### 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? Initialization will be considered in a later section, but the question of maintaining values across callbacks is crucial to making the effect work. 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:

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. Notice that 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 Introjucer 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::

setParameter(), 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 setParameterNotifyingHost() method is provided by JUCE and will ultimately result in a call to the setParameter() 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, JUCE’s Grid or FlexBox classes can be used to construct an interface which does not rely on hardcoded locations.

**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()

{

}