# Building Audio Effect Plug-ins

Audio plug-ins are the most common way of implementing audio effects and virtual instruments in software. A typical plug-in is a self-contained block of code which is compiled to run on a particular processor and operating system, but which can be used within audio software environments. This chapter will examine the process of creating plug-ins using the *JUCE* (*Jules’ Utility Class Extensions*) programming framework, which can be used to create plug-ins for many different software platforms.

## Plug-in basics

Audio plug-ins are designed to be self-contained effects which can be used within a host environment, usually a digital audio workstation (DAW). To ensure compatibility across different DAWs, several industry-standard plug-in formats have been developed. These include Steinberg’s *VST* (Virtual Studio Technology) format, widely supported in nearly all professional audio software; Apple’s *AudioUnit* format, supported on most Mac programs; and the *AAX* (Avid Audio Extension) format by Digidesign/Avid. Each format provides similar functionality, typically including ways of passing audio into and out of the plug-in, negotiating sample rates and number of channels, and querying and setting user-adjustable parameters for the effect.

### Programming language

Most audio plug-ins are written in the C or C++ languages. The code will be compiled to run on a specific processor and operating system. However, the same code can typically be compiled to run on any hardware and operating system, as long as the code does not use any OS-specific functionality.

This book is not intended as an introduction to C or C++. Examples in the remainder of this chapter will be presented in C++ with the assumption that the reader has a basic familiarity (though not necessarily significant expertise) with the language. Excellent introductions to C++ programming can be found in many sources, including [94-96]. An audio-focused introduction to C++ is included in [97], which also goes into detail on many aspects of audio programming not covered in this text.

### Plug-in properties

The essential task of an audio plug-in is to receive an input audio signal, apply an effect to it, and produce an output audio signal. In some cases, including virtual instrument or synthesizer plug-ins, no audio input is used, and the output may be produced in response to MIDI (Musical Instrument Digital Interface) messages.

Important properties of a plug-in include the number of channels it supports and the allowable sample rates. Many effects can operate with different numbers of channels (for example, on mono or stereo inputs), but others will require specific channel configurations (for example, a ping-pong delay would need at least two output channels, but it could take one or two inputs). Some effects may have restrictions on the sample rates they support, though it is useful wherever possible to write plug-ins that operate at any sample rate. Plug-ins will also define one or more user-adjustable parameters which can be changed either through a standard interface provided by the host environment or by a custom GUI created by the plug-in author.

### The JUCE framework

The example code for this book uses the JUCE environment, created by Julian Storer, which provides a cross-platform, multi-format method for building audio plug-ins. JUCE projects are written in C++, and the JUCE libraries allow the same code to be compiled into VST, AudioUnit, AAX and other plug-in formats. JUCE also provides a cross-platform set of graphical user interface (GUI) controls and a very large library of useful C++ classes. JUCE is free for use in open-source, noncommercial projects. Documentation and download links can be found on its website: http://www.juce.com

## Theory of operation

JUCE audio plug-ins are divided into two components: a processor that handles the audio calculations and an editor or GUI that lets the user interact with the plug-in. The processor provides several functions: a callback function that the DAW calls every time it needs a new block of audio samples; methods for getting and setting effect parameters; and initialization and cleanup routines. The editor provides graphical controls for the user to see and change the parameters. Many hosts will provide a generic editor when the plug-in does not define its own.

### Callback function

The most important task for an audio plug-in is receiving and processing audio samples. How does the plug-in know how many samples to process, and when to process them? If the plug-in is operating in real time, it is clearly impossible to wait for the entire audio signal to arrive before applying any processing. Instead, audio needs to be processed in small blocks or buffers of samples as it comes in. To receive blocks of samples, plug-ins implement a callback function, a function which the DAW calls every time it has new audio to process. Thus, it is always the host, and not the plug-in, which determines how many audio samples to process and when. The advantage of this arrangement is that the plug-in author never needs to be concerned with where the audio samples come from, when they should arrive, or where they go after the effect has been applied. The author simply needs to write a function which processes as many samples as requested by the DAW.

When the host application runs the callback function, it will provide several pieces of information. These include the buffer size (how many audio samples to process), the sample rate, the number of input and output channels, and a buffer (region of memory) containing the input audio. The host will also provide a buffer in which the audio output should be stored. In JUCE (as in many plug-in formats), the plug-in is expected to put its output in the same buffer where the input samples were found.

When the callback function finishes, it simply returns control to the host, which decides what should happen to the processed samples. The callback function will be called again when there are more samples to process. The buffer size used by the DAW partly determines the overall latency (delay) from input to output; smaller buffer sizes produce a low delay but increase the risk of underruns (gaps) if the computer cannot respond quickly enough. A conservative buffer size might be 1024 samples; a high-performance buffer size might be as small as 32 samples. At a 48 kHz sample rate, a buffer size of 32 means that the callback function would run over 1500 times per second!

### Managing parameters

Nearly every audio effect will have one or more user-adjustable parameters. For example, a dynamic range compressor plug-in might let the user change threshold, ratio, attack, release, knee and make-up gain parameters. Most plug-ins provide a user interface to allow the user to adjust parameters. Every operating system and plug-in format provides different routines for managing a GUI, but there are several common requirements. In particular, the plug-in must provide a set of functions for the user to see or change the parameters, and there must be a way for the callback function to discover the current parameter values.

JUCE provides several methods (C++ object functions) related to managing parameters, described in detail in the example in the next section. Two of the most important of these methods are getValue() and setValue() associated with listeners for GUI objects (see Figure 13.1). When the DAW calls getValue(), it provides a single argument indicating the index of the parameter it wants to query. The plug-in is expected to return the value of this parameter as a floating-point number (C++ type float). It is up to the plug-in to define what the indices mean (for example, a parametric equalizer might have frequency as parameter 0, gain as parameter 1 and Q as parameter 2). With setValue(), the host provides an index of the parameter to set and a floating-point value it should be set to. The plug-in stores this information so the audio callback function can access it later.

In JUCE, the plug-in keeps track of its current parameters by using instance variables, variables declared inside a C++ class which are accessible to any of its methods. When the DAW calls getValue(), the current value of the relevant instance variable is returned; when it calls setValue(), the value of the instance variable is changed. The callback function, processBlock(), accesses the values of these variables to discover the current parameter settings.

### Initialization and cleanup

Before a plug-in can process audio, certain initialization tasks must be performed. For example, in a delay plug-in, memory for delay buffers might need to be allocated; for an equalizer plug-in, filter coefficients might need to be calculated and internal variables holding previous samples might need to be initialized to 0. For plug-ins with low frequency oscillators (LFOs), the phase of the oscillator may need to be initialized.

Every plug-in format will provide a method for initializing the plug-in. In JUCE, basic initialization can be performed in the constructor of the audio processor object, which runs once when the plug-in is first loaded. JUCE also provides a method called prepareToPlay() which runs immediately before audio processing begins. By allocating resources just before the audio starts, the plug-in only uses resources while actively running.

All resources that are allocated by the plug-in will eventually need to be released. JUCE provides a method called releaseResources() which runs immediately after the host stops processing audio. This method should be used to free any resources allocated in prepareToPlay(). It is safe to assume that the audio callback function will never run after a call to releaseResources(). Similarly, the counterpart to the C++ constructor is the destructor, which runs once when the user removes the plug-in from the host environment. Anything that is allocated in the constructor should be freed in the destructor.

### Preserving state

Each time the host application calls the plug-in’s callback function, it will provide only a small block of input samples to be processed. The plug-in often depends on previous state information to know how to process these samples. For example, phases of oscillators, pointers within circular buffers, and previous values of input and output samples may be needed to calculate the output. It is up to the plug-in to save any state that it needs during the callback function.

To understand the importance of managing state, consider a simple effect where the current output is equal to the sum of the last two inputs: y[n] = x[n] + x[n-1]. Suppose that the host requests 256 (28) samples beginning at sample N. It will therefore supply a buffer containing 256 input samples x[N] to x[N+255] and require 256 output samples y[N] to y[N+255].

It may seem as if the callback function has enough information to calculate the output without any reference to what has happened before. But what about calculating y[N], the very first sample in the buffer? We have y[N] = x[N] + x[N-1], but x[N-1] was supplied last time the callback was run, and that value is no longer available. The plug-in must therefore use a separate instance variable to remember what has come before. Here, a single float for each channel would be needed to remember the previous sample. The float should be declared inside the class, so its value is preserved across calls to the callback function.

The following example shows the implementation of a delay plug-in in JUCE, including the management of parameters and preservation of state between callbacks.

## Example: building a delay effect in JUCE

This section will describe the implementation of a simple plug-in in the JUCE environment. We will take as an example a delay with feedback (Chapter 2), with user-adjustable controls for delay time, feedback level, wet and dry mix. The complete code for this effect can be found in the materials that accompany the book, along with example code for several other effects.

### Required software

To build the plug-ins that come with the book, the following software is required:

1. The JUCE (Jules’ Utility Class Extensions) C++ library by Julian Storer. JUCE runs on nearly every platform, including Mac, Windows and Linux, and can be downloaded free from http://www.juce.com.
2. A C++ compiler: Visual Studio on Windows, Xcode on Mac OS X, or gcc on Linux. They are all generally available as free downloads, sometimes after a registration process.
3. On the Mac, two other pieces are required: Xcode Audio Tools, which can be installed from within Xcode and the Core Audio Utility Classes, which can be found on Apple’s website. Platform-specific setup instructions often change over time; please consult the companion website for this book to find the most recent instructions.
4. To build VST plug-ins, the VST3 SDK is required. This can be obtained free from Steinberg (http://www.steinberg.net) after creating a developer account. The VST SDK is required on Windows, but is optional on Mac where AudioUnits can be created instead. Whether you use VST or AudioUnits, your program in JUCE will be written identically.
5. A suitable digital audio workstation (DAW) or other host environment in which to test the plug-ins. On the Mac, the Xcode Audio Tools provide the AU Lab program which is a simple, lightweight way of testing AudioUnit plug-ins. Several other cross-platform options are available, including Audacity and Reaper. JUCE also comes with its own audio plugin host, which we will often use.

### Creating a new plug-in in JUCE

The audio plug-in host needs to be compiled on your system before it can be used. Within the main JUCE folder, open the project in your development environment and build it. It should compile without errors if JUCE was correctly installed.

In JUCE, new projects are created using a program called the Projucer. Run the Projucer and select New Project from the File menu. Enter a project name and set the Project Type to Audio Plug-In (see Figure 13.2). Click Create to make the Juce project for your new plug-in. From this point, you will get a window where you can edit the details of the plug-in (Figure 13.3). Several fields are important:

* Version: the version number of your plug-in (the default value is fine for new plug-ins)
* Company Name: the name of your company or organization, as you want it to appear in the audio host environment
* Bundle Identifier: a unique identifier for the plug-in. The format looks like a reversed web address, which identifies the organization and the plug-in name.
* Built VST/AudioUnit/AAX: these boxes select which format of plug-in to build. On Windows, VST3 is the most common format; on Mac, AudioUnit is the usual selection.
* Plugin Name: the name of your plug-in as you want it to appear in the host environment
* Plugin Description: a short description of your plug-in
* Plugin Manufacturer: company name, similar to Company Name above
* Plugin Manufacturer Code: a four-letter code identifying your company or organization. Use a single code consistently across all the plug-ins you develop.
* Plugin Channel Configurations: pairs of numbers which identify valid configurations of input channels and output channels. For example, {1,1}, {2,2} indicates a plug-in that can take mono input and mono output, or stereo input and stereo output.
* Silence: most effects can leave this checked, but if your plug-in produces sound even when the input is silent, uncheck this box.

The other settings can usually be left at their defaults. The next step is to make a target for your specific compiler. By default, the Projucer should create a target for the compiler on your platform. If not, or if you want to make an additional target, right-click on the plug-in icon (Figure 13.4).

Selecting the compiler target from the list on the left-hand side of the window brings up settings specific to the platform. Most of these options can be left at their default values. It is important, though, to correctly set the Local JUCE folder to the location where you have installed Juce. If you are building a VST plug-in, setting the VST Folder is also important.

When you have finished entering all the details, click Save Project and Open in Xcode (on Mac) or Save Project and Open in Visual Studio (on Windows). This should bring up the project within your compiler, where it can be built using the same procedure you normally use to compile projects. On the Mac, building the project will automatically copy the resulting plug-in to the system plug-ins directory (~/Library/Audio/Plug-Ins/Components/) where it will be found by audio software including AU Lab. On Windows, your DAW may need to be manually set to look for your plug-in. The plug-in list may need to be regenerated every time the plug-in is recompiled. In some cases, restarting the host program is necessary.

### Opening example plug-ins

This book comes with many example audio plug-ins which can be opened in the compiler or your choice. Within each plug-in directory, look in the .jucer file in the Projucer for a version specific to your compiler. If you do not find one, create a new target for your platform. Following the instructions in the previous section, you should be able to compile the plug-in and get it running in your audio host environment.

### File overview

Juce plug-ins contain four source files by default:

* PluginProcessor.h: the header file for PluginProcessor.cpp; defines the C++ object whose code is filled out in PluginProcessor.cpp. Includes declaration of methods and object variables.
* PluginProcessor.cpp: the main file where the audio processing is done.
* PluginEditor.h: the header file for PluginEditor.cpp, containing the definition of the C++ object for the editor.
* PluginEditor.cpp: contains the code for the graphical user interface for the plug-in.

In your own projects, you might want to add extra source files to the project. But many basic audio effects and synths can be implemented in just these four. We will examine each file in turn. Not all lines of each file are included in this text, so please refer to the accompanying materials for the complete source.

### PluginProcessor.h

**Declaration and Methods**

PluginProcessor.h contains the declaration of the DelayAudioProcessor class which implements the delay plug-in. Its definition begins here:

class DelayAudioProcessor : public AudioProcessor

{

public:

//==============================================================================

DelayAudioProcessor();

~DelayAudioProcessor();

In your own effects, you may want to change the name of the class to reflect what your plug-in does. Be sure to change the name consistently across all four source files. This block of code also shows the first two methods of the DelayAudioProcessor class: DelayAudioProcessor() and ~DelayAudioProcessor(). These are the constructor and destructor, respectively. They run when the plug-in is loaded and unloaded by the DAW. The following lines declare a collection of other methods which are standard to every JUCE plug-in:

//==============================================================================

void prepareToPlay (double sampleRate, int samplesPerBlock);

void releaseResources();

void reset();

void processBlock (AudioSampleBuffer& buffer, MidiBuffer& midiMessages);

//==============================================================================

AudioProcessorEditor\* createEditor();

bool hasEditor() const;

//==============================================================================

const String getName() const;

int getNumParameters();

float getParameter (int index);

void setParameter (int index, float newValue);

const String getParameterName (int index);

const String getParameterText (int index);

const String getInputChannelName (int channelIndex) const;

const String getOutputChannelName (int channelIndex) const;

bool isInputChannelStereoPair (int index) const;

bool isOutputChannelStereoPair (int index) const;

bool silenceInProducesSilenceOut() const;

double getTailLengthSeconds() const;

bool acceptsMidi() const;

bool producesMidi() const;

//==============================================================================

int getNumPrograms();

int getCurrentProgram();

void setCurrentProgram (int index);

const String getProgramName (int index);

void changeProgramName (int index, const String& newName);

//==============================================================================

void getStateInformation (MemoryBlock& destData);

void setStateInformation (const void\* data, int sizeInBytes);

The contents of these methods will be found in PluginProcessor.cpp. These lines should not be modified, however it is common to add your own methods in more complex plug-ins. Additional methods can be declared either below these standard methods, or if the methods will only be called internally by the object itself, below the private: line later in the file.

**Variables**

Following the methods is a declaration of instance variables for the class:

//==============================================================================

// these are used to persist the UI's size - the values are stored along with the

// filter's other parameters, and the UI component will update them when it gets

// resized.

int lastUIWidth\_, lastUIHeight\_;

enum Parameters

{

kDelayLengthParam = 0,

kDryMixParam,

kWetMixParam,

kFeedbackParam,

kNumParameters

};

// Adjustable parameters:

float delayLength\_; // Length of delay line in seconds

float dryMix\_; // Mix level of original signal (0-1)

float wetMix\_; // Mix level of delayed signal (0-1)

float feedback\_; // Feedback level (0-just less than 1)

private:

// Circular buffer variables for implementing delay

AudioSampleBuffer delayBuffer\_;

int delayBufferLength\_;

int delayReadPosition\_, delayWritePosition\_;

//==============================================================================

JUCE\_DECLARE\_NON\_COPYABLE\_WITH\_LEAK\_DETECTOR (DelayAudioProcessor);

};

By declaring the variables here rather than in a particular method, the variables can be accessed from any method within the object. The variables declared below the private: line are accessible only within the DelayAudioProcessor class; the others are accessible to any object (public). Here, these variables are declared public because the DelayAudioProcessorEditor class needs to access them. The function of the variables is as follows:

1. lastUIWidth\_ and lastUIHeight\_ are used by Juce to remember the size of the graphical interface between times the plug-in was loaded.
2. The enum statement declares a group of constant values in sequential order. For example, kDelayLengthParam has a value of 0, kDryMixParam has a value of 1, and so forth. These define the indices of parameters that control the plug-in. Your plug-in may change this list, but always keep kNumParameters at the end so the code in PluginProcessor.cpp knows the number of valid parameters.
3. delayLength\_, dryMix\_, wetMix\_ and feedback\_ hold the current values of each parameter. The convention of putting an underscore at the end of the variable name helps distinguish instance variables from local variables declared in particular methods. For clarity it is helpful to maintain this convention when declaring new instance variables in your plug-in.
4. delayBuffer\_, delayBufferLength\_, delayReadPosition\_ and delayWritePosition\_ are internal state variables used by the delay code. They are declared private because only the DelayAudioProcessor class needs access to them. Their uses are explained in the section on PluginProcessor.cpp. Your plug-ins will likely declare a different list of state variables here.

The final line of the class is a macro used internally by JUCE, and should not be changed (except to update the name of the class if you change it). Notice that the initial values of the variables have not been declared here. This will be done in PluginProcessor.cpp.

### PluginProcessor.cpp

Of the four files in the plug-in, PluginProcessor.cpp is the one that handles the most important parts of the audio processing. The file implements the methods and uses the instance variables declared in PluginProcessor.h. The two most important elements of this file are the audio callback function processBlock() and a collection of methods for getting and setting parameters. The complete file can be found in the accompanying materials to this book. Rather than proceeding line-by-line, let us begin by examining the audio callback function.

**Audio callback**

void DelayAudioProcessor::processBlock (AudioSampleBuffer& buffer, MidiBuffer& midiMessages)

{

// Helpful information about this block of samples:

const int numInputChannels = getNumInputChannels(); // How many input channels for our

// effect?

const int numOutputChannels = getNumOutputChannels(); // How many output channels for our

// effect?

const int numSamples = buffer.getNumSamples(); // How many samples in the buffer for

// this block?

int channel, dpr, dpw; // dpr = delay read pointer; dpw = delay write pointer

// Go through each channel of audio that's passed in. In this example we apply identical

// effects to each channel, regardless of how many input channels there are. For some effects,

// like a stereo chorus or panner, you might do something different for each channel.

for (channel = 0; channel < numInputChannels; ++channel)

{

// channelData is an array of length numSamples which contains the audio for one channel

float\* channelData = buffer.getSampleData(channel);

// delayData is the circular buffer for implementing delay on this channel

float\* delayData = delayBuffer\_.getSampleData (jmin (channel,

delayBuffer\_.getNumChannels() - 1));

// Make a temporary copy of any state variables declared in PluginProcessor.h which need to

// be maintained between calls to processBlock(). Each channel needs to be processed

// identically which means that the activity of processing one channel can't affect the

// state variable for the next channel.

dpr = delayReadPosition\_;

dpw = delayWritePosition\_;

for (int i = 0; i < numSamples; ++i)

{

const float in = channelData[i];

float out = 0.0;

// In this example, the output is the input plus the contents of the delay buffer

// (weighted by delayMix)

// The last term implements a tremolo (variable amplitude) on the whole thing.

out = (dryMix\_ \* in + wetMix\_ \* delayData[dpr]);

// Store the current information in the delay buffer. delayData[dpr] is the delay

// sample we just read, i.e. what came out of the buffer. delayData[dpw] is what we

// write to the buffer, i.e. what goes in

delayData[dpw] = in + (delayData[dpr] \* feedback\_);

if (++dpr >= delayBufferLength\_)

dpr = 0;

if (++dpw >= delayBufferLength\_)

dpw = 0;

// Store the output sample in the buffer, replacing the input

channelData[i] = out;

}

}

// Having made a local copy of the state variables for each channel, now transfer the result

// back to the main state variable so they will be preserved for the next call of

// processBlock()

delayReadPosition\_ = dpr;

delayWritePosition\_ = dpw;

// In case we have more outputs than inputs, we'll clear any output

// channels that didn't contain input data, (because these aren't

// guaranteed to be empty - they may contain garbage).

for (int i = numInputChannels; i < numOutputChannels; ++i)

{

buffer.clear (i, 0, buffer.getNumSamples());

}

}

The first line of this block specifies the name of the method, which is prefaced by DelayAudioProcessor:: to indicate that this method is part of the DelayAudioProcessor class. The method takes two arguments, buffer and midiMessages, which hold the audio and MIDI input information, respectively. In this example we will ignore the MIDI messages, but they might be used in a synthesizer plug-in. Since the method is declared void, it does not return a value. processBlock() is the callback function that will be run by the DAW every time there are new audio samples to process.

The first three lines of the method establish some basic properties of the environment. JUCE provides standard functions for retrieving the number of input channels, the number of output channels, and the sample rate (getSampleRate()) though the latter is not used here. The input argument buffer is an object of type AudioSampleBuffer, which holds information about the number and content of input samples.

Given a number of input channels and number of requested samples, the basic procedure is to iterate through every sample of every channel and apply the effect to each sample. The method contains two nested for() loops, first iterating over the channels, then iterating over the samples within each channel. For each channel, the code first converts the JUCE AudioSampleBuffer class to a standard C array:

// channelData is an array of length numSamples which contains the audio for one channel

float\* channelData = buffer.getSampleData(channel);

// delayData is the circular buffer for implementing delay on this channel

float\* delayData = delayBuffer\_.getSampleData (jmin (channel,

delayBuffer\_.getNumChannels() - 1));

At this point, the array channelData holds the input audio samples for one channel and delayData holds the contents of the delay buffer. The core audio processing then takes place inside the inner for loop:

const float in = channelData[i];

float out = 0.0;

The above lines store the current input sample in the local variable in and declare a variable out which will hold the result.

out = (dryMix\_ \* in + wetMix\_ \* delayData[dpr]);

This applies the basic delay equation from Chapter 2, calculating the output in terms of the sum of the input and the output of the delay buffer. Each term is weighted by a parameter (dryMix\_ and wetMix\_ respectively). dpr is the read pointer into the delay buffer which keeps track of the next sample to read out. Next, the content of the delay buffer is updated:

delayData[dpw] = in + (delayData[dpr] \* feedback\_);

dpw is the write pointer into the delay buffer. The content of the buffer is updated as the sum of the input and the feedback from the output, as detailed in Chapter 2. Finally, the position of the read and write pointers is advanced, wrapping around at the end to form a circular buffer:

if (++dpr >= delayBufferLength\_)

dpr = 0;

if (++dpw >= delayBufferLength\_)

dpw = 0;

delayBufferLength\_ is an instance variable which has its value set elsewhere to indicate the size of the delay buffer in samples.

In addition to updating the read and write pointers each sample, two other questions must be addressed: first, how are the read and write pointers initially set? Second, how are their values maintained at the end of the callback? We will address the second question first.

**Maintaining values across callbacks**

The callback will process only a small buffer of samples at a time, and the pointers cannot be allowed to reset with each call. To remember the values across callbacks, the instance variables delayReadPosition\_ and delayWritePosition\_ are used. Since they are declared in PluginProcessor.h, their values will persist beyond the end of the callback.

It might seem reasonable to use delayReadPosition\_ and delayWritePosition\_ directly in place of dpr and dpw in processBlock(). However, consider the operation of a stereo effect, where each channel is processed in turn. We do not want the left channel to change the pointer locations for the right channel. dpr and dpw are therefore introduced as local copies of delayReadPosition\_ and delayWritePosition\_:

dpr = delayReadPosition\_;

dpw = delayWritePosition\_;

At the end of the callback, their values are written back to the instance variables:

delayReadPosition\_ = dpr;

delayWritePosition\_ = dpw;

This plug-in applies the same effect to all channels; in other cases, such as a stereo flanger or chorus where the channels are handled differently, the outer for() loop iterating over the channels might be handled differently, and in some situations, separate instance variables for each channel may be required to remember the state between callbacks.

**Initialization**

Before the audio callback can run, all parameters and instance variables need to be initialized with usable values. Initialization in JUCE plug-ins takes place in two functions: in the constructor DelayAudioProcessor(), which runs once when the plug-in is loaded, and in prepareToPlay(), which runs each time the audio is started, just before the first call to processBlock(). Here is the constructor:

DelayAudioProcessor::DelayAudioProcessor() : delayBuffer\_ (2, 1)

{

// Set default values:

delayLength\_ = 0.5;

dryMix\_ = 1.0;

wetMix\_ = 0.5;

feedback\_ = 0.75;

delayBufferLength\_ = 1;

// Start the circular buffer pointers at the beginning

delayReadPosition\_ = 0;

delayWritePosition\_ = 0;

lastUIWidth\_ = 370;

lastUIHeight\_ = 140;

}

In the first line, the colon following the method name starts the initialization list, a way of initializing variables and objects in C++. The delayBuffer\_ object, used to hold the internal delay buffer, is initialized to have two channels with one sample per channel. Its size will be updated later, when more is known about the audio sample rate. Within the constructor, all other variables declared in PluginProcessor.h are given initial values. Some of these, including delayReadPosition\_ and delayWritePosition\_, will be updated before audio begins playing. If a variable is not initialized, its value will be undefined and its behavior will be unpredictable.

The second round of initialization takes place in prepareToPlay(), at which point the audio sample rate and system buffer size are known:

void DelayAudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)

{

// Allocate and zero the delay buffer (size will depend on current sample rate)

// Sanity check the result so we don't end up with any zero-length calculations

delayBufferLength\_ = (int)(2.0\*sampleRate);

if(delayBufferLength\_ < 1)

delayBufferLength\_ = 1;

delayBuffer\_.setSize(2, delayBufferLength\_);

delayBuffer\_.clear();

// This method gives us the sample rate. Use this to figure out what the delay position

// offset should be (since it is specified in seconds, and we need to convert it to a number

// of samples)

delayReadPosition\_ = (int)(delayWritePosition\_ - (delayLength\_ \* getSampleRate())

+ delayBufferLength\_) % delayBufferLength\_;

}

This effect allows a maximum delay of 2 seconds. The necessary buffer size for a 2-second delay depends on the sample rate, so here delayBuffer\_ and delayBufferLength\_ are updated to the correct size. The actual length of delay depends on the difference between the read and write pointers, so delayReadPosition\_ is given an initial offset with respect to delayWritePosition\_ to achieve the delay specified by delayLength\_. This calculation features modulo arithmetic needed to manage the circular buffer, described further in Chapter 2.

**Managing parameters**

While the effect is running, the user may change the value of the parameters. PluginProcessor.cpp implements a number of methods related to parameter management, details of which can be found in the JUCE documentation. We will highlight the most important ones here. First, getNumParameters() reports how many parameters the plug-in supports: ***EDIT: is this relevant?***

int DelayAudioProcessor::getNumParameters()

{

return kNumParameters;

}

As long as the enum statement in PluginProcessor.h is correctly handled, this code should never need to be updated. Next, getParameter() returns the value of a parameter given its index:

float DelayAudioProcessor::getParameter (int index)

{

// This method will be called by the host, probably on the audio thread, so

// it's absolutely time-critical. Don't use critical sections or anything

// UI-related, or anything at all that may block in any way!

switch (index)

{

case kDryMixParam: return dryMix\_;

case kWetMixParam: return wetMix\_;

case kFeedbackParam: return feedback\_;

case kDelayLengthParam:return delayLength\_;

default: return 0.0f;

}

}

Generally speaking, it is sufficient to return the current value of the relevant instance variable in this method. As the comments in the code indicate, nothing time-consuming or GUI-related should take place in this method. Finally, getParameterName() provides a human-readable string describing each parameter:

const String DelayAudioProcessor::getParameterName (int index)

{

switch (index)

{

case kDryMixParam: return "dry mix";

case kWetMixParam: return "wet mix";

case kFeedbackParam: return "feedback";

case kDelayLengthParam:return "delay";

default: break;

}

return String::empty;

}

A complementary method allows external objects, including the GUI, to set the parameters.

void DelayAudioProcessor::setParameter (int index, float newValue)

{

// This method will be called by the host, probably on the audio thread, so

// it's absolutely time-critical. Don't use critical sections or anything

// UI-related, or anything at all that may block in any way!

switch (index)

{

case kDryMixParam:

dryMix\_ = newValue;

break;

case kWetMixParam:

wetMix\_ = newValue;

break;

case kFeedbackParam:

feedback\_ = newValue;

break;

case kDelayLengthParam:

delayLength\_ = newValue;

delayReadPosition\_ = (int)(delayWritePosition\_ - (delayLength\_ \* getSampleRate())

+ delayBufferLength\_) % delayBufferLength\_;

break;

default:

break;

}

}

The arguments to this method define which parameter to set and the new value it should take. In many cases, where the parameter is used directly by processBlock(), it will suffice to set the instance variable to the new value here as seen in the first three parameters.

In some cases, changing a parameter affects other aspects of the internal state of the plug-in. In the case of changing the delay length, processBlock() does not make direct use of the delayLength\_ variable but rather derives its delay from the difference between read and write pointers. Therefore, when the delay is changed, the read pointer is updated relative to the write pointer to achieve the new delay. It is important that the read pointer moves while the write pointer stays fixed; this avoids discontinuities being written into the delay buffer.

It is possible for setParameter() to be called at the same time processBlock() is running. In certain cases, including recalculating filter coefficients or reallocating buffers, changing a parameter might temporarily interfere with operations in processBlock() and could even cause a crash. This issue of thread safety is addressed at the end of the chapter.

Another pair of methods, getStateInformation() and setStateInformation(), allow the settings of a plug-in to be stored and retrieved across sessions. Further information can be found in the JUCE documentation, or your effects can simply adapt the example code to your own set of parameters.

**Clean-up**

It is important that the plug-in release all the resources it allocates when it is finished. There are two cleanup methods in JUCE plug-ins: releaseResources() and the destructor ~DelayAudioProcessor(). These are complementary to the two initialization methods. releaseResources() runs each time audio processing finishes, and it should be used to free any resources allocated in prepareToPlay(). In this particular example, no memory was allocated in prepareToPlay() and the delayBuffer\_ object will be released in the destructor, so releaseResources() is empty:

void DelayAudioProcessor::releaseResources()

{

// When playback stops, you can use this as an opportunity to free up any

// spare memory, etc.

// The delay buffer will stay in memory until the effect is unloaded.

}

The destructor runs once when the plug-in is removed by the host environment. Anything allocated in the constructor should be released here. C++ objects like delayBuffer\_ which have their own constructors will be released automatically, hence this method is also empty in the example. However, if your effect contains any new or malloc() statements in the constructor, they must be balanced here by a delete or free() statement.

DelayAudioProcessor::~DelayAudioProcessor()

{

}

### PluginEditor.h

PluginEditor.h contains the declaration of the DelayAudioProcessorEditor class which creates a graphical user interface for the plug-in:

class DelayAudioProcessorEditor : public AudioProcessorEditor,

public SliderListener,

public Timer

{

public:

DelayAudioProcessorEditor (DelayAudioProcessor\* ownerFilter);

~DelayAudioProcessorEditor();

//==============================================================================

// This is just a standard Juce paint method...

void timerCallback();

void paint (Graphics& g);

void resized();

void sliderValueChanged (Slider\*);

private:

Label delayLengthLabel\_, feedbackLabel\_, dryMixLabel\_, wetMixLabel\_;

Slider delayLengthSlider\_, feedbackSlider\_, dryMixSlider\_, wetMixSlider\_;

ScopedPointer<ResizableCornerComponent> resizer\_;

ComponentBoundsConstrainer resizeLimits\_;

DelayAudioProcessor\* getProcessor() const

{

return static\_cast <DelayAudioProcessor\*> (getAudioProcessor());

}

};

The first line declares the class name. Following the colon is a list of classes that DelayAudioProcessorEditor inherits from. A complete discussion of inheritance can be found in any C++ text, and in most cases your own plug-ins do not need to change this list. However, it is worth highlighting the SliderListener class. DelayAudioProcessorEditor contains JUCE Slider controls, and whenever these change value, they send a message to the object that created them. Including SliderListener in the list of parent classes is necessary for these messages to be received. If your plug-in editor uses other types of controls, additional parent classes may be needed. For example, using ComboBox controls may require the inclusion of ComboBox::Listener in the parent class list; this can be seen in the accompanying example code for phase vocoder effects.

The next lines declare the methods of DelayAudioProcessorEditor, beginning with the constructor and destructor. Like DelayAudioProcessor, these will be run once when the plug-in is loaded and removed by the host environment. The other methods are standard to JUCE and usually do not need to be changed. However, if additional types of controls are used, new methods may need to be declared here (e.g. comboBoxChanged() when ComboBox controls are used). The JUCE documentation contains a complete list of controls and the methods they require.

Following the private: line is a list of instance variables for this class. The Label and Slider types define text labels and user-adjustable sliders, respectively. Notice that each of the four parameters in PluginProcessor.h has both a text label (to identify it to the user) and a slider (to let the user adjust it). We will see in PluginEditor.cpp how adjusting the sliders results in updates to the parameters.

The remainder of the file is standard for all JUCE plug-ins and does not need to be changed, with the caveat that if you change the name of the DelayAudioProcessor class, it should be updated accordingly here.

### PluginEditor.cpp

PluginEditor.cpp implements the user interface whose methods and variables are declared in PluginEditor.h. The interface this file creates can be seen in Figure 13.5. The activities of PluginEditor.cpp can be divided into four categories: initialization, managing parameters, handling resizing and cleanup. We will consider each in turn.

**Initialization**

Initialization in PluginEditor.cpp takes place in the constructor, and is often more elaborate that the initialization in PluginProcessor.cpp. The initialization needs to define the meaning and ranges of all the on-screen controls. Note that as an alternative to manually writing the code for the interface, the Projucer offers a tool for creating GUI layouts graphically which automatically generates code equivalent to the example below.

DelayAudioProcessorEditor::DelayAudioProcessorEditor (DelayAudioProcessor\* ownerFilter)

: AudioProcessorEditor (ownerFilter),

delayLengthLabel\_("", "Delay (sec):"),

feedbackLabel\_("", "Feedback:"),

dryMixLabel\_("", "Dry Mix Level:"),

wetMixLabel\_("", "Delayed Mix Level:")

{

// Set up the sliders

addAndMakeVisible (&delayLengthSlider\_);

delayLengthSlider\_.setSliderStyle (Slider::Rotary);

delayLengthSlider\_.addListener (this);

delayLengthSlider\_.setRange (0.01, 2.0, 0.01);

addAndMakeVisible (&feedbackSlider\_);

feedbackSlider\_.setSliderStyle (Slider::Rotary);

feedbackSlider\_.addListener (this);

feedbackSlider\_.setRange (0.0, 0.995, 0.005);

addAndMakeVisible (&dryMixSlider\_);

dryMixSlider\_.setSliderStyle (Slider::Rotary);

dryMixSlider\_.addListener (this);

dryMixSlider\_.setRange (0.0, 1.0, 0.01);

addAndMakeVisible (&wetMixSlider\_);

wetMixSlider\_.setSliderStyle (Slider::Rotary);

wetMixSlider\_.addListener (this);

wetMixSlider\_.setRange (0.0, 1.0, 0.01);

delayLengthLabel\_.attachToComponent(&delayLengthSlider\_, false);

delayLengthLabel\_.setFont(Font (11.0f));

feedbackLabel\_.attachToComponent(&feedbackSlider\_, false);

feedbackLabel\_.setFont(Font (11.0f));

dryMixLabel\_.attachToComponent(&dryMixSlider\_, false);

dryMixLabel\_.setFont(Font (11.0f));

wetMixLabel\_.attachToComponent(&wetMixSlider\_, false);

wetMixLabel\_.setFont(Font (11.0f));

// add the triangular resizer component for the bottom-right of the UI

addAndMakeVisible(resizer\_ = new ResizableCornerComponent (this, &resizeLimits\_));

resizeLimits\_.setSizeLimits(370, 140, 500, 300);

// set our component's initial size to be the last one that was stored in the filter's settings

setSize(ownerFilter->lastUIWidth\_,

ownerFilter->lastUIHeight\_);

startTimer(50);

}

Similar to PluginProcessor.cpp, each method name is preceded by DelayAudioProcessorEditor:: to make clear that it implements a method declared in PluginEditor.h. Also like PluginProcessor.cpp, an initialization list follows the declaration of the constructor. Here, this list is used to initialize the parent class AudioProcessorEditor defined by JUCE and the values displayed in the four text labels (delayLengthLabel\_, feedbackLabel\_, dryMixLabel\_, wetMixLabel\_). Your plug-ins will not need to change the AudioProcessorEditor line, but depending on the parameters your plug-in uses, the number and value of labels may change.

Within the constructor, the first task is to initialize the four sliders which control each parameter. Each slider contains four lines of initialization, for example:

addAndMakeVisible (&feedbackSlider\_);

feedbackSlider\_.setSliderStyle (Slider::Rotary);

feedbackSlider\_.addListener (this);

feedbackSlider\_.setRange (0.0, 0.995, 0.005);

The first line adds the slider to the editor. The second causes the slider to display a rotary (rather than linear) control. The third line tells the slider to notify the editor whenever its value changes; the value this is a C++ keyword meaning the current object (here, DelayAudioProcessorEditor). The final line sets the minimum, maximum and granularity of the slider’s range. In this case, the slider is allowed to range from 0.0 to 0.995 in increments of 0.005. Notice that this is the feedback control, so its value is limited to strictly less than 1 to avoid instability. Your plug-ins will most likely declare different ranges and granularities for each control.

The next task of the constructor is to attach the text labels to each slider:

feedbackLabel\_.attachToComponent(&feedbackSlider\_, false);

feedbackLabel\_.setFont(Font (11.0f));

The label text itself was set in the initialization list. These lines ensure the label appears in the right place and that it has the correct font. In your plug-ins, you should have a similar pair of lines for each Label object.

The final lines handle the resizing of the editor window itself and the initialization of a timer that keeps the user interface synchronized with the parameter values even when the parameters are changed externally. The only line that may need to be changed in other plug-ins is this one:

resizeLimits\_.setSizeLimits(370, 140, 500, 300);

This line defines the minimum and maximum size of the window in pixels. Here, the window can range from 370x140 at its smallest to 500x300 at its largest. The right size to use for any given plug-in will depend on the number of controls in the window.

**Managing parameters**

PluginEditor.cpp implements two methods for keeping the on-screen controls synchronized with the values of each parameter in the DelayAudioProcessor object. The first method queries the current parameter values and changes the user interface to match:

void DelayAudioProcessorEditor::timerCallback()

{

DelayAudioProcessor\* ourProcessor = getProcessor();

delayLengthSlider\_.setValue(ourProcessor->delayLength\_, dontSendNotification);

feedbackSlider\_.setValue(ourProcessor->feedback\_, dontSendNotification);

dryMixSlider\_.setValue(ourProcessor->dryMix\_, dontSendNotification);

wetMixSlider\_.setValue(ourProcessor->wetMix\_, dontSendNotification);

}

In the constructor, a timer was started which causes this method to be called by the system every 50 milliseconds. It is possible for the plug-in parameter values to be changed by some means other than the user interface, for example by MIDI messages. This method ensures that the user interface always shows the actual current values of the parameters. One consequence of the timerCallback() method is that if in your plug-in, you forget to save the values of parameter changes, either in DelayAudioProcessorEditor::sliderValueChanged() or in DelayAudioProcessor::

setValue(), you will find that every time you change a control, it immediately snaps back to its original value.

The second parameter management method is called every time the user changes the value of a slider:

void DelayAudioProcessorEditor::sliderValueChanged (Slider\* slider)

{

// It's vital to use setParameterNotifyingHost to change any parameters that are automatable

// by the host, rather than just modifying them directly, otherwise the host won't know

// that they've changed.

if (slider == &delayLengthSlider\_)

{

getProcessor()->setParameterNotifyingHost (DelayAudioProcessor::kDelayLengthParam,

(float)delayLengthSlider\_.getValue());

}

else if (slider == &feedbackSlider\_)

{

getProcessor()->setParameterNotifyingHost (DelayAudioProcessor::kFeedbackParam,

(float)feedbackSlider\_.getValue());

}

else if (slider == &dryMixSlider\_)

{

getProcessor()->setParameterNotifyingHost (DelayAudioProcessor::kDryMixParam,

(float)dryMixSlider\_.getValue());

}

else if (slider == &wetMixSlider\_)

{

getProcessor()->setParameterNotifyingHost (DelayAudioProcessor::kWetMixParam,

(float)wetMixSlider\_.getValue());

}

}

This method first queries which Slider object generated the call. It then queries the current value of the slider and updates the audio processor object accordingly. Notice that this method uses the parameter indices declared in the enum statement inside PluginProcessor.h. In your plug-ins, you should have one if statement for each slider, and each slider should correspond uniquely to one of the enum values. The setValueNotifyingHost() method is provided by JUCE and will ultimately result in a call to the setValue() method declared in PluginProcessor.cpp.

If your plug-in uses other types of controls besides sliders, an analogous method to sliderValueChanged() will need to be implemented for each type of control

**Resizing**

Depending on the values provided to resizeLimits\_.setSizeLimits() in the constructor, the user may be able to change the size of the editor window. Whenever the window size changes (and at least once on startup), the resized() method will be called:

void DelayAudioProcessorEditor::resized()

{

delayLengthSlider\_.setBounds (20, 20, 150, 40);

feedbackSlider\_.setBounds (200, 20, 150, 40);

dryMixSlider\_.setBounds(20, 80, 150, 40);

wetMixSlider\_.setBounds(200, 80, 150, 40);

resizer\_->setBounds(getWidth() - 16, getHeight() - 16, 16, 16);

getProcessor()->lastUIWidth\_ = getWidth();

getProcessor()->lastUIHeight\_ = getHeight();

}

This method is responsible for placing all the sliders in the appropriate locations and informing the plug-in processor of the new window size. The pixel locations may have to be worked out experimentally to make the interface look right. Alternatively, the Projucer can be used to graphically construct an interface which automates some of the processes in PluginEditor.cpp.

**Cleanup**

When the plug-in is removed by the host environment, the destructor will be called. Any objects that were manually allocated using new or malloc() need to be released with delete or free(), respectively. In this case, no such allocation has taken place so the destructor is empty.

DelayAudioProcessorEditor::~DelayAudioProcessorEditor()

{

}

### Summary

This section has examined the code for an example plug-in that implements a delay with feedback. Four source files were required, two related to the audio processor and two related to the user interface. The most important task for the audio processor is to implement the audio callback function processBlock(), which the system calls every time it needs a new block of audio. Since audio is processed in small blocks, it is necessary to remember the state of the effect between callbacks, so instance variables were declared in PluginProcessor.h to record information that persists beyond the duration of processBlock().

The other important task was the management of plugin parameters, which required careful coordination between all four files. Implementing parameters required an ordered list of parameter indices to be declared in PluginProcessor.h alongside instance variables to hold their values. Controls to change their values were declared in PluginEditor.h. Both PluginProcessor.cpp and PluginEditor.cpp implemented methods for getting and setting parameters, which were ultimately used by the processBlock() method in PluginProcessor.cpp to render the effect.

## Advanced topics

### Efficiency considerations

For more complex effects, it is useful to consider how the audio processing can be most efficiently implemented. This is especially true for effects on mobile or embedded devices, but computational cost is even a concern for the fastest of computers when many effects are used together or the audio processing is highly computational. A standard metric of computational cost used in digital signal processing is multiplies per sample: how many multiplications does the computer have to perform for each audio sample? Though it is not a perfect metric of processing complexity, it provides a useful overall guide to how an effect can be made more efficient.

A straightforward example of considering multiplies per sample is filtering. An FIR filter of length *N* will require *N* multiplies per sample to implement; for long filters, this can become quite expensive. A particularly troublesome example is convolutional reverb where the input signal must be convolved with an impulse response that could potentially exceed 100,000 samples. Aside from shortening the filter, a variety of tricks are possible to improve efficiency using the Fast Fourier Transform. These are further discussed in Chapter 11.

Another example of managing computational complexity can be found in the wah-wah effect. Here, the effect is generated by a second order IIR filter whose coefficients change whenever the user adjusts a control. Applying the filter is inexpensive, requiring only 5 multiplications per sample. However, calculating the coefficients involves a complicated formula including trigonometric functions which require many more multiplications to implement. It is not efficient to recalculate the coefficients of the filter at each sample based on the current value of the cutoff frequency. Instead, a more efficient approach will recalculate the coefficients only when they are changed by the user, typically within the setValue() method, and save these values for future use by the callback. This significantly reduces calculations but has the potential to introduce a subtle bug, discussed in the next section.

A third example is the calculation of low-frequency oscillators (LFOs). In an auto-wah effect, the cutoff frequency of the filter changes continuously over time under the control of an LFO. It might appear that recalculating coefficients every sample is necessary. However, with most LFOs, the change in value from one audio sample to the next is so subtle that the filter coefficients can be updated less frequently, for example every 16 or 32 samples, with no audible change in performance.

A final example concerns the use of complex mathematical functions. Suppose a phaser effect wanted to implement a complicated exponential LFO waveform to vary the location of the allpass filters. Calculating the value of this function each sample might require a large number of multiplies since internally, exponential functions are much more complex to calculate than basic multiplications. However, it might be possible to generate a lookup table with all the values of the function pre-calculated and stored in a block of memory. Then on each sample, rather than running the calculation, the callback function could simply access the table to determine its value. If greater precision is needed, interpolation between values in the lookup table could be used while still maintaining a significant efficiency improvement over raw calculations. See Chapter 2 for further details on interpolating within buffers.

### Thread Safety

Most audio plug-ins are multi-threaded. This means that two different methods could be running simultaneously in different threads of execution, with no guarantee that one method finishes before the other begins. In particular, the thread that runs the audio callback function is often not the same thread that handles changing parameters. This results in the potential for subtle bugs and even outright crashes.

Consider the case of a phase vocoder plug-in which needs to gather samples in a buffer before performing a Fast Fourier Transform. Suppose that the user changes the window size of the effect using one of the graphical controls. When this happens, a call to setParameter() will be generated, and because the window size has changed, the plug-in may reallocate the window buffer to hold a different number of samples. In a multi-threaded environment, the call to setParameter() might happen at the same time as the callback processBlock() is running. The callback function will be unaware that the buffer is about to change size, so it may attempt to access an invalid index and cause the plug-in to crash, potentially bringing down the whole host environment. Similar problems may occur on recalculating coefficients for filters and other situations where memory is allocated or deallocated while the effect is running.

One solution is to ensure that only one of setParameter() and processBlock() can run at the same time. If processBlock() is running and setParameter() wants to change a buffer size, it must wait until processBlock() finishes, and the same is true in reverse of setParameter() begins first. This behavior can be achieved using a special variable type called a mutex (short for mutually exclusive). When one thread locks the mutex, the other thread must wait until it has been unlocked before continuing. Consider the following example code from a phase vocoder effect. First, in PluginProcessor.h, the mutex variable is defined within the audio processor class:

// Spin lock prevents the FFT settings from changing in the middle of the audio thread.

SpinLock fftSpinLock\_;

SpinLock is a type of lock object provided by JUCE. At the beginning of processBlock(), the method acquires the lock before beginning its processing and releases it upon completion:

void PVOCPassthroughAudioProcessor::processBlock (AudioSampleBuffer& buffer, MidiBuffer& midiMessages)

{

// Helpful information about this block of samples:

const int numInputChannels = getNumInputChannels(); // How many input channels for our effect?

const int numOutputChannels = getNumOutputChannels(); // How many output channels for our effect?

const int numSamples = buffer.getNumSamples(); // How many samples in the buffer for this block?

int channel, inwritepos, sampsincefft;

int outreadpos, outwritepos;

// Grab the lock that prevents the FFT settings from changing

fftSpinLock\_.enter();

// [...]

// \*\*\*\* Phase Vocoder processing goes here

// [...]

fftSpinLock\_.exit();

}

Meanwhile, any call that changes the window size or invalidates the internal buffers will also acquire the lock. If one thread holds the lock and a second tries to acquire it, the second thread blocks (waits) until the lock has been released. This ensures that no two threads act on the same resources in incompatible ways.

Multi-threaded programming can be quite subtle and challenging, particularly as the number of threads and number of shared resources increase. Several good texts exist on the subject, e.g., [98].

## Conclusion

This chapter has shown how to put the principles of audio programming into practice through the creation of audio plug-ins. The JUCE framework is a convenient environment for creating audio plug-ins since it allows the same code to be compiled into multiple plug-in formats on multiple operating systems. However, the general principles of writing audio callback functions, managing parameters, allocating resources and creating a user interface are similar across many plug-in formats. The example code included with the book offers templates from which you can start building your own effects, including examples from most chapters and a ‘Hello World’ plug-in into which you can add your own code.

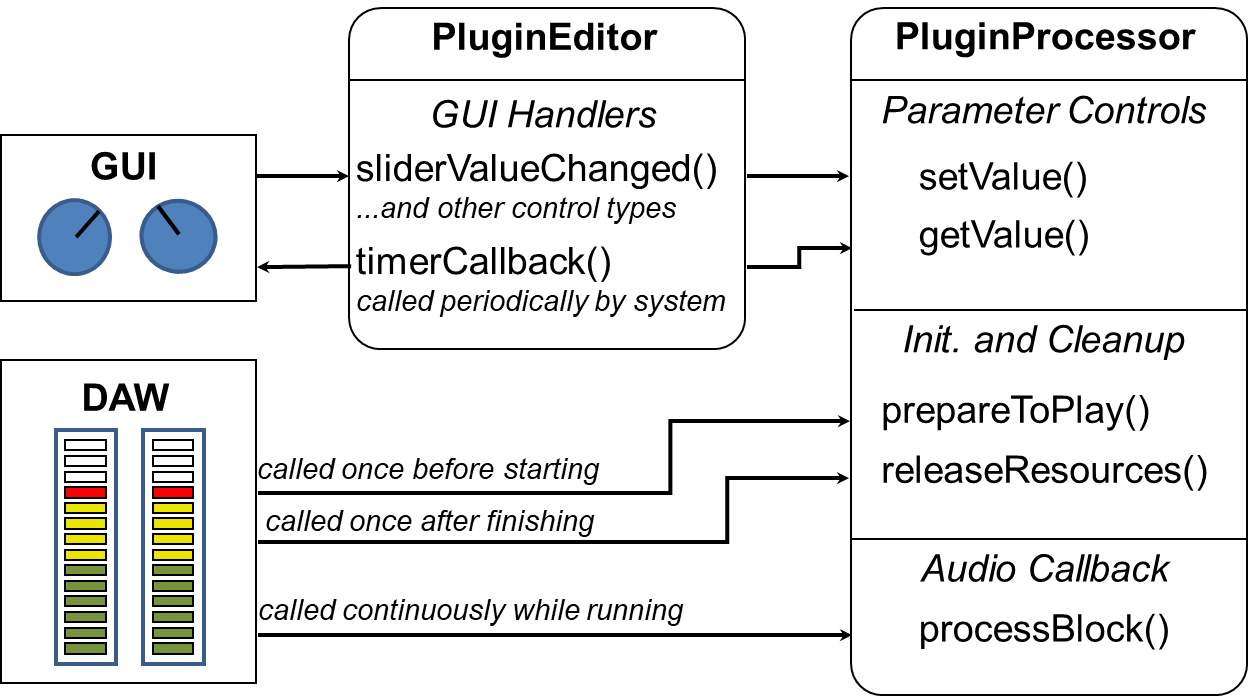


Figure .. Basic components of a JUCE audio plugin and their relationship to the digital audio workstation (DAW).

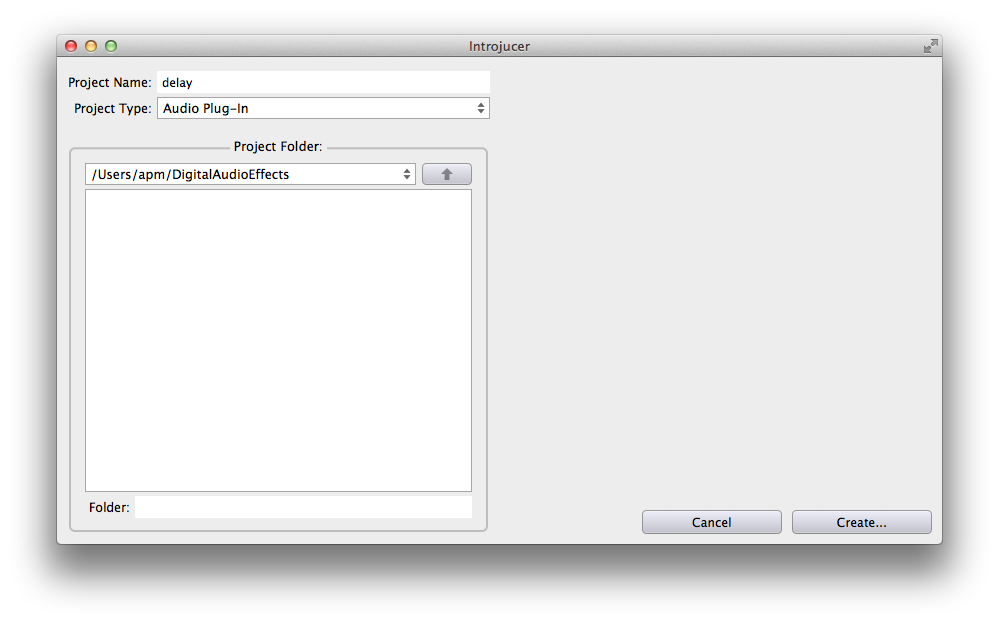


Figure .. Creating a new audio plug-in in Projucer.

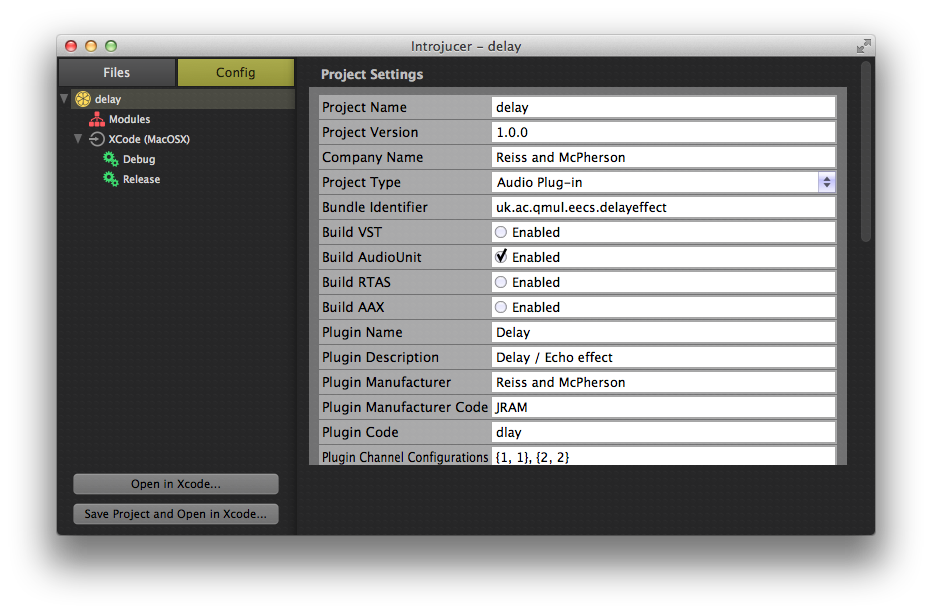


Figure .3. Projucer settings for audio plug-in. List at left shows a single build target for Xcode.

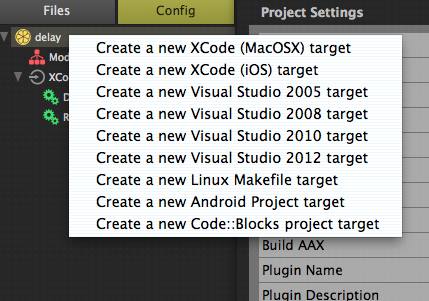


Figure .4. Right-clicking on the plug-in icon produces a menu for generating additional build targets.

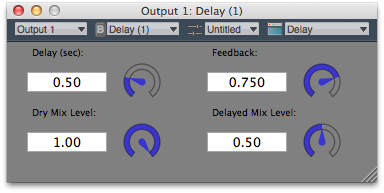


Figure .5. Editor (user interface) for example delay plug-in.

[94] B. Stroustrup, *The C++ Programming Language*: Addison Wesley, 2000.

[95] A. Koenig and B. E. Moo, *Accelerated C++*: Addison Wesley, 2000.

[96] S. B. Lippman*, et al.*, *C++ Primer*, 5th ed.: Addison Wesley, 2012.

[97] R. Boulanger*, et al.*, *The Audio Programming Book*: MIT Press, 2010.

[98] M. Walmsley, *Multi-Threaded Programming in C++*: Springer 1999.