Audio Units**
Audio unit documentation from the Core Audio team.
- **Building with Older OS SDKs**
Notes on building audio units that run on older versions of Mac OS X.

Additional Documentation

Within the `CoreAudio/Documentation/AudioUnits` folder, there are more than a dozen PDF, RTF, and text documents. Some apply to host application development, some to development of specific types of audio units, and some to advanced audio unit development. A few are of general interest to all audio unit developers.

The following list highlights the critical documents in this folder that every audio unit developer should read:

- **AUParameterStrings.rtf**
Information on controlling the user interface representation of parameter values and units.

- **AUValidationReadMe.rtf**

Detailed instructions on using the `auval` command-line tool for validating audio units, including debugging options, testing for memory issues, and miscellaneous usage tips.

- **IOChannelConfigurations.rtf**

Instructions for developing an audio unit whose number of inputs does not match its number of outputs. Also includes an explanation of audio unit architecture in terms of scopes, elements, groups, and parts.

- **WritingCocoaAUUIs.pdf**

Instructions for developing a Cocoa-based custom view.

The PublicUtility Folder

This folder contains a miscellaneous collection of C++ and Objective-C source files used by the Core Audio SDK's audio unit class hierarchy and by the sample projects in the SDK. You may want to examine these files to gain insight into how the class hierarchy works.

The Services Folder

This folder contains several Core Audio Xcode projects that build tools, audio units, and audio unit host applications. As with the projects in the `AudioUnits` folder, you can make use of these projects during audio unit development in a variety of ways:

- Directly using the the tools, audio units, and host applications built by these projects
- Studying the source code to gain insight into how Core Audio works, as well as insight into how to design and implement tools, audio units, and host applications
- Using the projects as starting points for your own tools, audio units, and host applications

The Services folder contains the following Xcode projects:

AudioFileTools

A project that builds a set of eight command-line tools for playing, recording, examining, and manipulating audio files.

AudioUnitHosting

A project that builds a Carbon-based audio unit host application. Useful in terms of sample code for host developers but deprecated as a host for testing your audio units. (For audio unit testing, use AU Lab.)

AUMixer3DTest

A project that builds an application that uses a 3D mixer audio unit.

AUViewTest

A project that builds a windowless application based on an audio processing graph. The graph includes an instrument unit, an effect unit, and an output unit. There's a menu item to open a view for the effect unit. The AUViewTest application uses a music player object to play a repeating sequence through the instrument unit.

CocoaAUHost

A project that builds a Cocoa-based audio unit host application. This project is useful in terms of sample code for host developers but deprecated as a host for testing your audio units. (For audio unit testing, use AU Lab.)

MatrixMixerTest

A project that builds an example mixer unit with a custom Cocoa view.

OpenALExample

A project that builds an application based on the OpenAL API with Apple extensions. The application demonstrates control of listener position and orientation within a two-dimensional layout of audio sources.

Tutorial: Building a Simple Effect Unit with a Generic View

This tutorial chapter leads you through the complete design and development of an effect unit with a generic view. As you work through this tutorial, you take a big step in learning how to develop your own audio units.

The format of this chapter lets you complete it successfully without having read any of the preceding material. However, to understand what you’re doing, you need to understand the information in [“Audio Unit Development Fundamentals”](#) (page 13) and [“The Audio Unit”](#) (page 43).

In particular, to gain the most from this chapter, you should understand:

- Properties, parameters, and factory presets
- Audio unit types, subtypes, manufacturer IDs, and version numbers
- Audio unit life cycle including instantiation, initialization, and rendering
- The role of the Core Audio SDK in audio unit development
- The role of the AU Lab application and the `auval` tool in audio unit development

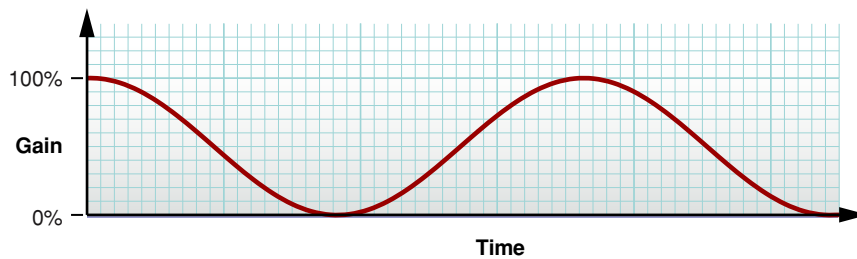
Overview

“Simple” effect units operate on individual audio samples without considering one audio sample’s relationship to another, and without using a processing buffer. You build such an effect unit in this chapter, further simplified by leaving out some advanced audio unit features: a custom view and custom properties.

Simple effect units can do very basic DSP, such as:

- Change level
- Add tremolo

Monaural tremolo is a continuous wavering produced by varying an audio channel’s gain at low frequency, on the order of a few Hertz. This figure illustrates monaural tremolo that varies from full gain to silence:

Figure 5-1 Monaural tremolo

Stereo tremolo is similar but involves continuous left/right panning at a low frequency.

In this chapter you design and build a monaural tremolo effect that uses the generic view. The steps, described in detail in the following sections, are:

1. Install the Core Audio development kit if you haven't already done so.
2. Perform some design work, including:
 - Specify the sort of DSP your audio unit will perform
 - Design the parameter interface
 - Design the factory preset interface
 - Determine the configuration information for your audio unit bundle, such as the subtype and manufacturer codes, and the version number
3. Create and configure the Xcode project.
4. Implement your audio unit:
 - Implement the parameter and preset interfaces.
 - Implement signal processing—the heart of your audio unit.
5. Finally, validate and test your audio unit.

Your development environment for building any audio unit—simple or otherwise—should include the pieces described under “[Required Tools for Audio Unit Development](#)” (page 11) in the “[Introduction](#)” (page 9). You do not need to refer to the Xcode documentation or the Core Audio SDK documentation to complete the tasks in this chapter.

Install the Core Audio Development Kit

1. If you haven't already done so, install the most recent Core Audio SDK. It is part of the Xcode Tools installation on the Mac OS X DVD. You can also download it from Apple's developer website at this location:

<http://developer.apple.com/sdk/>

The Core Audio SDK installer places C++ superclasses, example Xcode projects, and documentation at this location on your system:

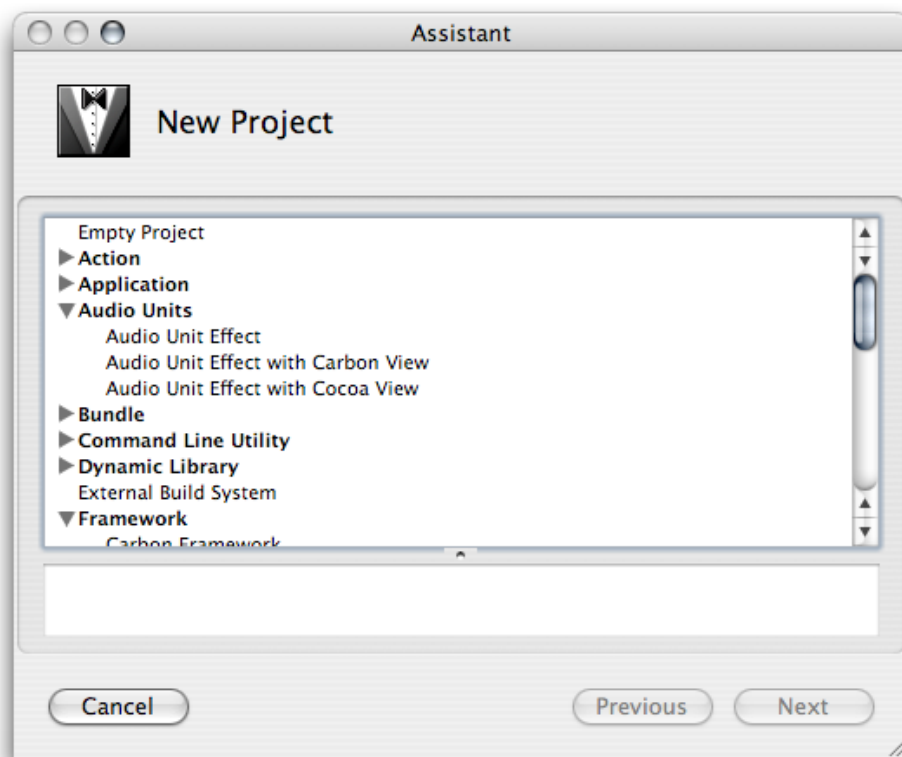
```
/Developer/Examples/CoreAudio
```

The SDK also installs Xcode project templates for audio units at this location on your system:

```
/Library/Application Support/Apple/Developer Tools/Project Templates/
```

2. After you have installed the SDK, confirm that Xcode recognizes the audio unit templates, as follows:
 - Launch Xcode and then choose File > New Project.
 - In the Assistant window, confirm that there is an Audio Units group.

Figure 5-2 Confirming that the audio unit templates are installed



Having confirmed the presence of the Audio Units template group, click Cancel, and then quit Xcode.

Note: If there is no Audio Units template group, check whether the SDK files were indeed installed as expected. If not, try installing again. Also make sure you are using the newest version of Xcode.

Specify the Function of Your Audio Unit

With tools in place, you begin your development of an effect unit by describing its digital signal processing. In this case, you specify an audio unit that provides monaural tremolo at a user selectable rate. You'll implement the DSP later, but establishing a bird's eye view up front lets you narrow down the implementation steps.

Next, pick an audio unit type based on the function the audio unit will perform. Look through the various types of standard audio units in the `AUComponent.h` file in the Audio Unit framework. Also, refer to the descriptions of types in *Audio Unit Specification*. The `kAudioUnitType_Effect`, with four-character code of `'aufx'`, looks appropriate for this purpose.

Design the Parameter Interface

In this section you work out the parameter portion of the programmatic interface between your audio unit and its view. End users will vary these parameters to control your audio unit in real time.

First, specify which sound attributes you'd like the user to be able to control. For this project, use the following:

- Tremolo frequency—the number of tremolo cycles per second.
- Tremolo depth—the mix between the input signal and the signal with tremolo applied. At a depth of 0%, there is no tremolo effect. At a depth of 100%, the effect ranges from full amplitude to silence.
- Tremolo waveform—the shape of the tremolo effect, such as sine wave or square wave.

Second, specify for each parameter:

- *User interface name* as it appears in the audio unit view
- *Programmatic name*, also called the parameter ID, used in the `GetParameterInfo` method in the audio unit implementation file
- *Unit of measurement*, such as gain, decibels, or hertz
- *Minimum value*
- *Maximum value*
- *Default value*

The following tables specify the parameter design. You'll use most of these values directly in your code. Specifying a "Description" is for your benefit while developing and extending the audio unit. You can reuse the description later in online help or a user guide.

Table 5-1 Specification of tremolo frequency parameter

Parameter attribute	Value
User interface name	Frequency

Parameter attribute	Value
Description	The frequency of the tremolo effect. When this parameter is set to 2 Hz, there are two cycles of the tremolo effect per second. This parameter's value is continuous, so the user can set any value within the available range. The user adjusts tremolo frequency with a slider.
Programmatic name	kTremolo_Frequency
Unit of measurement	Hz
Minimum value	0.5
Maximum value	10.0
Default value	2.0

Table 5-2 Specification of tremolo depth parameter

Parameter attribute	Value
User interface name	Depth
Description	The depth, or intensity, of the tremolo effect. When set to 0%, there is no tremolo effect. When set to 100%, the tremolo effect ranges over each cycle from silence to unity gain. This parameter's value is continuous, so the user can set any value within the available range. The user adjusts tremolo depth with a slider.
Programmatic name	kTremolo_Depth
Unit of measurement	Percent
Minimum value	0.0
Maximum value	100.0
Default value	50.0

Table 5-3 Specification of tremolo waveform parameter

Parameter attribute	Value
User interface name	Waveform
Description	The waveform that the tremolo effect follows. This parameter can take on a set of discrete values. The user picks a tremolo waveform from a menu.
Programmatic name	kTremolo_Waveform
Unit of measurement	Indexed
one value	Sine

Parameter attribute	Value
another value	Square
Default value	Sine

It's easy to add, delete, and refine parameters later. For example, if you were creating an audio level adjustment parameter, you might start with a linear scale and later change to a logarithmic scale. For the tremolo unit you're building, you might later decide to add additional tremolo waveforms.

Design the Factory Presets

Now that you've specified the parameters, you can specify interesting combinations of settings, or factory presets. For the tremolo unit, specify two factory presets. These two, invented for this tutorial, provide two easily distinguishable effects:

Table 5-4 Specification of Slow & Gentle factory preset

Parameter	Value
User interface name	Slow & Gentle
Description	A gentle waver
kTremolo_Frequency value	2.0 Hz
kTremolo_Depth value	50%
kTremolo_Waveform value	Sine

Table 5-5 Specification of Fast & Hard factory preset

Parameter	Value
User interface name	Fast & Hard
Description	Frenetic, percussive, and intense
kTremolo_Frequency value	20.0 Hz
kTremolo_Depth value	90%
kTremolo_Waveform value	Square

Collect Configuration Information for the Audio Unit Bundle

Now determine your audio unit bundle's configuration information, such as name and version number. You enter this information into your project's source files, as described later in [“Create and Configure the Xcode Project”](#) (page 95). Here's what you determine and collect up front:

- *Audio unit bundle English name*, such as Tremolo Unit. Host applications will display this name to let users pick your audio unit.
- *Audio unit programmatic name*, such as TremoloUnit. You'll use this to name your Xcode project, which in turn uses this name for your audio unit's main class.
- *Audio unit bundle version number*, such as 1.0.2.
- *Audio unit bundle brief description*, such as "Tremolo Unit version 1.0, copyright © 2006, Angry Audio." This description appears in the Version field of the audio unit bundle's Get Info window in the Finder.
- *Audio unit bundle subtype*, a four-character code that provides some indication (to a person) of what your audio unit does. For this audio unit, use 'tmlo'.
- *Company English name*, such as Angry Audio. This string appears in the audio unit generic view.
- *Company programmatic name*, such as angryaudio. You'll use this string in the reverse domain name-style identifier for your audio unit.
- *Company four-character code*, such as 'Aaud'.

You obtain this code from Apple, as a “creator code,” by using the [Data Type Registration](#) page. For the purposes of this example, for our fictitious company named Angry Audio, we'll use 'Aaud'.

- *Reverse domain name identifier*, such as com.angryaudio.audiounit.TremoloUnit. The Component Manager will use this name when identifying your audio unit.
- *Custom icon*, for display in the Cocoa generic view—optional but a nice touch. This should be a so-called “Small” icon as built with Apple's Icon Composer application, available on your system in the /Developer/Applications/Utilities folder.

Set Your Company Name in Xcode

Set your company name in Xcode (with Xcode not running) if you haven't already done so. Enter the following command in Terminal on one line:

```
defaults write com.apple.Xcode PBXCustomTemplateMacroDefinitions
'{"ORGANIZATIONNAME" = "Angry Audio";}'
```

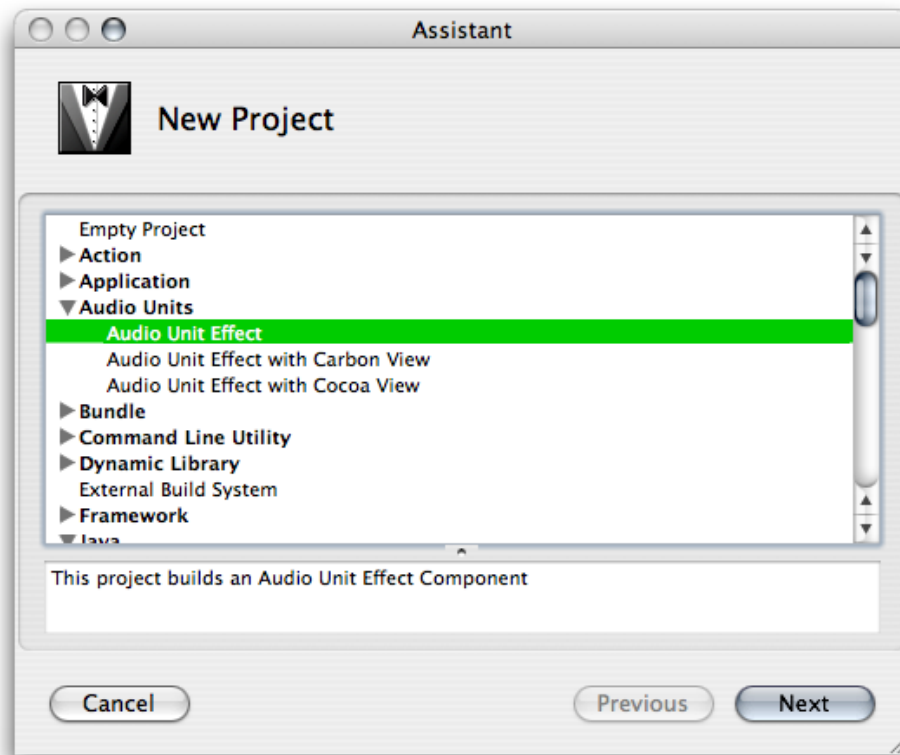
Now that you have set this so called “expert” preference, Xcode places your company's name in the copyright notice in the source files of any new project.

Create and Configure the Project

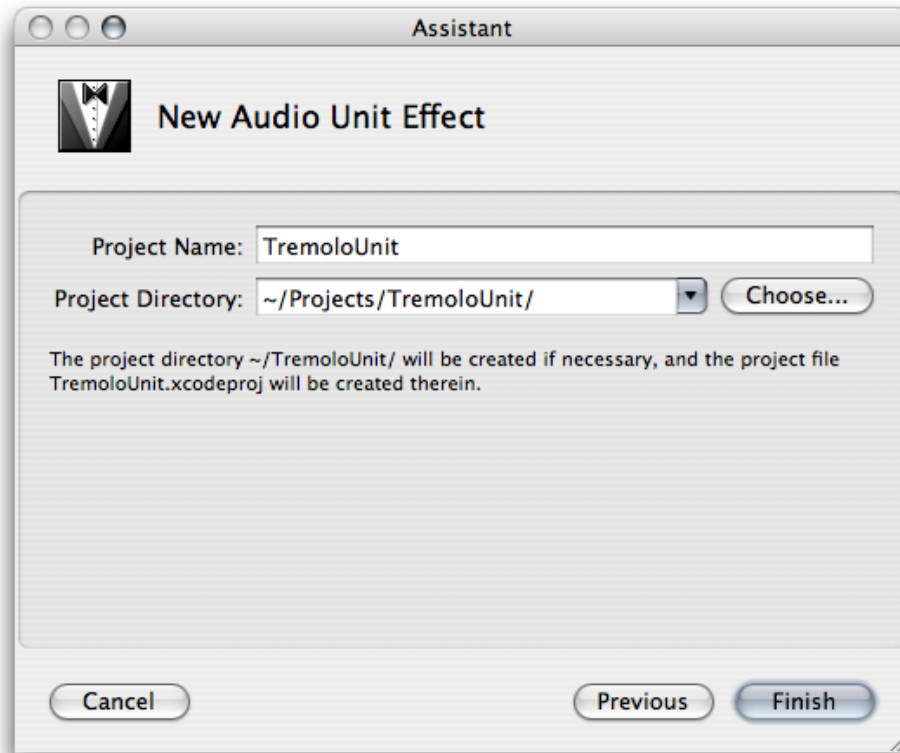
Now you create and configure an Xcode project for your audio unit. This section may seem long—it includes many steps and screenshots—but after you are familiar with the configuration process you can accomplish it in about five minutes.

Launch Xcode and follow these steps:

1. Choose File > New Project
2. In the New Project Assistant dialog, choose the Audio Unit Effect template and click Next.

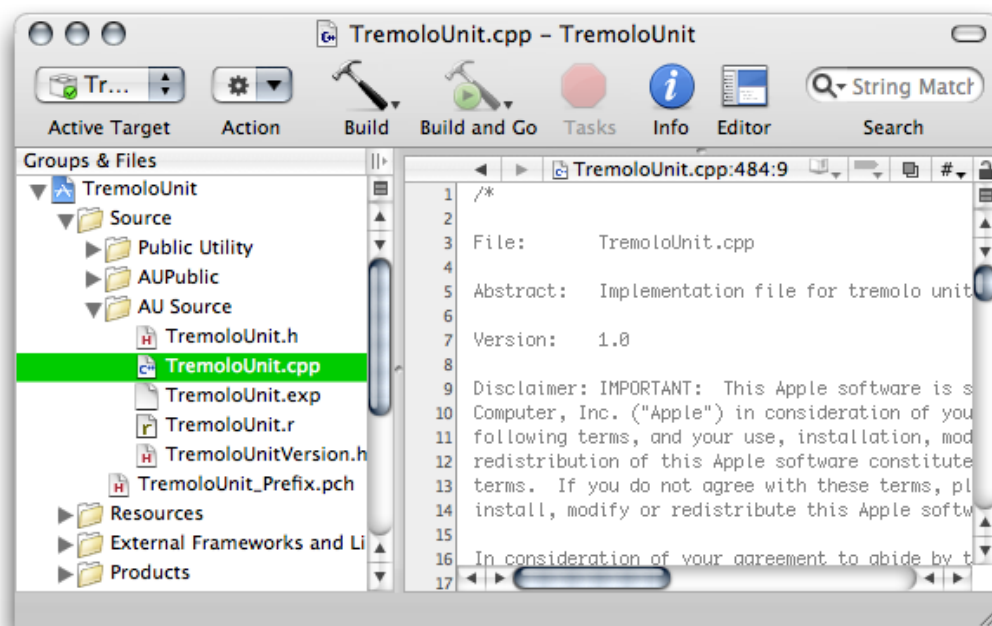


3. Name the project TremoloUnit and specify a project directory. Then click Finish.



Xcode creates the project files for your audio unit and the Xcode project window opens.

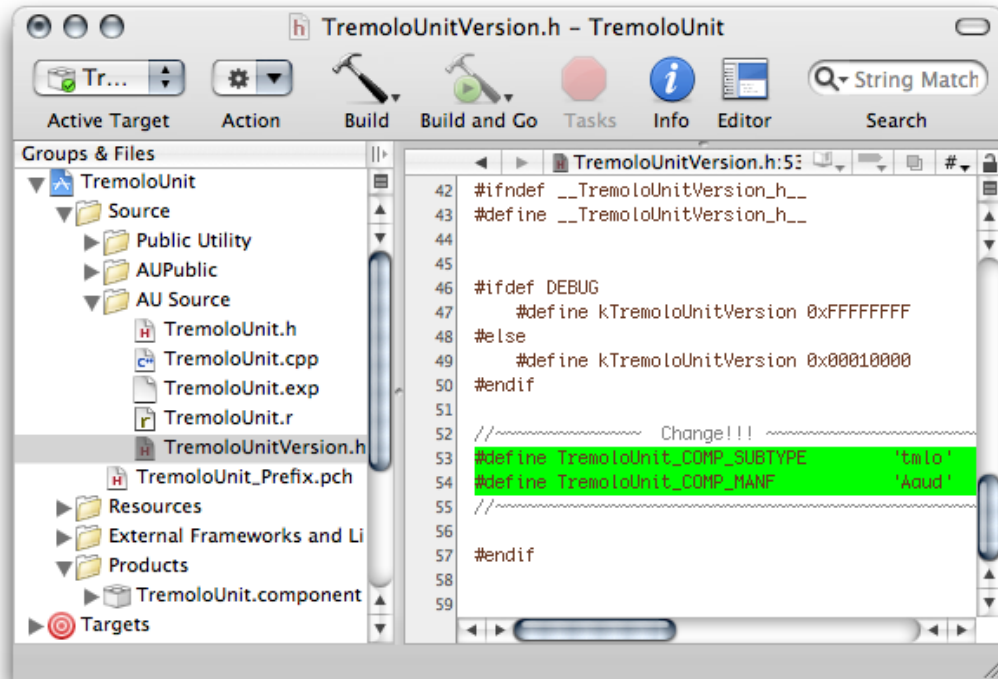
At this point, Xcode has used the audio unit project template file to create a subclass of the `AUEffectBase` class. Your custom subclass is named according to the name of your project. You can find your custom subclass's implementation file, `TremoloUnit.cpp`, in the AU Source group in the Xcode Groups & Files pane, shown next.



In later steps, you'll edit methods in your custom subclass to override the superclass's methods. `TremoloUnit.cpp` also contains a couple of methods from the `AUKernelBase` helper class; these are the methods that you later modify to perform digital signal processing.

4. With the AU Source group open as shown in the previous step, click `TremoloUnitVersion.h`. Then click the Editor toolbar button, if necessary, to display the text editor. There are three values to customize in this file: `kTremoloUnitVersion`, `TremoloUnit_COMP_SUBTYPE`, and `TremoloUnit_COMP_MANF`.

Scroll down in the editing pane to the definitions of `TremoloUnit_COMP_SUBTYPE` and `TremoloUnit_COMP_MANF`. Customize the subtype field with the four-character subtype code that you've chosen. In this example, 'tmlo' indicates (to developers and users) that the audio unit lets a user add tremolo.



Also customize the manufacturer name with the unique four-character string that identifies your company.

Note: There is no `#define` statement for component type in the `TremoloUnitVersion.h` file because you specified the type—effect unit—when you picked the Xcode template for the audio unit. The audio unit bundle type is specified in the `AU Source/TremoloUnit.r` resource file.

Now set the version number for the audio unit. In the `TremoloUnitVersion.h` file, just above the definitions for subtype and manufacturer, you'll see the definition statement for the `kTremoloUnitVersion` constant. By default, the template sets this constant's value to 1.0.0, as represented by the hexadecimal number `0x00010000`. Change this, if you like. See [“Audio Unit Identification”](#) (page 29) for how to construct the hexadecimal version number.

Save the `TremoloUnitVersion.h` file.

5. Click the `TremoloUnit.r` resource file in the "Source/AU Source" group in the Groups & Files pane. There are two values to customize in this file: NAME and DESCRIPTION.

- NAME is used by the generic view to display both your company name and the audio unit name
- DESCRIPTION serves as the menu item for users choosing your audio unit from within a host application.

To work correctly with the generic view, the value for `NAME` must follow a specific pattern:

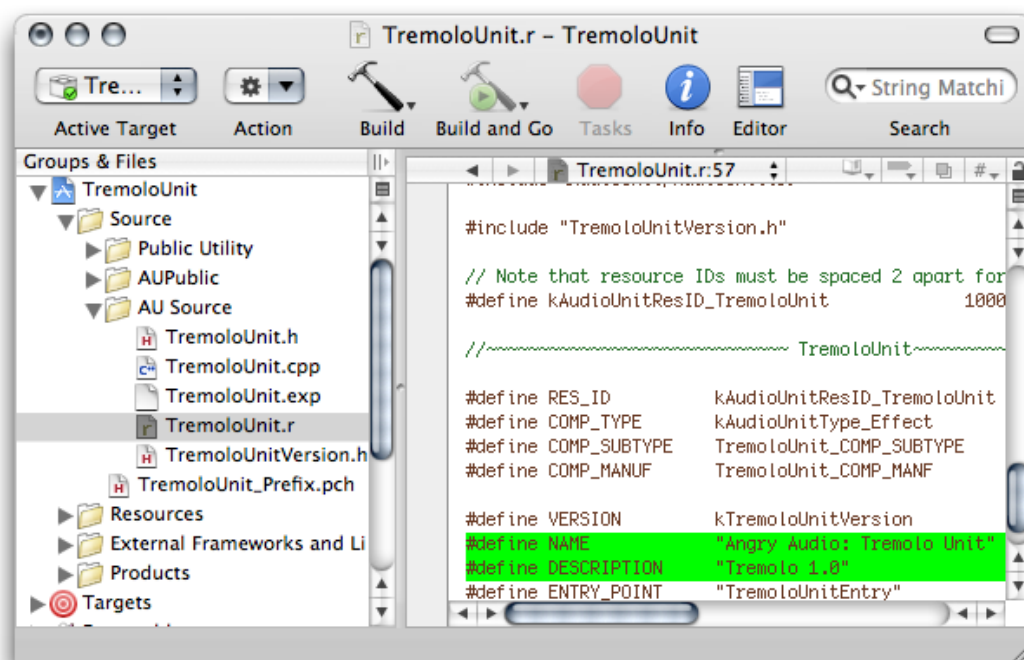
<company name>: <audio unit name>

For this example, use:

Angry Audio: Tremolo Unit

If you have set your company name using the Xcode expert preference as described earlier, it will already be in place in the `NAME` variable for this project; to follow this example, all you need to do is add a space between Tremolo and Unit in the audio unit name itself.

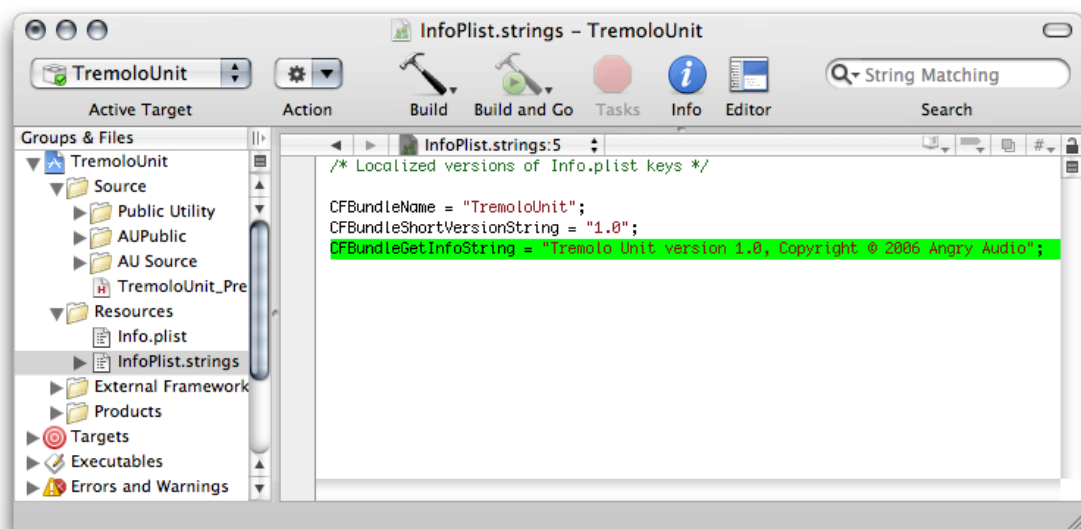
The Xcode template provides a default value for `DESCRIPTION`. If you customize it, keep it short so that the string works well with pop-up menus. The figure shows a customized `DESCRIPTION`.



As you can see in the figure, the resource file uses a `#include` statement to import the Version header file that you customized in step 4, `TremoloUnitVersion.h`. The resource file uses values from that header file to define some variables such as component subtype (`COMP_SUBTYPE`) and manufacturer (`COMP_MANUF`).

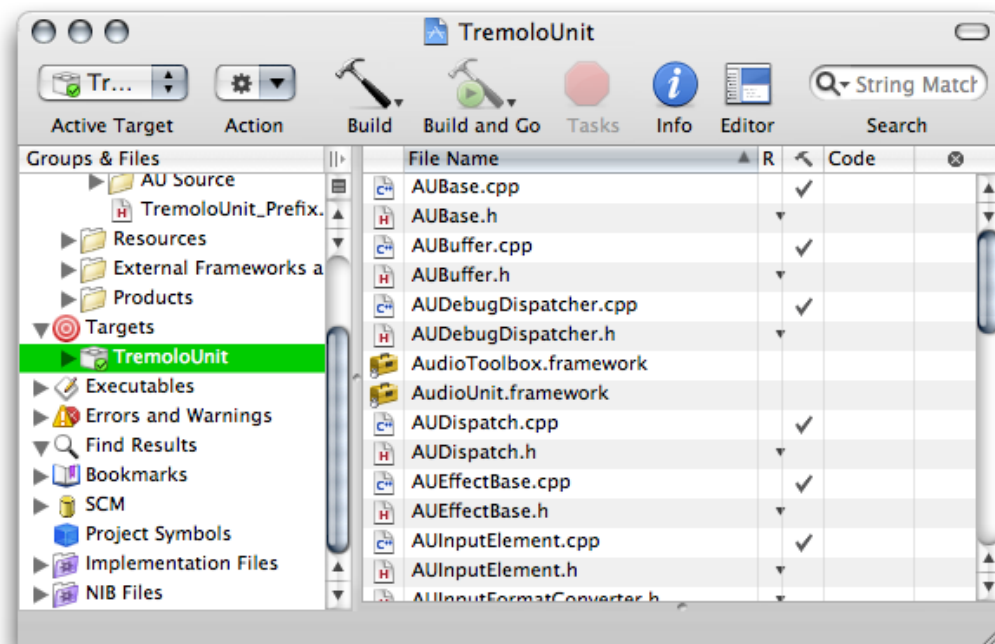
Save the `TremoloUnit.r` resource file.

6. Open the Resources group in the Groups & Files pane in the Xcode project window, and click on the InfoPlist.strings file.

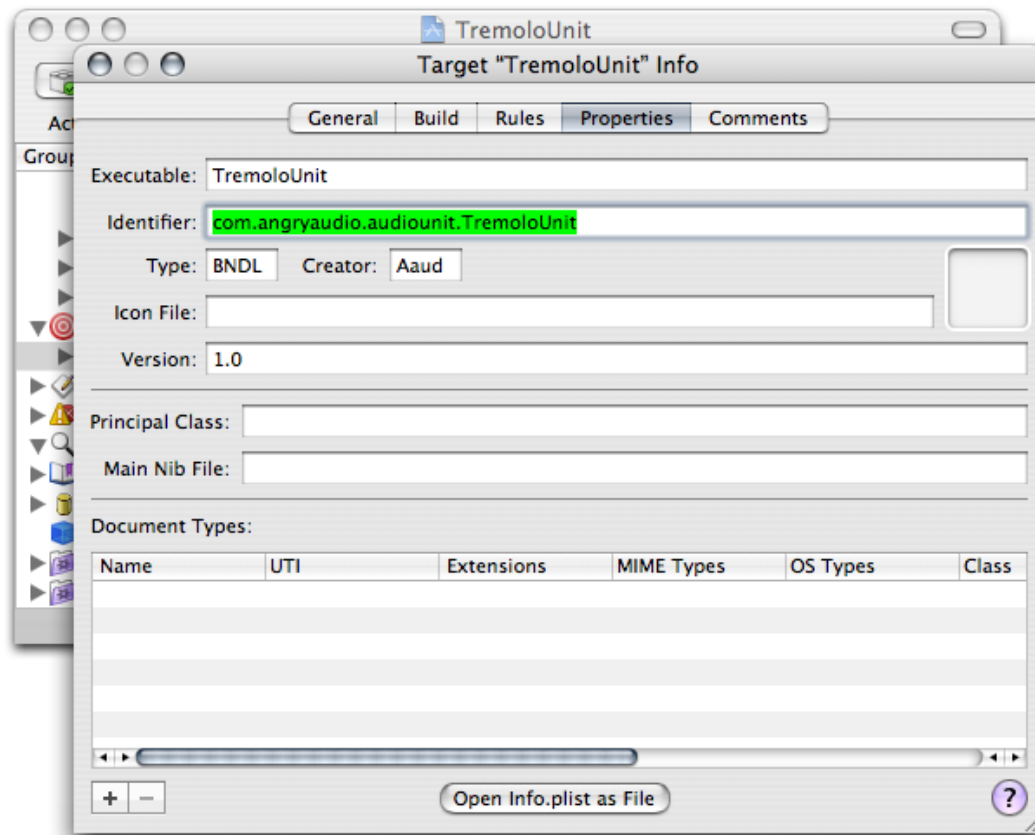


Using the editor, customize the value for `CFBundleGetInfoString` using the value you've chosen for the audio unit brief description. The figure provides an example. This string appears in the Version field of the audio unit bundle's Get Info window in the Finder. Save the `InfoPlist.strings` file.

7. Open the Targets group in the Groups & Files pane in the Xcode project window. Double-click the audio unit bundle, which has the same name as your project—in this case, TremoloUnit.



The Target Info window opens. Click the Properties tab.



In the Target Info window's Properties tab, provide values for Identifier, Creator, Version, and, optionally, a path to a Finder icon file for the bundle that you place in the bundle's Resources folder.

The audio unit bundle identifier field should follow the pattern:

```
com.<company_name>.audiounit.<audio_unit_name>
```

For this example, use the identifier:

```
com.angryaudio.audiounit.TremoloUnit
```

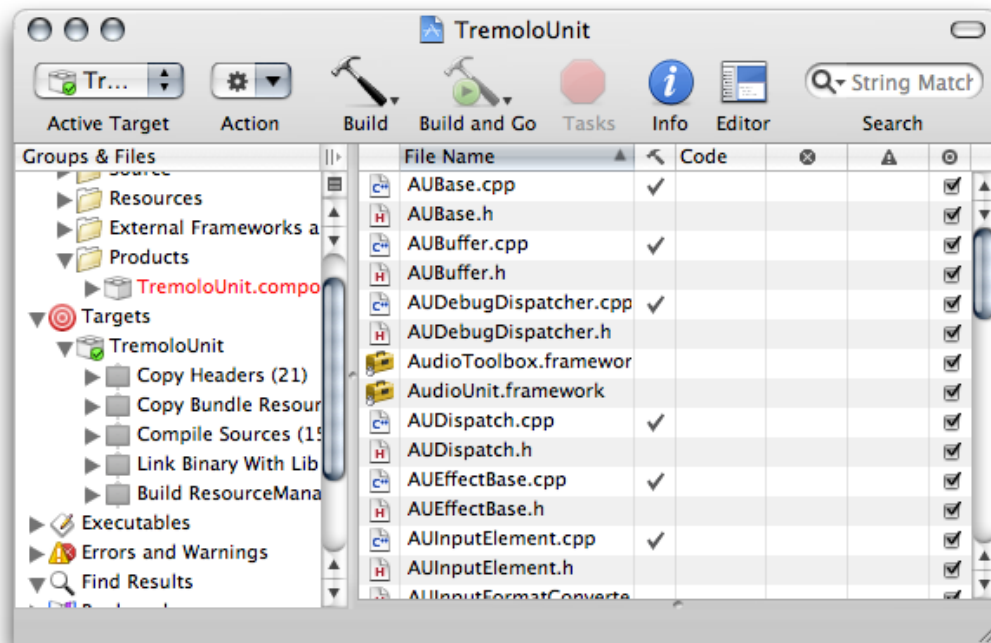
For the Creator value, use the same four-character string used for the manufacturer field in step 4.

Xcode transfers all the information from the Properties tab into the audio unit bundle's Info.plist file. You can open the Info.plist file, if you'd like to inspect it, directly from this dialog using the "Open Info.plist as File" button at the bottom of the window.

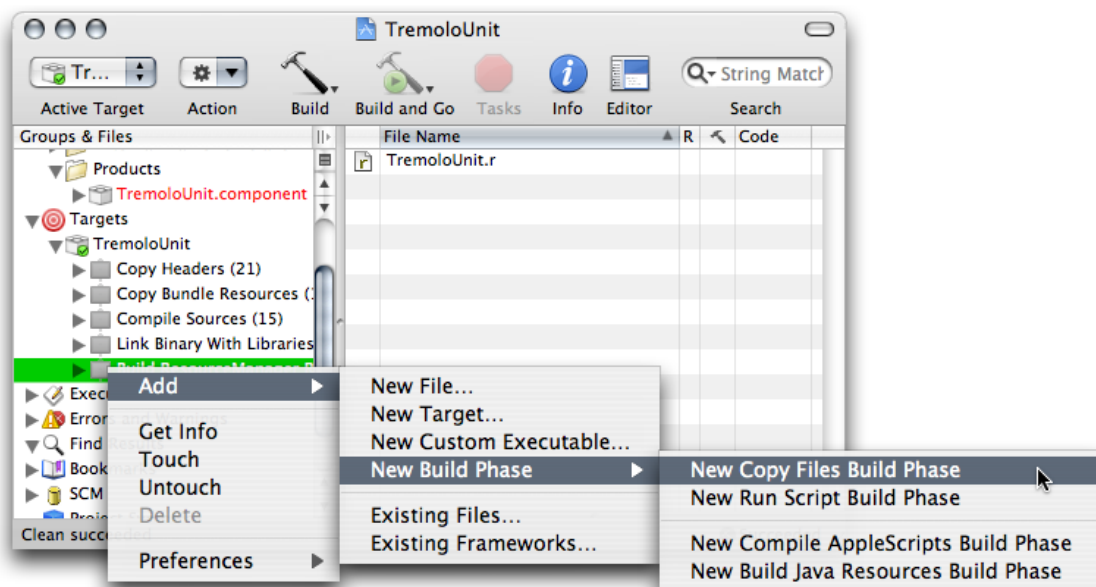
When finished, close the Info.plist file (if you've opened it) or close the Target Info window.

8. Now configure the Xcode project's build process to copy your audio unit bundle to the appropriate location so that host applications can use it.

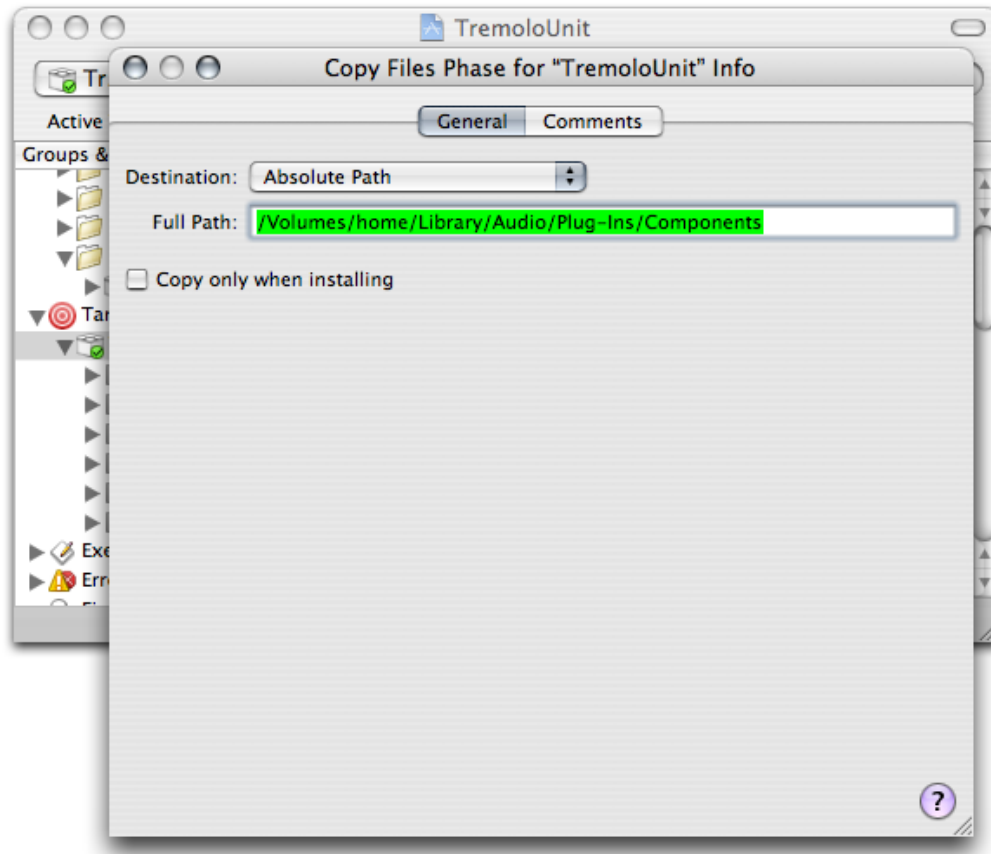
In the project window, disclose the Products group and the Targets group, as shown in the figure, so that you can see the icon for the audio unit bundle itself (`TremoloUnit.component`) as well as the build phases (under `Targets/TremoloUnit`).



9. Now add a new build phase. Right-click (or control-click) the final build phase for TremoloUnit and choose **Add > New Build Phase > New Copy Files Build Phase**.



The new Copy Files build phase appears at the end of the list, and a dialog opens, titled Copy Files Phase for "TremoloUnit" Info.



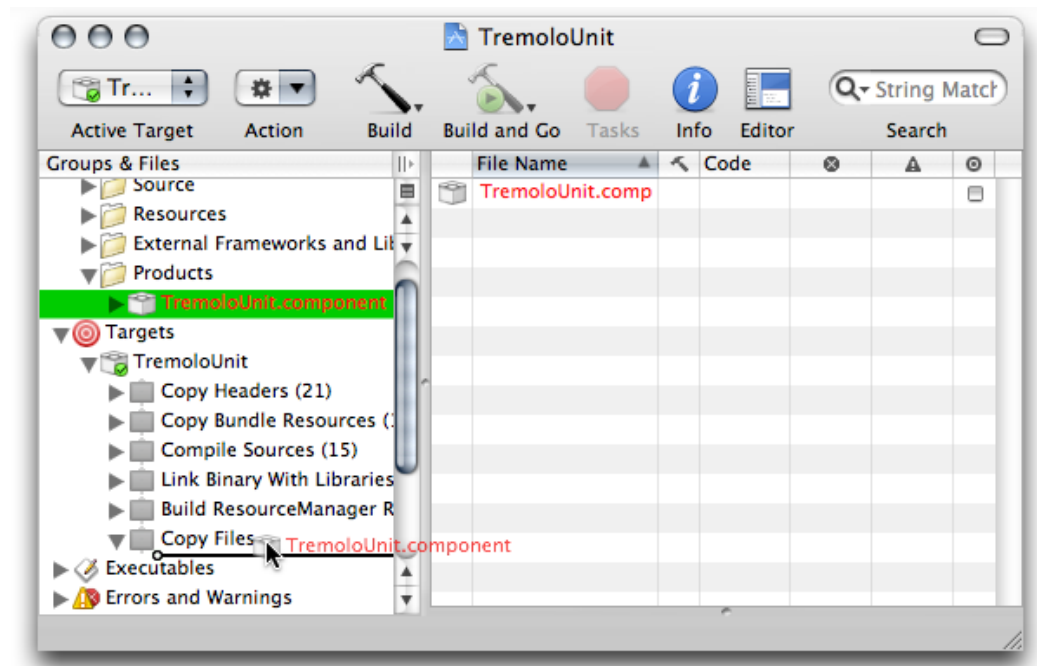
Change the Destination pop-up to Absolute Path, as shown in the figure.

Enter the absolute destination path for the built audio unit bundle in the Full Path field.

Note: The copy phase will not work if you enter a tilde (~) character to indicate your home folder. In Xcode 2.4, the Full Path field will let you enter a path by dragging the destination folder into the text field only if you first click in the Full Path field.

You can use either of the valid paths for audio unit bundles, as described in [“Audio Unit Installation and Registration”](#) (page 29). With the full path entered, close the dialog.

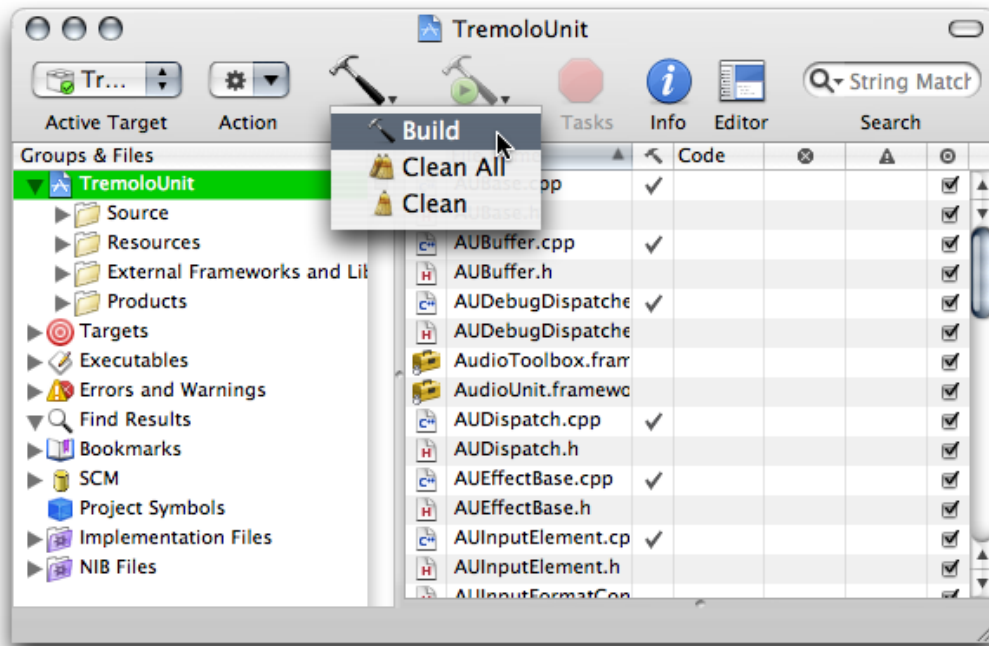
Now drag the `TremoloUnit.component` icon from the Products group to the new build phase.



You can later change the Copy Files location, if you want, by double clicking the gray Copy Files build phase icon. Alternatively, click the Copy Files icon and then click the Info button in the toolbar.

At this point, you have the makings for a working audio unit. You have not yet customized it to do whatever it is that you'll have it do (in our present case, to provide a single-channel tremolo effect). It's a good idea to ensure that you can build it without errors, that you can validate it with the `auval` tool, and that you can use it in a host application. Do this in the next step.

10. Build the project. You can do this in any of the standard ways: click the Build button in the toolbar, or choose Build from the Build button's menu, or choose Build > Build from the main menu, or type command-B.



If everything is in order, your project will build without error.

The copy files build phase that you added in the previous step ensures that a copy of the audio unit bundle gets placed in the appropriate location for the Component Manager to find it when a host application launches. The next step ensures that is so, and lets you test that it works in a host application.

Test the Unmodified Audio Unit

To test the newly built audio unit, use the AU Lab application:

- Use the `auval` tool to verify that Mac OS X recognizes your new audio unit bundle
- Launch the AU Lab application
- Configure AU Lab to test your new audio unit, and test it

Apple's Core Audio team provides AU Lab in the `/Developer/Applications/Audio` folder, along with documentation. You do not need to refer to AU Lab's documentation to complete this task. The `auval` command-line tool is part of a standard Mac OS X installation.

1. In Terminal, enter the command `auval -a`. If Mac OS X recognizes your new audio unit bundle, you see a listing similar to this one:

```

> auval -a

*****
AU Validation Tool
Version: 1.1.2c11
Copyright 2003-4, Apple Computer, Inc.

Specify -h (-help) for command options
*****

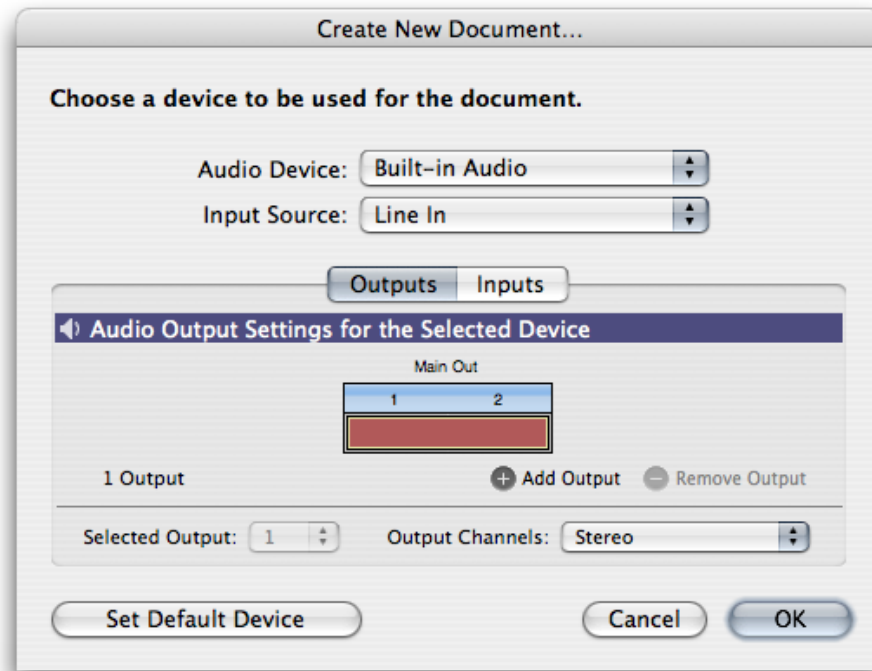
aufx: FILT: appl - Apple: Demo: Filter
aufx: bpas: appl - Apple: AUBandpass
aufx: dcmp: appl - Apple: AUDynamicsProcessor
aufx: dely: appl - Apple: AUDelay
aufx: filt: appl - Apple: AUFILTER
aufx: greg: appl - Apple: AUGraphicEQ
aufx: hpas: appl - Apple: AUHipass
aufx: hshf: appl - Apple: AUHighShelfFilter
aufx: lmtr: appl - Apple: AUPeakLimiter
aufx: lpas: appl - Apple: AULowpass
aufx: lshf: appl - Apple: AULowShelfFilter
aufx: mcmp: appl - Apple: AUMultibandCompressor
aufx: mrev: appl - Apple: AUMatrixReverb
aufx: nsnd: appl - Apple: AUNetSend
aufx: pmeq: appl - Apple: AUParametricEQ
aufx: sdly: appl - Apple: AUSampleDelay
aufx: tmlo: Aaud - Angry Audio: Tremolo Unit
aufx: tmpt: appl - Apple: AUPitch
aumu: dls: appl - Apple: DLSMusicDevice
aumx: 3dmx: appl - Apple: AUMixer3D
aumx: mxmx: appl - Apple: AUMatrixMixer
aumx: smxr: appl - Apple: AUMixer
aufc: conv: appl - Apple: AUConverter
aufc: defr: appl - Apple: AUDeferredRenderer
aufc: merg: appl - Apple: AUMerger
aufc: split: appl - Apple: AUSplitter
aufc: tmpt: appl - Apple: AUTimePitch
aufc: vari: appl - Apple: AUVarispeed
auou: ahal: appl - Apple: AudioDeviceOutput
auou: def: appl - Apple: DefaultOutputUnit
auou: genr: appl - Apple: GenericOutput
auou: sys: appl - Apple: SystemOutputUnit
augn: afpl: appl - Apple: AUAudioFilePlayer
augn: nrcv: appl - Apple: AUNetReceive
augn: sspl: appl - Apple: AUScheduledSoundPlayer

```

If your Xcode project builds without error, but you do not see the new audio unit bundle in the list reported by the `auval` tool, double check that you've entered the correct path in the Copy Files phase, as described in step 8 of [“Create and Configure the Xcode Project”](#) (page 95).

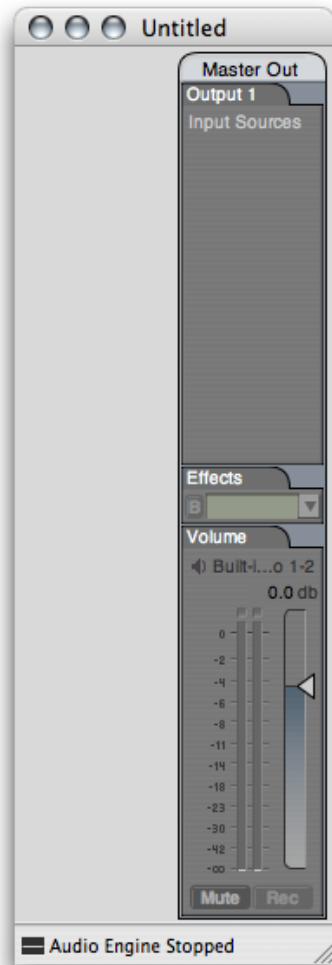
Note: The next few steps in this tutorial will be familiar to you if you went through [“Tutorial: Using an Audio Unit in a Host Application”](#) (page 20) in “Audio Unit Development Fundamentals”.

2. Launch AU Lab and create a new AU Lab document. Unless you've configured AU Lab to use a default document style, the Create New Document window opens. If AU Lab was already running, choose File > New to get this window.

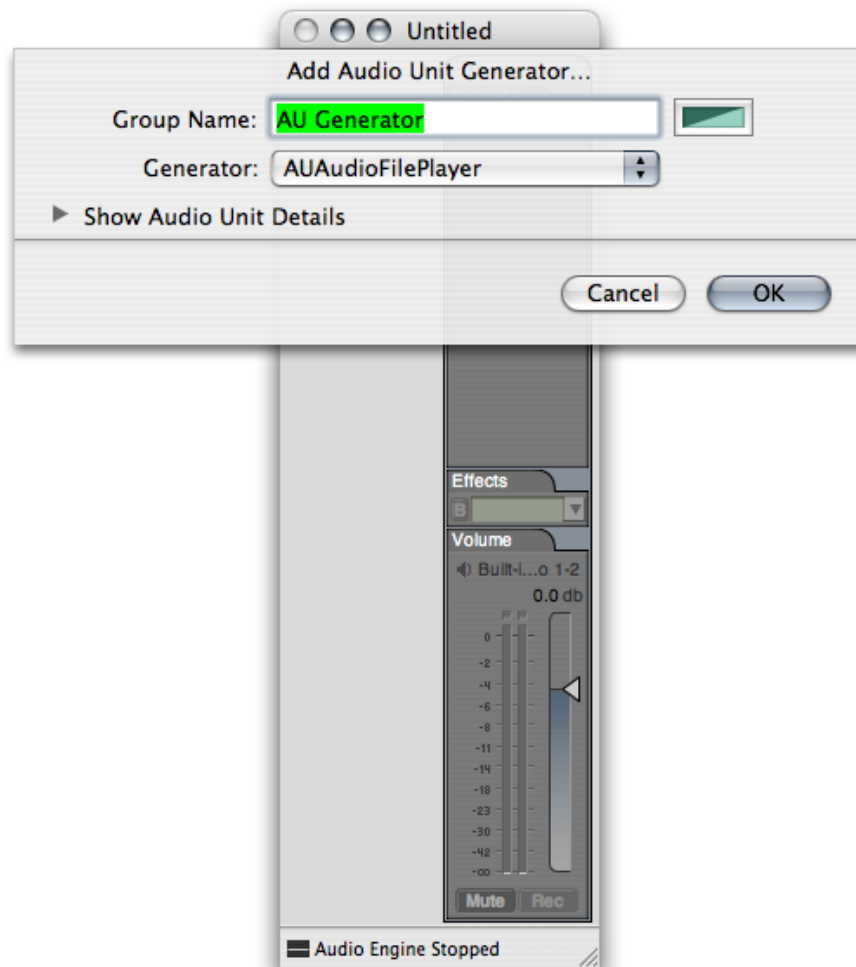


Ensure that the configuration matches the settings shown in the figure: Built-In Audio for the Audio Device, Line In for the Input Source, and Stereo for Output Channels. Leave the window's Inputs tab unconfigured; you will specify the input later. Click OK.

A new AU Lab window opens, showing the output channel you specified.



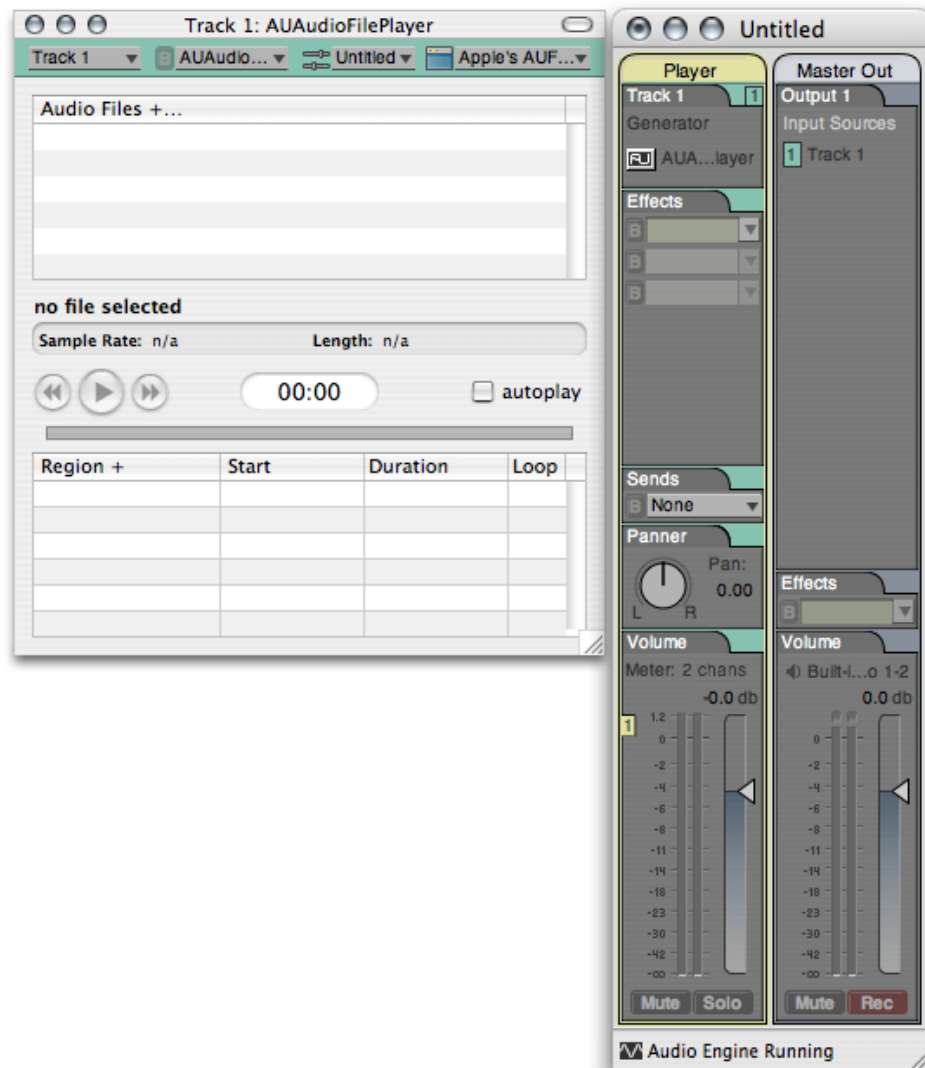
3. Choose Edit > Add Audio Unit Generator. A dialog opens from the AU Lab window to let you specify the generator unit to serve as the audio source.



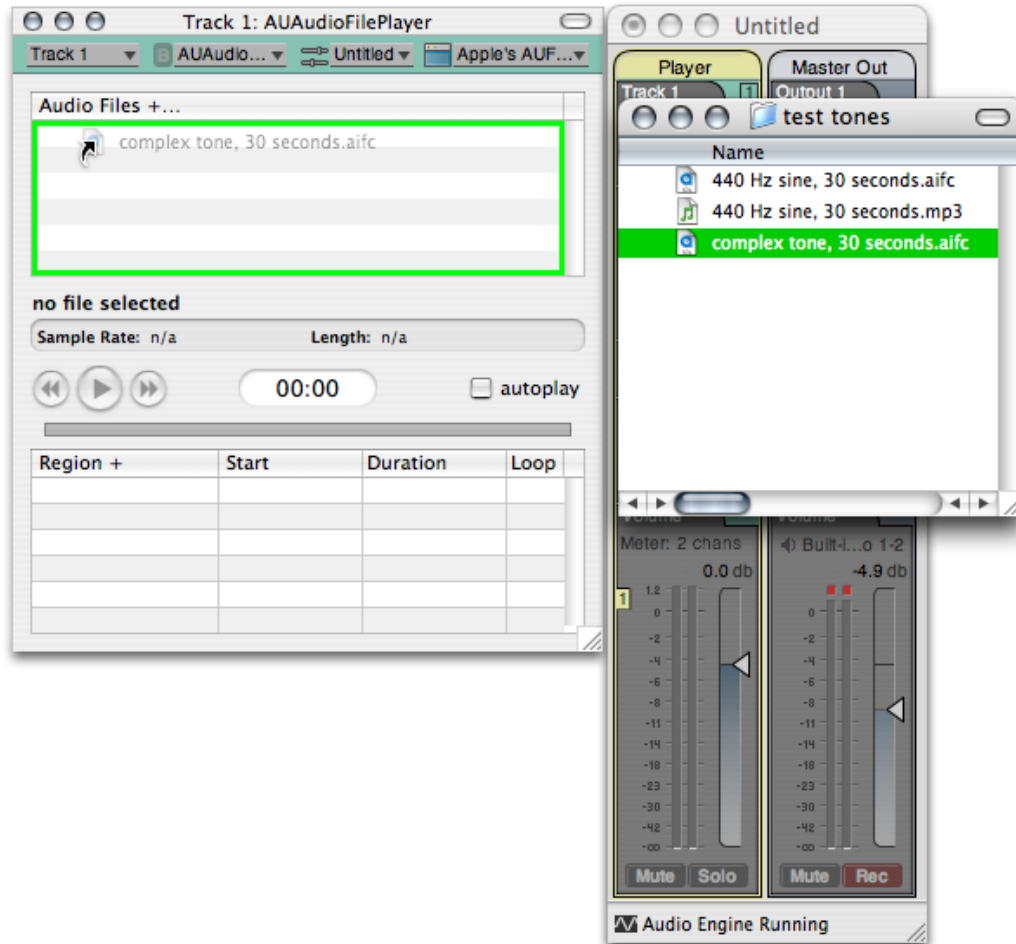
In the dialog, ensure that the AUAudioFilePlayer unit is selected in the Generator pop-up. To follow this example, change the Group Name to Player. Click OK.

Note: You can change the group name at any time by double-clicking it in the AU Lab window.

The AU Lab window now shows a stereo input track. In addition, an inspector window has opened for the player unit. If you close the inspector, you can reopen it by clicking the rectangular "AU" button near the top of the Player track.



4. Add an audio file to the Audio Files list in the player inspector window. Do this by dragging the audio file from the Finder, as shown. Putting an audio file in the player inspector window lets you send audio through the new audio unit. Just about any audio file will do, although a continuous tone is helpful for testing.



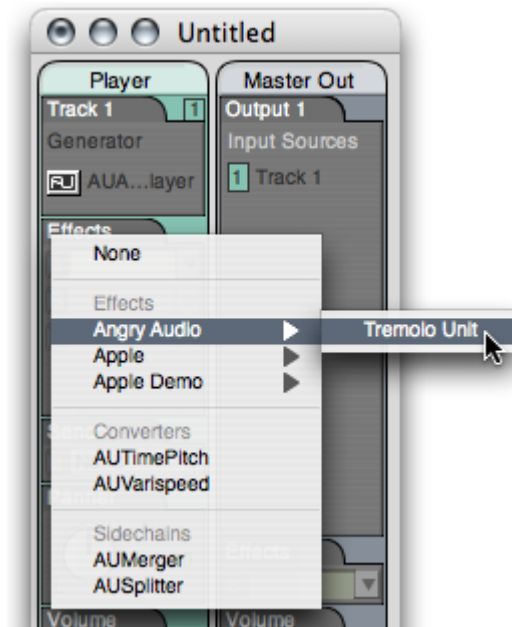
Now AU Lab is configured and ready to test your audio unit.

5. Click the triangular menu button in the first row of the Effects section in the Player track, as shown in the figure.

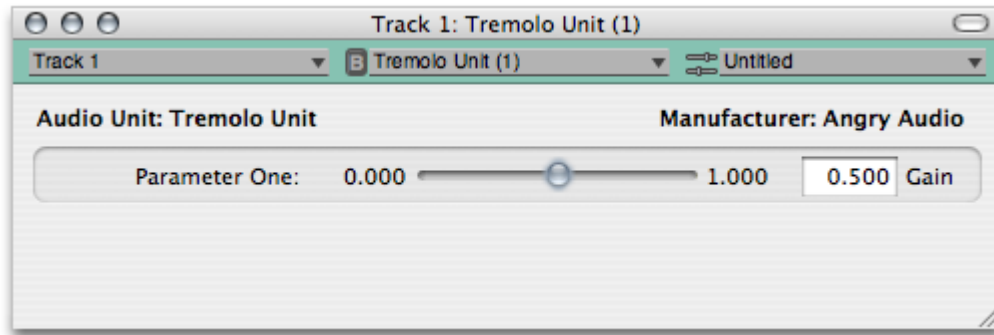


A menu opens, listing all the audio units available on your system, arranged by category and manufacturer. There is an Angry Audio group in the pop-up, as shown in the next figure.

Choose your new audio unit from the Effects first row pop-up.



AU Lab opens your audio unit's Cocoa generic view, which appears as a utility window.



The generic view displays your audio unit's interface as it comes directly from the parameter and property definitions supplied by the Xcode template. The template defines an audio unit that provides level adjustment. Its view, built by the AU Lab host application, features a Gain control. You modify the view in a later step in this task by changing your audio unit's parameter definitions.

Refer to [Table 3-1](#) (page 64) for information on where you define each user interface element for the generic view.

6. Click the Play button in the AUAudioFilePlayer inspector to send audio through the unmodified audio unit. This lets you ensure that audio indeed passes through the audio unit. Vary the slider in the generic view, as you listen to the audio, to ensure that the parameter is working.

7. Save the AU Lab document for use later, giving it a name such as "Tremolo Unit Test.trak". You will use it in the final section of this chapter, ["Test your Completed Audio Unit"](#) (page 135).

Next, you'll define your tremolo effect unit's parameter interface to give a user control over tremolo rate, depth, and waveform.

Implement the Parameter Interface

Up to this point in developing your audio unit, you have touched very little code. Here, you define the audio unit parameters by editing the source files for the audio unit custom subclass: `TremoloUnit.h` and `TremoloUnit.cpp`.

Note: There are many steps to perform in this and the remaining tasks in this chapter. It's a good idea to build your audio unit bundle after each section to check for errors and then correct them.

To define audio unit parameters you do three things:

- Name the parameters and give them values in the custom subclass's header file
- Add statements to the audio unit's constructor method to set up the parameters when the audio unit is instantiated
- Override the `GetParameterInfo` method from the SDK's `AUBase` class, in the custom subclass's implementation file, to define the parameters

The code you implement here does not make use of the parameters, per se. It is the DSP code that you implement later, in [“Implement Signal Processing”](#) (page 125), that makes use of the parameters. Here, you are simply defining the parameters so that they will appear in the audio unit generic view and so that they’re ready to use when you implement the DSP code.

Name the Parameters and Set Values

First, name your audio unit’s parameters and provide values for them. Do this by replacing the default parameter definition code in the `TremoloUnit.h` header file, provided by the Xcode template, with the following code. This listing implements the parameter design from the tables in [“Design the Parameter Interface”](#) (page 92).

Listing 5-1 Parameter names and values (`TremoloUnit.h`)

```
#pragma mark ____TremoloUnit Parameter Constants

static CFStringRef kParamName_Tremolo_Freq      = CFSTR ("Frequency");      // 1
static const float kDefaultValue_Tremolo_Freq  = 2.0;                       // 2
static const float kMinimumValue_Tremolo_Freq  = 0.5;                       // 3
static const float kMaximumValue_Tremolo_Freq  = 20.0;                      // 4

static CFStringRef kParamName_Tremolo_Depth     = CFSTR ("Depth");           // 5
static const float kDefaultValue_Tremolo_Depth = 50.0;
static const float kMinimumValue_Tremolo_Depth = 0.0;
static const float kMaximumValue_Tremolo_Depth = 100.0;

static CFStringRef kParamName_Tremolo_Waveform = CFSTR ("Waveform");        // 6
static const int kSineWave_Tremolo_Waveform    = 1;
static const int kSquareWave_Tremolo_Waveform  = 2;
static const int kDefaultValue_Tremolo_Waveform = kSineWave_Tremolo_Waveform;

// menu item names for the waveform parameter
static CFStringRef kMenuItem_Tremolo_Sine      = CFSTR ("Sine");             // 7
static CFStringRef kMenuItem_Tremolo_Square    = CFSTR ("Square");           // 8

// parameter identifiers
enum {                                         // 9
    kParameter_Frequency = 0,
    kParameter_Depth     = 1,
    kParameter_Waveform  = 2,
    kNumberOfParameters  = 3
};
```

Here’s how this code works:

1. Provides the user interface name for the Frequency (`kParamName_Tremolo_Freq`) parameter.
2. Defines a constant for the default value for the Frequency parameter for the tremolo unit, anticipating a unit of Hertz to be defined in the implementation file.
3. Defines a constant for the minimum value for the Frequency parameter.
4. Defines a constant for the maximum value for the Frequency parameter.

5. Provides a user interface name for the Depth (`kParamName_Tremolo_Depth`) parameter. The following three lines define constants for the default, minimum, and maximum values for the Depth parameter.
6. Provides a user interface name for the Waveform (`kParamName_Tremolo_Waveform`) parameter. The following three lines define constants for the minimum, maximum, and default values for the Waveform parameter.
7. Defines the menu item string for the sine wave option for the Waveform parameter.
8. Defines the menu item string for the square wave option for the Waveform parameter.
9. Defines constants for identifying the parameters; defines the total number of parameters.

For each parameter you've defined in the `TremoloUnit.h` file, your audio unit needs:

- A corresponding `SetParameter` statement in the constructor method, as described next in “Edit the Constructor Method”
- A corresponding parameter definition in the `GetParameterInfo` method, as described later in “Define the Parameters” (page 117)

Note: At this point, building your audio unit will fail because you have not yet edited the implementation file, `TremoloUnit.cpp`, to use the new parameter definitions. You work on the `TremoloUnit.cpp` file in the next few sections.

Edit the Constructor Method

Next, replace the custom subclass constructor method in the `TremoloUnit.cpp` file with the code in this section. This code instantiates the audio unit, which includes setting up the parameter names and values that you defined in the previous section.

For now, the `SetParameter` statements shown here are the only lines to customize in the constructor method. In a later step you'll add code to define the default factory preset.

Listing 5-2 Setting parameters in the constructor (`TremoloUnit.cpp`)

```
TremoloUnit::TremoloUnit (AudioUnit component) : AUEffectBase (component) {

    CreateElements ();
    Globals () -> UseIndexedParameters (kNumberOfParameters);

    SetParameter (                                // 1
        kParameter_Frequency,
        kDefaultValue_Tremolo_Freq
    );

    SetParameter (                                // 2
        kParameter_Depth,
        kDefaultValue_Tremolo_Depth
    );

    SetParameter (                                // 3
```

```

        kParameter_Waveform,
        kDefaultValue_Tremolo_Waveform
    );

    #if AU_DEBUG_DISPATCHER
        mDebugDispatcher = new AUDebugDispatcher (this);
    #endif
}

```

Here's how this code works:

1. Sets up the first parameter for the audio unit, based on values from the header file. In this project, this parameter controls tremolo frequency. The `SetParameter` method is inherited from superclasses in the Core Audio SDK.
2. Sets up the second parameter for the audio unit, based on values from the header file. In this project, this parameter controls tremolo depth.
3. Sets up the third parameter for the audio unit, based on values from the header file. In this project, this parameter controls the tremolo waveform.

Define the Parameters

Replace the default override of the `GetParameterInfo` method from the Xcode template, which defines just one default (Gain) parameter. Here, use the three-parameter design described earlier in [“Design the Parameter Interface”](#) (page 92).

Note: You may want to refer to [“Defining and Using Parameters”](#) (page 50) in “The Audio Unit” for a description of the role this method plays in the operation of an audio unit. You may want to refer to [“Audio Unit Scopes”](#) (page 44) for background on the Global scope as used in this method.

Listing 5-3 The customized `GetParameterInfo` method (`TremoloUnit.cpp`)

```

#pragma mark ____Parameters
ComponentResult TremoloUnit::GetParameterInfo (
    AudioUnitScope          inScope,
    AudioUnitParameterID    inParameterID,
    AudioUnitParameterInfo  &outParameterInfo
) {
    ComponentResult result = noErr;

    outParameterInfo.flags =      kAudioUnitParameterFlag_IsWritable          // 1
                                | kAudioUnitParameterFlag_IsReadable;

    if (inScope == kAudioUnitScope_Global) {                                // 2
        switch (inParameterID) {
            case kParameter_Frequency:                                       // 3
                AUBase::FillInParameterName (
                    outParameterInfo,
                    kParamName_Tremolo_Freq,
                    false
                );
                outParameterInfo.unit =                                       // 4

```

```

        kAudioUnitParameterUnit_Hertz;
    outParameterInfo.minValue = // 5
        kMinimumValue_Tremolo_Freq;
    outParameterInfo.maxValue = // 6
        kMaximumValue_Tremolo_Freq;
    outParameterInfo.defaultValue = // 7
        kDefaultValue_Tremolo_Freq;
    outParameterInfo.flags // 8
        |= kAudioUnitParameterFlag_DisplayLogarithmic;
    break;

case kParameter_Depth: // 9
    AUBase::FillInParameterName (
        outParameterInfo,
        kParamName_Tremolo_Depth,
        false
    );
    outParameterInfo.unit = // 10
        kAudioUnitParameterUnit_Percent;
    outParameterInfo.minValue =
        kMinimumValue_Tremolo_Depth;
    outParameterInfo.maxValue =
        kMaximumValue_Tremolo_Depth;
    outParameterInfo.defaultValue =
        kDefaultValue_Tremolo_Depth;
    break;

case kParameter_Waveform: // 11
    AUBase::FillInParameterName (
        outParameterInfo,
        kParamName_Tremolo_Waveform,
        false
    );
    outParameterInfo.unit = // 12
        kAudioUnitParameterUnit_Indexed;
    outParameterInfo.minValue =
        kSineWave_Tremolo_Waveform;
    outParameterInfo.maxValue =
        kSquareWave_Tremolo_Waveform;
    outParameterInfo.defaultValue =
        kDefaultValue_Tremolo_Waveform;
    break;

default:
    result = kAudioUnitErr_InvalidParameter;
    break;
}
} else {
    result = kAudioUnitErr_InvalidParameter;
}
return result;
}

```

Here's how this code works:

1. Adds two flags to all parameters for the audio unit, indicating to the host application that it should consider all the audio unit's parameters to be readable and writable.

2. All three parameters for this audio unit are in the “global” scope.
3. The first case in the switch statement, invoked when the view needs information for the `kTremolo_Frequency` parameter, defines how to represent this parameter in the user interface.
4. Sets the unit of measurement for the Frequency parameter to Hertz.
5. Sets the minimum value for the Frequency parameter.
6. Sets the maximum value for the Frequency parameter.
7. Sets the default value for the Frequency parameter
8. Adds a flag to indicate to the host that it should use a logarithmic control for the Frequency parameter.
9. The second case in the switch statement, invoked when the view needs information for the `kTremolo_Depth` parameter, defines how to represent this parameter in the user interface.
10. Sets the unit of measurement for the Depth parameter to percentage. The following three statements set the minimum, maximum, and default values for the Depth parameter.
11. The third case in the switch statement, invoked when the view needs information for the `kTremolo_Waveform` parameter, defines how to represent this parameter in the user interface.
12. Sets the unit of measurement for the Waveform parameter to “indexed,” allowing it to be displayed as a pop-up menu in the generic view. The following three statements set the minimum, maximum, and default values for the depth parameter. All three are required for proper functioning of the parameter’s user interface.

Provide Strings for the Waveform Pop-up Menu

Now you implement the `GetParameterValueStrings` method, which lets the audio unit’s generic view display the waveform parameter as a pop-up menu.

A convenient location for this code is after the `GetParameterInfo` method definition. If you add this method here as suggested, be sure to delete the placeholder method provided by the Xcode template elsewhere in the implementation file.

Listing 5-4 The customized `GetParameterValueStrings` method (`TremoloUnit.cpp`)

```
ComponentResult TremoloUnit::GetParameterValueStrings (
    AudioUnitScope          inScope,
    AudioUnitParameterID    inParameterID,
    CFArrayRef              *outStrings
) {
    if ((inScope == kAudioUnitScope_Global) &&                // 1
        (inParameterID == kParameter_Waveform)) {

        if (outStrings == NULL) return noErr;                  // 2

        CFStringRef strings [] = {                             // 3
            kMenuItem_Tremolo_Sine,
            kMenuItem_Tremolo_Square
        }
```

```

    };

    *outStrings = CFArrayCreate (                                // 4
        NULL,
        (const void **) strings,
        (sizeof (strings) / sizeof (strings [0])),             // 5
        NULL
    );
    return noErr;
}
return kAudioUnitErr_InvalidParameter;
}

```

Here's how this code works:

1. This method applies only to the waveform parameter, which is in the global scope.
2. When this method gets called by the `AUBase::DispatchGetPropertyInfo` method, which provides a null value for the *outStrings* parameter, just return without error.
3. Defines an array that contains the pop-up menu item names.
4. Creates a new immutable array containing the menu item names, and places the array in the *outStrings* output parameter.
5. Calculates the number of menu items in the array.

Implement the Factory Presets Interface

Next, you define the audio unit factory presets, again by editing the source files for the audio unit custom subclass, `TremoloUnit.h` and `TremoloUnit.cpp`.

The steps, described in detail below, are:

- Name the factory presets and give them values.
- Modify the `TremoloUnit` class declaration by adding method signatures for handling factory presets.
- Edit the audio unit's constructor method to set a default factory preset.
- Override the `GetPresets` method to set up a factory presets array.
- Override the `NewFactoryPresetSet` method to define the factory presets.

Note: The work you do here does not make use of the factory presets, per se. It is the DSP code that you implement in [“Implement Signal Processing”](#) (page 125) that makes use of the presets. Here, you are simply defining the presets so that they will appear in the audio unit generic view and so that they’re ready to use when you implement the DSP code.

Name the Factory Presets and Give Them Values

Define the constants for the factory presets. A convenient location for this code in the `TremoloUnit.h` header file is below the parameter constants section.

Listing 5-5 Factory preset names and values (`TremoloUnit.h`)

```
#pragma mark ____TremoloUnit Factory Preset Constants
static const float kParameter_Preset_Frequency_Slow = 2.0;    // 1
static const float kParameter_Preset_Frequency_Fast = 20.0;   // 2
static const float kParameter_Preset_Depth_Slow      = 50.0;   // 3
static const float kParameter_Preset_Depth_Fast      = 90.0;   // 4
static const float kParameter_Preset_Waveform_Slow   // 5
    = kSineWave_Tremolo_Waveform;
static const float kParameter_Preset_Waveform_Fast   // 6
    = kSquareWave_Tremolo_Waveform;
enum {
    kPreset_Slow      = 0,    // 7
    kPreset_Fast      = 1,    // 8
    kNumberPresets    = 2     // 9
};

static AUPreset kPresets [kNumberPresets] = {                // 10
    {kPreset_Slow, CFSTR ("Slow & Gentle")},
    {kPreset_Fast, CFSTR ("Fast & Hard")}
};

static const int kPreset_Default = kPreset_Slow;             // 11
```

Here’s how this code works:

1. Defines a constant for the frequency value for the “Slow & Gentle” factory preset.
2. Defines a constant for the frequency value for the “Fast & Hard” factory preset.
3. Defines a constant for the depth value for the “Slow & Gentle; Hard” factory preset.
4. Defines a constant for the depth value for the “Fast & Hard” factory preset.
5. Defines a constant for the waveform value for the “Slow & Gentle” factory preset.
6. Defines a constant for the waveform value for the “Fast & Hard” factory preset.
7. Defines a constant for the “Slow & Gentle” factory preset.
8. Defines a constant for the “Fast & Hard” factory preset.
9. Defines a constant representing the total number of factory presets.

10. Defines an array containing two Core Foundation string objects. The objects contain values for the menu items in the user interface corresponding to the factory presets.
11. Defines a constant representing the default factory preset, in this case the “Slow & Gentle” preset.

Add Method Declarations for Factory Presets

Now, provide method declarations for overriding the `GetPresets` and `NewFactoryPresetSet` methods from the `AUBase` superclass. Add these method declarations to the `public:` portion of class declaration in the `TremoloUnit.h` header file. You implement these methods in a later step in this chapter.

Listing 5-6 Factory preset method declarations (`TremoloUnit.h`)

```
#pragma mark ____TremoloUnit
class TremoloUnit : public AUEffectBase {

public:
    TremoloUnit (AudioUnit component);
    ...
    virtual ComponentResult GetPresets (          // 1
        CFArrayRef          *outData
    ) const;

    virtual OSStatus NewFactoryPresetSet (        // 2
        const AUPreset      &inNewFactoryPreset
    );
protected:
    ...
};
```

Here’s how this code works:

1. Declaration for the `GetPresets` method, overriding the method from the `AUBase` superclass.
2. Declaration for the `NewFactoryPresetSet` method, overriding the method from the `AUBase` superclass.

Set the Default Factory Preset

Now you return to the `TremoloUnit` constructor method, in which you previously added code for setting the audio unit parameters. Here, you add a single statement to set the default factory preset, making use of the `kTremoloPreset_Default` constant.

Listing 5-7 Setting the default factory preset in the constructor (`TremoloUnit.cpp`)

```
TremoloUnit::TremoloUnit (AudioUnit component) : AUEffectBase (component) {

    CreateElements ();
    Globals () -> UseIndexedParameters (kNumberOfParameters);
    // code for setting default values for the audio unit parameters
    SetAFactoryPresetAsCurrent (                // 1
```

```

        kPresets [kPreset_Default]
    );
    // boilerplate code for debug dispatcher
}

```

Here's how this code works:

1. Sets the default factory preset.

Implement the GetPresets Method

For users to be able to use the factory presets you define, you must add a generic implementation of the `GetPresets` method. The following generic code works for any audio unit that can support factory presets.

A convenient location for this code in the `TremoloUnit.cpp` implementation file is after the `GetPropertyInfo` and `GetProperty` methods.

Note: You can refer to “[Control Code: Parameters, Factory Presets, and Properties](#)” (page 50) for an architectural description of the `GetPresets` method, and how it fits into audio unit operation.

Listing 5-8 Implementing the `GetPresets` method (`TremoloUnit.cpp`)

```

#pragma mark ____Factory Presets
ComponentResult TremoloUnit::GetPresets (                // 1
    CFArrayRef *outData
) const {

    if (outData == NULL) return noErr;                    // 2

    CFMutableArrayRef presetsArray = CFArrayCreateMutable ( // 3
        NULL,
        kNumberPresets,
        NULL
    );

    for (int i = 0; i < kNumberPresets; ++i) {            // 4
        CFArrayAppendValue (
            presetsArray,
            &kPresets [i]
        );
    }

    *outData = (CFArrayRef) presetsArray;                // 5
    return noErr;
}

```

Here's how this code works:

1. The `GetPresets` method accepts a single parameter, a pointer to a `CFArrayRef` object. This object holds the factory presets array generated by this method.
2. Checks whether factory presets are implemented for this audio unit.

3. Instantiates a mutable Core Foundation array to hold the factory presets.
4. Fills the factory presets array with values from the definitions in the `TremoloUnit.h` file.
5. Stores the factory presets array at the `outData` location.

Define the Factory Presets

The `NewFactoryPresetSet` method defines all the factory presets for an audio unit. Basically, for each preset, it invokes a series of `SetParameter` calls.

A convenient location for this code in the `TremoloUnit.cpp` implementation file is after the implementation of the `GetPresets` method.

Listing 5-9 Defining factory presets in the `NewFactoryPresetSet` method (`TremoloUnit.cpp`)

```
OSStatus TremoloUnit::NewFactoryPresetSet (                // 1
    const AUPreset &inNewFactoryPreset
) {
    SInt32 chosenPreset = inNewFactoryPreset.presetNumber;    // 2

    if (                                                    // 3
        chosenPreset == kPreset_Slow ||
        chosenPreset == kPreset_Fast
    ) {
        for (int i = 0; i < kNumberPresets; ++i) {          // 4
            if (chosenPreset == kPresets[i].presetNumber) {
                switch (chosenPreset) {                      // 5

                    case kPreset_Slow:                       // 6
                        SetParameter (                      // 7
                            kParameter_Frequency,
                            kParameter_Preset_Frequency_Slow
                        );
                        SetParameter (                      // 8
                            kParameter_Depth,
                            kParameter_Preset_Depth_Slow
                        );
                        SetParameter (                      // 9
                            kParameter_Waveform,
                            kParameter_Preset_Waveform_Slow
                        );
                        break;

                    case kPreset_Fast:                       // 10
                        SetParameter (
                            kParameter_Frequency,
                            kParameter_Preset_Frequency_Fast
                        );
                        SetParameter (
                            kParameter_Depth,
                            kParameter_Preset_Depth_Fast
                        );
                        SetParameter (
                            kParameter_Waveform,
```

```

                                kParameter_Preset_Waveform_Fast
                                );
                                break;
                            }
                            SetAFactoryPresetAsCurrent (                // 11
                                kPresets [i]
                            );
                            return noErr;                               // 12
                        }
                    }
                }
                return kAudioUnitErr_InvalidProperty;                   // 13
            }
        }
    }
}

```

Here's how this code works:

1. This method takes a single argument of type `AUPreset`, a structure containing a factory preset name and number.
2. Gets the number of the desired factory preset.
3. Tests whether the desired factory preset is defined.
4. This `for` loop, and the `if` statement that follows it, allow for noncontiguous preset numbers.
5. Selects the appropriate case statement based on the factory preset number.
6. The settings for the “Slow & Gentle” factory preset.
7. Sets the Frequency audio unit parameter for the “Slow & Gentle” factory preset.
8. Sets the Depth audio unit parameter for the “Slow & Gentle” factory preset.
9. Sets the Waveform audio unit parameter for the “Slow & Gentle” factory preset.
10. The settings for the “Fast & Hard” factory preset. The three `SetParameter` statements that follow work the same way as for the other factory preset.
11. Updates the preset menu in the generic view to display the new factory preset.
12. On success, returns a value of `noErr`.
13. If the host application attempted to set an undefined factory preset, return an error.

Implement Signal Processing

With the parameter and preset code in place, you now get to the heart of the matter: the digital signal processing code. As described in [“Synthesis, Processing, and Data Format Conversion Code”](#) (page 55), the DSP performed by an *n*-to-*n* channel effect unit takes place in the `AUKernelBase` class, a helper class for the `AUEffectBase` class. Refer to [“Processing: The Heart of the Matter”](#) (page 36) for more on how DSP works in audio units.

To implement signal processing in an n -to- n channel effect unit, you override two methods from the `AUKernelBase` class:

- The `Process` method, which performs the signal processing
- The `Reset` method, which returns the audio unit to its pristine, initialized state

Along the way, you make changes to the default `TremoloUnitKernel` class declaration in the `TremoloUnit.h` header file, and you modify the `TremoloUnitKernel` constructor method in `TremoloUnit.cpp`.

DSP Design for the Tremolo Effect

The design and implementation of DSP code are central to real world audio unit development—but, as mentioned in the Introduction, they are outside the scope of this document. Nonetheless, this section describes the simple DSP used in this project to illustrate some of the issues involved with adding such code to an effect unit.

When you create a new audio unit project with an Xcode template, you get an audio unit with minimal DSP. It consists of only a multiplication that applies the value of a gain parameter to each sample of an audio signal. Here, you still use a simple multiplication—but with a bit more math up front, you end up with something a little more interesting; namely, tremolo. As you add the DSP code, you see where each part goes and how it fits in to the audio unit scaffolding.

The tremolo unit design uses a wave table to describe one cycle of a tremolo waveform. The `TremoloUnitKernel` class builds the wave table during instantiation. To make the project a bit more useful and instructive, the class builds not one but two wave tables: one representing a sine wave and one representing a pseudo square wave.

During processing, the tremolo unit uses sequential values from one of its wave tables as gain factors to apply to the audio signal, sample by sample. There's code to determine which point in the wave table to apply to a given audio sample.

Following the audio unit parameter design described earlier in this chapter, there is code to vary tremolo frequency, tremolo depth, and the tremolo waveform, all in real time.

Define Member Variables in the Kernel Class Declaration

To begin the DSP implementation for the tremolo unit, add some constants as private member variables to the `TremoloUnitKernel` class declaration. Defining these as private member variables ensures that they are global to the `TremoloUnitKernel` object and invisible elsewhere.

Note: As described in comment (1) below, the constructor signature you use here for the kernel helper class differs from the signature supplied in the Xcode template.

Listing 5-10 TremoloUnitKernel member variables (TremoloUnit.h)

```
class TremoloUnit : public AUEffectBase
{
public:
    TremoloUnit(AudioUnit component);
```

```

...

protected:
    class TremoloUnitKernel : public AUKernelBase {
    public:
        TremoloUnitKernel (AUEffectBase *inAudioUnit); // 1

        virtual void Process (
            const Float32      *inSourceP,
            Float32            *inDestP,
            UInt32              inFramesToProcess,
            UInt32              inNumChannels,    // equal to 1
            bool                &ioSilence
        );

        virtual void Reset();

    private:
        enum      {kWaveArraySize = 2000};           // 2
        float      mSine [kWaveArraySize];           // 3
        float      mSquare [kWaveArraySize];          // 4
        float      *waveArrayPointer;                 // 5
        Float32     mSampleFrequency;                 // 6
        long        mSamplesProcessed;                 // 7
        enum      {sampleLimit = (int) 10E6};          // 8
        float      mCurrentScale;                     // 9
        float      mNextScale;                        // 10
    };
};

```

Here's how this code works (skip this explanation if you're not interested in the math):

1. The constructor signature in the Xcode template contains a call to the superclass's constructor, as well as empty braces representing the method body. Remove all of these because you implement the constructor method in the implementation file, as described in the next section.
2. The number of points in each wave table. Each wave holds one cycle of a tremolo waveform.
3. The wave table for the tremolo sine wave.
4. The wave table for the tremolo pseudo square wave.
5. The wave table to apply to the current audio input buffer.
6. The sampling frequency, or "sample rate" as it is often called, of the audio signal to be processed.
7. The number of samples processed since the audio unit started rendering, or since this variable was last set to 0. The DSP code tracks total samples processed because:
 - The main processing loop is based on the number of samples placed into the input buffer...
 - But the DSP must take place independent of the input buffer size.
8. To keep the value of `mSamplesProcessed` within a reasonable limit, there's a test in the code to reset it when it reaches this value.

9. The scaling factor currently in use. The DSP uses a scaling factor to correlate the points in the wave table with the audio signal sampling frequency, in order to produce the desired tremolo frequency. The kernel object keeps track of a “current” and “next” scaling factor to support changing from one tremolo frequency to another without an audible glitch.
10. The desired scaling factor to use, resulting from a request by the user for a different tremolo frequency.

Write the TremoloUnitKernel Constructor Method

In the constructor method for your custom subclass of the `AUKernelBase` class, take care of any DSP work that can be done once per instantiation of the kernel object. In the case of the tremolo unit we're building, this includes:

- Initializing member variables that require initialization
- Filling the two wave tables
- Getting the sample rate of the audio stream.

A convenient location for the constructor is immediately below the `TremoloUnitEffectKernel` pragma mark.

Listing 5-11 Modifications to the TremoloUnitKernel Constructor (`TremoloUnit.cpp`)

```
#pragma mark ____TremoloUnitEffectKernel
// ~~~~~
// TremoloUnit::TremoloUnitKernel::TremoloUnitKernel()
// ~~~~~
TremoloUnit::TremoloUnitKernel::TremoloUnitKernel (AUEffectBase *inAudioUnit)
:
    AUKernelBase (inAudioUnit), mSamplesProcessed (0), mCurrentScale (0) // 1
{
    for (int i = 0; i < kWaveArraySize; ++i) { // 2
        double radians = i * 2.0 * pi / kWaveArraySize;
        mSine [i] = (sin (radians) + 1.0) * 0.5;
    }

    for (int i = 0; i < kWaveArraySize; ++i) { // 3
        double radians = i * 2.0 * pi / kWaveArraySize;
        radians = radians + 0.32;
        mSquare [i] =
            (
                sin (radians) +
                0.3 * sin (3 * radians) +
                0.15 * sin (5 * radians) +
                0.075 * sin (7 * radians) +
                0.0375 * sin (9 * radians) +
                0.01875 * sin (11 * radians) +
                0.009375 * sin (13 * radians) +
                0.8
            ) * 0.63;
    }
    mSampleFrequency = GetSampleRate (); // 4
}
```



```
}
```

Here's how this code works:

1. The constructor method declarator and constructor-initializer. In addition to calling the appropriate superclasses, this code initializes two member variables.
2. Generates a wave table that represents one cycle of a sine wave, normalized so that it never goes negative and so that it ranges between 0 and 1.

Along with the value of the Depth parameter, this sine wave specifies how to vary the audio gain during one cycle of sine wave tremolo.

3. Generates a wave table that represents one cycle of a pseudo square wave, normalized so that it never goes negative and so that it ranges between approximately 0 and approximately 1.
4. Gets the sample rate for the audio stream to be processed.

Override the Process Method

Put your DSP code into an override of the `AUKernelBase` class's `Process` method. The `Process` method gets called once each time the host application fills the audio sample input buffer. The method processes the buffer contents, sample by sample, and places the processed audio in the audio sample output buffer.

It's important to put as much processing as possible outside of the actual processing loop. The code for the TremoloUnit project demonstrates that the following sort of work gets done just once per audio sample buffer, outside of the processing loop:

- Declare all variables used in the method
- Get the current values of all the parameters as set by the user via the audio unit's view
- Check the parameters to ensure they're in bounds, and taking appropriate action if they're not
- Perform calculations that don't require updating with each sample. In this project's case, this means calculating the scaling factor to use when applying the tremolo wave table.

Inside the processing loop, do only work that must be performed sample by sample:

- Perform calculations that must be updated sample by sample. In this case, this means calculating the point in the wave table to use for tremolo gain
- Respond to parameter changes in a way that avoids artifacts. In this case, this means switching to a new tremolo frequency, if requested by the user, in a way that avoids any sudden jump in gain.
- Calculate the transformation to apply to each input sample. In this case, this means calculating a) the tremolo gain based on the current point in the wave table, and b) the current value of the Depth parameter.
- Calculate the output sample that corresponds to the current input sample. In this case, this means applying the tremolo gain and depth factor to the current input sample.
- Advance the indexes in the input and output buffers

- Advance other indexes involved in the DSP. In this case, this means incrementing the `mSamplesProcessed` variable.

Listing 5-12 The `Process` method (`TremoloUnit.cpp`)

```

void TremoloUnit::TremoloUnitKernel::Process (                // 1
    const Float32    *inSourceP,                            // 2
    Float32          *inDestP,                              // 3
    UInt32           inSamplesToProcess,                    // 4
    UInt32           inNumChannels,                        // 5
    bool             &ioSilence                           // 6
) {
    if (!ioSilence) {                                       // 7

        const Float32 *sourceP = inSourceP;                // 8

        Float32 *destP = inDestP,                          // 9
            inputSample,                                  // 10
            outputSample,                                 // 11
            tremoloFrequency,                             // 12
            tremoloDepth,                                 // 13
            samplesPerTremoloCycle,                       // 14
            rawTremoloGain,                               // 15
            tremoloGain;                                  // 16

        int    tremoloWaveform;                            // 17

        tremoloFrequency = GetParameter (kParameter_Frequency); // 18
        tremoloDepth     = GetParameter (kParameter_Depth);    // 19
        tremoloWaveform  =
            (int) GetParameter (kParameter_Waveform);          // 20

        if (tremoloWaveform == kSineWave_Tremolo_Waveform) { // 21
            waveArrayPointer = &mSine [0];
        } else {
            waveArrayPointer = &mSquare [0];
        }

        if (tremoloFrequency < kMinimumValue_Tremolo_Freq) // 22
            tremoloFrequency = kMinimumValue_Tremolo_Freq;
        if (tremoloFrequency > kMaximumValue_Tremolo_Freq)
            tremoloFrequency = kMaximumValue_Tremolo_Freq;

        if (tremoloDepth < kMinimumValue_Tremolo_Depth)    // 23
            tremoloDepth = kMinimumValue_Tremolo_Depth;
        if (tremoloDepth > kMaximumValue_Tremolo_Depth)
            tremoloDepth = kMaximumValue_Tremolo_Depth;

        if (tremoloWaveform != kSineWave_Tremolo_Waveform // 24
            && tremoloWaveform != kSquareWave_Tremolo_Waveform)
            tremoloWaveform = kSquareWave_Tremolo_Waveform;

        samplesPerTremoloCycle = mSampleFrequency / tremoloFrequency; // 25
        mNextScale = kWaveArraySize / samplesPerTremoloCycle;          // 26

        // the sample processing loop //////////////////////////////////

```

```

        for (int i = inSamplesToProcess; i > 0; --i) { // 27

            int index = // 28
                static_cast<long>(mSamplesProcessed * mCurrentScale) %
                    kWaveArraySize;

            if ((mNextScale != mCurrentScale) && (index == 0)) { // 29
                mCurrentScale = mNextScale;
                mSamplesProcessed = 0;
            }

            if ((mSamplesProcessed >= sampleLimit) && (index == 0)) // 30
                mSamplesProcessed = 0;

            rawTremoloGain = waveArrayPointer [index]; // 31

            tremoloGain = (rawTremoloGain * tremoloDepth - // 32
                tremoloDepth + 100.0) * 0.01;

            inputSample = *sourceP; // 33
            outputSample = (inputSample * tremoloGain); // 34
            *destP = outputSample; // 35
            sourceP += 1; // 36
            destP += 1; // 37
            mSamplesProcessed += 1; // 38
        }
    }
}

```

Here's how this code works:

1. The `Process` method signature. This method is declared in the `AUKernelBase` class.
2. The audio sample input buffer.
3. The audio sample output buffer.
4. The number of samples in the input buffer.
5. The number of input channels. This is always equal to 1 because there is always one kernel object instantiated per channel of audio.
6. A Boolean flag indicating whether the input to the audio unit consists of silence, with a `TRUE` value indicating silence.
7. Ignores the request to perform the `Process` method if the input to the audio unit is silence.
8. Assigns a pointer variable to the start of the audio sample input buffer.
9. Assigns a pointer variable to the start of the audio sample output buffer.
10. The current audio sample to process.
11. The current audio output sample resulting from one iteration of the processing loop.
12. The tremolo frequency requested by the user via the audio unit's view.
13. The tremolo depth requested by the user via the audio unit's view.

14. The number of audio samples in one cycle of the tremolo waveform.
15. The tremolo gain for the current audio sample, as stored in the wave table.
16. The adjusted tremolo gain for the current audio sample, considering the Depth parameter.
17. The tremolo waveform type requested by the user via the audio unit's view.
18. Gets the current value of the Frequency parameter.
19. Gets the current value of the Depth parameter.
20. Gets the current value of the Waveform parameter.
21. Assigns a pointer variable to the wave table that corresponds to the tremolo waveform selected by the user.
22. Performs bounds checking on the Frequency parameter. If the parameter is out of bounds, supplies a reasonable value.
23. Performs bounds checking on the Depth parameter. If the parameter is out of bounds, supplies a reasonable value.
24. Performs bounds checking on the Waveform parameter. If the parameter is out of bounds, supplies a reasonable value.
25. Calculates the number of audio samples per cycle of tremolo frequency.
26. Calculates the scaling factor to use for applying the wave table to the current sampling frequency and tremolo frequency.
27. The loop that iterates over the audio sample input buffer.
28. Calculates the point in the wave table to use for the current sample. This, along with the calculation of the `mNextScale` value in comment , is the only subtle math in the DSP for this effect.
29. Tests if the scaling factor should change, and if it's safe to change it at the current sample to avoid artifacts. If both conditions are met, switches the scaling factor and resets the `mSamplesProcessed` variable.
30. Tests if the `mSamplesProcessed` variable has grown to a large value, and if it's safe to reset it at the current sample to avoid artifacts. If both conditions are met, resets the `mSamplesProcessed` variable.
31. Gets the tremolo gain from the appropriate point in the wave table.
32. Adjusts the tremolo gain by applying the Depth parameter. With a depth of 100%, the full tremolo effect is applied. With a depth of 0%, there is no tremolo effect applied at all.
33. Gets an audio sample from the appropriate spot in the audio sample input buffer.
34. Calculates the corresponding output audio sample.
35. Places the output audio sample at the appropriate spot in the audio sample output buffer.
36. Increments the position counter for the audio sample input buffer.

37. Increments the position counter for the audio sample output buffer.
38. Increments the count of processed audio samples.

Override the Reset Method

In your custom subclass's override of the `AUKernelBase` class `Reset` method, you do whatever work is necessary to return the audio unit to its pristine, initialized state.

Listing 5-13 The `Reset` method (`TremoloUnit.cpp`)

```
void TremoloUnit::TremoloUnitKernel::Reset() {
    mCurrentScale      = 0;           // 1
    mSamplesProcessed   = 0;           // 2
}
```

Here's how this code works:

1. Resets the `mCurrentScale` member variable to its freshly initialized value.
2. Resets the `mSamplesProcessed` member variable to its freshly initialized value.

Implement the Tail Time Property

Finally, to ensure that your audio unit plays well in host applications, implement the tail time property, `kAudioUnitProperty_TailTime`.

To do this, simply state in your `TremoloUnit` class definition that your audio unit supports the property by changing the return value of the `SupportsTail` method to `true`.

Listing 5-14 Implementing the tail time property (`TremoloUnit.h`)

```
virtual bool SupportsTail () {return true;}
```

Given the nature of the DSP your audio unit performs, its tail time is 0 seconds—so you don't need to override the `GetTailTime` method. In the `AUBase` superclass, this method reports a tail time of 0 seconds, which is what you want your audio unit to report.

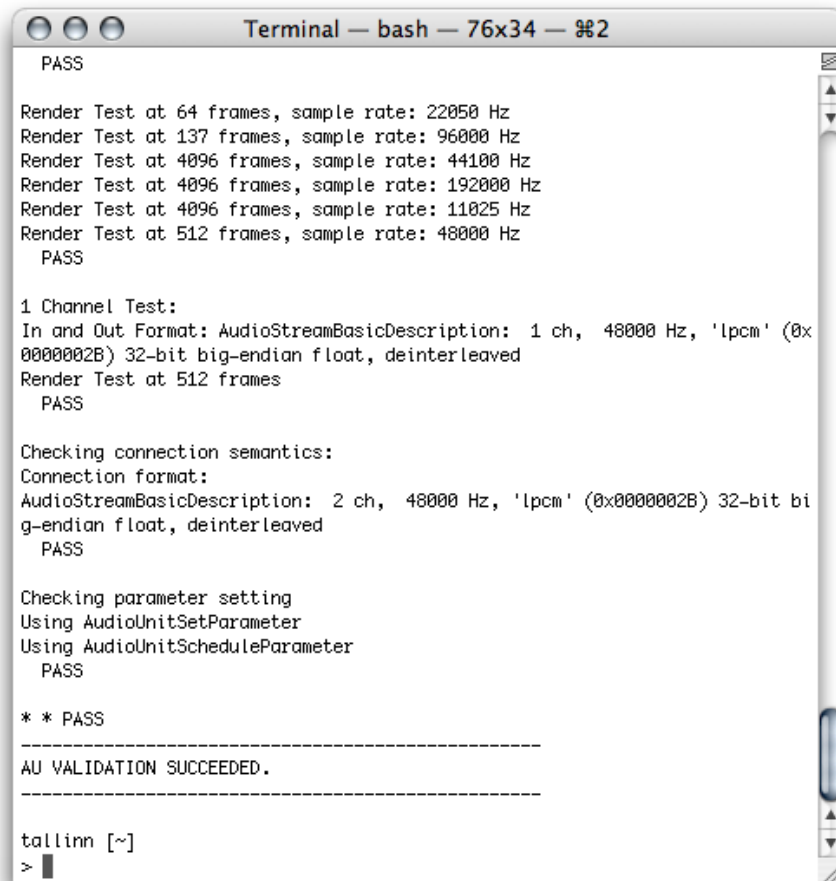
Validate your Completed Audio Unit

Now you've implemented all of the code for the tremolo unit project. Build the project and correct any errors you see. Then use the `auval` tool to validate your audio unit.

1. In Terminal, enter the command to validate the tremolo unit. This consists of the `auval` command name followed by the `-v` flag to invoke validation, followed by the type, subtype, and manufacturer codes that identify the tremolo unit. The complete command for this audio unit is:

```
auval -v aux tmlo Aud
```

If everything is in order, `auval` should report that your new audio unit is indeed valid. The figure shows the last bit of the log created by `auval`:



```

PASS

Render Test at 64 frames, sample rate: 22050 Hz
Render Test at 137 frames, sample rate: 96000 Hz
Render Test at 4096 frames, sample rate: 44100 Hz
Render Test at 4096 frames, sample rate: 192000 Hz
Render Test at 4096 frames, sample rate: 11025 Hz
Render Test at 512 frames, sample rate: 48000 Hz
PASS

1 Channel Test:
In and Out Format: AudioStreamBasicDescription: 1 ch, 48000 Hz, 'lpcm' (0x0000002B) 32-bit big-endian float, deinterleaved
Render Test at 512 frames
PASS

Checking connection semantics:
Connection format:
AudioStreamBasicDescription: 2 ch, 48000 Hz, 'lpcm' (0x0000002B) 32-bit big-endian float, deinterleaved
PASS

Checking parameter setting
Using AudioUnitSetParameter
Using AudioUnitScheduleParameter
PASS

* * PASS

-----
AU VALIDATION SUCCEEDED.
-----

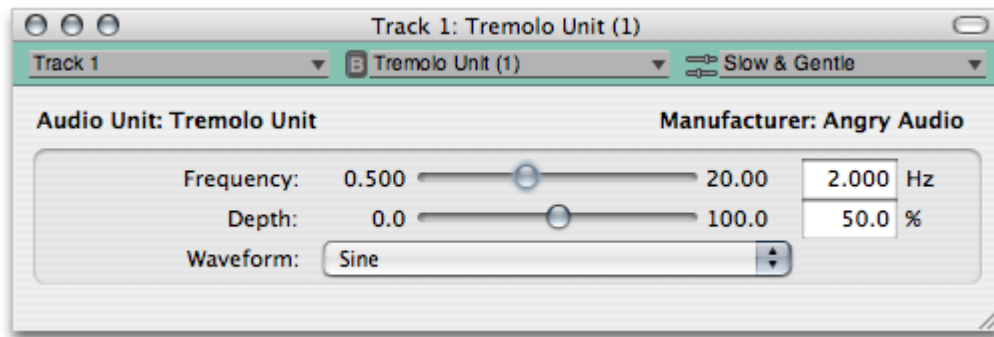
tallinn [~]
>

```

Test your Completed Audio Unit

Your audio unit is ready for you to test in a host application. Apple recommends the AU Lab application for audio unit testing. You may also want to test your audio unit in other hosts. This section describes testing with AU Lab. Follow these steps:

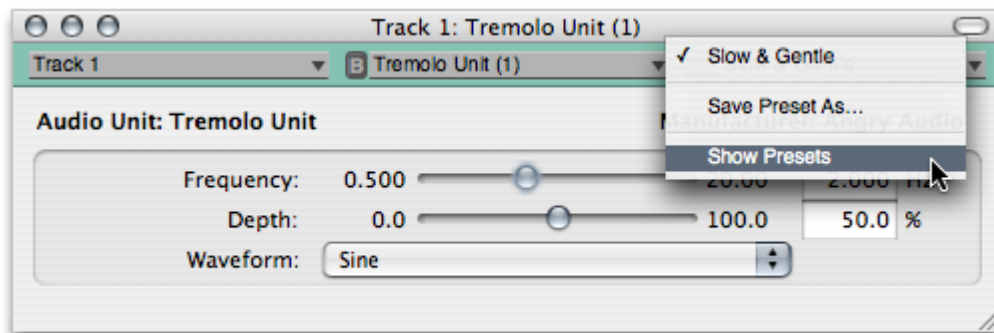
1. Open the AU Lab document that you saved from the [“Test the Unmodified Audio Unit”](#) (page 106) section. If you don’t have that document, run through the steps in that section again to set up AU Lab for testing your audio unit.
2. Your completed audio unit appears with the generic view as shown in the figure:



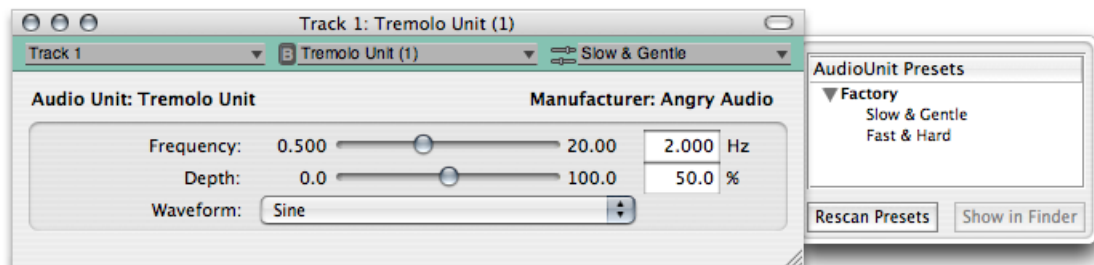
(You may need to quit and reopen AU Lab for your completed audio unit's view to appear.)

Notice the sliders for the Frequency and Depth parameters, the pop-up menu for the Waveform parameter, and the default preset displayed in the Factory Presets pop-up menu.

3. Click the Play button in the AUAudioFilePlayer utility window. If everything is in order, the file you have selected in the player will play through your audio unit and you'll hear the tremolo effect.
4. Experiment with the range of effects available with the tremolo unit: adjust the Frequency, Depth, and Waveform controls while listening to the output from AU Lab.
5. In the presets menu toward the upper right of the generic view, choose Show Presets.



The presets drawer opens.



Disclose the presets in the Factory group and verify that the presets that you added to your audio unit, using the `NewFactoryPresetSet` method, are present. Double-click a preset to load it.

You can add user presets to the audio unit as follows:

- Set the parameters as desired
- In the presets menu, choose **Save Preset As**
- In the dialog that appears, enter a name for the preset.

Appendix: Audio Unit Class Hierarchy

This appendix describes the Core Audio SDK audio unit class hierarchy, including starting points for common types of audio units.

Core Audio SDK Audio Unit Class Hierarchy

The following figure illustrates the main classes and class relationships in the Core Audio SDK audio unit class hierarchy, for Core Audio SDK v1.4.3.

```

graph TD
    AUElementCreator --> ComponentBase
    AUElementCreator --> AUBase
    AUBase --> AUOutputBase
    AUBase --> AUEffectBase
    AUBase --> MusicDeviceBase
    AUEffectBase --> AUMIDIBase
    AUEffectBase --> AUMIDIEffectBase
    MusicDeviceBase --> AUInstrumentBase
    AUInstrumentBase --> AUMonotimbralInstrumentBase
    AUInstrumentBase --> AUMultitimbralInstrumentBase
    AUBase --> AUScope
    AUScope --> AUElement
    AUElement --> AUIOElement
    AUElement --> SynthElement
    AUIOElement --> AUInputElement
    AUIOElement --> AUOutputElement
    SynthElement --> SynthPartElement
    SynthElement --> SynthGroupElement
    SynthGroupElement --> SynthNoteList
    SynthNoteList --> SynthNote
  
```

- 140

Document Revision History

This table describes the changes to *Audio Unit Programming Guide*.

Date	Notes
2006-11-07	Added “Provide Strings for the Waveform Pop-up Menu” (page 119) section in “Tutorial: Building a Simple Effect Unit with a Generic View” (page 89) chapter.
	Corrected “Define the Factory Presets” (page 124) section in “Tutorial: Building a Simple Effect Unit with a Generic View” (page 89) chapter.
	Improved naming scheme for parameter and preset constants in “Tutorial: Building a Simple Effect Unit with a Generic View” (page 89) chapter.
	Added link to the TremoloUnit sample code project in “Introduction” (page 9).
2006-08-07	First version

REVISION HISTORY

Document Revision History