



presents

The Complete Beginner's Guide to Audio Plug-in Development

by Matthijs Hollemans

The Complete Beginner's Guide to Audio Plug-in Development

Matthijs Hollemans



Published by The Audio Programmer
www.theaudioprogrammer.com

Copyright © 2024 The Audio Programmer
All rights reserved

Contents

Foreword by Joshua Hodge	1
Where to get the source code	3
About this book	4
1: Introduction	5
What you will build	6
Who this book is for	8
What you need	9
A note about software versions	10
Why JUCE?	11
Why C++?	13
Get ready to give your brain a workout	14
What about math?	16
The source code	16
2: How plug-ins work	18
Effects versus synths	19
Plug-in formats	20
Communicating with the host	21
Plug-in parameters	24
Introduction to digital audio	26
The sampling rate	29
A closer look at samples	31
Quantization and bit depth	35
Processing audio in the plug-in	36
Audio blocks and buffers	37
Don't miss that deadline!	39
How plug-ins are made	41
What is an IDE?	43

Cross-platform development	44
3: JUCE and Projucer	45
Downloading JUCE	46
Setting the JUCE path (Mac)	48
Setting the JUCE path (Windows)	49
The JUCE folder	51
Playing with the JUCE demos	52
The plug-in project	55
Creating the project	58
Getting to know Projucer	60
Configuring the project	63
4: Getting started with Xcode	67
Required versions	67
Installing Xcode	68
Method 1: Using the Mac App Store	68
Method 2: Downloading from Apple's developer website	70
First time running Xcode	71
Exporting the Projucer project	72
A quick tour of Xcode	74
Compilation targets	76
Compiling the plug-in	77
Where is that plug-in?	81
Intel versus Apple Silicon	83
5: Getting started with Visual Studio	85
Required versions	85
Installing Visual Studio	86
First time running Visual Studio	88
Exporting the project	90
A quick tour of Visual Studio	92
Compilation targets	94
Compiling the plug-in	95
Warnings and errors	98
Where is that plug-in?	101
6: Using a host for testing	103
Using a DAW	103
Making your first changes to the code	106

Getting to know C++	109
AudioPluginHost	114
Creating a session	116
Playing audio files (Mac)	119
Playing audio files (Windows)	120
Automatically starting AudioPluginHost	123
The JUCE splash screen (JUCE 7 only)	126
7: Output gain	129
The audio processor	130
Inside PluginProcessor.cpp	133
Applying gain	135
Using loops	139
A closer look at processBlock	144
Stereo only	146
Decibels	148
8: Plug-in parameters	150
Improving the parameters	156
Saving the parameter values	160
Creating a Parameters class	163
Improving the class	169
Zipper noise and parameter smoothing	173
9: Adding the delay	180
The delay line	180
Using juce::dsp::DelayLine	182
Putting the delay line into action	187
Setting the delay time with a parameter	189
Improving the parameter	191
Parameter smoothing, redux	195
Dry/wet mix	202
Testing the delay effect	205
10: The user interface	208
The editor	209
Adding a rotary knob	213
Adding a label	215
Connecting the slider to the parameter	216
Making a RotaryKnob class	218

Adding the other knobs	225
Grouping the knobs	228
Value-from-string functions	232
11: Styling the editor	235
Defining the colors	236
Look-and-feel for the slider	238
Drawing the knob	242
Drawing the track	247
Drawing the dial	250
Coloring the track	253
Fonts	257
Styling the group components	262
Fixing a bug	264
Styling the text boxes	267
Drawing images	272
12: Feedback	278
Multi-tap delays	278
The block diagram	281
Protect your ears	283
Adding the feedback loop	287
Negative feedback	292
13: Ping-pong delay	296
Exploring stereo	297
Ping-pong	298
Stereo width	301
What about mono inputs?	307
The Stereo parameter in mono mode	315
14: Filters	317
What is a filter?	317
JUCE filter classes	320
Adding the parameters	321
Filter that sound	325
More efficient parameter updates	328
Block-based processing	331
15: Tempo sync	334
Setting the note length	334

Getting the tempo information	340
Converting note lengths to delay time	344
Calculating the delay length	346
Styling the button	349
Hiding the Time knob	353
16: Diving deeper into DSP	360
The DelayLine class	360
Why a circular buffer?	361
Implementing the circular buffer	365
Replacing the JUCE DelayLine	371
Interpolation	373
Hermite interpolation	379
More about wrapping around	382
Unit tests	384
17: Adding a level meter	386
Audio thread vs. UI thread	386
Measuring the peak levels	387
The LevelMeter component	390
Drawing the meter	394
Drawing the levels	398
One-pole filter for smoothing	402
Improving the measurements	405
18: Sweating the details	410
Fixing the delay time knob artifacts	410
Ducking	417
Adding a bypass button	423
About presets and programs	427
Using the debugger	428
19: Releasing your plug-in	435
Making a release build	435
Testing with pluginval	438
Testing on different hosts	439
Performance testing	440
Improving real-time performance	442
Signing the plug-in (Mac)	443
Signing the installer (Windows)	449

Building for iOS	450
Licensing and legal stuff	453
20: Where to go from here	454
Ideas for new features	454
Accessibility	456
Further reading	457
The end beginning	464

Foreword by Joshua Hodge

“I want to create my own plug-in! How do I get started if I’ve never written a line of code?”

If you’re asking this question, then this book is for you. In fact, it’s a question that comes up nearly every day in our Audio Programmer Community! The challenge is that the answer always starts with “*It depends!*” and inevitably leads to more questions...

- *“Ok, I’ve downloaded Xcode. What do I do next?”*
- *“It says ‘Build Failed’ and I don’t know how to fix it.”*
- *“How do I test my plug-in?”*
- *“What is threading and why is it important?”*

There wasn’t a journey or resource we could point towards and say “*Start here...*”

...until now.

The Complete Beginner’s Guide to Audio Plug-in Development is a foundational guide for anyone looking to get from “0 to 1” in creating their first audio plug-in.

It’s also the starting point to gaining a true understanding about what you’re actually doing (and more importantly, why you’re doing it)! We also introduce some of the basic terminology that you didn’t know but were afraid to ask — IDE, linking, buffers, API, class inheritance, and so on.

I’m especially excited about this book because **I know** this will be the starting point for future audio software developers. Matthijs Hollemans has concisely written this book with the true beginner in mind — no assumptions are made about where you’re starting from... except that you know the basics of using a computer, of course!

One way this book has been beneficial for me personally is that it's helped me to fill in some blank spots in my own knowledge base... what some of those other sections in Xcode actually mean, a better understanding of the life cycle of a plug-in, how file structures differ between Windows and Mac operating systems and much more!

I'm truly grateful for the amazing community that we've created together, and proud to serve as its custodian. If you haven't already joined us, we would love to meet you and hear about your journey! Join us at theaudioprogrammer.com/community¹.

Welcome to the world of audio programming!

Kindly,
Joshua Hodge
Founder, The Audio Programmer

¹<https://theaudioprogrammer.com/community>

Where to get the source code

The source code that accompanies this book can be found on GitHub:

[github.com/TheAudioProgrammer/getting-started-book²](https://github.com/TheAudioProgrammer/getting-started-book)

To download, click the big green **Code** button and choose **Download ZIP**.

The **Resources** folder contains images and sound files that you will need to follow along with this book.

For each chapter there is a folder with the finished source code from that chapter. This is useful for verifying that you didn't make any mistakes when typing in the code from the book. If the code isn't doing what you think it should, compare it to the downloaded version to see if you missed something.

The **Finished Project** folder contains the completed plug-in in its entirety.

Found a bug or have a question? The GitHub link is also a good place to go for support. It has a list of errata and known issues. If you can't find your question in the errata, go to the **Issues** tab on the GitHub page and report a new issue. Thanks!

The source code from this book is licensed under the terms of the MIT license. This means you may use the source code in your own projects but at your own risk.

²<https://github.com/TheAudioProgrammer/getting-started-book>

About this book

Hi, my name is Matthijs Hollemans and I'm a professional audio software developer and the author of this book. Previously I have been employed as a machine learning engineer and iOS developer.

You can find my open source projects on [GitHub³](#). I recently started a new [blog about audio development⁴](#) that you might find interesting as well.

I wish to thank Joshua Hodge from The Audio Programmer for commissioning me to write this book, it was a fun project!

Also many thanks to the members of The Audio Programmer Discord server for lots of interesting discussions, and for helping me become a better audio programmer.

Cheers to all of you DSP fanatics out there sharing knowledge and code snippets through discussion forums, chat groups, mailing lists, blogs, videos, and books. May this great tradition continue forever!

Credits:

- JUCE is copyright © Raw Material Software.
- The Lato font is copyright (c) 2010-2014 Łukasz Dziedzic⁵ and is licensed under the SIL Open Font License.
- C++ logo by [Jeremy Kratz⁶](#).

Enjoy the book, it was a lot of fun to write!

— Oosterhout, Netherlands, spring of 2024

³<https://github.com/hollance>

⁴<https://audiodev.blog>

⁵<http://lukaszdziedzic.eu>

⁶<https://github.com/isocpp/logos>

1: Introduction

Welcome to the fun and exciting world of audio programming! In this book you'll learn how to make an audio plug-in, also known as a VST — even if you have never written a line of computer code in your life.

This is a tutorial-style book that goes step-by-step and explains everything you need to know along the way. It is written for complete beginners. No prior programming knowledge is required and there is not a lot of math involved.

The goal of this book is to make audio programming accessible to everyone — without requiring a computer science degree, audio engineering degree, or any other qualifications. All you need is a functioning brain, an eagerness to learn, and a little patience. The book does the rest by giving detailed explanations for everything, including as many screenshots and illustrations as possible.

The programming language you will be using is C++. To handle the low-level audio stuff, you'll use the JUCE framework. Since you may not know how to use either of these tools yet, we'll start at the very beginning. I can't promise that you'll be a C++ genius after finishing this book, but you'll have learned enough to start writing your own plug-ins.

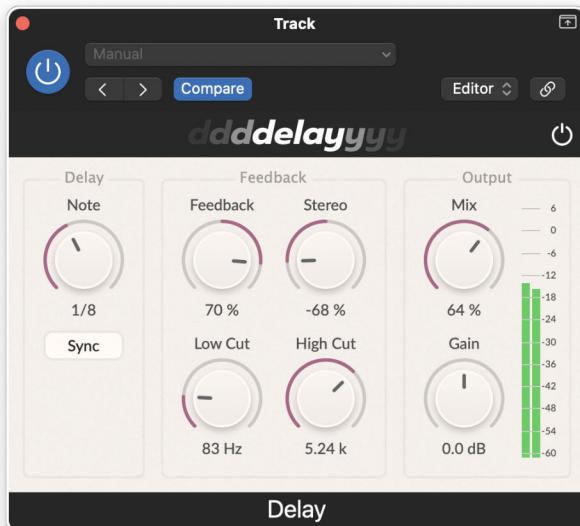
The combination of C++ and JUCE is the industry standard way of making plug-ins, so whether you're just looking to learn as a hobby or to get a job, you've come to the right place! This is what the pros use, and exactly how commercial plug-ins are made.

Creating a plug-in from the ground up is an excellent way to get familiar with audio software development. The same programming techniques are also used in other kinds of audio software, so this book will be useful even if plug-ins aren't your thing.

Let's get started!

What you will build

This is the audio plug-in that you will learn how to make:



The finished Delay plug-in

The plug-in is shown running in Logic Pro but it will work in other DAWs such as Ableton Live, REAPER, FL Studio, Cubase, and any other host application that supports VST3 or Audio Unit plug-ins.

As you may have guessed from its name, this is a delay plug-in. It will take the incoming audio and repeat it after a set amount of time to create a series of echoes. Used tastefully, delay can add extra depth and ambiance to a recording. Delay plug-ins are an invaluable tool for the music producer and are used everywhere in modern mixes, from vocals to guitar, on synths, and even on drums.

Our delay plug-in has the following features:

- The delay time can be set anywhere from 5 milliseconds to 5 seconds, but also can be synced to the tempo of the music.
- There is a feedback section that causes the echoes to keep repeating while slowly fading them out. A low-cut and high-cut filter in the feedback path allow you to restrict the feedback to a certain frequency range.

- When used on stereo sound, the plug-in acts like a ping-pong delay, bouncing the echoes between the left and right channels with a configurable amount of stereo width.
- The delayed sound can be dry/wet mixed into the original sound. There is an output gain control for setting the overall sound level.
- A level meter shows the plug-in's output levels in real time.
- A bypass button toggles the entire plug-in off or back on.

If you've ever used a delay plug-in before, these features should look very familiar as they form the basis of pretty much any delay ever made.

Over the course of this book, we'll look at every single piece of code that makes up this plug-in, and you'll learn what it does, how it works, and why it's done that way.

When you're finished with the book, you will have a complete, professional sounding delay to liven up your mixes. But that's not all: You can reuse parts of the source code in your own plug-ins. For example, you could integrate the ping-pong delay into the effects section of a synth.

Yawn... a delay plug-in?!

Perhaps the idea of creating a delay plug-in sounds a little underwhelming to you. True, it's not the flashiest, most awe-inspiring plug-in ever. But as a fledgling audio developer you need to start somewhere and this is a great place to begin your journey.

Building a delay plug-in is not too extreme for a beginner to handle but it's also not trivial — the difficulty level is just right.

The aim of this book is to show all the steps involved in plug-in development, so that by the end of the book you'll be ready to start writing your own, way more awesome plug-ins! The knowledge you'll pick up here forms the foundation for anything else you might want to do with audio programming.

For example, you'll learn about the delay line building block. Delay lines are not only used in pure delay plug-ins, but in all kinds of other effects too — such as flanger, chorus, reverb, pitch shifting, physical modeling, filters, and many more.

All the theory and every technique you'll learn in this book are things you'll need time and again when making your own plug-ins, no matter what kind of amazing plug-in idea you have in mind.

There is a lot going on under the hood of this delay plug-in that you may not realize at first. After all, it takes me almost 500 pages to explain how it all works!

Just to give an example: When the Time knob is turned while audio is playing, a curious pitch-shifting effect happens. Effectively, time is being stretched, causing the delayed signal to be temporarily played back faster or slower until it catches up with the clock again.

We will investigate a few ways to avoid this phenomenon. In audio programming there are many such details that may not be immediately obvious, but you'll still need to deal with them to make the plug-in sound good. Even a plug-in that appears to be simple can get complicated on the inside.

Make no mistake, we're going to be doing some real digital signal processing (DSP) in this book! But you'll have to start at the very beginning and work your way up from there, and a delay plug-in is an excellent vehicle for that.

Who this book is for

If the following describes you, then this book should be right up your alley:

- You have never written a line of computer code in your life.
- You do have some experience using a DAW (digital audio workstation) or other audio software.

The ideal reader for this book is an absolute newbie at software development. Of course, if you already know a bit of computer programming (or even a lot) you can still read this book to learn about digital audio.

Just know that it's perfectly OK if you've never seen the inside of an IDE before — or don't even know what an IDE is! All will be explained as you work through the book.

Music-making experience is useful, but not an absolute prerequisite. However, since you're wanting to learn how to make plug-ins, it's probably safe to assume you are already somewhat familiar with using them.

What you need

The two main technologies you'll be using in this book are:

1. the C++ programming language
2. the JUCE audio framework

To use C++ you will need an IDE, which stands for Integrated Development Environment. The IDE is the editor that lets you type in the source code for the plug-in.

The IDE also includes the compiler that converts the source code into something the computer can actually run. And it has a debugger to help track down programming mistakes (bugs). You will be spending most of your time working inside the IDE — it's the place that software developers call home.

For Mac the IDE is called Xcode, for Windows it is Visual Studio. Both the IDE and JUCE must be downloaded and installed first. Fortunately, they can be downloaded for free, so you can get started at no cost. Chapters 3–5 have full instructions on how to do this.

To use the IDE, you will need a PC running Windows or a Mac. The Mac can have an Apple Silicon processor or an Intel processor, either is fine. You won't be able to write plug-ins on an iPad or another tablet, you'll need a real computer for this. It doesn't have to be a very expensive or powerful machine, any laptop or desktop computer released in the last ten years will do. It's possible to use a Linux machine but this book does not include instructions for developing on Linux.

For trying out the plug-in while you're developing it, you will be using a simple host application that is included with the JUCE framework (AudioPluginHost). You'll probably want to test the plug-in inside a proper DAW as well.

If you currently don't have a DAW, then REAPER⁷ is a powerful DAW that is free to try. On Mac you can install GarageBand from the Mac App Store for free. Another popular free DAW is Cakewalk⁸.

⁷<https://www.reaper.fm>

⁸<https://www.bandlab.com/products/cakewalk>

A note about software versions

This book was written using JUCE framework version 7.0.9 but also tested with a preview release of JUCE 8. On the Mac, I used macOS 14 Sonoma and Xcode 15.2. On my PC, I used Windows 10 and Visual Studio 2022.

One fact about software, and especially software development tools, is that it gets updated on a regular basis. Chances are that you are using a different version of JUCE, a different version of the operating system, or a different version of the IDE. This shouldn't generally be a problem... in theory (unless it's the year 2100 when you read this, you might be out of luck then).

Sometimes new software versions break things. Source code written for JUCE 7 or 8 may not work anymore in JUCE 9. Code that compiled perfectly fine with Xcode 15 may now give a warning or error message with Xcode 16. A menu option may have been renamed or a button been moved, or any myriad of other things that can change.

If the images or descriptions in the book don't look exactly like what you're seeing on your screen, don't panic! It's probably because you're using a newer software version than what the book was written for.

You may sometimes need to use your imagination a little. For example, if you're on an older version of macOS, the Settings menu item will be named Preferences instead. Same thing, different name.

If you run into strange errors with the source code from this book, and you suspect it might be because you're using a newer version of JUCE, Xcode, Visual Studio, or the operating system, please check the [errata⁹](#) on the book's website or ask on The Audio Programmer discord.

⁹<https://github.com/TheAudioProgrammer/getting-started-book>



Why JUCE?

An important characteristic of audio plug-ins is that the same plug-in can be used in many different DAWs. One user might be using Ableton Live, while another prefers to work in Studio One, and a third makes music with FL Studio. In the ideal situation, all of these people are able to load your plug-in, regardless of the particular audio software they're using.

This is accomplished by all DAWs agreeing on how plug-ins should work, known as the VST (virtual studio technology) specification. This is why plug-ins are often named VSTs. The current version of this specification is VST3.

Unfortunately, not all DAWs support the VST3 plug-in format. For example, Logic Pro requires that the plug-in is written according to Apple's Audio Unit specification. And Pro Tools has its own format called AAX. You'll need to make a different version of the plug-in for each of these formats, or certain DAWs will not recognize the plug-in and users won't appreciate that.

In addition, some users will run their DAW on Windows while others use a Mac or even are on an iPad. Fundamentally, Windows, macOS, and iOS are very different operating systems — a program for Windows cannot run on Mac, and vice versa. This means you will also need to make a separate version of the plug-in for every OS that you want to support. Ugh.

This leads to a number of possible combinations:

- Windows, VST3 format
- Windows, AAX format (for Pro Tools)
- Mac, VST3 format
- Mac, Audio Unit format (AU)

- Mac, AAX format
- iOS, AUv3 format (different from Mac AU)

...and possibly others as well, such as an LV2 plug-in for Linux. What a drag! No one wants to implement the same plug-in six or more times, to support all these DAWs, plug-in formats, and operating systems. That is where JUCE comes in.

JUCE is a programming framework, which means it provides pre-made source code that you can use in your own projects, saving you from having to write that code yourself. The big benefit of JUCE is that it allows you to write the plug-in just once, after which you can export it to VST3, Audio Units, AAX, and LV2 for the various operating systems.

JUCE hides the differences between all these plug-in formats and operating system audio APIs (application programming interfaces). Thanks to JUCE, you don't have to worry about any of that stuff so that you can focus on the fun part: writing the audio processing code. The people who created JUCE already went through the effort of making it work with all the different plug-in formats, so we don't have to. Nice!

In addition, JUCE has cross-platform support for creating user interfaces, so that you also design and build the UI only once. JUCE includes many ready-made building blocks for audio programming, such as a delay line, filters, waveshaping, panning, mixing and much more. We'll use a number of these building blocks in this book.

A framework like JUCE saves a lot of time. This is why JUCE is used by many commercial developers and is considered to be the industry standard solution for making plug-ins. JUCE also has very good support from its development team. Many people in the audio community have experience with JUCE, so it's possible to get help if you get stuck. All of these things together make JUCE the best place to get started. (And no, I didn't get paid for saying this!)

By the way, JUCE isn't just for plug-ins. You can build full-fledged desktop and mobile applications with it, including complete host applications. The [Tracktion DAW engine](#)¹⁰, for example, is written in JUCE.

There are alternatives to JUCE, such as [iPlug2](#)¹¹ or using the VST, Audio Unit, and AAX SDKs (software development kits) directly, but this book does not cover those other frameworks.

¹⁰<https://www.tracktion.com/develop/tracktion-engine>

¹¹<https://iplug2.github.io>



Why C++?

The programming language used in this book is C++, pronounced “C plus plus”.

I will be frank: C++ strikes fear into the heart of many a noble software developer. It is not an easy language, nor a universally beloved one, and the mere mention of C++ makes even experienced developers run away screaming. Needless to say, C++ has a bit of a reputation.

Why then use C++ and not a more modern or beginner-friendly language? The simple fact is that most audio programming these days is done in C++.

We could have chosen some other language such as Python, which is often used as an introductory programming language in schools and colleges, but then we wouldn’t be able to make a real plug-in. And we’re here for the real thing!

If you want to do audio programming — and plug-ins in particular — then C++ is the most suitable language, like it or not.

A lot of its reputation is well deserved, by the way. C++ has plenty of sharp edges and it’s harder to get into than other languages. Learning to write computer programs is difficult enough by itself already and C++ is definitely not the easiest language to start out with.

But that’s no reason to be scared of it! You don’t need to learn all of C++ right away, only enough to start writing plug-ins. As you get more experienced, your knowledge of C++ will naturally grow too.

I’m a big fan of teaching programming in a context that the student cares about. Learning C++ on its own is not only a major challenge, it’s also rather boring. If I told you to first go do a C++ course and come back here when you’ve mastered the language, you’d likely give up on the whole idea long before ever finishing that course. It’s just no fun to learn a demanding language like C++ in isolation.

But you're not here to learn C++, not really. You're here to learn how to make audio software and C++ is merely one of the tools you need to pick up along the way. It's much more enjoyable to absorb some C++ knowledge while you're studying the thing you actually want to learn. This is why we will do both at the same time: learn C++ while building a cool audio plug-in.

This book has been designed so you can read it without knowing anything about C++ or any kind of computer programming. All the basics will be explained while at the same time you're building a fun audio plug-in.

Obviously, this book won't be teaching the entire C++ language, but I will explain enough so that you can make sense out of what we're doing. Afterwards, if you're not scarred for life by the horrors of C++, you may want to dive deeper into C++ by taking a proper course.

Get ready to give your brain a workout

I think it's important to set some expectations: learning to code is difficult, especially if you don't have a background in tech. A lot of things in this book will not immediately make sense to you. This is fine and totally to be expected.

Here is a typical piece of C++ source code that you'll be writing:

```
void Tempo::update(const juce::AudioPlayHead* playhead) noexcept
{
    reset();

    if (playhead == nullptr) { return; }

    const auto opt = playhead->getPosition();
    if (!opt.hasValue()) { return; }

    const auto& pos = *opt;

    if (pos.getBpm().hasValue()) {
        bpm = *pos.getBpm();
    }
}
```

If you're new to coding, this might look like complete gibberish! Some of it may seem to resemble English but there are a lot of strange symbols such as & and : and what are all these { } braces even for?!

Don't worry too much about it for now. I'm asking you to roll with it and trust the process. Try to understand each snippet of source code that's shown in this book as best as you can, but if there are symbols or words you can't figure out, then just pretend you know what's going on and continue reading.

All will be revealed in due time as you work through the book. I've tried not to overload you with too much new information at any given time, and so the book progressively peels back the curtains.

We will start out easy, with an empty plug-in that doesn't do anything yet. In every new chapter we will add a new feature to the plug-in. Even if the code looks like undecipherable spaghetti, at least you get to see how a plug-in works on the inside. And with some luck, the code will start to make sense once you get into it.

It's possible that by the end of this book, still none of this will be clear to you. Perhaps you won't even make it halfway and will throw the book away in frustration. Guess what? This is fine, you're not dumb. Software development is incredibly difficult to learn and I'm throwing you into the deep end here, with one of the hardest programming languages to learn (C++) in a very specialized field (audio programming).

It will be hard and you may think of giving up. It's OK to put the book down for a while and do something else. Perhaps pick up a C++ beginner's book, or watch some C++ videos on YouTube — or even some cat videos. Breaks are important. Just as with weightlifting, the actual growth takes place when you're resting.

One day, and you may not even be at the computer when it happens, a thought will drift into your mind and collide with some other idea percolating up from deep within your subconscious and... bam! A light bulb goes on in your head. Something will finally click and whatever problem you were struggling with suddenly makes sense.

Believe me, once you've had your first a-ha! moment, you will be hooked on this wonderful pursuit of computer programming. I hope you'll receive such moments of insight while reading this book, but if you don't, simply keep going and keep learning and keep struggling, and you'll get there. Give it time and have patience.

Fighting with this material, the frustration of not understanding something, is not something to shy away from but to embrace. It means you are pushing yourself, and when you do get that breakthrough, when your brain has made those new connections, you can be proud of this intellectual achievement. And be amazed by all the wonderful new stuff that's still out there to learn.

What about math?

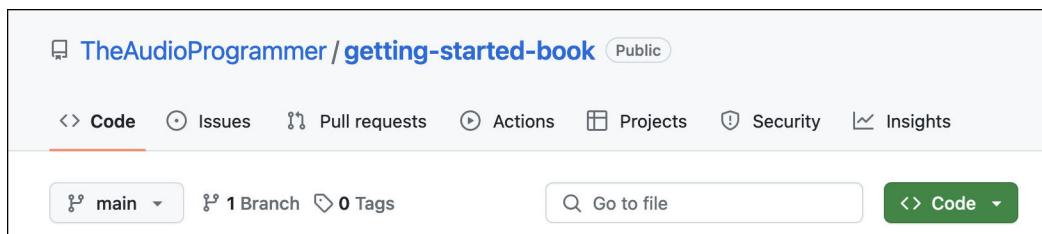
I have good news and I have bad news. The good news is that this book doesn't use a lot of math beyond basic arithmetic — addition, multiplication, and some exponentiation. The book is written so that you should be able to follow along even if your math is not very good. Phew!

The “bad” news is that, if this book gets you excited about audio programming and you’re planning to make a career out of it, eventually you’ll need to brush up on your math. There’s no shame in this, I did it myself. A few years ago, I bought some beginner math books for adults and worked my way back up from early high school level math to college level math.

It’s not easy, but it’s worth doing, especially in the field of digital signal processing (DSP). And trust me, if you’re able to grok C++, you’ll also be able to get that math through your thick skull!

The source code

All the source code you need is listed in this book. You’ll build up the plug-in project step-by-step, adding to the code in each chapter. That means you should read this book from front to back. If you skip sections, you might miss out important code snippets.



The GitHub page for this book

Please also download the book’s [source code from GitHub](#)¹² as you’ll need some of the files this includes. Go to the link, click the big green **Code** button and choose **Download ZIP** from the popup. This downloads the source code as a zip file to your computer.

¹²<https://github.com/TheAudioProgrammer/getting-started-book>

Unzip the download and you'll find the following files and folders:

- **Chapter Code.** The finished source code for each chapter in the book.
- **Finished Project.** The source code for the completed plug-in.
- **Resources.** This folder contains images and sound files that you will need to follow along with the book.
- **Errata.** A list of known mistakes in the book.

Maybe it's my age and I fondly remember typing in source code listings from computer magazines, but I do believe that it's good to type in all the source code yourself. It's more work than copy-pasting the code but typing everything in does make you pay more attention to what's going on.

When you get stuck and your plug-in will no longer compile, and you can't make heads or tails of the error messages, it's a good idea to compare your code to the official version from GitHub. Usually it's a simple typo somewhere, but they can be hard to find if you don't know what you're looking for. The code from the download is guaranteed to work.

2: How plug-ins work

This book is about writing audio plug-ins. So, what exactly is a plug-in?

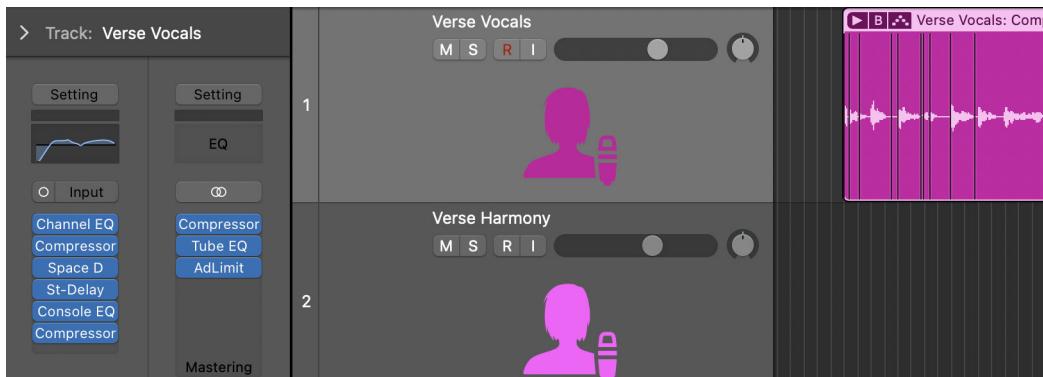
These days, most music is recorded and edited using a DAW or Digital Audio Workstation software. Examples are Logic Pro, Ableton Live, REAPER, Cubase, FL Studio, GarageBand, Bitwig Studio, Pro Tools, and many others.

DAWs come with tons of functionality already built in, but they also allow third-party developers such as yourself to add new functionality through plug-ins.

A plug-in is a piece of software that gets loaded into another software program, known as the **host**, to extend the capabilities of that host. In the case of audio plug-ins, the host is typically a DAW but could also be a video editing application, for example.

An audio plug-in is often called a VST, after the technology that first made such plug-ins possible, Steinberg's Virtual Studio Technology.

In most DAWs, the plug-in is placed on a track in the user's project, so that it gets inserted into the audio processing for that track. There can be multiple plug-ins per track. Each plug-in will read the incoming audio and modify it somehow.



Plug-ins on a vocal track in Logic Pro

The screen capture on the previous page shows the effects that are on a vocal track from Billie Eilish's hit single Ocean Eyes, which is one of the demo projects included with Logic Pro. The plug-ins are represented by the blue buttons on the left.

The plug-ins are applied from top to bottom. First there is an equalizer (Channel EQ) that cuts out the low end and some of the mids, followed by a compressor to even out the levels. Next up is Space Designer for reverb. After that, a stereo delay (St-Delay) that is similar to the plug-in you'll be making in this book. Finally, there is another EQ and compressor for good measure. The output bus (where it says Mastering) has its own compressor, EQ, and limiter plug-ins on it.

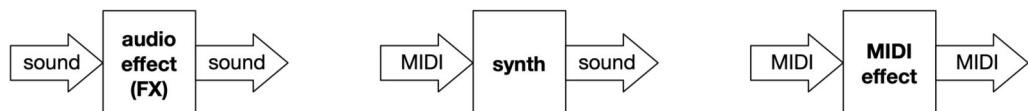
Effects versus synths

There are a few types of plug-ins but the most common are:

1. the effect or FX plug-in
2. the instrument or synth plug-in
3. MIDI effects

The main difference is that an effect plug-in processes existing sound, while a synth creates completely new sounds based on MIDI notes. It's also possible to make plug-ins that do both.

MIDI effects transform notes, for example to transpose a melody up or down, or to create arpeggios. If you don't know what MIDI is, think of it as sheet music in electronic form. We won't be using MIDI in this book.



The different plug-in types

From the point-of-view of the plug-in, the DAW is the host application that provides access to the computer's audio inputs and outputs. An effect plug-in receives audio from the track, modifies it in some fashion, and sends the result back to the DAW. For a synth plug-in, the DAW provides it with MIDI events that let the synthesizer know which notes to play.

Other plug-in types are possible but are not supported by all DAWs. In this book we'll focus on building an effect plug-in. To learn more about making synths, see my other book [Creating Synthesizer Plug-Ins with C++ and JUCE¹³](#).

Plug-in formats

For plug-ins to be possible, there needs to be some kind of agreement on how the DAW host program and the plug-in will communicate, so that the DAW can tell the plug-in when there is new audio to be processed. Conversely, the plug-in must notify the host of any changes, for example when the user interacts with the plug-in's user interface.

Here's the rub: There are multiple of these communication protocols and they are all incompatible.

- **VST2** is the original standard that created a thriving ecosystem of plug-ins, but it has now been deprecated and most developers won't (legally) be able to make new VST2 plug-ins anymore.
- **VST3** is the replacement for VST2. Like its predecessor, it is cross-platform and can be used on Mac, Windows, and Linux.
- **Audio Units (AU)** are Apple's plug-in standard. Like VST, there are several versions. AUv3 has found a home on iOS devices, but on the Mac, developers are mostly sticking with the older AUv2.
- **AAX** is the plug-in format for Avid's Pro Tools software. This type of plug-in is used only by Pro Tools, but this DAW is still the industry standard for professional audio work and therefore worth supporting.
- **LV2** is a plug-in standard that is especially popular on Linux. This is the successor to the older LADSPA format.
- There are a number of other plug-in formats but they are less common. A group of developers from the audio community have been working on a new open source format, CLAP, which looks to be quite exciting but it was not fully supported by JUCE at the time of writing this book.

¹³<https://theaudioprogrammer.com/synth-plugin-book>

As a maker of audio plug-ins, of course you want as many users as possible to use your products. That means you'll need to support more than one plug-in format. VST3 is the most universal format as it works on all platforms, but not all DAWs support VST3. A good example is Logic Pro, which only accepts Audio Units.

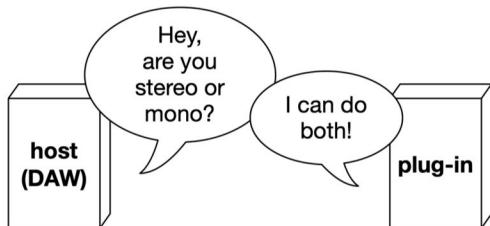
Most commercially available plug-ins come as VST3, AU, and AAX versions, for both Windows and Mac. Not everyone supports Linux but that's slowly improving. Some plug-ins are also available on iOS or Android.

Because you will be using JUCE, all you need to do is write your plug-in against the JUCE framework API (application programming interface) and JUCE will handle VST3, AU, AAX, LV2, or any other new plug-in format in the future. It seems reasonable to assume that JUCE will eventually support CLAP too.

Thanks to JUCE you don't have to deal with these different formats yourself, allowing you to focus on making a cool plug-in and not on the low-level plumbing of getting it to work with all the different DAWs and operating systems.

Communicating with the host

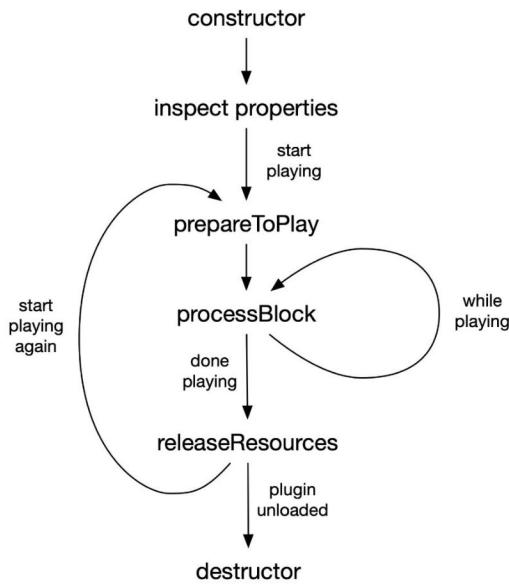
The plug-in and the host program need to talk to each other. For example, the DAW will query the plug-in what kind of bus configurations it supports, asking questions such as, "Do you support a mono bus? Do you support a stereo bus?" and so on. The plug-in needs to respond to these inquiries.



The plug-in and host talk to each other

In this relationship, the host program is in charge. The DAW will send commands and requests for information to the plug-in and the plug-in has to respond. For example, if a plug-in says it can only be used on stereo sound, the DAW may decide not to let the user put this plug-in on a mono track.

The life of a plug-in goes through the following stages:



The lifecycle of a plug-in

When the user adds a plug-in to a track, the DAW will load the plug-in into the computer’s memory (also known as RAM) using the plug-in’s constructor.

Next, the DAW will inspect the plug-in’s properties by asking it questions such as:

- is this an effect or a synth?
- does it accept or produce MIDI messages?
- how many input and output channels can it handle?
- does the plug-in have a user interface (UI)?
- are there any factory presets?
- is the output of the plug-in delayed; in other words, is there any latency?

There are separate requests for all these questions: `acceptsMidi`, `hasEditor`, `getNumPrograms`, `getLatencySamples`, and so on. In the illustration above, these are grouped under “inspect properties”.

Once the host understands what the plug-in does and wants to start the audio processing, it sends the command `prepareToPlay`. This also informs the plug-in about the sample rate and audio buffer size to use.

This is the plug-in's chance to get everything ready to go before it starts receiving audio. For example, upon receiving the `prepareToPlay` command, our delay plug-in will reserve enough memory for storing five seconds of delayed sound.

The counterpart of `prepareToPlay` is the command `releaseResources`, which is sent when the DAW has finished playing audio. This is a good place for the plug-in to free up any resources it no longer needs.

The most important command is `processBlock`. This request is sent hundreds or even thousands of times per second by the host application and is where the plug-in will do all its work, namely process the incoming audio.

There are other commands the DAW may send to the plug-in, but we will spend most of our time working with `prepareToPlay` and `processBlock`.

When the user decides they didn't want to use this plug-in after all and removes it from the track, the destructor unloads the plug-in from the computer's memory.

The above describes how the **processor** part of the plug-in communicates with the host. But a plug-in is actually made up of two separate parts: the processor that handles the audio processing, and the **editor** that deals with the plug-in's user interface or UI.

It is not required but most plug-ins will have an editor. When the user clicks on the plug-in to open it, the host sends the `createEditor` command to the plug-in so that it can present the user interface. There is always an instance of the processor, but the editor will only exist when the host application needs to show the UI.

Plug-in parameters

Another way in which the host and plug-in communicate is through the **plug-in parameters**.

Every plug-in has knobs and other controls that allow the user to change the sound that is produced. Our delay plug-in will have knobs for the delay time, the amount of feedback, the stereo width, the output level, and so on.

For each of the knobs you will define a parameter, which exposes these controls to the host application. Giving the host access to the plug-in's settings makes it possible for the user to do things like record automation, for example to automatically fade in the amount of wet signal to use in the mix.

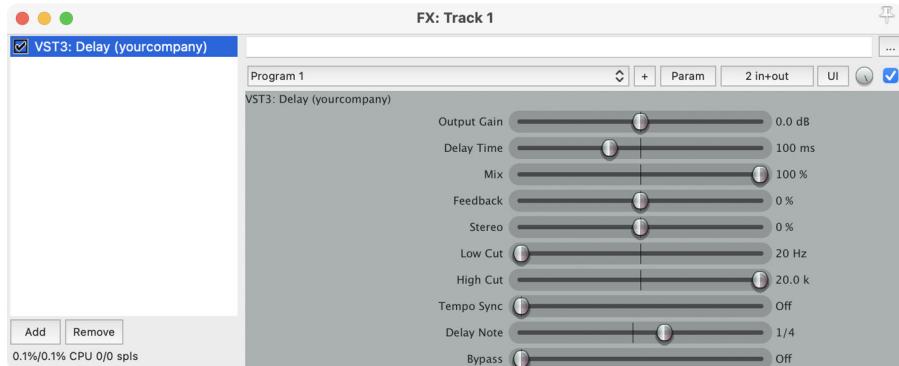
The parameters used by our delay plug-in are:

- the delay time measured in milliseconds
- the delay time as a note value, from 1/32nd note to a whole note
- whether tempo sync is enabled
- the feedback amount, a percentage between -100% and 100%
- the stereo width, a percentage between -100% and 100%
- the cutoff frequency for the low-cut filter, from 20 Hz to 20 kHz
- the cutoff frequency for the high-cut filter, also from 20 Hz to 20 kHz
- the dry/wet mix amount, a percentage from 0% to 100%
- the output gain, from -12 to +12 dB (decibels)
- whether the plug-in is currently bypassed

When the user turns a knob or presses a button in the plug-in's editor, JUCE will notify the host of this through the corresponding parameter.

Conversely, when the host plays back automation, JUCE makes sure the plug-in is notified of any changes so that it can update the user interface and adapt how the audio gets processed.

Most DAWs will let you view the plug-in parameters. For example, in REAPER press the UI button in the top-right corner of the FX window. The display switches from the plug-in's editor to a list of parameters:



REAPER's generic UI for the parameters

The plug-in's parameters also appear in the automation menu for your DAW. For example, in Logic Pro:



The parameters in Logic Pro's automation menu

Not everything a plug-in does has to be covered by a parameter. For example, if the plug-in has a resizable user interface, you wouldn't make a parameter for the window's width and height. Parameters are only to be used for settings related to the audio processing.

The **bypass** parameter is special. Most DAWs let the user temporarily disable an effect, without removing it from the track, by toggling a bypass button. This button

is hooked up to the plug-in's bypass parameter, so that the plug-in knows from now on it should pass through the audio unchanged.

When the user saves their music project from within the DAW, the plug-in's settings must be stored along with that project as well. Otherwise, the next time the user opens their project, all the plug-in's knobs would be at their default positions.

The host does not automatically handle saving and loading, it requires the help of the plug-in. This is called **state restoration** and it has two commands that the plug-in must implement:

- `getStateInformation`: The host asks the plug-in to save all its parameter values and anything else it wants to remember.
- `setStateInformation`: The host tells the plug-in that it should restore its state from a previously saved project.

It's common for JUCE plug-ins to save their parameter values into an XML document, and that's what we'll do in this book too, but ultimately the plug-in is free to choose how it saves this data — as long as it can load the data again when the hosts tells it to.

A plug-in can also have factory presets, which are typically accessed through a menu item in the window that holds the plug-in editor, although not all DAWs have such a menu. In JUCE terminology the factory presets are called programs. Each program is basically a collection of parameter settings.

Introduction to digital audio

The main job of a plug-in is to process audio. Every few milliseconds, the host will send the `processBlock` command to the plug-in, telling it to perform its magic on a new block of audio samples. Let's look at what digital audio is, what samples are, and why we process them one block at a time.

To make sound, molecules in the air need to be moved back and forth with enough force so that our ears can pick up the vibrations. These air vibrations must happen quite rapidly for the human brain to register it as sound, between 20 and 20,000 times per second.

To become music, the air molecules should also move in a way that feels pleasant to listen to — or unpleasant if grindcore is your thing.

As a programmer, you don't need to worry about physically making the air vibrate. The computer's loudspeakers take care of this. You do need to describe to the computer exactly how you want the loudspeakers to move. This description consists of a stream of 44100 or more numbers per second.

The computer's DAC or digital-to-analog converter turns this stream of numbers into electrical signals that rapidly move the electromagnetic coil in the loudspeaker back and forth, which in turn causes the speaker's diaphragm to push and pull against the air. The resulting air movement is what we call sound.

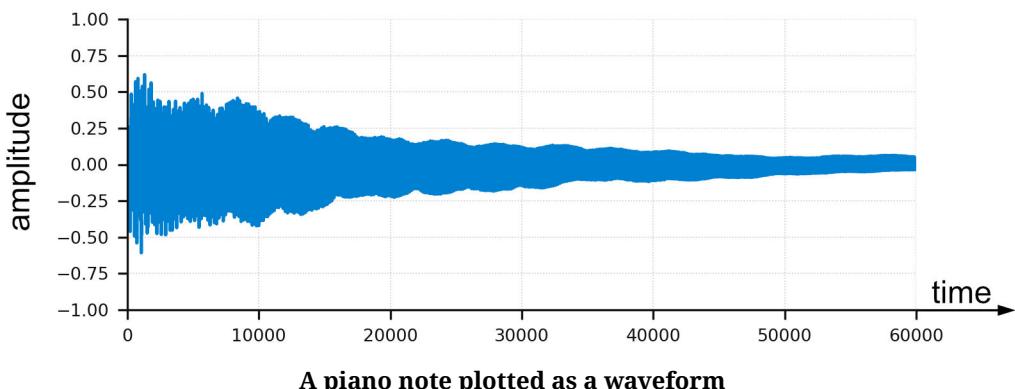
DSP programming is all about dealing with such lists of numbers. In JUCE these numbers are values between -1.0 and +1.0, in computer terms known as **floating-point** numbers because they have a decimal point.

For example, if you tell the computer to output the following sequence of numbers through its DAC, you will hear a sound with a frequency of 261.63 Hz, which is the pitch of middle C on the piano.

```
0.0      ,  0.0223604 ,  0.04468973,  0.06695697,  0.08913118,  0.11118156,  0.13307748,  0.1547885 ,  0.17628447,  0.19753552,
0.21851213,  0.23918516,  0.25952587,  0.27950601,  0.29909783,  0.3182741 ,  0.33700818,  0.35527404,  0.37304631,  0.39030029,
0.40701201,  0.42315826,  0.4387166 ,  0.45366541,  0.46798393,  0.48165226,  0.49465141,  0.50696333,  0.5185709 ,  0.52945801,
0.53960952,  0.54901133,  0.55765038,  0.56551466,  0.57259326,  0.57887633,  0.58435515,  0.5890221 ,  0.5928707 ,  0.5958956 ,
0.59809261,  0.59945866,  0.59999187,  0.59969148,  0.59855792,  0.59659277,  0.59379874,  0.59017973,  0.58574076,  0.580488 ,
0.57442875,  0.56757142,  0.55992554,  0.55150174,  0.54231172,  0.53236825,  0.52168513,  0.51827722,  0.49816037,  0.4853514 ,
0.47186812,  0.45772925,  0.44295444,  0.42756422,  0.41157997,  0.3950239 ,  0.377919 ,  0.36028905,  0.34215854,  0.32355265,
0.30449724,  0.28501878,  0.26514433,  0.24490151,  0.22431844,  0.20342371,  0.18224636,  0.16081581,  0.13916183,  0.11731451,
0.0953042 ,  0.07316148,  0.05091712,  0.02860201,  0.00624717, -0.01611636, -0.03845749, -0.06074519, -0.0829485 , -0.10503656,
-0.12697869, -0.14874441, -0.17030347, -0.19162592, -0.21268214, -0.23344287, -0.25387927, -0.27396295, -0.293666 , -0.31296105,
```

Digital audio is just a list of numbers

Of course, a list of numbers isn't very easy to understand, so we will usually plot this in a graph, such as the one shown below.

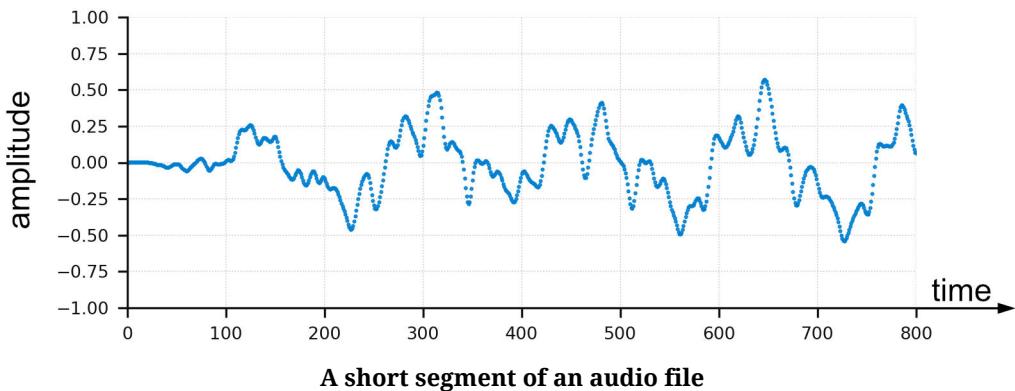


This is a picture of an audio waveform, about 1.4 seconds of a sampled sound of a piano playing middle C.

The **amplitude** is on the vertical axis and describes the position of the loudspeaker's diaphragm at a given instance in time. A larger amplitude means the speaker cone gets pushed further outward. The time is on the horizontal axis, going from left to right, and is measured in number of samples.

Notice how this signal oscillates up and down but is centered around the amplitude value of 0.0? That is the resting position of the speaker cone. An amplitude larger than zero means the speaker cone is pushed outward and an amplitude smaller than zero (a negative amplitude) means the cone is pulled inward. Sound is made by the cone moving from one position to another. At rest, the speaker cone is sitting in the center, which is represented by amplitude value of zero.

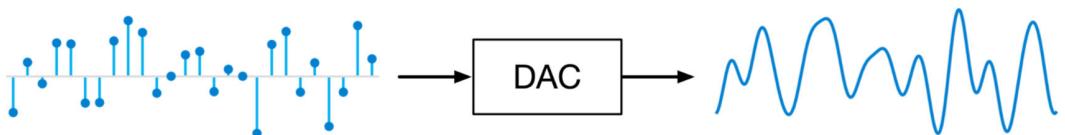
If we zoom in on this waveform, we can see the points it is made of:



Any individual point on this graph is called a **sample**. For this particular waveform, it takes 44100 of these samples to output one second of sound. The 800 or so samples in this illustration describe only a tiny fraction of time, about 18 milliseconds worth of audio. Note how these samples make an up-and-down movement — there can be no sound waves unless the loudspeaker cone is moving back and forth.

Inside the computer, then, sound is described by a stream of 44100 (or more) numbers per second that are called samples. The DAW will pass this list of numbers through all the plug-ins on the track, and the plug-ins take turns modifying these numbers in some — hopefully! — interesting way.

Eventually the computer's DAC takes this sampled data and turns it into a continuous voltage that it uses to drive the loudspeakers. The samples specify how the speaker's diaphragm should move back and forth to make the air vibrate — and turn it into something that our ears and brains interpret as sound.



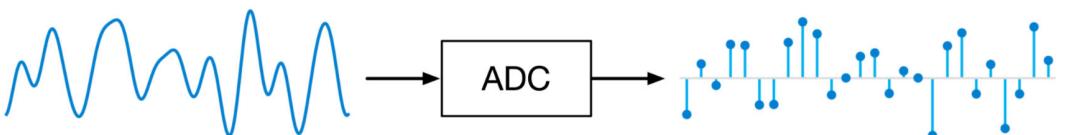
From digital to analog

For **stereo** sound, the plug-in needs to output two times 44100 samples per second, one stream for the left speaker and one stream for the right speaker. Ultimately, digital audio is nothing more than a stream of numbers that create interesting patterns of air movement!

The sampling rate

Why does it take 44100 samples to capture one second of audio, that seems like such a strange number?

Let's say that we're going to record some sound with a microphone. The movement of air molecules in the sound wave will cause the microphone's membrane to vibrate inside a magnetic field, creating a continuously changing voltage. This voltage is converted by the computer's ADC (analog-to-digital converter) into a series of numbers, the samples.



From analog to digital

The ADC does this by taking a voltage reading every few microseconds and storing that voltage into a 16-bit or 24-bit integer. In computer terms, an **integer** is a whole number, as opposed to a floating-point number that can have a fractional part.

The number of bits determines how large the integer number can be. A 24-bit integer can hold larger numbers than a 16-bit integer, but also takes up more space in the computer's memory.

The ADC takes an analog, continuous signal that is made up of a smoothly changing voltage, and turns it into a digital, discrete signal consisting of a stream of integer sample values. The host application maps these integers into floating-point numbers between -1 and +1, as they are easier to work with.

There is a very important theorem in digital signal processing that says the **sampling rate**, or the speed at which the ADC takes these voltage readings, must be at least twice the highest frequency in the signal that is being sampled. If the sampling rate is not high enough, the original analog signal will not be faithfully represented by the sampled digital signal.

Human hearing tops out at roughly 20000 Hz (or 20 kHz, which stands for "kilo hertz") although many people cannot hear frequencies that high — I'm in my mid-forties and like most people my age, I can't hear anything over 15 kHz. If we take 20 kHz as the upper limit of the frequencies to record, the sampling rate should be at least 40 kHz or 40000 samples per second.

In practice, a sampling rate of 44100 samples per second or 44.1 kHz is used. This odd number is the result of choices that were made in older technologies for video and audio, such as the Compact Disc, and the math for that happened to work out to 44.1 kHz. As this sampling rate is sufficiently high to capture the highest frequencies that (young) human ears can hear, that's what the industry settled on.

These days, it's common to use a sampling rate of 48 kHz or 48000 samples per second. The difference with 44.1 kHz is small and mostly does not matter. Some folks even use 96 kHz or higher, although a higher sample rate isn't necessarily better. 44.1 kHz is already good enough to capture all sounds humans can hear, and the higher the sample rate, the more numbers it takes to describe the sound. At some point, increasing the sampling rate is a waste of perfectly good numbers.

That said, there are valid reasons for using a higher sample rate inside a plug-in, a technique known as oversampling. This helps to reduce the occurrence of every audio programmer's nemesis: aliases. An **alias** is a frequency that is too high to represent with the chosen sample rate and gets reflected as a lower frequency. Usually these aliases show up as inharmonic buzzing sounds messing up your carefully crafted mix, and so they are unwanted.

Note: In your audio programming career, you may run into audio files with much lower sample rates such as 16 kHz or even 8 kHz. These lower sample rates aren't typically used in music production but can be OK for recording sound that doesn't have many high frequencies, such as speech.

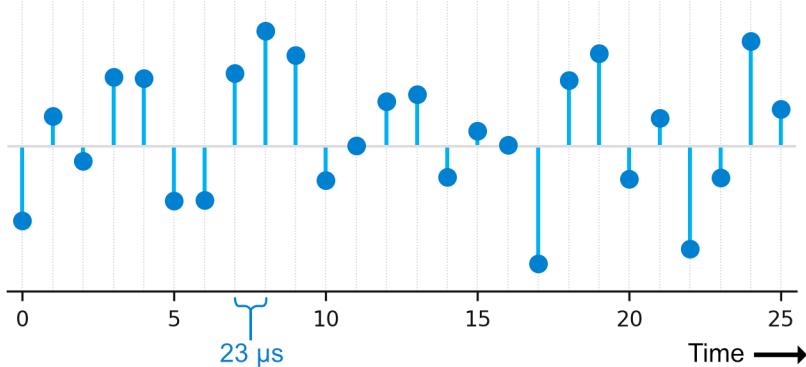
A closer look at samples

For DSP programmers, dealing with sampled data is our bread and butter. Therefore, it's a good idea to take a closer look at exactly what samples are.

Here's a picture of a very short snippet of audio. The sample rate is 44100 Hz and there are 26 samples in this snippet, so this corresponds to approximately 0.6 milliseconds of sampled data.

If you're wondering how I got that number: 26 samples divided by 44100 samples per second gives 0.000589 seconds or 0.6 msec rounded off.

Each of these sample values represents the displacement of the loudspeaker cone at a specific time, what we call the amplitude of the sample.

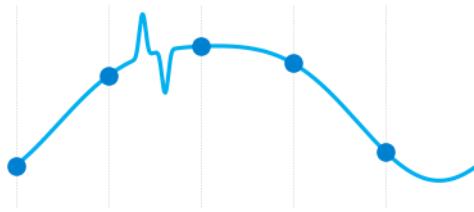


A zoomed-in view of a sampled signal

The horizontal axis has the time, not in seconds but expressed as number of samples. Note that there is spacing between the samples on the time axis. This is correct since the computer's ADC only takes voltage readings every few microseconds.

At the sample rate of 44.1 kHz, the ADC takes one sample every $1 / 44100 = 0.0000227$ seconds or 23 microseconds, so the time interval from one sample to the next is also 23 microseconds (23 μ s). The higher the sample rate, the smaller this spacing.

You might be wondering how the sampled data can faithfully represent the original signal. Surely, since we only take readings every so often, we must lose some information along the way? For example, what if the following happens...



A wiggle in between two samples

After sampling this signal, we have no idea that this “wiggle” occurred in between the two samples, since the ADC never saw it happen. Here’s the kicker: if such a wiggle can take place, the chosen sampling rate is too low. The original analog signal apparently has much higher frequencies in it than we expected.

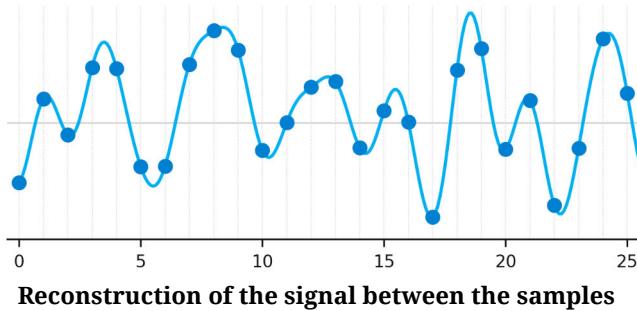
When using sampled data, we will make the assumption that the signal is **band-limited**, meaning it does not contain frequencies that are higher than half the sampling rate. In other words, we need to make sure that the sampling rate is at least twice as high as the highest frequency that occurs in the signal.

The maximum frequency the signal can contain is known as the **Nyquist limit**. When using a sampling rate of 44100 Hz, the Nyquist limit is 22050 Hz, and the sampled signal can faithfully represent any frequencies lower than 22050 Hz — well above the hearing range of humans. But frequencies higher than 22050 Hz won’t work.

By the way, an ADC has an electronic circuit that filters out all frequencies that are too high for the chosen sampling rate, so the “wiggle” situation isn’t a problem in practice when recording audio — unless you actually want to capture these high frequency fluctuations in your digital signal, in which case you’d need to increase the sampling rate of the recording device.

The takeaway here is that, when the sampling rate is high enough, the sampled signal is equivalent to the original, continuous signal. Given a band-limited signal, we know *exactly* what the audio looks like between the samples, as this is the only possible signal that can be described by these samples.

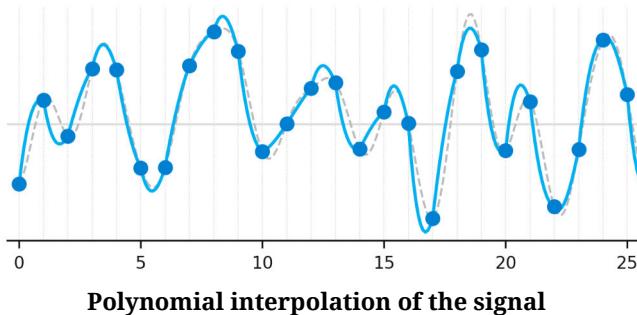
For the samples shown previously, the real signal is as follows:



Note that the line that connects the samples is smooth and curved. It may sometimes go higher or lower than the two closest samples, but it never wiggles.

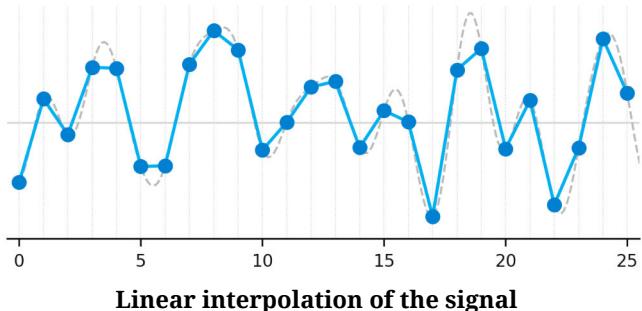
We can precisely calculate this signal from the given samples. For readers who are good at math: this is done by convolving the sampled signal with the “sinc” function, $\sin(\pi x) / \pi x$. You may immediately forget this for now.

The important thing to remember is: There is only one correct mathematical solution. You can't just draw any line between the samples. For example, a polynomial fit might also go through all the sample points but this does not represent the original analog signal (the dotted line is the true original):



The polynomial approximation might be fairly similar to the real signal at first glance, but sometimes still misses the mark completely.

Linear interpolation is even worse. With linear interpolation we imagine there is a straight line between each pair of samples. As you can tell from the picture on the next page, that isn't even close to what the signal really looks like.



Unfortunately, the math for finding the exact shape of the continuous analog signal is slow, so in practice we have no choice but to use some kind of approximation if we want to know what happens in between the samples.

I'm pointing this out because determining what the signal looks like in between samples is a very common task in audio programming, and something we will encounter in detail in a later chapter where we have to deal with delay lengths that are a fractional number of samples.

The main things to remember about the sample rate are:

1. The audio should not try to describe any frequencies that are higher than the Nyquist limit of half the sample rate. In other words, the digital signal should be band-limited. When recording, the ADC's filter takes care of this already, but for a plug-in it's easy to go over the Nyquist limit, especially when using non-linear processing such as waveshaping.
2. Even though you only have samples that describe a fraction of the true analog signal, this is enough to reproduce the analog signal exactly. Assuming that it is band-limited, sampling a signal does not lose any information about that signal! You can always predict what the continuous signal looks like between any two samples, although it's costly to find an exact match.
3. Often a plug-in will have parameters that are measured in seconds or milliseconds, such as the Delay Time knob in the plug-in we will be building. But the audio processing code measures time by the duration of each sample. This is called the **sampling interval T** and is equal to $1/\text{sampleRate}$. As you've seen, at a sample rate of 44.1 kHz, the sampling interval is 23 microseconds. To convert between a time in seconds and a time in number of samples, we will therefore need to use the sample rate.

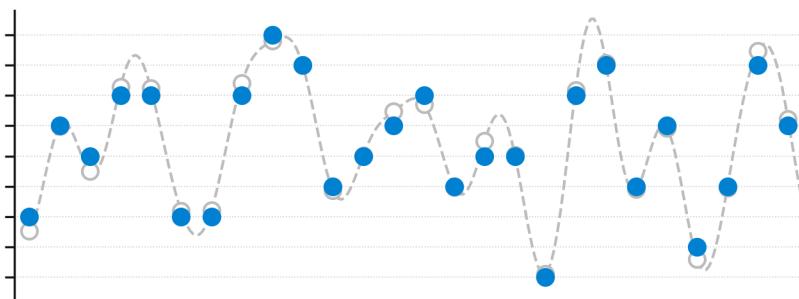
Quantization and bit depth

So far, we've discussed what happens along the time axis, but the vertical axis — the height of the samples — is also important. When recording audio, each sample is a measurement of the air pressure on the microphone's membrane at that instance. As we play back that sample, it represents how much air pressure the speaker cone generates.

In JUCE the sample values are floating-point numbers between -1 and $+1$, but the ADC and DAC work with integers (whole numbers). On modern computers, those are usually 16-bit or 24-bit integers. This is called the **bit depth** of the sample.

With a bit depth of 16 bits, the sample can record the sound pressure level as 65536 possible values. With a 24-bit integer this becomes 16.8 million possible values.

While sampling, the ADC must put the continuous voltage reading into a discrete number with a limited number of bits. It does this by rounding off the voltage to the nearest whole number. That rounding-off process is called **quantization** and the error it introduces into the signal is called quantization noise.



Exaggerated depiction of quantization. The blue dots are rounded to the nearest integer.

The more bits you're using to store the samples, the smaller the effect of this quantization noise. 24 bits gives a more faithful representation of the original voltage than 16 bits. The question is: does this difference matter in practice?

With 16-bit samples, the noise from the quantization error is 65536 times quieter than the actual sound, giving a signal-to-noise ratio of 96 dB. It's pretty much impossible for human ears to hear sounds at -96 dB, unless a massive amount of amplification is applied.

For this reason, 16-bit audio is generally considered to be good enough. However, 24-bit audio is very common these days and makes the noise floor even lower.

For a plug-in programmer, this distinction doesn't matter that much, as we're using floating-point numbers instead of integers.

If you've worked with "floats" before you might be a little suspicious because this data type can have precision issues. For example, the number 0.1 is really 0.10000000149 inside the computer. Weird, huh!

Fortunately, 32-bit single-precision floats have a precision of 24 bits, so they should be able to handle most audio just fine. If necessary, JUCE lets you use 64-bit double-precision floating-point for an even lower noise floor.

Bit depth and quantization noise aren't something to worry about when writing plug-ins. At the end of the signal path, the DAW can add so-called dithering. This adds small amounts of noise into the sound on purpose to hide the rounding errors from quantization.

Tip: Christopher "Monty" Montgomery made an [excellent video^a](#) about how digital audio works. Recommended viewing for all audio programmers!

^a<https://www.youtube.com/watch?v=cIQ9IXSUzuM>

Processing audio in the plug-in

You've seen that digital audio is described by a list of numbers that represent the sample amplitudes. The plug-in will read this list of numbers and modify them somehow to create the output signal.

For example, if the plug-in were to multiply all samples by a factor of 2, the audio would become louder since all the sample amplitudes are now larger — in fact, multiplying by two means the loudness goes up by approximately 6 dB. Likewise, dividing the sample amplitudes by two means a level reduction of -6 dB.

The trick in building an effect plug-in is to come up with some interesting way to modify these samples. The plug-in you'll build in this book will delay the sound for a set amount of time and mix it with the incoming sound.

Audio blocks and buffers

Whenever the host application needs new audio from the plug-in, it will tell the plug-in to perform the `processBlock` command. This is also referred to as the **audio callback**. This happens hundreds or even thousands of times per second.

Even though digital audio is made up of a “stream” of 44100 or more samples per second, it would be really inefficient for the DAW to ask the plug-in to output its audio one sample at a time. That would involve sending the `processBlock` command forty-four thousand times per second!

Instead, the audio stream is chopped up into smaller portions, so-called **blocks**. The plug-in will output one block’s worth of samples at a time.

A typical block contains 128 samples but they can be as small as 16 samples or as large as 4096 samples. If the block size is 128 samples and the sample rate is 44.1 kHz, the plug-in’s `processBlock` is performed $44100 / 128 = 344.53$ times per second on average, so let’s say 345 times.

That’s still a lot, and so you can imagine the audio callback needs to be fast. The smaller the size of the block, the more often the host needs to send the `processBlock` command.

The size of these blocks is an important trade-off between latency and overhead:

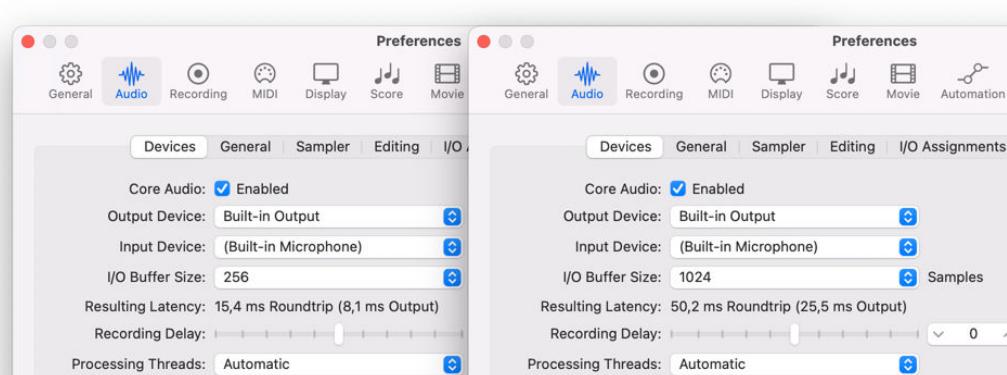
- There is a certain cost to the host for each `processBlock`. By making the plug-in work on blocks of samples instead of individual samples, the same amount of overhead is spread out across multiple samples, which is more efficient.
- If the block size is too small, the computer may not be able to keep up and the audio will stutter.
- On the other hand, the larger the block is, the longer the delay between asking for the block and receiving the results from the plug-in. That delay is known as the **latency**.

Imagine what happens if the block size is set to a whopping 44100 samples. Since each call to `processBlock` now needs to fill up the block with an entire second of audio, there is a one second delay between the audio being played and the sound coming out of the speakers. That makes it impossible to record anything as you’re always a whole second behind the rest of the band. Ideally, the latency is so small that you don’t notice it.

Latency is unavoidable when using digital audio and it comes from several different places: the USB driver from your audio interface, the computer's audio subsystem, the internal processing of the host, and finally, the block size used for audio processing.

Most DAWs let the user set the block size, although they usually call it the **buffer size** instead. We often use the terms block and buffer interchangeably — both mean the same thing.

For example, the image below shows the audio settings screen in Logic Pro with I/O Buffer Size being the setting for the block size. Notice how this calculates the total latency, under Resulting Latency.



Setting the I/O buffer size in Logic Pro

For larger buffer sizes, the latency increases. At a sample rate of 44.1 kHz, a buffer of 256 samples has 5.8 milliseconds latency while a buffer of 1024 samples has 23.2 milliseconds latency.

That's exactly the difference shown in these two images: 8.1 ms output latency versus 25.5 ms. The remainder of the latency, about 2.3 ms, is caused by the computer's hardware, the operating system, and Logic Pro itself.

Even though the user configures the audio buffer size in the DAW, the size of the block isn't necessarily the same from one invocation of `processBlock` to the next. The DAW may decide to call `processBlock` with a smaller block size if it thinks that is a good idea, usually when parameter automation is used.

Think of the host's buffer size as the maximum number of samples that `processBlock` will need to handle at once.

Don't miss that deadline!

One of the most important rules in audio programming is that we want to avoid glitches in the sound. Glitches can be things like cracks or pops or other nasty sounds that aren't supposed to be there. There are lots of ways glitches can happen but one common cause is the plug-in being unable to meet its processing deadline. That's bad! Users won't like your plug-in if it glitches.

Because `processBlock` gets performed hundreds or thousands of times per second, each of these invocations has only a very short time available to do its work. With a 128-sample buffer at 44.1 kHz, `processBlock` is called 345 times per second on average. That means each of these calls should complete in less than 1/345 seconds or approximately 2.9 milliseconds. Not a lot of time!

There are two things that can cause `processBlock` to miss its deadline for processing the current block of audio:

1. The buffer size is too small. The host's overhead plus the actual work from the plug-in is taking too long. This results in an audible stutter, as the host will not have received a filled-up audio buffer in time and so it doesn't have any new sound to output.
2. The audio thread was temporarily halted and there is not enough time left to process the remaining samples from the current block. This can even be an issue when the buffer size is ample.

Like any computer program, the plug-in will run on the computer's CPU or Central Processing Unit. Modern CPUs have multiple cores, which means they can run several programs at the same time. This is known as multitasking. They can even run different parts of one program at the same time using separate **threads** of execution, a technique known as multithreading.

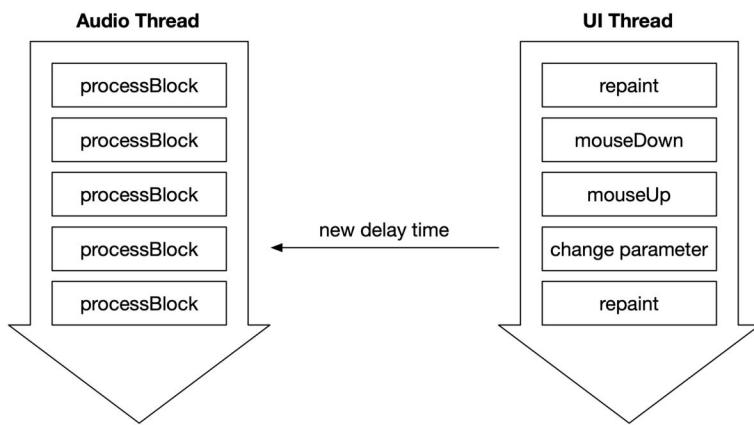
The audio processing logic — anything that happens in `processBlock` — runs in its own thread, the so-called **audio thread**. This is a high-priority thread, meaning that the CPU considers the audio thread to be more important than any other threads.

At any given time, your computer will have hundreds of active threads that all demand attention. The audio thread should take precedence because it has a deadline to meet in order to keep sound playback glitch-free.

It is essential that the plug-in's audio callback never pauses the thread it's running on. Operations such as memory allocations and system calls are perfectly fine anywhere

else but are forbidden from `processBlock`, as they may halt the audio thread for an indeterminate amount of time. If these operations take too long, `processBlock` will end up missing the deadline and cause the audio to start cracking up.

Any operation that can potentially put the audio thread to sleep should therefore be avoided. I'm sure this all seems very vague to you now, but we'll run into these issues for real when we start writing code. For the time being, just remember that anything happening in the plug-in's `processBlock` needs to follow special rules to keep the audio thread happy.



A plug-in uses a thread for audio processing and a thread for the editor

There are other threads of execution in the plug-in too. The user interface from the plug-in's editor is handled by a lower priority thread, usually called the **UI thread** or the main thread, or in JUCE parlance, the message thread. The UI thread doesn't have strict deadlines like the audio thread does. However, the plug-in does need to communicate between these two threads from time to time.

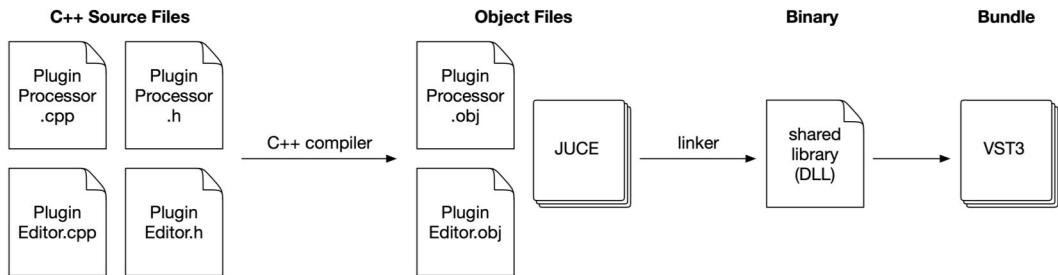
For example, if the user turns the Delay Time knob, the value of the corresponding plug-in parameter is set to the new time. This happens in the UI thread since the knob is part of the plug-in's editor. Naturally, the audio thread also needs to use the new delay time in `processBlock`, so somehow the UI thread needs to tell the audio thread that the parameter has changed.

Most standard ways to communicate between threads involve putting one of the threads to sleep for a short while using a so-called lock or mutex. But sleeping isn't allowed in audio code, so for audio programming there are special communication mechanisms that avoid these locks.

To summarize: We process audio in blocks (also known as buffers) of multiple samples at a time. The size of these blocks matters because larger blocks mean more latency, while smaller blocks means it becomes harder to meet the deadline. There are things you're not allowed to do in the audio thread because it should never be kept waiting.

How plug-ins are made

Now that you've got a sense of the responsibilities of an audio plug-in, the next question is: How do you actually make one?



The process of compiling and linking the source code

It starts with the source code of the plug-in. You write your code in the C++ language but the computer does not understand this language directly. It only speaks machine language, sometimes called assembly, a very low-level set of instructions that is hard for humans to make sense of. If you've ever heard the saying that a computer works with ones and zeros, well, that's machine language.

Ideally we'd write our programs in a human language like English, but that is not precise enough to express to the computer what we want it to do. So instead, we use special programming languages like C++, which are a compromise between something humans can understand and the precision the computer needs. A compiler turns the C++ code into machine language.

You write the source code in text files with the file extensions **.cpp** (for “C plus plus”) and **.h**. There will be separate source files for the audio processor part of the plug-in and for the editor that handles the plug-in’s user interface.

The C++ source files are converted by the compiler into so-called object files that contain the machine code for the computer. This is known as **building** the program. There is one object file for every **.cpp** file.

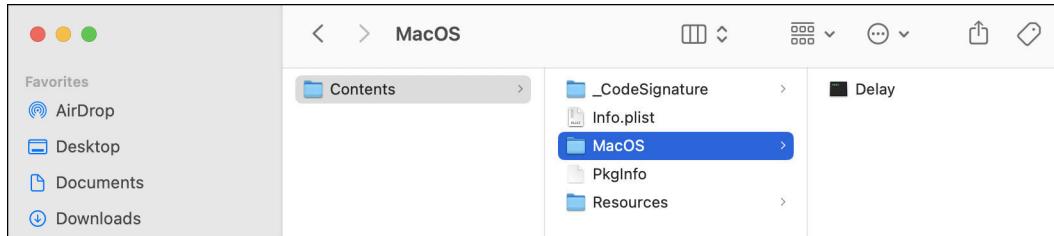
Next, a program called the linker combines these separate object files into a single file known as the **binary**. In the case of a plug-in the binary is a shared library or DLL (dynamic link library). A DLL is not a program that can run on its own but must be loaded into another program, the host.

Finally, the shared library is placed in a **bundle**, which is just a folder on the computer's file system. This bundle has the extension **.vst3** for plug-ins in the VST3 format or **.component** for the Audio Unit format.

For fun, let's look at what's inside such a bundle. I'm assuming you already have one or more plug-ins installed on your computer.

On macOS, navigate to the folder **/Library/Audio/Plug-Ins/VST3**. Right-click on a **.vst3** file and choose **Show Package Contents** from the popup menu. This will reveal the inside of the bundle.

The binary for the shared library is inside the **Contents/MacOS** subfolder. The **Resources** subfolder is for additional files that are used by the plug-in such as images. (It works the same for Audio Unit plug-ins.)



Inside the Delay plug-in VST3 bundle

On Windows, use the file explorer to go to **C:\Program Files\Common Files\VST3**. Plug-ins in this directory may have a folder icon or a file icon. If it's a folder you can double-click it to look inside. The shared library (DLL) for the binary is in the subfolder **Contents\x86_64-win**.

Note that on Windows not all VST3 plug-ins use the bundle structure. Older VST3 plug-ins are a plain DLL file, not a folder.

What is an IDE?

The compiler and linker are part of the **IDE**, which is Xcode on Mac and Visual Studio on Windows.

If you're new to software development, you might not have used an IDE before. The IDE, which stands for Integrated Development Environment, is an application that combines different tasks related to software development into a single tool.

The IDE has:

- a text editor where you'll write the C++ source code
- a compiler and linker to convert the C++ source code into machine code that the computer can execute
- a debugger to help track down problems in your program
- a profiler for measuring potential performance problems
- version control for keeping track of changes to the code

Building the plug-in may sound like a complicated process but is easier than you might think: All you need to do is press a single button and the IDE will compile the source code, link everything together, and create the shared library. When it's done, the bundle for the plug-in is copied into the standard plug-ins folder on your computer, so that DAWs can find and use it.

This book has instructions for both Xcode and Visual Studio, to accommodate readers who use a Mac as well as readers who use Windows. It's possible to use another IDE to write JUCE plug-ins, but this book can't possibly include instructions for all of them.

Some of the other IDEs that you might see audio developers use are:

- JetBrains's CLion, which runs on both Windows and Mac.
- VSCode, or Visual Studio Code, is a cross-platform code editor than can function as a full IDE. This is not something I would recommend for beginners as it can be tricky to get the compiler and debugger toolchain working.
- You can also use a different code editor, such as Notepad++ or Emacs or vim, to write JUCE code. Compiling the project is usually done from the command line.

While these other options may be worth exploring once you've got some experience, if this is your first time writing code then I suggest sticking with Xcode (on Mac) or Visual Studio (on Windows).

This book only covers development on Mac and Windows computers. That said, the code will also work on Linux. If you're a Linux user, you can use your text editor of choice and build the projects from a Makefile.

Note: VSCode (short for Visual Studio Code) and Visual Studio are completely unrelated and have confusing names. Just to make things even more confusing, there used to be a Visual Studio for Mac but that was for the C# language, not C++.

Cross-platform development

Most plug-in developers will make their products available for both Mac and Windows. To do this, you will need access to both types of machines. Unfortunately, you can't make a Windows plug-in on a Mac, and you can't make a Mac plug-in from Windows. Making an iOS plug-in is done from a Mac.

There are two kinds of Macs, older machines using an Intel processor and newer machines using an ARM-based processor, also known as Apple Silicon (the M1/M2/M3 series of processors). The good news is you don't need two Macs: you can make plug-ins for both types of Mac from either an Intel-based or ARM-based machine. However, if you're serious about making plug-ins, then for testing you would ideally have access to both an Intel and ARM-based Mac.

Running macOS on a PC is possible but tricky (a so-called “Hackintosh”), so it's not really worth trying. The other way around is easier: you can run Windows on a Mac, either in a virtual machine or using a special BootCamp partition, but this only works on an Intel machine — it can't be done on an ARM Mac. Likewise, you can run Linux in a virtual machine.

For now, let's not worry about making the plug-in run on multiple platforms yet, so don't feel like you need to rush out and buy that second computer. Just be aware that a VST3 that was built on your Windows PC will not run on your friend's Mac, and vice versa.

In the next chapters you will learn how to install the right IDE for your system and how to get started with JUCE.

3: JUCE and Projucer

JUCE is a software framework that makes it easy to build audio plug-ins.

The C++ programming language comes with a standard library of commonly used functionality — such as data structures, mathematics routines, and sorting algorithms — that will save a lot of time when writing your own programs.

Xcode and Visual Studio will install their own libraries and frameworks that let programs use the features of the computer's operating system.

The terms library, framework, package, module all refer to the same thing: a collection of source code that has been written by someone else so you don't have to.

In the case of JUCE, the framework takes care of jobs such as manipulating text, drawing lines and other figures on the screen, building user interfaces, and of course handling audio.

Many libraries are provided only in binary form. That is, you can link them into your program but you cannot look at their source code. JUCE fortunately comes with full source code, which is handy to figure out how it works internally — sometimes necessary when the documentation is unclear, or to fix bugs.

Note: The following instructions were written for JUCE 7. By the time you read this, it's quite possible the JUCE website and software looks different from the images in the book. However, the process of downloading, installing, and setting up JUCE will still be largely the same. Check the book's errata for updated instructions.

Downloading JUCE

The easiest way to install JUCE is to download it from the official website. Go to juce.com¹⁴ and click the **GET JUCE** button in the top-right corner:

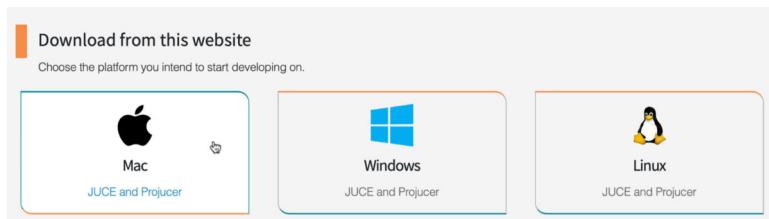


The JUCE website

The JUCE website requires you to select a plan. JUCE can be used for free for personal and educational use, or when making open source plug-ins. Once you start selling plug-ins made with JUCE, it is necessary to pay for a license.

Since you're still learning how to do any of this, click the **Download** button on the free plan (Personal for JUCE 7, Starter for JUCE 8). Note that there is only one version of JUCE, all plans download the exact same thing.

Next, you are given several options to download JUCE. Click the big button that says **Mac** or **Windows**, depending on your operating system.



Downloading the JUCE package

¹⁴<https://juce.com>

Note: The “Download from GitHub” option takes you to the JUCE source code on GitHub. Unless you are familiar already with open source and cloning repositories, I don’t suggest using this option.

The downloaded file can be found in your **Downloads** folder. On Mac, this is a ZIP file named something like **juce-7.0.9-osx.zip**. On Windows, it is a ZIP file named **juce-7.0.9-windows.zip**. The version number may be different for you.

Mac installation instructions:

- Double-click the ZIP file to extract its contents. This creates a new folder named **JUCE**. You’re supposed to move this folder to a convenient location on your Mac, such as your home folder.
- Open a new Finder window and go to your home folder. From the menu bar, this is **Go > Home** or press the key combination **Shift+Command+H**.
- Now drag the **JUCE** folder into your home folder.

Windows installation instructions:

- Right-click on the ZIP file and from the pop-up menu choose **Extract All...**
- In the dialog that appears, select the folder **C:\Users\yourname\Documents** as the destination where the files will be extracted, filling in your actual username instead of “**yourname**”.



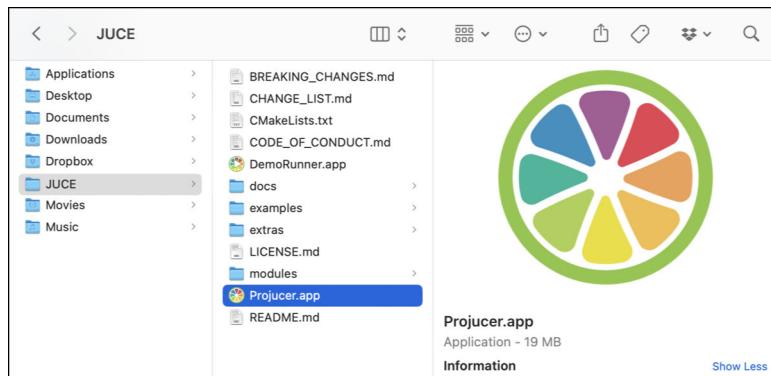
Extracting the ZIP file

- Click **Extract** to begin. This creates a new folder named **JUCE** inside your **Documents** directory. You can move it anywhere you want but I’m going to assume you’re keeping it in **Documents**.

To complete the installation, you have to tell JUCE where it is located on your computer. This is done inside the Projucer app. We will take a detailed look at this app soon, but for now, you'll just use it to set up the JUCE paths.

Setting the JUCE path (Mac)

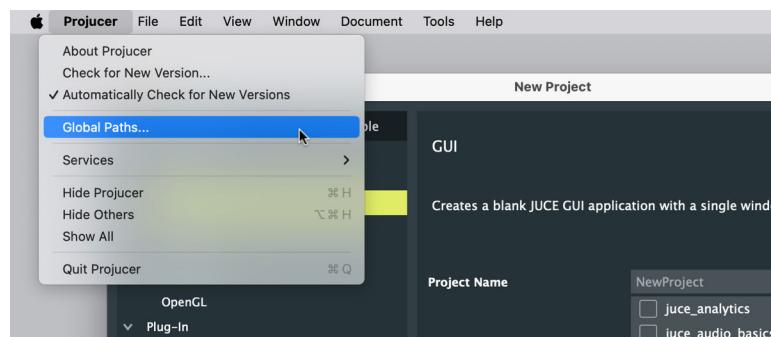
From the **JUCE** folder, double-click **Projucer.app** to open it. It's possible your Mac only says **Projucer** without the .app at the end.



The Projucer app in the JUCE folder on Mac

The computer may warn that you're about to open a file that has been downloaded from the internet. Don't worry, it's perfectly safe! Click **Open** to continue.

In the menu bar, under **Projucer**, select the **Global Paths...** option.



The Global Paths menu item

This opens the following window:



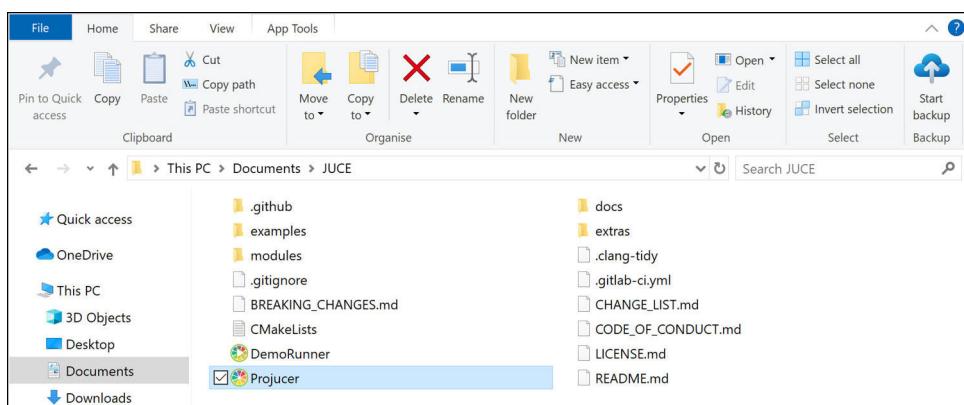
The Global Paths settings window

The two important fields are **Path to JUCE** and **JUCE Modules**. In the screenshot above these start with `~/JUCE`. The tilde `~` is a special symbol that refers to your home folder. In other words, on my computer the path `~/JUCE` actually refers to the folder `/Users/matthijs/JUCE`.

Assuming you placed JUCE in your home folder, this is correct and you don't need to change anything. If you put the JUCE folder somewhere else, use the `...` button to select its location. When you're done, close the **Global Paths** window and exit Projucer.

Setting the JUCE path (Windows)

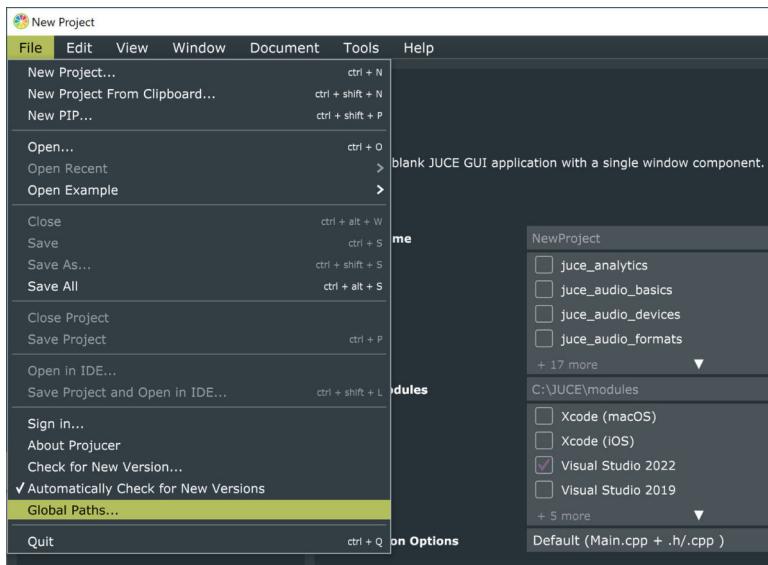
From the JUCE folder, double-click **Projucer.exe** to open it. It's possible your PC only says **Projucer** without the `.exe` at the end.



The Projucer app in the JUCE folder on Windows

Microsoft Defender SmartScreen may warn that you're trying to start an unrecognized app. Don't worry, Projucer is perfectly safe! If this happens, click on **More info** and then **Run anyway**.

In the Projucer menu bar, under **File**, select the **Global Paths...** option.



The Global Paths menu item

This opens the following window:



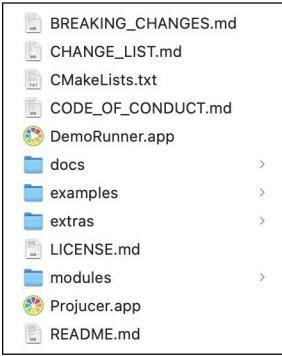
The Global Paths settings window

The two important fields are **Path to JUCE** and **JUCE Modules**. By default these say `C:\JUCE` and `C:\JUCE\modules`. However, you extracted the ZIP file to the `Documents` directory, so you'll have to change these paths to point to where the JUCE folder is.

Use the ... button to select the location C:\Users\yourname\Documents\JUCE for **Path to JUCE**. The location for **JUCE Modules** should be the modules folder inside that. When you're done, close the **Global Paths** window and exit Projucer.

The JUCE folder

Let's take a quick look at the contents of the **JUCE** folder, to give you some sense of what exactly is included with JUCE.



Name	Date modified	Type	Size
.github	09/02/2024 16:29	File folder	
docs	09/02/2024 16:29	File folder	
examples	09/02/2024 16:29	File folder	
extras	09/02/2024 16:30	File folder	
modules	09/02/2024 16:30	File folder	
.clang-tidy	09/02/2024 16:29	CLANG-TIDY File	1 KB
.gitignore	09/02/2024 16:29	GITIGNORE File	2 KB
.gitlab-ci.yml	09/02/2024 16:29	YML File	1 KB
BREAKING_CHANGES.md	09/02/2024 16:29	MD File	94 KB
CHANGE_LIST.md	09/02/2024 16:29	MD File	31 KB
CMakeLists	09/02/2024 16:29	Text Document	8 KB
CODE_OF_CONDUCT.md	09/02/2024 16:29	MD File	6 KB
DemoRunner	09/02/2024 16:29	Application	11.064 KB
LICENSE.md	09/02/2024 16:29	MD File	2 KB
Projucer	09/02/2024 16:29	Application	9.962 KB
README.md	09/02/2024 16:29	MD File	7 KB

The files in the JUCE folder on Mac (left) and Windows (right)

In addition to two applications, **Projucer** and **DemoRunner**, there are several text files and subfolders. You've already briefly seen Projucer and we'll explore DemoRunner in the next section. The files and folders starting with a dot, such as **.clang-tidy**, can be ignored. (These files are hidden from view on Mac but not on Windows.)

Of the text files, **README.md**, is the most important one. The file extension **.md** stands for Markdown. If you don't have a Markdown viewer on your computer, you can open these **.md** files in a regular text editor.

The other text files contain important information about changes that were made to JUCE over time. Sometimes such a change will cause problems with existing code, which is documented in **BREAKING_CHANGES.md**. This is not something you need to concern yourself with right now, but it's good to know where you can find this information.

CMakeLists.txt provides CMake support to JUCE. We won't use CMake in this book so you can safely ignore this file.

Have a look through the subfolders too. The folders are:

- **docs** contains additional JUCE documentation. You can find the main documentation online at docs.juce.com¹⁵.
- **examples** has source code for several example plugins, as well as the demos shown by the DemoRunner application. I suggest coming back to this folder after you've finished this book, as there is a lot to learn from these examples.
- **extras** contains the source code for additional utilities that ship with JUCE, including the code for the Projucer application. For us, the AudioPluginHost app is the most important thing here, which we'll be using to test the plug-in we'll be developing.
- **modules** has the actual source code of the JUCE framework. For efficiency reasons, the framework has been split up into separate modules so that your plug-in only needs to include the modules it requires. For example, we won't be using the `juce_video` module as our plug-in won't need to play videos.

This was just a quick overview of what's inside the JUCE framework. You don't really need to remember any of this. JUCE is structured in such a way that it hides all its inner complexity from the user. All you need to do is use the Projucer application to set up your JUCE project and everything else is handled behind the scenes.

Playing with the JUCE demos

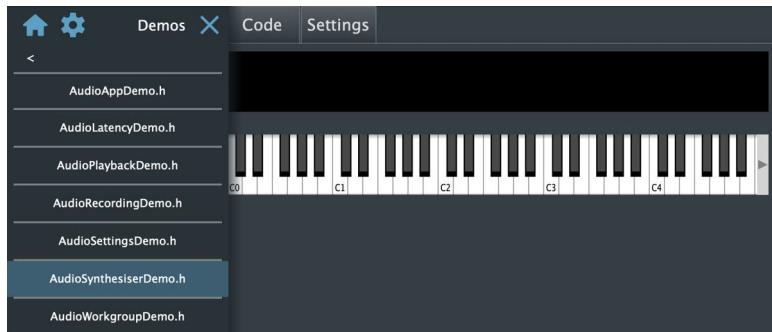
The **DemoRunner** application shows off the capabilities of JUCE. Let's have a play with it to get a sense of what JUCE can do. Double-click to run the app.

- macOS may warn that you're about to open a file that has been downloaded from the internet. Click **Open** to continue. Next, it asks for permission to use the microphone. Click **OK** to grant permission.
- Windows may warn that you're about to open an unrecognized app. Click **More info** and then **Run anyway** to continue.

¹⁵<https://docs.juce.com/>

In the DemoRunner window, click the **Browse Demos** button. This brings up a panel with several categories: Audio, DSP, Utilities, and GUI. Each of these categories has several demo projects inside it.

Click on **Audio** and then select **AudioSynthesiserDemo.h**. The screen changes to the following:

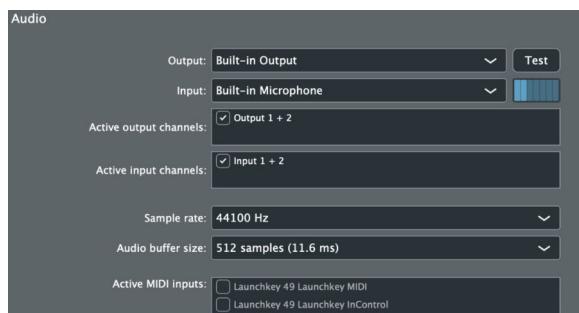


The DemoRunner application

The **Demo** tab shows a piano keyboard. Click on the keys of the piano keyboard to hear a basic synthesizer playing notes. You can choose between a plain sine wave sound or a sampled cello sound.

In addition to clicking on the on-screen piano you can play using the middle row of keys on your computer keyboard. For example, the **A** key plays the note **C4**. If you have a MIDI controller connected to your computer, you can even use that to play with this synth.

If you're not hearing any sound, go to the **Settings** tab. In the **Audio** section you can change the audio devices used for input and output, as well as set up your MIDI controller (if you have one).



The audio settings in DemoRunner

Finally, if you're curious what the source code for any demo looks like, go to the **Code** tab and it will display the C++ code in all its glory. This is not a full source code editor, so you won't be able to make any changes to the code here. It's OK if none of this code makes any sense yet — that's what this book is for!

```

1 #pragma once
2
3 #include "../Assets/DemoUtilities.h"
4 #include "../Assets/AudioLiveScrollingDisplay.h"
5
6 //=====
7 /* Our demo synth sound is just a basic sine wave.. */
8 struct SineWaveSound final : public SynthesiserSound
9 {
10     bool appliesToNote (int /*midiNoteNumber*/) override { return true; }
11     bool appliesToChannel (int /*midiChannel*/) override { return true; }
12 };
13
14 //=====
15 /* Our demo synth voice just plays a sine wave.. */
16 struct SineWaveVoice final : public SynthesiserVoice
17 {
18     bool canPlaySound (SynthesiserSound* sound) override
19     {
20         return dynamic_cast<SineWaveSound*> (sound) != nullptr;
21     }
22
23     void startNote (int midiNoteNumber, float velocity,
24                      SynthesiserSound*, int /*currentPitchWheelPosition*/) override
25     {
26         currentAngle = 0.0;
27         level = velocity * 0.15;
28         tailOff = 0.0;
29
30         auto cyclesPerSecond = MidiMessage::getMidiNoteInHertz (midiNoteNumber);
31         auto cyclesPerSample = cyclesPerSecond / getSampleRate();
32
33         angleDelta = cyclesPerSample * MathConstants<double>::twoPi;
34     }
35 };

```

DemoRunner shows the source code for the selected demo

Give the other demos a try. Some of my favorites are:

- **Audio > PluckedStringsDemo:** This lets you play a harp-like instrument with the mouse.
- **DSP > ConvolutionDemo:** Lets you load an audio file (such as a .wav file) and then applies an impulse response to it, to simulate the audio being played back through a speaker cabinet or cassette recorder.
- **Utilities > Box2DDemo:** JUCE includes the Box2D physics engine, in case you might want to write a game with it, I suppose.
- **GUI > GraphicsDemo:** Demonstrates the graphics drawing capabilities of JUCE.
- **GUI > WidgetsDemo:** Shows the different kinds of user interface controls that are included with JUCE.

The DemoRunner is a handy tool, not just to impress you with what JUCE can do, but also if you ever find yourself wondering how to accomplish something — opening a new window for example — then DemoRunner may have a demonstration of this, including a source code example. A very convenient reference!

The plug-in project

When creating a new piece of software, we usually speak of this in terms of a *project*. The project organizes all the files and configuration options needed to construct the software.

Each IDE has its own way to organize projects. For Visual Studio this is a **.sln** file, which stands for “solution”, and contains one or more **.vcxproj** files. For Xcode this is a **.xcodeproj** bundle, a special type of folder containing the various files that constitute the project.

Let’s say you are going to develop your plug-in on a Mac using Xcode. In theory, since JUCE is a cross-platform framework, the exact same code should run without issues on Windows. That’s great, because it means Windows users will be able to use the plug-in too, without any additional effort from you!

There’s just one small problem: Windows does not have Xcode and Visual Studio cannot open **xcodeproj** projects. Likewise, macOS does not have Visual Studio and Xcode cannot open **sln** projects or **vcxproj** files. You’d need to create a completely new project in Visual Studio and import all your source files.

Having separate Xcode and Visual Studio projects may not sound too bad, but any time you make a change to one of these projects, you’d also have to remember to update the other one. What a headache! And it gets worse when you add Linux, iOS, or Android into the mix.

The idea behind JUCE is that it should be easy to develop your plug-in on one platform, and then just as easy to make it run on another platform. The only way to do this is to take the IDE out of the loop. That’s where **Projucer** comes in.

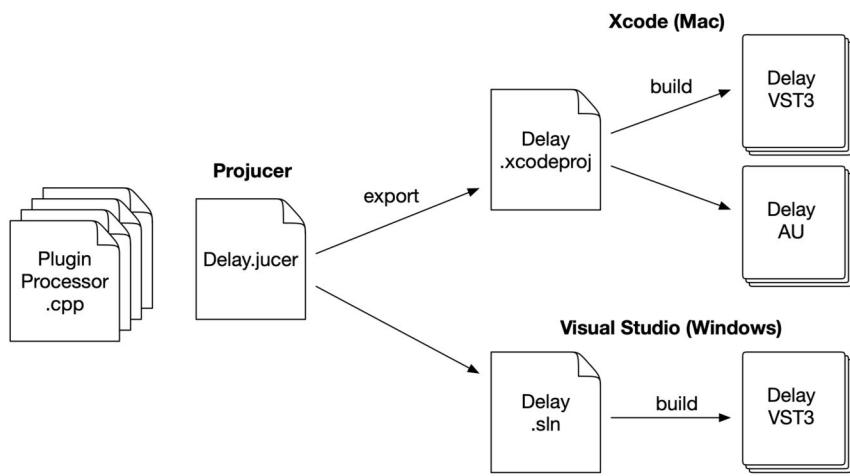
Projucer is a tool that manages JUCE projects in a cross-platform manner. Projucer creates a **.jucer** file that describes everything about the plug-in project:

- The C++ source code files it uses. The project may include source code files in other languages, such as C, Objective-C, Swift, JavaScript, and so on. In this book we’ll restrict ourselves to C++, as learning one language is already enough of a challenge.
- Images, fonts, sound files, and other assets. For example, if your plug-in displays a logo, this could be a PNG or JPEG image that gets embedded inside the plug-in when it is compiled.

- Settings for the compiler and other configuration options that determine how the source code and the other files will be compiled into an actual plug-in.

Projucer is not a full IDE like Xcode or Visual Studio. It only manages your project files, it does not compile the C++ code. So you'll still need to use Xcode or Visual Studio in combination with Projucer.

The figure below illustrates this process:



Projucer manages the project, the IDE builds the plug-in

The workflow is as follows:

1. First, you create the plug-in project using Projucer. The resulting **jucer** file describes which source code files and other files make up the plug-in. You'll see how to do this in the next section.
2. Projucer has a **Save and Open in IDE** button that creates the **xcdeproj** project and opens it in Xcode. On Windows, it creates the **sln** project and opens it in Visual Studio. Also supported are Linux Makefiles and Android Studio projects.
3. On Mac, you use Xcode to edit the source code files and compile the project into a VST3 or AU plug-in.
4. On Windows, you use Visual Studio to edit the source code files and compile the project into a VST3 plug-in.

Most developers prefer either Mac or Windows, so typically you'd do your development just on Mac or just on Windows. Let's say you used Projucer on Mac to create the **jucer** file and you wrote the plug-in's source code in Xcode. To make a Windows version of the plug-in, copy everything to a Windows machine, open the **jucer** file in Projucer on Windows, and export the project to Visual Studio.

Or vice versa: Create the project on Windows and do all your development in Visual Studio. Copy the **jucer** file along with the source code files to Mac when you're ready to make a Mac version of the plug-in, and export to Xcode.

Thanks to Projucer, you don't have to worry about maintaining separate projects for multiple IDEs. Everything in your project is described by the **jucer** file.

The important thing to remember is that every time you want to add a new source file to the project, this needs to be done in Projucer, not in Xcode or Visual Studio. Afterwards, export from Projucer to Xcode or Visual Studio again to update the **xcodeproj** or **sln** files. Likewise for changing the compiler options and other configuration settings.

This approach is a little different to how C++ development normally works. If you have some programming experience, you may have expected to add new source files directly inside Xcode or Visual Studio itself. Any changes that you make to the project settings inside the IDE will be overwritten by Projucer, so it's important to treat the **xcodeproj** and **sln** files as temporary.

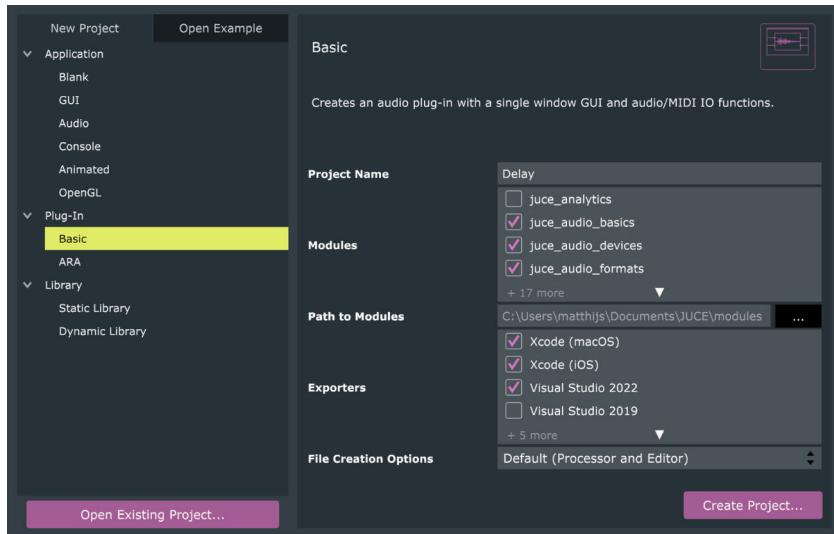
The rule is: Projucer is in charge of the project, not the IDE (Xcode or Visual Studio). This may be a different way of working than you're used to, but it's quite convenient for doing cross-platform development.

When using JUCE, you exclusively use Projucer to create and manage your projects!

Note: Projucer is a handy tool to get started but it can become cumbersome for larger projects. That's why JUCE also supports CMake. This is a command-line tool that serves the same purpose as Projucer but is also used outside of JUCE. CMake is very popular among C++ developers. In this book we're sticking with Projucer but it's worth exploring CMake once you start running into Projucer's limitations.

Creating the project

Go to the **JUCE** folder and open the **Projucer** application. This brings up the **New Project** window:



The New Project window in Projucer

On the left are different options for the kind of project to create. JUCE isn't just for making plug-ins, it can also create full applications — they don't even need to have anything to do with audio.

We want to create a new plug-in project, so select **Basic** under **Plug-In**.

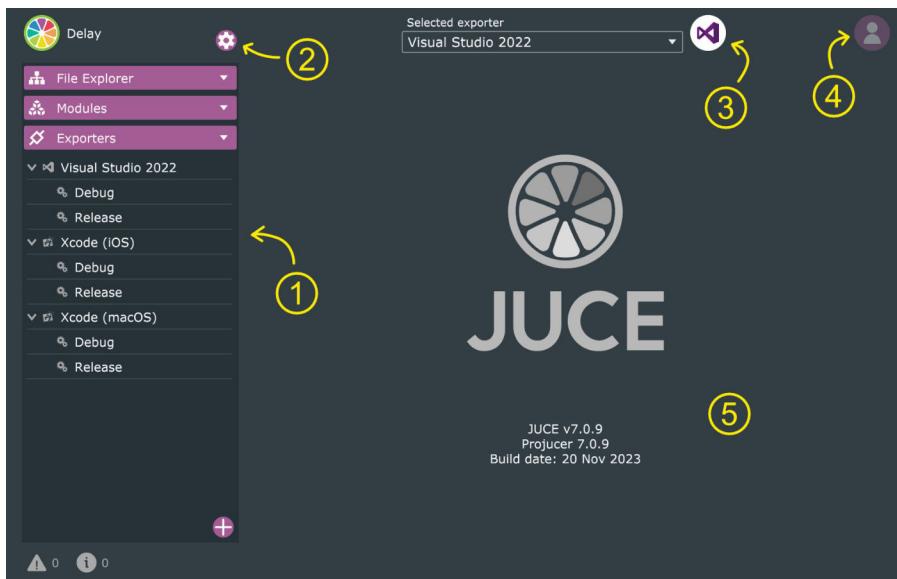
On the right you can fill in the properties of the project:

- **Project Name.** This is where you give the project a name. It says **NewProject** but you can click this to type in something else. Change it to **Delay**. Not a very exciting name, I know, but you can rename the plug-in to something cool later.
- **Modules.** The JUCE modules that will be included. Leave these to the default selection for now.
- **Exporters.** If you're on Mac, make sure **Xcode (macOS)** is selected here. To also build an iOS version of the plug-in, select **Xcode (iOS)** as well. Readers on Windows should select **Visual Studio 2022**.

I selected all three exporters, since I want to build versions for Mac, iOS, and Windows from the same **jucer** project.

When you're done, click the purple **Create Project...** button and choose where to save the project. This will automatically create a new folder.

Once the project is saved, the Projucer window changes to show the newly created project:



The new project

The elements of this window are:

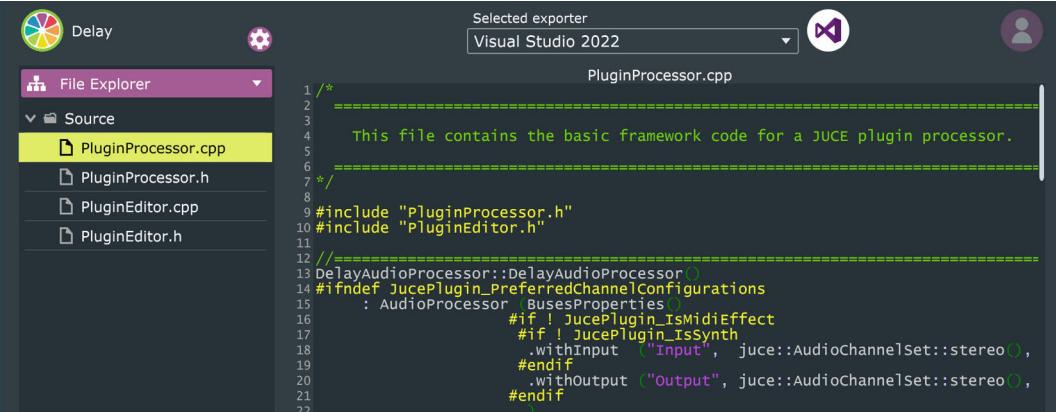
1. The contents of the project. This lists the C++ source code files, the JUCE modules that are included, and the IDEs you will be using.
2. The gear button brings up additional project settings.
3. With this button you export the Projucer project to Visual Studio or Xcode.
4. User account sign in at juce.com. (More about this in a later chapter.)
5. The main working area. Initially, this shows the JUCE logo and version number. When you select an item in the list on the left, its properties will be shown here.

Note: If at this point there is a small pop-up in the bottom-left corner of the Projucer window that says, “The path to your JUCE folder is incorrect,” it means that Projucer cannot find where the JUCE folder is installed. To remedy this, click on **Set path** to open the **Global Paths** window and make sure the **Path to JUCE** and **JUCE Modules** fields point to the place on your hard drive where you put the JUCE folder (see earlier in this chapter).

Getting to know Projucer

The panel on the left of the Projucer window has three sections: File Explorer, Modules, and Exporters. Initially, the Exporters section is expanded. You can open the other sections by clicking their purple headers.

Let's look at the **File Explorer** first:



The screenshot shows the Projucer application window. On the left, the 'File Explorer' panel is open, displaying a tree view of project files under a 'Source' folder. The files listed are PluginProcessor.cpp, PluginProcessor.h, PluginEditor.cpp, and PluginEditor.h. The file 'PluginProcessor.cpp' is currently selected, indicated by a yellow highlight. On the right, the main workspace shows the source code for 'PluginProcessor.cpp'. The code includes comments explaining the basic framework for a JUCE plugin processor, defines for DelayAudioProcessor, and specific configurations for JucePlugin_PREFERREDCHANNELCONFIGURATIONS. The code editor has syntax highlighting and line numbers.

```

Selected exporter
Visual Studio 2022
PluginProcessor.cpp

/*
=====
This file contains the basic framework code for a JUCE plugin processor.
=====

*/
#ifndef JucePlugin_PREFERREDCHANNELCONFIGURATIONS
: AudioProcessor (BusesProperties)
    #if ! JucePlugin_IsMidiEffect
        #if ! JucePlugin_IsSynth
            .withInput ("Input", juce::AudioChannelSet::stereo),
        #endif
        .withOutput ("Output", juce::AudioChannelSet::stereo),
    #endif
}
#endif

```

The File Explorer shows the source files

The File Explorer lists the C++ source code files that are part of the project, as well as any image files, sound files, fonts, and so on. Since this is a new project, it has just four files inside a folder named **Source**. Later you will add more source files to the project, and the File Explorer is where you will do that.

In the above image, the file **PluginProcessor.cpp** has been selected. This displays the source code from that file on the right. You can edit the source code here, although this editor is very limited compared to a full IDE like Xcode or Visual Studio, and you cannot compile the code from Projucer anyway.

We'll spend a lot more time with these source files later, but so you know what's going on, this is what they are for:

- **PluginProcessor.cpp** and **PluginProcessor.h**: These contain the code for the `DelayAudioProcessor` object, which is the part of the plug-in that will do the actual audio processing. It will receive audio from the host, change it in some way, and send the processed audio back to the host.
- **PluginEditor.cpp** and **PluginEditor.h**: The `DelayAudioProcessorEditor` object, which handles the plug-in's user interface, also known as the editor. This will draw all the knobs and buttons and text on the screen, and allows the user to interact with them.

Feel free to pause reading at this point and scroll through these source code files to get an idea of what is inside them. It's OK if this looks like gobbledegook right now. It will start to make more sense over the coming chapters.

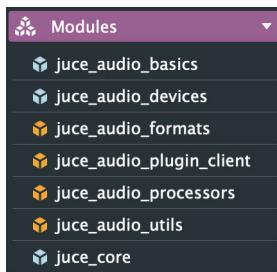
Notice that the code has been split up into **.cpp** and **.h** files. This is typical for the C++ language.

- The **h** file is known as the “header” and is a declaration of what sort of things your program is made out of.
- The **cpp** file, known as the source file or the implementation file, defines how those things actually work.

Sometimes there is only a **.h** file and no corresponding **cpp** file. More modern languages don't have this split between header and source files, as it's kind of annoying to use, and even upcoming versions of C++ attempt to do away with it. For now, just know that we'll usually need both of these files to write C++ code.

Note: Some developers use the extension **.hh** or **.hpp** on their header files and **.cc** or **.cxx** for the implementation files. It means the same thing. I merely mention it because you might come across this when looking at other people's code.

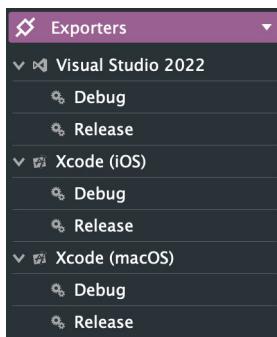
The **Modules** panel lists the JUCE modules that are included in this plug-in. To keep things flexible, the JUCE framework is made up of separate modules that each do a specific thing. You only need to include the modules your project needs.



The Modules panel

Since this is an audio plug-in project, it includes various `juce_audio` modules as well as `juce_graphics` and `juce_gui` modules for creating the plug-in's user interface. If the project needs additional modules, you can add them here. You can even create your own modules with reusable code.

Finally, the **Exporters** panel lists the IDEs that you'll be using to build this plug-in. In the screenshot below it shows three exporters since I'm planning to build my plug-in for Mac, iOS, and Windows.



The Exporters panel

Below each exporter are two configurations, **Debug** and **Release**.

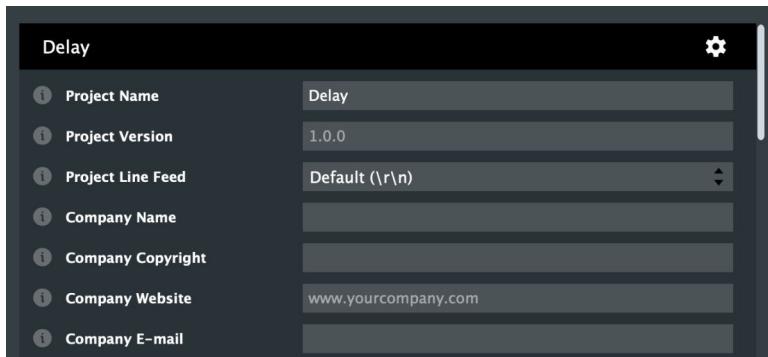
- You will use the Debug configuration while developing the plug-in. This adds useful information during the compilation process that can help the debugger figure out what went wrong when the program crashes.
- The Release configuration is for the final version of the plug-in that will be distributed to end users. This strips all the debugging information from the plug-in (to make it smaller) and compiles it with optimizations enabled (to make it faster).

For example, if you select the **Debug** configuration under the **Xcode** exporter and scroll down to the **Optimisation** setting, it will say **-O0 (no optimisation)**. But under the **Release** configuration the **Optimisation** option is set to **-O3 (fastest with safe optimisations)**.

Likewise, if you select the **Debug** configuration under the **Visual Studio** exporter and scroll to the **Optimisation** setting, it will say **Disabled (/Od)**. But under the **Release** configuration the **Optimisation** option is set to **Full optimisation (/Ox)**.

Configuring the project

We will now complete setting up our brand-new plug-in project. Click on the purple **gear icon** at the top of the Projucer window to bring up the project settings pane:



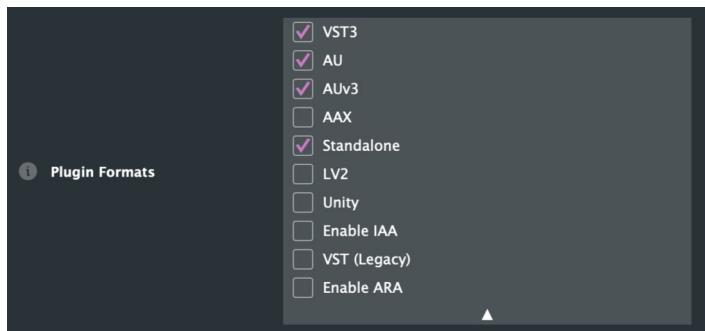
The project settings in Projucer

Here you will fill in the details of what the plug-in does. There are a lot of settings but you can keep the defaults for most of these fields. If you're ever wondering what a setting is for, click the **(i)** button next to its name.

In particular, pay attention to the following settings:

Plugin Formats. There are multiple plug-in formats in common use and JUCE can build all of them from the same codebase. By default this has **VST3** and **AU** (Audio Units) selected. If you want to build for iOS as well, then also select **AUv3** here.

Click the small downward arrow to expand this section and make sure **Standalone** is selected too. This will create a self-contained app that hosts the plug-in, which is very useful for debugging and something we'll use in a later chapter.



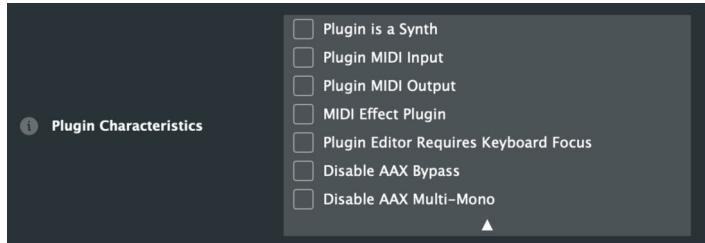
The Plugin Formats setting

Besides VST3 and AU, many plug-in developers make AAX versions of their plug-ins. The AAX format is for Pro Tools, but requires getting in touch with AVID to obtain a special developer version of Pro Tools, and is outside the scope of this book.

You can always change the **Plugin Formats** setting later, so if you're starting out with just VST3, AU, and Standalone, but decide later you want to add AUV3 for iOS or AAX, simply tick those boxes and export the project again.

Note: Even though on Windows you can select AU or AUV3, you won't be able to build those formats on a PC — those are Mac-only technologies.

Plugin Characteristics. In this book we are going to be building an effect plug-in. It's possible to build other kinds of plug-ins with JUCE, such as a synth or a MIDI effect. For our purposes, all these options must be left blank.



The Plugin Characteristics setting

Plugin Name. If you thought my choice of "Delay" for the name of this plug-in is waaay too cringe, then fill in the much better name you came up with here (MySuperAwesomeDelay?). It will show up under this name in the DAW.

Plugin Manufacturer. This is where you put your company name. Plug-ins are identified by a combination of their own name and the name of their manufacturer. By default, JUCE chooses **yourcompany** as the manufacturer name. You can leave this as is or fill in your own name.

If you change this option, make sure to use the new name in the **Bundle Identifier** option as well.



The plug-in and manufacturer code settings

Plugin Manufacturer Code. This is an ID that uniquely identifies your company. It must be four characters and have at least one uppercase letter. The default is **Manu**. If you changed the Plugin Manufacturer option, you should change this one as well.

Plugin Code. The plug-in itself should also have a unique four-character ID that starts with an uppercase letter. Projucer generates this code randomly. Change this to **Dlay**.

C++ Language Standard. Set this to **C++20**. This is the version number of the C++ language that we will be using.

This is a good point to save your changes using the menu option **File > Save Project**. Sometimes Projucer will spontaneously crash and you don't want to lose your work.

Note: If the Save Project menu item is grayed out or if you get an error message saying, "Error writing to file... Save and export is disabled." then change the setting **Display the JUCE Splash Screen** to **Enabled**.

Next, you'll change some of the exporter settings. Open the **Exporters** panel and select the **Debug** entry under **Xcode (macOS)**. Review the following:

- **Enable Plugin Copy Step.** Make sure this is set to **Enabled**. This will copy the plug-in into the `~/Library/Audio/Plug-Ins` folder after compilation, so it can be found by hosts.

- **Add Recommended Compiler Warning Flags.** Set this to **Enabled** to enable additional warnings for the compiler.

For Visual Studio, select the **Debug** entry under **Visual Studio 2022**. Review the following options:

- **Enable Plugin Copy Step.** Set this to **Enabled**. This will copy the plug-in into the `C:\Program Files\Common Files\VST3` folder after compilation, so it can be found by hosts.
- **Warning Level.** Make sure this is set to **High** to enable additional warnings for the compiler.

Having the compiler emit plenty of warnings is useful for avoiding silly programming mistakes. These recommended warnings can be pretty pedantic and you might want to disable this option for your own projects, but as this is a beginner book we'll take all the help we can get.

Phew, that should do it for now. There are loads more settings that you could change, but many of these are for advanced use cases, or for when you're ready to release the final version of the plug-in.

Make sure to save the Projucer project again before continuing.

The next step is to export the Projucer project so it can be opened in your IDE. The following chapters will explain in detail how to install Xcode (for Mac) and Visual Studio (for Windows), and how to get started using these IDEs.

4: Getting started with Xcode

This chapter is for Mac users. If you're on a Windows PC, skip to the next chapter.

Required versions

The screenshots in this chapter were made with Xcode 15.2 on macOS 14 (Sonoma), using JUCE 7.0.9. The plug-in was also tested with a preview release of JUCE 8.

By the time you're reading this book, Xcode, macOS and JUCE may all have newer versions. I suggest always using the latest versions available. Just keep in mind that what you see on your screen might be a little different from the pictures in the book.

If you have an older Mac you may not be able to install the latest versions of Xcode and macOS. That's OK too. JUCE 7 requires a minimum of Xcode 10.1 and macOS 10.13.6 (High Sierra). JUCE 8 requires at least Xcode 12.4 and macOS 10.15.4 (Catalina).

As long as your computer is less than ten years old it should be able to run all the required software. In that case, install the best available version of Xcode for your computer.



Installing Xcode

Xcode is the IDE for developers using a Mac computer. It can be downloaded for free from Apple.

There are two ways to obtain Xcode:

1. Using the Mac App Store.
2. Direct download from Apple's developer website.

The easiest method is to use the Mac App Store, but many developers prefer to download from the Apple developer site as this gives more control over which version of Xcode gets installed, as well as access to beta versions.

To download Xcode you will need an **Apple ID**. You can use your existing Apple ID for this. In the unlikely case you don't have an Apple ID yet, you can get one for free at appleid.apple.com¹⁶.

Apple has a paid Developer Program, which costs \$99 per year. You do *not* need to be a member of the Developer Program to download and start using Xcode!

Developer Program membership is only required to distribute your finished plug-in to other users. To run the plug-in on your own computer you don't have to pay for anything.

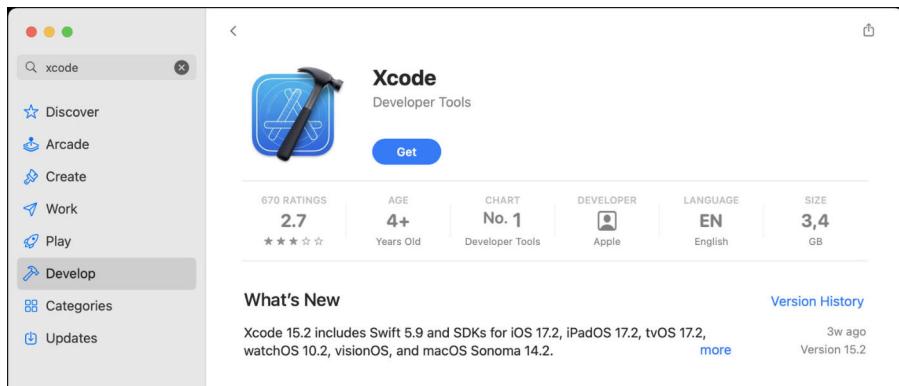
More about the Apple Developer Program in Chapter 19 where we'll talk about releasing your plug-in into the world.

Method 1: Using the Mac App Store

Open the **App Store** from the Apple icon in the menu bar. Inside the App Store app, search for **xcode** or navigate to the **Develop** category where Xcode is usually ranked number one. Don't let the bad rating scare you off, developers are hard to please.

Click the **Get** button to download Xcode to your computer. The App Store will ask you to sign in with your Apple ID if you aren't signed in yet.

¹⁶<https://appleid.apple.com/>



Xcode on the Mac App Store

Installing Xcode can take a while. It's a big download (several gigabytes) and the installation process is not very fast. This is probably a good time to make yourself a cup of coffee.

Once the installation finishes, there is a new icon for **Xcode** in your **Applications** folder.



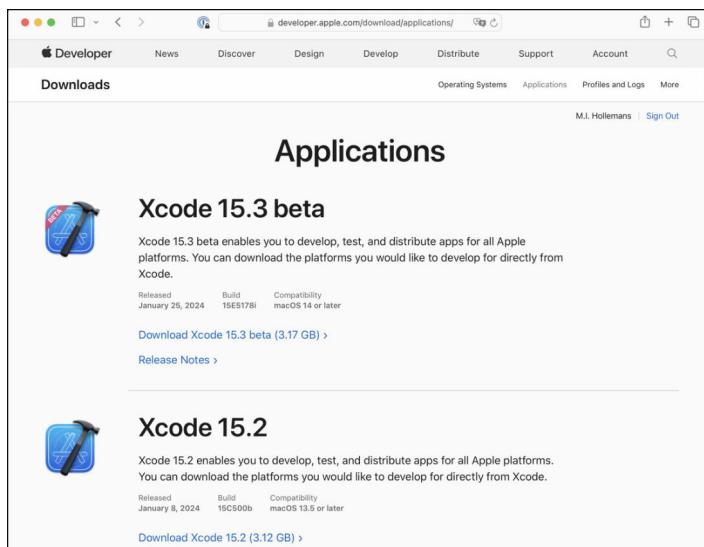
The Xcode icon in the Applications folder

Tip: You may want to disable Automatic Updates in the App Store's settings, otherwise your computer will automatically install every update to Xcode that gets released. This involves downloading the entire thing again. It's better to manually decide when Xcode gets updated.

Method 2: Downloading from Apple's developer website

Alternatively, you can download Xcode from Apple's developer portal.

Go to developer.apple.com¹⁷ and sign in with your Apple ID. After signing in, go to **Applications** (in the top-right corner) to list the available versions of Xcode.



Xcode on the Apple developer website

There are two things to watch out for:

1. You will want to download the latest stable version, not one with “beta” in the name. Beta versions often have bugs and aren’t guaranteed to work correctly.
2. Make sure the version you download is compatible with your macOS, see under “Compatibility”. For example, Xcode 15.2 is “macOS 13.5 or later”.

Click the download link to begin the download. After the huge download finishes, you will have a file named **Xcode_version.xip** in your **Downloads** folder (where “version” is the actual version number). Double-click this file to unpack it. This will take a while!

Once extracting the xip file has completed, there is a new file **Xcode.app**. Drag this into your **Applications** folder to finish the installation.

¹⁷<https://developer.apple.com/download/>

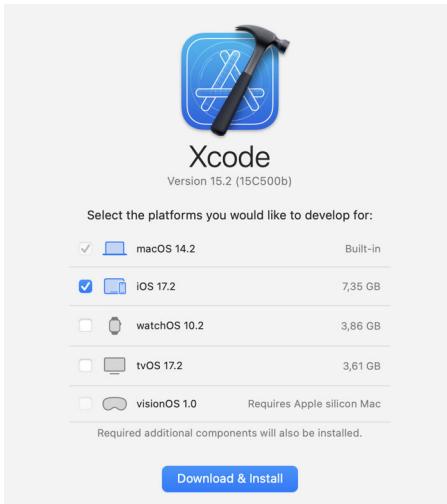
First time running Xcode

Click the **Xcode** icon to launch it. The very first time this might take a few moments. Xcode will also pop up a few dialogs during this process.

You must agree to the license agreement. Click **Agree**.

Xcode may ask for admin privileges to install additional components. Enter your computer's password and click **OK**.

You must select the platforms you wish to develop for. **macOS** is always selected. Select **iOS** if you have an iPhone or iPad and want to run your plug-ins on these devices.

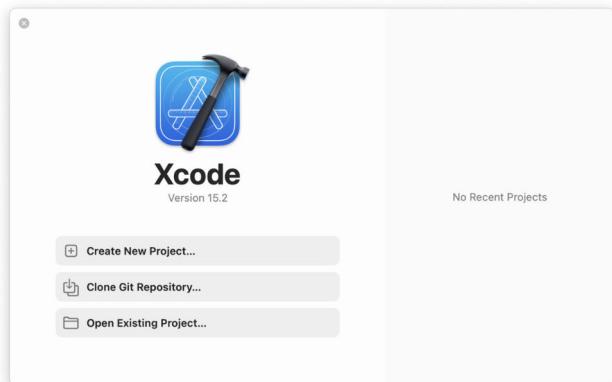


Choosing the platforms

Now Xcode will do some more installing, downloading the iOS simulators if you enabled the iOS option.

If a What's New in Xcode screen pops up, simply click **Continue** to dismiss it.

Finally, the Xcode start screen will appear where you can create a new project or open an existing project. Excellent! That completes the installation of the IDE.

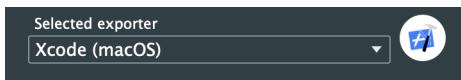


The Xcode welcome screen

Exporting the Projucer project

In the last chapter, you used Projucer to create and configure the plug-in project. Now that Xcode is installed, you can export from Projucer to Xcode, so that you can finally write some code and compile the plug-in.

This is done using the exporter section at the top of Projucer:



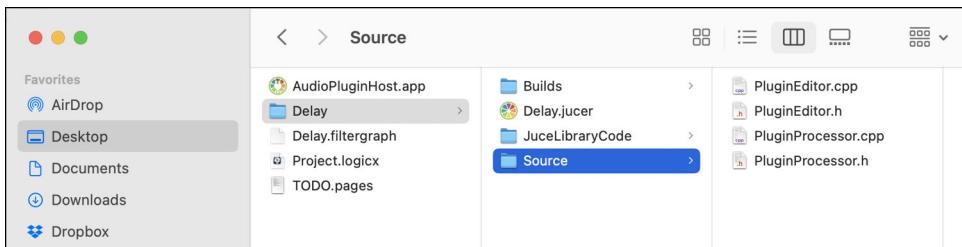
The selected exporter

If you have added more than one exporter to the project you can switch between them here. Make sure it's set to **Xcode (macOS)**.

Click the **circular button** to start Xcode and load the project. This is also known as the **Save and Open in IDE** button, or just the **export button**.

Note: The export button will become disabled when you select the Visual Studio exporter on a Mac. You'll have to open the **jucer** file in Projucer on a Windows machine for that.

Before we start using Xcode, let's see what the project looks like in the file system. I saved my project on the Desktop, and opening that folder shows the following.



The generated files on macOS

The Delay folder contains:

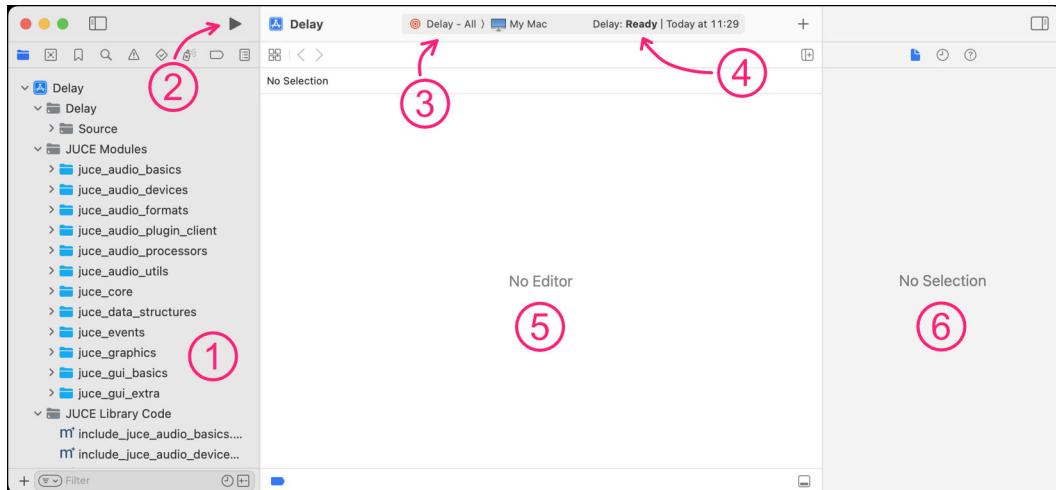
- **Delay.jucer.** This is the Projucer project. Whenever you need to make changes to the project, such as adding new source code files or changing any of the settings, this file is what you open. (If your computer does not show file extensions, note that **jucer** files have the same colorful icon as Projucer.)
- **Builds.** This folder is where Projucer has saved the Xcode and Visual Studio projects. Take a look inside. There will be iOS, MacOSX, and VisualStudio2022 subfolders, depending on which exporters you added in Projucer. Inside the **MacOSX** folder is the **Delay.xcodeproj** file. This is also where the compiled binaries go when you build the plug-in.
- **JuceLibraryCode.** This contains the JUCE modules that will be added to the project. You should not edit any of these files yourself, they will be overwritten by Projucer the next time you export.
- **Source.** These are the C++ source code files for your plug-in. It contains the four source files you already saw in the File Explorer in Projucer. When you add new files to the project in Projucer, they go into this folder.

Tip: You can safely delete the Builds and JuceLibraryCode folders at any time. Sometimes removing these folders is even necessary, for example if you get weird compilation errors after updating to a new version of JUCE. After removing these folders, simply open the **jucer** file in Projucer again and do a new export. This will reconstruct the Builds and JuceLibraryCode folders.

Note: If you are familiar with version control tools such as git, you typically would add the Builds and JuceLibraryCode folders to your `.gitignore` file so that they are not included in the repository.

A quick tour of Xcode

After exporting the project from Projucer, it will open in Xcode. That should look something like the following.



The new project in Xcode

Xcode's window is divided into different sections:

1. The Project Navigator shows the files that make up the project.
2. The “play” button is used to build the project.
3. This control selects the active target. In the screenshot this is **Delay - All** but on your computer a different target may be selected.
4. A status indicator that shows warnings, errors, and other messages.
5. The text editor. This is where you will edit the source code files.
6. Properties of the selected item. Right now there's nothing selected.

In the **Project Navigator** on the left, expand the **Source** folder (under **Delay**). This reveals the four C++ source code files that were also visible in Projucer's File Explorer. Selecting a file from this list will display its source code in the text editor.

```

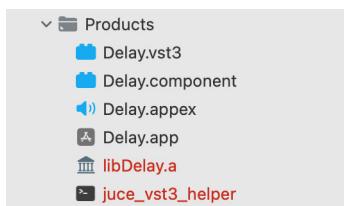
1 /*
2 =====
3 This file contains the basic framework code for a JUCE plugin processor.
4 =====
5 */
6 #include "PluginProcessor.h"
7 #include "PluginEditor.h"
8 //=====
9 #include "DelayAudioProcessor.h"
10 #ifndef JucePlugin_PREFERREDCHANNELCONFIGURATIONS
11     : AudioProcessor (BusesProperties()
12         .if ! JucePlugin_IsMidiEffect
13             .if ! JucePlugin_IsSynth
14                 .withInput ("Input", juce::AudioChannelSet::stereo(), true)
15             .endif
16             .withOutput ("Output", juce::AudioChannelSet::stereo(), true)
17         .endif
18     )
19     .withBuses (BusesProperties()
20         .withInput ("Input", juce::AudioChannelSet::stereo(), true)
21         .withOutput ("Output", juce::AudioChannelSet::stereo(), true)
22     )
23     .withMidi (MidiProperties()
24         .withInstruments (InstrumentsProperties()
25             .withInstrument ("Instrument", juce::MidiInstrument (60, 127))
26         )
27         .withOutstruments (InstrumentsProperties()
28             .withInstrument ("Instrument", juce::MidiInstrument (60, 127))
29         )
30     )
31     .withVst3 (Vst3Properties()
32         .withVst3Format (Vst3FormatProperties()
33             .withFormat ("Format", juce::Vst3Format::vst3)
34         )
35     )
36     .withAup (AupProperties()
37         .withAupFormat (AupFormatProperties()
38             .withFormat ("Format", juce::AupFormat::auv3)
39         )
40     )
41     .withStandalone (StandaloneProperties()
42         .withStandaloneFormat (StandaloneFormatProperties()
43             .withFormat ("Format", juce::StandaloneFormat::standalone)
44         )
45     )
46     .withJucer (JucerProperties()
47         .withJucerFormat (JucerFormatProperties()
48             .withFormat ("Format", juce::JucerFormat::jucer)
49         )
50     )
51     .withXcode (XcodeProperties()
52         .withXcodeFormat (XcodeFormatProperties()
53             .withFormat ("Format", juce::XcodeFormat::xcode)
54         )
55     )
56     .withProtools (ProtoolsProperties()
57         .withProtoolsFormat (ProtoolsFormatProperties()
58             .withFormat ("Format", juce::ProtoolsFormat::protools)
59         )
60     )
61     .withAaf (AafProperties()
62         .withAafFormat (AafFormatProperties()
63             .withFormat ("Format", juce::AafFormat::aaf)
64         )
65     )
66     .withWav (WavProperties()
67         .withWavFormat (WavFormatProperties()
68             .withFormat ("Format", juce::WavFormat::wav)
69         )
70     )
71     .withMp3 (Mp3Properties()
72         .withMp3Format (Mp3FormatProperties()
73             .withFormat ("Format", juce::Mp3Format::mp3)
74         )
75     )
76     .withOgg (OggProperties()
77         .withOggFormat (OggFormatProperties()
78             .withFormat ("Format", juce::OggFormat::ogg)
79         )
80     )
81     .withFlac (FlacProperties()
82         .withFlacFormat (FlacFormatProperties()
83             .withFormat ("Format", juce::FlacFormat::flac)
84         )
85     )
86     .withWavPack (WavPackProperties()
87         .withWavPackFormat (WavPackFormatProperties()
88             .withFormat ("Format", juce::WavPackFormat::wavPack)
89         )
90     )
91     .withAac (AacProperties()
92         .withAacFormat (AacFormatProperties()
93             .withFormat ("Format", juce::AacFormat::aac)
94         )
95     )
96     .withOpus (OpusProperties()
97         .withOpusFormat (OpusFormatProperties()
98             .withFormat ("Format", juce::OpusFormat::opus)
99         )
100    )
101   )
102   .withMidiConnections (MidiConnectionsProperties()
103       .withMidiConnection ("Midi Connection", juce::MidiConnection (juce::MidiConnection::input))
104   )
105   .withVst3Connections (Vst3ConnectionsProperties()
106       .withVst3Connection ("Vst3 Connection", juce::Vst3Connection (juce::Vst3Connection::input))
107   )
108   .withAupConnections (AupConnectionsProperties()
109       .withAupConnection ("Aup Connection", juce::AupConnection (juce::AupConnection::input))
110   )
111   .withStandaloneConnections (StandaloneConnectionsProperties()
112       .withStandaloneConnection ("Standalone Connection", juce::StandaloneConnection (juce::StandaloneConnection::input))
113   )
114   .withJucerConnections (JucerConnectionsProperties()
115       .withJucerConnection ("Jucer Connection", juce::JucerConnection (juce::JucerConnection::input))
116   )
117   .withXcodeConnections (XcodeConnectionsProperties()
118       .withXcodeConnection ("Xcode Connection", juce::XcodeConnection (juce::XcodeConnection::input))
119   )
120   .withProtoolsConnections (ProtoolsConnectionsProperties()
121       .withProtoolsConnection ("Protools Connection", juce::ProtoolsConnection (juce::ProtoolsConnection::input))
122   )
123   .withAafConnections (AafConnectionsProperties()
124       .withAafConnection ("Aaf Connection", juce::AafConnection (juce::AafConnection::input))
125   )
126   .withWavConnections (WavConnectionsProperties()
127       .withWavConnection ("Wav Connection", juce::WavConnection (juce::WavConnection::input))
128   )
129   .withMp3Connections (Mp3ConnectionsProperties()
130       .withMp3Connection ("Mp3 Connection", juce::Mp3Connection (juce::Mp3Connection::input))
131   )
132   .withOggConnections (OggConnectionsProperties()
133       .withOggConnection ("Ogg Connection", juce::OggConnection (juce::OggConnection::input))
134   )
135   .withFlacConnections (FlacConnectionsProperties()
136       .withFlacConnection ("Flac Connection", juce::FlacConnection (juce::FlacConnection::input))
137   )
138   .withWavPackConnections (WavPackConnectionsProperties()
139       .withWavPackConnection ("Wav Pack Connection", juce::WavPackConnection (juce::WavPackConnection::input))
140   )
141   .withAacConnections (AacConnectionsProperties()
142       .withAacConnection ("Aac Connection", juce::AacConnection (juce::AacConnection::input))
143   )
144   .withOpusConnections (OpusConnectionsProperties()
145       .withOpusConnection ("Opus Connection", juce::OpusConnection (juce::OpusConnection::input))
146   )
147   .withJucerConnections (JucerConnectionsProperties()
148       .withJucerConnection ("Jucer Connection", juce::JucerConnection (juce::JucerConnection::input))
149   )
150   .withXcodeConnections (XcodeConnectionsProperties()
151       .withXcodeConnection ("Xcode Connection", juce::XcodeConnection (juce::XcodeConnection::input))
152   )
153   .withProtoolsConnections (ProtoolsConnectionsProperties()
154       .withProtoolsConnection ("Protools Connection", juce::ProtoolsConnection (juce::ProtoolsConnection::input))
155   )
156   .withAafConnections (AafConnectionsProperties()
157       .withAafConnection ("Aaf Connection", juce::AafConnection (juce::AafConnection::input))
158   )
159   .withWavConnections (WavConnectionsProperties()
160       .withWavConnection ("Wav Connection", juce::WavConnection (juce::WavConnection::input))
161   )
162   .withMp3Connections (Mp3ConnectionsProperties()
163       .withMp3Connection ("Mp3 Connection", juce::Mp3Connection (juce::Mp3Connection::input))
164   )
165   .withOggConnections (OggConnectionsProperties()
166       .withOggConnection ("Ogg Connection", juce::OggConnection (juce::OggConnection::input))
167   )
168   .withFlacConnections (FlacConnectionsProperties()
169       .withFlacConnection ("Flac Connection", juce::FlacConnection (juce::FlacConnection::input))
170   )
171   .withWavPackConnections (WavPackConnectionsProperties()
172       .withWavPackConnection ("Wav Pack Connection", juce::WavPackConnection (juce::WavPackConnection::input))
173   )
174   .withAacConnections (AacConnectionsProperties()
175       .withAacConnection ("Aac Connection", juce::AacConnection (juce::AacConnection::input))
176   )
177   .withOpusConnections (OpusConnectionsProperties()
178       .withOpusConnection ("Opus Connection", juce::OpusConnection (juce::OpusConnection::input))
179   )
180   .withJucerConnections (JucerConnectionsProperties()
181       .withJucerConnection ("Jucer Connection", juce::JucerConnection (juce::JucerConnection::input))
182   )
183   .withXcodeConnections (XcodeConnectionsProperties()
184       .withXcodeConnection ("Xcode Connection", juce::XcodeConnection (juce::XcodeConnection::input))
185   )
186   .withProtoolsConnections (ProtoolsConnectionsProperties()
187       .withProtoolsConnection ("Protools Connection", juce::ProtoolsConnection (juce::ProtoolsConnection::input))
188   )
189   .withAafConnections (AafConnectionsProperties()
190       .withAafConnection ("Aaf Connection", juce::AafConnection (juce::AafConnection::input))
191   )
192   .withWavConnections (WavConnectionsProperties()
193       .withWavConnection ("Wav Connection", juce::WavConnection (juce::WavConnection::input))
194   )
195   .withMp3Connections (Mp3ConnectionsProperties()
196       .withMp3Connection ("Mp3 Connection", juce::Mp3Connection (juce::Mp3Connection::input))
197   )
198   .withOggConnections (OggConnectionsProperties()
199       .withOggConnection ("Ogg Connection", juce::OggConnection (juce::OggConnection::input))
200   )
201   .withFlacConnections (FlacConnectionsProperties()
202       .withFlacConnection ("Flac Connection", juce::FlacConnection (juce::FlacConnection::input))
203   )
204   .withWavPackConnections (WavPackConnectionsProperties()
205       .withWavPackConnection ("Wav Pack Connection", juce::WavPackConnection (juce::WavPackConnection::input))
206   )
207   .withAacConnections (AacConnectionsProperties()
208       .withAacConnection ("Aac Connection", juce::AacConnection (juce::AacConnection::input))
209   )
210   .withOpusConnections (OpusConnectionsProperties()
211       .withOpusConnection ("Opus Connection", juce::OpusConnection (juce::OpusConnection::input))
212   )
213   .withJucerConnections (JucerConnectionsProperties()
214       .withJucerConnection ("Jucer Connection", juce::JucerConnection (juce::JucerConnection::input))
215   )
216   .withXcodeConnections (XcodeConnectionsProperties()
217       .withXcodeConnection ("Xcode Connection", juce::XcodeConnection (juce::XcodeConnection::input))
218   )
219   .withProtoolsConnections (ProtoolsConnectionsProperties()
220       .withProtoolsConnection ("Protools Connection", juce::ProtoolsConnection (juce::ProtoolsConnection::input))
221   )
222   .withAafConnections (AafConnectionsProperties()
223       .withAafConnection ("Aaf Connection", juce::AafConnection (juce::AafConnection::input))
224   )
225   .withWavConnections (WavConnectionsProperties()
226       .withWavConnection ("Wav Connection", juce::WavConnection (juce::WavConnection::input))
227   )
228   .withMp3Connections (Mp3ConnectionsProperties()
229       .withMp3Connection ("Mp3 Connection", juce::Mp3Connection (juce::Mp3Connection::input))
230   )
231   .withOggConnections (OggConnectionsProperties()
232       .withOggConnection ("Ogg Connection", juce::OggConnection (juce::OggConnection::input))
233   )
234   .withFlacConnections (FlacConnectionsProperties()
235       .withFlacConnection ("Flac Connection", juce::FlacConnection (juce::FlacConnection::input))
236   )
237   .withWavPackConnections (WavPackConnectionsProperties()
238       .withWavPackConnection ("Wav Pack Connection", juce::WavPackConnection (juce::WavPackConnection::input))
239   )
240   .withAacConnections (AacConnectionsProperties()
241       .withAacConnection ("Aac Connection", juce::AacConnection (juce::AacConnection::input))
242   )
243   .withOpusConnections (OpusConnectionsProperties()
244       .withOpusConnection ("Opus Connection", juce::OpusConnection (juce::OpusConnection::input))
245   )
246   .withJucerConnections (JucerConnectionsProperties()
247       .withJucerConnection ("Jucer Connection", juce::JucerConnection (juce::JucerConnection::input))
248   )
249   .withXcodeConnections (XcodeConnectionsProperties()
250       .withXcodeConnection ("Xcode Connection", juce::XcodeConnection (juce::XcodeConnection::input))
251   )
252   .withProtoolsConnections (ProtoolsConnectionsProperties()
253       .withProtoolsConnection ("Protools Connection", juce::ProtoolsConnection (juce::ProtoolsConnection::input))
254   )
255   .withAafConnections (AafConnectionsProperties()
256       .withAafConnection ("Aaf Connection", juce::AafConnection (juce::AafConnection::input))
257   )
258   .withWavConnections (WavConnectionsProperties()
259       .withWavConnection ("Wav Connection", juce::WavConnection (juce::WavConnection::input))
260   )
261   .withMp3Connections (Mp3ConnectionsProperties()
262       .withMp3Connection ("Mp3 Connection", juce::Mp3Connection (juce::Mp3Connection::input))
263   )
264   .withOggConnections (OggConnectionsProperties()
265       .withOggConnection ("Ogg Connection", juce::OggConnection (juce::OggConnection::input))
266   )
267   .withFlacConnections (FlacConnectionsProperties()
268       .withFlacConnection ("Flac Connection", juce::FlacConnection (juce::FlacConnection::input))
269   )
270   .withWavPackConnections (WavPackConnectionsProperties()
271       .withWavPackConnection ("Wav Pack Connection", juce::WavPackConnection (juce::WavPackConnection::input))
272   )
273   .withAacConnections (AacConnectionsProperties()
274       .withAacConnection ("Aac Connection", juce::AacConnection (juce::AacConnection::input))
275   )
276   .withOpusConnections (OpusConnectionsProperties()
277       .withOpusConnection ("Opus Connection", juce::OpusConnection (juce::OpusConnection::input))
278   )
279   .withJucerConnections (JucerConnectionsProperties()
280       .withJucerConnection ("Jucer Connection", juce::JucerConnection (juce::JucerConnection::input))
281   )
282   .withXcodeConnections (XcodeConnectionsProperties()
283       .withXcodeConnection ("Xcode Connection", juce::XcodeConnection (juce::XcodeConnection::input))
284   )
285   .withProtoolsConnections (ProtoolsConnectionsProperties()
286       .withProtoolsConnection ("Protools Connection", juce::ProtoolsConnection (juce::ProtoolsConnection::input))
287   )
288   .withAafConnections (AafConnectionsProperties()
289       .withAafConnection ("Aaf Connection", juce::AafConnection (juce::AafConnection::input))
290   )
291   .withWavConnections (WavConnectionsProperties()
292       .withWavConnection ("Wav Connection", juce::WavConnection (juce::WavConnection::input))
293   )
294   .withMp3Connections (Mp3ConnectionsProperties()
295       .withMp3Connection ("Mp3 Connection", juce::Mp3Connection (juce::Mp3Connection::input))
296   )
297   .withOggConnections (OggConnectionsProperties()
298       .withOggConnection ("Ogg Connection", juce::OggConnection (juce::OggConnection::input))
299   )
299 
```

Editing a source file in Xcode

The other folders in the Project Navigator are:

- **JUCE Modules and JUCE Library Code:** These contain the source code of JUCE itself, as this will need to be compiled along with your own code to build the plug-in. I usually collapse those groups as there is no reason to edit this code.
- **Resources:** This group contains several **.plist** files. These are so-called **property list** files that describe to macOS and iOS what sort of properties an app or plug-in has. You do not need to edit these files, Projucer fills them in using the settings from the **jucer** project.
- **Frameworks:** Lists all the system frameworks that the plug-in uses. These are low-level libraries provided by the operating system and used by JUCE.
- **Products:** This shows all the files Xcode will generate from your plug-in code.

Let's look at the Products group in more detail. It contains the following items: **Delay.vst3** is the VST3 version of the plug-in. **Delay.component** is the Audio Unit version. **Delay.appex** is part of the AUv3 format used by iOS (if you enabled this plug-in format in Projucer). **Delay.app** is the standalone version of the plug-in.



The Products group in the Project Navigator

There is also **libDelay.a**. This is a so-called static library. It contains the compiled plug-in code and is linked into the other products. **juce_vst3_helper** is a little helper tool that's used by JUCE to build the VST3 version of the plug-in. Note that these last two items are in red, which means these files do not exist — because you haven't compiled the project yet.

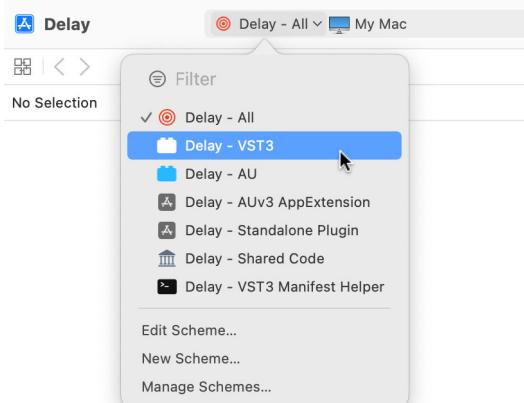
The only files shown in the Project Navigator that you need to concern yourself with are the C++ source code files in the **Delay/Source** group at the top. Everything else is stuff that Projucer created to make JUCE work seamlessly with Xcode. You should not change the contents of those other files and folders by hand, as it may break the build process.

Compilation targets

As mentioned, JUCE can build multiple plug-in formats from the same source code. In Xcode each of these things is called a compilation target, or just target.

The active target is shown in the toolbar at the top of the Xcode window. Initially the **Delay - All** target is selected, although it's possible that on your computer another target is selected by default.

Click the active target to bring up a pop-up that lists all available targets. What's shown here depends on the Plugin Formats you enabled in Projucer. The order of the items in this list is somewhat arbitrary, so the pop-up may look different on your computer.



The targets in the Xcode project

The targets in our plug-in project are:

- Delay - All
- Delay - VST3
- Delay - AU
- Delay - AUV3 AppExtension
- Delay - Standalone Plugin
- Delay - Shared Code
- Delay - VST3 Manifest Helper

The **VST3**, **AU**, and **AUV3 AppExtension** targets build the VST3, macOS Audio Unit, and iOS Audio Unit plug-ins, respectively.

The **Standalone Plugin** target builds a self-contained app that runs the plug-in without needing a separate host program.

The other targets are not something you need to worry about, but I'll explain what they are in case you're curious.

The **Shared Code** target builds the **libDelay.a** static library that contains all the plug-in code that is independent of any specific plug-in format. The other targets link to this static library, so that Xcode doesn't have to recompile the same code multiple times.

The **VST3 Manifest Helper** target builds the **juce_vst3_helper** tool that's used to generate the bundle for the VST3 plug-in.

Finally, the **All** target will build all the different plug-in formats in a single go.

Compiling the plug-in

To wrap up this chapter, let's build the VST3 version of the plug-in. First, use the pop-up to choose the **Delay - VST3** target.

The Xcode toolbar has a button that's shaped like a play button. This is known as the **run button**. Click this button to build the plug-in. You can press **Command+B** to do the same.

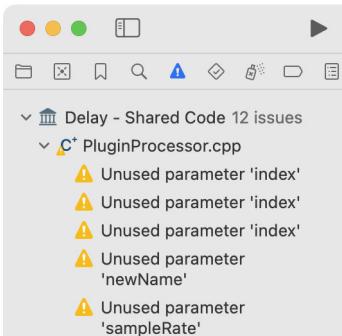
Xcode will start chugging away, compiling the four C++ source files as well as the JUCE modules. After a short while, the info bar at the top of the Xcode window should say: **Build Succeeded**.



Excellent! However, the yellow warning icon indicates there are some warnings about potential issues with the code. Uh oh, you didn't even write any code yet and already it has problems...

When you created the project with Projucer, it put enough code in the C++ source files to make a minimal plug-in, but with placeholders for much of the functionality. You still need to fill in these placeholders. Until you do, the compiler warns that parts of the code are incomplete. For the time being you can safely ignore these warnings.

The panel on the left side of the Xcode window has switched to the **Issue navigator**, showing the warnings in detail. You may see slightly different warnings on your computer, depending on the version of Xcode and JUCE you are using.



The warnings in the Issue Navigator

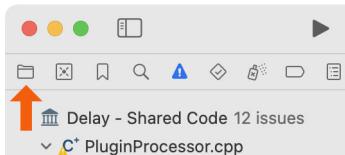
Tip: If the warning is in a file starting with “juce_”, such as `juce_String.cpp`, then it's something that happens in JUCE itself and you can ignore the warning.

Compiling will not always succeed. If there is an error in the source code, the info bar will say **Build Failed** instead of **Build Succeeded**.

Let's put a small error in the code to make the compiler fail on purpose. This is important because you might inadvertently make such mistakes in the coming chapters,

especially if you're new to C++, and I want to show what happens when the compiler isn't happy with the code you wrote.

Return to the **Project Navigator** by clicking the small folder icon on the left.



Click this icon to go back to the Project Navigator

In the Project Navigator, select **PluginProcessor.cpp** to open it. In the text editor, scroll down to where it says `void DelayAudioProcessor::processBlock`. This is where the plug-in does all the audio processing work.

At the end of this `processBlock` function there is a section of code that looks like this:

```
for (int channel = 0; channel < totalNumInputChannels; ++channel)
{
    auto* channelData = buffer.getWritePointer (channel);

    // ..do something to the data...
}
```

Remove that closing } brace and try to build the plug-in again (press the **run button** or **Command+B**). Xcode will say **Build Failed** at the top of the window and shows several error messages saying, “Function definition is not allowed here.”

```
Delay - VST3 > My Mac
Build Failed | Today at 12:30 +
```

```
C* PluginProcessor.cpp
```

```
Delay > Delay > Source > C* PluginProcessor.cpp > M DelayAudioProcessor::processBlock(buffer, midiMessages) < x >
```

```
132 void DelayAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer, juce::MidiBuffer& midi...
```

```
151 // Alternatively, you can process the samples with the channels
152 // interleaved by keeping the same state.
153 for (int channel = 0; channel < totalNumInputChannels; ++channel)
154 {
155     auto* channelData = buffer.getWritePointer (channel); ⚠ Unused variable 'channelData'
156
157     // ..do something to the data...
158 }
159
160 //=====
161 bool DelayAudioProcessor::hasEditor() const ✖ Function definition is not allowed here
162 { ✖ Function definition is not allowed here
163     return true; // (change this to false if you choose to not supply an editor)
164 }
```

Uh oh, the compiler found an error

Notice that these errors do not occur in the place where you removed the } but further below. That's because the compiler is confused and no longer understands the code.

Missing a closing brace or a semicolon is a common mistake made by new programmers. Unfortunately, the error message doesn't say, "You missed a closing brace here." If you get an unexpected error message after typing in some new code, make sure all the braces and semicolons are in place.

Put the } back where it was and build again. The error messages should disappear.

Note: With Xcode it can sometimes take a short while before the error messages disappear even if the status says Build Succeeded. This is a small annoyance of Xcode. As long as it says Build Succeeded you're good to go.

So, what's the difference between warnings and errors? First of all, the color: in Xcode warnings are yellow and errors are red.

An error is when the C++ compiler does not understand the code you've written, such as when a } is missing. It's like writing a sentence in English using the wrong spelling or grammar. The C++ compiler is very strict and can only work with sentences that are spelled 100% correctly and follow all the C++ grammar rules. English can be ambiguous, but computer languages must be exact.

Just because the code compiles without errors doesn't mean the program works correctly, by the way. You can still make the program do something that you didn't intend. The red error messages in Xcode only tell you something is wrong with your C++ syntax and that the compiler doesn't understand what you're trying to do.

A warning, on the other hand, is merely a hint that something might be amiss. A program full of warnings may run correctly. The warnings you're currently getting are about unused parameters and unused variables. This is the compiler trying to be helpful, saying, "Are you sure you didn't forget something here?"

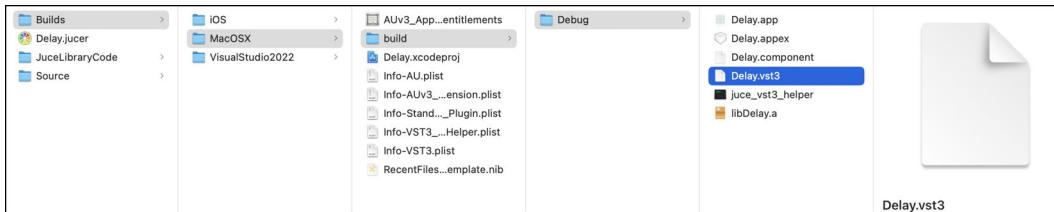
Compilers can give warnings for many possibly dangerous situations. We turned on most of the possible warnings in Projucer using the Add Recommended Compiler Warning Flags setting, because at this point in your programming career it's good to get these kinds of hints from the compiler.

Once you're more experienced, you can turn on only the warnings you're interested in and ignore the others. There are developers who tell the compiler to treat all warnings as fatal errors, but that's a little extreme for my taste.

Where is that plug-in?

In the next chapter we'll actually try to run the plug-in, but first I want to point out a couple of important things.

Go to the **Project Navigator** and expand the **Products** folder. Right-click on **Delay.vst3** and choose **Show in Finder** from the pop-up menu. This opens a new Finder window with the VST3 file that you just created.



The build folder in Finder

The **Builds** folder is where Xcode puts the plug-ins after successfully compiling them. The full path is: `Delay/Builds/MacOSX/build/Debug`.

Recall that in Projucer the exporters had a Debug and Release configuration. As `Delay.vst3` is inside a folder named `Debug`, this means it was built using the Debug configuration. That's what we want indeed, since we're still developing the plug-in. Once the plug-in is finished, you will build it in Release mode and it will end up in a separate Release folder.

Note: There is also a **Delay.component** file in the `build/Debug` folder. This is the Audio Unit version of the plug-in. If this shows up as a regular folder instead of a bundle, it means the actual binary for the AU plug-in hasn't been built yet, and Finder doesn't recognize this folder as being a bundle. Recall that you can look inside a bundle by right-clicking it and choosing **Show Package Contents**.

Host programs will look for the plug-in in a special folder on your computer. Assuming you've used plug-ins before, you may have installed them by hand to the system folder `/Library/Audio/Plug-Ins`.

The `Plug-Ins` folder has subfolders for the different formats: `VST3` and `Components` (for Audio Units). There may be a `VST` folder, which is for VST version 2, which is an obsolete format but some plug-ins are still available as VST2.

Hosts also look in the folder `/Users/yourname/Library/Audio/Plug-Ins`, allowing every user on the computer to install their own plug-ins.

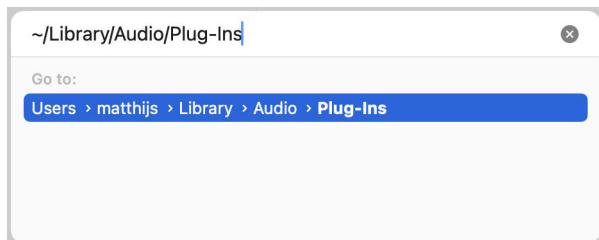
To make it possible for hosts to find the **Delay.vst3** plug-in, it must be copied to one of these folders.

In Projucer the **Enable Plugin Copy Step** option was set to Enabled. Because of this, after building the plug-in, Xcode copies the VST3 bundle from the `Builds/MacOSX/build/Debug` folder to `~/Library/Audio/Plug-Ins`.

This is not the system-level plug-ins folder, but the one for your user account. Notice the tilde ~ in front of the path, this refers to the folder `/Users/yourname`. Let's go to this folder in Finder.

Open a new Finder window. The Library folder is hidden by macOS so you won't be able to see it. From Finder's **Go** menu (in the menu bar at the top of the screen) choose the option **Go to Folder...** or press **Shift+Command+G**.

In the box that pops up, type `~/Library/Audio/Plug-Ins/` and press the **enter** key.



Navigating to the Library folder using Finder

Now if you go into the VST3 subfolder, you'll see a copy of **Delay.vst3** there. Whenever you do a new build of the project in Xcode, upon success it copies the **Delay.vst3** bundle into this folder, so that hosts can immediately pick up this new version.

Note: Most hosts will require closing and opening the application again before they can see new or updated plug-ins.

Intel versus Apple Silicon

There are two types of Mac computers with completely different CPUs: Intel and Apple Silicon (also referred to as ARM). Sometimes people build their plug-in for the wrong CPU architecture and then it won't work. Let's double check to make sure.

To verify that the binary inside the Delay.vst3 bundle supports the right architecture for your Mac, you need to use the Terminal. Go to your **Applications** folder, then to **Utilities**, and select **Terminal** to open it.

Type the following into the Terminal window (all on one line):

```
lipo -archs ~/Library/Audio/Plug-Ins/VST3/Delay.vst3/Contents/MacOS/Delay
```

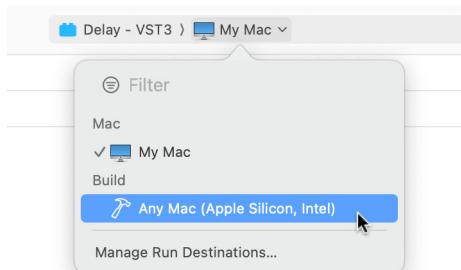
Press the **enter** key to execute this command. For me it prints:

```
x86_64
```

That means this plug-in will only run on an Intel Mac, or under Rosetta on an Apple Silicon Mac. If you're on an Apple Silicon Mac, it should print:

```
arm64
```

It is possible to build the plug-in so it will run on all types of Macs. In Xcode, click on the top bar where it says **My Mac** and change this to **Any Mac**:



Switching the run destination to Any Mac

Build the plug-in again (click the **run button** or press **Command+B**).

When Xcode is done building, go back to the Terminal and press the **arrow-up key** to repeat the `lipo` command and press **enter**. Now the output is:

```
x86_64 arm64
```

This means the plug-in supports both architectures. This works regardless of whether you are currently using an Intel or Apple Silicon machine.

The reason this isn't the default setting is that usually you'll be developing on just your own Mac and adding both CPU architectures requires compiling the plug-in twice, so it's slower than having one architecture. During development you'll want to keep the setting in Xcode to **My Mac**.

The reason I mention this here is that sometimes people are using an Apple Silicon Mac but a host that runs in Rosetta. That host won't be able to load the `arm64` version of the plug-in, even though your Mac has an ARM-based CPU. Because the host is running in Intel mode under the Rosetta emulator, the plug-in also needs to be built as `x86_64`. In that case you'll need to set Xcode to **Any Mac**.

Note: If you're new to the Terminal, you'll find that from time to time you may need it to run commands. The Terminal is a very powerful tool that is used a lot by software developers. It's worth learning more about it.

The next chapter is for installing Visual Studio on Windows. You may skip this unless you also have a Windows PC. In the chapter after that, you will learn how to load the plug-in into a host, and you'll make your first real changes to the source code.

5: Getting started with Visual Studio

This chapter is for Windows users. If you're on a Mac, skip to the next chapter.

Required versions

The screenshots in this chapter were made with Visual Studio 2022 on Windows 10, using JUCE 7.0.9. The plug-in was also tested with a preview release of JUCE 8.

By the time you're reading this book, Visual Studio and JUCE may have newer versions. I suggest always using the latest versions available. Just keep in mind that what you see on your screen might be a little different from the pictures in the book.

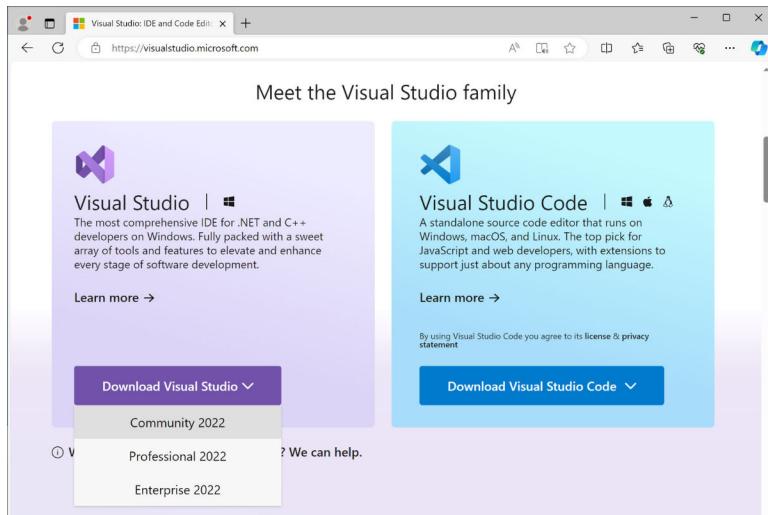
If you have an older PC you may not be able to install the latest versions of Windows and Visual Studio. That's OK too. JUCE 7 requires at least Visual Studio 2017 and Windows 8.1, while JUCE 8 requires Visual Studio 2019 and Windows 10. So as long as your computer can run those you should be good to go.



Installing Visual Studio

Visual Studio is the IDE for developers using a Windows PC. It can be downloaded for free from Microsoft.

Go to visualstudio.microsoft.com¹⁸ and scroll down to where it lets you choose between the different versions of Visual Studio. The website may ask you to sign up but you don't need to do this.



The Visual Studio website

Confusingly, there are two completely different applications named Visual Studio:

- Visual Studio — this is the one you want
- Visual Studio Code

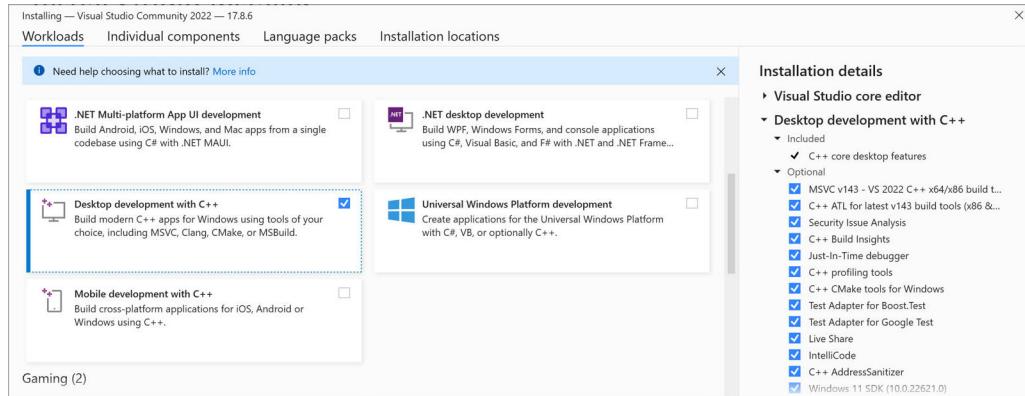
Besides the name they are otherwise completely unrelated. Visual Studio Code, also known as VSCode, is a source code editor that is popular with web developers but it's not a full IDE. You will want to download the regular Visual Studio.

When you hover over the **Download Visual Studio** button, it gives a choice of three different editions. The Professional and Enterprise editions are paid but the Community edition is free and has all the features you need. Click on **Community 2022** to start the download.

¹⁸<https://visualstudio.microsoft.com>

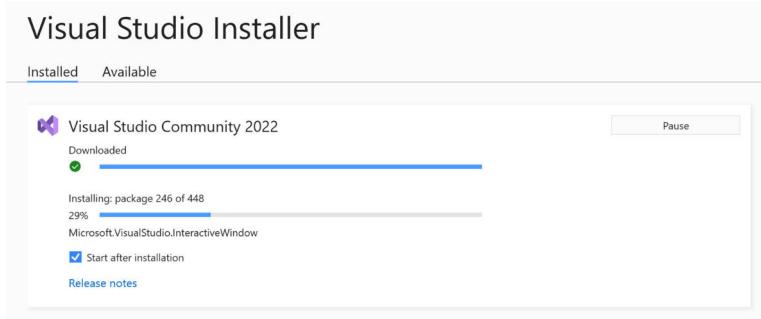
After a few moments you will have a file **VisualStudioSetup.exe** in your **Downloads** folder that is the installer for Visual Studio. Double-click to run this. Windows will ask whether you allow this app to make changes to your computer. Click **Yes**.

After downloading some additional components, the installer will ask what you will be using Visual Studio for. Scroll to the section **Desktop & Mobile** and select **Desktop development with C++**. You do not need to select any of the other items.



Choosing what to install

Click the **Install** button in the bottom right corner to start the installation process.



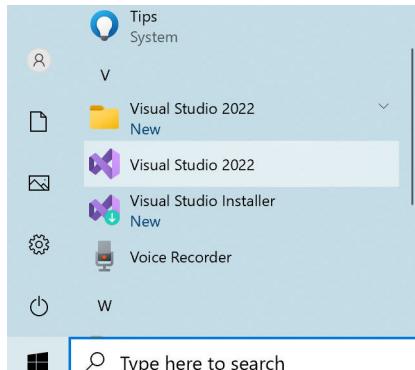
The installer in action

This can take a while as it will download several gigabytes of software packages and install them on your computer. This is probably a good time to make yourself a cup of coffee.

When the installer is done, restart your computer to finalize the installation process.

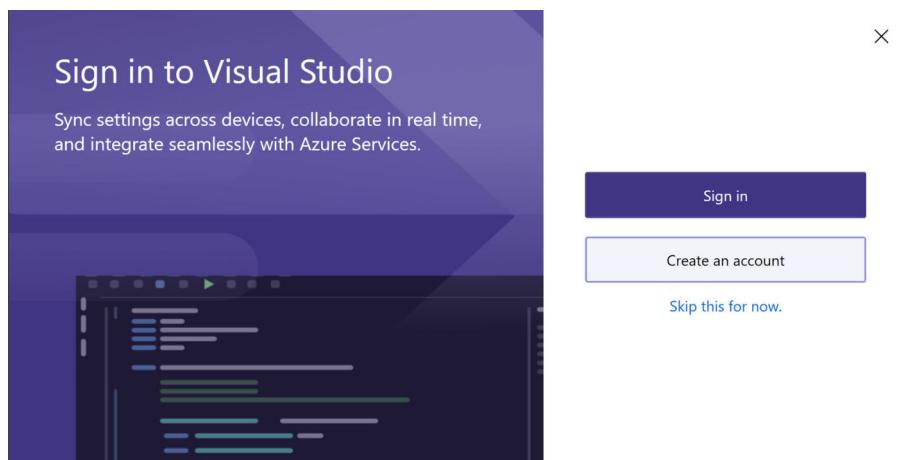
First time running Visual Studio

The Windows Start menu now has a **Visual Studio 2022** item in it. Click this to launch Visual Studio.



Visual Studio in the Start menu

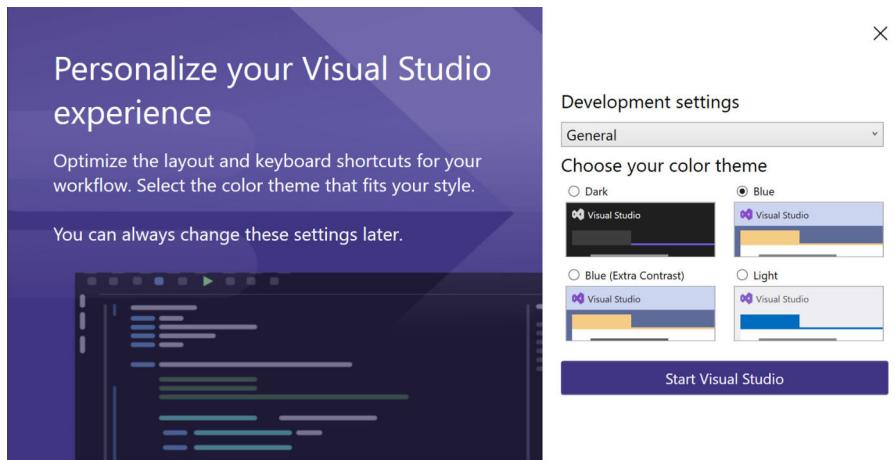
You will be asked to sign in to Visual Studio. If you already have a Microsoft account, you can use this to sign in. Otherwise, click **Skip this for now**.



Signing in to Visual Studio

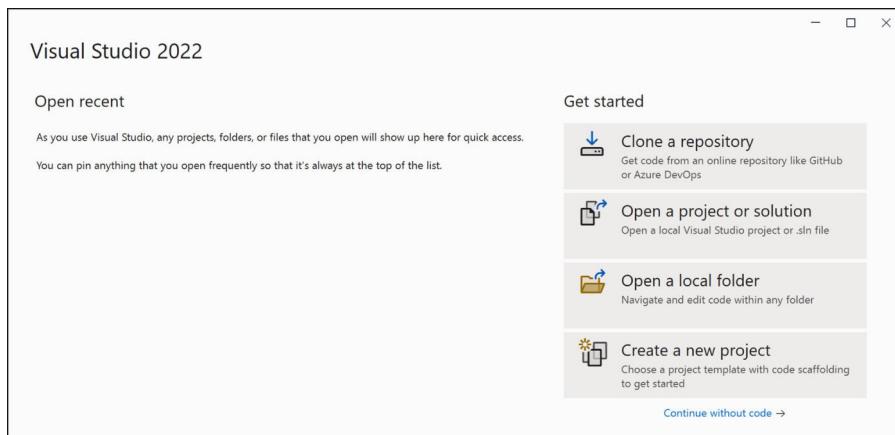
Note that every so often Visual Studio will prompt you to sign in again, but you can always skip this.

Next, it asks you to choose a color theme. I picked **Blue**. Leave the Development Settings option to General. Click **Start Visual Studio** to continue.



Personalizing Visual Studio

After a short while, the start screen will appear where you can create a new project or open an existing project:



The Visual Studio welcome screen

Excellent! That completes the installation of the IDE. You can close this screen to exit Visual Studio.

Exporting the project

In the last chapter, you used Projucer to create and configure the plug-in project. Now that Visual Studio is installed, you can export from Projucer to Visual Studio, so that you can finally write some code and compile the plug-in.

This is done using the exporter section at the top of Projucer:



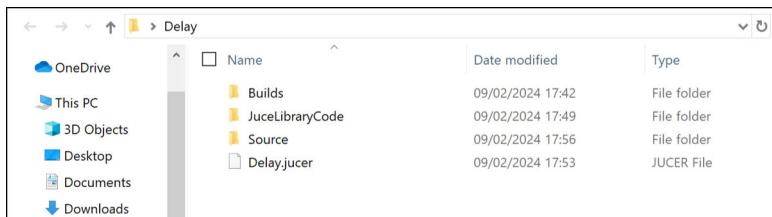
The selected exporter

If you have added more than one exporter to the project you can switch between them here. Make sure it's set to **Visual Studio 2022**.

Click the **circular button** to start Visual Studio and load the project. This is also known as the **Save and Open in IDE** button, or just the **export button**.

Note: The export button will be disabled when you select the Xcode exporter on a Windows computer. You'll have to open the **jucer** file in Projucer on a Mac for that.

Before we start using Visual Studio, let's see what the project looks like in the file system. I saved my project on the Desktop, and opening that folder shows the following:



The generated files on Windows

The Delay folder contains:

- **Delay.jucer**. This is the Projucer project. Whenever you need to make changes to the project, such as adding new source code files or changing any of the settings, this file is what you open. (If your computer does not show file extensions, note that **jucer** files have the same colorful icon as Projucer.)

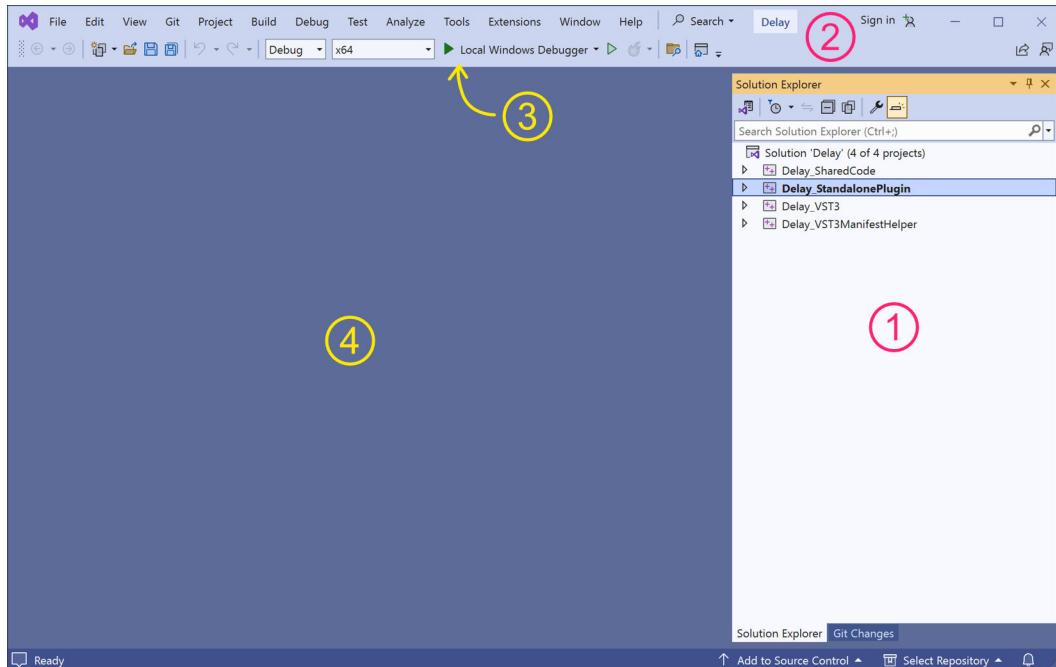
- **Builds.** This folder is where Projucer has saved the Visual Studio and Xcode projects. Take a look inside. There will be iOS, MacOSX, and VisualStudio2022 subfolders, depending on which exporters you added in Projucer. Inside the **VisualStudio2022** folder are the **Delay.sln** and **vcxproj** files. This is also where the compiled binaries go when you build the plug-in.
- **JuceLibraryCode.** This contains the JUCE modules that will be added to the project. You should not edit any of these files yourself, they will be overwritten by Projucer the next time you export.
- **Source.** These are the C++ source code files for your plug-in. It contains the four source files you already saw in the File Explorer in Projucer. When you add new files to the project in Projucer, they go into this folder.

Tip: You can safely delete the Builds and JuceLibraryCode folders at any time. Sometimes removing these folders is even necessary, for example if you get weird compilation errors after updating to a new version of JUCE. After removing these folders, simply open the **jucer** file in Projucer again and do a new export. This will reconstruct the Builds and JuceLibraryCode folders.

Note: If you are familiar with version control tools such as git, you typically would add the Builds and JuceLibraryCode folders to your **.gitignore** file so that they are not included in the repository.

A quick tour of Visual Studio

After exporting the project from Projucer, it will open in Visual Studio. That should look something like the following.

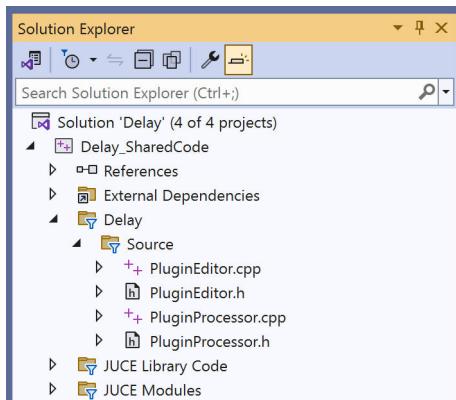


The new project in Visual Studio

The Visual Studio window is divided into different sections:

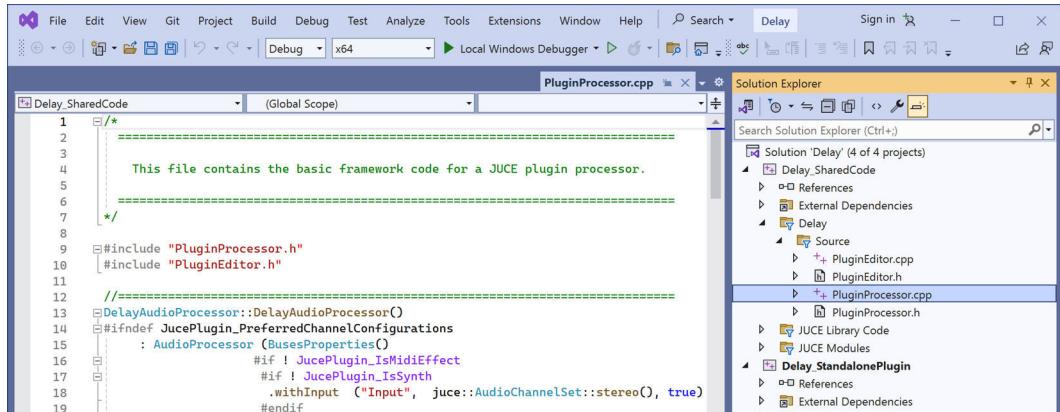
1. The Solution Explorer shows the files that make up the project.
2. The toolbar contains the main functions of the IDE.
3. The “play” button is used to build the project.
4. The editing area. This is where you will edit the source code files.

In the **Solution Explorer** on the right, expand the **Delay_SharedCode** item, then expand the **Delay** folder and then **Source**. This reveals the four C++ source code files that were also visible in Projucer’s File Explorer.



The source files in the Solution Explorer

Clicking a file will select it and opens the code in the text editor.



Editing a source file in Visual Studio

The other folders in the **Delay_SharedCode** group are:

- **JUCE Library Code** and **JUCE Modules**: These contain the source code of JUCE itself, as this will need to be compiled along with your own code to build the plug-in.
- **External Dependencies**: This folder contains the source code from the system frameworks that the plug-in uses. These are low-level libraries provided by the operating system and used by JUCE.
- **References**: Not used by our project.

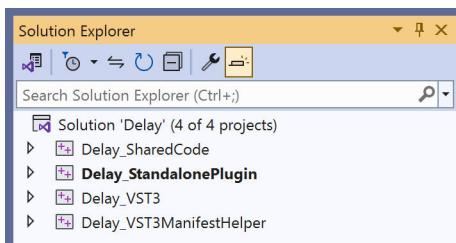
The only files shown in the Solution Explorer that you need to concern yourself with are the C++ source code files in the **Delay/Source** group under **Delay_SharedCode**.

Everything else is stuff that Projucer created to make JUCE work seamlessly with Visual Studio. You should not change the contents of those other files and folders by hand, as it may break the build process.

Compilation targets

As mentioned, JUCE can build multiple plug-in formats from the same source code. In Visual Studio each of these things is called a project, but I will use the more common term compilation target, or just target.

The active target is shown in the Solution Explorer in bold. Initially the **Delay_StandalonePlugin** target is selected, although it's possible that on your computer another target is selected by default.



The targets in Visual Studio

The targets in our plug-in project are:

- Delay_SharedCode
- Delay_StandalonePlugin
- Delay_VST3
- Delay_VST3ManifestHelper

The **Delay_VST3** target builds the VST3 plug-in. There is no target for the Audio Unit plug-in in Visual Studio, as that can only be built on a Mac using Xcode.

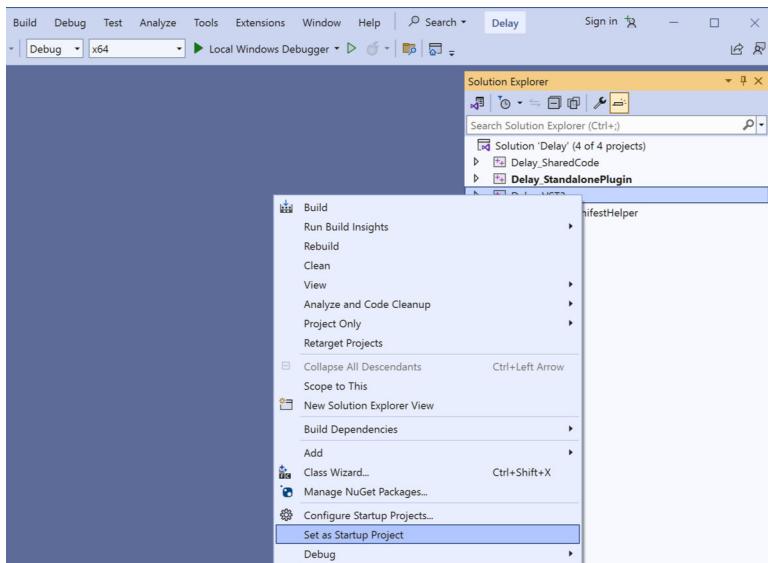
The **Delay_StandalonePlugin** target builds a self-contained app that runs the plug-in without needing a separate host program.

The other targets are not something you need to worry about, but I'll explain what they are in case you're curious.

The **Delay_SharedCode** target builds a so-called static library (named **Delay.lib**) that contains all the plug-in code that is independent of any specific plug-in format. The other targets link to this static library, so that Visual Studio doesn't have to recompile the same code multiple times.

The **Delay_VST3ManifestHelper** target builds a tool named **juce_vst3_helper.exe** that's used to generate the bundle for the VST3 plug-in.

To change the active target, right-click it and choose **Set as Startup Project** from the pop-up menu. The new target will now appear bolded in the Solution Explorer.



Changing the active target in Visual Studio

Compiling the plug-in

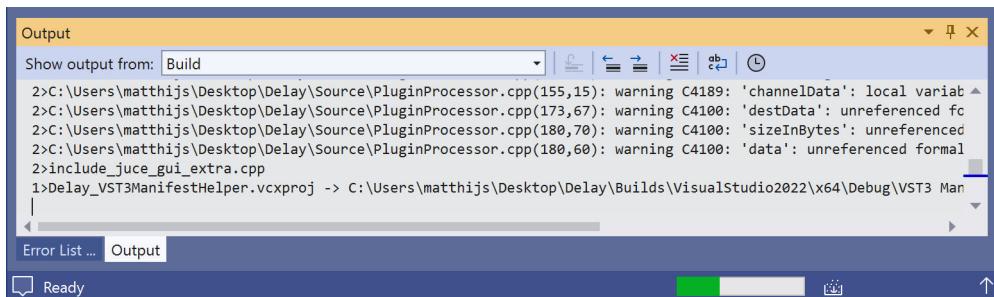
To wrap up this chapter, let's build the VST3 version of the plug-in.

First, set **Delay_VST3** as the active target using **Set as Startup Project**.

Right-click on **Delay_VST3** again and choose **Build** from the pop-up menu, or press **Ctrl+B** to build the plug-in.

Note: Normally to build a project in Visual Studio, you would click the “play” button in the toolbar or press F5. However, if you try that at this point you will get an “Unable to start program” error. That’s because a plug-in needs to run inside a host program. We will fix this in the next chapter.

Visual Studio will start chugging away, compiling the four C++ source files as well as the JUCE modules. You can follow along with the progress in the **Output** pane that opens:



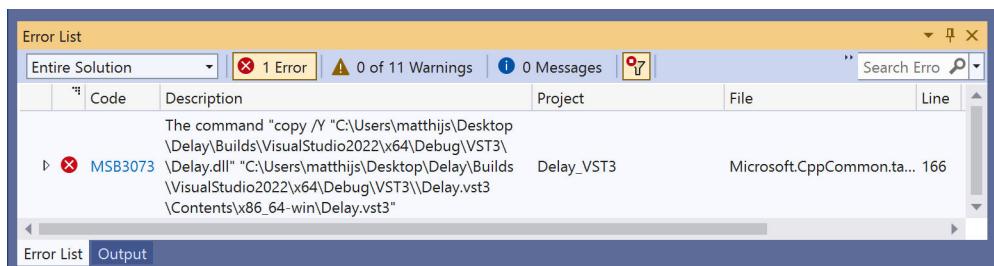
The Output pane shows the build progress

If all went well, after a short while, the output pane should say:

===== Build: 3 succeeded, 0 failed, 0 up-to-date, 0 skipped =====

It says “3 succeeded” because to build the plug-in, Visual Studio had to compile three targets: Delay_SharedCode, Delay_VST3, and Delay_VST3ManifestHelper. The important thing is that it says **0 failed**, meaning there were no errors.

However, you may have gotten an error result at this point:

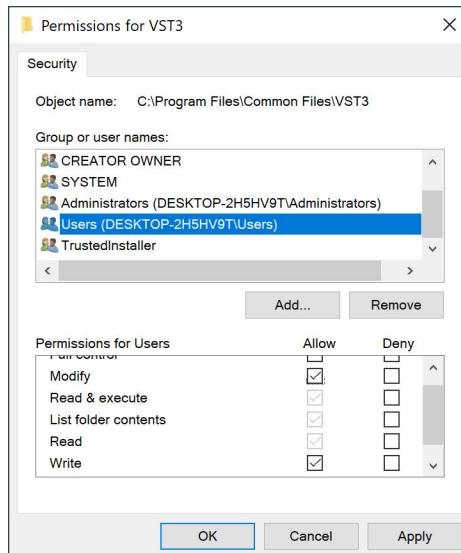


Building failed with an error

This error happens because you set the option **Enable Plugin Copy Step** in Projucer to Enabled back when you made the project. Visual Studio has successfully built the plug-in but it was not able to copy **Delay.vst3** into the `C:\Program Files\Common Files\VST3` system folder. On most Windows machines this folder is protected so that only users with Administrator rights may write into it.

To fix this, you'll have to change the rights on this folder. You will need to have Administrator rights on your computer to do this. Use the Windows file explorer to go to `C:\Program Files\Common Files`. Then right-click the **VST3** folder and from the pop-up menu choose **Properties**. This brings up a dialog box. Go to the **Security** tab and click the **Edit** button to bring up a second dialog box.

In the list of usernames, select the entry that starts with **Users**. Then in the list below, check the boxes in the **Allow** column for **Modify** and **Write**. To finish, click **OK** on both dialogs. Now try building the plug-in again in Visual Studio.



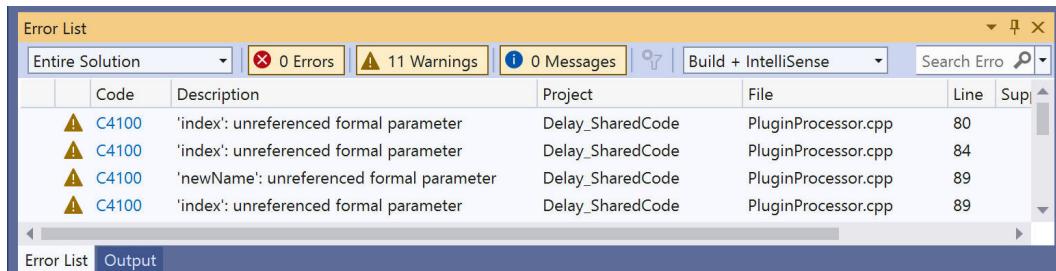
Changing permissions on the VST3 system folder

If you've never installed any plug-ins on this computer before, the **VST3** folder may not even exist. In that case, click the **New folder** button inside `C:\Program Files\Common Files`. You need to confirm this operation, so click **Continue**. Type in the new folder name, **VST3**. Then change the permissions on this new folder using the above procedure.

Note: If you are unable (or unwilling) to change the permissions on the VST3 folder, go to Projucer and disable the Enable Plugin Copy Step (under Debug for the Visual Studio exporter) and save the project again. You will need to manually copy Delay.vst3 into C:\Program Files\Common Files\VST3 after every build, or hosts won't be able to find the plug-in. You can also run Visual Studio as administrator.

Warnings and errors

The build may have been successful, but the **Error List** tab under the output pane shows there are some warnings about potential issues with the code. Uh oh, you didn't even write any code yet and already it has problems...



The warnings in the Error List

You may see slightly different warnings on your computer, depending on the version of Visual Studio and JUCE you are using. Tip: If the warning is in a file starting with `juce_`, such as `juce_String.cpp`, then it's in JUCE itself and you can ignore the warning.

When you created the project with Projucer, it put enough code in the C++ source files to make a minimal plug-in, but with placeholders for much of the functionality. You still need to fill in these placeholders. Until you do, the compiler warns that parts of the code are incomplete. For the time being you can safely ignore these warnings.

Compiling will not always succeed. If there is an error in the source code, the Output pane will say something like the following and the **Error List** will display the error messages.

```
===== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped =====
```

Let's put a small error in the code to make the compiler fail on purpose. This is important because you might inadvertently make such mistakes in the coming chapters, especially if you're new to C++, and I want to show what happens when the compiler isn't happy with the code you wrote.

In the **Solution Explorer**, click on **PluginProcessor.cpp** to open it. In case you forgot where this is located, navigate to **Delay_SharedCode**, then **Delay** and **Source** under that.

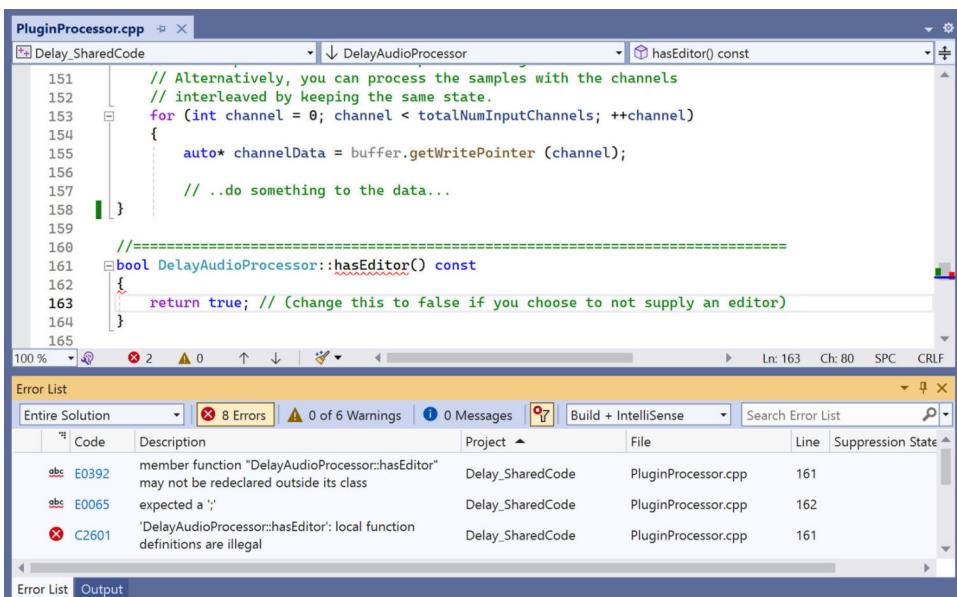
In the text editor, scroll down to where it says `DelayAudioProcessor::processBlock`. This is where the plug-in does all the audio processing work.

At the end of this `processBlock` function there is a section of code that looks like this:

```
for (int channel = 0; channel < totalNumInputChannels; ++channel)
{
    auto* channelData = buffer.getWritePointer (channel);

    // ..do something to the data...
}
```

Remove that closing `}` brace and try to build the plug-in again (press **Ctrl+B**). Now the build will fail and the **Error List** shows several error messages.



Uh oh, the compiler found an error

Notice that these errors do not occur in the place where you removed the } but further below. Visual Studio puts a red squiggly line where it is confused and no longer understands the code you wrote. It thinks something is wrong with `hasEditor`.

Missing a closing brace or a semicolon is a common mistake made by new programmers. Unfortunately, the error message doesn't say, "You missed a closing brace here." So, if you get an unexpected error message after typing in some new code, make sure all the braces and semicolons are in place.

Put the } back where it was and build again. The error messages should disappear.

So, what's the difference between warnings and errors?

An error is when the C++ compiler does not understand the code you've written, such as when a } is missing. It's like writing a sentence in English using the wrong spelling or grammar. The C++ compiler is very strict and can only work with sentences that are spelled 100% correctly and follow all the C++ grammar rules. English can be ambiguous, but computer languages must be exact.

Just because the code compiles without errors doesn't mean the program works correctly, by the way. You can still make the program do something that you didn't intend. The error messages in Visual Studio only tell you something is wrong with your C++ syntax and that the compiler doesn't understand what you're trying to do.

A warning, on the other hand, is merely a hint that something might be amiss. A program full of warnings may run correctly. The warnings you're currently getting are about unused parameters and unused variables. This is the compiler trying to be helpful, saying, "Are you sure you didn't forget something here?"

Compilers can give warnings for many possibly dangerous situations. You turned on most of the possible warnings in Projucer using the Warning Level setting, because at this point in your programming career it's good to get these kinds of hints from the compiler.

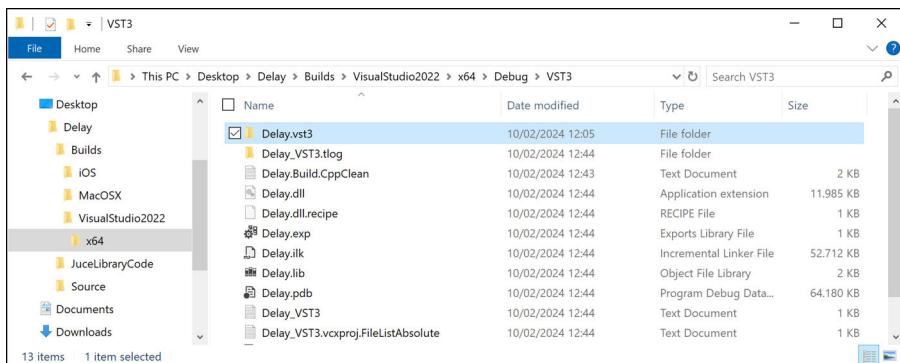
Once you're more experienced, you can turn on only the warnings you're interested in and ignore the others. There are also developers who tell the compiler to treat all warnings as fatal errors, but that's a little extreme for my taste.

Where is that plug-in?

In the next chapter we'll actually try to run the plug-in, but first I want to point out where you can find the compiled files.

Go to the **Solution Explorer** and right-click on **Delay_VST3**, then from the pop-up menu choose the open **Open Folder in File Explorer** (near the bottom of the menu). This opens a new file explorer window for the **Builds** folder, which is where Visual Studio puts the plug-in after successfully compiling it.

To find the VST3 file that you just created, you have to drill down into the **x64\Debug\VST3** subfolders.



The build folder

The full path to Delay.vst3 is: `Delay\Builds\VisualStudio2022\x64\Debug\VST3`

Recall that in Projucer the exporters had a Debug and Release configuration. As Delay.vst3 is inside a folder named Debug, this means it was built using the Debug configuration. That's what we want indeed, since we're still developing the plug-in. Once the plug-in is finished, you will build it in Release mode and it will end up in a separate Release folder.

Host programs will look for the plug-in in a special folder on your computer. For VST3 plug-ins, this folder is `C:\Program Files\Common Files\VST3`.

Because in Projucer the Enable Plugin Copy Step option was set to Enabled, Visual Studio automatically copied the VST3 bundle into this folder so that hosts can immediately pick up this new version. If you use the Windows file explorer to navigate to that folder you should see a copy of Delay.vst3 in there.

If you disabled the Enable Plugin Copy Step option, you will have to manually copy Delay.vst3 from the Delay\Builds\VisualStudio2022\x64\Debug\VST3 folder to C:\Program Files\Common Files\VST3.

Note that **Delay.vst3** is itself just another folder, sometimes referred to as the “bundle”. This folder has a **Contents\x86_64-win** subfolder that contains the shared library (DLL) for the binary, which is also named Delay.vst3. You’re supposed to copy the entire Delay.vst3 folder into C:\Program Files\Common Files\VST3, not only the DLL!

Note: Most hosts will require closing and opening the application again before they can see new or updated plug-ins.

In the next chapter you will learn how to load the plug-in into a host, and you’ll make your first real changes to the source code.

6: Using a host for testing

In the previous chapter you've used the Xcode or Visual Studio compiler to generate a VST3 bundle from the plug-in's source code, but haven't actually tried running the plug-in yet. In this chapter you'll load the plug-in in a host program and make your first real changes to the code.

Using a DAW

Both Xcode and Visual Studio have a triangular button in their toolbar that resembles the play button from your DAW's transport controls or from an old tape deck or CD player.



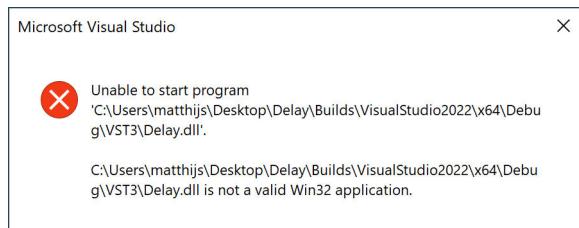
The run buttons in Xcode (left) and Visual Studio right

This button is commonly referred to as the “Start” or “Run” button. It first compiles the program and then runs it.

Visual Studio has two of these buttons, one to run the program inside the debugger (the dark green button) and one to run without the debugger (the light green button).

However, pressing the run button in Xcode only builds the plug-in, it doesn't actually run it. In Visual Studio, pressing this button even gives an error message.

The reason you can't directly run a plug-in is that it needs to run inside a host program such as your DAW.

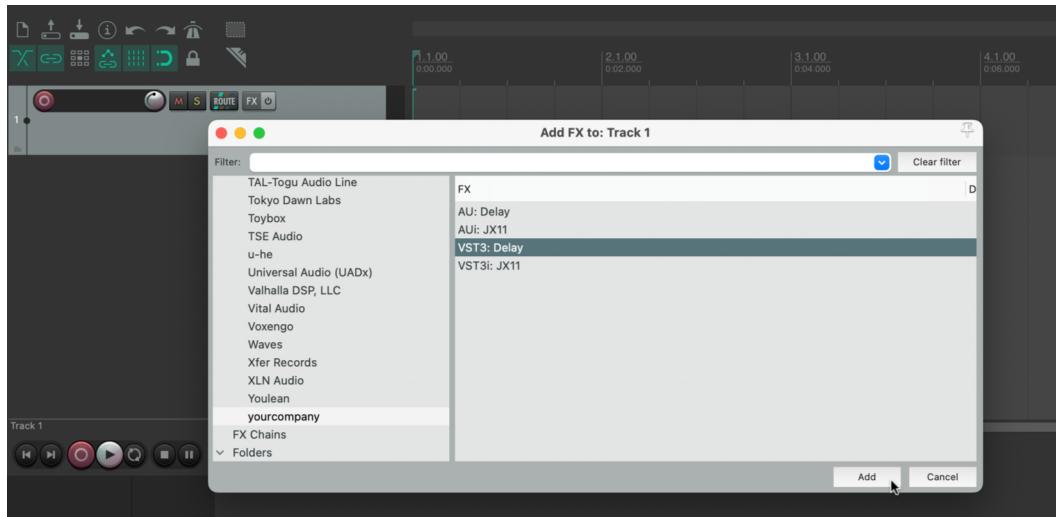


Trying to run the plug-in gives an error with Visual Studio

Launch your favorite DAW and try to load the plug-in. The plug-in can be inserted as an effect on either a mono or stereo track.

If you do not have a DAW installed on your computer, I suggest downloading [REAPER¹⁹](#), which is free to try. In this book we will mostly use a small mini DAW named AudioPluginHost that comes with JUCE, but it's also a good idea to test the plug-in with a real DAW.

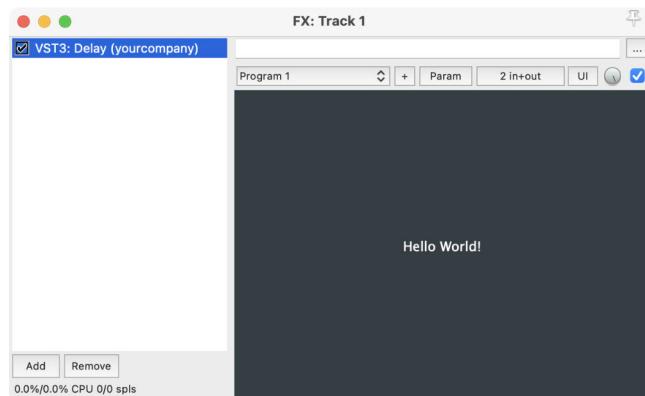
After launching the DAW, create a new audio track, and bring up the effects picker for that track. In REAPER that looks like this:



Adding the plug-in to a track in REAPER

Navigate to **yourcompany**, which is the manufacturer name set in Projucer, and then select **Delay**. After loading the plug-in, its editor appears in the DAW.

¹⁹<https://www.reaper.fm>



The plug-in running in REAPER

Congrats! You've managed to build and run your first plug-in. It doesn't really do anything yet but this is an important first step.

The default plug-in source code generated by Projucer just displays the text *Hello World!* and does nothing else. Any audio is passed through the plug-in unchanged. Try it out, put an audio clip on the track and it should play as normal. If you toggle the host's bypass button, there should be absolutely no change in sound.

Note: If the DAW cannot find the plug-in, make sure **Delay.vst3** is present in the system's plug-in folder. On Windows it is C:\Program Files\Common Files\VST3. If Delay.vst3 is not there, verify that the Enable Plugin Copy Step option in Projucer is turned on (under the Visual Studio exporter, Debug configuration). Or manually copy Delay.vst3 from Delay\Builds\VisualStudio2022\x64\Debug\VST3 into the system's plug-ins folder. Then restart the DAW.

Logic Pro or GarageBand users

Readers who are using Logic Pro or GarageBand on Mac will need to build the Audio Units version of the plug-in as these hosts cannot load VST3 files.

In the Xcode toolbar, activate the **Delay - AU** target and press the **run button** to build the Audio Unit version of the plug-in. Make sure to restart the DAW if it was already open.

If the Delay plug-in does not appear in the list of available plug-ins inside Logic Pro, first check the Plug-In Manager window to see if the plug-in exists under the manufacturer **yourcompany**. If not, press the Full Audio Unit Reset button and restart Logic Pro.

If the plug-in still does not show up, you may have run into an annoying macOS bug that prevents the Mac’s audio subsystem from detecting new plug-ins. One workaround that usually fixes this is to open a **Terminal** (from Applications/Utilities) and execute the following command:

```
killall -9 AudioComponentRegistrar
```

Now restart Logic Pro or GarageBand and it should see the plug-in. If all else fails, restarting your Mac should do the trick.

Making your first changes to the code

Let’s finally get our hands dirty and write some code. You’re going to change the text that’s displayed on the plug-in’s user interface.

The user interface — also known as the GUI (pronounced “gooey”) or just UI (“you-eye”) — lives in the source code file **PluginEditor.cpp**.

In your IDE, open **PluginEditor.cpp** and find the `paint` function. It looks like this:

```
void DelayAudioProcessorEditor::paint (juce::Graphics& g)
{
    // (Our component is opaque, so we must completely fill the background
    // with a solid colour)
    g.fillAll (
        getLookAndFeel().findColour (juce::ResizableWindow::backgroundColourId));

    g.setColour (juce::Colours::white);
    g.setFont (15.0f);
    g.drawFittedText (
        "Hello World!", getLocalBounds(), juce::Justification::centred, 1);
}
```

This is the part of the source code that is responsible for drawing the user interface on the screen. Even if you’re new to C++ you can probably sort of guess what’s going on here, but I will explain exactly what this code does in the following sections.

Note: The source code in the book will occasionally look a bit different from what you're seeing in the IDE. The lines from the paint function were too wide to fit in the book, so I had to wrap them. Fortunately, in C++ spaces have no meaning.

For example, the following line of code,

```
g.setColour (juce::Colours::white);
```

will do the same as,

```
g.setColour(      juce::Colours::white  
);
```

and is also the same as:

```
g.setColour (juce::Colours::white );
```

Let's make the drawing code say something other than *Hello World!*. Using the IDE's text editor, change the code of the paint function to the following:

```
void DelayAudioProcessorEditor::paint (juce::Graphics& g)  
{  
    g.fillAll (juce::Colours::blue);           // 1  
    g.setColour (juce::Colours::white);  
    g.setFont (40.0f);                      // 2  
    g.drawFittedText ("My First Plug-in!",   // 3  
                      getLocalBounds(), juce::Justification::centred, 1);  
}
```

I've used the // 1, // 2 and // 3 markers to indicate the places in the source code that have changed. I will be doing that throughout the book. You don't have to type in these markers, although it's fine if you do.

These are the changes:

1. `fillAll` now fills the entire editor background with the color blue.
2. The size of the font is 40 points.
3. The text is "My First Plug-in!". Note that the text must be placed inside double quote " characters.

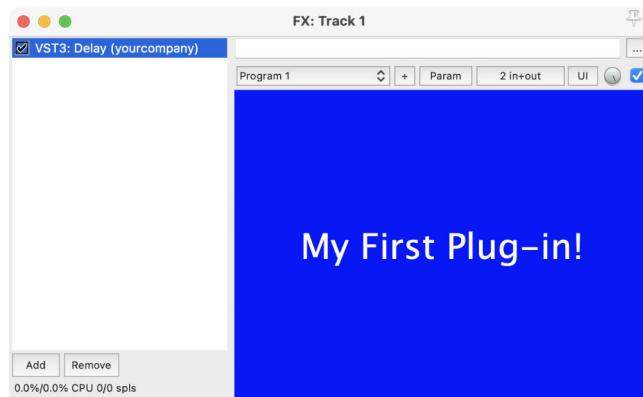
When making these changes, I left out the first line, which was:

```
// (Our component is opaque, so we must completely fill the background with ...)
```

In C++, everything that follows the // symbol is ignored by the compiler, until the end of that line. These are called **comments** and are not intended for the compiler but for the human reader. Comments are useful because they can serve as helpful reminders of what the code is actually doing — and more importantly why. The comment here says that we need to fill the editor's background with a solid color.

Build the plug-in again. In Xcode you can click the **run button** or press **Command+B**. In Visual Studio, right-click **Delay_VST3** in the Solution Explorer and choose **Build** from the pop-up menu, or press **Ctrl+B**.

If your DAW is still open you'll need to close it first. Most DAWs will not “hot reload” plug-ins. Re-open your DAW and instantiate the plug-in on a new track. The editor should look like this:



The modified plug-in editor in REAPER

Excellent! You've made your first plug-in editor. Feel free to mess around with this code some more before we continue. Maybe try some different colors.

Note: If Visual Studio gave the error message that it can't copy the VST3 into the C:\Program Files\Common Files\VST3 folder, then check that your DAW isn't currently open and using the plug-in. Close the DAW and perform the build again. Also make sure the permissions on that folder are correct (see the previous chapter).

Getting to know C++

If you are completely new to software development, I can imagine that the code we just looked at may seem an odd mix of weird symbols such as `::` and `g.` and `&`, and short phrases made up of English words that might almost make sense, such as `paint`, `fillAll` and `drawFittedText`.

In this section we'll take a look at the basics of C++ and learn to “decipher” this source code. Experienced programmers, feel free to skip ahead.

Programs written in C++ consists of **objects** and **functions**. An object is some kind of “thing” that lives inside the computer’s memory (or RAM), while a function performs an action or computation.

- `DelayAudioProcessorEditor` is an object. It represents the plug-in’s user interface. When loading the plug-in, the DAW places the `DelayAudioProcessorEditor` object into a window and shows it to the user.
- `paint` is a function. It draws the contents of the editor on the screen, and therefore is part of the `DelayAudioProcessorEditor` object.

Functions in C++ look like this:

```

3           1           2
void DelayAudioProcessorEditor::paint (juce::Graphics& g)
{
    4
}

```

I’ve marked the different parts with numbers:

1. The **name** of the function and the object it belongs to. The `::` symbol (made of two colon characters) is used to indicate that one thing, here the function `paint`, belongs to another thing, the `DelayAudioProcessorEditor` object.
2. Between the parentheses (and) are the **arguments** to the function. `paint` has one argument, an object of type `juce::Graphics` that it can use to draw text and other graphics on the screen. The `&` symbol is something you can ignore for now.

3. The `void` at the front is the function's **return type**. Using a return type of `void` means that this function does not return a result. Later on, you'll see examples of functions that do return something.
4. The **body** of the function is everything in between the `{` and `}` braces and consists of one or more lines of code. The `{` and `}` braces are used all over the place in C++ to demarcate the beginning and end of something, in this case the code that constitutes the `paint` function.

This is the code that is currently inside `paint`:

```
g.fillAll (juce::Colours::blue);
g.setColour (juce::Colours::white);
g.setFont (40.0f);
g.drawFittedText ("My First Plug-in!",
    getLocalBounds(), juce::Justification::centred, 1);
```

When the DAW decides to redraw the plug-in's editor, it will call the `paint` function. This executes the code in the function from top to bottom, line-by-line:

1. First, fill the background with the color blue.
2. Then, change the drawing color to white.
3. Then, set the font size to 40 points.
4. Finally, draw the text *My First Plug-in!* in the center of the window.

It's important to understand that the order of these statements matters. For example, if you were to change the code inside `paint` to the following,

```
g.setColour (juce::Colours::white);
g.setFont (40.0f);
g.drawFittedText ("My First Plug-in!",
    getLocalBounds(), juce::Justification::centred, 1);
g.fillAll (juce::Colours::blue);
```

then the text would not be visible. Even though the text is drawn as before, it immediately gets painted over by the `fillAll` command, which erases everything in the editor window.

Let's look at these lines of code in more detail, starting with `fillAll`.

```
g.fillAll (juce::Colours::blue);
```

Recall that the `paint` function has the argument `juce::Graphics& g`. This means that when the host tells the plug-in's editor to paint itself, the host will also provide such a `juce::Graphics` object.

The `juce::Graphics` object knows how to draw things such as lines, text, and images into the host's window. Inside the `paint` function, we'll refer to this object using the very short name `g`.

`fillAll` is a function that belongs to the `juce::Graphics` object. By writing `g.fillAll(juce::Colours::blue)` we say, "Hey `juce::Graphics` object, I want you to fill the entire background with the color blue."

`juce::Colours::blue` is then the argument that is passed to the `g.fillAll` function. To use a different color, write `juce::Colours::orange` or any other color name that JUCE knows about. You can see the full list of colors [here](#)²⁰.

All the objects provided by JUCE start with "juce::". This is called the **namespace**. The JUCE objects you've seen here are `juce::Graphics`, `juce::Colours`, and `juce::Justification`. There are many more that we'll use in this book. Again, the double colon `::` is used to refer to names of things inside other things. Another namespace you'll be using is `std::`, which contains the C++ standard library.

A line of code such as `g.fillAll (juce::Colours::blue);` is known as a **statement**. It tells C++ to go and do something: "Paint the background in blue." Statements must always end with a `;` semicolon.

If you forget, the compiler will usually be able to figure out that the semicolon is missing and provide a helpful error message. But sometimes the compiler gets confused and the error message may point to some other part of the code, not the line that is wrong.

In the last chapter you've seen what happens if you forget a `}` brace. The same thing applies to semicolons: When you get a compiler error, first check that all semicolons are there.

The other statements in the `paint` function also call functions of the `juce::Graphics` object. The second statement was the following:

```
g.setColour (juce::Colours::white);
```

²⁰<https://docs.juce.com/master/namespaceColours.html>

This sets the drawing color to white using the `setColour` function. Note that JUCE uses British English to spell its names. This is different from most other software libraries, which use American English. It's a little weird if you're not from the UK, but it's important to spell the function and object names exactly the way JUCE does it. Writing `g.setColor(juce::Colors::white)` will result in a compiler error.

Note that doing `g.setColour` does not draw anything by itself. It simply changes the color of the pen used by subsequent drawing operations.

```
g.setFont (40.0f);
```

Next, we set the size of the font to 40 points. The font is the typeface that's used for drawing text. The value passed into the `setFont` function here is written as `40.0f`. The `f` at the end is important: it tells C++ that we want this to be a **floating-point number** or `float` in C++ speak. A `float` is a number with a decimal point. For example, you could have written `40.5f` to get a font size of 40.5 points.

Finally, we tell the `Graphics` object to draw some text using `drawFittedText`:

```
g.drawFittedText ("My First Plug-in!",
    getLocalBounds(), juce::Justification::centred, 1);
```

This function will draw using the current color (white) and font size (40 points), but also takes several arguments:

1. The first argument is a **string** that contains the text to display, "My First Plug-in!". Values in C++ have a so-called **data type**: a floating-point number is for numbers with a decimal point, an integer is for whole numbers, and a string is for a piece of text. A string is always between a pair of straight double quotes ". Don't forget the closing quotes or you'll get a compiler error.
2. The second argument is the area (a rectangle) into which to fit the text. To get the size of the entire editor window, we call the function `getLocalBounds()`. This is a function that belongs to the `DelayAudioProcessorEditor` object, not to the `juce::Graphics` object, and so it does not have `g.` in front.
3. The third argument determines how the text is positioned inside the provided area. Here we use `juce::Justification::centred`, which centers the text horizontally and vertically. There are other options you can play with there. For example, replace `centred` with `top` or `bottomRight`.

4. The fourth argument is the maximum number of lines to use, here 1. This time we didn't write `1.0f` for the number, since this argument is supposed to be a whole number, it has no decimal point. In other words, it's an integer or `int` in C++ terminology.

Some functions have no arguments, such as `getLocalBounds`. Often functions have a single argument, such as `g.setColour` and `g.setFont`, and `paint` itself. But many functions have multiple arguments, such as `g.drawFittedText`.

You always need to provide the correct number of arguments, in the correct order, and of the correct data type, otherwise the C++ compiler will give a warning or even an error message.

To know what arguments a function needs, or even what functions a certain object has, you can refer to the JUCE documentation. For example, here are the [docs](#)²¹ for `drawFittedText` from `juce::Graphics`.

◆ `drawFittedText()` {2/2}

```
void Graphics::drawFittedText (const String & text,
                             Rectangle< int > area,
                             Justification justificationFlags,
                             int int maximumNumberOfLines,
                             float float minimumHorizontalScale = 0.0f) const
```

Tries to draw a text string inside a given space.

This does its best to make the given text readable within the specified rectangle, so it's useful for labelling things.

If the text is too big, it'll be squashed horizontally or broken over multiple lines if the `maximumLinesToUse` value allows this. If the text just won't fit into the space, it'll cram as much as possible in there, and put some ellipsis at the end to show that it's been truncated.

A `Justification` parameter lets you specify how the text is laid out within the rectangle, both horizontally and vertically.

The `minimumHorizontalScale` parameter specifies how much the text can be squashed horizontally to try to squeeze it into the space. If you don't want any horizontal scaling to occur, you can set this value to `1.0f`. Pass `0` if you want it to use a default value.

See also

- [GlyphArrangement::addFittedText](#)

The JUCE documentation for `juce::Graphics::drawFittedText`

According to the documentation, this function actually takes five arguments, not four! That's because the final argument is optional. If you don't provide it, the value `0.0f` is assumed (another floating-point value).

Hopefully the code in the `paint` function has started to make some sense now. If not, no worries, we'll revisit these concepts several more times.

²¹<https://docs.juce.com/master/classGraphics.html>

Note: It's common for people to use the term "parameters" when talking about function arguments. The JUCE documentation also does this. In this book, however, I will call them arguments and use the term parameters only for the plug-in parameters. Function arguments/parameters and plug-in parameters are two very different things and I want to avoid any confusion.

AudioPluginHost

Testing the plug-in with a full DAW such as REAPER or Logic Pro can be slow. Most DAWs are big applications that take a while to load and the DAW must be restarted every time you make a change to the plug-in. All this waiting adds up...

When the plug-in doesn't work correctly due to a bug, you may want to examine its internals by using the IDE's debugger (more about this in a later chapter). This requires running the entire DAW in the debugger. Unfortunately, to thwart software pirates, not all DAWs allow attaching a debugger. On Apple Silicon machines this requires disabling SIP (system integrity protection) which you may not want to do.

Fortunately, JUCE comes with a teeny tiny DAW called **AudioPluginHost** that is really fast to start up and can be used from the debugger. While developing, it's more convenient to run the plug-in inside AudioPluginHost.

The first step is to build AudioPluginHost from its source code:

1. Go to the **JUCE folder** and then to **extras/AudioPluginHost**. Open the **AudioPluginHost.jucer** file in Projucer.

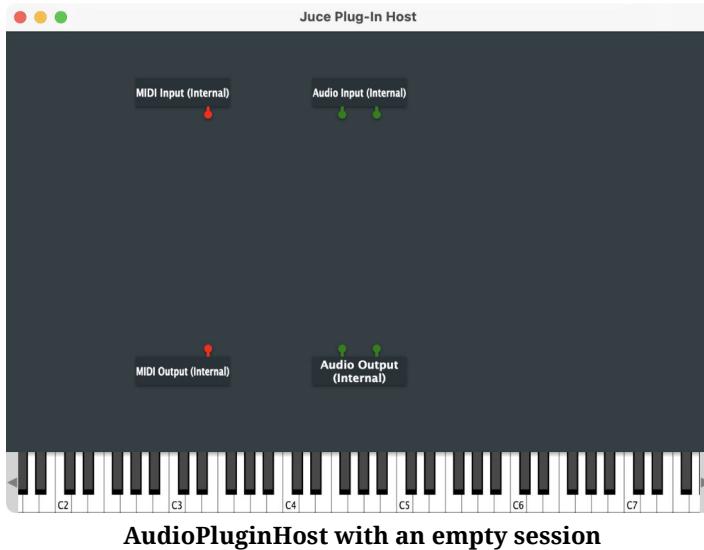
2. As before, you'll need to export this project from Projucer to your IDE, so make sure your IDE is selected in Projucer and press the round export button.

Xcode may give a warning: AudioPluginHost is a project downloaded from the Internet. Are you sure you want to open it? Click **Trust and Open**.

3. Press the IDE's **run button** (the one that looks like a play button) to compile and run the application. In Visual Studio, use the dark green button that says **Local Windows Debugger** or press F5. In Xcode, you can use **Command+R** as a keyboard shortcut.

After the build is successful, the AudioPluginHost application will launch. On Mac it will ask for access to the microphone. It may also ask for permission to access files in your Documents folder.

The first time you start AudioPluginHost, it looks like this:



AudioPluginHost with an empty session

You should see several rectangular blocks that represent the available MIDI and audio inputs and outputs. These blocks may look different on your computer, depending on what kind of audio hardware you have. At the bottom of the window is a virtual piano keyboard.

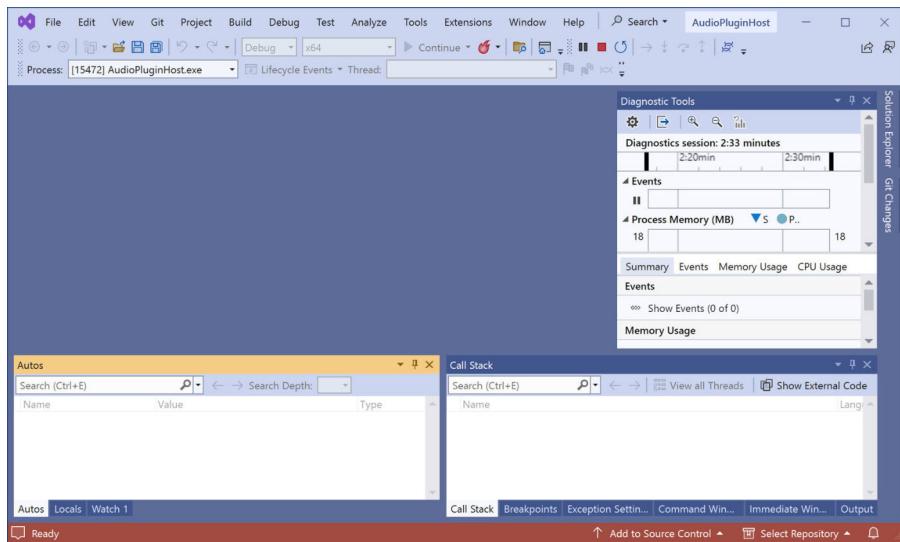
To use AudioPluginHost, you will add a new block for the Delay plug-in, and then connect the output and input pins of these blocks together to create a signal flow from an audio source, through the plug-in, to the audio output.

Before we continue, notice that the IDE looks a little different now, since you're running AudioPluginHost inside the debugger. In Xcode, there is a new "stop" button in the toolbar that can be used to terminate the application. It also pops up a small panel at the bottom of the window with informational messages.



Xcode's toolbar when running AudioPluginHost

In Visual Studio, the entire window changes to show several debugger panels. In the toolbar the run buttons have been replaced by pause and stop buttons.



Visual Studio when running AudioPluginHost

Creating a session

To configure the audio devices in AudioPluginHost, double click on one of the blocks, for example **Audio Input**. This brings up the Audio Settings window.



The Audio Settings window in AudioPluginHost

Here you can change the properties of the audio inputs and outputs used by AudioPluginHost. Use the Test button to verify that your speaker or headphones indeed work. You can also set a sample rate and audio buffer size here.

We won't be using the MIDI inputs and outputs in this book — to learn more about using MIDI, see our book [Creating Synthesizer Plug-Ins with C++ and JUCE²²](#).

If you're happy with the audio settings, close the window.

Before you can add the Delay plug-in to the session, you must tell AudioPluginHost where to find it. In the menu bar, go to **Options > Edit the List of Available Plugins...** to bring up the following window:

Available Plugins				
Name	Format	Category	Manufacturer	
Arpeggiator	Internal	Synth	JUCE	
Audio Input	Internal	I/O devices	JUCE	
Audio Output	Internal	I/O devices	JUCE	
AudioPluginDemo	Internal	Effect	JUCE	
AUv3 Synth	Internal	Synth	JUCE	
DSPModulePluginDemo	Internal	Effect	JUCE	
Gain PlugIn	Internal	Effect	JUCE	
MIDI Input	Internal	I/O devices	JUCE	
MIDI Logger	Internal	Synth	JUCE	
MIDI Output	Internal	I/O devices	JUCE	
Multi Out Synth PlugIn	Internal	Generator	JUCE	
NoiseGate	Internal	Effect	JUCE	
Reverb	Internal	Effect	JUCE	
SamplerPlugin	Internal	Synth	JUCE	
Sine Wave Synth	Internal	Synth	JUCE	
Surround PlugIn	Internal	Effect	JUCE	

The list of available plug-ins

This window shows all the plug-ins that AudioPluginHost knows about. Initially it will only have a few JUCE-provided plug-ins listed, since AudioPluginHost hasn't scanned the plug-ins on your computer yet.

At the bottom of the window is a **Scan mode** option. Here, choose **Out-of-process**. Sometimes plug-ins will misbehave when scanned, crashing AudioPluginHost. By scanning out-of-process, such crashes won't happen.

²²<https://theaudioprogrammer.com/synth-plugin-book>

Click the **Options...** button and then **Scan for new or updated VST3 plug-ins**. This brings up a dialog where you can select the folders to scan. Simply press the **Scan** button to continue.

If you have many plug-ins installed, scanning may take a while. Some plug-ins may ask you to authorize them. On Mac you may wish to scan for Audio Unit plug-ins as well.

Once scanning is done you should have an entry for **Delay** in the list. You can now close this window.

To start using the Delay plug-in, right-click somewhere in **AudioPluginHost**'s main window to bring up a pop-up menu with all the installed plug-ins. You can also use **Plugins > Create Plug-in** from the menu bar.

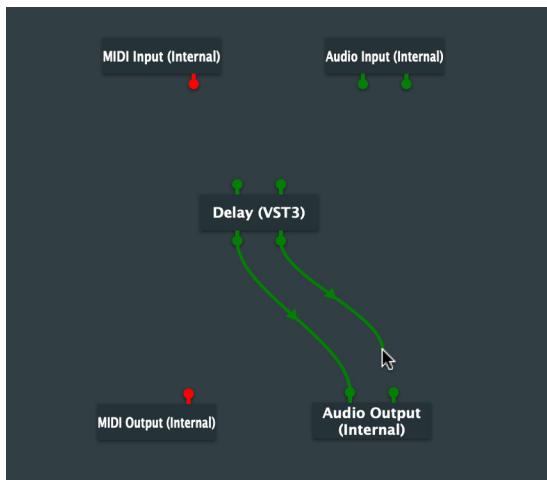


Adding the plug-in to the session

In the pop-up, navigate to **yourcompany** and select **Delay** — or whatever name you decided to give the plug-in. This adds a new block to the session that represents the plug-in.

To view the user interface of the plug-in, double-click the **Delay** block.

Drag from the green pin on the bottom of the **Delay** block to the **Audio Output** block. Repeat this for the other green pin. This connects the output of the plug-in to the system's speakers or headphones.



Connecting the plug-in to the audio output

Note: It's possible your Audio Output has more than two pins. In that case, which pins to use depends on your speaker setup and you may need to experiment a little to find the correct ones.

You also need to connect something to the input of the plug-in. Warning: DO NOT connect the plug-in to the Audio Input block!

If the Audio Input is set to your computer's microphone and the Audio Output is the computer's speakers, the microphone will pick up the sound coming out of the speakers, creating a feedback loop that rapidly gets amplified to an unpleasant volume.

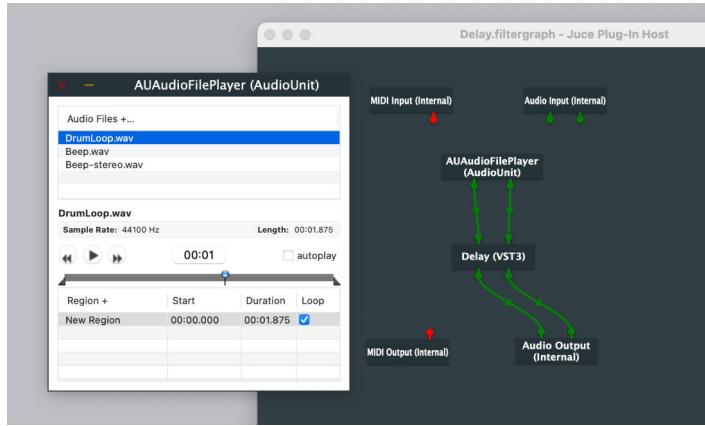
Instead of using the microphone as input, I find it more useful (and less dangerous!) to test the plug-in with a set of audio files.

Before we set that up, first let's save the session to a file. From the menu bar, choose **File > Save** to save this session as a **.filtergraph** file. I saved it as **Delay.filtergraph** on the desktop.

Playing audio files (Mac)

On Mac you can use a plug-in from Apple to play back audio files. Right-click inside the AudioPluginHost window again and from the pop-up choose **Apple > AUAudioFilePlayer**.

Connect the outputs of the new **AUAudioFilePlayer** block to the inputs of the **Delay** block. Double-click the **AUAudioFilePlayer** block to bring up its UI.



Using the AUAudioFilePlayer plug-in in AudioPluginHost

Click **Audio Files+...** at the top to bring up a file picker that lets you choose one or more audio files to add. You can also drag the files into the window. Select the file to play in the list and check the **Loop** option. Then press the **play button** to start playing the audio file.

You should hear the original audio coming out of the computer's speakers, as the Delay plug-in currently simply passes the incoming audio through unchanged. When you're done, press **Command+S** to save the changes to the filtergraph file.

Playing audio files (Windows)

On Windows there unfortunately is not anything built-in that we can use for playing audio files. There are a few options:

1. Use a synthesizer or sampler plug-in. For example, right-click inside the AudioPluginHost window and choose **Sine Wave Synth (Internal)**. Connect the outputs of the new **Sine Wave Synth** block to the inputs of the **Delay** block. Connect the input of **Sine Wave Synth** to the **MIDI Input** block.

Now you can use the on-screen piano keyboard to play notes. Of course, if you have any other synth plug-ins installed you can use those too. If you have a sampler plug-in you can use that to load and play back audio files.

2. Download a plug-in that can play audio files. A quick search finds [vstPlayer](#)²³ by Mirax Labs. I didn't try it and I don't know if it's any good.
3. Download the source code of Matkat Music's AudioFilePlayer and compile it yourself. Since this is a book about learning to build plug-ins, I suggest giving this a try, especially since you've already managed to successfully compile a few things.

Use your favorite web browser to go to [github.com/hollance/AudioFilePlayerPlugin](#)²⁴. If you haven't used GitHub before, this is a website where software developers share their open source projects. The source code of JUCE itself is hosted on GitHub too.

The screenshot shows the GitHub repository page for `hollance / AudioFilePlayerPlugin`. The repository is public and has been forked from `matkatmusic/AudioFilePlayer`. The main tab is `Code`, which is currently selected. The commit history shows one commit ahead of the `master` branch. The commits are:

- `always loop the audio file` by `efdf5e0` 3 weeks ago (24 commits)
- `Source` by `efdf5e0` 3 weeks ago
- `.gitignore` by `efdf5e0` 3 years ago
- `AudioFilePlayer.jucer` by `efdf5e0` 3 weeks ago
- `LICENSE` by `efdf5e0` 3 years ago

On the right side, there is an `About` section with links to `Readme`, `GPL-3.0 license`, `Activity`, `1 star`, `0 watching`, `0 forks`, and a `Report repository` link.

The AudioFilePlayer project on GitHub

Compiling this plug-in involves nothing that you haven't done before. So, let's go! The steps for building the AudioFilePlayer plug-in are:

1. On the GitHub page, click the green **Code** button and choose **Download ZIP**.
2. After a few moments your Downloads folder will receive a new file, **AudioFilePlayerPlugin-master.zip**. Extract the contents of this file.
3. Open the file **AudioFilePlayer.jucer** in Projucer.
4. At the top of the Projucer window, select the **Visual Studio 2022** exporter and press the round button to open the IDE.

²³<https://miraxlabs.com/products/vstplayer/>

²⁴<https://github.com/hollance/AudioFilePlayerPlugin>

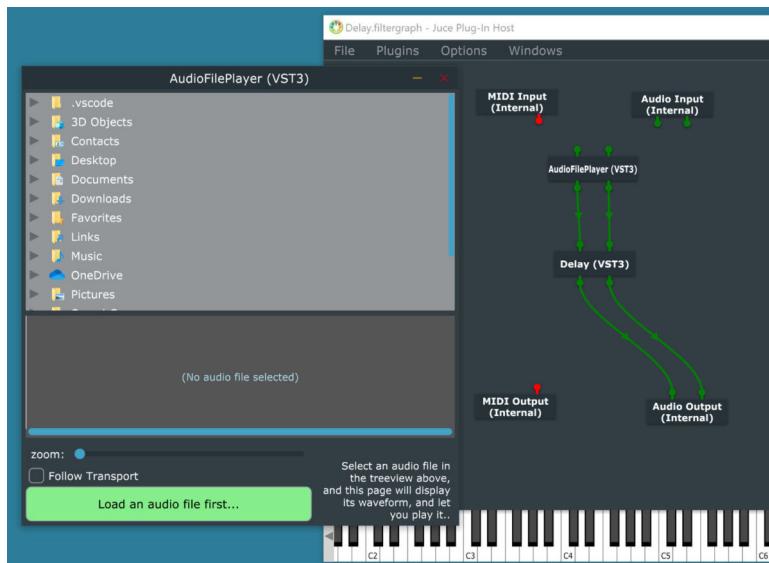
5. In Visual Studio in the **Solution Explorer**, right-click on **AudioFilePlayer_VST3** and choose **Build** from the pop-up menu.

If all goes well, within a few minutes the build should complete and you will have a new **AudioFilePlayer.vst3** in `C:\Program Files\Common Files\VST3`. You can now close the Visual Studio window.

Quit **AudioPluginHost**, saving any changes, and start it again. It should automatically open the most recently used session, but if not, use **File > Open...** to load your **Delay.filtergraph** session.

From the **Options** menu, choose **Edit the List of Available Plug-ins...**, and scan for new VST3 plug-ins so that **AudioPluginHost** can find the plug-in you just compiled. Back in the main **AudioPluginHost** window, right-click and choose **Matkat Music LLC > AudioFilePlayer** to add it to the session.

Connect the outputs of the new **AudioFilePlayer** block to the inputs of the **Delay** block. Double-click the **AudioFilePlayer** block to bring up its UI:



Using the AudioFilePlayer plug-in in AudioPluginHost

Use the file browser to navigate to an audio file. Once you've selected an audio file, the green button at the bottom says **Start**. Press this button to start playing the audio file.

You should hear the original audio coming out of the computer's speakers, as the Delay plug-in currently simply passes the incoming audio through unchanged. The AudioFilePlayer plug-in will automatically loop the audio file until you press **Stop**.

When you're done, press **Ctrl+S** to save the changes to the filtergraph file.

Automatically starting AudioPluginHost

So far you've been running the AudioPluginHost application from your IDE. However, it's cumbersome to have an IDE window open all the time just to run this application. There is a smarter way to do this.

First, quit AudioPluginHost. There are two ways to quit it:

1. Close its window like any other application, or choose **File > Quit** from the menu.
2. Press the **stop button** in the IDE's toolbar.

The difference is that closing the window will properly shut down the app, allowing it to save any changes to the session first. The stop button will immediately terminate the application, for example when it's frozen and no longer responds to user input.

Under normal circumstances it's better quit AudioPluginHost the regular way. But during development it can happen that your code has a mistake and starts acting weird — such as outputting very loud sounds — in which case the stop button is a useful emergency break.

Next, it's a good idea to copy the AudioPluginHost application to somewhere more accessible, as right now it's hidden deep away in a build folder somewhere.

For Xcode: In the **Project Navigator**, scroll down to the **Products** group and expand it. There is one entry there, **AudioPluginHost.app**. Right-click and choose **Show in Finder**. This brings up a new Finder window. From here, drag the **AudioPluginHost** app onto the Desktop, or any other folder of your choosing.

For Visual Studio: In the **Solution Explorer**, right-click on **AudioPluginHost_App** and from the pop-up menu choose **Open Folder in File Explorer**. This brings up a new File Explorer window. Go into the **x64** subfolder, then **Debug**, then **App**. From here, drag the **AudioPluginHost.exe** application onto the Desktop, or any other folder of your choosing.

Now AudioPluginHost is easy to find and you can simply double-click to launch it. However, we are going to set up the IDE so that it automatically launches AudioPluginHost and loads the Delay.filtergraph session after building the plug-in.

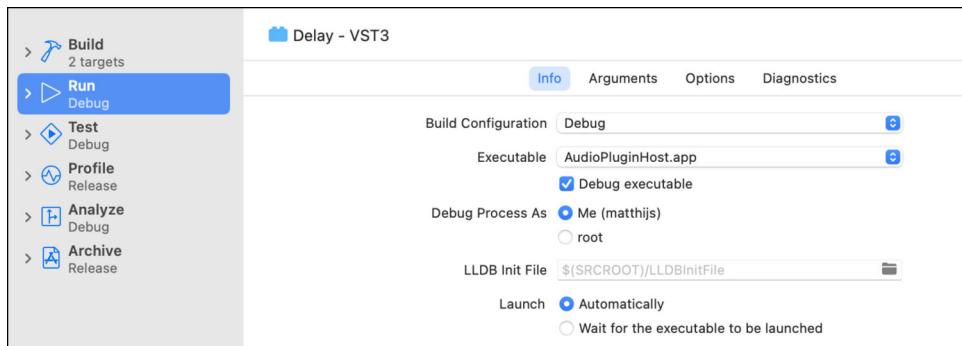
You can close the AudioPluginHost window in your IDE and in Projucer, we won't need it again.

Note: Whenever you update JUCE to a new version, make sure to also build the new version of AudioPluginHost using the same steps as described above and then copy it to the desktop (or wherever you put it). This way you're always using the most recent version.

You will now set up your IDE so that pressing the **run button**, which as you'll recall didn't work for the plug-in, will automatically launch AudioPluginHost. First, go back to the IDE window for the Delay plug-in project.

In Xcode, select the **Delay - VST3** target and choose **Edit Scheme...** from the pop-up.

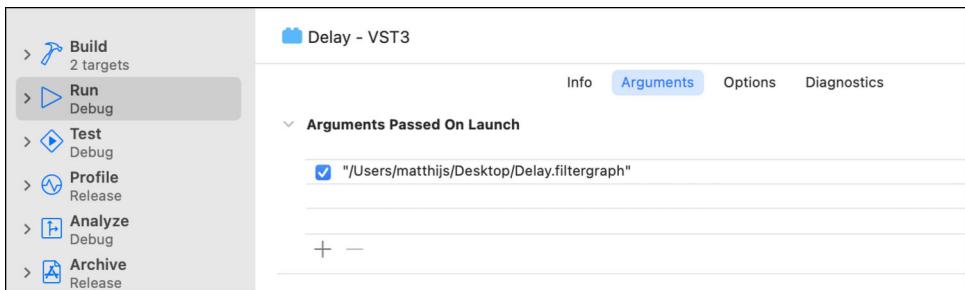
In the window that appears, select **Run** on the left. Click on **Executable** and choose **Other...** from the pop-up. Navigate to the desktop and select **AudioPluginHost.app**.



Setting up Xcode to automatically run AudioPluginHost

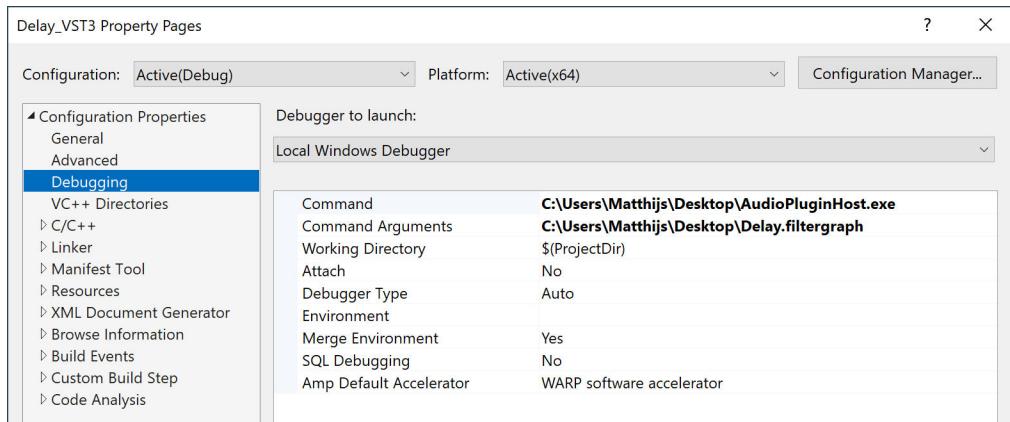
AudioPluginHost will automatically load the most recent session when it starts up, but you can also tell it to open a specific session.

Still in the Xcode scheme editor, in the **Arguments** tab, under **Arguments Passed On Launch**, add the path to the **Delay.filtergraph** file. If there are spaces in this path, put the whole thing in double quotes.



Telling AudioPluginHost which graph to load

In Visual Studio, right-click **Delay_VST3** in the **Solution Explorer** and choose **Properties**. In the window that pops up, go to **Debugging**. Change **Command** to where `AudioPluginHost.exe` lives. Set **Command Arguments** to the path to where you saved the `Delay.filtergraph` file.



Setting up Visual Studio to automatically run AudioPluginHost

Now when you press the **run button**, the IDE will compile the VST3 file and copy it into the system folder for VSTs, just like before. Then it automatically launches `AudioPluginHost`, which should load in a matter of seconds, and open the session from the `Delay.filtergraph` file, so that you can immediately start using and testing your plug-in. Try it out!

There is no need to rescan for new plug-ins — `AudioPluginHost` will always load the most recent version of the plug-in. Using this method, `AudioPluginHost` will run inside the IDE's debugger, so that if the plug-in is not doing what it's supposed to, you can use the debugger to figure out what is wrong.

Note: You may sometimes find that, after making certain changes in Projucer, the settings for launching AudioPluginHost have mysteriously vanished from your IDE. Usually, Projucer will update the exported project without breaking things but sometimes it completely overwrites the project and you'll have to set everything up again. So, if the **run button** suddenly stops working or gives that “Unable to start program” error message, double-check whether AudioPluginHost is still set as the executable for the project.

The JUCE splash screen (JUCE 7 only)

If you’re using JUCE 7, you probably noticed that the plug-in briefly displays a JUCE logo when you open the editor window:



The JUCE splash screen

When using the free educational or personal plan of JUCE, you are required to show this splash screen to credit the makers of JUCE. The paid indie and professional plans do not need to show the splash screen. However, there is a free way around this: you can put Projucer in GPLv3 mode.

GPLv3 is an open source license, and plug-ins that are made open source under this license do not need to show the JUCE logo. Using GPLv3 mode doesn’t mean that you are required to open source your plug-ins — as long as you’re writing the plug-in for personal use or for learning, you can do whatever you want.

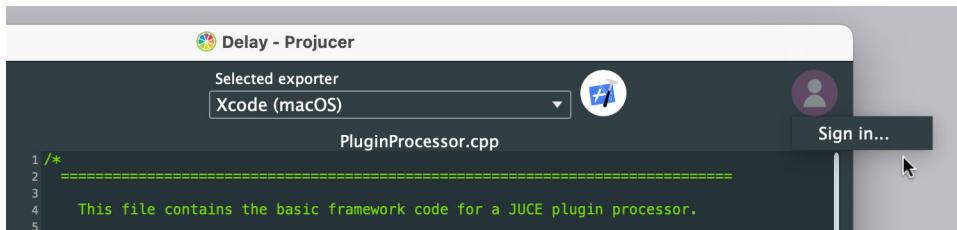
Note: JUCE 8 removed this splash screen requirement for the free plans. You can still use JUCE 8 in open source mode, in which case the license is GPLv3. You may skip the following instructions if you’re using JUCE 8.

It's only when you start distributing the plug-in to other people that you need to choose between:

- open source using the GPLv3 license, or
- educational / personal license but showing the splash screen, or
- a paid license.

So, if you're just following along with this book, and you don't want to see this splash screen, then GPLv3 mode is the right choice.

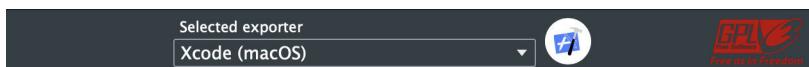
Go back to **Projucer** if you still have it open. If not, open it again and load the **Delay.jucer** project. Notice there is an account icon in the top-right corner.



The account icon in Projucer

Click this icon and choose **Sign In**. In the dialog that appears, click **Enable GPL Mode**. Of course, if you do have a paid license, then you can sign in here.

The icon in Projucer's top-right corner now changes to a logo that says GPLv3.



GPLv3 mode enabled

Next, click the gear icon to go to the project settings and change the option **Display the JUCE Splash Screen** to **Disabled**. This will turn off the logo. Note that disabling this option while you haven't enabled GPLv3 mode (or have not signed in) is not allowed and you won't be able to save the project.

Important for Windows users! From now on, when making changes in Projucer, do not press the round button to export the project to Visual Studio. Instead, use **File > Save Project** or press **Ctrl+P**.

When you press the round button, Projucer will overwrite all the Visual Studio project files and opens the project in a new IDE window. You'll lose the settings that automatically start AudioPluginHost (see previous section). Using **Save Project** prevents this from happening. Now Visual Studio will ask to reload the project (choose **Reload All**) rather than treat it as a completely new project.

On Mac with Xcode, this doesn't happen, although it's perfectly fine to use **Save Project** or **Command+P** there too.

Press the **run button** to compile and run the plug-in again. The JUCE logo is gone when you open the plug-in's editor.

Note: At the time of writing, the cost for the JUCE 7 indie plan is \$40/month per developer up to a revenue limit of \$500k per year. The expected cost for JUCE 8 will be a little higher. That's not cheap but it's very fair for a single developer or a small company making commercial plug-ins, especially when you consider how much development time is saved by using JUCE.

For a hobby developer who wants to make free plug-ins for fun, JUCE 8's starter plan is a good choice. It does not have the splash screen requirement. Of course, it's always possible to use JUCE for free if you open source your work under the terms of the GPLv3 (for JUCE 7) or AGPLv3 (for JUCE 8) licenses.

If you want your plug-ins to come across as professional, there are other costs involved in doing audio development as well, such as an Apple Developer Program membership (\$100 per year) and signing certificates for Windows (at least \$100 per year but typically more), web hosting costs, and more. There are also free alternatives to JUCE such as iPlug2.

Excellent, in this chapter you've learned how to load the plug-in into a DAW and AudioPluginHost. In the next chapter you'll start writing your first audio processing code.

7: Output gain

In the previous chapter you got a little taste of writing code when you changed the paint function of the plug-in's editor object. In this chapter you're going to write code for real and actually make the plug-in process some audio.

You'll start with the traditional beginner project in audio programming: making a gain control. In audio, **gain** means a change in the amplitude of the sound. Roughly, the amplitude corresponds to how loud or how quiet the sound is. Lots of gain means lots of loudness.

Many plug-ins have a knob to set the **Output Gain**, to compensate for any level increases or decreases resulting from the plug-in's audio processing. Our Delay plug-in will also need such a control.

The red outline highlights the part of the plug-in you'll build in this chapter and the next:



For now, we'll focus just on the audio processing side of the plug-in, and create the UI in a later chapter. Before we start writing the gain code, let's first take a tour of the audio processor object.

The audio processor

Currently, the plug-in code consists of two objects:

- `DelayAudioProcessorEditor`. This handles the user interface of the plug-in. You've briefly played with this in the previous chapter.
- `DelayAudioProcessor`. This is the main object of the plug-in. It handles communication with the host and does the most important thing a plug-in can do: process the audio (hence its name).

By the end of this chapter you should have a pretty good understanding of how the `DelayAudioProcessor` works, and which parts you need to change to do your own audio processing.

The code for `DelayAudioProcessor` lives in two files: `PluginProcessor.h` and `PluginProcessor.cpp`. We'll look at the header file first.

Open up `PluginProcessor.h` in the IDE. Skipping a few lines at the beginning, this file has the following code:

```
class DelayAudioProcessor : public juce::AudioProcessor
{
public:
    //==============================================================================
    DelayAudioProcessor();
    ~DelayAudioProcessor() override;

    //==============================================================================
    void prepareToPlay (double sampleRate, int samplesPerBlock) override;
    void releaseResources() override;

#ifndef JucePlugin_PREFERREDCHANNELCONFIGURATIONS
    bool isBusesLayoutSupported (const BusesLayout& layouts) const override;
#endif

    void processBlock (juce::AudioBuffer<float>&, juce::MidiBuffer&) override;

    // ...other functions omitted...

private:
    //==============================================================================
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (DelayAudioProcessor)
};
```

The **.h** or **header file** tells the compiler that our plug-in has an object named `DelayAudioProcessor`. It also lists the functions that belong to this object.

In general, the definition of an object in C++ has this form:

```
class DelayAudioProcessor : public juce::AudioProcessor
{
    // ... everything that belongs to the object...
};
```

The first line says that `DelayAudioProcessor` is a new object that is based on another object, `juce::AudioProcessor`.

The `juce::AudioProcessor` object is provided by JUCE and knows everything about how to communicate with the host. By making our own object **inherit** the functionality from `juce::AudioProcessor`, we don't have to write all that host communication code ourselves. Instead, we only have to write the functions that deal with the audio processing that is specific to our plug-in.

Everything that belongs to the `DelayAudioProcessor` object is inside { and } braces. After the final } there needs to be a semicolon — this semicolon is often overlooked but forgetting it results in a compiler error.

Following the opening { brace is the next line:

```
public:
```

An object has public and private parts. Everything that follows `public:` is visible to other objects in the program. Everything after `private:` can only be used by `DelayAudioProcessor` itself.

As an analogy, consider a car. The controls needed to operate the car such as the steering wheel and gas pedal are accessible to the driver (public), but the engine is hidden below the bonnet (private) and isn't something you normally need to deal with when driving.

The other lines in the **.h** file declare the functions that the `DelayAudioProcessor` should provide in order to act as a proper plug-in. Whenever the host needs the plug-in to do something, it will call the corresponding function.

In this book we will mostly work with the following functions.

```
DelayAudioProcessor();
```

A function with the same name as the object is a **constructor**. When the host loads the plug-in into the computer's memory, it does so using the constructor.

```
-DelayAudioProcessor() override;
```

There is another function with the same name as the object but with a ~ tilde in front. This is the **destructor**. If the user removes the plug-in from a track, the host unloads the plug-in from the computer's memory by calling the destructor.

```
void prepareToPlay (double sampleRate, int samplesPerBlock) override;
```

The host calls `prepareToPlay` when it wants to start playing audio through this plug-in. Having advance notice gives the plug-in the opportunity to get ready. For example, our Delay plug-in will use `prepareToPlay` to reserve enough free space in the computer's memory to store up to 5 seconds of stereo sound.

This function has two arguments: the sample rate and the maximum block size. Recall that audio is processed in groups of multiple samples at a time, known as blocks. Through the `samplesPerBlock` argument, the host tells the plug-in how large a block size to expect.

```
void releaseResources() override;
```

The `releaseResources` function is the counterpart to `prepareToPlay`. It is called when the host has stopped playing audio, and is used to let the plug-in free any resources it no longer needs.

```
#ifndef JucePlugin_PREFERREDCHANNELCONFIGURATIONS
bool isBusesLayoutSupported (const BusesLayout& layouts) const override;
#endif
```

The host will transport the audio data to and from the plug-in over a **bus**. This terminology comes from physical mixing consoles and refers to how audio is routed between different parts of the mixing console. The `isBusesLayoutSupported` function negotiates with the host whether the buses used will be mono, stereo, or have even more channels.

Note: The `#ifndef` and `#endif` directives surrounding this function will disable `isBusesLayoutSupported` if an option in Projucer named Plugin Channel Configurations is used. We left this option blank, so our plug-in *will* use the `isBusesLayoutSupported` function. I only mention this because you'll encounter other `#ifndef` and `#if` directives in the code, so now you know these are used to enable or disable Projucer configuration options.

The most important function in `DelayAudioProcessor` is `processBlock`:

```
void processBlock (juce::AudioBuffer<float>&, juce::MidiBuffer&) override;
```

This is where the **DSP** or digital signal processing happens. Inside this function is where you will put the plug-in's audio processing code.

The host calls the `processBlock` function several hundred or even thousands of times per second, depending on the chosen sample rate and block size. Isn't that amazing? It does mean that whatever happens in `processBlock` better be fast, or it will miss the audio deadline!

```
void getStateInformation (juce::MemoryBlock& destData) override;
void setStateInformation (const void* data, int sizeInBytes) override;
```

These two functions are used by the host to save and restore the plug-in's parameter values. This allows the user to save their DAW session and then load it again at a later time. Users wouldn't appreciate it if the plug-in forgot what the values for its parameters were!

There are many more functions in `DelayAudioProcessor` but for most plug-ins it's enough to implement the ones we just looked at.

Inside PluginProcessor.cpp

The header file tells the compiler which functions are in the `DelayAudioProcessor` object, but not what they actually do. To see the code that is inside these functions, we have to look at `PluginProcessor.cpp`.

Open `PluginProcessor.cpp` and scroll down to about halfway where you'll find the `prepareToPlay` function.

```
void DelayAudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
{
    // Use this method as the place to do any pre-playback
    // initialisation that you need..
}
```

This is the **implementation** of the `prepareToPlay` function. It is currently empty except for a helpful comment.

Your IDE may warn that the arguments `sampleRate` and `samplesPerBlock` are unused, which is correct as there is no actual code inside the `{` and `}` brackets yet.

Notice that the comment refers to `prepareToPlay` as a “method”. That’s another commonly used name for “function” — older gray-haired coders like myself might also use the terms “procedure” or “subroutine” to mean the same thing. You’ll often find that a particular concept in software development has multiple names, which can be confusing at times.

It is possible to write the code for `prepareToPlay` or any other function directly inside the `.h` file, but in C++ it’s more common to put it in the `.cpp` file. Most other programming languages don’t have this kind of division between a header file containing the declarations (the “what”) and a source file containing the implementations (the “how”). It’s one of the peculiarities of C++.

At this point I suggest having a quick look through the rest of `PluginProcessor.cpp` just to get a sense of what’s going on.

When Projucer created these source code files it added basic implementations for the `DelayAudioProcessor`’s functions so that they already do something meaningful where necessary. Sometimes they do nothing. The idea is that you will fill in these functions with the functionality your plug-in needs.

Many of these functions don’t need changing. For example, the `isMidiEffect` function says to the host, “No this plug-in is not a MIDI effect”. That’s correct, since we will be handling audio data only, not MIDI. Its implementation looks like the following:

```
bool DelayAudioProcessor::isMidiEffect() const
{
    #if JucePlugin_IsMidiEffect
    return true;
    #else
    return false;
    #endif
}
```

The `#if` directive checks whether the `JucePlugin_IsMidiEffect` configuration option is enabled to determine whether this plug-in supports MIDI or not. This corresponds to the Plugin Characteristics option in Projucer.

In our case this option is not enabled because we're building an audio effect plug-in, not a MIDI effect. And so when the host asks the plug-in whether it can do MIDI, the `isMidiEffect` function returns `false`, which is the C++ way of saying "no".

You can find the values of these configuration options in the source file **JucePlugin-Defines.h** inside the **JUCE Library Code** group. That's how Projucer's settings make it into the C++ code. (In Visual Studio this is under the `Delay_SharedCode` target.)

All right, that's enough exploration. Time to write some audio code!

Note: I called `DelayAudioProcessor` an object. Technically speaking I should have said it's a **class**, which is like a blueprint. An object is an actual thing in the computer's memory made using that blueprint. That may be a weird concept to wrap your head around, so maybe an example will help.

Let's say we have a class `Filter` that can remove high or low tones from audio. In the plug-in we may want to have multiple filters at the same time, such as low-pass filter followed by a high-pass filter. This means we will need two **instances** of class `Filter`, in other words two filter objects, one configured to act as low-pass and one as high-pass.

Usually when we say something is an object, we mean a specific instance of a class rather than the class itself. I hope that distinction makes sense. If not, don't worry about it for now. Just remember that writing `class XYZ` means we're defining something that can be used as an object in our program.

Applying gain

Changing the level of an audio signal is simply a matter of doing a multiplication. For example, if you multiply every sample in the signal by 0.5, the new amplitude is only half as high. This doesn't mean the sound is also half as loud — amplitude is related to loudness but they're not exactly the same thing — but it will definitely be quieter than before.

Let's implement the simplest gain possible. Go to **PluginProcessor.cpp** and find the `processBlock` function. As mentioned, this is the place where the plug-in will do the

actual audio processing.

The code currently looks like the following. As before, I've wrapped some lines to make them fit in the book.

```
void DelayAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer,
                                         juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;
    auto totalNumInputChannels = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();

    // In case we have more outputs than inputs, this code clears any output
     // channels that didn't contain input data, (because these aren't
     // guaranteed to be empty - they may contain garbage).
     // This is here to avoid people getting screaming feedback
     // when they first compile a plugin, but obviously you don't need to keep
     // this code if your algorithm always overwrites all the output channels.
    for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
        buffer.clear (i, 0, buffer.getNumSamples());

    // This is the place where you'd normally do the guts of your plugin's
     // audio processing...
     // Make sure to reset the state if your inner loop is processing
     // the samples and the outer loop is handling the channels.
     // Alternatively, you can process the samples with the channels
     // interleaved by keeping the same state.
    for (int channel = 0; channel < totalNumInputChannels; ++channel)
    {
        auto* channelData = buffer.getWritePointer (channel);

        // ..do something to the data...
    }
}
```

I want you to replace the following portion of the code,

```
for (int channel = 0; channel < totalNumInputChannels; ++channel)
{
    auto* channelData = buffer.getWritePointer (channel);

    // ..do something to the data...
}
```

with this single line:

```
buffer.applyGain(0.5f);
```

So that `processBlock` looks like this, leaving out the comments for brevity:

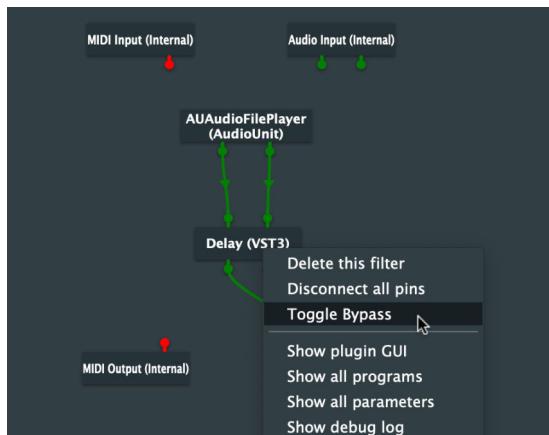
```
void DelayAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer,
                                         juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;
    auto totalNumInputChannels = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();

    for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
        buffer.clear (i, 0, buffer.getNumSamples());

    buffer.applyGain(0.5f);
}
```

That's all we're going to do for now. Build the plug-in and run it in `AudioPluginHost`. Play some sound through it. You should still hear the audio but it will be less loud than before. After all, we cut its amplitude in half.

To verify the plug-in indeed works as intended, while the sound is playing, right-click on the **Delay** block in `AudioPluginHost` and choose **Toggle Bypass** from the pop-up.



Bypassing the Delay plug-in

The **Delay** block turns gray, which means the plug-in is being bypassed, and the sound should be back to its original volume. If you can't hear the difference very well, try it with headphones. Also feel free to try this out in your DAW of choice and toggle the DAW's bypass button to hear the difference, or look at the output level meter on the mixing console.

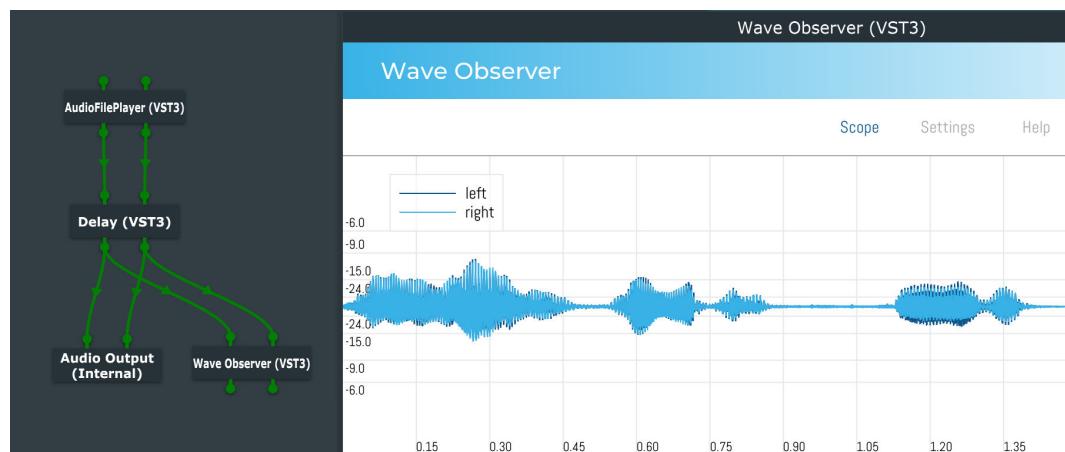
Try playing with gain other than $0.5f$. Remember to put an f behind the number, to indicate this is a floating-point number or **float**.

Some tips:

- Be careful with values larger than $1.0f$ as they increase the level.
- A gain of $1.0f$ has no effect, since multiplying by one doesn't change anything.
- A gain of $0.0f$ creates silence.
- Values less than $0.0f$ will work too. This does the same as gains larger than zero but they also invert the polarity (swap the phase) of the signal.

While audio programming involves a lot of listening to make sure stuff sounds good, it always helps to have visual confirmation as well. I suggest that you download and install an oscilloscope or waveform viewer plug-in, such as [s\(M\)exoscope²⁵](#), or [Signalizer²⁶](#), or [Wave Observer²⁷](#). After installing one of these plug-ins, restart AudioPluginHost and rescan for new plug-ins.

Hook up the oscilloscope plug-in like so in AudioPluginHost:



Viewing the output of the plug-in on an oscilloscope

Toggling the bypass on the **Delay** plug-in clearly shows on the oscilloscope that the amplitude of the waveform gets halved.

²⁵<http://armandomontanez.com/smexoscope/>

²⁶<https://www.jthorborg.com/signalizer.html>

²⁷<https://pressplay-music.com/wave-observer/>

Using loops

Great, we successfully changed the level of the audio signal! But you may be wondering how any of this actually works.

Let's look at the arguments for the `processBlock` function.

```
void DelayAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer,
                                         juce::MidiBuffer& midiMessages)
```

There are two arguments:

1. `juce::AudioBuffer<float>& buffer`
2. `juce::MidiBuffer& midiMessages`

Recall from chapter 2 that every few milliseconds the host sends a block of audio samples to the plug-in, and expects the plug-in to process these samples before the deadline.

The audio samples are placed in a `juce::AudioBuffer` object. This object is given to `processBlock` in the first argument. Inside the function you can refer to this `juce::AudioBuffer` object by the name `buffer`. The terms `block` and `buffer` mean the same thing and are used interchangeably.

The audio samples in the `AudioBuffer` are floating-point values, typically between `-1.0f` and `1.0f`. If the block size is 128, there are 128 of these `float` values in the `AudioBuffer`. For a stereo signal, the `AudioBuffer` contains two times 128 samples, one list of numbers for the left channel and one list of numbers for the right channel.

The job of `processBlock` is to overwrite the sample values inside the `AudioBuffer` with new ones. The line `buffer.applyGain(0.5f);` does exactly that: It multiplies all the samples in the `AudioBuffer` with 0.5, making them less loud.

The `buffer.applyGain` function is very handy but we can also apply the gain ourselves. Let's try it, just to see how to do this by hand. It will hopefully help you to understand a little better what exactly goes on under the hood of `buffer.applyGain`.

Replace the line `buffer.applyGain(0.5f);` with the following code.

```

for (int channel = 0; channel < totalNumInputChannels; ++channel) {
    auto* channelData = buffer.getWritePointer(channel);

    for (int sample = 0; sample < buffer.getNumSamples(); ++sample) {
        channelData[sample] = channelData[sample] * 0.5f;
    }
}

```

This does the exact same thing, but spelled out. Build and run the plug-in and it should work the same as before.

The **for** keyword creates a **loop**, which is a section of code that is repeated one or more times. Here we have two loops, one inside the other. The { and } braces determine which lines of code belong to which loop.

If you squint, it looks like this:

```

for (... loop through the channels ...) {
    // code that belongs to the outer loop

    for (... loop through the samples ...) {
        // code that belongs to the inner loop
    }
}

```

The `juce::AudioBuffer` object contains the audio data for all the channels in the signal. The first loop, known as the **outer loop**, looks at these channels in turn. For a mono signal there is only one channel and the loop is performed just once. But a stereo track has two channels and everything inside the loop is done twice: first for the left channel, and then for the right channel.

Notice how the code inside the loop has been indented by four spaces. This makes it easier to see to which loop each line of code belongs.

The second loop, known as the **inner loop**, looks at all the samples in the current channel and multiplies each by `0.5f`, using the statement:

```
channelData[sample] = channelData[sample] * 0.5f;
```

The `*` symbol means multiplication. After multiplying, it writes the new value back into the channel's audio data.

If the block size is 128 samples, the inner loop does the same as:

```
channelData[0] = channelData[0] * 0.5f;
channelData[1] = channelData[1] * 0.5f;
channelData[2] = channelData[2] * 0.5f;

// ... and so on ...

channelData[126] = channelData[126] * 0.5f;
channelData[127] = channelData[127] * 0.5f;
```

But we wouldn't want to write these 128 lines of code by hand — that would be enormously tedious — so we use a loop to let the computer do this for us!

An important thing to remember is that in C++ and most other computer languages, counting starts at 0. That means, if there are 128 samples in the buffer, the first sample has index 0 and the last sample has index 127.

Another reason to use a loop is that we don't actually know the block size will be 128. It could be more or it could be less. The block size is determined by the host and by the user. On some hosts, in particular FL Studio, the block size might occasionally be as small as 1 sample. When doing so-called offline rendering in the DAW, the block size may be as large as 4096 samples. And sometimes it can even be 0 samples!

The code in `processBlock` therefore can't assume anything about what the block size will be, only that it's not going to be larger than the maximum block size that was passed into `prepareToPlay`.

This is why the inner loop is written as follows:

```
for (int sample = 0; sample < buffer.getNumSamples(); ++sample) {
    channelData[sample] = channelData[sample] * 0.5f;
}
```

The expression `int sample = 0;` creates a **variable** named `sample` that starts counting at 0. The keyword `int` tells the compiler that `sample` is an integer, in other words, a variable that stores whole numbers.

We use the value of `sample` to index `channelData`, by writing `channelData[sample]`. This looks at the `sample`-th element in the `AudioBuffer`'s list of sample values for this channel.

On the first go through the loop, `sample` is 0, and so we look at `channelData[0]`. This is a float value with the amplitude of the first sample in the buffer.

After every repetition of the loop, the expression `++sample` increments the value of the `sample` counter by one. On the second pass through the loop, `sample` is 1 and we look at `channelData[1]`. On the third iteration, `sample` is 2 and we look at `channelData[2]`, and so on.

The loop keeps repeating as long as `sample` is less than the number of samples in the buffer, as expressed by `sample < buffer.getNumSamples()`. The symbol `<` is the less-than operator. It checks if the value of the thing on the left, `sample`, is still less than the value of the thing on the right, `buffer.getNumSamples()`.

In our example of a 128-sample block, the loop keeps repeating until `sample` equals 128 and then it stops. Since the loop will be skipped at that point, the very last sample it has looked at is `channelData[127]`.

It takes a little getting used to understanding how these `for` loops work, but in general the form is:

```
for (start value; condition; increment) {
    // ... the code inside the loop ...
}
```

In our inner loop these are:

- start value: `int sample = 0`
- condition: `sample < buffer.getNumSamples()`
- increment: `++sample`

In the outer loop they are:

- start value: `int channel = 0`
- condition: `channel < totalNumInputChannels`
- increment: `++channel`

The loop keeps repeating until the condition is false. On every new repetition of the loop, it increments the counter and then checks the condition again. As long as the condition is true, the code inside the loop keeps getting executed.

Note: I don't know if you noticed this small detail, but we put the { brace on the same line as the `for`, whereas the opening brace for the `processBlock` function is on the next line. C++ doesn't really care where you put the { and } as long as they go around the lines of code that belong to the loop or function. With `for` loops and `if` statements I personally prefer to put the opening brace on the same line, but for functions and classes I put them on the next line. Different developers use different coding styles — many flamewars have been fought over which one is best.

By the way, instead of writing,

```
channelData[sample] = channelData[sample] * 0.5f;
```

you can use the shorthand *= operator:

```
channelData[sample] *= 0.5f;
```

This reads the value from the audio buffer, multiplies it by `0.5f`, and then puts it back into the audio buffer in the same location.

By now you know that writing `0.5f` means this is a floating-point number or `float`. You can also write `0.5` without the `f` at the end. That still is a floating-point number but of type `double`, which stands for double precision. Values with the `f` are single precision.

The difference is that a `double` can describe much smaller and much larger numbers than a `float`, but at the cost of using twice the memory. Some developers do their audio processing with `double` to take advantage of the increased precision, but in this book we will use `float` for everything.

It's not an error to write `0.5` instead of `0.5f` when using `float`, but if the compiler expects a `float` value instead of a `double`, it's better to include the `f`.

You could even have written `channelData[sample] /= 2`; which mathematically is the same thing — dividing by two is the same as multiplying by $1/2$ or `0.5`. However, the value `2` without a decimal point is an `int`, not a `float`.

Whenever the C++ compiler comes across code that mixes numbers of different types — `float` with `double`, or `float` with `int` — it needs to first do a data type conversion. The `int` must be turned into a `float` before it can do the division. These extra conversions mean the CPU needs to do more work and are therefore less optimal.

A closer look at processBlock

Since you'll be spending a lot of time working in `processBlock`, it will be good to understand what else is going on in there.

At the top of the `processBlock` function is the following line:

```
juce::ScopedNoDenormals noDenormals;
```

As you've seen, the sample amplitudes in the `AudioBuffer` object are floating-point numbers, typically between `-1.0f` and `1.0f`. But the full range of such a number is much greater: `floats` can store values as small as 10^{-38} and as large as 10^{38} . We won't be using the very large numbers (that would be extremely loud!) but we do use the small numbers.

For example, when a sound is fading out, its amplitude becomes smaller over time, slowly dropping towards zero. Once the sample value becomes smaller than 10^{-38} , the `float` switches to a different internal representation known as a denormal or subnormal number. The problem is that these denormals are super slow, causing the CPU usage of your plug-in to suddenly spike.

By using a `juce::ScopedNoDenormals` object we tell the CPU to automatically round off floating-point numbers to zero when they become really small instead of making them denormals. For audio programming this rounding to zero is perfectly acceptable, since these very small sample values are not audible anyway. Now we no longer have to worry about this issue at all. Nice!

Note: The term “scoped” in `ScopedNoDenormals` means that JUCE automatically restores the CPU's previous behavior once `processBlock` ends, so that this rounding-to-zero only happens inside `processBlock` but doesn't affect any other operations using floating-point numbers in our program.

A **scope** in C++ is the region between an opening `{` and closing `}` brace. For example, everything inside `processBlock` is in that function's scope, while everything inside the `for` loop is in its own scope as that has its own `{` and `}` braces. Scope is an important concept in C++ because it is used to manage the lifetime of objects and other variables.

The next piece of code in `processBlock` is as follows:

```
auto totalNumInputChannels = getTotalNumInputChannels();
auto totalNumOutputChannels = getTotalNumOutputChannels();

// In case we have more outputs than inputs, this code clears any output
// channels that didn't contain input data, (because these aren't
// guaranteed to be empty - they may contain garbage).
// This is here to avoid people getting screaming feedback
// when they first compile a plugin, but obviously you don't need to keep
// this code if your algorithm always overwrites all the output channels.
for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
    buffer.clear (i, 0, buffer.getNumSamples());
```

The purpose of these lines is to write silence into the audio buffer — samples with the value `0.0f` — if there are more output channels than input channels.

Why is this needed? Suppose the plug-in has a mono input bus (one channel) and a stereo output bus (two channels). The `juce::AudioBuffer` object will then contain two channels of sample data. The first channel is filled with the audio from the mono input, but the second channel will be filled with arbitrary numbers, or in official C++ terminology, garbage.

To prevent the plug-in from outputting these garbage numbers — which doesn't sound very pleasant to say the least — we replace everything in that unused input channel with silence.

This is a technique known as “defensive programming”. Usually this situation won't happen — typically the number of output channels equals the number of input channels — but just in case it does, this little code snippet makes sure it saves everyone's eardrums.

Notice that the `//` comments above the `for` loop essentially say the same thing, which is why leaving comments in the code is important. It explains why that code is there.

In case you've heard it before but are not sure what it means, the term “algorithm” in the comment refers to the audio processing logic that you'll be putting inside `processBlock`. An algorithm is simply a series of instructions for the computer to execute.

The `for` loop has a counter variable `i`. Often loop counters have short names; `i` is short for index. Inside the loop, the `buffer.clear()` function is used to set the contents of the `i`-th channel to silence.

How many times this `for` loop repeats depends on the kind of track the plug-in is loaded on and plug-in's bus configuration. It's also possible that this loop is completely skipped, when `totalNumInputChannels` is greater than or equal to `totalNumOutputChannels`. In fact, that's the usual case.

Note: This `for` loop does not have `{` and `}` braces. This is allowed in C++ when the loop only contains a single line of code. Personally, I think it's clearer to always use `{` and `}` as not having them can easily lead to bugs. (Mention the words "goto fail" to any software developer and they'll nod sagely.)

The second argument to `processBlock` is a `juce::MidiBuffer` object containing any MIDI messages that require handling. MIDI messages describe events such as "note on" and "note off" and are used by synthesizer plug-ins. We will not be doing anything with MIDI in this book, and so the `midiMessages` argument will not be used by the code inside `processBlock`.

Both Xcode and Visual Studio have been warning about this. Xcode says "Unused parameter 'midiMessages'", while Visual Studio says, "'midiMessages': unreferenced formal parameter". Let's tell the C++ compiler we don't intend to use this argument, so it will know this wasn't an unintended mistake.

Change the second argument of `processBlock` to the following:

```
void DelayAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer,
[[maybe_unused]] juce::MidiBuffer& midiMessages)
```

By adding the `[[maybe_unused]]` attribute to the argument, the C++ compiler will understand that we're not going to use the `juce::MidiBuffer` object inside the function. When you re-build the plug-in, the warning message should disappear.

Stereo only

To keep things simple, the code in the following chapters will assume that the Delay plug-in will always be used with stereo inputs. Later we'll also make it work with mono inputs.

Currently, however, the `isBusesLayoutSupported` function in `PluginProcessor.cpp` is telling the host that the Delay plug-in supports both stereo and mono buses.

Delete that entire function and replace it with the following:

```
bool DelayAudioProcessor::isBusesLayoutSupported(const BusesLayout& layouts) const
{
    return layouts.getMainOutputChannelSet() == juce::AudioChannelSet::stereo();
}
```

The host will call the plug-in's `isBusesLayoutSupported()` function to see what kind of buses it supports — mono, stereo, surround, and possibly other formats too. The function is supposed to return `true` for any bus layout the plug-in can handle.

Previously the functions you've worked with, `paint` and `processBlock`, had a return type of `void`. The return type of `isBusesLayoutSupported` is `bool`, short for **boolean**. A boolean is a data type that can only have two possible values: `true` or `false`.

The return value is how the function sends a reply back to the caller, in this case the host. The host asks, “Do you support a mono bus?” and `isBusesLayoutSupported` responds with `false` or “no”. Then the host calls `isBusesLayoutSupported` again and asks, “Do you support a stereo bus?” and now it responds with `true` or “yes”.

You can think of the function as a question and the return value as the answer.

The `==` operator is used to compare two values. We use `==` to check whether `layouts.getMainOutputChannelSet()`, which represents the requested channels for the output bus, equals `juce::AudioChannelSet::stereo()`. If so, the `==` operator evaluates to `true`.

For any other requested channel configuration such as mono, the `==` comparison evaluates to `false`.

The `return` statement exits the function and returns the result of the comparison.

Note: It's possible that if you load the plug-in in your DAW again that it will still offer a Mono option. Go to your DAW's plug-in manager and rescan the plug-in as the DAW may be caching the old bus configuration. Sometimes it takes a computer restart to clear this up.

Decibels

So far you have used a fixed gain amount of 0.5. It cuts the amplitude in half but how does that relate to the loudness of the sound?

A gain value of 0.5 is expressed in so-called linear units, which is convenient mathematically since we can simply multiply the sample amplitudes by it, but on the other hand 0.5 is not a musically meaningful value.

As a musician you might be more used to working with decibels. Human hearing is logarithmic in nature. This means we can hear differences between soft sounds very well, but the louder the sound becomes, the less sensitive our ears are. Specifying the gain in decibels or dB, which also follows a logarithmic scale, corresponds better to how we perceive loudness.

In `processBlock`, change the loop that applies the gain as follows:

```
// 1
float gainInDecibels = -6.0f;

// 2
float gain = juce::Decibels::decibelsToGain(gainInDecibels);

for (int channel = 0; channel < totalNumInputChannels; ++channel) {
    auto* channelData = buffer.getWritePointer(channel);

    for (int sample = 0; sample < buffer.getNumSamples(); ++sample) {
        channelData[sample] *= gain; // 3
    }
}
```

I have marked the parts that changed with comments. They are:

1. The gain amount is placed in a new variable named `gainInDecibels`. It has the value `-6.0f` for -6 dB.
2. To be able to use this gain, it needs to be converted into linear units. For that we use a handy JUCE function `juce::Decibels::decibelsToGain()`. The result of that is placed in the `gain` variable. For -6 dB the value of `gain` will be `0.5f` again (approximately).
3. Instead of the constant number `0.5f`, use the `gain` variable to multiply the sample values by.

That's all you need to do to use decibels. Give it a try. The plug-in should work as before: when it's engaged it will drop the output level by 6 dB, which is equivalent to halving the amplitude.

Try some other decibel values. Remember that 0 dB means no change in the level (gain is now 1.0), while dB values larger than zero mean the level is boosted.

If you're interested in the math formula used by `decibelsToGain()`, it is:

$$10^{\left(\frac{\text{gainInDecibels}}{20}\right)}$$

First, the `gainInDecibels` value is divided by 20 and then we raise 10 to the power of that. When we do $10^{(-6/20)}$ the answer is 0.5012, which is almost 0.5. That's why we usually say a gain of 0.5 equals -6 dB, even though it's only approximately the same. Likewise, a gain of +6 dB is $10^{(6/20)} = 1.995$ or 2.0 rounded off.

We can also go the other way around, from linear gain to decibels:

$$20 \log_{10}(\text{gain})$$

If we do $20 \log_{10}(0.5)$ we get -6.02 dB. Likewise, $20 \log_{10}(2) = 6.02$ dB. Note that the formula uses the logarithm with base 10. If you try this on a calculator and type in `log`, it may use the natural logarithm (or base e) instead, which gives the wrong result.

Anyway, if you're not into math you may forget these formulas right away. That's why JUCE has `decibelsToGain` and `gainToDecibels` helper functions.

Note: You've seen that the host program calls certain functions from the `DelayAudioProcessor` when it needs something, such as `isBusesLayoutSupported` to negotiate whether to use mono or stereo, `paint` to draw the editor, and `processBlock` to process audio. However, your own code can also call functions. That's what you're doing when you write `juce::Decibels::decibelsToGain(gainInDecibels)`. This function takes the decibel amount from the `gainInDecibels` variable as an argument and returns the converted value to your code, so that you can use it for something.

In the next chapter you will make the gain amount be changeable by the user, so that they can change the plug-in's output level by dragging a slider or turning a knob.

8: Plug-in parameters

A plug-in that always applies a fixed gain amount isn't very useful or interesting, so let's hook up the gain to a parameter. That way the user can control the gain by turning a knob in the plug-in's editor. It also allows the DAW to automate the gain.

Warning: Some of the code that follows may be a bit confusing. We're suddenly going to jump into the deep end of C++. I'll do my best to explain what's happening but I also don't expect you to understand everything right now. Just roll with it.

The plug-in parameters are owned by the `DelayAudioProcessor` object. To help us manage the parameters, we will use a helper object with a very long name, the `juce::AudioProcessorValueTreeState` or APVTS for short.

You will need to make one of these APVTS objects. This happens in **PluginProcessor.h**. Open that file and immediately below the line that says `private:`, add the following code:

```
juce::AudioProcessorValueTreeState apvts {  
    *this, nullptr, "Parameters", createParameterLayout()  
};  
  
juce::AudioProcessorValueTreeState::ParameterLayout createParameterLayout();
```

This adds two things to the `DelayAudioProcessor` object:

1. A variable named `apvts`. This is the `AudioProcessorValueTreeState` object.
2. A new function named `createParameterLayout`. This is a helper function that creates the plug-in parameter objects that will be added to the APVTS.

The code between the `{` and `}` braces are the arguments that get passed to the constructor of `AudioProcessorValueTreeState`. A constructor is a special function that makes room for the object in the computer's memory and then initializes it, meaning that it configures the object to behave how we want it.

The arguments are:

- `*this` – The first argument connects the APVTS to the `DelayAudioProcessor`. The C++ keyword `this` means “myself”, referring to the object that this code belongs to. The asterisk `*` as used here is not for multiplication. I don’t want to overload you with new concepts, so we’ll ignore this symbol for now.
- `nullptr` – The second argument allows you to provide an undo manager to the APVTS. Undo/redo is an advanced topic this book doesn’t cover, so the special keyword `nullptr` is used to tell the APVTS we don’t want an undo manager.
- “Parameters” – The third argument is a text string containing the name for the APVTS.
- `createParameterLayout()` – The last argument is a list of the parameters the plug-in has. Rather than write out the whole list here, we call our helper function `createParameterLayout` to make the parameter objects, and pass the list it returns into the fourth argument of the APVTS constructor.

Did that make any sense? The important thing to remember is that we’re adding a new object to `DelayAudioProcessor`, the APVTS, and the `createParameterLayout` function is used to create that APVTS.

Note: The `{ }` braces used here have a different meaning than before. This is one of the oddities of C++ that makes it so confusing to learn. Normally when you call a function, you’d put `()` around the arguments. However, for a constructor it is recommended that you use `{ }` otherwise the compiler can sometimes get confused into thinking you’re trying to call a normal function instead of a constructor. So here the `{ }` braces don’t create a scope like they did with the `for` loop or the function body — instead, they list the arguments that will be given to the `juce::AudioProcessorValueTreeState` constructor.

To implement the `createParameterLayout` function we’ll switch to **PluginProcessor.cpp**. Add the following code to that file. You can put it anywhere you want — as long as it’s not between the braces of another function — but my suggestion is to add it at the bottom of the file.

```

juce::AudioProcessorValueTreeState::ParameterLayout
    DelayAudioProcessor::createParameterLayout()
{
    juce::AudioProcessorValueTreeState::ParameterLayout layout;

    layout.add(std::make_unique<juce::AudioParameterFloat>(
        juce::ParameterID { "gain", 1 },
        "Output Gain",
        juce::NormalisableRange<float> { -12.0f, 12.0f },
        0.0f));

    return layout;
}

```

Oh my, lots of new stuff going on here. Let's look at each piece of this code in turn.

You can tell this is a function because it follows the usual form:



The shape of a function in C++

Normally you'd put the return type and function name on a single line but that won't fit in the book, so I had to wrap that across two lines.

You already know that `DelayAudioProcessor::createParameterLayout` means this is the `createParameterLayout` function that lives in the `DelayAudioProcessor` object. So far so good.

This function has no arguments so there is nothing between the () parentheses. You still need to write the () because that's how C++ knows this is a function and not something else.

The return type is `juce::AudioProcessorValueTreeState::ParameterLayout`. Some of these JUCE object names are quite verbose... Fortunately, most IDEs have auto-complete to save you from typing in those long names all the time.

The purpose of the `juce::AudioProcessorValueTreeState::ParameterLayout` object is to describe the plug-in parameters that should be added to the APVTS.

Inside `createParameterLayout` you're filling in this `ParameterLayout` object and then return it. If I simplify the code a little, it's clearer to see what happens:

```
juce::AudioProcessorValueTreeState::ParameterLayout
    DelayAudioProcessor::createParameterLayout()
{
    juce::AudioProcessorValueTreeState::ParameterLayout layout;
    layout.add( ... );
    return layout;
}
```

The line `layout.add(...);` will add the new parameter object to the `ParameterLayout`. Right now, the plug-in only has one parameter, but over the course of this book you'll add more `layout.add()` statements to this function.

Let's look in detail at the full `layout.add()` line:

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    juce::ParameterID { "gain", 1 },
    "Output Gain",
    juce::NormalisableRange<float> { -12.0f, 12.0f },
    0.0f));
```

The `std::make_unique<juce::AudioParameterFloat>(...)` notation means that we're creating a new `juce::AudioParameterFloat` object. This is the object that describes the actual plug-in parameter and is what you'll be using in the actual audio processing code.

The `std::make_unique` bit is there to put the `juce::AudioParameterFloat` into a `std::unique_ptr` object. This is a detail that you can ignore for now. Note that `std::` refers to the C++ standard library, so the `make_unique` function is a part of C++ itself, not JUCE.

The constructor for the `AudioParameterFloat` object takes four arguments:

1. A unique identifier in the form of a `juce::ParameterID` object. Here we give it the identifier "gain". This is a string so it needs to be between double quotes. The number 1 is a version hint, which is needed for Audio Unit plug-ins.
2. The name of the parameter. This should be a human-readable string. Here you're using "Output Gain".

3. The range of the parameter, as a `juce::NormalisableRange` object. The `<float>` part means that the range is expressed using `float` values. `-12.0f` is the minimum value for this parameter, `12.0f` the maximum value.
4. The final argument is the default value of the parameter, set to `0.0f` since we want the output gain to be 0 dB initially.

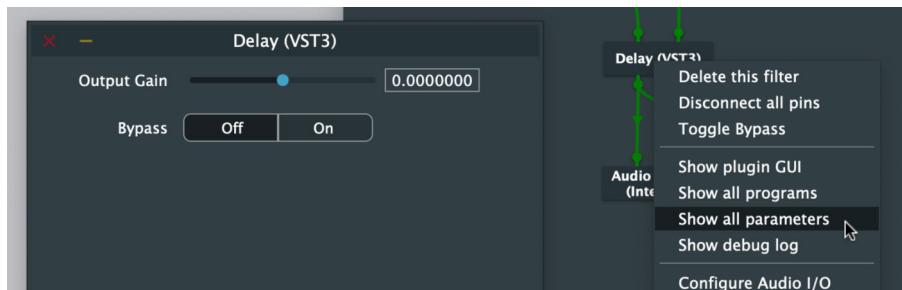
Note that `juce::ParameterID` and `juce::NormalisableRange` are objects, so to create them you invoked their constructors using the `{ arguments }` syntax.

Pew, this is the most complex code we've encountered so far and it may take some deciphering if you're new to C++. To summarize, we've constructed a new `juce::AudioParameterFloat` object that describes the plug-in parameter. This gets added to a `juce::AudioProcessorValueTreeState::ParameterLayout` object, which is then used to create the APVTS.

Build the plug-in and run it in `AudioPluginHost`. You should see... the exact same thing as before. You didn't create a user interface for this parameter yet, so it doesn't show up in the plug-in's editor window.

Tip: If the compiler gives an error in a `memory` or `unique_ptr.h` file when you try to build the plug-in, you probably made a mistake in the parameter definition, typically by providing too few or too many arguments. The error message isn't very clear, but the error happens because the C++ compiler doesn't understand how to construct the parameter object using `std::make_unique` if the number or the type of arguments does not match the constructor for `AudioParameterFloat`. Also make sure you have the correct number of closing parentheses `)`, since it's easy to forget one.

To reveal the plug-in parameters, right-click the **Delay** block and choose **Show all parameters**:

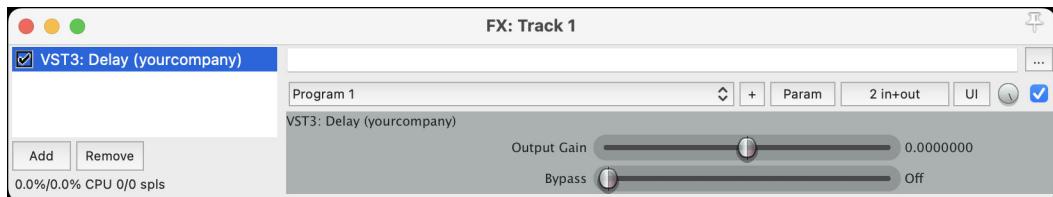


Showing the plug-in's parameters

This pops up a new window that has a slider for the **Output Gain** parameter. Try dragging it and you'll see that it goes from +12 down to -12, with the initial value at 0, just like we said it should. We will interpret these to be decibel values, even though the `juce::NormalisableRange` object doesn't care about the particular units.

There is also a Bypass button, which works the same way as the Toggle Bypass menu option you've used before.

If you're using a DAW to test the plug-in, most DAWs will have a way to view the plug-in's parameters. For example, in REAPER you can click the UI button:



The gain and bypass parameters in REAPER

However, the parameter doesn't work yet. Try moving the slider when audio is playing and the output level stays the same. That makes sense: You have only told the plug-in that this parameter exists, but didn't change the audio processing code in `processBlock` yet to do something with this parameter.

Update the line for `gainInDecibels` in `processBlock` to the following:

```
float gainInDecibels = apvts.getRawParameterValue("gain")->load();
```

This no longer uses the hardcoded value of `-6.0f` but reads the value from the gain parameter instead. Run the plug-in again and the **Output Gain** parameter should work. Nice!

Remember that a gain larger than 0 dB will increase the level of the sound. Putting the parameter on +12 dB is probably not a good idea if the audio you're playing back is already kind of loud.

When the sample values in the audio signal go outside the range from `-1.0` to `+1.0`, the computer's audio output may start to clip and the sound will crackle. It's also possible the computer will simply drive your speakers harder than it should, which is not necessarily good for your speakers.

If you do want to test large values for **Output Gain**, my suggestion is to unhook the connections to the **Audio Output** block in `AudioPluginHost`. If you're testing in a DAW, mute the master track. This way no sound gets sent to your speakers but you can still see what happens to the signal using an oscilloscope plug-in.

Improving the parameters

Even though our new parameter works just fine, there are a few things we can improve. We've used the string "gain" to refer to the parameter but employing strings as identifiers this way is error prone. For example, if you had mistyped and wrote `getRawParameterValue("gian")` the plug-in would crash, because the APVTS does not have a parameter with the identifier "gian".

It's better to define the parameter ID as a constant and from then on refer to it using that constant's name. This way the C++ compiler will warn you if you make a typo. Here's how to do that.

Go to **PluginProcessor.h** and add the following line at the top, below the `#include` line:

```
const juce::ParameterID gainParamID { "gain", 1 };
```

This declares a **constant** named `gainParamID` that is a `juce::ParameterID` object. As before this uses the identifier "gain" and the version hint 1. The idea is that this is the only place you'll ever type the string "gain". Everywhere else you'll use `gainParamID` instead.

Back in **PluginProcessor.cpp**, in `createParameterLayout` change the lines that add the parameter to the following:

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    gainParamID,           // this changed!
    "Output Gain",
    juce::NormalisableRange<float> { -12.0f, 12.0f },
    0.0f));
```

Everything's still the same except now it uses the `gainParamID` constant instead of constructing a `juce::ParameterID`. This is possible because `gainParamID` is in fact a `juce::ParameterID` object.

In `processBlock`, update the code for `gainInDecibels` to the following. The difference is that you replaced the string "gain" by `gainParamID.getParamID()`.

```
float gainInDecibels = apvts.getRawParameterValue(gainParamID.getParamID())->load();
```

In the end it does the exact same thing, but it tells the compiler exactly what our intentions are and then the compiler can check that we're not messing things up. Very helpful!

Build and verify that the plug-in still works.

We can do even better. Using `apvts.getRawParameterValue()` is rather slow. We're just calling it once per block, but that easily can be 100s or 1000s of times per second. There is a faster and more convenient way to read the parameter values.

Go to **PluginProcessor.h** and add the following line, again in the `private:` section of the class.

```
juce::AudioParameterFloat* gainParam;
```

This creates a `juce::AudioParameterFloat` object, named `gainParam`. Well, actually, the * means this is a **pointer** to such an object.

`gainParam` is not itself a `juce::AudioParameterFloat` object but will refer to one of the `juce::AudioParameterFloat` objects from the APVTS. Since the APVTS owns and manages the plug-in parameters, we'll merely use `gainParam` to access one of those parameters.

If you're new to the idea of pointers, don't worry, it's one of the most difficult programming concepts to get your head around. For now, all you need to understand is that we will be able to directly access the **Output Gain** parameter through the `gainParam` object.

We do still need to make `gainParam` point to the actual parameter. This is done in the constructor of `DelayAudioProcessor`. Switch to **PluginProcessor.cpp** and scroll to the top. There you'll find the following, rather ugly-looking code.

```

DelayAudioProcessor::DelayAudioProcessor()
#ifndef JucePlugin_PREFERREDCHANNELCONFIGURATIONS
    : AudioProcessor (BusesProperties()
                      #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                          .withInput ("Input", juce::AudioChannelSet::stereo(), true)
                      #endif
                      .withOutput ("Output", juce::AudioChannelSet::stereo(), true)
                      #endif
                  )
#endif
{
}

```

This is the implementation of the `DelayAudioProcessor` constructor. This constructor is executed when the host loads the plug-in.

The `#ifndef` and `#if` and `#endif` stuff makes this really hard to read. Those lines are there so that Projucer can use a single template project to cover all the different possible plug-in types. Let's clean up this mess so it's easier to understand what's going on here.

Replace the above code with the following:

```

DelayAudioProcessor::DelayAudioProcessor() :
    AudioProcessor(
        BusesProperties()
            .withInput("Input", juce::AudioChannelSet::stereo(), true)
            .withOutput("Output", juce::AudioChannelSet::stereo(), true)
    )
{
}

```

That's a little cleaner! So, what does this do?

A constructor is a special function that always has the same name as the object, which is why this is called `DelayAudioProcessor::DelayAudioProcessor`. A constructor does not have a return type. It can have arguments in between the `()` parentheses, but this constructor has none.

Since `DelayAudioProcessor` is based on `juce::AudioProcessor`, we need to call the constructor of that class first. In C++ that's done by writing a colon followed by `AudioProcessor` and the arguments for its constructor. The `juce::` namespace isn't necessary here although it would be OK to write it.

The argument for the `juce::AudioProcessor` constructor is a `BusesProperties` object that says the plug-in has a stereo audio input and a stereo audio output.

Now that we understand a bit better what this constructor does, add the following lines to it. The constructor works like a regular function, so the code goes between the curly { } braces.

```
auto* param = apvts.getParameter(gainParamID.getParamID());
gainParam = dynamic_cast<juce::AudioParameterFloat*>(param);
```

I'm sure this looks scary but we can pick it apart to see what it does:

- `apvts.getParameter(gainParamID.getParamID())` – This looks up the parameter object in the APVTS.
- The parameter is temporarily stored in a **local variable** named `param`. The type of this variable is `juce::RangedAudioParameter*` but I don't really care about that, nor did I feel like typing that all in, so I used the shorthand `auto*`, which means “some kind of pointer” (recall that * is used for pointers).
- `dynamic_cast<juce::AudioParameterFloat*>` – A **cast** is a way to convert between different data types. This tells the C++ compiler that the object from the `param` variable is actually a pointer to a `juce::AudioParameterFloat` object.
- The result of the `dynamic_cast` gets stored in the `gainParam` variable.

Made no sense at all? That's fine. What matters is that from now on we can use `gainParam` to directly read the current value for the parameter.

Let's do that in `processBlock`. Replace the existing line for `gainInDecibels` with:

```
float gainInDecibels = gainParam->get();
```

Since `gainParam` is a pointer, you must use the -> arrow notation to call the parameter object's `get()` function.

Previously when you called a function, you used the notation `object.function()`, for example `buffer.getNumSamples()`. Here, `buffer` is not a pointer. If `buffer` had been a pointer, it would have been necessary to write `buffer->getNumSamples()`.

This is the difference between . and -> in C++. The language uses different syntax to make it clear whether you're using a pointer or not.

Note: This pointer approach is a lot faster than doing `apvts.getRawParameterValue`. The issue with `getRawParameterValue` is that it looks up the parameter by name in the APVTS, which may involve doing many string comparisons, especially when you have lots of parameters. By keeping track of the `AudioParameterFloat*` object, we can immediately access the parameter through this pointer, without looking it up over and over again. This is a type of “caching”, memorizing things so they don’t have to be computed more than once.

Try it out, build and run the plug-in and everything should still work.

Saving the parameter values

Try the following experiment: Open the plug-in with `AudioPluginHost` or a DAW, change the **Output Gain** parameter from 0 dB to -12 dB, save the session, then quit the host. Now restart `AudioPluginHost` or the DAW, load the session back in, and view the plug-in’s parameters. What does **Output Gain** say?

You would expect the plug-in to remember the value from when you saved it, which was -12 dB, but it’s back at the default value of 0 dB. That’s a problem because it means users of your plug-in will lose their carefully designed sounds once they quit the DAW.

Fortunately, there is a way for the plug-in to save its current state and load it back the next time the plug-in is used. You want to keep those users happy!

The audio processor has two functions that are used to serialize (save) and deserialize (load) the plug-in’s state: `getStateInformation` and `setStateInformation`.

In `PluginProcessor.cpp`, find these functions and fill them in like so:

```
void DelayAudioProcessor::getStateInformation (juce::MemoryBlock& destData)
{
    copyXmlToBinary(*apvts.copyState().createXml(), destData);
}

void DelayAudioProcessor::setStateInformation (const void* data, int sizeInBytes)
{
    std::unique_ptr<juce::XmlElement> xml(XmlFromBinary(data, sizeInBytes));
    if (xml.get() != nullptr && xml->hasTagName(apvts.state.getType())) {
        apvts.replaceState(juce::ValueTree::fromXml(*xml));
    }
}
```

That's all you have to do. Run the experiment again, and now you'll find that the plug-in does remember its settings. Nice!

I will refrain from describing exactly how the code in these two functions works. It only matters that you have a general understanding of what happens here.

The `getStateInformation` function is used to save the plug-in's current state. You are supposed to serialize anything that is important into the given `juce::MemoryBlock`. Serializing is a fancy word for putting data into a format that can be stored inside a file. The serialization format we'll use is XML, which is a common way to store structured data in files.

For our plug-in, we want to save the current values of all the parameters from the APVTS. One of the convenient features of the APVTS is that it can serialize its contents to an XML document using the `createXml` function. And then `copyXmlToBinary` is used to put the XML into the memory block.

In `setStateInformation` it goes the other way around. You are provided with a block of binary data. First, this calls `getXmlFromBinary` to parse the binary data into an XML document. Then it reads the parameter values from this XML, and finally it calls `apvts.replaceState()` to update all the parameters to the new values.

In case you're curious about the XML that is generated, add the following line to the code inside `getStateInformation` and run `AudioPluginHost` gain.

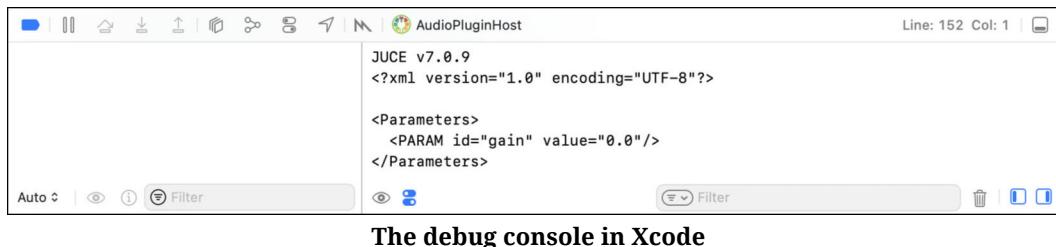
```
DBG(apvts.copyState().toXmlString());
```

The `DBG` command stands for “debug print” and it writes the value of its argument to the IDE's debug console. Here you're using it to write the contents of the APVTS as an XML text string. When you save the session in `AudioPluginHost`, the `DBG` statement will write something like the following to the debug console.

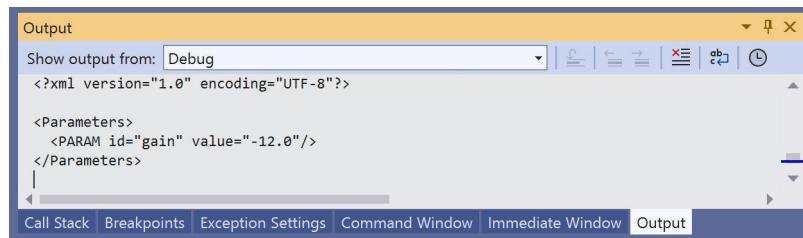
```
<?xml version="1.0" encoding="UTF-8"?>  
<Parameters>  
  <PARAM id="gain" value="-12.0"/>  
</Parameters>
```

This is how the plug-in's state gets saved to an XML document.

In Xcode you can find the debug console at the bottom of the window. If you don't see this, click the small controls next to the trashcan to reveal it.



In Visual Studio, locate the **Output** tab at the bottom of the debugger window.



The debug console will print lots of other messages too but you can ignore those. Some look like error messages but they are mostly from the operating system, not our plug-in. The **DBG** command is handy for writing informational messages to the IDE's debug output pane to inspect what the plug-in is doing. We'll be using it from time to time in this book.

You may be wondering exactly where the XML gets saved. This depends on the host application; it will typically end up in the project file. In the case of AudioPluginHost, the XML is saved in the **.filtergraph** file, which itself is XML. Open the filtergraph file in a text editor, and you'll see something like this:

```

<PLUGIN name="Delay" format="VST3" category="Fx"
        manufacturer="yourcompany" ... />
<STATE>Ea0cVZtMEcgQWY9vSRC8Vav8lak4Fc9DCL3HiKV0jZLcFUq3hKt3xSqX1UgM1ZI...</STATE>
</PLUGIN>

```

The **<STATE>** field contains the plug-in's serialized data. Note that this doesn't show the XML from before but some sort of encoded version. That's because the serialization format does not need to be XML. JUCE doesn't care and neither do host applications. The only thing that matters is that your plug-in can serialize and deserialize its state to a blob of binary data. The main advantage of using XML is that APVTS has built-in support for it already.

Note: You can remove the DBG statement again from `getStateInformation`. Or you can keep it but “comment it out” by adding `//` in front of the line to turn it into a comment that is ignored by the C++ compiler. That way you can easily enable it later by “uncommenting it” or removing the `//` characters.

Creating a Parameters class

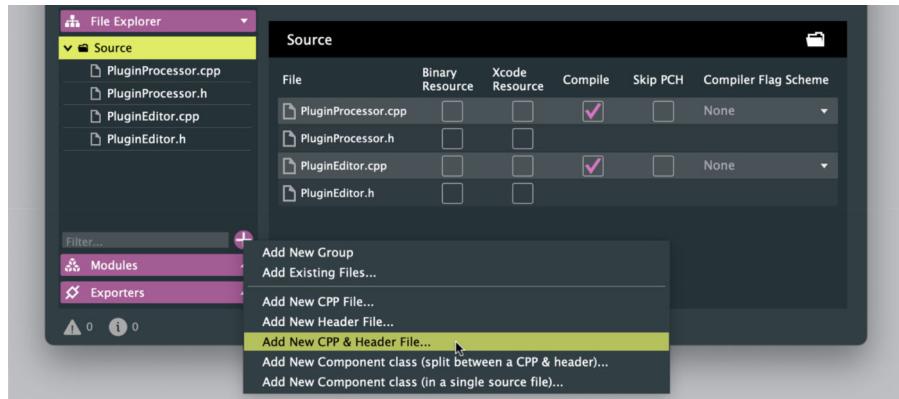
Right now, we have only one parameter but by the time the plug-in is finished it will have several more, ten in total.

Some developers will add all the parameters to the audio processor object like we’ve just done, but it’s cleaner to create a new `Parameters` object for this. That also gives me a good excuse to explain how to make your own objects.

It’s standard practice to put new objects into their own source files, so let’s add some new files to the project.

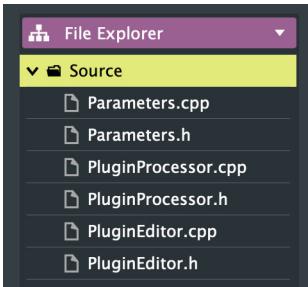
Note: Your first instinct might be to add the new source files straight in the IDE, but that’s not how it works with JUCE. You’ll have to use Projucer for this kind of thing. Projucer manages the project and that includes which source files it has.

Go to **Projucer** and open the **Delay.jucer** project. In the **File Explorer** pane select the **Source** folder, so that the new files will be added inside this folder. Click on the round + button at the bottom of the pane and choose **Add New CPP & Header File...**



Adding a new source file in Projucer

A system file dialog pops up, asking you to enter the filename. Name it **Parameters** and click **Save**. After you're done, two new files have been added in the File Explorer, **Parameters.cpp** and **Parameters.h**.

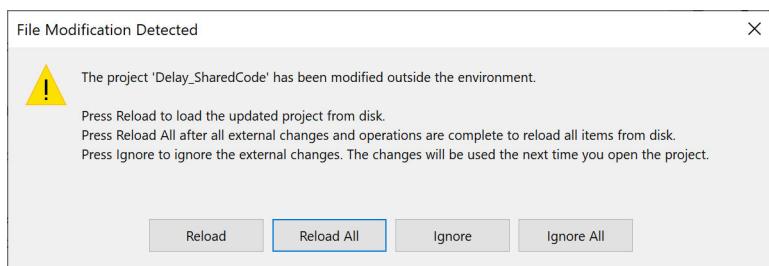


The new files appear in the File Explorer

Next, you will need to export the project from Projucer to your IDE again. From now on, when making changes in Projucer, do not press the round button at the top of the window to export the project. Instead, use **File > Save Project** or press **Ctrl+P** (Windows) or **Command+P** (Mac).

This is especially important for Windows users! When you press the round button, Projucer will overwrite all the Visual Studio project files and opens the project in a new IDE window, rather than reloading the existing project. You'll also lose the settings that automatically start AudioPluginHost. Using **Save Project** prevents this from happening.

Choose **File > Save Project** from the Projucer menu bar. The IDE will reload and show the two new files in the Project Navigator (Xcode) or Solution Explorer (Visual Studio). In Visual Studio you'll have to click **Reload All** to update the project.



Visual Studio warns that the project has changed

Open **Parameters.h** and replace its contents with the following:

```
// 1
#pragma once

// 2
#include <JuceHeader.h>

// 3
class Parameters
{
public:
    // 4
    Parameters(juce::AudioProcessorValueTreeState& apvts);

    // 5
    static juce::AudioProcessorValueTreeState::ParameterLayout
        createParameterLayout();

    // 6
    juce::AudioParameterFloat* gainParam;
};
```

This defines a new class named **Parameters** — our own first class! — and its purpose is to make it easier to deal with all the plug-in parameters you'll be adding later. Recall that a **class** describes what an object is and what functions it has.

There are some new things in this code, marked with comments. From top-to-bottom:

1. The `#pragma once` line tells the C++ compiler to only read the `Parameters.h` file once, even if it's `#include`'d multiple times. Don't sweat this for now.
2. Since this file refers to JUCE objects, we include the `JuceHeader.h` file so that the C++ compiler knows what `juce::` means.
3. Define a new class named **Parameters**.
4. The constructor that is used to initialize a new **Parameters** object. It has one argument, a reference to the APVTS.
5. You've seen `createParameterLayout` before, it creates the plug-in parameters when the APVTS is initialized. This function currently lives in `DelayAudioProcessor` but you'll put it into the **Parameters** object shortly.
6. Similarly, the `gainParam` pointer will now belong to the **Parameters** object.

Next, you’re going to move some code from `DelayAudioProcessor` into the new `Parameters` class. If you think it’s silly that I first made you write code in one file only to have you cut-and-paste it into another file, then welcome to programming!

This process is called **refactoring** and it’s a normal part of software development. It’s very common to start out writing code one way and realizing halfway through you should rearrange things. Most IDEs even have built-in tools for refactoring, but we’ll do it by hand.

Open `PluginProcessor.h` and remove the following lines. You don’t need these anymore as this function and variable have moved to the `Parameters` object.

```
juce::AudioProcessorValueTreeState::ParameterLayout createParameterLayout();
juce::AudioParameterFloat* gainParam;
```

Still in `PluginProcessor.h`, update the code that declares the `APVTS` object to read:

```
juce::AudioProcessorValueTreeState apvts {
    *this, nullptr, "Parameters", Parameters::createParameterLayout()
};
```

The difference is that now you’re calling `Parameters::createParameterLayout()` instead of just `createParameterLayout()`.

Directly below, add the following line:

```
Parameters params;
```

This tells the `DelayAudioProcessor` that it has a `Parameters` object named `params`. Or in C++ speak, it has an instance of the `Parameters` class.

As soon as you make these changes, the IDE will display an error message. Xcode says “Unknown type name ‘Parameters’”. Visual Studio puts a red squiggly line under `Parameters` and says “identifier ‘Parameters’ is undefined”. This happens because `DelayAudioProcessor` does not know yet what `Parameters` is.

Even though you added the `Parameters.cpp` and `.h` files to the project, that doesn’t mean all the other source files in the project automatically know about them. To solve this, you need to **include** the header file.

Scroll to the top of `DelayAudioProcessor` and add the following line:

```
#include "Parameters.h"
```

It's best if you put this immediately below the other `#include` line. The error message should disappear. This is why C++ source files usually have one or more `#include` statements at the top, to import any names from other files in the project that the code needs to know about.

You're not done yet! Because you removed a bunch of stuff from `PluginProcessor.h`, the code in `PluginProcessor.cpp` is broken now. Let's finish making the changes.

Cut the following line from `PluginProcessor.h` and paste it into `Parameters.h`, below the `#include` line:

```
const juce::ParameterID gainParamID { "gain", 1 };
```

Open `Parameters.cpp`. Replace its contents with the following:

```
// 1
#include "Parameters.h"

// 2
Parameters::Parameters(juce::AudioProcessorValueTreeState& apvts)
{
    // 3
    auto* param = apvts.getParameter(gainParamID.getParamID());
    gainParam = dynamic_cast<juce::AudioParameterFloat*>(param);
}
```

What this does:

1. First this includes `Parameters.h` so that the C++ compiler knows what object we're talking about.
2. Here you implement the constructor for the `Parameters` object.
3. The constructor does the exact same thing that the `DelayAudioProcessor` constructor does. In fact, you can copy this code from `DelayAudioProcessor`.

Back in **PluginProcessor.cpp**, update the constructor with the following:

```
DelayAudioProcessor::DelayAudioProcessor() :
    AudioProcessor(
        BusesProperties()
            .withInput("Input", juce::AudioChannelSet::stereo(), true)
            .withOutput("Output", juce::AudioChannelSet::stereo(), true)
        ),
        // watch the comma!
    params(apvts) // this is new
{
    // do nothing
}
```

There are three changes to make here:

- There no longer is any code in between the { } brackets. The code that was here has been moved to the **Parameters** constructor. If a function has no code, I like the write the comment // do nothing so I know that it's intentional.
- You've added `params(apvts)` to the `DelayAudioProcessor` class's **member initializer list**.
- Make sure there is a comma between `AudioProcessor(...)` and `params(apvts)`.

The purpose of a constructor is to prepare the object for use, which involves giving all its variables reasonable values (known as initializing). If the object is based on another object, known as inheriting from that object, then it should also invoke the constructor of the base object.

In C++ this is done using the member initializer list, which is everything following the : behind the constructor's name, up to the { brace.

You've already seen that the member initializer list calls `AudioProcessor(...)`, the constructor for the `juce::AudioProcessor` object that `DelayAudioProcessor` is based on, and passes it a `BusesProperties` object.

The new thing is that we also need to initialize the `Parameters` object here, giving it a reference to the APVTS. That is done by writing `params(apvts)`.

Almost there! In **PluginProcessor.cpp**, scroll to where the `createParameterLayout` function is located. Cut out this entire function and paste it into **Parameters.cpp** (put it at the end of that file).

As this function no longer belongs to `DelayAudioProcessor` but is now part of the class `Parameters`, change `DelayAudioProcessor::` to `Parameters::`, like so:

```
juce::AudioProcessorValueTreeState::ParameterLayout  
    Parameters::createParameterLayout()  
{  
    // ...existing code...  
}
```

The final thing to change is in **PluginProcessor.cpp** in `processBlock`:

```
float gainInDecibels = params.gainParam->get();
```

The `gainParam` variable is now part of the `Parameters` object, so you'll have to access it through `params.gainParam`. Since `params` is not a pointer, you use `.` to access its member variable. However, `gainParam` itself still is a pointer and therefore we use the `->` arrow to call `get()`.

Verify that you can build and run the plug-in again. All right, refactoring complete! You've successfully extracted the parameter stuff from the audio processor into a class of its own.

Improving the class

What was the point of this exercise? Well, we're not quite done yet. I want to move as much logic for dealing with the parameters, including the decibel-to-linear gain calculation, out of `DelayAudioProcessor` and into `Parameters`.

This way the audio processor only has to worry about the actual DSP code and not where the parameter values come from.

Change the code in **Parameters.h** to the following:

```
#pragma once

#include <JuceHeader.h>

class Parameters
{
public:
    Parameters(juce::AudioProcessorValueTreeState& apvts);

    static juce::AudioProcessorValueTreeState::ParameterLayout
        createParameterLayout();
    // 1
    void update() noexcept;

    // 2
    float gain;

    // 3
private:
    juce::AudioParameterFloat* gainParam;
};
```

The changed parts are:

1. You added a new function, `update()`. This where we'll read the current **Output Gain** parameter setting and put it into the `gain` variable.
2. You added a variable `gain`. This is a `float` that holds the current gain amount expressed in linear units, not decibels.
3. The `gainParam` variable has been moved into the `private:` section. Any variables or functions that are declared as `private` will not be usable by other objects.

Currently the code in `processBlock` does `params.gainParam->get()` but that will no longer compile. Try it. The C++ compiler will say that `gainParam` is now a private member of `Parameters`, meaning it can only be used by the functions from `Parameters` itself — such as the new `update` function — but not by anyone else.

This principle is known as **information hiding**. The `DelayAudioProcessor` shouldn't have to know that the gain amount comes from a `juce::AudioParameterFloat` that uses decibels. Instead, the `DelayAudioProcessor` can simply read the value from the `gain` variable whenever it needs to know the amount of output gain.

Add the update function to **Parameters.cpp** at the bottom of the file:

```
void Parameters::update() noexcept
{
    gain = juce::Decibels::decibelsToGain(gainParam->get());
}
```

This reads the current value from the **Output Gain** parameter using `gainParam->get()` and converts it from decibels to a linear gain value. Very similar to what you did previously in `processBlock`. Later you'll add more code to this function as new plug-in parameters are added.

Note: The `noexcept` behind the function name means that we're guaranteeing this function won't throw an exception. In C++ and many other programming languages, **exceptions** are used to handle certain types of errors that may happen during the operation of the program. By telling the C++ compiler we don't want to use exceptions in this function, it can generate more optimal code. Audio code should not throw exceptions, so it's a good habit to mark all your audio processing functions `noexcept` to get that extra speed boost.

To use this new update function, replace the following lines in `processBlock` in **Plug-inProcessor.cpp**,

```
float gainInDecibels = params.gainParam->get();
float gain = juce::Decibels::decibelsToGain(gainInDecibels);
```

with:

```
params.update();
float gain = params.gain;
```

It does the exact same thing as before, but now the logic for reading the **Output Gain** parameter and converting from decibels lives in the `Parameters` object, in the function `params.update()`. The `DelayAudioProcessor` can simply read `params.gain` to get the current output gain value without wondering where it came from.

Even though the code appears to work fine, there's a subtle bug. The `gain` variable in the `Parameters` object is not initialized inside that object's constructor. As a result, the initial value of `gain` could be anything.

In certain programming languages, variables are initially set to zero but the C++ compiler does not guarantee this. Reading `params.gain` will return an arbitrary number until the first time `params.update()` is called.

Visual Studio indeed warns that the variable `Parameters::gain` is uninitialized. It's not going to cause an issue in our code, because we always call `params.update()` before using `params.gain`. Just in case, to prevent ourselves from making a mistake here later, it's best to assign a "safe" initial value to all our variables.

This could be done inside the constructor or in the member initializer list, but we will simply change the declaration of `float gain` in **Parameters.h** to give it the initial value of zero:

```
float gain = 0.0f;
```

One final thing: The `dynamic_cast<>` in the `Parameters` constructor is fine but as you start adding more parameters, you'll have a whole list of these statements and the code will get unwieldy. Let's add a helper function that simplifies this.

In **Parameters.cpp** add the following immediately above the constructor.

```
template<typename T>
static void castParameter(juce::AudioProcessorValueTreeState& apvts,
                           const juce::ParameterID& id, T& destination)
{
    destination = dynamic_cast<T>(apvts.getParameter(id.getParamID()));
    jassert(destination); // parameter does not exist or wrong type
}
```

This code needs to go above the constructor because we'll call it from the constructor. In C++ you can't call a function unless it has been declared, and things are declared from top-to-bottom in the source file.

Change the constructor to this:

```
Parameters::Parameters(juce::AudioProcessorValueTreeState& apvts)
{
    castParameter(apvts, gainParamID, gainParam);
}
```

The `dynamic_cast` is now done in the `castParameter` function. It's a little cleaner this way, and makes the code more readable.

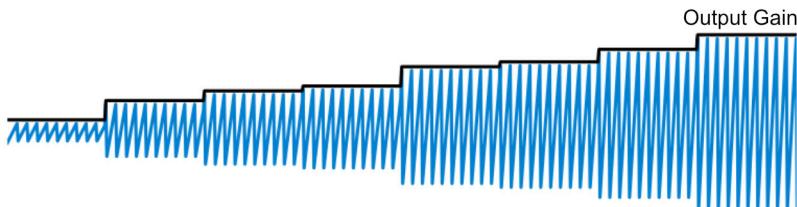
Note: The `template<typename T>` stuff allows the `castParameter` function to also handle plug-in parameters that are not `AudioParameterFloat` but `AudioParameterBool` or `AudioParameterChoice`. This is a technique known as generic programming. Templates are an advanced feature of C++. You don't need to understand how this works right now, but the idea is that it reduces the amount of code we have to write. Also note that `castParameter` is a so-called free function, it does not belong to any object in particular.

Build and run to make sure everything still works. Nice, our `Parameters` class is starting to shape up — but there's even more we can do to improve it!

Zipper noise and parameter smoothing

One thing you might have noticed when moving the **Output Gain** slider is a tiny amount of crackling in the sound, especially when dragging it quickly back and forth. If you haven't heard this phenomenon yet, try it with an audio file that is relatively smooth such as a synth lead, while wearing headphones. This crackling is known as **zipper noise** and is very undesirable.

These glitches occur because moving the slider happens at a relatively low speed compared to the sampling rate of the audio. Even if you drag the slider up and down like a maniac, `processBlock` will render many samples between each parameter update. If we plot the parameter update over time versus the audio signal, it will look something like this:



The parameter (thick black line) moves in steps, which makes the audio jump too

Those steps are what creates the zipper noise. The amplitude level of the audio signal makes the same kind of discrete jumps whenever the parameter is changed. Those jumps introduce extra frequency content in the sound, which is audible as short clicks.

To eliminate this zipper noise, we must avoid the jumps between successive parameter values. For example, if **Output Gain** changes from 0 dB to -6 dB, we don't immediately make this change but gradually bring down the level: -0.1 dB, -0.2 dB, -0.3 dB, -0.4 dB, and so on until the target of -6 dB is reached.

This is known as **parameter smoothing** and JUCE has a `juce::LinearSmoothedValue` object that is ideal for this job.

In **Parameters.h**, add the following line in the `private:` section of the class:

```
juce::LinearSmoothedValue<float> gainSmoothen;
```

This creates a new `juce::LinearSmoothedValue` object for a `float` value, with the name `gainSmoothen`.

Also add three new functions to the `public:` section:

```
void prepareToPlay(double sampleRate) noexcept;
void reset() noexcept;
void smoothen() noexcept;
```

Just to make sure we're all on the same page, the full **Parameters.h** should look like this now. The order of the functions and variables does not matter, although traditionally the constructor goes first, then the other functions, and finally the variables.

```
#pragma once

#include <JuceHeader.h>

const juce::ParameterID gainParamID { "gain", 1 };

class Parameters
{
public:
    Parameters(juce::AudioProcessorValueTreeState& apvts);

    static juce::AudioProcessorValueTreeState::ParameterLayout
        createParameterLayout();

    void prepareToPlay(double sampleRate) noexcept;
    void reset() noexcept;
    void update() noexcept;
    void smoothen() noexcept;

    float gain = 0.0f;
```

```
private:
juce::AudioParameterFloat* gainParam;
juce::LinearSmoothedValue<float> gainSmoother;
};
```

In **Parameters.cpp**, add the `prepareToPlay` function:

```
void Parameters::prepareToPlay(double sampleRate) noexcept
{
    double duration = 0.02;
    gainSmoother.reset(sampleRate, duration);
}
```

The `juce::LinearSmoothedValue` object needs to know what the current sample rate is, and how long it should take to transition from the previous parameter value to the new one. This duration is specified in seconds, where 0.02 seconds means 20 milliseconds.

The smoothing time should not be too short or you'll still hear zipper noise, but not too long or you'll literally hear the sound fade in or out. The smoothing should be long enough to get rid of the zipper noise but short enough so you won't consciously hear it. Twenty milliseconds at a sampling rate of 48 kHz is 960 samples (calculation: $48000 \times 0.02 = 960$) and is a good compromise.

Add the implementation for the `reset` function:

```
void Parameters::reset() noexcept
{
    gain = 0.0f;

    gainSmoother.setCurrentAndTargetValue(
        juce::Decibels::decibelsToGain(gainParam->get()));
}
```

This sets the value of `gain` to `0.0f`, but why this is needed? Well, when you changed the declaration of the `gain` variable to be `float gain = 0.0f` to initialize it, this only happens once in the plug-in's lifetime, when the plug-in is first loaded.

However, hosts may start and stop audio many times, and to be safe we should set the value of `gain` back to zero every time this happens. That's why we're using this `reset` function to re-initialize the `Parameters` object. It also loads the current value of the parameter into the `gainSmoother` object.

Change the update function to the following:

```
void Parameters::update() noexcept
{
    float gainInDecibels = gainParam->get();
    float newGain = juce::Decibels::decibelsToGain(gainInDecibels);
    gainSmoothen.setTargetValue(newGain);
}
```

This no longer directly changes the gain variable, but tells the smoother what the new parameter value is.

1. First, gainParam->get() reads the current value from the parameter.
2. Then juce::Decibels::decibelsToGain converts from decibels to linear units.
3. Finally, gainSmoothen.setTargetValue tells the smoother about the new value. If that differs from the previous value, the smoother will get to work.

Thanks to the decibel conversion, what gets smoothed isn't the gain in decibels but in linear units. This is more efficient because we only have to do decibelsToGain conversion just once per block, not once per sample.

Assuming the above code makes sense to you, let's rewrite it by combining these three operations into a single line, like so:

```
void Parameters::update() noexcept
{
    gainSmoothen.setTargetValue(juce::Decibels::decibelsToGain(gainParam->get()));
}
```

This is less typing and it does exactly the same thing. You read a line like this from the inside-out: First it does gainParam->get(), then it does decibelsToGain, and finally it does setTargetValue.

Note: In reset you used setCurrentAndTargetValue but here it is setTargetValue. The difference is that in reset we want the smoother to immediately jump to the parameter value but in update we only want to set the value to ramp towards.

Finally, add the last new function, smoothen().

```
void Parameters::smoothen() noexcept
{
    gain = gainSmoothen.getNextValue();
}
```

This reads the currently smoothed value and puts it into the `gain` variable. You will call this function on every new sample timestep, so that `gain` is always moving towards the latest setting of the parameter.

To start using this smoothed parameter, go to **PluginProcessor.cpp** and find the `prepareToPlay()` function. This function is called just before the host wants to start playing audio. This is a good place to reset the state of our objects.

```
void DelayAudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
{
    params.prepareToPlay(sampleRate);
    params.reset();
}
```

This first calls `params.prepareToPlay()` to let it know about the sample rate, and then resets the `Parameters` object.

You will also have to rewrite the audio processing code in `processBlock` a little. Change `processBlock` to the following:

```
void DelayAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer,
                                         [[maybe_unused]] juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;
    auto totalNumInputChannels = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();

    for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
        buffer.clear (i, 0, buffer.getNumSamples());

    params.update();

    float* channelDataL = buffer.getWritePointer(0);
    float* channelDataR = buffer.getWritePointer(1);

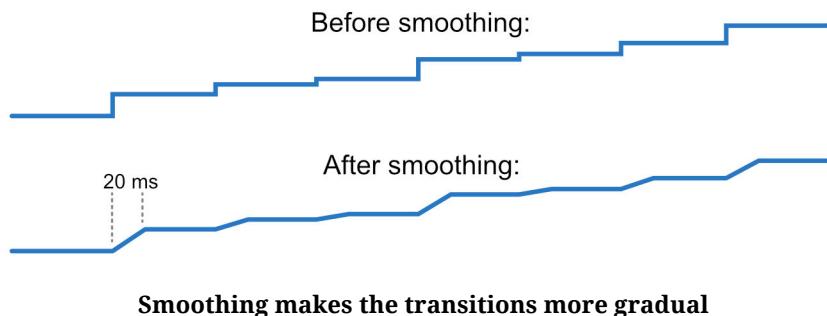
    for (int sample = 0; sample < buffer.getNumSamples(); ++sample) {
        params.smoothen();

        channelDataL[sample] *= params.gain;
        channelDataR[sample] *= params.gain;
    }
}
```

Previously this used two nested loops, the outer loop for the channels and the inner loop for the samples. Now there is only a single loop that processes both the left channel and the right channel at the same time.

`params.update()` is called once per block to read the most recent parameter value, updating the target value of the `gainSmoother` if the parameter has changed since the last block.

`params.smoothen()` is called once per sample. This means `params.gain` may have a (slightly) different value on each timestep — it will gradually move towards the new parameter value, one sample at a time. This kind of smooth motion is what prevents the zipper noise.



That's it for smoothing the gain parameter! Try the plug-in again. The zipper noise should have disappeared and the parameter updates are silky-smooth.

From now on, whenever we add a new parameter to the plug-in, we'll also give it its own `juce::LinearSmoothedValue` object and update this in the `Parameters` object's `update` and `smoothen` functions.

Maybe you're wondering why we didn't use two nested loops like before? Something like this perhaps:

```
params.update();

for (int channel = 0; channel < totalNumInputChannels; ++channel) {
    auto* channelData = buffer.getWritePointer(channel);

    for (int sample = 0; sample < buffer.getNumSamples(); ++sample) {
        params.smoothen();
        channelData[sample] *= params.gain;
    }
}
```

This first processes all the samples in the left channel and then all the samples in the right channel. But note that `params.smoothen()` is called twice as often now, once for every sample in the left channel and once for every sample in the right channel.

Worse, the smoothing jumps from the last sample in the left channel to the first sample in the right channel, possibly creating zipper noise of its own.

To avoid this, we process the left and right channels at the same time. We're going to have to do that anyway once we introduce feedback into the delay effect, but let's not get ahead of ourselves yet.

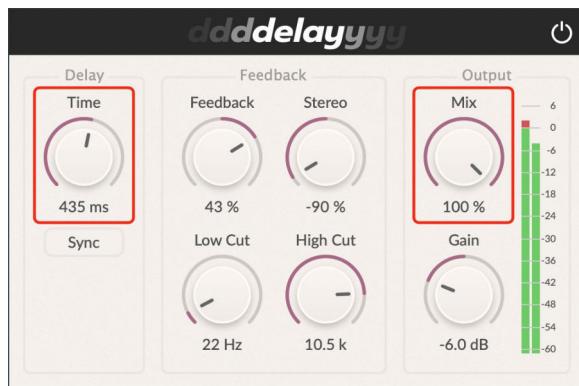
It may seem like you've been doing very little audio programming so far and a lot of housekeeping, especially with managing the parameters. This is true, but now that the basics are in place you're in pretty good shape to build out the rest of the plug-in and dig into some real audio programming!

In the next chapter you'll add a delay line to create the basic delay effect.

9: Adding the delay

All right, let's get dirty with the DSP (digital signal processing). In this chapter you will add a basic delay effect, which is the foundation for our plug-in.

This is the part of the plug-in you'll build in this chapter:



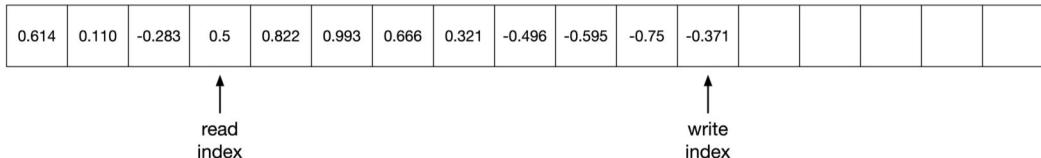
We're not building the UI yet, that's for the next chapter. This one is all about the audio.

The delay line

To delay a sound by a certain amount of time, we will have to keep track of its samples somehow. Let's say the delay time is a half second. At a sample rate of 48 kHz that is 24000 samples. Suppose the block size is 128 samples, then it takes 187 calls to `processBlock` before we should start outputting the delayed sound.

In the meantime, we have to store the samples for the delayed sound in the computer's memory somewhere, and wait for the right moment to start outputting them. The tool for this is the **delay line**.

A delay line is nothing more than a chunk of memory that will store sample data. In that sense it serves the same purpose as `juce::AudioBuffer`. In general programming terms this is called an **array**.



A delay line is an array of sample values with a read and write index

At every timestep, we insert the next sample value from the incoming audio into the delay line at the **write index** and then move this index ahead by one position. In the illustration time goes from left to right, meaning 0.614 is the oldest sample and -0.371 is the latest sample received.

To get the delayed signal, we will read the sample value from an earlier position, the **read index**. For the purpose of fitting it in this illustration, the read position is 8 samples in the past. For a delay time of half a second, the read position in the array would be 24000 samples earlier than the write position.

In our plug-in, the maximum delay time will be 5 seconds, meaning that at a sample rate of 48 kHz, the array should be $48,000 \times 5 = 240,000$ elements long. That's a lot of numbers!

The length of the delay line depends on the sample rate. For a sample rate of 44.1 kHz, the array will be smaller. For 96 kHz, the array must be much longer. This kind of thing happens a lot in audio code: the size of some object that you need depends on the current sampling rate. This is why `DelayAudioProcessor` has a `prepareToPlay` function that takes two parameters: the sample rate and the maximum expected block size. Often you'll need both to prepare your DSP objects before use.

In chapter 16 you will learn how to write your own delay line from scratch, but we'll start off by using JUCE's built-in `juce::dsp::DelayLine` object.

Perhaps you're wondering why we don't simply use a `juce::AudioBuffer`, since that also holds an array of sample values, or even a standard C++ class such as `std::vector`? There are two reasons:

1. To make the delay line efficient it must act as a so-called **circular buffer** when storing the sample data. `juce::AudioBuffer` and `std::vector` are not circular.

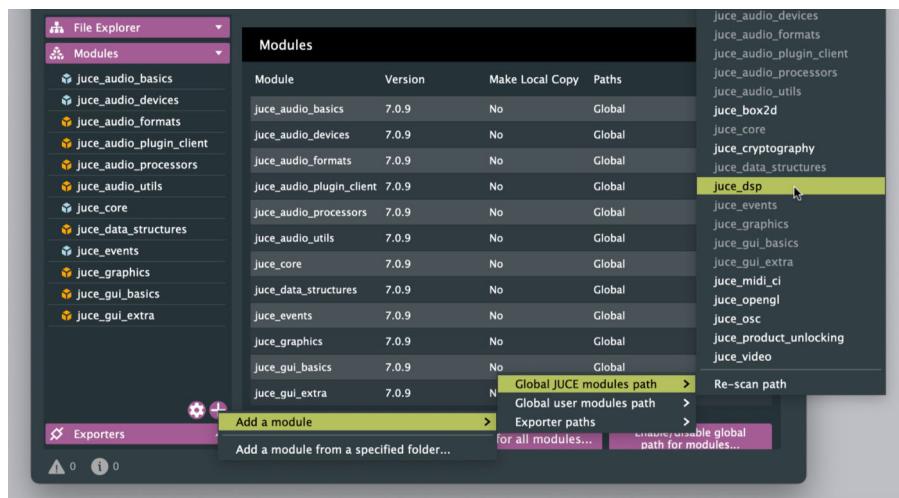
2. The user will set the delay time in seconds or milliseconds. Such delay times do not always correspond to a whole number of samples. For example, at a sample rate of 44.1 kHz and a delay time of 25 ms (milliseconds), the delay length is 1102.5 samples. This means we will need to read “in between” the past samples 1102 and 1103 somehow.

The standard solution to fractional delay lengths is to **interpolate** the two sample values to guess what the value in the middle is. `juce::dsp::DelayLine` offers various ways to do this.

We will go into more detail on these topics when we’ll implement our own delay line. For now, it’s only important to know that `juce::dsp::DelayLine` is a better choice than a regular `AudioBuffer` or `vector`.

Using `juce::dsp::DelayLine`

The `juce::dsp` module is a very handy collection of DSP objects, such as filters, convolution, oversampling, a delay line, and more, including full effects such as chorus, compressor, and reverb. It has plenty of pre-built DSP building blocks that you can use in your own plug-ins.



Adding the `juce_dsp` module in Projucer

The `juce::dsp` module wasn’t added to the project when you created it in Projucer, so you’ll have to add it manually. Go to **Projucer** and open the **Delay.jucer** project.

Click on **Modules** on the left to reveal the modules in use by the project. Click the round + button, then **Add a module** > **Global JUCE modules path** > **juce_dsp**.

The **juce_dsp** module should now be added to the list of modules. Save the project again and open it in your IDE.

Go to **PluginProcessor.h** and add the following line in the `private:` section, below the other variables:

```
juce::dsp::DelayLine<float, juce::dsp::DelayLineInterpolationTypes::Linear> delayLine;
```

This syntax may require some explanation. The code declares a new variable named `delayLine` of type `juce::dsp::DelayLine`, but you can't just write:

```
juce::dsp::DelayLine delayLine;
```

That would give a compiler error. `juce::dsp::DelayLine` is not only a class but a **class template**. It is incomplete until you specify some additional properties. In the case of `juce::dsp::DelayLine`, the compiler needs to know two more things before you can use it:

1. The data type of the samples it will hold. We're using `float`.
2. How to do the interpolation for delay lengths that fall in between samples. For now, we'll go with `juce::dsp::DelayLineInterpolationTypes::Linear` for linear interpolation, but we'll explore other interpolation types later.

These two things are the **template arguments** and should be placed inside the `<` and `>` brackets. They are like function arguments except they're evaluated at compile-time, not while the program is running.

You've already seen this `<float>` notation used with `juce::LinearSmoothedValue`, `juce::AudioBuffer`, and `juce::NormalisableRange` too. It's very common for JUCE's objects to require such template arguments.

Before we can put the `juce::dsp::DelayLine` object to work, it needs to be initialized. Open **Parameters.h** and add the following lines inside the `public:` section of the class:

```
static constexpr float minDelayTime = 5.0f;
static constexpr float maxDelayTime = 5000.0f;
```

This defines two constants, one named `minDelayTime` with the value `5.0f` and another `maxDelayTime` with the value `5000.0f`. These are the minimum and maximum possible delay times expressed in milliseconds. Since there are 1000 seconds in a millisecond, this makes the maximum delay 5 seconds.

The reason we're putting this into a constant is so that whenever we need to refer to the maximum delay time, we can write `Parameters::maxDelayTime` instead of a hardcoded number. If for some reason we change our minds and want to have a different maximum time, we just need to update the constant definition here and the rest of the code will immediately pick up the new value.

You're adding these constants to the `Parameters` class because the delay time will be a plug-in parameter, so it makes sense that the minimum and maximum values also belong in the same class.

Now that we've determined what the maximum delay time will be, we can calculate how many samples the `juce::dsp::DelayLine` must be able to hold for the given sample rate. This happens in `prepareToPlay` in **PluginProcessor.cpp**.

First, add the following lines to `prepareToPlay`:

```
juce::dsp::ProcessSpec spec;
spec.sampleRate = sampleRate;
spec.maximumBlockSize = juce::uint32(samplesPerBlock);
spec.numChannels = 2;

delayLine.prepare(spec);
```

The `juce::dsp::DelayLine` object needs to know the current sample rate, the maximum expected block size, as well as the number of audio channels (2 for stereo). You use a `juce::dsp::ProcessSpec` object for this.

Note: The expression `juce::uint32(samplesPerBlock)` looks like a function call but is actually a **type cast** from the data type of `samplesPerBlock`, which is an `int`, to the data type as expected by `spec.maximumBlockSize`, which is a `juce::uint32`. The difference is that `int` can hold positive and negative numbers while `uint32` can only hold positive numbers (the `u` stands for `unsigned`). This is a tiny detail that doesn't really matter in this case, but without the cast the compiler will give a warning. By using the type cast we say to the compiler, "It's OK, you can safely put this `samplesPerBlock` value into the `maximumBlockSize` variable."

Also add the following to `prepareToPlay`, below the lines you just added:

```
double numSamples = Parameters::maxDelayTime / 1000.0 * sampleRate;
int maxDelayInSamples = int(std::ceil(numSamples));
delayLine.setMaximumDelayInSamples(maxDelayInSamples);
delayLine.reset();
```

This tells the `juce::dsp::DelayLine` object what its maximum expected delay length is, measured in number of samples. This is important because the delay line needs to know how much memory to reserve to remember enough “old” samples.

To convert a time in milliseconds to a number of samples, first divide the millisecond amount by 1000 to convert it to seconds, and then multiply by the sample rate. The `/` symbol is the division operator in C++.

Hopefully this formula makes sense. If the delay time is one second and the sample rate 48 kHz, the delay length should be 48000 samples. In that case the delay time is 1000 milliseconds, so `1000 / 1000.0 * sampleRate` is 48000 indeed. If the delay time is 0.5 seconds, or 500 ms, the delay length should be half that, 24000 samples. This works out too: `500 / 1000.0 * 48000 = 24000`.

I always find it useful to think through a few examples this way, to verify my math is correct. Note that `/` and `*` are evaluated from left-to-right, so we first divide and then multiply. You could add parenthesis to make the evaluation order more explicit: `(maxDelayTime / 1000.0) * sampleRate`.

Small detail: The result of this calculation is a `double` because `sampleRate` is `double` instead of `float`. This is also why you wrote `1000.0` instead of `1000.0f`.

The maximum possible delay length should be a whole number of samples. For example, if `numSamples` comes out to 170.2, we should round it up to 171 samples. That’s what `std::ceil()` does, where “ceil” stands for ceiling. `int(...)` casts this result into an integer variable.

Finally, `delayLine.reset()` makes sure the delay line is ready for use. To verify that the formula works, add the following line to the bottom of `prepareToPlay`:

```
DBG(maxDelayInSamples);
```

Build the plug-in and run it in `AudioPluginHost`. The `DBG` command will write the value of `maxDelayInSamples` to your IDE’s debug output pane. It should print 220500 for a sample rate of 44.1 kHz and 240000 for 48 kHz.

You can change the sample rate in `AudioPluginHost` by double-clicking the **Audio Output** block. As soon as you pick a new sample rate, `prepareToPlay` is called again and the `juce::dsp::DelayLine` object is re-initialized with the new maximum length. (Sometimes you will see the same value being printed several times. This is because `prepareToPlay` may be called more than once by the host. This is fine.)

Once you're done playing with the sample rate, you may remove the `DBG` line from `prepareToPlay`, or comment it out by adding `//` in front of it.

Note: The code you just wrote used `Parameters::maxDelayTime`. Why not `params.maxDelayTime`, since `params` is the name of the `Parameters` object from the audio processor? Good question! The reason is that `maxDelayTime` is declared as **static** in `Parameters.h`, meaning that it does not belong to any particular instance of `Parameters` but to the class itself. This is needed because later we'll use `maxDelayTime` from `createParameterLayout()`, which is also a static member of `Parameters`.

Putting the delay line into action

The delay line doesn't actually do anything yet, since we haven't added it to the audio processing code. Go to `processBlock` and change the audio processing loop to the following:

```
void DelayAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer,
                                         [[maybe_unused]] juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;
    auto totalNumInputChannels = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();

    for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
        buffer.clear (i, 0, buffer.getNumSamples());

    params.update();

    // 1
    delayLine.setDelay(48000.0f);

    float* channelDataL = buffer.getWritePointer(0);
    float* channelDataR = buffer.getWritePointer(1);

    for (int sample = 0; sample < buffer.getNumSamples(); ++sample) {
        params.smoothen();

        // 2
        float dryL = channelDataL[sample];
        float dryR = channelDataR[sample];

        // 3
        delayLine.pushSample(0, dryL);
        delayLine.pushSample(1, dryR);

        // 4
        float wetL = delayLine.popSample(0);
        float wetR = delayLine.popSample(1);

        // 5
        channelDataL[sample] = wetL * params.gain;
        channelDataR[sample] = wetR * params.gain;
    }
}
```

Most of the processing loop is still the same. I've highlighted the differences with numbered comments.

1. Set the current delay length. In `prepareToPlay` you only told the delay line what its maximum allowed length was. Here we set the actual delay length, again measured in samples. This time the value is a `float` instead of an `int` because fractional delay lengths are possible.
2. Read the incoming audio samples into new variables, `dryL` and `dryR`, for the left and right channels, respectively. The `dry` signal is what we call the unprocessed audio.
3. Write the `dryL` and `dryR` values into the delay line using `delayLine.pushSample`. We do this twice, once for the left channel (index 0) and once for the right channel (index 1).
4. Read the delayed audio using `delayLine.popSample`, once for each channel. These are the `wet` sample values, the name we give to the processed signal.
5. Write the wet samples back into the `AudioBuffer`, multiplied by the current gain value.

For now, the delay line uses a fixed delay length of 48000 samples, which is one second of delay at 48 kHz, or slightly more at 44.1 kHz. In the next section you'll use a proper plug-in parameter for this.

Try it out! Build the plug-in and run it. When you press play on the audio file player, there should be a one second delay before you will hear any sound. Likewise, after you press stop on the audio file player it keeps the sound playing for another second. That means the delay line works! Everything is delayed by one second.

Usually you'd want to mix the delayed audio with the original dry sound. This is called the **dry/wet mix** and we'll talk about it more later in this chapter. For the time being, change the last two lines in the audio processing loop to the following:

```
channelDataL[sample] = (dryL + wetL) * params.gain;  
channelDataR[sample] = (dryR + wetR) * params.gain;
```

This performs a basic mix that first adds the dry and wet signals together and then applies the output gain.

Build and run the plug-in and you'll now hear the original sound plus the delayed version one second later, playing at the same time. This works best with a fairly sparse audio file so you can clearly hear the delay.

Also notice that the sound may become a lot louder, since adding a signal to itself will boost the amplitude by a factor of two, which corresponds to a 6 dB gain. You may want to turn the **Output Gain** parameter down by about 6 decibels to compensate and avoid clipping.

Setting the delay time with a parameter

Always using a delay of one second doesn't make for a very interesting plug-in, so let's add a new plug-in parameter. This is of course done in our **Parameters** class.

Open **Parameters.h** and add the following variable to the `public:` section of the class:

```
float delayTime = 0.0f;
```

Add the next variable to the `private:` section:

```
juce::AudioParameterFloat* delayTimeParam;
```

We won't add a smoother for this parameter just yet, so there's no variable for that.

Also add a new constant below the existing one at the top of the file. This declares `delayTimeParamID` to be the `juce::ParameterID` for our new parameter.

```
const juce::ParameterID delayTimeParamID { "delayTime", 1 };
```

Go to **Parameters.cpp**. In the constructor, add a line to put a pointer to this parameter into the new `delayTimeParam` variable:

```
castParameter(apvts, delayTimeParamID, delayTimeParam);
```

In the `createParameterLayout` function, just before the line with `return layout;`, create the new plug-in parameter object:

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    delayTimeParamID,
    "Delay Time",
    juce::NormalisableRange<float> { minDelayTime, maxDelayTime },
    100.0f));
```

This code is very similar to the definition of the Output Gain parameter. The range is from `minDelayTime` to `maxDelayTime`, or 5 milliseconds to 5 seconds. The default setting is `100.0f` or 100 ms.

In the `reset()` function, add the following line to re-initialize the variable:

```
delayTime = 0.0f;
```

The last change to the `Parameters` class is adding a new line to `update()`. This will read the value from the parameter and store it in the `delayTime` variable.

```
void Parameters::update() noexcept
{
    gainSmoother.setTargetValue(juce::Decibels::decibelsToGain(gainParam->get()));

    delayTime = delayTimeParam->get(); // add this line
}
```

Note that the `delayTime` variable holds the time in milliseconds, since that's the unit we're using for this parameter.

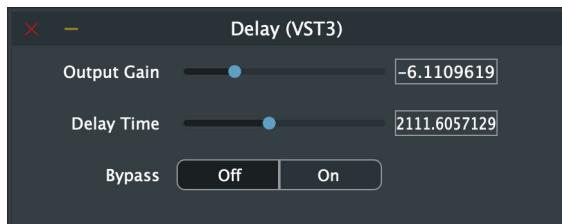
Now let's use this `delayTime` in the audio processing code. Go to **PluginProcessor.cpp** and in `processBlock` change the line `delayLine.setDelay(48000.0f);` to the following:

```
float sampleRate = float(getSampleRate());
float delayInSamples = params.delayTime / 1000.0f * sampleRate;
delayLine.setDelay(delayInSamples);
```

This uses the same formula as before to convert a time in milliseconds into a number of samples: `delayTime / 1000.0f * sampleRate`.

Here, however, we're not using `std::ceil()` to round up the number of samples to an integer. It's perfectly fine to set a delay length that is a fractional number of samples. The `juce::dsp::DelayLine` object will interpolate between samples if necessary.

Try it out. You can change the delay time by dragging the **Delay Time** slider. Sweet!



The plug-in has a **Delay Time** parameter

Note: Since `params.update()` happens on every block, perhaps you're wondering what happens if the parameter didn't change since the last `processBlock`? In that case we'll call `delayLine.setDelay(...)` with exactly the same delay amount as in the previous block. This is perfectly fine. The `juce::dsp::DelayLine` object is smart enough to see that the delay length didn't change and so this call to `setDelay` will simply be ignored.

Improving the parameter

The **Delay Time** parameter works, but it could be better. First of all, it's really hard to dial in short delays.

In `AudioPluginHost`, notice that the default setting of 100 ms is almost all the way to the left on the slider (double-click or Alt-click to reset the parameter to the default). It's nearly impossible to choose a shorter delay time!

This happens because JUCE divides the parameter's range — from 5 ms to 5000 ms — equally over the slider. That's not ideal. Instead, we want to have more precision when setting short delay times. A 20 ms delay will sound very different from a 30 ms delay, but 1020 ms and 1030 ms sound pretty much the same.

Fortunately, JUCE makes this easy. We can add a **skew** factor to the parameter definition that will make more room on the slider for small values.

In `Parameters.cpp`, in `createParameterLayout()`, change the line that sets the range on the **Delay Time** parameter to the following:

```
juce::NormalisableRange<float> { minDelayTime, maxDelayTime, 0.001f, 0.25f },
```

This adds two arguments: `0.001f` is the step size, meaning that the slider will move in steps of 0.001 ms. The last argument, `0.25f`, is the skew factor. You can play with this value to see what you like best. The smaller this number is, the more emphasis low values get on the slider.

Try it out. Run the plug-in and the slider now gives a lot more control over small delay time values:



The skew factor makes it easier to choose short delay times

Another thing we can do is improve the way the parameter's value is displayed in the editor. Currently, the delay time always has three digits behind the decimal point. This is useful for short delay times but a little pointless for larger delays. Again, JUCE makes this easy.

First add a new function at the top of **Parameters.cpp**, below the `castParameter` function.

```
static juce::String stringFromMilliseconds(float value, int)
{
    if (value < 10.0f) {
        return juce::String(value, 2) + " ms";
    } else if (value < 100.0f) {
        return juce::String(value, 1) + " ms";
    } else if (value < 1000.0f) {
        return juce::String(int(value)) + " ms";
    } else {
        return juce::String(value * 0.001f, 2) + " s";
    }
}
```

This `stringFromMilliseconds` function takes two arguments. The first, `value`, is a floating-point number containing the parameter's value — a time in milliseconds. The second argument is an `int` with the maximum allowed string length but we're going to ignore that. That's why this argument doesn't have a name.

`stringFromMilliseconds` converts the parameter's time value into text, stored in a `juce::String` object, and returns that string to the caller.

Inside the function, there is a series of `if` statements that look at how large the parameter's value is and construct the string based on that. The purpose of an `if` statement is to allow our programs to make decisions.

Here's how this logic works:

- `if (value < 10.0f)`: If the value is less than 10 milliseconds, return a string that has the value rounded to two decimals. For example, the value 7.1234 becomes the text "7.12 ms".
- `else if (value < 100.0f)`: Otherwise, if the value is less than 100 milliseconds, return a string with one decimal. For example, 56.83 becomes "56.8 ms".
- `else if (value < 1000.0f)`: Else, if the value is less than 1000 milliseconds — or 1 second — return a string with no decimal point. The `int(value)` chops off anything after the decimal point. For example, 819.25 becomes "819 ms".
- `else`: Finally, if none of the above are true, we know the parameter is set to 1000 milliseconds or more, and we'll return an amount in seconds. Note that `value * 0.001f` is the same as `value / 1000.0f`. The result is rounded to two places. For example, 3154.7 becomes "3.15 s".

The `if` statements are arranged from smallest to largest on purpose. By first checking if the value is less than 10, we catch anything between 5 ms and 10 ms (since 5 ms is the smallest the parameter can get). If that's not the case, we check for values less than 100 ms. This catches anything between 10 ms and 100ms. And so on.

To use this `stringFromMilliseconds` function, we have to add something to the parameter definition in `createParameterLayout`. Change the code that creates the **Delay Time** parameter to the following.

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    delayTimeParamID,
    "Delay Time",
    juce::NormalisableRange<float> { minDelayTime, maxDelayTime, 0.001f, 0.25f },
    100.0f,
    juce::AudioParameterFloatAttributes()
        .withStringValueFunction(stringFromMilliseconds)
));
```

Using `juce::AudioParameterFloatAttributes().withStringFromValueFunction()` is how we tell JUCE to use the new `stringFromMilliseconds` function when the host asks for a textual representation of the parameter's value.

This is known as the **string-from-value** function. Note that I had to split this into two lines to make it fit in the book, but normally I'd put that all on a single line.

There indeed should be two closing parentheses for this entire statement, that is not a typo. The first belongs to `std::make_unique<...>(...)` and the last belongs to `layout.add(...)`. Also don't forget to put a comma behind the `100.0f` now.

Might as well do this for the gain parameter too. Add the following function below `stringFromMilliseconds`:

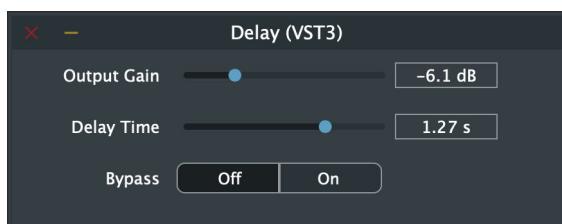
```
static juce::String stringFromDecibels(float value, int)
{
    return juce::String(value, 1) + " dB";
}
```

This rounds off the decibel amount to one decimal and adds “dB” behind it. Change the **Output Gain** parameter definition in `createParameterLayout` to use `stringFromDecibels` as its string-from-value function:

```
layout.add(std::make_unique

```

Try it out. The editor now shows both parameters in a much clearer way.



The parameter values have units now

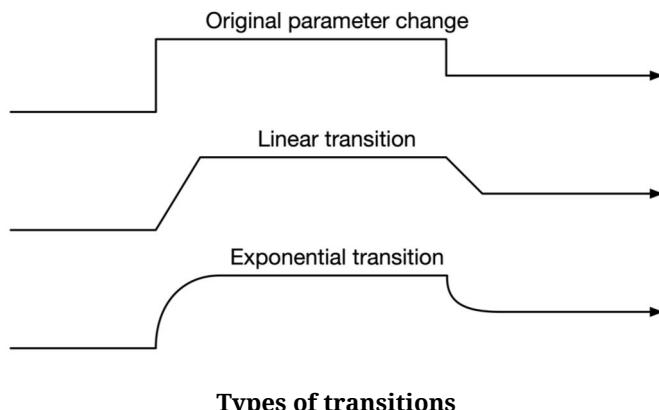
Note: The `stringFromMilliseconds` and `stringFromDecibels` functions are not part of the `Parameters` object but are free functions. Notice that they have the extra keyword `static`. This means they can only be used inside `Parameters.cpp`. Similar to private members of a class, static functions are not visible to any of the other source files in the project. You've already seen the keyword `static` used elsewhere too, with `minDelayTime` and `maxDelayTime`. It had a different meaning there. In C++ the same keyword can mean different things, depending on the context where it's used (there are at least three ways to use `static`, believe it or not).

Parameter smoothing, redux

You may have noticed that changing the **Delay Time** while sound is playing will create that nasty crackling again that we also ran into with the **Output Gain** parameter, the dreaded zipper noise.

Perhaps you were expecting that we'd fix this by using a `juce::LinearSmoothedValue` object like we did for Output Gain. That certainly is possible, but for the Delay Time parameter I want to show another method for smoothing values over time.

We're going to use a so-called **one-pole filter** to do the smoothing. This is one of the simplest kinds of filters and it's mostly useful for creating exponential transitions between signals.



With `juce::LinearSmoothedValue`, when the parameter changes there is a linear fade between the old and new value. Linear means in a straight line. Linear fades are quite useful, and appropriate for doing parameter smoothing, but sometimes we may want exponential transitions instead. The exponential curve starts out fast and then slows down. Such transitions often sound more natural to us.

Recall that zipper noise happens because the control signal for the parameters is updated at a much lower rate than the audio signal is rendered. The one-pole filter is a low-pass filter that essentially upsamples the control signal so that it looks smooth at audio rate too.

For most parameters I'd suggest using a `juce::LinearSmoothedValue` like we did before, but I find that using the exponential curves of a one-pole filter sound better for the **Delay Time** parameter. This is more like what happens with a classic analog tape delay. Plus, it gives me an excuse to talk about this super handy filter that you'll find all over audio code, not just for smoothing parameters.

How this works is that we'll set a **target** delay time that is read from the parameter. On every timestep, the filter will move the actual delay time a bit closer to the target.

In **Parameters.h**, add two new variables to the `private:` section of the class:

```
float targetDelayTime = 0.0f;
float coeff = 0.0f; // one-pole smoothing
```

The `targetDelayTime` variable is the value that the one-pole filter is trying to reach. The `coeff` variable determines how fast the smoothing happens. I added a comment to remind myself what the purpose is of these variables.

In **Parameters.cpp** in the `update` function, replace this line,

```
delayTime = delayTimeParam->get();
```

with the following:

```
targetDelayTime = delayTimeParam->get();
if (delayTime == 0.0f) {
    delayTime = targetDelayTime;
}
```

Instead of putting the parameter's value into `delayTime`, it is now placed into `targetDelayTime`. Rather than immediately switching to the new setting, we'll treat the **Delay Time** as the target to move towards.

We also need to calculate the **filter coefficient** for the variable `coeff`. A filter usually has one or more coefficients that determine exactly what gets filtered. Since our filter is a one-pole filter, it has just one coefficient.

The coefficient depends on the sample rate, so a good place to calculate it is in `Parameters prepareToPlay`. Add the following line to this function:

```
coeff = 1.0f - std::exp(-1.0f / (0.2f * float(sampleRate)));
```

I'm not going to go into the math here, save to say that the `0.2f` in this formula means 200 milliseconds. After 200 ms, the one-pole filter will have approached the target value to within 63.2%. Why that particular percentage? Well, this formula describes the charge time of an analog capacitor and it's common to use the same charge curve in DSP code as well. A lot of what we do in digital signal processing is actually based on the behavior of analog electronic components!

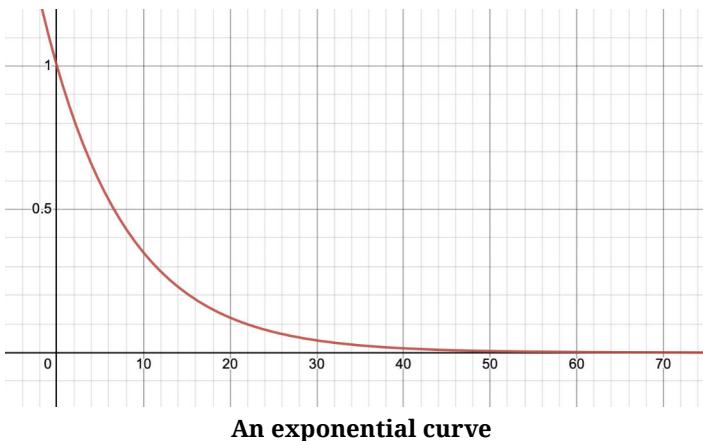
Finally, add the following line to `smoothen()` to do the actual work. This is the famous one-pole filter formula.

```
delayTime += (targetDelayTime - delayTime) * coeff;
```

All right, all the pieces are in place, so let's get a sense of how this works. The `coeff` variable is a value between 0 and 1. Whenever you multiply some value over and over again with something like 0.9, it will become smaller and smaller, in an exponential fashion.

```
1.0 * 0.9 = 0.9
0.9 * 0.9 = 0.81
0.81 * 0.9 = 0.729
0.729 * 0.9 = 0.656
0.656 * 0.9 = 0.591
0.591 * 0.9 = 0.531
0.531 * 0.9 = 0.478
```

...and so on. If we plot this in a graph, you can see the typical exponential curve. Initially the steps between successive values are large, but they become smaller over time. The curve never quite reaches zero.



What happens in the one-pole filter is similar, although it doesn't just multiply by `coeff`, it also multiplies by $1 - \text{coeff}$.

In general, the formula for the one-pole filter is:

```
currentValue += (targetValue - currentValue) * coeff;
```

If it's not immediately obvious what's going on here, you can rewrite the formula using standard arithmetic rules. First, expand the `+=` operator. This is shorthand for adding the value on the right to `currentValue` and then assigning the result back to `currentValue`, like so:

```
currentValue = currentValue + (targetValue - currentValue) * coeff;
```

Then expand the parentheses:

```
currentValue = currentValue + targetValue*coeff - currentValue*coeff;
```

And finally, reorder and group the terms:

```
currentValue = currentValue*(1 - coeff) + targetValue*coeff;
```

This is the exact same formula, except it does two multiplications instead of one. Because multiplications take time, we generally prefer the formulation that multiplies only once.

It should be clear now that on every timestep the current value is multiplied by $(1 - \text{coeff})$ and the target value by coeff . Note that $1 - \text{coeff}$ and coeff always add up to one. This way the filter doesn't introduce any additional gain.

Let's work through an example: If coeff is 0.1, then `currentValue` is multiplied by 0.9 and `targetValue` by 0.1. The current value gradually becomes less, just like in the exponential curve from the picture. At the same time, we add in a small portion of the target value. As a result, we're always moving towards the target.

The trick is that we assign the result of this formula back to `currentValue` again, so that with every timestep the distance between `currentValue` and `targetValue` becomes smaller and the movement slows down. That is what creates the exponential shape: it starts out fast but the steps become smaller and smaller over time.

Assuming the sample rate is 48 kHz, coeff will be 0.0001041612. That's very small number, but keep in mind that we're performing these multiplications 48000 times per second — even small values add up when used many times.

The smaller coeff is, the longer it takes for the filter to reach the target value. A smaller multiplier therefore means more smoothing. In other words, it makes the cutoff frequency of the filter lower — the fewer high frequencies are in the signal, the slower it moves.

The nice thing about the one-pole filter is that the `targetValue` can change at any time without causing any issues. So, if the user moves the **Delay Time** slider while it's still smoothing, the filter simply adjusts and keeps moving, just towards a new target.

There's one more thing to do before you can try it out. In `processBlock` the delay length is set once per block. However, now that we're smoothing the delay time, you must move the call to `delayLine.setDelay()` inside the loop, so that it happens on every time step, after `params.smoothen()`.

In **PluginProcessor.cpp**, in `processBlock`, cut these two lines and move them immediately below the line that does `params.smoothen()`.

```
float delayInSamples = params.delayTime / 1000.0f * sampleRate;
delayLine.setDelay(delayInSamples);
```

The line that sets `float sampleRate = ...` can remain outside the loop as the sample rate will never change during the block.

That's all we need. Try it out! You'll find that dragging the **Delay Time** slider no longer gives undesirable crackling noises... but it does create kind of a "whooshing" sound, almost like scratching a record on a turntable. This might sound like a bug but it is also what happens when you move the position of the read head on an analog tape machine while it's playing.

What you're hearing is the sound being pitched up or down, depending on the direction you're moving the slider. That's what happens if you change the length of a delay line while it's playing. In fact, this is a common method for implementing a pitch shifting effect.

Feel free to experiment with the smoothing time / filter coefficient to see what kind of effect this has on the sound. Do this by replacing the `0.2f` in the `coeff` calculation in `prepareToPlay` with a larger or smaller value. For example, try `0.01f` for a very short smoothing time (almost sounds like lasers) and `1.0f` for a rather long one.

With the long smoothing time, it may be fun to remove the dry signal from the output by temporarily changing the last lines in `processBlock` like so:

```
channelDataL[sample] = wetL * params.gain;  
channelDataR[sample] = wetR * params.gain;
```

Now you can really hear what happens when the delay time changes. Moving the slider rapidly to the left (towards a smaller delay) will initially pitch the sound up — and speed it up — before returning to the regular speed. Moving the slider to the right will do the opposite: slow down and play at a lower pitch.

Even though you may have been surprised by this "whooshing" sound effect, it's a natural side effect of changing — also known as **modulating** — the delay time while sound is playing. There are several well-known audio effects that are based on the idea of modulating the delay time. I already mentioned pitch shifting (known as vibrato if it's done in small amounts), but chorus and flanger are two others.

I kind of like the sound this makes, and it's certainly appropriate when trying to emulate an analog tape delay unit. At the very least, it doesn't sound nasty as zipper noise did — the result is still musical. It's a matter of taste whether you like it or not.

In a later chapter we'll investigate a way to get rid of this pitching sound because it can start to behave weirdly once we add feedback.

Tip: I'm sure you believe my explanation of how the one-pole filter works, but wouldn't you like to *see* it for yourself? I often find it useful to visually inspect my audio processing code, since the eye can spot issues that the ear might miss.

Here is a quick-and-dirty way to visualize the output of the one-pole filter: We can change `processBlock` to output the value of `params.delayTime` as it changes over time by pretending it's an audio signal. It won't sound like audio but we can still inspect it using an oscilloscope.

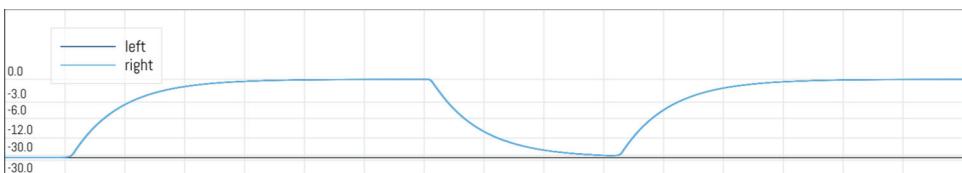
In `processBlock` write the following at the end of the audio processing loop:

```
channelDataL[sample] = params.delayTime / 5000.0f;
channelDataR[sample] = params.delayTime / 5000.0f;
```

Instead of the wet signal, this now outputs the value of `params.delayTime` as it evolves over time. One caveat: audio signals should be in the range -1.0 to 1.0 , whereas the delay time is between 5 and 5000 milliseconds. So, we divide by 5000 to get something more reasonable.

Mute your computer's speaker volume and run the plug-in. In `AudioPluginHost`, disconnect the **Delay** block from the **Audio Output** block so that it won't try to output this signal as audio. It won't hurt your ears but it may pop your speakers and that's not good for them.

Make sure the **Delay** block is connected to an oscilloscope plug-in and move the **Delay Time** slider around a bit. You should see the exponential curves in the oscilloscope. This is the value of `params.delayTime` transitioning between different settings of the plug-in parameter.



Viewing the smoothed delay time in an oscilloscope

Outputting values as if they were an audio signal and looking at them using an oscilloscope is a simple trick to help debug the plug-in and to verify everything works as it should. It looks good to me, so make sure to put the code in `processBlock` back the way it was.

Dry/wet mix

Right now, the delayed sound (the wet signal) is mixed at full strength into the original sound (the dry signal). Most plug-ins have a knob that lets the user change the dry/wet mix amount. We'd be remiss not to add such a control to our plug-in too.

First let's add the parameter. In **Parameters.h**, add a new ParameterID declaration below the others:

```
const juce::ParameterID mixParamID { "mix", 1 };
```

Add a new variable to the `public:` section of the **Parameters** class:

```
float mix = 1.0f;
```

This variable will hold a value between `0.0f`, which means 0% mix, and `1.0f`, which means 100% mix. We initialize it to 100%, which seems reasonable.

Also add the following two lines to the `private:` section:

```
juce::AudioParameterFloat* mixParam;
juce::LinearSmoothedValue<float> mixSmoother;
```

Just like before, we want a pointer to the `AudioParameterFloat` object so we can easily read it, and a `LinearSmoothedValue` to get rid of zipper noise.

In **Parameters.cpp** add a new string-from-value function named `stringFromPercent`. Place this near the other `static` functions at the top of the file.

```
static juce::String stringFromPercent(float value, int)
{
    return juce::String(int(value)) + " %";
}
```

This first uses `int(value)` to remove anything after the decimal point, then puts this number into a `juce::String` object, and adds the % sign to the end of the string.

In the constructor for the **Parameters** class, grab the `AudioParameterFloat` object from the APVTS and store it in our own pointer.

```
castParameter(apvts, mixParamID, mixParam);
```

Add the definition of the new parameter in `createParameterLayout`, after the others:

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    mixParamID,
    "Mix",
    juce::NormalisableRange<float>(0.0f, 100.0f, 1.0f),
    100.0f,
    juce::AudioParameterFloatAttributes()
        .withStringFromValueFunction(stringFromPercent)
));

```

The range of this parameter is from `0.0f` to `100.0f` in steps of `1.0f`. The default setting is `100.0f` or 100%, which means mixing in the wet signal fully. The `stringFromPercent` function is used to display these values on the screen as percentages.

Still in **Parameters.cpp**, in `prepareToPlay`, tell the smoother about the sample rate:

```
mixSmoothen.reset(sampleRate, duration);
```

In the `reset` function, set the smoother's current and target values:

```
mix = 1.0f;
mixSmoothen.setCurrentAndTargetValue(mixParam->get() * 0.01f);
```

This does `mixParam->get() * 0.01f`, which is equivalent to dividing the parameter's value by 100. So, if the parameter is at 100%, the `mix` value will be `1.0f`.

In `update`, set the smoother's target value. Again we multiply by `0.01f` here:

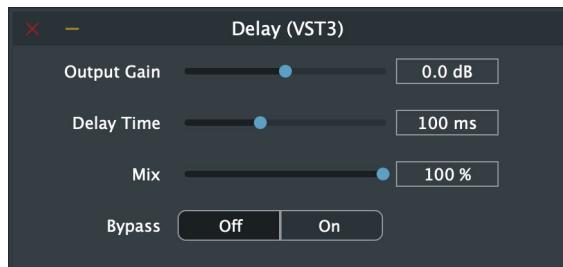
```
mixSmoothen.setTargetValue(mixParam->get() * 0.01f);
```

And finally in `smoothen` we read the value from the smoother and put it in the `mix` variable:

```
mix = mixSmoothen.getNextValue();
```

This is all stuff you've done before so it should start to make sense now. Plus, you'll be doing it a few more times before the end of the book.

Let's make sure the code compiles without errors before continuing. The user interface has a new **Mix** parameter that goes between 0 and 100%.



The new **Mix** parameter

There are several different ways to do a dry-wet mix. For a delay plug-in it makes most sense to me to always keep the dry signal and add the wet signal based on the mix amount.

In other words,

- 0% uses the dry signal only
- 50% is fully dry plus 50% wet signal
- 100% means both dry and wet are fully present

Since adding (a portion of) the wet signal to the dry signal will make the combined sound louder, the **Output Gain** knob can be used to dial down the output if it becomes too loud.

The **Mix** parameter goes from `0.0f` to `100.0f` percent, but `params.mix` is always a value between `0.0f` and `1.0f`. I did that on purpose so that the dry/wet mix logic is really easy to implement.

Go to `processBlock` in **PluginProcessor.cpp** and change the lines that write the output signal to the following:

```
float mixL = dryL + wetL * params.mix;
float mixR = dryR + wetR * params.mix;

channelDataL[sample] = mixL * params.gain;
channelDataR[sample] = mixR * params.gain;
```

First this multiplies the wet signal by `params.mix`. If **Mix** is at 100%, this has no effect since we'd be multiplying by 1.0. But if **Mix** is less than 100%, it will multiply by a value smaller than 1.0 and attenuate the wet signal. The dry signal is added and the output gain is applied to the combined signal.

Try it out. You should now have a working dry/wet mix control. Nice!

There are other ways to do dry-wet mixing. In some plug-ins 0% means the output signal is fully dry, 50% means both dry and wet are present in equal amounts, and 100% means the output is fully wet. To do that, you could write:

```
float mixL = dryL * (1.0f - params.mix) + wetL * params.mix;
float mixR = dryR * (1.0f - params.mix) + wetR * params.mix;
```

I don't think it makes a lot of sense for a delay plug-in to have the delayed signal be louder than the original, although this would allow you to treat the delay as a "send effect" and send just the echoes to an aux track. For many types of plug-ins it does make sense to turn down the dry signal. For example, in a compressor you may want to blend between dry and wet to get a parallel compression effect.

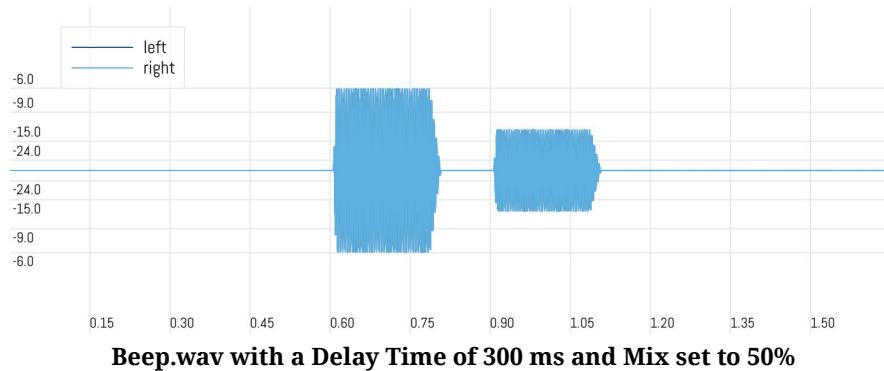
Note: The formula $a*(1 - c) + b*c$ where c is some value between 0 and 1, is known as a **linear interpolation**. You've used this same formula with the one-pole filter. `juce::dsp::DelayLine` uses it to find the amplitude between two samples if the delay length is a fractional number. And it's also one of the ways to do a dry/wet mix. Linear interpolation — or "lerp" for short — is everywhere in audio, so it's good to be able to recognize this formula whenever you come across it.

Testing the delay effect

To test whether your delay works it's useful to have a sound file that's just a short tone followed by a couple of seconds of silence. There is one included in the book's resources, **Beep.wav**. This plays a sine wave beep for 200 ms. Put the delay time at 300 ms and you should now hear two short tones play in succession.

Put the delay at 200 ms and there should be no gap between the two beeps. They should not overlap either. You can verify this with an oscilloscope: when the two beeps overlap, there is a portion where the sound is louder / the amplitude is higher, because both sound waves get added together there.

Also notice that the **Mix** setting changes the amplitude of the second beep.



Using simple audio files like this is a great way to verify that what the plug-in doing is correct. With regular music files there is usually so much going on that problems can be masked, but with a plain sine wave there isn't anywhere for problems to hide and you will immediately hear clicks or glitches or anything else that shouldn't be there.

By the way, did you notice that when the delay time is made very short, the plug-in starts to act as a filter? If you haven't tried it yet, load an audio file with white noise into the AudioPluginHost session and set the delay time to 5 ms. A white noise sample, **WhiteNoise.wav**, is included in the book's resources.

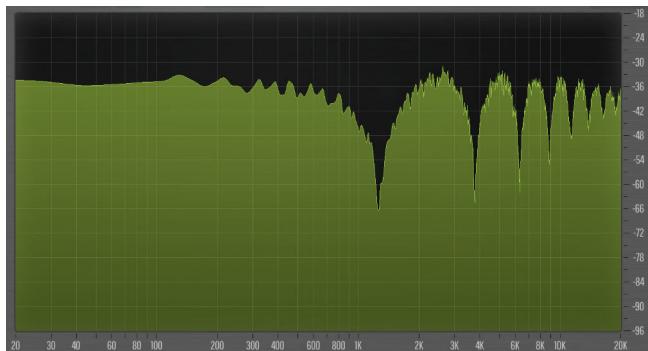
While the white noise is playing, slowly move the **Delay Time** slider between 5 and 10 ms and back again. You can hear that the sound gets filtered in an interesting way. White noise is a good test for this since it contains all possible frequencies. Try this with some other audio files too.

For fun, in **Parameters.h**, change `minDelayTime` to `0.1f` to allow even shorter delays.

It's useful to install a frequency analyzer or spectrum analyzer plug-in. I'm using [Voxengo SPAN²⁸](https://www.voxengo.com/product/span/), which is free and exceedingly capable. Hook up the **SPAN** plug-in to the **Delay** block in the AudioPluginHost session. Click the settings icon and then set **Range Lo** to `-96` and **Slope** to `0`. Turn down the **Avg Time** to about 300.

²⁸<https://www.voxengo.com/product/span/>

Now play the white noise at a **Delay Time** of 0.1 ms. Make sure **Mix** is at 100%. Notice how this shows a peculiar shape in the spectrum? Normally the spectrum of white noise should be approximately flat, as it has all frequencies in it. However, here it looks like there are arches. Slowly increase the delay time to 1 ms or so, and the arches will move.



The comb filter that results from a Delay Time of 0.4 ms

Adding a delayed signal to the original signal, like what happens with our dry/wet mix, creates a type of filter called a **comb filter**. It's named that because the arches resemble a comb. The length of the delay line determines where the notches are carved out of the spectrum. I just wanted to quickly show you this as comb filters are commonly used in audio DSP and now you know how they are made — using a delay line.

When you're done, put `minDelayTime` back to `5.0f`.

Awesome, we've got a pretty capable delay plug-in already, but there are still many more features we can add to the audio processing, such as feedback and filtering the feedback signal, ping-pong for stereo sounds, tempo sync to set the delay time based on the BPM of the DAW, and more. But first we'll start building out the user interface of the plug-in and make the editor look a lot better!

10: The user interface

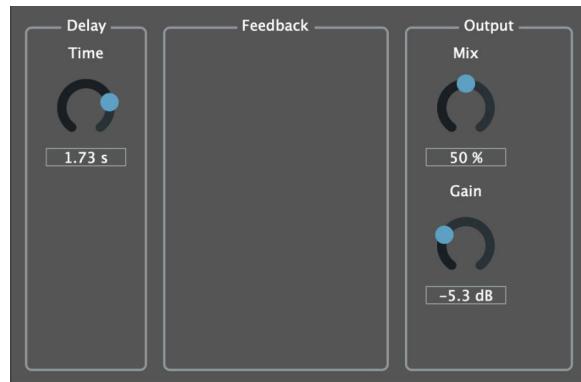
You now have a working delay plug-in, even though it's a bit basic, but it doesn't have a user interface yet. `AudioPluginHost` can render a “generic” UI but it's not likely to win any beauty contests.

In this chapter you'll learn how to create your own UI in JUCE. We will go from the generic UI that uses sliders:



The generic UI

to a custom design that uses rotary knobs in the default JUCE style:



The editor with knobs

In the next chapter you'll add styling to end up with the final result, shown on the next page.



The editor with a custom look-and-feel

There are still a few blank areas in this UI. Those will be filled up in the coming chapters as you add more features to the plug-in.

The editor

You've briefly explored the plug-in's editor in an earlier chapter when you changed the background color and text that was drawn. Let's have a closer look at the editor's source files. Hopefully you've leveled up your C++ skills enough by now that you'll be able to decipher what's going on here.

First there is **PluginEditor.h**. I stripped out some of the comments to save space.

```
#pragma once

#include <JuceHeader.h>
#include "PluginProcessor.h"

class DelayAudioProcessorEditor : public juce::AudioProcessorEditor
{
public:
    DelayAudioProcessorEditor (DelayAudioProcessor&);
    ~DelayAudioProcessorEditor() override;

    void paint (juce::Graphics&) override;
    void resized() override;

private:
    DelayAudioProcessor& audioProcessor;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (DelayAudioProcessorEditor)
};
```

Just like the `DelayAudioProcessor` class is based on `juce::AudioProcessor`, the `DelayAudioProcessorEditor` is based on `juce::AudioProcessorEditor`. That way we get a lot of the required editor functionality for free and we only have to implement those functions we actually need.

The important functions in the editor class are:

- The constructor. This is where you configure all the UI elements used by the editor.
- `paint()` for drawing the contents of the editor.
- `resized()`, which is used to position and arrange the UI elements that make up the editor.

The editor is made up of multiple user interface elements, such as text labels, sliders, rotary knobs, buttons, and so on. In JUCE parlance, UI elements are known as **components**.

The `DelayAudioProcessorEditor` is created by the `DelayAudioProcessor` in the function `createEditor`. If you open **PluginProcessor.cpp** and scroll to the code for `createEditor`, it looks like this:

```
juce::AudioProcessorEditor* DelayAudioProcessor::createEditor()
{
    return new DelayAudioProcessorEditor (*this);
}
```

When the user tries to open the UI for a plug-in, for example in `AudioPluginHost` by double-clicking the Delay block, the host will call this `createEditor` function.

The expression `new DelayAudioProcessorEditor` creates a new instance of the editor.

The constructor for `DelayAudioProcessorEditor` takes one argument, a reference to the audio processor object. The expression `*this` creates such a reference. This way, the editor can always access the `DelayAudioProcessor` object.

Now let's look at **PluginEditor.cpp** in more detail. It looks like the following, without the comments.

```

#include "PluginProcessor.h"
#include "PluginEditor.h"

DelayAudioProcessorEditor::DelayAudioProcessorEditor (DelayAudioProcessor& p)
    : AudioProcessorEditor (&p), audioProcessor (p)
{
    setSize (400, 300);
}

DelayAudioProcessorEditor::~DelayAudioProcessorEditor()
{
}

void DelayAudioProcessorEditor::paint (juce::Graphics& g)
{
    // ...drawing code...
}

void DelayAudioProcessorEditor::resized()
{
}

```

Apart from the drawing code in `paint()`, there's not much there yet.

The constructor has a member initializer list, which if you'll recall is everything after the `:` colon. The initializer list is used to give member variables an initial value, before the code from the constructor is performed.

Here the member initializer list contains two items:

- `AudioProcessorEditor`. This is the constructor for the base class, which is the class we inherit from. This class is provided by JUCE but the `juce::` prefix isn't necessary here.
- `audioProcessor`. This is a private member variable (see `PluginEditor.h`). It stores the reference to the `DelayAudioProcessor` object.

The constructor also has a single line of code that sets the dimensions of the editor window to 400 by 300 pixels.

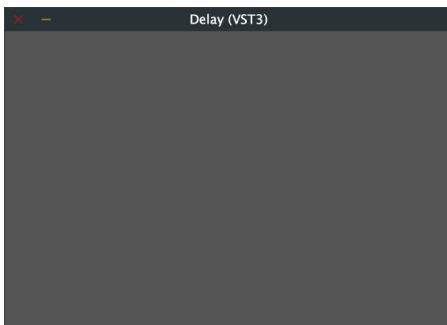
Let's already make a few changes here. In the constructor change `setSize()` to the following to make the editor slightly larger:

```
setSize(500, 330);
```

Replace the `paint` function with a single line that fills up the background with a gray color:

```
void DelayAudioProcessorEditor::paint (juce::Graphics& g)
{
    g.fillAll(juce::Colours::darkgrey);
}
```

Build and run and you should have an editor with a dark gray background:



An empty user interface

By the way, the `JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR` line at the bottom of the class in `PluginEditor.h`, which you've also seen in `PluginProcessor.h`, is a so-called **C++ preprocessor macro**. Usually such macros are spelled in uppercase.

This particular macro adds a few useful features to the class, namely it prevents the class from being copied and it adds leak detection.

In addition to the regular constructor that is used when creating an instance of the class, C++ classes have a **copy constructor** and **copy assignment operator**. Those are used when making a copy of the object. For the editor (and audio processor) object, we always want there to be just one unique instance and so copying it makes no sense.

The macro inserts some code into the header file that deletes the copy constructor and copy assignment operator from the class. Now if you were to write code that makes a copy — perhaps by mistake — the C++ compiler flags this as an error.

The **leak detector** is useful for preventing so-called memory leaks. When an instance of a class is created, some room is allocated in the computer's memory to store all its variables. This memory must be freed, or deallocated, when the instance is no

longer needed, for example if the editor window is closed or the user removes the plug-in from a track. If this deallocation doesn't happen, there is a memory leak. If a program has too many memory leaks, it may run out of available memory and crash.

For the audio processor and editor, you don't have to worry about deallocating them as JUCE will handle this for you, but other objects may require more care. The JUCE leak detector is there to tell you when you've missed such a deallocation. This won't be an issue for our plug-in, but for larger projects managing memory can get complex and it will be useful to have such a leak detector tool.

You can also add `JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(Parameters)` to the `Parameters` class. It's a good idea to add it to all your own classes, just in case.

Adding a rotary knob

Most plug-ins have rotary knobs (or dials) for their parameters, so that's what we will have too. Let's start with the **Output Gain** knob.

In JUCE, knobs are instances of the `juce::Slider` class. This class is also used for the sliders from the generic editor, which is why it's named `Slider` and not `Knob`. The way JUCE sees it, a knob is a round slider.

In `PluginEditor.h`, add a new variable in the `private:` section of the class:

```
juce::Slider slider;
```

This tells the editor we're making a `juce::Slider` object named `slider`.

In `PluginEditor.cpp`, in the constructor for `DelayAudioProcessorEditor`, add the following lines. Make sure you put these *before* the `setSize` line.

```
slider.setSliderStyle(juce::Slider::SliderStyle::RotaryHorizontalVerticalDrag);
slider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 70, 16);
slider.setBounds(0, 0, 70, 86);
addAndMakeVisible(slider);
```

This configures the `slider` object to be a rotary knob with a text box below the slider. There are many possible slider styles. `RotaryHorizontalVerticalDrag` means the knob is round and responds to both horizontal and vertical mouse movements but not circular motion. On desktop computers, dragging the mouse up-down or left-

right is the preferred way to interact with knobs. On mobile devices, making a circular motion with your fingers is more natural.

The knob itself will be 70×70 pixels. The text box is 16 pixels high. Since the `juce::Slider` object draws both the knob and text box, its combined size is 70 pixels wide by 86 pixels high.

`addAndMakeVisible` adds the `Slider` object to the editor, so that it will be drawn when the editor is painted. This makes the `slider` a **child component** of the editor.

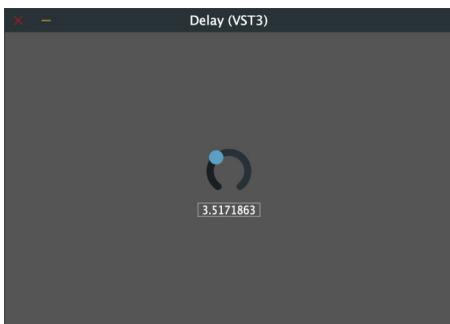
The call to `slider.setBounds` placed the slider in the very top-left corner of the editor, at coordinate (0, 0). Let's move it to a slightly different position, which happens in `resized`. This function is supposed to do the layout of all the components.

```
void DelayAudioProcessorEditor::resized()
{
    slider.setTopLeftPosition(215, 120);
}
```

Here you've moved the slider to position (215, 120). The first number is the x-coordinate, or the horizontal position. The second number is the y-coordinate, or the vertical position. In JUCE, the coordinates of UI components are integers or whole numbers. (0, 0) is in the top-left corner.

Try it out. You now have a single rotary knob in the middle of the editor window. It doesn't look very pretty yet but we will fix that soon enough.

You can click the knob and drag the mouse vertically or horizontally to turn it. You can also use the mouse wheel while the mouse pointer is over the knob, or type a value into the text box. If you've ever used any plug-in before, this should all feel natural.



The rotary knob in the middle of the editor window

If the knob is not in the middle of the window but in the top-left corner, make sure `setSize(500, 330);` is the last thing that happens in the constructor.

Note that you didn't need to draw the `slider` object in the `paint` function. JUCE will first call `paint` on the editor itself so it can draw the background, and then it calls `paint` on all the components that are part of the editor — the child components — including `slider`.

There are a few problems with this rotary knob:

1. It does not have label so we don't know what parameter it is for.
2. It is not connected to any parameter yet, so turning the knob has no effect. It currently goes between the values 0 and 10.
3. It's ugly.

Before we get to styling the look of the UI, let's first add the missing pieces to the rotary knob.

Adding a label

It's useful to add a label that indicates what this knob controls. To draw text on the screen we can use `g.drawFittedText()` in the `paint` function, as we did before. But we can also create a `juce::Label` component that will do the drawing for us.

In **PluginEditor.h** add a new variable in the `private:` section, below the `slider`:

```
juce::Label label;
```

Then in **PluginEditor.cpp** in the constructor, add the following lines. Again this must be done before `setSize`:

```
label.setText("Gain", juce::NotificationType::dontSendNotification);
label.setJustificationType(juce::Justification::horizontallyCentred);
label.setBorderSize(juce::BorderSize<int>{ 0, 0, 2, 0 });
label.attachToComponent(&slider, false);
addAndMakeVisible(label);
```

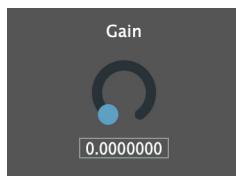
This sets the text string "Gain" on the label. The `dontSendNotification` argument means that `setText` won't send a notification to any listeners as the text changes. Listeners are an advanced feature that we're not using, so you can ignore this.

The label will be attached to the top of the `slider` component, and is horizontally centered over the knob (the justification type). `setBorderSize` puts 2 pixels of extra spacing between the bottom of the label and the top of the knob.

We do `addAndMakeVisible` to actually add the label component to the editor.

There's no need to manually position the label in `resized()`. Because we did `attachToComponent`, it will always automatically put itself on top of the slider component, wherever that may be located.

Try it out. The editor now looks like this:



The knob has a label

Even though it says Gain, the knob still isn't hooked up to the **Output Gain** parameter, so let's do that next.

By the way, there is no reason that the parameter's name, here **Output Gain**, needs to correspond with the label shown on the screen. Later in this chapter you'll put the **Gain** knob inside a panel that says **Output**, so it should be immediately clear to the user that this knob handles the output gain, as opposed to some other gain.

Connecting the slider to the parameter

One of the benefits of using the APVTS for managing the plug-in parameters, is that this makes it really easy to associate the parameter with a UI component. All you have to do is create an **attachment** object and hook it up to both the parameter and the component, and voila, you're done.

In **PluginEditor.h**, add this to the top of the file, below the other `#include` lines:

```
#include "Parameters.h"
```

This will import the definition of the `Parameters` class and everything else from the `Parameters.h` file, so that the code in `PluginEditor.h` knows what we mean when we write `gainParamID` next.

In the `private:` section of `DelayAudioProcessorEditor` add the following lines:

```
juce::AudioProcessorValueTreeState::SliderAttachment attachment {
    audioProcessor.apvts, gainParamID.getParamID(), slider
};
```

This creates a `SliderAttachment` object named `attachment`.

The `SliderAttachment` class is part of the APVTS, which is why it has the very long type name `juce::AudioProcessorValueTreeState::SliderAttachment`.

Important! This attachment code must be placed *below* the `juce::Slider slider;` line. When the editor window is closed, the `DelayAudioProcessorEditor` class is deallocated so that the memory it used is returned to the operating system. In doing so, its member variables are deallocated from bottom-to-top. The `attachment` variable must be deallocated before the `slider` or JUCE will give an error, so it must be closer to the bottom of the class definition.

The things inside the `{ }` braces are the arguments for the `SliderAttachment` constructor:

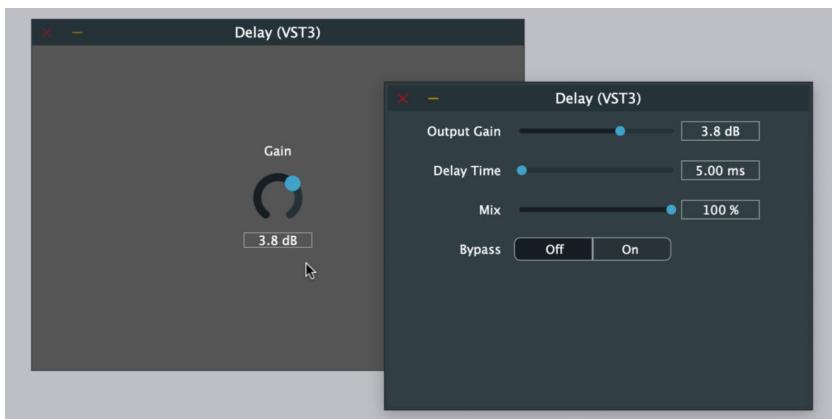
- The first argument is a reference to the APVTS, which is owned by the `DelayAudioProcessor`. Fortunately, the editor can use the member variable `audioProcessor` to access that class.
- The second argument is the parameter ID, which we imported from `Parameters.h`.
- The final argument is the `slider` object.

So, this declaration says: Create a new attachment between the parameter with the identifier `gainParamID` and the `slider` object named `slider`.

There is a small problem: `apvts` is a private member of `DelayAudioProcessor`, so we cannot directly access it from here. In `PluginProcessor.h`, move the lines that create the APVTS object into the `public:` section of the class. Now it can be accessed by anyone who has a reference to the `DelayAudioProcessor`.

Try it out! Right-click the **Delay** block in AudioPluginHost and choose **Show all parameters** to bring up the generic editor. Put this side-by-side with the actual editor. Rotating the **Gain** knob in the editor will update the **Output Gain** slider, and vice versa.

You can also type into the text box and both the knob and slider will update. If the value you type is larger than 12, or smaller than -12, the knob will automatically clamp it to the allowed range. Note that we never wrote code to handle text input. JUCE automatically takes care of this for us.



The knob is connected to the plug-in parameter

Holding down Alt or Option and clicking on the knob resets it to its default value; likewise for double-clicking it. Holding down Ctrl while dragging the knob lets you make fine-grained adjustments.

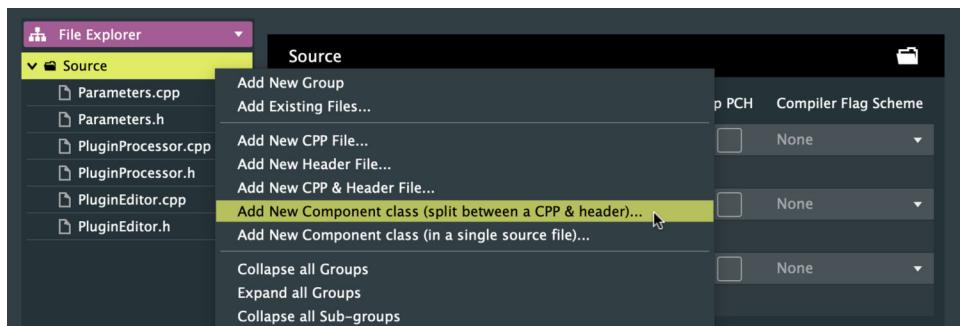
If you want to test this in a DAW, make sure to set some automation on the track. The DAW knows that the **Output Gain** parameter can be automated, and the knob's position will follow that of the automation. Nice!

Making a **RotaryKnob** class

What we did works fine, but the plug-in has more than one parameter. In fact, by the end of the book it will have 10 parameters. You could copy-paste the code for the **Output Gain** parameter's knob, label, and attachment for all other parameters, but that would end up being a lot of code.

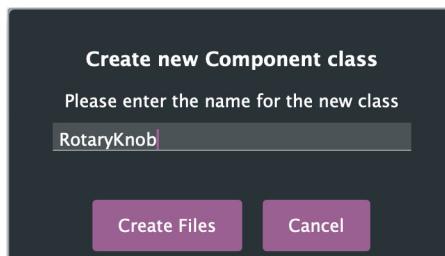
It's smarter to put the logic that we just wrote into a new class, `RotaryKnob`, that we can re-use for all parameters. That will make the code in `PluginEditor.cpp` a little cleaner.

Since we're going to add new source files to the project, fire up our old friend **Projucer** and load the `Delay.jucer` project. Go to the **File Explorer** and right-click on the **Source** group. From the pop-up menu choose **Add New Component class (split between a CPP & header)**...



Adding a new component in Projucer

A dialog appears asking for the name of the component. Type in **RotaryKnob**. Then press **Create Files** and save the file along with the project's other source files.



Naming the new component class

Note that Projucer does not automatically sort the files by name, so right-click the **Source** folder again and choose **Sort Items Alphabetically** if you care about that sort of thing (that was a pun). Save the project and open it in your IDE.

Let's look at the contents of the new `RotaryKnob.h` file. It looks like the following.

```
#pragma once

#include <JuceHeader.h>

class RotaryKnob : public juce::Component
{
public:
    RotaryKnob();
    ~RotaryKnob() override;

    void paint (juce::Graphics&) override;
    void resized() override;

private:
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (RotaryKnob)
};
```

I skipped the comments for brevity, and in fact I generally remove these lines by hand after making the source files. Anything starting with // or between /* and */ characters is a comment.

This code looks a lot like **PluginEditor.h**, except that the new **RotaryKnob** class is based on **juce::Component** instead of **juce::AudioProcessorEditor**, but it has the same **paint** and **resized** functions. This is not so strange, as **juce::AudioProcessorEditor** is itself based on **juce::Component** too.

Let's have a quick look at **RotaryKnob.cpp**. It has placeholders for the various functions that are mostly empty. The **paint** function has some drawing code in it.

For now, simply remove the **paint** function from **RotaryKnob** altogether. Remove it from both **RotaryKnob.h** and **RotaryKnob.cpp**.

When a **paint** function isn't used, the background of the component is transparent, meaning that anything that was drawn below it will shine through. For knobs this is usually what you want, especially if there's a background image drawn behind all the knobs (which is what we'll do).

Back in **RotaryKnob.h**, add the variables that will hold the slider and the label component. These go in the **public:** section of the class so we can easily access them from the editor object.

```
juce::Slider slider;
juce::Label label;
```

Go to **PluginEditor.h** and add an include at the top, below the other includes.

```
#include "RotaryKnob.h"
```

Replace these two lines in the `private:` section of the editor class,

```
juce::Slider slider;
juce::Label label;
```

with this single line:

```
RotaryKnob gainKnob;
```

You'll also need to change the attachment declaration to use `gainKnob.slider` instead of just `slider`:

```
juce::AudioProcessorValueTreeState::SliderAttachment attachment {
    audioProcessor.apvts, gainParamID.getParamID(), gainKnob.slider
};
```

In **PluginEditor.cpp** in the constructor, cut out all the lines except for `setSize(...)` and paste them into the constructor in **RotaryKnob.cpp**:

```
RotaryKnob::RotaryKnob()
{
    slider.setSliderStyle(juce::Slider::SliderStyle::RotaryHorizontalVerticalDrag);
    slider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 70, 16);
    slider.setBounds(0, 0, 70, 86);
    addAndMakeVisible(slider);

    label.setText("Gain", juce::NotificationType::dontSendNotification);
    label.setJustificationType(juce::Justification::horizontallyCentred);
    label.setBorderSize(juce::BorderSize<int>(0));
    label.attachToComponent(&slider, false);
    addAndMakeVisible(label);

    setSize(70, 110);
}
```

We've added a `setSize` call at the bottom. Without this line, the `RotaryKnob` has a size of zero and it won't show up. We're making it large enough to fit the slider, which is 70×86 pixels, plus the label that goes on top, which is 24 pixels high, so the total height of the `RotaryKnob` is $86 + 24 = 110$ pixels.

You must also implement the `resized` function, to move the slider down so that the label will become visible, like so:

```
void RotaryKnob::resized()
{
    slider.setTopLeftPosition(0, 24);
}
```

The reason for this is that child components cannot draw outside the bounds of their parent component. Here, `slider` and `label` are the child components and `RotaryKnob` is the parent.

The label is attached to the slider and will always try to position itself at the top of the slider. If the y-coordinate of the slider is 0, the label will have a negative y-coordinate relative to the rectangle that the `RotaryKnob` is in, and therefore it won't be visible. By moving the slider down by 24 pixels, the label will fit inside `RotaryKnob`'s bounds.

Back in `PluginEditor.cpp`, change the constructor to the following. All we have to do here is add the `gainKnob` to the editor component.

```
DelayAudioProcessorEditor::DelayAudioProcessorEditor (DelayAudioProcessor& p)
    : AudioProcessorEditor (&p), audioProcessor (p)
{
    addAndMakeVisible(gainKnob); // add this line
    setSize(500, 330);
}
```

Finally, in `resized()` we need to use the `gainKnob` variable instead of `slider`:

```
void DelayAudioProcessorEditor::resized()
{
    gainKnob.setTopLeftPosition(215, 120);
}
```

Phew, that was a bit of refactoring but the `juce::Slider` and `juce::Label` components are now safely tucked away in a new component that we made ourselves, `RotaryKnob`.

Build and run the plug-in to verify that the knob and label are still in the correct place.

The text that goes on the knob is hardcoded to be "Gain" in the `RotaryKnob` constructor. We can make this more flexible by adding an argument to the constructor.

First change `RotaryKnob.h` to declare the constructor will take a `juce::String` object as an argument:

```
class RotaryKnob : public juce::Component
{
public:
    RotaryKnob(const juce::String& text); // change this line
    ~RotaryKnob() override;

    // ... and so on ...
}
```

The & symbol means we're passing the `juce::String` object by reference. Without the & it would still work but the `juce::String` would be copied, which is less optimal and not needed in this case.

In general, when passing an object as an argument to a function you want to use & to make it a reference. Go look through all the code we've written so far and you'll see that indeed the & makes an appearance almost everywhere an object is used as a function argument (as opposed to an `int` or `float`).

We also declare the argument to be `const`, which means that the `RotaryKnob` constructor promises not to try and change the `juce::String` object, for example by adding new text to it. Using `const` is always a good idea if you know you're not going to attempt to modify an object, as it allows the C++ compiler to generate more optimal machine code. Plus, it documents to the programmer (you!) what to expect.

Since this argument is `const juce::String&` we know it will be passed by reference and the object will not be modified.

In `RotaryKnob.cpp`, change the constructor implementation to the following:

```
RotaryKnob::RotaryKnob(const juce::String& text) // add the argument
{
    // ... slider stuff ...

    label.setText(text, juce::NotificationType::dontSendNotification);
    // ... rest of the label stuff ...
}
```

Instead of doing `label.setText("Gain", ...)` this uses the value of the `text` string from the argument.

We'll have to give the `RotaryKnob` some text to display where it's being created. In `PluginEditor.h`, replace the line that declares the `gainKnob` variable with:

```
RotaryKnob gainKnob { "Gain" };
```

Previously the `RotaryKnob` constructor did not take any arguments, so we didn't need the `{ }`. Now it does take an argument and we must provide it or the compiler will give an error.

Try it out to verify everything works as expected.

To complete the code migration to `RotaryKnob`, we'll also put the attachment object in there. In `RotaryKnob.h`, add the following in the `public:` section. Again, make sure this line is somewhere below the `juce::Slider` line.

```
juce::AudioProcessorValueTreeState::SliderAttachment attachment;
```

We can't initialize the `SliderAttachment` object in the header file like we did before. Instead, we'll have to add some arguments to the `RotaryKnob` constructor. Still in `RotaryKnob.h`, change the constructor declaration to add two additional arguments. To make it fit in the book, I split it into multiple lines.

```
RotaryKnob(const juce::String& text,
           juce::AudioProcessorValueTreeState& apvts,
           const juce::ParameterID& parameterID);
```

Notice that the APVTS object is a reference but is not `const`. We're going to use these two new arguments to initialize the `SliderAttachment` object. The constructor of `SliderAttachment` takes a reference to the APVTS that is not `const`. That's why we cannot declare this argument as `const` here.

In `RotaryKnob.cpp`, change the implementation of the constructor to the following:

```
RotaryKnob::RotaryKnob(const juce::String& text,
                      juce::AudioProcessorValueTreeState& apvts,
                      const juce::ParameterID& parameterID)
: attachment(apvts, parameterID.getParamID(), slider)
{
    // ... code is unchanged ...
}
```

This uses the member initializer list to call the constructor for the `SliderAttachment` object, and passes in the APVTS reference, the parameter ID, and a reference to the `juce::Slider`.

It does the exact same thing you did before in `PluginEditor.h`, but using a slightly different method. C++ sometimes lets you do the same thing in a variety of ways, and here using the member initializer list is more convenient.

That's it for RotaryKnob but we still need to update `PluginEditor.h`. Change the declaration of the knob to pass in these two new arguments:

```
RotaryKnob gainKnob { "Gain", audioProcessor.apvts, gainParamID };
```

Also remove the old attachment variable from the `DelayAudioProcessorEditor` class, since this is no longer used.

Try it out to verify everything works as before. Nice!

Adding the other knobs

One parameter down, two more to go. Let's add knobs for the **Mix** and **Delay Time** parameters too. This should be easy now.

In `PluginEditor.h`, add the following two lines below the `gainKnob` variable:

```
RotaryKnob mixKnob { "Mix", audioProcessor.apvts, mixParamID };
RotaryKnob delayTimeKnob { "Time", audioProcessor.apvts, delayTimeParamID };
```

In `PluginEditor.cpp` in the constructor, add the two new knobs to the editor:

```
addAndMakeVisible(mixKnob);
addAndMakeVisible(delayTimeKnob);
```

As always, make sure `setSize(...)` is the last line in the constructor.

We need to position these knobs in the editor window. That happens in the `resized` function, which currently puts the gain knob in the middle. Rewrite this function to do the following.

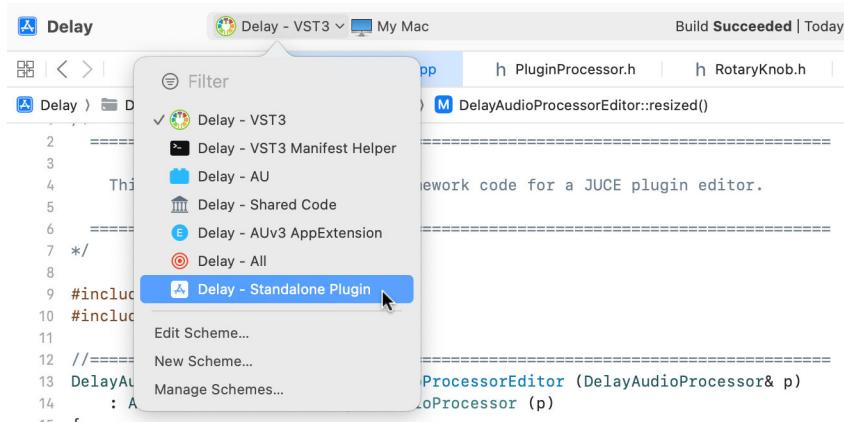
```
void DelayAudioProcessorEditor::resized()
{
    delayTimeKnob.setTopLeftPosition(20, 10);
    mixKnob.setTopLeftPosition(delayTimeKnob.getRight() + 20, 10);
    gainKnob.setTopLeftPosition(mixKnob.getRight() + 20, 10);
}
```

This puts the **Time** knob in the top-left corner of the editor, with a small margin to the left (20 pixels) and top (10 pixels). The **Mix** knob goes next to it, 20 pixels from the right edge of the Time knob. The **Gain** knob sits to the right of the Mix knob.

Try it out to see what it looks like.

Tip: Since we're currently only constructing the UI for the plug-in, not writing any new audio code, you don't need to load it into AudioPluginHost for testing. The project also includes a **Standalone Plugin** target that embeds the plug-in into a self-contained app.

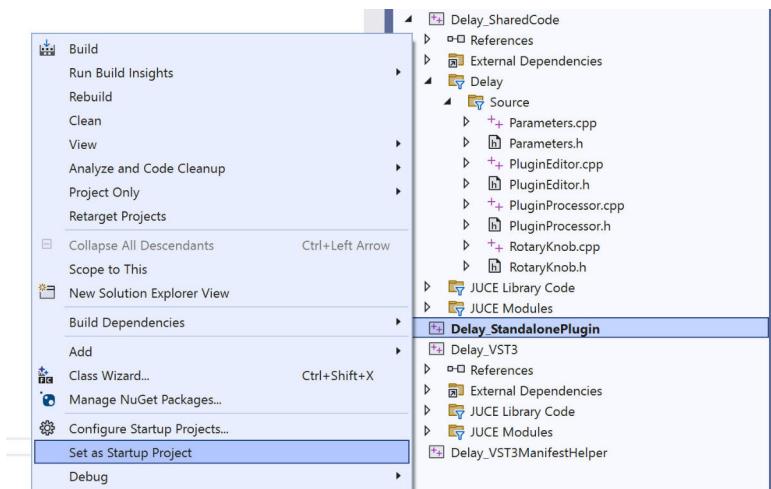
For Xcode: In the bar at the top of the Xcode window, select the **Delay - Standalone Plugin** target.



Choosing the Standalone Plugin target in Xcode

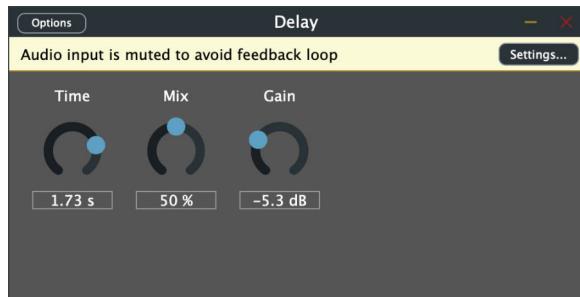
If you don't see this target in the popup, close Xcode, delete the **Builds** and **JuceLibraryCode** folders, and export from Projucer again. Sometimes this target is missing when you export (perhaps this is a bug in Projucer).

For Visual Studio: In the **Solution Explorer**, right-click **Delay_StandalonePlugin** and choose **Set as Startup Project** from the pop-up menu.



Making the Standalone Plugin the startup project in Visual Studio

Build and run and the following will show up:



The plug-in running as a standalone app

This app will read incoming sound from the currently selected audio input, send that to the plug-in's `processBlock`, and play the results on the selected audio output. You can change these inputs and outputs by clicking the **Settings...** button.

Notice the yellow bar at the top. The reason the input is muted is to avoid screaming feedback in case you're using a microphone as input that will pick up the output from the speakers. Running with the audio muted is fine for our purposes, as we're only using the Standalone Plugin target to test out the UI and we don't need to hear the audio anyway.

From now on I suggest using this Standalone Plugin target to test out the editor as it's a little faster to load than `AudioPluginHost`.

Grouping the knobs

The finished plug-in will have three distinct stages:

1. the delay
2. a feedback section
3. the output stage

There will be several knobs in each stage. It makes sense to visually separate these stages in the user interface, so that related knobs are grouped together. We can do this by adding the knobs into a `juce::GroupComponent`.

In `PluginEditor.h`, add this line to the `private:` section of the class:

```
juce::GroupComponent delayGroup, feedbackGroup, outputGroup;
```

This declares three new `juce::GroupComponent` objects in one go. Now all we need to do is add our knobs to the right group, and position everything on screen in `resized()`.

In `PluginEditor.cpp`, change the code inside the constructor to:

```
delayGroup.setText("Delay");
delayGroup.setTextLabelPosition(juce::Justification::horizontallyCentred);
delayGroup.addAndMakeVisible(delayTimeKnob);
addAndMakeVisible(delayGroup);

feedbackGroup.setText("Feedback");
feedbackGroup.setTextLabelPosition(juce::Justification::horizontallyCentred);
addAndMakeVisible(feedbackGroup);

outputGroup.setText("Output");
outputGroup.setTextLabelPosition(juce::Justification::horizontallyCentred);
outputGroup.addAndMakeVisible(gainKnob);
outputGroup.addAndMakeVisible(mixKnob);
addAndMakeVisible(outputGroup);

setSize(500, 330);
```

The knobs are no longer added to the editor but to their respective group component. The group components are then added to the editor. Each group has a title label that is horizontally centered above the group.

Rewrite the `resized` function as follows:

```
void DelayAudioProcessorEditor::resized()
{
    auto bounds = getLocalBounds();

    int y = 10;
    int height = bounds.getHeight() - 20;

    // Position the groups
    delayGroup.setBounds(10, y, 110, height);

    outputGroup.setBounds(bounds.getWidth() - 160, y, 150, height);

    feedbackGroup.setBounds(delayGroup.getRight() + 10, y,
                           outputGroup.getX() - delayGroup.getRight() - 20,
                           height);

    // Position the knobs inside the groups
    delayTimeKnob.setTopLeftPosition(20, 20);
    mixKnob.setTopLeftPosition(20, 20);
    gainKnob.setTopLeftPosition(mixKnob.getX(), mixKnob.getBottom() + 10);
}
```

First this gets the **bounds** of the component and puts it into a `juce::Rectangle<int>` variable named `bounds` (what else?). The bounds describe where the component is positioned and how wide and tall it is.

Inside `resized()` we'll usually be looking at the **local bounds**, which are relative to the component itself, and so the (x, y) coordinate of this rectangle is always (0, 0).

Note that the code never actually says `juce::Rectangle<int>` anywhere. Instead, it says `auto bounds`. Since the compiler can figure out that `getLocalBounds()` returns a `juce::Rectangle<int>`, writing `auto` saves you from having to type in the entire thing. The `auto` keyword can be used anywhere the compiler can infer what the data type should be and is very convenient.

Once we know how large the editor component is, we can position the child components. We create two helper variables, `y` and `height`, since these will be the same for all groups.

`y` is the y-coordinate for the group components, 10 pixels from the top of the editor. We want the components to be 10 pixels from the bottom of the editor too, so their height is `bounds.getHeight()` minus 20 pixels.

Note: bounds, y, and height are **local variables**. These variables only exist inside the resized function. This is different from the other variables such as delayGroup and delayTimeKnob — these **member variables** belong to the object and exist for as long as the editor is open.

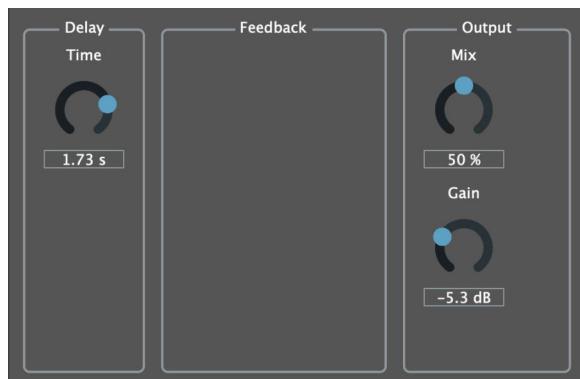
We move the group components into place by calling the `setBounds` function on them. This takes four arguments: x, y, width, height.

- The **Delay** group is placed in the top-left corner, 10 pixels away from the edges, and is 110 pixels wide.
- The **Output** group is placed on the right-hand side of the editor. Its x-coordinate is `bounds.getWidth() - 160`, making its width 150 pixels with a 10-pixel space to the edge of the editor.
- The **Feedback** group is wedged in between the other two groups. It covers the area between the Delay group's right edge and the Output group's left edge, with 10 pixels of padding on each side.

After setting the positions and sizes of the group components, we move the knobs into place. For the knobs we use `setTopLeftPosition` instead of `setBounds` because the width and height are already correct.

Note that we set the bounds of the group components in the coordinate system of the editor, but the positions of the knobs are relative to the bounds of the group they belong to.

It's already starting to resemble an actual plug-in UI:

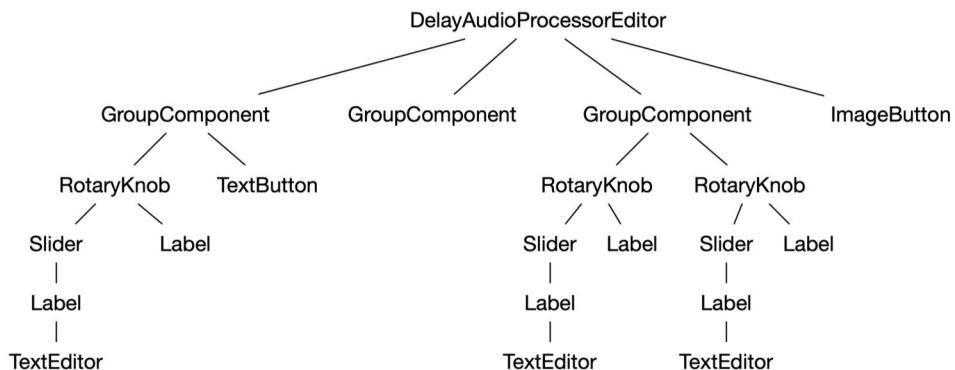


The editor with knobs

The **Feedback** group is currently empty since we didn't add parameters or knobs for this yet. The **Output** group is a little wider than the knobs; this leaves some room for a level meter. In the next chapter we'll give this UI a new coat of paint.

As you have seen, the UI in a JUCE plug-in editor is made up of components inside of components. Together, these components form a hierarchy.

The `DelayAudioProcessorEditor` is the root of this hierarchy. Below that are the `GroupComponents`. These contain `RotaryKnob` components, which in turn have a `Label` and a `Slider` as child components. The `Slider` itself has child components such as a `Label` that draws the knob's value and a `TextEditor` that is used when you're typing text into the text box. Later we'll also add buttons to the editor.



The UI is a hierarchy of components containing other components

To learn more about what UI components are available in JUCE, I remind you of the **DemoRunner** app. It has a **GUI > WidgetsDemo** that shows all the built-in user interface components that JUCE offers. There are many more than what we've looked at in this chapter.

Note: Just to keep this book a manageable length, I have hardcoded the sizes of the groups and the knobs. This means the user interface is not resizable. There are various ways to make the editor resizable, for example by dividing the total width of the parent component across the child components. Consider this an exercise for the user. To make the plug-in resizable, add the following below `setSize(...)` in the constructor: `setResizable(true, true);` There is also a `setResizeLimits` function that lets you specify the minimum and maximum sizes the window can have.

Value-from-string functions

To display the parameter value in the text box under the knob, JUCE uses the string-from-value function that was provided to the `AudioParameterFloat` object when we constructed it. The **Delay Time** knob, for example, uses the function `stringFromMilliseconds()` from `Parameters.cpp`.

When the user types something into the edit box, JUCE uses a related function, the **value-from-string** function, to convert the text back into a `float` number.

This conversion is really simple: It tries to read the text from left-to-right until it finds a character that isn't a number or decimal point and then stops parsing. For instance, the text string "123.4 ms" is read as "123.4", which is then turned into the `float` value 123.4. The "ms" portion of the text is ignored.

That logic is sufficient in most cases. However, when the delay time is more than 1000 milliseconds, the `stringFromMilliseconds()` function changes it to seconds, such as "3.45 s". It would be convenient if the user could type in a value in seconds too, by adding the "s" suffix.

Right now, the text "3.45 s" is interpreted as "3.45", which becomes the float 3.45. Since this is less than the minimum delay time of 5 ms, the knob will jump to its minimum position and the delay time is 5 ms. Instead, what we would like to see happen is that the text "3.45 s" becomes the float 3450.0f, the same value but expressed in milliseconds.

You can implement such custom text parsing logic by providing a custom value-from-string function. Go to `Parameters.cpp` and add the following code below the existing `stringFromMilliseconds()` function:

```
static float millisecondsFromString(const juce::String& text)
{
    // 1
    float value = text.getFloatValue();
    // 2
    if (!text.endsWithIgnoreCase("ms")) {
        // 3
        if (text.endsWithIgnoreCase("s") || value < Parameters::minDelayTime) {
            return value * 1000.0f;
        }
    }
    return value;
}
```

The argument to this function is a `juce::String` reference containing the text to interpret. The return value is a floating-point number.

How this works:

1. Convert the text to a `float` value using `text.getFloatValue()`. This is what the standard value-from-string function does too. But we won't stop here.
2. If the text does *not* end with "ms", then the code inside the `if` statement gets performed. In other words, if the string ends with "ms" then we're done and skip to the `return` statement at the end of the function.

The `!` checks whether something is *not* true. If `text.endsWithIgnoreCase("ms")` is not true — in other words, `false` — we continue with the code in step 3. Confused? It can take a second to wrap your head around negative logic like this. By the way, the `IgnoreCase` means that "MS" is also allowed.

3. Check if the string ends with "s". If so, multiply by `1000.0f` to convert the amount in seconds to milliseconds.

Or... if the suffix is not "s" or "ms" but the value is less than the minimum delay time, for example 3.45, then it's likely the user intended seconds. In this case also multiply by `1000.0f`.

We can check both these things at the same time using the `||` operator, which stands for **or**. If the first thing is true *or* the second thing is true, then the code inside the `{ }` brackets will be run.

To use this value-from-string function with the parameter, in `createParameterLayout`, change the code that adds the **Delay Time** parameter to do:

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    delayTimeParamID,
    "Delay Time",
    juce::NormalisableRange<float> { minDelayTime, maxDelayTime, 0.001f, 0.25f },
    100.0f,
    juce::AudioParameterFloatAttributes()
        .withStringFromValueFunction(stringFromMilliseconds)
        .WithValueFromStringFunction(millisecondsFromString) // add this
));
```

Give it a try! Type in a value like "2.8 s" or just "2.8" and the **Delay Time** parameter should properly interpret this as 2800 milliseconds.

The logic for parsing the text is rather simple. A nonsense input like "1 ms s" is valid and interpreted as being one second, since the string ends with "s". I'm personally OK with this — most users won't type in something silly like that and the plug-in will behave reliably even if the input doesn't make much sense.

For the **Mix** and **Output Gain** parameters we don't need a custom value-from-string function. With the default logic, any text following the number is ignored, so if the user types in "-6" or "-6.0" or "-6.0 dB" these all get interpreted as -6 decibels.

It took a bit of work but the editor is fully functional now. However, it still uses the default JUCE look-and-feel. In the next chapter you'll make the user interface look a lot better.

Note: JUCE 8 also allows you to build the editor UI using web technology, such as HTML, CSS and JavaScript, instead of doing everything with C++ like we've done in this chapter. Using web technologies has a few advantages: Designing is faster since the UI can be hot reloaded rather than having to be recompiled after every change. You can also use web frameworks such as React, or even hire a web developer to build the plug-in's UI. We don't use this new feature of JUCE 8 in this book, as I can't assume you're already familiar with HTML, CSS, and JavaScript, and I don't want to teach three other languages in addition to C++.

11: Styling the editor

I think we can all agree that the standard `juce::Slider` does not look very pretty. Fortunately, the JUCE designers made it possible to style the slider, as well as the other components, by implementing what's known as a **look-and-feel**. This is a separate object that performs the drawing of the components.

To customize how the knobs get drawn, we could make our own `CustomSlider` class that is based on `juce::Slider` and give it a new `paint` function. But JUCE makes this easier — we only need to create a `LookAndFeel` class that has a function `drawRotarySlider` and then any `juce::Slider` will automatically draw using this custom style.

When you're done with this chapter, the plug-in's editor will look like this:



The editor with a custom look-and-feel

Note: There is going to be a lot of code in this chapter. Feel free to copy-paste this code from the book's resources, as it can be a little tedious to type in yourself. The JUCE look-and-feel system is quite powerful but it's also complicated at times and so don't feel discouraged if this all seems a bit much. As long as you're getting a sense of how this works, that's all I am hoping for.

Defining the colors

Go to **Projucer** and add a new source file to the project using **Add New CPP & Header File...**. Name it **LookAndFeel**. Save the project (**File > Save Project**) and open it in your IDE.

First we'll define some colors that will be used by the drawing code. Add the following code to **LookAndFeel.h**, below the `#pragma once` line.

```
#include <JuceHeader.h>

namespace Colors
{
    const juce::Colour background { 245, 240, 235 };
    const juce::Colour header { 40, 40, 40 };

    namespace Knob
    {
        const juce::Colour trackBackground { 205, 200, 195 };
        const juce::Colour trackActive { 177, 101, 135 };
        const juce::Colour outline { 255, 250, 245 };
        const juce::Colour gradientTop { 250, 245, 240 };
        const juce::Colour gradientBottom { 240, 235, 230 };
        const juce::Colour dial { 100, 100, 100 };
        const juce::Colour dropShadow { 195, 190, 185 };
        const juce::Colour label { 80, 80, 80 };
        const juce::Colour textBoxBackground { 80, 80, 80 };
        const juce::Colour value { 240, 240, 240 };
        const juce::Colour caret { 255, 255, 255 };
    }
}
```

This creates a namespace `Colors` that contains a number of `juce::Colour` objects.

So far you've used `juce::Colours` to select one of the predefined colors, such as `juce::Colours::darkgrey`. The `juce::Colour` class — without the “s” at the end — lets you define your own colors.

The constructor for a `juce::Colour` object takes three arguments: the values for the red, green, and blue color components. These are integers between 0 and 255.

We've put the colors into named constants, because writing `juce::Colour { 245, 240, 235 }` only tells you what the RGB values are but not what they are for. The name `Colors::background` makes it clear this color is intended to be for the background of the component.

By putting these constants in a namespace, we can simply refer to them as `Colors::background` if we want to get the background color, `Colors::Knob::label` to get the color for the knob's text label, and so on. Using namespaces is a good practice for organizing constants like this. By putting the RGB values in a single place, we can easily change the colors later.

Note: There is no need to put a ; behind the namespace's closing brace. This is different from a class, which does need the semicolon at the end. Also notice that namespaces can be **nested**, that is, a namespace can be placed inside another namespace, like what we're doing with `Colors::Knob`.

Go to **PluginEditor.h** and add an include for the new **LookAndFeel.h** header file, so that we can use these namespaces inside the editor code:

```
#include "LookAndFeel.h"
```

There are a number of these `#include` lines in **PluginEditor.h**. The order of these includes generally doesn't matter, as long as they are located at the top of the file. However, you do want to put `#include <JuceHeader.h>` as the first one.

Notice that for **JuceHeader.h** you're using `< >` and for the others "`" "`". The difference is subtle but for now all you need to remember is that for things like frameworks and libraries you need to write `#include <filename>` and for your own source files you write `#include "filename"`.

Next, go to **PluginEditor.cpp** and change the `paint` function to draw the new background color:

```
void DelayAudioProcessorEditor::paint (juce::Graphics& g)
{
    g.fillAll(Colors::background);
}
```

Build to make sure everything still compiles. The editor now has an off-white background. The text labels are hard to read since they are white, but we'll fix this shortly.



Using the new background color

Look-and-feel for the slider

We'll start by styling the slider's knob. For this we have to write a new class that we'll name `RotaryKnobLookAndFeel`.

Put the following code in `LookAndFeel.h`, below the `Colors` namespace.

```

// 1
class RotaryKnobLookAndFeel : public juce::LookAndFeel_V4
{
public:
    // 2
    RotaryKnobLookAndFeel();

    // 3
    static RotaryKnobLookAndFeel* get()
    {
        static RotaryKnobLookAndFeel instance;
        return &instance;
    }

    // 4
    void drawRotarySlider(juce::Graphics& g, int x, int y, int width, int height,
                           float sliderPos, float rotaryStartAngle,
                           float rotaryEndAngle, juce::Slider& slider) override;

private:
    // 5
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(RotaryKnobLookAndFeel)
};
```

This doesn't look too different from the other classes you've made before.

1. The `RotaryKnobLookAndFeel` class is based on the `juce::LookAndFeel_V4` class. There are several versions of the look-and-feels that ship with JUCE. The drab gray-greenish UI from V4 is the latest and that's what we'll base our own look-and-feel on.
2. The constructor. It takes no arguments.
3. This function lets us use the `RotaryKnobLookAndFeel` from anywhere in the code by writing `RotaryKnobLookAndFeel::get()`. It will return a pointer to a single shared instance of this class. That's simpler and more efficient than having to construct a new instance every time we want to use this look-and-feel.
4. The `drawRotarySlider()` function is what will draw the knob. As you can see it has a lot of arguments that are needed to properly paint the knob part of the `juce::Slider` object. This function is called from `juce::Slider`'s own `paint` function.
5. As usual there is the `JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR` macro that makes sure the object cannot be copied (no need for it) and does not create memory leaks.

Note: The `override` keyword is used when you're "overriding" the implementation of a function from the base class. In this case, `juce::LookAndFeel_V4` already has its own version of `drawRotarySlider`. But since we want to draw the knob differently, we tell the C++ compiler through the `override` keyword that we're going to be providing a new version of this function. You've already been doing the same for `processBlock` and `prepareToPlay` in the `DelayAudioProcessor`, and `paint` and `resized` in the editor too. Leaving out `override` is not an error but will give a compiler warning.

Open `LookAndFeel.cpp` and add the implementation for the constructor:

```
RotaryKnobLookAndFeel::RotaryKnobLookAndFeel()
{
    setColour(juce::Label::textColourId, Colors::Knob::label);
    setColour(juce::Slider::textBoxTextColourId, Colors::Knob::label);
}
```

The constructor changes the colors used to draw the labels in the rotary knob. The first `setColour` statement is for the label that goes above the knob, the second is for the text box below the knob.

JUCE is designed so it's possible to make themes for the UI. This is done by assigning colors to so-called **color IDs**. Here we say to any `juce::Label` components that the `textColourId` should use the color from `Colors::Knob::label`. That is one of the colors we defined in our namespace in `LookAndFeel.h`, a dark gray color.

The constructor also tells any `juce::Slider` components that `textBoxTextColourId` should use that same dark gray color. From now on, the `Label` and `Slider` components inside the `RotaryKnob` will draw their text with that new color.

Also implement the `drawRotarySlider` function, which you'll leave empty for now. You'll be adding a lot more code there soon.

```
void RotaryKnobLookAndFeel::drawRotarySlider(
    juce::Graphics& g,
    int x, int y, int width, int height,
    float sliderPos,
    float rotaryStartAngle, float rotaryEndAngle,
    juce::Slider& slider)
{
}
```

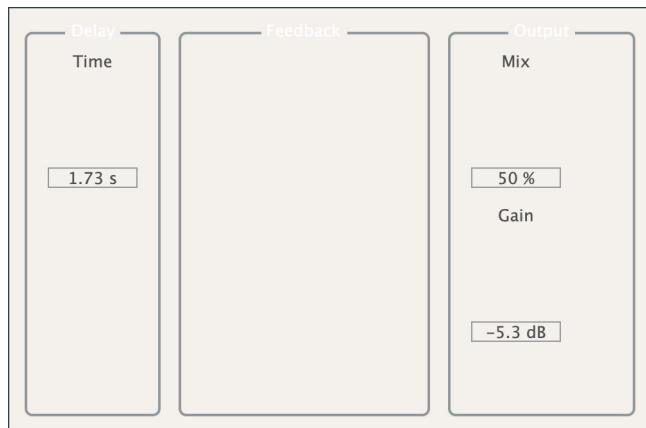
Go to **RotaryKnob.cpp**. At the top, add an include for the look-and-feel below the others:

```
#include "LookAndFeel.h"
```

Then in the `RotaryKnob` constructor, add the following line to set our own look-and-feel on the component:

```
setLookAndFeel(RotaryKnobLookAndFeel::get());
```

Build and run. There are a few new compiler warnings about unused arguments in the `drawRotarySlider` function but the compilation should succeed. The plug-in now looks like the following.



The labels are drawn in dark gray

The labels from the `RotaryKnob` component are no longer white, which is good, but all the knobs have disappeared! That makes sense because while we did override `drawRotarySlider`, we left the function empty — it doesn't have any drawing commands yet.

However, the knob still works... Try dragging up and down where the knob used to be. The values in the text box will change as usual. If you run the plug-in inside `AudioPluginHost`, you can hear that turning the knobs will change the plug-in parameter values as before.

Put another way: the behavior of the slider — how it responds to user input and changing parameters — is **decoupled** from how it gets drawn. This is why the same `juce::Slider` class is used for horizontal sliders, vertical sliders, and rotary knobs. They all behave the same way but get drawn by different functions from the look-and-feel object.

Note: The `RotaryKnob` component has two child components: the slider and the label. By doing `setLookAndFeel` on the `RotaryKnob` itself, it will also set the look-and-feel of its child components. And if they have child components of their own, here the `Label` for the slider's text box, those get that look-and-feel too. The colors of the labels in the group components remain white, since we did not change the look-and-feel of the `juce::GroupComponent` objects.

Drawing the knob

An editor with invisible knobs is a little hard to use, so let's write the knob drawing code. As there's a lot of code here, we're going to build it up step by step.

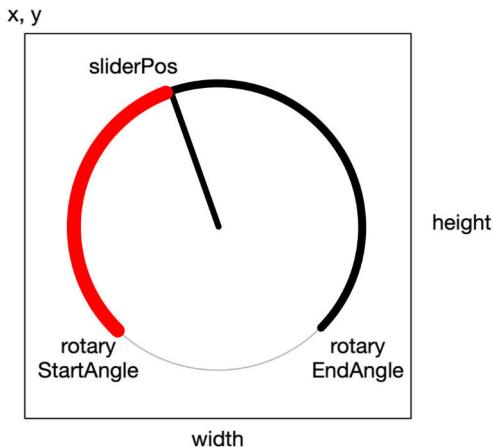
In `LookAndFeel.cpp`, change the `drawRotarySlider` code to:

```
void RotaryKnobLookAndFeel::drawRotarySlider(
    juce::Graphics& g,
    int x, int y, int width, [[maybe_unused]] int height,
    float sliderPos,
    float rotaryStartAngle, float rotaryEndAngle,
    juce::Slider& slider)
{
    auto bounds = juce::Rectangle<int>(x, y, width, width).toFloat();
    auto knobRect = bounds.reduced(10.0f, 10.0f);
    g.setColour(Colors::Knob::outline);
    g.fillEllipse(knobRect);
}
```

Notice that `drawRotarySlider` has arguments for:

- The `juce::Graphics` object. This is known as the **graphics context**. We will do all our drawing using this object.
- `x, y, width, height`: These are the bounds that the `juce::Slider` object should be drawn in. `x` and `y` describe the position of the top-left corner. These are all integer values.
- `sliderPos`: The current value of the slider, as a proportion of its length. This is a value between 0 and 1. It also takes into account the `skew` set on the corresponding parameter.
- `rotaryStartAngle, rotaryEndAngle`: The rotary knob has a starting angle and an ending angle, both measured in radians.
- The `juce::Slider` object that we're currently drawing. This is useful to have because we can inspect its properties and change how the slider is drawn. For example, if `slider.isEnabled()` is false, we could draw it looking disabled so it's clear to the user they can't interact with this knob.

The following illustration visualizes these arguments:



The geometry of a juce::Slider object

The code you just added will draw a solid circle in the center of the knob. It first creates a `bounds` variable that is a `juce::Rectangle<int>` from the `x,y`-position and the `width` arguments.

For the height of the rectangle this also uses the `width` argument. This is not a typo! Since we're drawing a circle, the width and height of the circle must be the same or it will look like an ellipse — recall that the slider component is taller than wide since it needs to fit the textbox too.

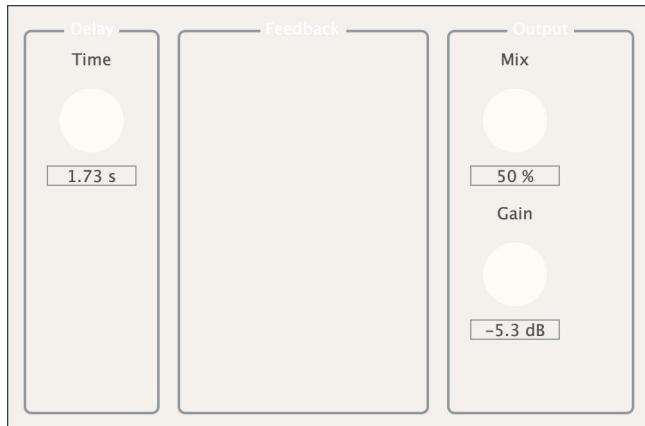
Since we're not using the `height` argument, I've marked it with `[[maybe_unused]]` to let the compiler know this isn't a mistake.

The `.toFloat()` turns the rectangle into type `juce::Rectangle<float>`. This is done because a lot of the drawing commands we're going to use expect floating-point values instead of integers. Sometimes it's a little annoying that JUCE works with `int` values for some UI things and with `float` values for other UI things, but at least it's straightforward to convert from one to the other.

Next, `bounds . reduced()` will subtract 10 pixels from each side of the rectangle to inset it inside the component's bounds. In other words, we're adding a 10-pixel margin around the circle.

Finally, we set the current color to `Colors::Knob::outline` and do `g.fillEllipse` to draw a filled circle inside the reduced rectangle.

Try it out. For every knob there is now a white circle. (Well, almost white.)



Knobs are drawn as white circles

To visually separate this circle from the background, which is also quite light, I want to add a drop shadow. This is pretty easy with JUCE.

Go to `LookAndFeel.h` and add the following line inside the `private:` section of the `RotaryKnobLookAndFeel` class:

```
juce::DropShadow dropShadow { Colors::Knob::dropShadow, 6, { 0, 3 } };
```

This creates a new `juce::DropShadow` object named `dropShadow`. The constructor takes three arguments:

- the color, here one of the `juce::Colour` objects from our `Colors` namespace
- the radius, here 6 pixels
- the offset, which is a `juce::Point<int>` object. It is `{ 0, 3 }`, meaning the offset is 0 pixels horizontally and 3 pixels vertically.

Note that you didn't need to write `juce::Point<int>(0, 3)` for the offset, although you could if you wanted to. The C++ compiler knows that the third argument should be a `juce::Point<int>` object and so all it requires is the arguments for its constructor in between `{ }` brackets.

In **LookAndFeel.cpp**, in `drawRotarySlider`, change the existing drawing code to draw the drop shadow before the circle, so that the circle lies on top of the shadow:

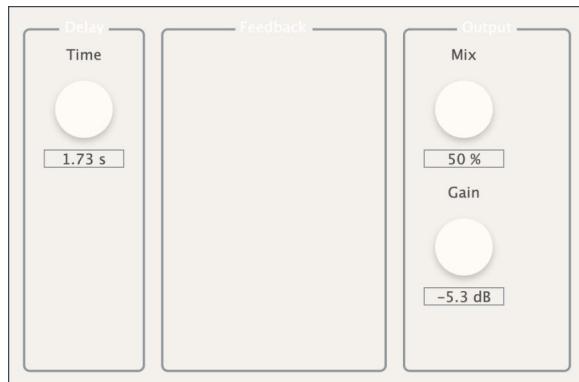
```
auto bounds = juce::Rectangle<int>(x, y, width, width).toFloat();
auto knobRect = bounds.reduced(10.0f, 10.0f);

// add these three new lines
auto path = juce::Path();
path.addEllipse(knobRect);
dropShadow.drawForPath(g, path);

g.setColour(Colors::Knob::outline);
g.fillEllipse(knobRect);
```

The new code creates a `juce::Path` object. In graphics terminology, a **path** is a series of points that are connected to create a certain shape. By doing `path.addEllipse`, we add a circle to the path using the dimensions of `knobRect`. Then we tell the `dropShadow` object to draw itself inside the graphics context with the given path.

Try it out. Now the knobs look like this:



The white circles have a drop shadow

Next, we will draw a subtle gradient inside the circle. Add the following lines to `drawRotarySlider`, below the existing code:

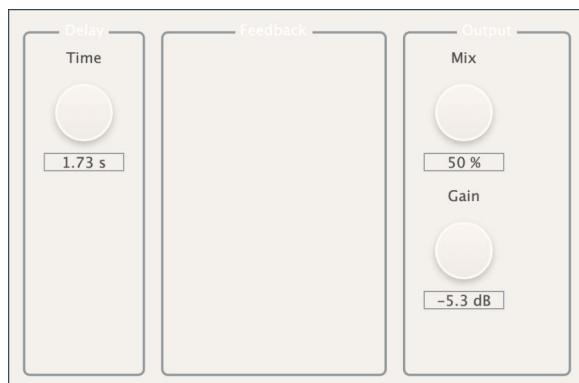
```
auto innerRect = knobRect.reduced(2.0f, 2.0f);
auto gradient = juce::ColourGradient(
    Colors::Knob::gradientTop, 0.0f, innerRect.getY(),
    Colors::Knob::gradientBottom, 0.0f, innerRect.getBottom(), false);
g.setGradientFill(gradient);
g.fillEllipse(innerRect);
```

This calls `knobRect.reduced` to take the square containing the knob and subtracts two pixels from all sides. Then it creates a `juce::ColourGradient` object that goes from the color given by `Colors::Knob::gradientTop` to the color `Colors::Knob::gradientBottom`.

`g.setGradientFill` tells the graphics context that from now on it should use this gradient to fill shapes instead of the color that was previously set with `g.setColour`.

And finally, `g.fillEllipse` draws a circle again, this time two pixels smaller than previously. This way the white circle that we drew earlier will look like a nice outline around the gradient.

Try it out. The gradient added some depth and it's starting to resemble an actual knob.



The knob with a subtle gradient

Note: I'm using `auto` a lot in this drawing code, mainly because it saves space. Instead of using `auto`, you could have declared the `innerRect` and `gradient` variables by fully writing out their data types, like so:

```
juce::Rectangle<float> innerRect = knobRect.reduced(2.0f, 2.0f);
juce::ColourGradient gradient = juce::ColourGradient( ... );
```

The advantage is that this is more explicit about the data types that are being used. The downside is that it's more effort to type it in. Personally, I also find it makes the code harder to read. `auto gradient = juce::ColourGradient(...)` does the same thing and is simpler.

Drawing the track

Next, we'll draw an arc around the outside of the knob. We'll call this the "track".

Add the following code to `drawRotarySlider`:

```
auto center = bounds.getCentre();
auto radius = bounds.getWidth() / 2.0f;
auto lineWidth = 3.0f;
auto arcRadius = radius - lineWidth/2.0f;

juce::Path backgroundArc;
backgroundArc.addCentredArc(center.x,
                            center.y,
                            arcRadius,
                            arcRadius,
                            0.0f,
                            rotaryStartAngle,
                            rotaryEndAngle,
                            true);

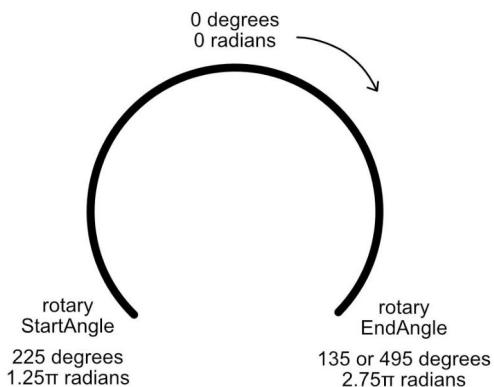
auto strokeType = juce::PathStrokeType(
    lineWidth, juce::PathStrokeType::curved, juce::PathStrokeType::rounded);
g.setColour(Colors::Knob::trackBackground);
g.strokePath(backgroundArc, strokeType);
```

This creates a new `juce::Path` object named `backgroundArc` that describes an arc — a portion of a circle — with its center in the middle of the rectangle and a radius of half the rectangle.

Actually, we'll subtract a few extra pixels from the radius because we're going to draw this arc with a certain `lineWidth`, which extends outwards. If we didn't subtract half the line width from the radius, some of the arc would be drawn outside the component where it wouldn't be visible.

The arc starts at the angle given by `rotaryStartAngle` and ends at the angle from `rotaryEndAngle`. Both angles were passed into the function as arguments. They are measured in radians.

If you're fuzzy on your trigonometry, there are 2π or 6.283 radians in a single trip around the circumference of a circle. In JUCE, 0 radians means the top center of the arc.

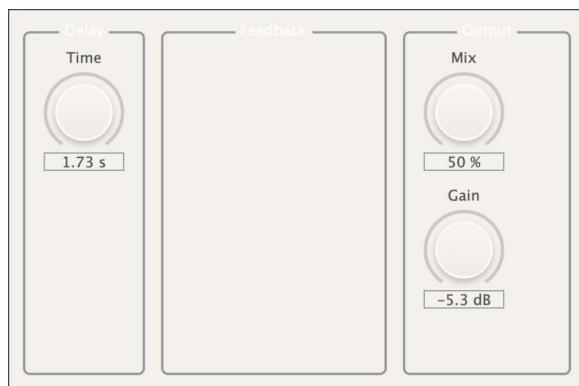


The arc used to draw the slider's track

Previously when we drew shapes we always filled them completely with a solid color or a gradient. However, here we are going to stroke the path, which means we only draw its outline.

The `g.strokePath` function requires a `juce::PathStrokeType` object that tells it how to draw the line. Here, we're drawing a line around the arc that is three pixels wide, and that has rounded ends.

Try it out. There is now a track around the knob:



The knob with a track around it

Personally, I find the default start and end angles too extreme. We can easily fix this by adding a bit of code to **RotaryKnob.cpp**. In the `RotaryKnob` constructor, add the following lines.

```
float pi = juce::MathConstants<float>::pi;
slider.setRotaryParameters(1.25f * pi, 2.75f * pi, true);
```

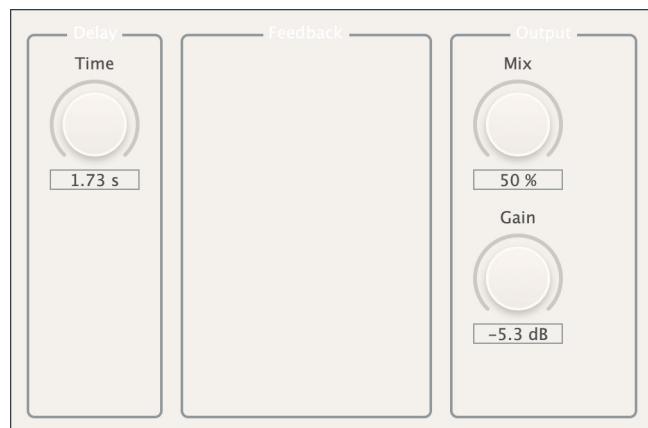
The first two arguments to `setRotaryParameters` are the start and end angle, again in radians. Here we're using 1.25π pi and 2.75π , which correspond to 225 degrees and 495 degrees, respectively. (The standard JUCE angles are 216 and 504 degrees.)

You may find it strange that the ending angle is larger than 360 degrees, but there is a good reason for this. If we were to specify an end angle of 135 degrees, which is in the same position on the circle as 495 degrees, the arc would be drawn across the bottom of the knob, like so:



The arc is drawn the wrong way around

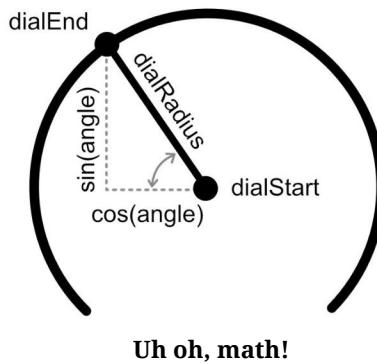
By adding an extra 360 degrees to the end angle, the arc will go the long way around, which is what we want. These new angles looks neater, I think:



The start and end angles are less extreme

Drawing the dial

Next, we will draw the dial on the knob. The dial is a line from the center of the knob to the point on the track that corresponds to the plug-in parameter's current value. Drawing the dial will require a little math.



The dial is a line between two points that I named `dialStart` and `dialEnd`. The starting point is easy, this is the center of the bounds rectangle. We already have a variable `center` for that.

The end point, however, lies somewhere on the circle, depending on the value of the parameter that's attached to this slider. JUCE already gives us the `sliderPos` argument, a value between 0 and 1, so we can calculate what the angle should be based on that, and then figure out where the corresponding point sits on the circle.

The following code does this. Add it to `LookAndFeel.cpp`, below the existing code in `drawRotarySlider`:

```
auto dialRadius = innerRect.getHeight() / 2.0f;
auto toAngle = rotaryStartAngle + sliderPos * (rotaryEndAngle - rotaryStartAngle);

juce::Point<float> dialStart(center.x, center.y);
juce::Point<float> dialEnd(center.x + dialRadius * std::sin(toAngle),
                           center.y - dialRadius * std::cos(toAngle));
```

In case you're wondering how the math works for calculating the `dialEnd` point, this is basic trigonometry. Given an angle of rotation and a radius, the x-position is `radius * cos(angle)` and the y-position is `radius * sin(angle)`. You may remember this from high school math class. If not, it's a useful formula to write down somewhere.

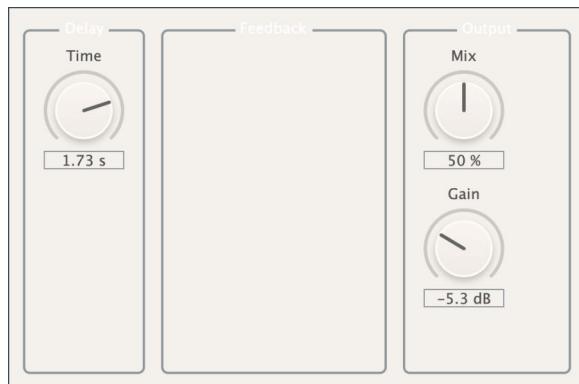
Math alert! In the above code, `sin` and `cos` are actually switched around and the `y`-position *subtracts* the rotated value from the center point. This is because JUCE considers 0 degrees to be at the top of the circle while in math 0 degrees is on the right.

To put 0 degrees at the top like how JUCE wants it, we would need to subtract $\pi/2$ from the angle, which is equivalent to rotating the frame of reference by 90 degrees. Since `cos` is the same as `sin` but shifted by $\pi/2$, doing `sin(angle)` is the same as `cos(angle - \pi/2)` and `-cos(angle)` is the same as `sin(angle - \pi/2)`.

Next we'll create a new path that describes the line between the `dialStart` and `dialEnd` points, and we'll stroke it using the same `strokeType` as before:

```
juce::Path dialPath;
dialPath.startNewSubPath(dialStart);
dialPath.lineTo(dialEnd);
g.setColour(Colors::Knob::dial);
g.strokePath(dialPath, strokeType);
```

Try it out and this is what it looks like:



The knob has a dial that shows its value

Now if you drag the knob, you can see the dial rotate along with it. I think this is a good start but I want to tweak the dial a little.

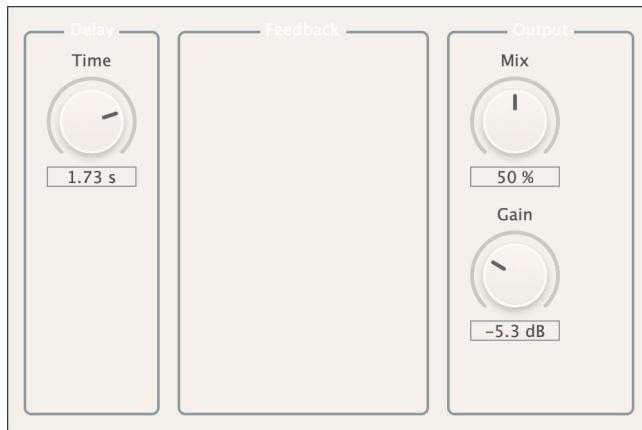
First, let's offset it from the center. Do this by changing the line that creates the `dialStart` variable to the following. This does the same kind of calculation as `dialEnd`, to put the start point at 10 pixels from the center.

```
juce::Point<float> dialStart(center.x + 10.0f * std::sin(toAngle),  
                               center.y - 10.0f * std::cos(toAngle));
```

Also change the `dialRadius` variable to be smaller, so it doesn't extend all the way to the end:

```
auto dialRadius = innerRect.getHeight()/2.0f - lineWidth;
```

Nice, that looks pretty good:



The dials are shorter but sweeter

Note: I am spoiled for working on a Retina screen with my Mac, which draws four pixels for every one pixel specified in the code. On my Mac the dial is not 3 pixels wide but 6. If you have a HiDPI screen on a Windows computer, it will do the same thing. But since the circle has an even width of 50 pixels (total width is 70 pixels minus 10 on each side), drawing a 3-pixel line in the center of that circle does not look very sharp on a 1x screen.

I don't think that's a problem for this particular knob, but something to keep in mind if you always want to draw your lines as sharp as possible. In that case using a 2 or 4-pixel wide dial would have been better. Many Windows users will have 125% zoom enabled anyway, in which case none of the lines are sharp.

Coloring the track

The knob will look even better if we fill the track up to where the dial points with a different color. This is done by drawing another arc on top of the track, but one that goes from the `rotaryStartAngle` to the `toAngle` rather than the `rotaryEndAngle`.

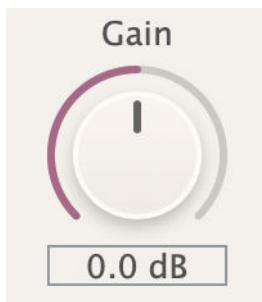
Add the following code to `drawRotarySlider`:

```
if (slider.isEnabled()) {
    juce::Path valueArc;
    valueArc.addCentredArc(center.x,
                           center.y,
                           arcRadius,
                           arcRadius,
                           0.0f,
                           rotaryStartAngle,
                           toAngle,
                           true);

    g.setColour(Colors::Knob::trackActive);
    g.strokePath(valueArc, strokeType);
}
```

This does nothing you haven't seen before, except for the `slider.isEnabled()` check. If the slider is disabled, we won't highlight the active portion of the track.

Try it out:



The active portion of the track is now colored

That's a good-looking knob, if I say so myself.

However... the eagle-eyed readers among you may have noticed that something's off with the **Gain** knob. This knob goes from -12 dB to +12 dB, and is 0 dB in the center when the dial is pointing straight up.

Having the track colored like this makes it look like the **Gain** knob is “active” while it’s actually at the neutral value of 0 dB. For this particular knob, it would be better to fill the track from the center out.

We could make a new look-and-feel class for this and assign that to only the **Gain** knob, but since it would have pretty much the same drawing code as the other knobs — except for how to color the track — that seems wasteful. It would be better if we could write something like this in `drawRotarySlider`:

```
if (slider.drawsFromMiddle()) {
    // draw the arc from the middle
} else {
    // draw the arc from the start
}
```

Unfortunately, `juce::Slider` does not have such a `drawsFromMiddle` property. But all is not lost... `juce::Component` objects have a set of properties that we can give values, where a property is a **name-value pair**.

For the knobs that we want to draw from the middle, we can give the `juce::Slider` a property named "drawFromMiddle" with the value `true`. In the look-and-feel we can check if this property is set and change the drawing logic accordingly.

In **RotaryKnob.h**, change the constructor to read:

```
RotaryKnob(const juce::String& text,
           juce::AudioProcessorValueTreeState& apvts,
           const juce::ParameterID& parameterID,
           bool drawFromMiddle = false);
```

You’ve added a new argument, `drawFromMiddle`, that is a `bool` value. Recall that in programming, a boolean is a value that is either `true` or `false`. Here, by default it’s set to `false`. That means if we don’t specify this argument when creating a new `RotaryKnob`, the `drawFromMiddle` argument will be `false`.

In **RotaryKnob.cpp**, change the constructor implementation to the following.

```

RotaryKnob::RotaryKnob(const juce::String& text,
                      juce::AudioProcessorValueTreeState& apvts,
                      const juce::ParameterID& parameterID,
                      bool drawFromMiddle)
    : attachment(apvts, parameterID.getParamID(), slider)
{
    // ... keep the existing code ...

    slider.getProperties().set("drawFromMiddle", drawFromMiddle);
}

```

The `slider.getProperties()` function returns a reference to the component's `juce::NamedValueSet` object. With `.set("name", value)` we can insert a new name-value pair into the component's properties.

Go to **PluginEditor.h** and change the line for the gainKnob to:

```
RotaryKnob gainKnob { "Gain", audioProcessor.apvts, gainParamID, true };
```

This sets the value of the `drawFromMiddle` argument to `true`. You don't need to change the other knobs — if the last argument is not supplied, it automatically is `false` because we gave it a default value in the `RotaryKnob` constructor.

Switch back to **LookAndFeel.cpp** and make the following changes to the code that draws the colored arc:

```

if (slider.isEnabled()) {
    // these lines are new
    float fromAngle = rotaryStartAngle;
    if (slider.getProperties()["drawFromMiddle"]) {
        fromAngle += (rotaryEndAngle - rotaryStartAngle) / 2.0f;
    }

    juce::Path valueArc;
    valueArc.addCentredArc(center.x,
                           center.y,
                           arcRadius,
                           arcRadius,
                           0.0f,
                           fromAngle, // this also changed!
                           toAngle,
                           true);

    g.setColour(Colors::Knob::trackActive);
    g.strokePath(valueArc, strokeType);
}

```

Most of this code is still the same but we now draw starting at the new `fromAngle` variable and not from `rotaryStartAngle`.

The notation `["drawFromMiddle"]` will read the value of the `"drawFromMiddle"` property from the `juce::NamedValueSet` object. In C++ the `[]` brackets are generally used to look up elements of a collection, such as a vector or array. You've previously used it to read samples from the `juce::AudioBuffer` in `processBlock`. Here it's used to look up something by name.

An `if` statement always checks if something is true, so if the `"drawFromMiddle"` property is set to `true` on the slider, then `fromAngle` is calculated to fall exactly in between the start and end angle. For our current start and end angles of 225 and 495 degrees, that makes `fromAngle` equal to 360 degrees, which is the same as 0 degrees, or at the top of the circle.

Give it a try! The Gain knob's track is colored from the middle out and not from the bottom-left like the other knobs.



We set the color with `g.setColour(Colors::Knob::trackActive)` when stroking the arc. However, there is a color ID for this, `juce::Slider::rotarySliderFillColourId`. It would be better if we used that JUCE color ID instead of the hardcoded color values, so that the UI becomes easier to theme.

In the knob drawing code, replace the line `g.setColour(Colors::Knob::trackActive)` with the following. This now calls `slider.findColour` to look up the color with ID `rotarySliderFillColourId`.

```
g.setColour(slider.findColour(juce::Slider::rotarySliderFillColourId));
```

To set this to the color we want, add the following line to the `RotaryKnobLookAndFeel` constructor:

```
setColour(juce::Slider::rotarySliderFillColourId, Colors::Knob::trackActive);
```

This draws the same as before, but if you want to, the color of the active part of the track can be overridden on a per-knob basis. For example, in `PluginEditor.cpp`'s constructor you could write something like the following if you wanted to give one of the knobs a different color. In other words, using these color IDs in the drawing code lets you override the default colors from the look-and-feel.

```
gainKnob.slider.setColour(juce::Slider::rotarySliderFillColourId,  
                          juce::Colours::green);
```

Run the code and notice that the other knobs still use the normal color. (I've commented out this green color again, but feel free to keep it if you like it.)

Fonts

Choosing a good font can give your UI style and character. For our plug-in we will use Lato, a font that is freely available. You can get the Lato TTF file from the book's resources.

There are two ways to change the font in your JUCE program:

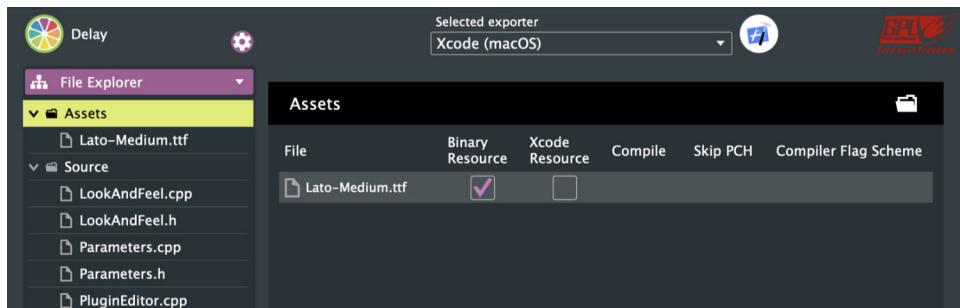
1. Specify a font name, such as "Arial". The downside is that the font you're using needs to be installed on the user's computer, and Mac and Windows do not come with the same collection of fonts. "Arial" will work, since that is one of the fonts both Mac and Windows have in common, but "Lato" will not.
2. Embed the font file inside the plug-in. Now the font will work regardless of the platform, but the plug-in's binary size will become larger (usually not a big deal) and you can only do this with fonts that are freely distributable (such as Lato) or that you bought a license for.

We will choose option two. Embedding resources such as fonts in your plug-in is easier than you might think, as Projucer can automatically take care of this.

The **Resources** folder that comes with the book²⁹ contains the file **Lato-Medium.ttf**.

We'll put this file in its own group instead of in the **Source** group, so inside Projucer's **File Explorer** click the + button and choose **Add New Group**. Name this **Assets**.

Make sure the **Assets** group is selected. Then click the + button again and choose **Add Existing Files...** and select **Lato-Medium.ttf**. You can also simply drag-and-drop this file into the **Assets** group.



The font added to the new Assets group

With the **Assets** group selected, the pane on the right shows that **Lato-Medium.ttf** will be included in the project as a Binary Resource. Save the project and open it in your IDE.

Projucer will have generated two new files: **BinaryData.h** and **BinaryData.cpp**. You can find these files in the **Juce Library Code** folder in your IDE.

For Xcode: In the **Project Navigator**, expand the **Juce Library Code** folder to reveal these two new files.

For Visual Studio: In the **Solution Explorer**, look at the **Juce Library Code** folder in the **Delay_SharedCode** project.

If you open **BinaryData.h** you'll see the following:

```
namespace BinaryData
{
    extern const char* LatoMedium_ttf;
    const int LatoMedium_ttfSize = 663564;

    // ...other stuff that we don't need...
}
```

²⁹<https://github.com/TheAudioProgrammer/getting-started-book>

Projucer created a new namespace `BinaryData` containing an object `LatoMedium_ttf` that holds the `Lato-Medium.ttf` font file, and a variable `LatoMedium_ttfSize` that describes how large this file is in bytes, here 663564 bytes or about 664 KB.

The contents of `BinaryData.cpp` looks like this:

```
namespace BinaryData
{
    //===== Lato-Medium.ttf =====
    static const unsigned char temp_binary_data_0[] =
    { 0,1,0,0,0,17,1,0,0,4,0,16,71,80,79,83,120,98,80,99,0,8,171,72,0,1,75,194,71,
    // ...many more lines here...
    11,194,11,197,11,200,0,0 };

    const char* LatoMedium_ttf = (const char*) temp_binary_data_0;
}
```

This is a very big source file as it contains every single byte that makes up the `Lato-Medium.ttf` file but written as C++ source code. The data type of the `LatoMedium_ttf` object is `const char*`, which means it's an array of “characters”, which really means an array of bytes as each `char` takes up a single byte of memory.

We can now refer to this embedded font file using `BinaryData::LatoMedium_ttf`. We will use this binary data to construct a `juce::Typeface` object, which can then be used to create a `juce::Font` that will be used for drawing the text in the user interface.

Note: You shouldn't make any changes to `BinaryData.h` or `.cpp` yourself. Every time you save the project from Projucer, it will overwrite the contents of these files.

Open `LookAndFeel.h`. Add the following code to it. I suggest placing it between namespace `Colors` and the `RotaryKnobLookAndFeel` class. It doesn't matter where you put it, but that seems like a good spot.

```
class Fonts
{
public:
    static juce::Font getFont(float height = 16.0f);

private:
    static const juce::Typeface::Ptr typeface;
};
```

This defines a new class named `Fonts`. We're using this for convenience so that we'll be able to write `Fonts::getFont()` whenever we want to use the Lato font anywhere.

The `getFont` function and the `typeface` variable are both `static`, meaning we don't need to create an instance of `Fonts` in order to use it. You can enforce this by adding the following line to class `Fonts`:

```
Fonts() = delete;
```

This tells the C++ compiler that class `Fonts` does not have a constructor, and therefore you cannot write something like `Fonts fonts;` to create a new instance of this class. Trying that will give a compilation error.

Anyway, enough C++ trivia. Let's implement this `Fonts` class. Go to `LookAndFeel.cpp` and add the following code at the top but below the `#include` statement:

```
const juce::Typeface::Ptr Fonts::typeface = juce::Typeface::createSystemTypefaceFor(
    BinaryData::LatoMedium_ttf, BinaryData::LatoMedium_ttfSize);

juce::Font Fonts::getFont(float height)
{
    return juce::Font(typeface).withHeight(height);
}
```

The first line creates the `typeface` variable. This is done only once when the plug-in is first loaded into memory. The call to `juce::Typeface::createSystemTypefaceFor` will load the font from the font file data that we just added to `BinaryData`.

The `Fonts::getFont()` function will return a font based on this typeface, with a particular height. We assigned a default value of 16 points to the `height` argument in the header file, but you can ask for a different font height too.

You can now write `label.setFont(Fonts::getFont())` to make a `juce::Label` object use this font. However, we can also do this in one go for all labels by putting it in the look-and-feel.

In `LookAndFeel.h`, inside the `RotaryKnobLookAndFeel` class, add the following line to the `public:` section:

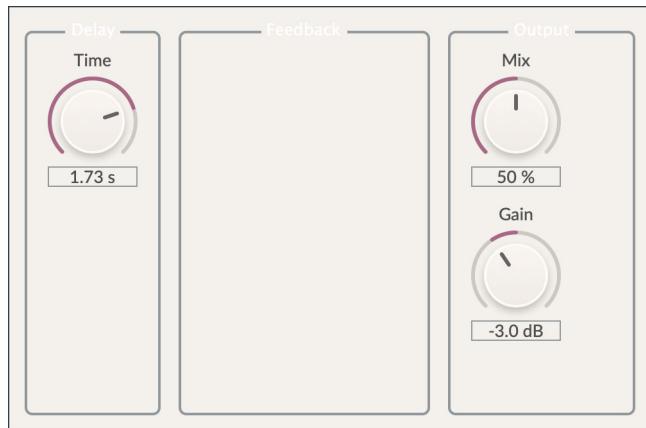
```
juce::Font getLabelFont(juce::Label&) override;
```

Then in **LookAndFeel.cpp**, add the implementation of this function:

```
juce::Font RotaryKnobLookAndFeel::getLabelFont([[maybe_unused]] juce::Label& label)
{
    return Fonts::getFont();
}
```

When the label is being painted, the look-and-feel calls `getLabelFont` to select the font for the label. By overriding `getLabelFont` we can change the font that's being used, here returning our Lato font with the default height of 16 points. The `label` argument isn't used inside this function, so I've marked it as `[[maybe_unused]]`.

Try it out. Notice that both the label above the knob as well as the text box are using the new font:



The knobs use a custom font

The labels on the group components still use the default font, so we'll style these things next.

Note: The above code will work fine with JUCE 7 but gives a compiler warning with JUCE 8. If you're using JUCE 8 or later, the compiler says something like: “‘Font’ is deprecated: Use the constructor that takes a `FontOptions` argument”.

Deprecated means this API — in this case the `Font` class — has changed and should be replaced with something else. It will still work for now but may be permanently removed in a future version of JUCE.

To silence the warning with JUCE 8, change the `getFont` function to the following.

```
juce::Font Fonts::getFont(float height)
{
    return juce::FontOptions(typeface)
        .withMetricsKind(juce::TypefaceMetricsKind::legacy)
        .withHeight(height);
}
```

Instead of Font this uses a new class, FontOptions. This class was introduced to better handle Unicode and modern font rendering. One wrinkle is that it uses different metrics than JUCE 7, meaning it produces a font that looks slightly larger. To use the same font sizes between JUCE 7 and JUCE 8, I added the legacy metrics option. In your own projects, you can leave out this option.

Styling the group components

The reason the labels on the group components didn't get the new font is that they are not using the RotaryKnobLookAndFeel object. Which makes sense because they aren't rotary knobs. So, let's create a new look-and-feel.

Go to **LookAndFeel.h** and add some new colors to the namespace:

```
namespace Colors
{
    // ...existing code omitted...

    namespace Group
    {
        const juce::Colour label { 160, 155, 150 };
        const juce::Colour outline { 235, 230, 225 };
    }
}
```

Also add the new class at the bottom of the file:

```
class MainLookAndFeel : public juce::LookAndFeel_V4
{
public:
    MainLookAndFeel();

    juce::Font getLabelFont(juce::Label&) override;

private:
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(MainLookAndFeel)
};
```

I'm calling this `MainLookAndFeel` because we're going to assign it to the entire editor and it will handle all the look-and-feel stuff for everything except the knobs.

Go to `LookAndFeel.cpp` and add the implementation for this new class:

```
MainLookAndFeel::MainLookAndFeel()
{
    setColour(juce::GroupComponent::textColourId, Colors::Group::label);
    setColour(juce::GroupComponent::outlineColourId, Colors::Group::outline);
}

juce::Font MainLookAndFeel::getLabelFont([[maybe_unused]] juce::Label& label)
{
    return Fonts::getFont();
}
```

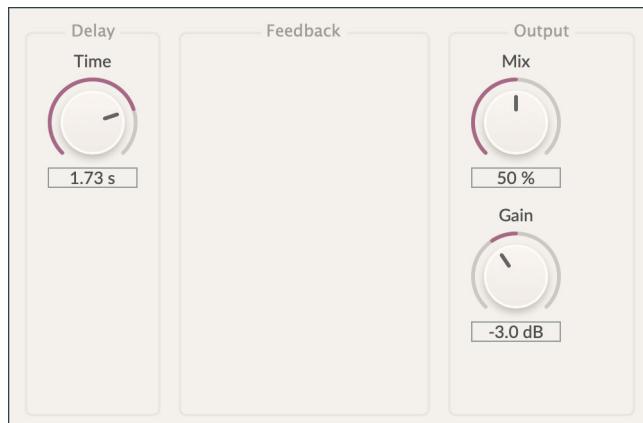
This look-and-feel just sets the color IDs on the `juce::GroupComponent` to the new colors, and changes the label font like before.

Then in `PluginEditor.cpp`, in the constructor do the following:

```
setLookAndFeel(new MainLookAndFeel);
```

This puts the new `MainLookAndFeel` on the whole editor component. Any child components that do not have a look-and-feel set explicitly, such as the group components, will use their parent component's look-and-feel, in this case `MainLookAndFeel`. We did not set a look-and-feel on the rotary knobs, so they'll keep using their own.

Build and run, and it looks like this:



The groups have been styled too

It's possible to override the function `drawGroupComponentOutline` in `MainLookAndFeel` to completely change how the group component is drawn, but I'm happy with the way it looks now, so that's left as an exercise for the reader.

Fixing a bug

Close the app by clicking the red × button in the top-right corner or from the menu bar, or quit the host if you're not running the Standalone Plugin app.

What happens next will shock you! The plug-in crashes... We have a bug!

It looks like this in Xcode:

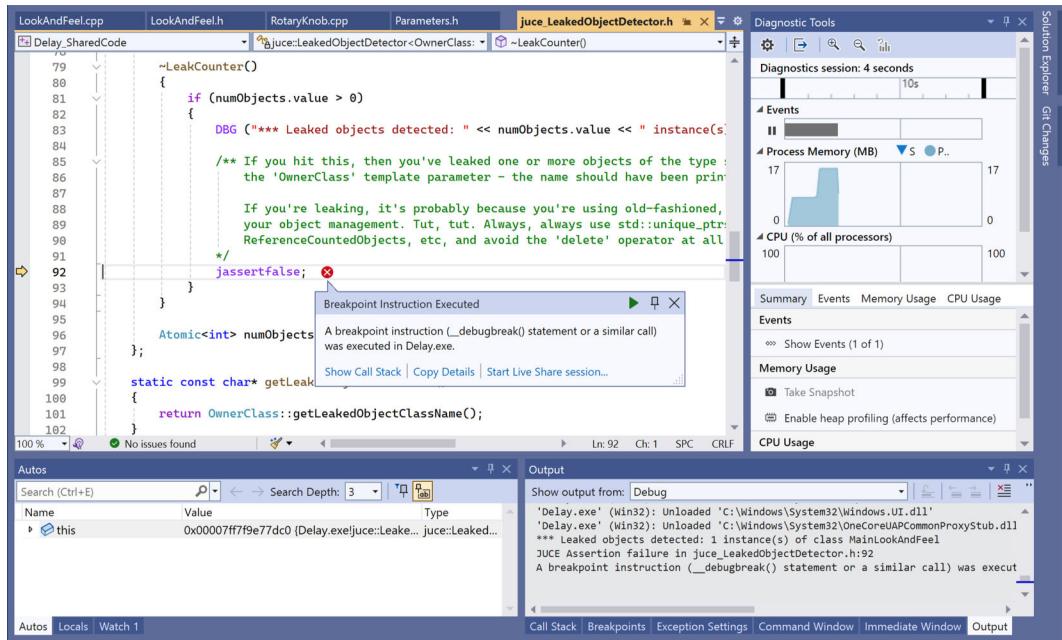
```

42 class LeakedObjectDetector
43 {
44     ~LeakCounter()
45     {
46         if (numObjects.value > 0)
47         {
48             DBG ("*** Leaked objects detected: " << numObjects.value << " instance(s) of class " <<
49                 getLeakedObjectClassName());
50
51             /**
52             * If you hit this, then you've leaked one or more objects of the type specified by
53             * the 'OwnerClass' template parameter - the name should have been printed by the line above.
54             */
55             jassertfalse; // JUCE Message Thread (1): EXC_BREAKPOINT (code=EXC_I386_BPT, subcode=0x0)
56         }
57     }
58
59     Atomic<int> numObjects;
60 };
61
62 <Parameters>
63 <PARAM id="delayTime" value="1725.031127929688"/>
64 <PARAM id="gain" value="-3.011911392211914"/>
65 <PARAM id="mix" value="50.0"/>
66 </Parameters>
67
68 *** Leaked objects detected: 1 instance(s) of class MainLookAndFeel
69 JUCE Assertion failure in juce_LeakedObjectDetector.h:92
70
71 (lldb)

```

The plug-in crashed!

The crash looks like this in Visual Studio:



The plug-in crashed!

There is a message in the output pane saying something like this:

```
*** Leaked objects detected: 1 instance(s) of class MainLookAndFeel
JUCE Assertion failure in juce_LeakedObjectDetector.h:92
```

It looks like there is a memory leak on our `MainLookAndFeel` object. In the source code editor the error points at a line that says `jassertfalse;`. An **assertion** is a kind of safety check that will stop the program and jump into the debugger when something is wrong.

In this case, the explanation is in the comment right above the `jassertfalse;` line. Remember that macro `JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR` at the bottom of our classes? That inserts a leak detector into the class and, well, it has detected a leak.

What happened is that I made you write the following in the `PluginEditor.cpp` constructor to use this look-and-feel, even though it's actually incorrect.

```
setLookAndFeel(new MainLookAndFeel); // wrong!
```

The expression `new MainLookAndFeel` will create an instance of the `MainLookAndFeel` class, but this instance does not get deallocated when the plug-in exits, since technically speaking it has no owner.

Object ownership in C++ can get complicated so we won't get deeply into it in this book, but what we should have done was the following. In `PluginEditor.h`, add the following line to the `private:` section of the class:

```
MainLookAndFeel mainLF;
```

and then in the constructor change the `setLookAndFeel` line to:

```
setLookAndFeel(&mainLF);
```

This way, the `mainLF` object now has an owner, the `DelayAudioProcessorEditor` instance, and that will ensure the `MainLookAndFeel` object gets deallocated when the plug-in is unloaded from the host.

Try it again to see what happens. Run the code and quit using the `x` button. What the hey? We get another error message in the output pane:

```
JUCE Assertion failure in juce_LookAndFeel.cpp:73
```

This time the IDE points at some code inside `LookAndFeel::~LookAndFeel()`. Note that this is JUCE's `LookAndFeel` class, not our own — you're looking at the file **juce_LookAndFeel.cpp** here. The number 73 in the error message refers to the line number in the source file. You may see a different number here depending on your version of JUCE.

Again, the error got triggered by a `jassert` and the comment above it explains why. Long story short, JUCE considers it problematic when a look-and-feel object is being deallocated while it's still in use. The memory leak is gone, but it's been replaced by a new bug.

The solution is simple enough. Go to `PluginEditor.cpp` and find the code for the **destructor**. Just like a constructor is a special function that is used when an instance of the class is created, the destructor is called when the instance is no longer being used and is deallocated.

Currently the code for the destructor looks like this:

```
DelayAudioProcessorEditor::~DelayAudioProcessorEditor()
{
}
```

The ~ in the name means this is a destructor. Right now, it's not doing anything since there isn't any code inside the function. Add this line to it:

```
setLookAndFeel(nullptr);
```

This tells the editor component to stop using the `mainLF` object as its look-and-feel, so that it can safely be deallocated. In C++, the `nullptr` keyword means “no object”. `setLookAndFeel(nullptr);` is the same as saying, “Hey `DelayAudioProcessorEditor`, you are no longer using a look-and-feel now.”

Try it once more and the program should exit without any errors.

The reason I did it wrong was to demonstrate that the JUCE leak detector is quite handy to find such issues, but also to show you what happens if you trigger an assertion, because you probably will run into those occasionally.

Styling the text boxes

The UI is already starting to look nice but there are improvements we can make to the text box that displays the knob's value. I don't like the border or how it looks when you click the value to edit it.

The border is drawn by the look-and-feel's `drawLabel` function. In case you're curious, you can see this in the JUCE source code in `juce_LookAndFeel_V2.cpp`. You can find this file in your IDE in the **JUCE Modules** folder, under `juce_gui_basics > lookandfeel`.

We could override `drawLabel` and remove the call to `g.drawRect` at the end, but it's simpler to just set the `outlineColourId` to a transparent color. The look-and-feel will still draw a rectangle, but it won't be visible.

In `LookAndFeel.cpp`, in the constructor for `RotaryKnobLookAndFeel`, add the line:

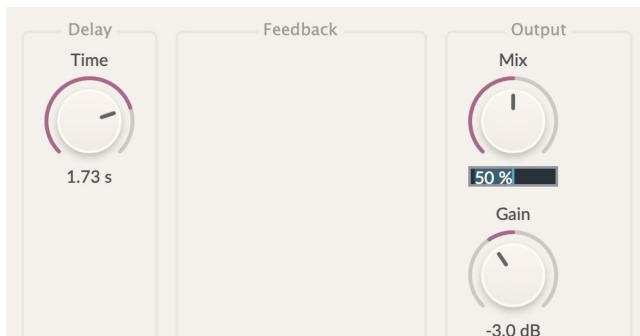
```
setColour(juce::Slider::textBoxOutlineColourId, juce::Colours::transparentBlack);
```

In the code for `drawLabel` in `juce_LookAndFeel_V2.cpp`, the color ID that gets used for drawing the rectangle is actually `juce::Label::outlineColourId`, but if you were to use that instead of `juce::Slider::textBoxOutlineColourId`, the rectangle doesn't go away.

This is one of the gotchas of the JUCE look-and-feel system. There is a function named `createSliderTextBox` that overwrites the value of `Label::outlineColourId` with that of `Slider::textBoxOutlineColourId`, and this happens after the `RotaryKnobLookAndFeel` constructor is performed. So, the appropriate color ID to use is `Slider::textBoxOutlineColourId`.

Sometimes this kind of behavior can be very puzzling: You're attempting to use a color ID that seems like it would be the correct one, but changing it has no effect. The solution is to dig through the JUCE source code to find what really happens, and which color ID you should have used instead.

If you run the plug-in you'll see that the borders have disappeared around the text boxes. We can also change what happens when the user clicks the text box to edit the value. Right now that looks like this (see the Mix knob).



No more borders around the text boxes

Not horrible, but we can do better. It does require writing a whole bunch of code. In `LookAndFeel.h`, add the following line to the `public:` section of the `RotaryKnobLookAndFeel` class:

```
juce::Label* createSliderTextBox(juce::Slider&) override;
```

In `LookAndFeel.cpp`, add the implementation for this function. I suggest putting it below the other functions from `RotaryKnobLookAndFeel`, to keep the source file organized the same as the header file.

```

juce::Label* RotaryKnobLookAndFeel::createSliderTextBox(juce::Slider& slider)
{
    auto l = new RotaryKnobLabel();
    l->setJustificationType(juce::Justification::centred);
    l->setKeyboardType(juce::TextInputTarget::decimalKeyboard);
    l->setColour(juce::Label::textColourId,
                  slider.findColour(juce::Slider::textBoxTextColourId));
    l->setColour(juce::TextEditor::textColourId, Colors::Knob::value);
    l->setColour(juce::TextEditor::highlightedTextColourId, Colors::Knob::value);
    l->setColour(juce::TextEditor::highlightColourId,
                  slider.findColour(juce::Slider::rotarySliderFillColourId));
    l->setColour(juce::TextEditor::backgroundColourId,
                  Colors::Knob::textBoxBackground);
    return l;
}

```

The `createSliderTextBox` function is called once by `juce::Slider` when the slider is created. This gives the look-and-feel a chance to set up the text box.

This function should return a `juce::Label` object. When the user clicks the text box to edit the value, this `Label` object creates a new `juce::TextEditor` object that temporarily replaces the label until the user is done editing.

In `createSliderTextBox`, you have the opportunity to tweak the settings of the label. Here, we're setting the text to be centered and the keyboard to the decimal keyboard (for mobile devices), and load the appropriate colors. The background color will be solid gray when editing the value.

We're not really doing anything that the default look-and-feel isn't doing — check `createSliderTextBox` in `juce_LookAndFeel_V2.cpp` and notice that it's essentially the same code. The reason for overriding `createSliderTextBox` is so we can return our own `juce::Label` subclass named `RotaryKnobLabel`.

Add the following to `LookAndFeel.cpp`, immediately *above* the `createSliderTextBox` function. The code in `createSliderTextBox` tries to create a new `RotaryKnobLabel` object but the C++ compiler doesn't know yet what that is. That's why this new class must come before `createSliderTextBox` in the source file.

```

class RotaryKnobLabel : public juce::Label
{
public:
    RotaryKnobLabel() : juce::Label() {}

    void mouseWheelMove(const juce::MouseEvent&,
                        const juce::MouseWheelDetails&) override {}
}

```

```

        std::unique_ptr<juce::AccessibilityHandler>
            createAccessibilityHandler() override
    {
        return createIgnoredAccessibilityHandler(*this);
    }

    juce::TextEditor* createEditorComponent() override
    {
        auto* ed = new juce::TextEditor(getName());
        ed->applyFontToAllText(getLookAndFeel().getLabelFont(*this));
        copyAllExplicitColoursTo(*ed);

        ed->setBorder(juce::BorderSize<int>());
        ed->setIndents(2, 1);
        ed->setJustification(juce::Justification::centredTop);
        return ed;
    }
};

```

This creates a new class named `RotaryKnobLabel`. Previously we defined our classes in the `.h` file but here we're doing it in the `.cpp` file. That's perfectly OK. No one outside this `.cpp` file will be able to use the class, but that's fine as it's only needed by the look-and-feel.

The `RotaryKnobLabel` is based on `juce::Label`. We override the `mouseWheelMove` and `createAccessibilityHandler` functions. No idea why, this is also what the original JUCE look-and-feel did, so I copied it from that.

For our purposes, the important function is `createEditorComponent`. This gets called when the user clicks the text box to edit the value. Overriding this function lets us customize the text editor component before it gets shown on screen. Here we set the font and colors on the text editor, remove any borders, and set the text to be centered rather than on the left.

Just setting the border to an empty `juce::BorderSize<int>` object doesn't mean the border rectangle will automatically disappear. For that we must override another function in the look-and-feel. Add the following function declaration to the `RotaryKnobLookAndFeel` class in `LookAndFeel.h`:

```
void drawTextEditorOutline(juce::Graphics&, int, int,
                           juce::TextEditor&) override { }
```

We don't want to draw an outline so this function should be empty. That's done by writing `{ }` after the function name. There's no need to add anything to the `.cpp` file for this function.

Try it out and the text box should now look like this when editing:



The text box while editing the value

The highlight color is the same as the track color on the knob. Changing the track color using `setColour(juce::Slider::rotarySliderFillColourId, ...)` will automatically change this highlight color too. That's the benefit of using JUCE's color ID system.

There are still a couple of small things to tweak. You can right-click the edit box to bring up a pop-up menu with Cut/Copy/Paste options. I'm not a big fan of this pop-up menu, plus we'd need to write another look-and-feel to style that. So, let's take the easy way out and disable it.

Additionally, there is no limit to the number of characters you can type into the text box. To fix both these issues, add the following lines to `createEditorComponent` in the `RotaryKnobLabel` class, just before the `return` statement:

```
ed->setPopupMenuEnabled(false);
ed->setInputRestrictions(8);
```

This turns off the pop-up menu and limits the input to 8 characters max, which is enough for our purposes.

The caret, the blinking vertical bar, currently has the wrong color. Fix this by adding a new color ID in the `RotaryKnobLookAndFeel` constructor:

```
setColour(juce::CaretComponent::caretColourId, Colors::Knob::caret);
```

Finally, I'm not a big fan of sharp edges. It's nicer to draw the background of the text editor using rounded corners instead. Add the following function declaration to the `RotaryKnobLookAndFeel` class in the `LookAndFeel.h` header file.

```
void fillTextEditorBackground(juce::Graphics&, int width, int height,
                               juce::TextEditor&) override;
```

Notice that not all of these arguments have names. That's fine. The names in the **.h** file do not have to correspond with the names in the **.cpp** file. Just another one of those C++ peculiarities.

Add the implementation of this new function to the source file:

```
void RotaryKnobLookAndFeel::fillTextEditorBackground(
    juce::Graphics& g, [[maybe_unused]] int width, [[maybe_unused]] int height,
    juce::TextEditor& textEditor)
{
    g.setColour(Colors::Knob::textBoxBackground);
    g.fillRoundedRectangle(
        textEditor.getLocalBounds().reduced(4, 0).toFloat(), 4.0f);
}
```

Nice, it looks pretty sweet:



The finished text box

It took quite a bit of code to make it happen, but that's the price you pay to get a good-looking UI in JUCE.

Drawing images

To conclude this whirlwind tour of look-and-feels and drawing with `juce::Graphics`, let's add a logo image to the plug-in.

In the **Resources** folder for this book is a **Logo.png** image file. Add this to the project's binary data in Projucer, just like you did with the font.

If you look at **BinaryData.h**, there are some new variables that contain the bytes from the PNG file:

```
namespace BinaryData
{
    extern const char* LatoMedium_ttf;
    const int LatoMedium_ttfSize = 663564;

    extern const char* Logo_png;
    const int Logo_pngSize = 8719;

    // ...skipped the rest...
}
```

You can now use `BinaryData::Logo_png` to read the embedded PNG image.

Note: If you've added an image to Projucer but you're still making changes to it in an image editor, you will need to re-save the Projucer project to update the **BinaryData.cpp** file. If you ever edit the image and wonder why the old version keeps showing up in the plug-in, you may have forgotten to export from Projucer. You also need to compile the plug-in again in the IDE.

We will draw the PNG inside the `DelayAudioProcessorEditor`, so switch to **PluginEditor.cpp** and change the `paint` function to the following:

```
void DelayAudioProcessorEditor::paint (juce::Graphics& g)
{
    g.fillAll(Colors::background);

    auto rect = getLocalBounds().withHeight(40);
    g.setColour(Colors::header);
    g.fillRect(rect);

    auto image = juce::ImageCache::getFromMemory(
        BinaryData::Logo_png, BinaryData::Logo_pngSize);

    int destWidth = image.getWidth() / 2;
    int destHeight = image.getHeight() / 2;
    g.drawImage(image,
                getWidth() / 2 - destWidth / 2, 0, destWidth, destHeight,
                0, 0, image.getWidth(), image.getHeight());
}
```

First, this creates a rectangle with the width of the component and a height of 40 pixels, and fills it with a dark gray color from `Colors::header`.

Next, it uses the `juce::ImageCache` to load the `Logo.png` file from the `BinaryData` into a `juce::Image` object named `image`.

The reason for using `juce::ImageCache` is that this will only load and decode the PNG once to put it in memory, and from then on it will keep using the decoded version from memory. Decoding an image can be slow, so we'd rather do that just once.

Finally, the image is drawn using the `g.drawImage` command. This takes a whole bunch of arguments:

- The first argument is the `juce::Image` object that we got back from the `juce::ImageCache`.
- The next four arguments are the destination x,y-coordinate, width, and height. We horizontally center the image in the editor.
- The last four arguments are the source x,y-coordinate, width, and height.

The `drawImage` function is quite powerful and lets you not just draw the entire image, but also portions of it (which is useful for film strips and texture atlases). In our case we want to draw the entire image, so the source rectangle covers the full image.

Note that the destination width and height have been divided by 2. That's because the image is actually twice as wide and tall as needed: We'll draw it into a 40-pixel tall region on the screen but the PNG image is 80 pixels high.

This is done to support Retina screens on Mac and HiDPI screens on Windows, where one logical pixel might correspond to multiple physical pixels. A 40-pixel high image would look blurry on these high-resolution screens.

Before you can run the plug-in, first you should move all the other components down by 40 pixels or they will be drawn on top of the logo image.

In the `resized` function in `PluginEditor.cpp`, change the lines for the `y` and `height` variables to the following. This adds 40 pixels to `y` and subtracts an additional 40 pixels from `height`.

```
int y = 50;  
int height = bounds.getHeight() - 60;
```

Try it out. Run the plug-in and the editor now looks like the following.



The plug-in has a header bar with a logo image

In a later chapter we'll put some more UI components inside the header.

Tip: When designing images in a painting application, make sure the colorspace is set to **sRGB**, otherwise the colors in the image won't match the colors created with `juce::Colour`.

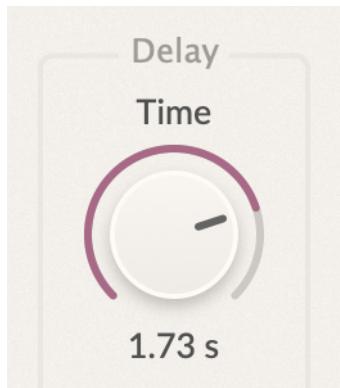
Just for fun we can also add some texture to the background using a tiled image. In the book's **Resources** folder is a file **Noise.png**. Add this to Projucer as before.

In `paint`, replace the `g.fillAll(Colors::background)` line with the following:

```
auto noise = juce::ImageCache::getFromMemory(
    BinaryData::Noise_png, BinaryData::Noise_pngSize);
auto fillType = juce::FillType(noise, juce::AffineTransform::scale(0.5f));
g.setFillType(fillType);
g.fillRect(getLocalBounds());
```

This grabs the **Noise.png** image from the `juce::ImageCache` and uses this to define a tiled fill. The `juce::FillType` object has a scale of `0.5f`, which again is to support Retina and HiDPI screens. Then `g.fillRect` will draw the entire component by tiling the image.

It's a subtle noise pattern but you can see it when you zoom in:



The noise texture on the background

Notice how the knob is transparent and that the background texture shows through. That's because `drawRotarySlider` in our look-and-feel does not paint the background of the knob. Any pixels that do not get painted will remain transparent.

Note: In this plug-in we've done all our drawing using `juce::Graphics` paths and lines. An alternative and popular way to draw knobs is to create a series of images for the different positions of the knob. Depending on the position of the slider you draw the corresponding knob image into the component. Usually for efficiency reasons the different images are stacked into a single PNG file, known as a film strip or texture atlas. If you're not much of a graphics designer yourself, you can find free knob and button images online, or even buy them from companies that specialize in plug-in UI design.

Phew, this was a lot of code to build the UI and make it look nice! That's true for most plug-ins: the user interface requires more code than the audio processing and DSP. On the flip side, once you've created a nice look-and-feel, you can use it in your other plug-in projects too.

By the way, you may be wondering how I knew how to write all this drawing code. If you look at the [documentation³⁰](#) for `juce::LookAndFeel_V4::drawRotarySlider`, for example, it's not very enlightening as to what the function arguments mean. The answer is: I looked at the JUCE source code to figure out how they do their drawing, and then I tweaked it until I got the results I wanted.

³⁰https://docs.juce.com/master/classLookAndFeel_V4.html

Most of the JUCE framework has pretty good documentation but the look-and-feel stuff is not well documented. You'll have to dig through the JUCE source code to make sense of it.

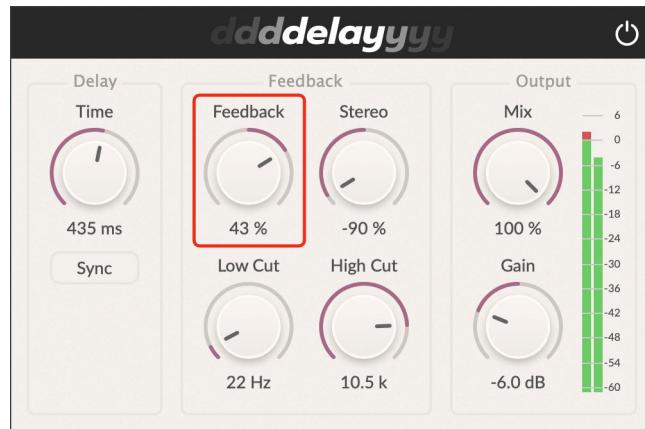
But that's not necessarily a bad thing: learning to read the JUCE code base is a useful exercise and it will allow you to get the most out of JUCE.

In the next chapter we'll get back to audio programming and add feedback to the delay effect.

12: Feedback

Right now, the plug-in repeats the sound just once, after the specified delay time. It's also possible to make a delay that outputs the sound multiple times. Using feedback, it can even have infinite repetitions.

In this chapter you'll build the following part of the plug-in:



Multi-tap delays

Creating a delay with more than one echo can be done by reading the delay line at multiple places, known as **taps**. Currently, we only have a single tap because we do `popSample` once at every timestep (well, twice, since there are two channels of audio).

We could read from a second tap that's another `delayInSamples` timesteps behind the first tap, and mix that delayed sample value into the output sound, perhaps at a slightly lower volume.

Let's review the audio processing code that is currently in `processBlock`:

```
for (int sample = 0; sample < buffer.getNumSamples(); ++sample) {
    params.smoothen();

    float delayInSamples = params.delayTime/1000.0f * sampleRate;
    delayLine.setDelay(delayInSamples);

    float dryL = channelDataL[sample];
    float dryR = channelDataR[sample];

    delayLine.pushSample(0, dryL);
    delayLine.pushSample(1, dryR);

    float wetL = delayLine.popSample(0);
    float wetR = delayLine.popSample(1);

    float mixL = dryL + wetL * params.mix;
    float mixR = dryR + wetR * params.mix;

    channelDataL[sample] = mixL * params.gain;
    channelDataR[sample] = mixR * params.gain;
}
```

To recap what this does:

1. Smoothen changes to the plug-in parameters and get the current delay length.
2. Read the next audio sample into `dryL` for the left channel and `dryR` for the right channel.
3. Push these dry samples into the delay line.
4. Read the delayed samples into `wetL` and `wetR` using `delayLine.popSample()`.
5. Calculate a dry/wet mix.
6. Write the output samples back into the `juce::AudioBuffer`, after applying the final gain.

If you feel adventurous, try to turn this into a multi-tap delay on your own. I'll give some hints.

To read from a second tap with the `juce::dsp::DelayLine` object, you'd do this:

```
delayLine.popSample(0, delayInSamples * 2.0f, false);
```

Previously, `popSample` only took one argument, the channel index (0 for left, 1 for right). But here are two additional arguments:

- The second argument is the delay length. Here I simply doubled it for the second tap, but you could add another knob and plug-in parameter that lets the user set the time manually.
- The third argument should be `false`, which tells JUCE that you're reading more than one tap from the delay line.

All of this should be done after the regular `popSample`.

Go give it a try now, see if you can implement this second tap by yourself. The solution is given below.

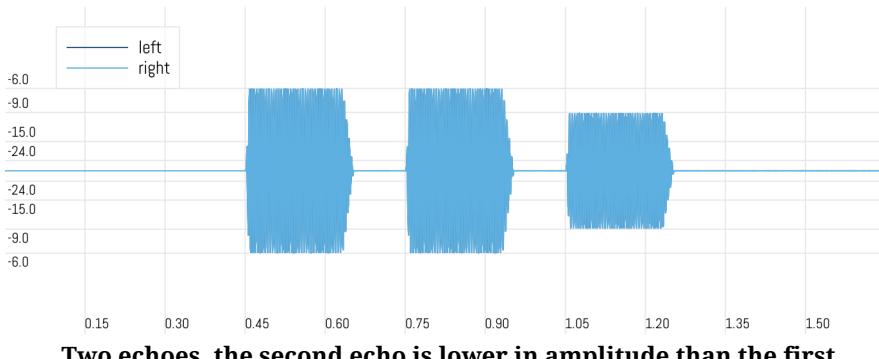
Here is one way you could implement this.

First you read from the second tap, and then add this sample to the wet signal to mix it with the delayed sample from the first tap:

```
wetL += delayLine.popSample(0, delayInSamples * 2.0f, false) * 0.7f;
```

The reason for multiplying by something like `0.7f` is to make the second echo less loud than the first echo.

To test whether your multitap delay works, use the **Beep.wav** sound file again. You should get two repetitions for every beep instead of just one. Of course, there's no need to stop at two taps, you can add even more taps if you want to.



Be careful when making the delay time very large. There only is room in the delay line for up to 5 seconds of samples. If you set the delay time longer than 2.5 seconds, the second tap will try to read further than 5 seconds in the past.

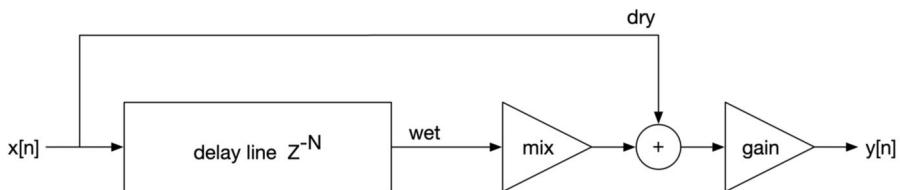
`juce::dsp::DelayLine` will trigger an assertion when this happens, crashing the plug-in. To prevent this, you'll need to change the maximum delay length `maxDelayInSamples` in `prepareToPlay` to be twice as large.

Making a multi-tap delay is a fun project, but in this chapter we'll investigate another method of getting repeating echoes: using **feedback**.

Feedback loops are an essential feature of many audio effects, such as IIR filters, dynamic range compression, sound synthesis, and of course a delay. Feedback involves taking the output of the delay and adding it back into the input signal.

The block diagram

For effects that start to get complicated in their design, it's useful to sketch out a block diagram of the signal flow inside the effect. Here is the delay that we have so far:



Block diagram of the delay plug-in

The input signal is known as $x[n]$ or just x and is on the left side of the diagram. The signal flows from left-to-right, as indicated by the arrows.

The signal splits into two parts, one of which goes into the delay line block. Delays are often notated in these kinds of diagrams as z^{-N} where N is the number of samples in the delay. This is a reference to the mathematics that underlies digital signal processing.

The signal that comes out of the delay line has been delayed by N samples. This delayed signal gets combined with the non-delayed or dry input signal.

First the delayed signal is attenuated (reduced in amplitude) by a multiplication block for the dry/wet mix percentage, represented by the `mix` triangle, and then the dry and wet signals are added together to perform the mix.

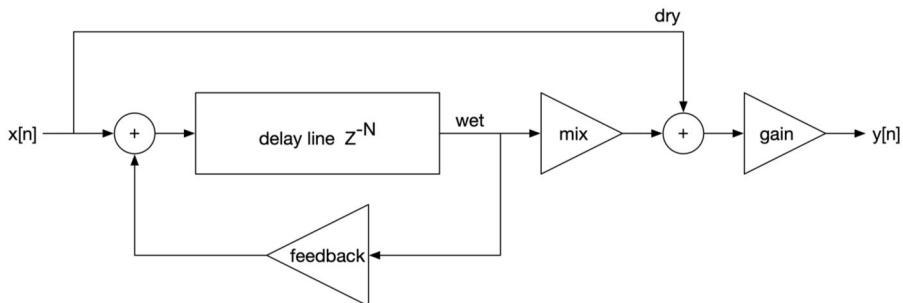
The final signal goes through another multiplication triangle that applies the output gain, and comes out on the right side of the block diagram as $y[n]$ or simply y .

Verify for yourself that what happens in this block diagram indeed corresponds to the code in `processBlock`.

The $[n]$ in the name $x[n]$ signifies this is a sampled signal, where n is the index of whatever the current sample is. It's similar to what you wrote in `processBlock` to read the incoming audio, namely `channelDataL[sample]`. This literally is the same thing as $x[n]$, just using different names.

Likewise, when you write the result into `channelDataL[sample]`, you're creating the output signal $y[n]$. The reason the block diagram uses x and y is that mathematicians love short names for their variables. Using x for the input and y for the output is a standard convention in DSP, so that's why I'm using them here as well. In the code we need "left" and "right" versions of these variables since we have two audio channels, but in the block diagram $x[n]$ stands for the entire stereo signal.

In this chapter we will expand the block diagram to have a feedback path in it:



Block diagram with feedback added

Note the direction of the arrows on the feedback path. These point the other way, from right-to-left, indicating that we're reading the output of the delay block (the wet signal) and feeding it back into the input of the delay line. The feedback path has a gain element in it that is used to attenuate the feedback signal. You will add a new parameter for this, a percentage like the dry/wet mix.

In analog electronics, when feedback is used in an electronic circuit it is instantaneous. In the digital world this is not possible, as it would lead to circular reasoning — we cannot use $y[n]$ to calculate $y[n]$. Therefore, the feedback signal will always be delayed by one timestep.

When implementing this diagram into code, the feedback signal will be the `wet` value from the last timestep, not the current one. In math terms, it's $\text{wet}[n - 1]$ rather than $\text{wet}[n]$.

Protect your ears

Warning! Feedback can be dangerous. Have you ever heard that high-pitched screaming noise when a microphone is too close to a speaker and picks up the sound, which is then played back through the speaker, which is then picked up by the microphone again, which is then amplified again, which is then... and so on. That's an example of feedback gone out-of-control.

It's possible to create that kind of feedback in software too and it can damage your speakers, or worse, your eardrums! Feedback that's carefully being applied can sound awesome, feedback that blows up is bad...

During development it's smart to protect your ears from programming mistakes that might lead to such eardrum-piercing noise. One solution is put a limiter after your plug-in in the `AudioPluginHost` session, or to use a DAW such as REAPER that will automatically turn off the sound when it detects an unwanted peak.

It's also a good idea not to prototype any new ideas while wearing headphones. Better to damage your speakers than your ears.

Just in case, you can add some code that checks that the values that are being written into the `juce::AudioBuffer` are reasonable, and if not, silences the output and prints an error message. That way you'll be notified by a benign error message rather than by EXTREMELY LOUD SOUNDS.

The **Resources** for this book come with a source file **ProtectYourEars.h**. Use the Mac Finder or Windows File Explorer to copy this file into the **Source** folder that holds the project's source code files.

Go to **Projucer** and in the **File Explorer**, right-click on the **Source** group, choose **Add Existing Files...** and select **ProtectYourEars.h**. Save the project and open the updated project in your IDE.

The contents of **ProtectYourEars.h** are as follows:

```
inline void protectYourEars(juce::AudioBuffer<float>& buffer)
{
    bool firstWarning = true;
    for (int channel = 0; channel < buffer.getNumChannels(); ++channel) {
        float* channelData = buffer.getWritePointer(channel);
        for (int sample = 0; sample < buffer.getNumSamples(); ++sample) {
            float x = channelData[sample];
            bool silence = false;
            if (std::isnan(x)) {
                DBG("!!! WARNING: nan detected in audio buffer, silencing !!!");
                silence = true;
            } else if (std::isinf(x)) {
                DBG("!!! WARNING: inf detected in audio buffer, silencing !!!");
                silence = true;
            } else if (x < -2.0f || x > 2.0f) { // screaming feedback
                DBG("!!! WARNING: sample out of range, silencing !!!");
                silence = true;
            } else if (x < -1.0f || x > 1.0f) {
                if (firstWarning) {
                    DBG("!!! WARNING: sample out of range: " << x << " !!!");
                    firstWarning = false;
                }
            }
            if (silence) {
                buffer.clear();
                return;
            }
        }
    }
}
```

This defines a function named `protectYourEars` that takes a `juce::AudioBuffer` as its argument. `protectYourEars` looks at all the samples in the buffer to see if they have acceptable values. This uses two nested loops like you've seen before: one for the channels and one for the samples in a channel.

As you know, in digital audio the sound samples should have values between `-1.0f` and `1.0f`. An amplitude of `1.0f` corresponds to a loudness of 0 dB, while smaller amplitudes correspond to negative decibel levels. For example, `0.5f` is `-6 dB`.

A sample that is above `1.0f` or below `-1.0f` is louder than 0 dB, and this may indicate a problem.

`protectYourEars` checks for the following situations:

- Are there “bad” sample values in the buffer, such as `nan` or `inf`? `nan` stands for not-a-number and `inf` stands for infinity. These are special `float` values that can occur when the code does a division by zero, for example. You want to avoid having these special values in the output audio, as they don’t sound particularly pleasant.
- Is the sound way too loud? I have defined this as sample values that are above `2.0f` or below `-2.0f`, which means the output signal is louder than 6 dB.

When bad values or too loud samples are detected, `protectYourEars` will silence the entire buffer, just to be safe. It also prints an error message to the IDE’s output pane with `DBG`, so you know why the sound suddenly dropped out.

- Is the sound louder than 0 dB, but less than 6 dB? If so, it prints a message so you know the sound went outside the range `-1.0f` to `1.0f`, but it doesn’t change the samples in the buffer. This is merely a warning that the amplitude exceeded the desirable range but is not dangerous yet.

You’ve seen the `||` operator before, but I thought I should mention it again. The statement `if (x < -1.0f || x > 1.0f)` checks whether the sample value in the variable `x` is less than `-1.0f` or is greater than `1.0f`. The `||` is the “or” in that sentence.

To print the warnings, the code does:

```
DBG("!!! WARNING: sample out of range: " << x << " !!!");
```

The `<<` symbol is the C++ operator for sending data to an output stream. Here it means that first we print the text `!!! WARNING: sample out of range:`, then it prints the value of `x`, and then the text `!!!`, followed by a newline character.

Note: It’s actually not such a big deal for a plug-in to output samples with values above `1.0f` or below `-1.0f`, in other words louder than 0 dB, as long as something else in the processing chain attenuates the signal afterwards. In a DAW that might be the next plug-in that’s on the track, or the track’s volume fader, or even the master volume fader. However, while you’re developing the plug-in it is useful to know when the sound becomes too loud. This is why we’ll enable `protectYourEars` during development, but not in the final version of the plug-in that will be given to users.

Go to **PluginProcessor.cpp** and add a new include statement below the others:

```
#include "ProtectYourEars.h"
```

Then in `processBlock`, immediately before the very last closing brace, add the lines:

```
#if JUCE_DEBUG
protectYourEars(buffer);
#endif
```

With this function in place, you can rest a bit safer, knowing that hurting your ears is much less likely now. Still, it's a good idea to turn the volume down when trying out new ideas — I often have one hand on the mute button just in case.

To test that this works, run the plug-in and feed it a relatively loud audio file. Then slowly turn up the **Output Gain** knob. At some point the sound will exceed 0 dB, and `protectYourEars` kicks in and prints the `!!! WARNING: sample out of range` message. Once the sound goes over 6 dB (you may want to turn your volume down while testing this), the output will become silent.

The `#if JUCE_DEBUG ... #endif` lines only enable the call to `protectYourEars` in debug builds of the plug-in, which is what we've been using so far. When it's time to release your plug-in into the world, you will make a so-called release build. The symbol `JUCE_DEBUG` is not defined in release builds, so there `#if JUCE_DEBUG` will be false and `protectYourEars` will never be called.

I suggest adding `protectYourEars` as the first thing when starting a new plug-in project. Even the tiniest mistake, such as forgetting to clear out additional output channels, or dividing something by zero, or using a filter with a cut-off frequency higher than the Nyquist limit (half the sample rate), can all cause nasty output. You want to catch these issues and fix them without the risk of blowing out your eardrums.

Note: The `DBG` command is very handy for printing informative messages, but you shouldn't get carried away or it will cause the audio to stutter. Recall that `processBlock` should not perform any code that may cause the audio thread to miss its deadline. `DBG` is technically speaking not allowed in the audio thread as it's rather slow. Here it's OK to use `DBG` since `protectYourEars` only prints something in an exceptional situation: the audio is too loud or has invalid values (`nan` or `inf`).

Adding the feedback loop

Now that we can rest assured our ears are safe, let's add the feedback loop into the audio processing code. In the diagram shown earlier, the feedback path starts at the output of the delay line, then it is multiplied by a gain, and finally it is added to the input of the delay line.

We need a new parameter for the feedback gain. In **Parameters.h**, add a new parameter ID below the others:

```
const juce::ParameterID feedbackParamID { "feedback", 1 };
```

In the **Parameters** class, add a new variable to the **public:** section:

```
float feedback = 0.0f;
```

In the **private:** section, add a new **AudioParameterFloat** pointer and a smoother:

```
juce::AudioParameterFloat* feedbackParam;
juce::LinearSmoothedValue<float> feedbackSmoother;
```

Excellent, switch to **Parameters.cpp** where we'll actually fill in these variables. In the constructor, do the usual type cast:

```
castParameter(apvts, feedbackParamID, feedbackParam);
```

In **createParameterLayout**, add the following code before the **return** statement:

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    feedbackParamID,
    "Feedback",
    juce::NormalisableRange<float>(0.0f, 100.0f, 1.0f),
    0.0f, // make sure this is zero!
    juce::AudioParameterFloatAttributes()
        .withStringFromValueFunction(stringFromPercent)
));
```

This is the same code that you used for the **Mix** parameter, a percentage between 0% and 100%, but with a different ID and name. Also make sure the default value is **0.0f** instead of **100.0f**, as we want the feedback to be initially turned off.

The rest of the code is likewise very similar to what you did for the **Mix** parameter. In fact, you can copy the lines and rename the variables.

In `prepareToPlay`:

```
feedbackSmoother.reset(sampleRate, duration);
```

In `reset`:

```
feedback = 0.0f;  
feedbackSmoother.setCurrentAndTargetValue(feedbackParam->get() * 0.01f);
```

In `update`:

```
feedbackSmoother.setTargetValue(feedbackParam->get() * 0.01f);
```

And finally, in `smoothen`:

```
feedback = feedbackSmoother.getNextValue();
```

The `feedback` variable now contains a value between `0.0f` and `1.0f` that is the gain that will be applied in the feedback loop.

Since the default value of the **Feedback** parameter is 0%, feedback is initially `0.0f`, which means the feedback signal is not used. Or rather, the feedback signal is always used but multiplying a signal by a gain of zero gives silence, and adding silence to the input signal will not change it — so effectively, the feedback is turned off by default.

You may be starting to realize that adding new parameters is basically always the same. We'll be doing this same dance a few more times before the end of the book. It's good practice!

While you're at it, you might as well add the knob in the editor. Go to **PluginEditor.h** and add a new `RotaryKnob` variable below the others:

```
RotaryKnob feedbackKnob { "Feedback", audioProcessor.apvts, feedbackParamID };
```

In **PluginEditor.cpp**, in the constructor for `DelayAudioProcessorEditor`, add the knob to the **Feedback** group component.

```
feedbackGroup.addAndMakeVisible(feedbackKnob);
```

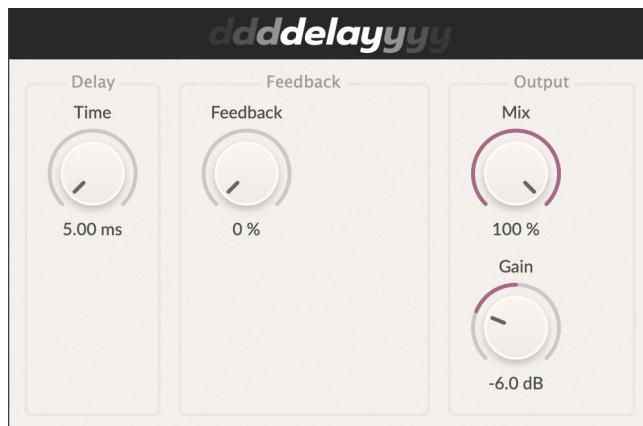
It doesn't really matter where you put this line, but I suggest placing it with the other lines that deal with `feedbackGroup`, to keep related code organized together.

Then in `resized` you need to move this knob into place. Add the following line:

```
feedbackKnob.setTopLeftPosition(20, 20);
```

That's the UI done! Adding a new knob was pretty painless, since we did all the work of making a nice `RotaryKnob` class already with a look-and-feel that knows how to draw the knob.

Build the project to make sure there are no errors. Run the plug-in in `AudioPlugin-Host` and the editor should look like this now:



The Feedback parameter and knob

Of course, the knob doesn't do anything yet, so let's write the DSP code for that. We will need to create a variable that will hold the current sample value that's traveling over the feedback path. Since we're working in stereo we need one of these variables for the left channel and one for the right channel.

In `PluginProcessor.h`, add these two variables to the `private:` section of the class:

```
float feedbackL = 0.0f;
float feedbackR = 0.0f;
```

Initially we want these variables to be zero, which is why you wrote `= 0.0f`. However, that assignment is done just once, when the audio processor is constructed.

The plug-in's audio processing code can be started and stopped many times. The `feedbackL` and `feedbackR` variables will remember the most recent feedback sample from whatever audio was previously playing. If we're starting processing again on some new audio, we don't want to keep that old feedback sample.

The host tells the plug-in that audio playback is about to start by calling the `DelayAudioProcessor`'s `prepareToPlay` function. It is there that we should reset any variables that keep state.

The proper solution therefore is to add the following lines in `PluginProcessor.cpp` in `prepareToPlay`. This clears out any old sample values from the feedback path.

```
feedbackL = 0.0f;
feedbackR = 0.0f;
```

To use the feedback, change the audio processing loop in `processBlock`, like so:

```
for (int sample = 0; sample < buffer.getNumSamples(); ++sample) {
    params.smoothen();

    float delayInSamples = params.delayTime/1000.0f * sampleRate;
    delayLine.setDelay(delayInSamples);

    float dryL = channelDataL[sample];
    float dryR = channelDataR[sample];

    // 1
    delayLine.pushSample(0, dryL + feedbackL);
    delayLine.pushSample(1, dryR + feedbackR);

    float wetL = delayLine.popSample(0);
    float wetR = delayLine.popSample(1);

    // 2
    feedbackL = wetL * params.feedback;
    feedbackR = wetR * params.feedback;

    float mixL = dryL + wetL * params.mix;
    float mixR = dryR + wetR * params.mix;

    channelDataL[sample] = mixL * params.gain;
    channelDataR[sample] = mixR * params.gain;
}
```

Everything is still the same, except for:

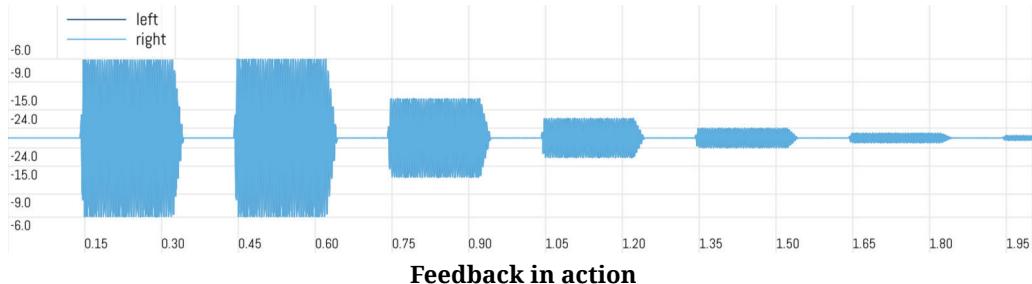
1. The `pushSample` lines now add the sample coming from the feedback path to the dry signal, and put the sum into the delay line.
2. Take the output from the delay line — the wet signal — and multiply it with the gain from `params.feedback` to get the new feedback sample.

Note that the value that we're writing into `feedbackL` and `feedbackR` isn't used until the next iteration of the loop. This is why we need to keep these variables around as member variables.

The other variables used in the loop, such as `dryL`, `wetL`, and `mixL`, are all local variables. They cease to exist when the loop ends, as we don't need them after that. But we do need to keep track of `feedbackL` and `feedbackR` from one block to the next.

That's all the code you need to add feedback to the delay. Try it out! When feedback is at 0%, the sound is repeated just once, like before. When feedback is at 50%, the sound is repeated several times until the recurrent application of the feedback gain has made the signal so small you can't hear it anymore.

Be warned: At 100%, the sound keeps repeating forever. This can massively boost the total output level, so if you're going to experiment with very high feedback values — and I know you are — turn down the **Output Gain** first.



The above image is the **Beep.wav** audio file with a delay of 300 ms, **Mix** at 100% and **Feedback** at 50%. Because the dry/wet mix is 100%, the first echo has the same amplitude as the initial beep. The other echoes are each half the height of the preceding one.

Set **Feedback** to 80% and play the audio file once (no looping) so it outputs only a single beep. Then all the other echoes you hear are from the feedback path. Notice

how long it keeps going. With feedback at 100%, the beeps keep repeating forever. (Make sure the delay time is at least 200 ms when you try this so that the echoes do not partially overlap.)

The reason a high feedback gain creates a sound that keeps going up in loudness, is that the feedback loop causes the signal to repeat forever. With a low feedback gain, the sound in the loop eventually dies out as the attenuation keeps it from going out of control.

When the feedback gain is large, and new audio is continually being added to the samples already in the delay line, the feedback loop causes the sample values in the delay line to keep growing and growing over time. The result: very loud output.

So be careful! Even with `protectYourEars`, the output from the delay with high feedback can be uncomfortable, even damaging over the long run.

Tip: With the Feedback knob set to a high percentage, audio will keep getting added to the delay line even if you stop the input. To get the plug-in to output silence again, temporarily turn Feedback down to 0%. This way the feedback path will gradually overwrite the contents of the delay line with silence.

Negative feedback

This isn't about criticism or bad reviews. Currently the feedback gain is a value between `0.0f` and `1.0f`. There's no reason we can't make it negative, all the way down to `-1.0f`. When the gain is a negative value, the delayed signal is *subtracted* from the input, rather than added to it. This can give a slightly different effect. It's pretty easy to build into our plug-in.

In `Parameters.cpp`, in `createParameterLayout`, change the code that creates the feedback parameter to the following:

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    feedbackParamID,
    "Feedback",
    juce::NormalisableRange<float>(-100.0f, 100.0f, 1.0f), // this line changed
    0.0f,
    juce::AudioParameterFloatAttributes()
        .withStringFromValueFunction(stringFromPercent)
));
```

The only thing that's different is the minimum value of the `juce::NormalisableRange` changed from `0.0f` to `-100.0f`. The default stays at `0.0f` so that feedback is initially disabled.

In `PluginEditor.h`, change the definition of `feedbackKnob` to set the `drawFromMiddle` argument to `true`, so that 0% is at the top, -100% is at the left, and 100% is at the right:

```
RotaryKnob feedbackKnob {
    "Feedback", audioProcessor.apvts, feedbackParamID, true
};
```

That's all you need to do. Previously the `feedback` variable could only have values between `0.0f` and `1.0f`, but by allowing the parameter to go down to `-100.0f`, now `feedback` can have a value between `-1.0f` and `1.0f`.

Multiplying the feedback signal with a gain less than zero will simply flip it upside down (inverting the polarity) in addition to attenuating it.

Try it out, the **Feedback** knob can now also have negative values. Compare the sound of -50% feedback to +50% and you should be able to hear a subtle difference, especially with shorter delay times.

To hear the difference with **Beep.wav**, make sure the delay time is less than 200 ms so that the echoes overlap with the original sound and each other. (If the echoes do not overlap, you won't hear a difference because a sine wave that's upside down sounds the same.)

Note: We don't want the feedback gain to go higher than 1.0 or lower than -1.0. Even at $\pm 100\%$ the effect is barely under control and the sound can rapidly grow louder over time, but at higher percentages the audio would immediately blow up.

Tip: To better understand what is happening in your plug-in — or in other people's plug-ins — you can use the handy tool [Pluginductor](#)³¹. This is a paid app but worth getting if you're serious about making plug-ins.

Here is a screenshot of the LinearAnalysis tool in IR mode (impulse response). Pluginductor sends an impulse signal into the plug-in and plots the response. An impulse is a very simple audio signal that is zero everywhere except the first sample is 1.0. Plotted in an oscilloscope it shows a single peak.

³¹<https://ddmf.eu/pluginductor/>



Analyzing the delay plug-in with Plugindocor

Since our plug-in is a delay, we would expect to see another peak after the Delay Time value chosen in the editor, here approximately 60 ms. Plugindocor confirms this is indeed the case.

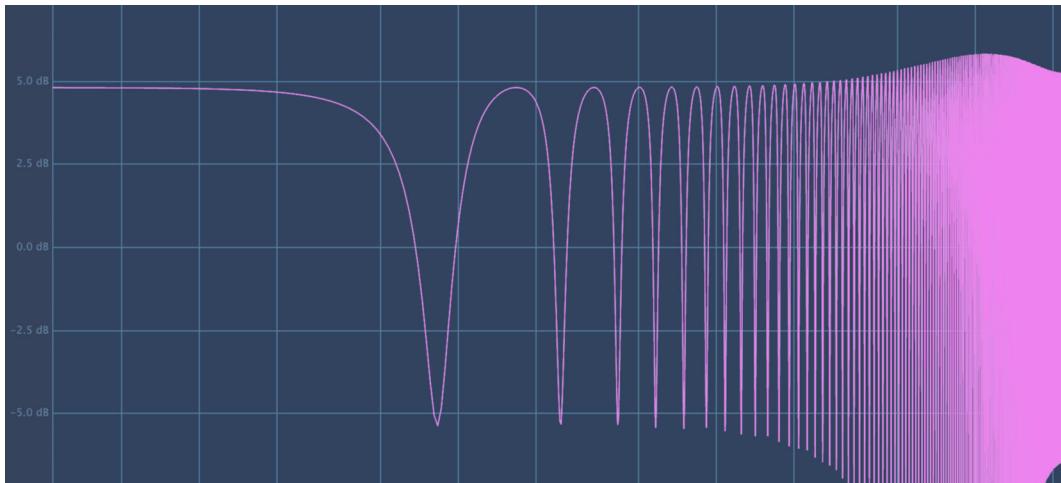
Additionally, I have feedback enabled at -50% and you can see that additional peaks are visible, spaced at multiples of the delay time, and that every other one is flipped upside down. When Feedback is a positive value, all these peaks will be positive.

So, this is a nice way to visually check that your DSP code is doing the right thing.

There are also other, smaller peaks visible. This is because of the feedback. Plugindocor doesn't just measure an impulse once, it periodically sends a new impulse, and if our feedback is high enough, the old impulses are still in the delay line, which is what causes the additional peaks.

Note that the first echo should have the same height as the initial impulse if Mix is 100% but in the screenshot it isn't. This is because the peak doesn't fall exactly on a sample but somewhere in between and Plugindocor has to interpolate (guess) the true sample value when drawing the peak. Even such a tool, handy as it is, comes with a few caveats for interpreting the results.

The Freq view of the LinearAnalysis pane is also interesting. This shows the frequency response of the plug-in. We found that at very short delay times the effect acts like a comb filter. You can see that here too. Turning the Feedback knob makes it clear that negative feedback and positive feedback have different frequency responses, which explains why they sound different.



The frequency response in Plugindocitor

If you don't want to shell out for Plugindocitor, there is the excellent [Bertom EQ Curve Analyzer³²](#) that can display these frequency response plots and is free (or donation-ware). However, it doesn't have as many features as Plugindocitor.

In the next chapter you'll do more fun stuff with feedback, such as making a true stereo delay that bounces the echoes between the left and right channels.

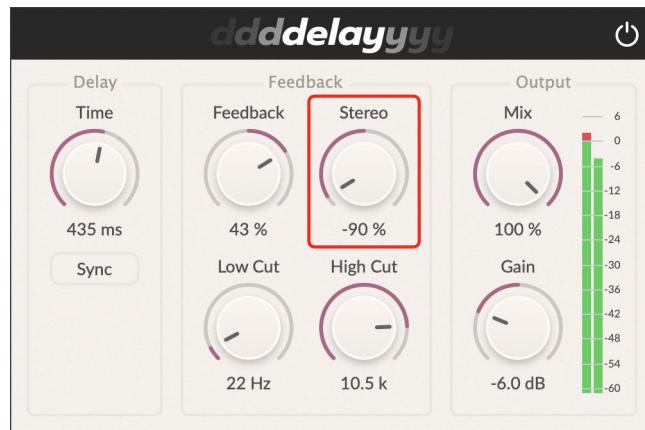
³²<https://bertomaudio.com/eq-curve-analyzer.html>

13: Ping-pong delay

Our plug-in currently works with stereo sound but it processes the channels completely independently. Many plug-ins have some kind of stereo control that makes it possible to change how wide the sound is or where it's located in the stereo image.

There are several ways to add stereo effects to a delay plug-in. For this book I've chosen to create a classic ping-pong effect.

In this chapter you'll build the following part of the plug-in:



Exploring stereo

The book's **Resources** folder contains an audio file **Beep-stereo.wav** that has a low-pitched beep on the left channel and a beep two octaves higher on the right channel. This is useful for testing how the plug-in deals with stereo inputs.

If you play this audio file through the plug-in with some feedback, you'll always hear the low beep on the left only and the high beep on the right only. It's best to test this with headphones, or by changing the balance control on your speakers to output just the left or right channel.

There's nothing wrong with a delay plug-in that has independent channels, but you could make this more interesting by giving each channel its own delay time. Here are some possible approaches:

1. Use the existing Delay Time parameter for the left channel and create another Delay Time Right parameter for the right channel.
2. Make the delay of the right channel twice as long as the delay of the left channel. Don't forget to increase the maximum length the delay line can handle if you do this.
3. To set the delay length on the left channel, subtract a certain amount from the Delay Time. For the delay length of the right channel, add that same amount. The amount to subtract and add can be set with a new parameter called Spread. At 0% spread, the difference between the two delay times is 0 ms. At 100% it could be something like 50 ms.

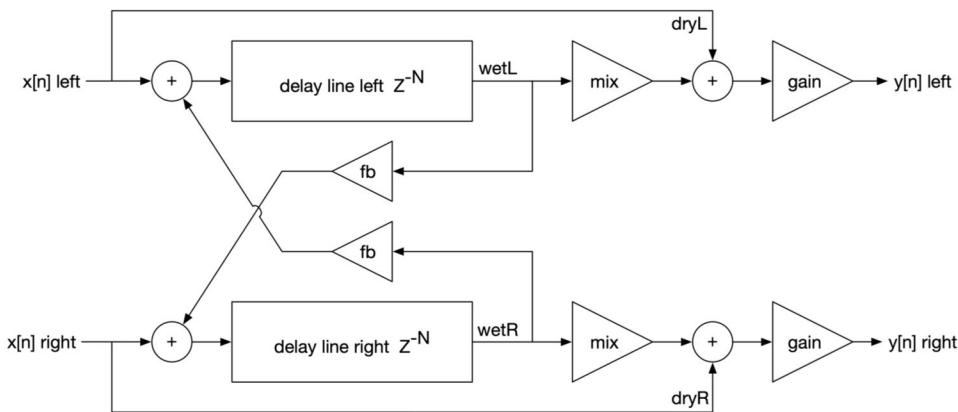
All the knowledge you need to build either of these effects has been covered in the previous chapters of the book, so maybe this is something you can try as a challenge. Good luck!

Note: If you decide to try this for yourself, note that `juce::dsp::DelayLine` uses the same delay length for both channels. Instead of doing `popSample(0)` and `popSample(1)` like before, you now must provide the delay time for the channel as an argument: `popSample(0, delayLengthLeft)` and `popSample(1, delayLengthRight)`.

Ping-pong

A ping-pong delay bounces the signal between the left and right channels. It's especially effective in combination with feedback.

There are different ways to design a ping-pong delay. For example, you could connect the left channel's feedback to the right channel's delay line input, and vice versa. In a block diagram that looks like this:



Block diagram of delay with crossed feedback paths

For clarity this block diagram shows the left and right channels individually.

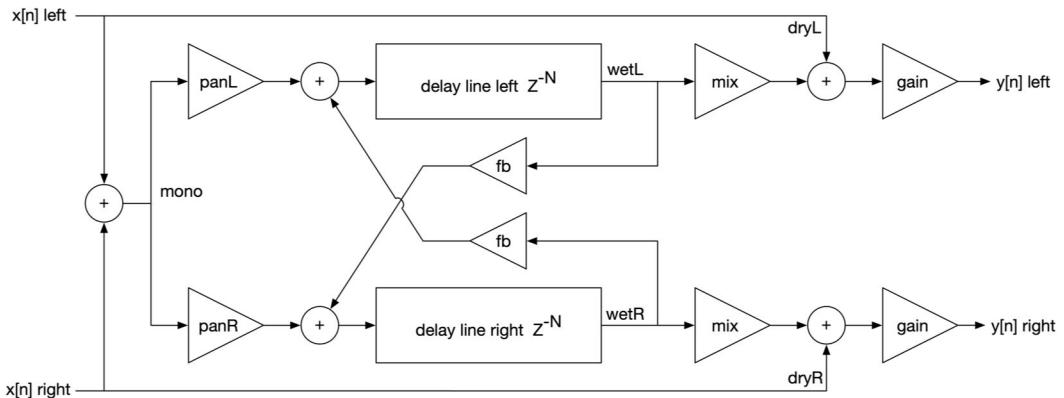
To implement this, simply change the lines in `processBlock` that push the new sample into the delay line to the following:

```
delayLine.pushSample(0, dryL + feedbackR);
delayLine.pushSample(1, dryR + feedbackL);
```

Now you're adding `feedbackR` to `dryL` and `feedbackL` to `dryR`. That's the only change needed.

Try it out with **Beep-stereo.wav** and you'll hear that the low beep alternates between the left and right channels, and the high beep between right and left. Naturally, the **Feedback** parameter should be not zero for this to work.

Crossing the feedback paths works, but I also want to have a knob that changes the stereo width of the delay. For that we will implement the following block diagram.



Ping-pong delay with stereo panning

This is very similar to the previous block diagram, but the stereo input signal is first converted to mono and then this mono signal bounces from the right channel to the left channel and back again.

In the previous approach, the contents of the left and right channels would alternate, but here the entire signal will ping-pong from left to right. This means we lose any stereo information that is in the input, but on the flip side we can control how wide we want the stereo image to be.

Converting a stereo signal to mono is simple: add the sample from the left channel to the sample from the right channel and divide by two. In other words, we are taking the average value of both channels. In practice we will multiply by 0.5 rather than divide by 2, since multiplications are a little faster.

In **PluginProcessor.cpp**, in `processBlock`, change the following part of the audio processing code:

```

float dryL = channelDataL[sample];
float dryR = channelDataR[sample];

// convert stereo to mono
float mono = (dryL + dryR) * 0.5f;

// push the mono signal into the delay line
delayLine.pushSample(0, mono + feedbackR);
delayLine.pushSample(1, mono + feedbackL);

float wetL = delayLine.popSample(0);
float wetR = delayLine.popSample(1);
    
```

The change here is that we now push the mono signal into the delay line instead of the stereo signal. Again, we add `feedbackR` to the left channel's delay line and `feedbackL` to the right channel's delay line.

However, there is a small problem with this code. Since the same sample value, `mono`, goes into both delay lines, the echoes will always appear in the center only. This does not ping-pong at all! Try it out with an audio file that has very different content on the left and right, such as **Beep-stereo.wav**, and you'll hear that all the echoes end up smack in the middle. It's best to test this with headphones on.

Instead, what we should do is pan the `mono` signal before it goes into the delay lines. This is done by multiplying it with a panning value:

```
delayLine.pushSample(0, mono*params.panL + feedbackR);
delayLine.pushSample(1, mono*params.panR + feedbackL);
```

The `params.panL` and `params.panR` variables are values between 0 and 1 that determine how much of the `mono` sample will be added to the left and right delay lines respectively.

For example, if `panL` is `1.0f` and `panR` is `0.0f`, then all of the `mono` signal goes into the left delay line and none into the right delay line. Conversely, if `panL` is `0.0f` and `panR` is `1.0f`, all of the signal will end up on the right.

Let's add these variables to the `Parameters` class. In **Parameters.h**, put these lines into the `public:` section of the class:

```
float panL = 0.0f;
float panR = 1.0f;
```

In **Parameters.cpp**, add the following to `reset()` to restore these variables to their initial value:

```
panL = 0.0f;
panR = 1.0f;
```

Now try the plug-in again. Make sure **Feedback** is at 50%, so that you can clearly hear multiple echoes. The sound will initially repeat on the right channel. Then, because the feedback from the right delay line is connected to the input of the left delay line, it will echo on the left channel. After that it goes back to the right, and so on. If **Feedback** is 0%, there is only a single echo, on the right channel.

The echo is always made up of the mono signal, which we created by summing the two input samples, plus the feedback from the other channel. So even though we insert an identical mono signal into the two delay lines, the output sound is a blend of the mono input and the stereo feedback signal, giving a spacious sound.

Pop quiz: What happens if **Feedback** is negative? Answer: It will still echo on the right side first. The mono signal will simply be flipped upside down in this case.

Stereo width

It's nice to give the user some control over how much the delayed signal will be panned, and in which direction, so we will add a new **Stereo** parameter to the plug-in that will be used to fill in `panL` and `panR`.

If you're up for a challenge, try adding this parameter yourself to the **Parameters** class. You've already done this a few times, and the process is the same as before.

This new **Stereo** parameter will be a percentage between -100% and 100%, just like the **Feedback** parameter. It will also be smoothed, although you don't need to do anything in `smoothen()` yet since that will be a bit different.

In case you didn't feel up to the challenge, or want to check your work, this is what you should have done:

In **Parameters.h**, add a new parameter ID:

```
const juce::ParameterID stereoParamID { "stereo", 1 };
```

In the **private:** section of the class, add a pointer to the **AudioParameterFloat** and add the smoother:

```
juce::AudioParameterFloat* stereoParam;
juce::LinearSmoothedValue<float> stereoSmoothener;
```

There is no need to add a new variable to the **public:** section that will hold the parameter's current value, because we'll use this parameter to fill in the existing `panL` and `panR` variables.

In **Parameters.cpp**, in the constructor, get the pointer to the parameter object:

```
castParameter(apvts, stereoParamID, stereoParam);
```

In **createParameterLayout**, add the code to create the parameter:

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    stereoParamID,
    "Stereo",
    juce::NormalisableRange<float>(-100.0f, 100.0f, 1.0f),
    0.0f,
    juce::AudioParameterFloatAttributes()
        .withStringFromValueFunction(stringFromPercent)
));
```

In **prepareToPlay**, tell the smoother about the sample rate and smoothing duration:

```
stereoSmoother.reset(sampleRate, duration);
```

In **reset**, assign the parameter's current value to the smoother:

```
stereoSmoother.setCurrentAndTargetValue(stereoParam->get() * 0.01f);
```

In **update**, change the target value of the smoother:

```
stereoSmoother.setTargetValue(stereoParam->get() * 0.01f);
```

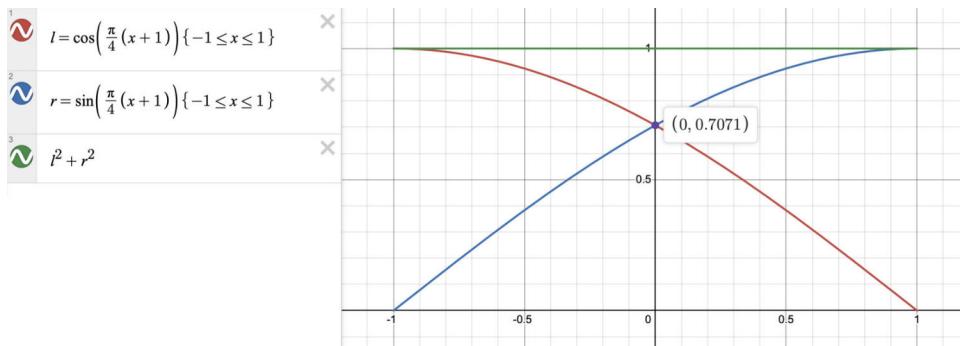
In **smoothen**, we somehow need to fill in the **panL** and **panR** variables by reading the value of the **Stereo** parameter using **stereoSmoother.getNextValue()**. For this we will turn to the so-called **panning laws**.

Stereo panning is simply a matter of changing the gain:

- If we increase the gain of the left channel and decrease the gain of the right channel, the sound appears to come more from the left speaker.
- If we do the opposite and make the right channel louder than the left, the sound appears to be panned more to the right.
- If both channels have equal gain, the sound sits in the center.

Exactly how to increase the gain in one channel while decreasing the gain in the other channel is described by the panning law. There are several different ones. We will be using so-called **equal power** panning, also known as constant power panning or -3 dB panning.

How this works is best illustrated with an image:



The panning curves for the left (red) and right (blue) channels

How to read this graph: the x-axis (the horizontal axis) goes from −1 to 1. This represents the possible settings of the **Stereo** parameter.

The red curve is the gain for the left channel for all the possible settings of the **Stereo** parameter. In other words, this describes what panL will do when the **Stereo** knob is turned from −100% to 100%. The blue curve is panR, the gain for the right channel.

- When **Stereo** is −100%, x is −1, and we're panning hard to the left. The red curve, or panL has the value 1.0 here, while the blue curve, or panR is 0.0. That makes sense: we only want to have signal in the left channel and want the right channel to be silent.
- When **Stereo** is 100%, x is 1, and the sound is panned hard to the right. Now panL (the red curve) is 0.0, turning off the sound in the left channel, and panR (the blue curve) is 1.0.
- As the **Stereo** parameter goes from −100% towards 100%, panL gradually becomes less while panR slowly increases.
- When **Stereo** is in the middle, at 0%, the red and blue curves have the same value. Since both channels are equally loud, the listener perceives the sound as being centered in the stereo field.

The reason this is called equal power panning is that if we square the `panL` and `panR` values and add them up, the sum will be 1.0 everywhere, for any setting of the **Stereo** parameter. That's what the green curve shows.

Squaring the amplitude value gives the “power” of the signal. Since the green curve is 1.0 everywhere, the power of the combined signal is the same everywhere too.

Perhaps you're wondering why the red and blue lines are curved instead of straight? This has to do with human perception of sound. With these curved lines, the perceived strength of the signal stays the same as it is panned from one side to the other.

If we didn't use the constant power formula, the sound would appear louder when panned to the sides than in the middle, which is undesirable.

As a result of this curviness, when the sound is in the center, `panL` and `panR` both have the value 0.7071. This corresponds to a gain of -3 dB, which is why this panning law is often referred to as **-3 dB panning**. It actually makes the sound a little bit quieter, but that's the price you pay for keeping the sound intensity consistent when panning to one side or the other.

Note: The above illustration was made with [Desmos](#)^a, an excellent online graphing calculator. Try it out, it's a great tool for playing with math equations and visualizing them.

^a<https://www.desmos.com/calculator>

Let's implement this equal power panning law. It will be useful to write a new function for this, named `panningEqualPower`, that you can use in other projects as well. This is a good point to start building our own little library of DSP code, so we'll put this function into a new source file.

Go to **Projucer**. In the **File Explorer**, right-click the **Source** group and choose **Add New Header File...**. Name it **DSP.h**. We're not adding a **DSP.cpp** file here since the code is small enough to go into just the header file. Save the project and open in your IDE.

Open the **DSP.h** source file and replace its contents with the following code.

```
#pragma once

#include <cmath>

inline void panningEqualPower(float panning, float& left, float& right)
{
    float x = 0.7853981633974483f * (panning + 1.0f);
    left = std::cos(x);
    right = std::sin(x);
}
```

As always there is a `#pragma once` at the top that prevents this file from being `#include`'d more than once.

We also include the `<cmath>` file, which contains the `std::cos` and `std::sin` functions that we'll need. These math functions are part of the C++ standard library.

The code inside `panningEqualPower` does a bit of math. This is what creates the curves shown in the illustration earlier. The number `0.7853981633974483f` is $\pi/4$.

The `panning` argument is a value between `-1.0f` and `1.0f`, where `-1.0f` means the sound is panned fully left, `1.0f` means it's panned fully right, and `0.0f` means it's smack in the center. In other words, this is the value from our **Stereo** parameter.

The other two arguments, `left` and `right`, are **output arguments**. You've seen functions that return a value before, but `panningEqualPower` is `void`, meaning it has no return value.

C++ functions can only return a single value, but here we want to return two values: the gain for the left channel and the gain for the right channel. The caller should pass in two variables as **reference** arguments, and the result will be written into those variables. This is why there is a `&` after `float`, which means these arguments are passed by reference.

Note: `panningEqualPower` is marked as `inline`, which is necessary when placing the complete code of a function in a header file. Without this keyword, if **DSP.h** would be included by more than one source file in the project, you'd get an error message that the function name `panningEqualPower` was used multiple times. The `inline` keyword tells the C++ compiler that these are all the same function. This kind of thing happens because for historical reasons C++ compiles all the `.cpp` files separately and then links the resulting object files together.

In **Parameters.cpp**, in `smoothen`, add the following to use this new function:

```
panningEqualPower(stereoSmoothen.getNextValue(), panL, panR);
```

This first calls `stereoSmoothen.getNextValue()` to get the current value from the **Stereo** parameter as a number between `-1.0f` and `1.0f`. It passes `panL` and `panR` as the two reference arguments, so that the gains for the left and right channel will be written into the `panL` and `panR` member variables directly. That's all you need to do to use this function.

Whoops, it won't compile. Do you know why not? Yes, you're right, you still need to include the new header file so that the C++ compiler knows what `panningEqualPower` actually does. Just checking to see if you're paying attention. Add the include at the top of **Parameters.cpp**:

```
#include "DSP.h"
```

Before we can test this, we need to add a knob for the **Stereo** parameter into the UI. In **PluginEditor.h**, add the following line to add the new knob:

```
RotaryKnob stereoKnob { "Stereo", audioProcessor.apvts, stereoParamID, true };
```

In **PluginEditor.cpp**, in the constructor, add this new knob to the **Feedback** group:

```
feedbackGroup.addAndMakeVisible(stereoKnob);
```

And in `resized`, place the **Stereo** knob to the right of the `feedbackKnob`. Add this below the line that positions the **Feedback** knob:

```
stereoKnob.setTopLeftPosition(feedbackKnob.getRight() + 20, 20);
```

Try it out! With **Stereo** at 0%, the echoes always appear in the center just like before. Put **Stereo** at 100% and you'll hear the echoes start on the right and then bounce to the left and back again. With **Stereo** at -100%, the echoes will start on the left.

To make the stereo image less wide, set the **Stereo** knob to something like 50%. Now the echoes will appear in both channels, but still more in one channel than in the other.

Note: In case you're wondering, no I did not come up with this panning formula on my own. This is the sort of thing you look up in a book or on the internet. It's a standard formula, which is why it's called a "panning law". There are other panning laws too, each with its own formula, and you can look them up and try them if you want. The one we are using here is the most common and gives good results. Your DAW will also use a panning law for its pan and balance controls, typically -3 dB panning by default. Most DAWs will let you choose between different panning laws.

Sweet, we have a pretty sweet sounding stereo ping-pong delay that adds a lot of spaciousness to the sound. But what if the input sound is mono?

What about mono inputs?

So far we've used the plug-in with stereo inputs only because `processBlock` does the following,

```
float* channelDataL = buffer.getWritePointer(0);
float* channelDataR = buffer.getWritePointer(1);
```

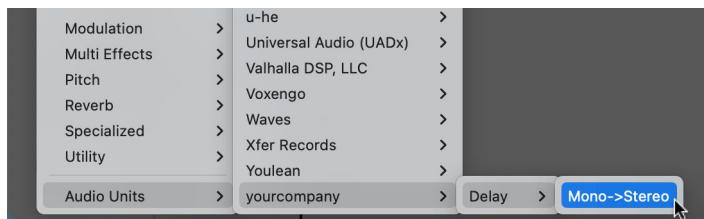
which means we're assuming there are always two channels of incoming audio data.

Many DAWs make a distinction between mono and stereo tracks and it would be nice if the plug-in worked on mono tracks too. We won't get the benefit of ping-pong in that case, but everything else should still work.

In an earlier chapter I made you replace the `isBusesLayoutSupported` function in `PluginProcessor.cpp` with the following that tells the host that this plug-in may be used on stereo tracks only:

```
bool DelayAudioProcessor::isBusesLayoutSupported(const BusesLayout& layouts) const
{
    return layouts.getMainOutputChannelSet() == juce::AudioChannelSet::stereo();
}
```

In some DAWs such as Logic Pro, if you try to insert the plug-in on a mono track anyway, it gives you the option **Mono → Stereo**, which means that the DAW will first convert the audio data from mono to stereo before it goes into your plug-in, and the track is considered to be a stereo track for everything that happens after the plug-in.



Inserting a stereo-only plug-in on a mono track in Logic Pro

Even though our plug-in only supports a stereo bus, there's no need to handle this mono-to-stereo conversion yourself. Both `channelDataL` and `channelDataR` will simply contain the same samples. Since in this situation the plug-in operates in stereo mode as usual, the ping-pong effect still works.

Note that mono can actually mean two things:

1. There is one channel: the plug-in is inserted on a mono bus. Currently we don't support this.
2. There are two channels: the plug-in is inserted on a stereo bus, but both channels contain exactly the same samples. We can use panning to move this mono sound around in the stereo field. This is what happens when the DAW offers a **Mono → Stereo** option.

Let's talk a bit more about buses (sometimes spelled "busses"). Go to **PluginProcessor.cpp** and look at the `DelayAudioProcessor` constructor again:

```
DelayAudioProcessor::DelayAudioProcessor() :
    AudioProcessor(
        BusesProperties()
            .withInput("Input", juce::AudioChannelSet::stereo(), true)
            .withOutput("Output", juce::AudioChannelSet::stereo(), true)
    ),
```

To initialize the `juce::AudioProcessor` base class, we pass in a `BusesProperties` object. The full name for this class is `juce::AudioProcessor::BusesProperties` but we can leave off the `juce::` and `AudioProcessor::` qualifiers because the C++ compiler understands what we mean when we just write `BusesProperties`.

The `BusesProperties` object describes what kind of input and output buses the plug-in has. Here we have an input bus named "Input" that can have up to two channels, since it's set to `juce::AudioChannelSet::stereo()`. Likewise for the output bus, which is named "Output". The `true` argument indicates that we want these buses to

be activated. This input and output are considered to form the **main bus**.

The above setup is typical for an effect plug-in. A synthesizer plug-in, on the other hand, wouldn't have an input bus, only an output bus. Most synths do not process incoming audio data but process MIDI commands, which they use to synthesize new audio that is placed on the output bus.

It's possible to add extra buses. For example, certain plug-ins allow you to connect a **sidechain**, which would be an additional input bus.

This is not the complete story, though. The other place where buses are configured is the `isBusesLayoutSupported` function. When the plug-in is loaded, the host will repeatedly call this function with different options to see which buses the plug-in supports. This allows the host and the plug-in to negotiate what would be the best kind of bus for the given situation.

Right now, our implementation of `isBusesLayoutSupported` returns `true` if the host asks about a stereo bus, but `false` for mono or anything else. Replace the `isBusesLayoutSupported` function with the following:

```
bool DelayAudioProcessor::isBusesLayoutSupported(const BusesLayout& layouts) const
{
    const auto mono = juce::AudioChannelSet::mono();
    const auto stereo = juce::AudioChannelSet::stereo();
    const auto mainIn = layouts.getMainInputChannelSet();
    const auto mainOut = layouts.getMainOutputChannelSet();

    if (mainIn == mono && mainOut == mono) { return true; }
    if (mainIn == mono && mainOut == stereo) { return true; }
    if (mainIn == stereo && mainOut == stereo) { return true; }

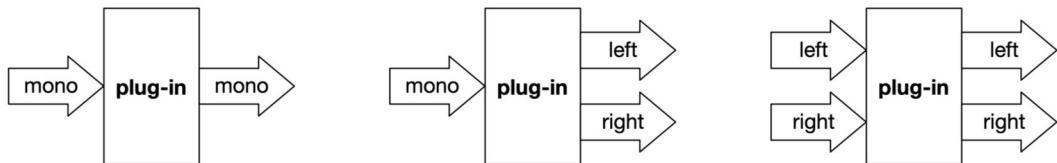
    return false;
}
```

To make this code a little easier to read, it first defines some constants. For example, `mainIn` is the same as `layouts.getMainInputChannelSet()`, which describes the input channel layout on the main bus.

The logic inside the function checks for the following situations:

- the input bus is mono and the output bus is also mono (`mono → mono`)
- the input bus is mono but the output bus is stereo (`mono → stereo`)
- the input bus is stereo and the output bus is also stereo (`stereo → stereo`)

Any other configuration is not supported, so stereo → mono is not something this plug-in will handle (I'm not sure if that is even a thing).



The bus configuration our plug-in supports

The `==` operator in C++ compares two values and outputs `true` if they are equal, or `false` if they are not. It's important that you write this with two equals signs, as a single `=` is used for assignment (putting a value into a variable). It's a common mistake to write `=` when you meant `==`.

The `&&` operator means **and**. The comparisons on both sides of the `&&` operator need to be true for the entire expression to be true. (This is different from the **or** operator `||`, for which only one side needs to be true.)

Before you can run the plug-in again, we first have to fix the audio processing code in `processBlock` because right now it has a bug when using the plug-in on a mono track!

When the input and output bus are both mono, the `juce::AudioBuffer` object will have one input channel and one output channel. In that case we cannot do `buffer.getWritePointer(1)`. This will trigger an assertion since only channel 0 will exist. Channel 1 is not available on a mono bus.

At the top of `processBlock`, it currently does the following:

```
auto totalNumInputChannels = getTotalNumInputChannels();
auto totalNumOutputChannels = getTotalNumOutputChannels();
```

We could use the `totalNumInputChannels` variable to determine how many input channels there are: 1 for mono, 2 for stereo. Likewise for `totalNumOutputChannels` and the number of output channels.

However, that won't work for plug-ins where there also is a sidechain input in addition to the main bus. So, I want to show you the code for doing it properly.

In `processBlock`, replace these lines,

```
float* channelDataL = buffer.getWritePointer(0);
float* channelDataR = buffer.getWritePointer(1);
```

with the following:

```
auto mainInput = getBusBuffer(buffer, true, 0);
auto mainInputChannels = mainInput.getNumChannels();
auto isMainInputStereo = mainInputChannels > 1;
const float* inputDataL = mainInput.getReadPointer(0);
const float* inputDataR = mainInput.getReadPointer(isMainInputStereo ? 1 : 0);

auto mainOutput = getBusBuffer(buffer, false, 0);
auto mainOutputChannels = mainOutput.getNumChannels();
auto isMainOutputStereo = mainOutputChannels > 1;
float* outputDataL = mainOutput.getWritePointer(0);
float* outputDataR = mainOutput.getWritePointer(isMainOutputStereo ? 1 : 0);
```

The `juce::AudioBuffer` object that's passed into `processBlock` contains channels for all the input buses and output buses. When you do `buffer.getNumChannels()` and the input is mono but the output is stereo, it gives 2 since that is the largest number of channels.

Sadly, the `buffer` object does not make a distinction between the number of input channels versus the number of output channels. So, using this method we don't know how many input channels the main input bus actually has.

With `getBusBuffer()` we can obtain a `juce::AudioBuffer` object that is specific to a given input or output bus. This makes `mainInput` a `juce::AudioBuffer` for the main input bus, and `mainOutput` a `juce::AudioBuffer` for the output bus. If the input bus is mono, `mainInput` will have only 1 channel.

Instead of having `channelDataL` and `channelDataR` pointers that were used for both reading and writing the audio data, we now have `inputDataL` and `inputDataR` for reading, and `outputDataL` and `outputDataR` for writing.

To support mono input and output, I'm using a little trick. The main processing loop doesn't change, it still assumes there is a left and right channel. However, if the input is mono, `inputDataR` will point to the same data as `inputDataL`.

The expression `isMainInputStereo ? 1 : 0` is a simplified version of an `if` statement. If `isMainInputStereo` is true, the result of this expression is 1, otherwise it is 0.

For a stereo input, `inputDataR` is `mainInput.getReadPointer(1)` like before. But if the input is mono, `inputDataR` is assigned `mainInput.getReadPointer(0)`, making it exactly the same as `inputDataL`. Likewise for `outputDataR` and the output bus.

In the processing loop, change these lines,

```
float dryL = channelDataL[sample];
float dryR = channelDataR[sample];
```

to use the new read pointers:

```
float dryL = inputDataL[sample];
float dryR = inputDataR[sample];
```

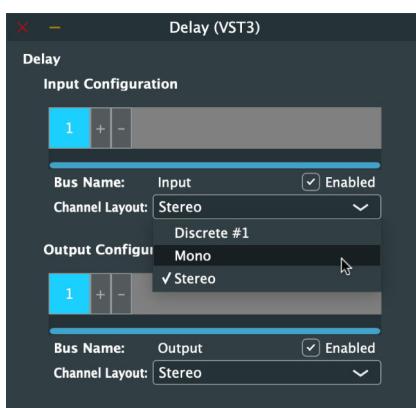
And change these lines,

```
channelDataL[sample] = mixL * params.gain;
channelDataR[sample] = mixR * params.gain;
```

to use the new write pointers:

```
outputDataL[sample] = mixL * params.gain;
outputDataR[sample] = mixR * params.gain;
```

The plug-in will now work on a mono bus as well as on a stereo bus. Nice! You can test the different channel layouts in `AudioPluginHost`. Right-click the **Delay** block and choose **Configure Audio I/O** to bring up this dialog:



Changing the input and output bus in `AudioPluginHost`

Here you can configure how many inputs and outputs the plug-in has when running in AudioPluginHost. It's a good way to test that all possible bus combinations work OK: mono-mono, mono-stereo, stereo-stereo. Note that it won't let you choose stereo in, mono out. Important: You must close this window before the changes take effect!

If you're testing in a DAW and it won't let you insert the plug-in on a mono track, it may be caching the old bus configuration somewhere. Some tricks that may help:

- Open your DAW's plug-in manager and manually re-scan the plug-in.
- The DAW may be loading an old version of the plug-in that's on your computer somewhere, so make sure you don't have any duplicates in the system plug-in folder or in the trashcan.
- If that doesn't work, go to Projucer, open the project settings panel, and change the **Project Version** number from 1.0.0 to 1.0.1. Save the project and rebuild. This will force the DAW to rescan the plug-in.
- If that didn't work, try changing the **Plugin Code** setting in Projucer too.
- If all else fails, rebooting your computer should work.

In Logic Pro, if I try to insert the plug-in on a mono track it now gives two options: **Mono** and **Mono -> Stereo**.



The plug-in can be used on a mono bus

Your DAW may also offer a **Dual Mono** option when the plug-in is on a stereo track. This runs two copies of the plug-in and lets you mix between them. To enable this option, it's only necessary that the plug-in supports a mono bus. The plug-in thinks it's in mono mode and doesn't know another instance of the plug-in is used on the other channel. So, you don't have to write any additional code to support this.

By the way, if you want to see the negotiation that goes on between your host and the plug-in, add this to `isBusesLayoutSupported` just before the `if` statements:

```
DBG("isBusesLayoutSupported, in: " << mainIn.getDescription()
    << ", out: " << mainOut.getDescription());
```

Run the plug-in from within your IDE to see what happens in `AudioPluginHost` when you choose the **Configure Audio I/O** menu item.

This prints out something like the following. That's a lot of negotiation going on!

```
isBusesLayoutSupported, in: Stereo, out: Stereo
isBusesLayoutSupported, in: Mono, out: Stereo
isBusesLayoutSupported, in: 0th Order Ambisonics, out: Stereo
isBusesLayoutSupported, in: 0th Order Ambisonics, out: 0th Order Ambisonics
isBusesLayoutSupported, in: Stereo, out: 0th Order Ambisonics
isBusesLayoutSupported, in: LCR, out: Stereo
isBusesLayoutSupported, in: LCR, out: LCR
isBusesLayoutSupported, in: Stereo, out: LCR
isBusesLayoutSupported, in: Quadraphonic, out: Stereo
...and many more lines...
```

If you want to try this with a DAW, change the executable that gets run in the debugger from `AudioPluginHost` to your DAW. As a quick reminder, for Xcode this is in the **Edit Scheme** menu. For Visual Studio, right-click **Delay_VST3** and choose **Properties**, then look under **Debugging**.

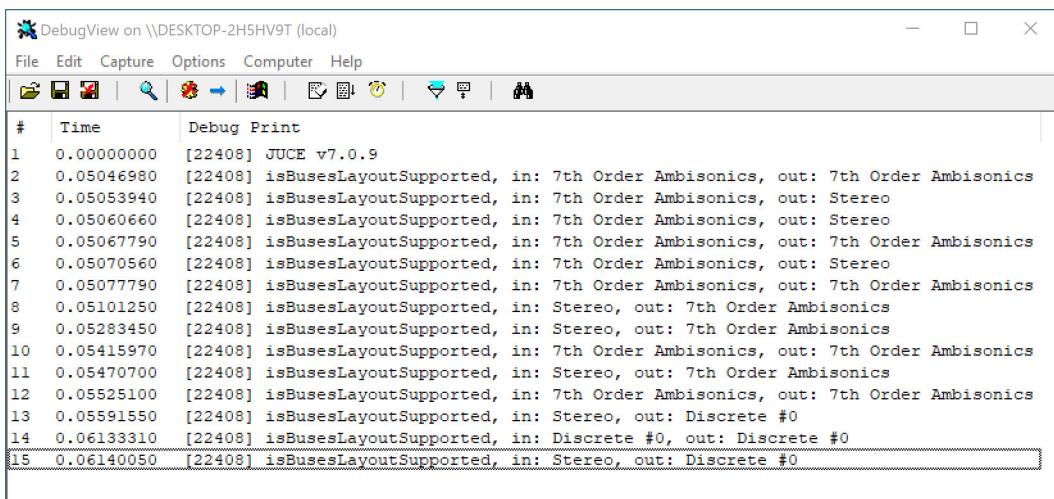
Unfortunately, the DAW — or any of the other plug-ins you have installed — may not allow a debugger to attach and can even crash when you try. So, this method won't always work.

The simplest way to see the `DBG` messages on Mac is to run the DAW from the Terminal. Any debugging information that gets printed will be printed to the Terminal window. For example, to run Logic Pro from the Terminal on macOS, write:

```
/Applications/Logic\ Pro\ X.app/Contents/MacOS/Logic\ Pro\ X
```

On Windows, the easiest method is to use a program called [DebugView³³](#). Keep in mind that this will also show debug output from other applications, not just your own.

³³<https://learn.microsoft.com/en-us/sysinternals/downloads/debugview>



The screenshot shows a window titled "DebugView on \DESKTOP-2H5HV9T (local)". The menu bar includes File, Edit, Capture, Options, Computer, and Help. Below the menu is a toolbar with various icons. The main area is a table with columns "#", "Time", and "Debug Print". The data in the table is as follows:

#	Time	Debug Print
1	0.00000000	[22408] JUCE v7.0.9
2	0.05046980	[22408] isBusesLayoutSupported, in: 7th Order Ambisonics, out: 7th Order Ambisonics
3	0.05053940	[22408] isBusesLayoutSupported, in: 7th Order Ambisonics, out: Stereo
4	0.05060660	[22408] isBusesLayoutSupported, in: 7th Order Ambisonics, out: Stereo
5	0.05067790	[22408] isBusesLayoutSupported, in: 7th Order Ambisonics, out: 7th Order Ambisonics
6	0.05070560	[22408] isBusesLayoutSupported, in: 7th Order Ambisonics, out: Stereo
7	0.05077790	[22408] isBusesLayoutSupported, in: 7th Order Ambisonics, out: 7th Order Ambisonics
8	0.05101250	[22408] isBusesLayoutSupported, in: Stereo, out: 7th Order Ambisonics
9	0.05283450	[22408] isBusesLayoutSupported, in: Stereo, out: 7th Order Ambisonics
10	0.05415970	[22408] isBusesLayoutSupported, in: 7th Order Ambisonics, out: 7th Order Ambisonics
11	0.05470700	[22408] isBusesLayoutSupported, in: Stereo, out: 7th Order Ambisonics
12	0.05525100	[22408] isBusesLayoutSupported, in: 7th Order Ambisonics, out: 7th Order Ambisonics
13	0.05591550	[22408] isBusesLayoutSupported, in: Stereo, out: Discrete #0
14	0.06133310	[22408] isBusesLayoutSupported, in: Discrete #0, out: Discrete #0
15	0.06140050	[22408] isBusesLayoutSupported, in: Stereo, out: Discrete #0

DebugView showing REAPER's format negotiation

The Stereo parameter in mono mode

While the above modifications allow the plug-in to work on mono tracks, the **Stereo** parameter makes it behave a little strangely.

On a mono bus, `dryL` and `dryR` will be identical, no problem there. However, `wetL` and `wetR` may be different, depending on the setting of the **Stereo** knob, since that parameter determines into which delay line the input signal goes.

Because `outputDataR` in mono mode points to the same sample values as `outputDataL`, and therefore overwrites them, only the samples from the right channel delay line will be output by the plug-in.

What this means is setting **Stereo** to 100% will output the first echo, the third echo, the fifth echo, and so on. Putting **Stereo** at -100% outputs the second echo, the fourth echo, and so on. For any values in between, the odd numbered echoes will have a different gain than the even numbered echoes.

Kind of weird... or perhaps kind of cool. At the very least, the name **Stereo** doesn't make much sense for the parameter now.

I'll leave it up to you whether you want to keep this "feature" or not, but let's say you don't. You can create a separate audio processing loop that handles mono mode. This won't use the **Stereo** parameter and only needs a single delay line.

In `processBlock`, you would rewrite the audio processing code as follows:

```
if (isMainOutputStereo) {
    // ... put the existing stereo audio processing loop in here...
} else {
    // this is the processing loop for mono
    for (int sample = 0; sample < buffer.getNumSamples(); ++sample) {
        params.smoothen();

        float delayInSamples = params.delayTime / 1000.0f * sampleRate;
        delayLine.setDelay(delayInSamples);

        float dry = inputDataL[sample];
        delayLine.pushSample(0, dry + feedbackL);

        float wet = delayLine.popSample(0);
        feedbackL = wet * params.feedback;

        float mix = dry + wet * params.mix;
        outputDataL[sample] = mix * params.gain;
    }
}
```

We know that if the output is stereo, the input is either stereo or mono. We want to keep handling those situations with the existing processing code. That's what the `if (isMainOutputStereo)` check is for.

But if the output is mono, the input must also be mono. In that case, we enter the simpler audio processing loop that just handles the mono situation.

The simplified loop only reads from `inputDataL` and only writes to `outputDataL`, since there's just one channel's worth of data in the audio buffer. It always uses channel 0 on the `delayLine`.

It's very common to write such a separate loop for mono operation, since otherwise we'd be processing every sample twice, which is less efficient, plus we can avoid the issue with the **Stereo** parameter, by leaving it out of this loop.

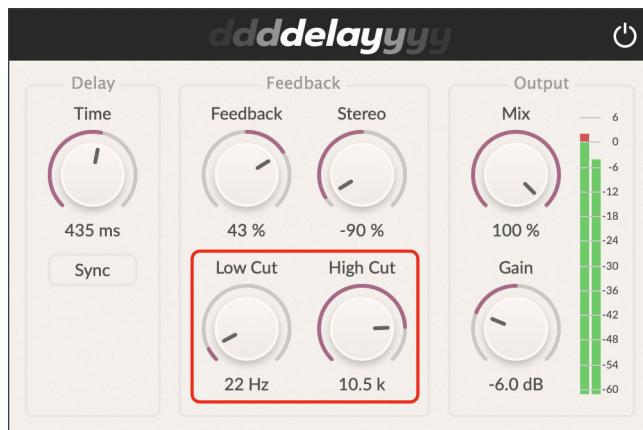
If you try the plug-in now on a mono track, the **Stereo** knob will have no effect. It would be a good idea to disable the knob in the editor, but that requires communication between potentially different threads, which is something we'll discuss later in the book.

In the next chapter, we'll continue adding features to the feedback path, a set of filters that will remove low or high frequencies from the sound.

14: Filters

The feedback path currently applies a simple gain to attenuate the feedback signal, but we can do other creative things here such as applying a filter or a waveshaper.

In this chapter you'll build the following part of the plug-in:



We will add two filters inside the feedback path: a low-cut filter to reduce the “boomy” low end, and a high-cut filter to reduce “hissy” high end. This mimics what happens on analog tape delays (loss of high frequencies), but these filters are a useful sound design tool of their own.

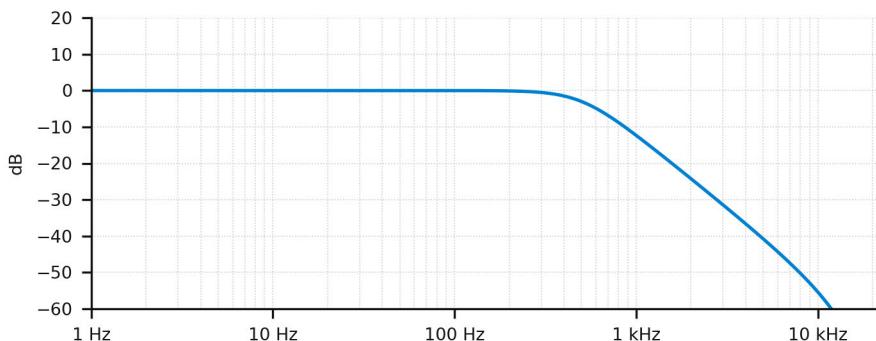
What is a filter?

A filter is used to remove frequencies from the audio signal. A **high-cut** filter, also known as a **low-pass** filter, passes through the low frequencies unchanged, but above a certain point it will fade out the higher frequencies. This point is known as the **cutoff frequency** or cutoff point.

The frequencies that are higher than the cutoff point do not completely disappear but fall off at a certain rate. This rate is known as the **slope** of the filter. For the filter we'll be using, a so-called **second order** filter, this slope is 12 dB per octave.

For example, if the cutoff is at 500 Hz, the frequencies one octave higher at 1 kHz are 12 dB quieter. Another octave higher (at 2 kHz) they are 24 dB quieter, and so on.

To describe the behavior of a filter we usually plot the **frequency response** of the filter. It lets us view how the filter affects all the possible frequencies.



The frequency response of a low-pass / high-cut filter with cutoff = 500 Hz

The horizontal axis shows the frequencies from low to high. This axis is logarithmic because we hear frequencies in a logarithmic fashion, similar to how we perceive loudness. Our ears are very sensitive to sounds with low frequencies (down to about 20 Hz), but the higher the frequency goes, the less clearly we hear the differences.

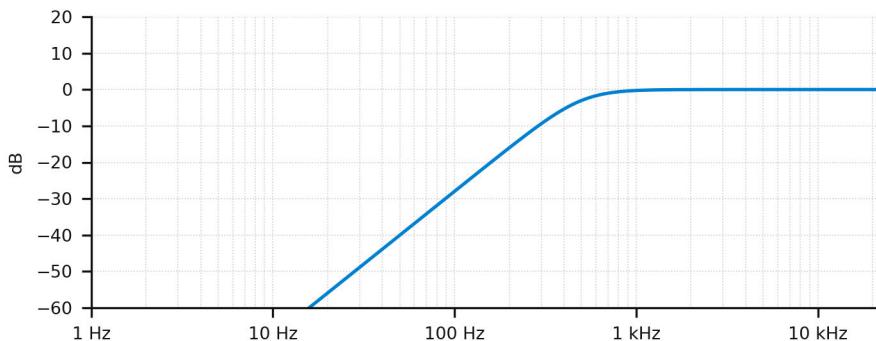
For example, the transition from a 200 Hz tone to a 300 Hz tone is extremely obvious but the same 100 Hz difference between 9800 Hz and 9900 Hz is much less noticeable.

The vertical axis shows the **magnitude response** for the frequencies, or how much the loudness of each frequency is affected by the filter. In the image above, the magnitude response is a flat line up until the cutoff frequency at 500 Hz but beyond that it quickly drops off. This axis is logarithmic too. The scale used here is the decibel scale.

- When the line is flat, at 0 dB, it means there is no change in loudness. This area of the curve is known as the **passband** because the frequencies are passed through unchanged. Since this is a low-pass filter, the passband is made up of the frequencies below the cutoff point.

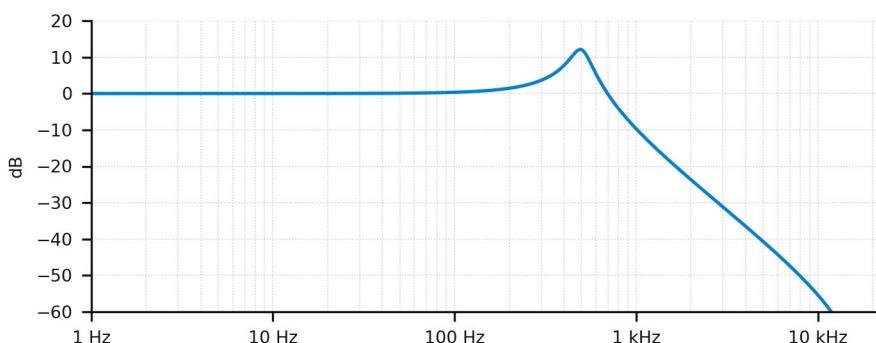
- When the line dips below 0 dB, it means those frequencies are made quieter. The area of the curve where the frequencies are reduced in amplitude, in this example anything to the right of 500 Hz, is known as the **stopband**.

A **low-cut** filter, also known as a **high-pass** filter, does the opposite: It keeps the higher frequencies but removes the low frequencies. The frequency response of a second-order low-cut filter looks like this:



The frequency response of a high-pass / low-cut filter with cutoff = 500 Hz

Filters don't just cut, they can boost certain frequencies to make them louder. A second-order filter can increase the magnitude of frequencies around the cutoff point using **resonance**, often called **Q** (short for "quality factor"). A resonant low-pass filter would look something like this:

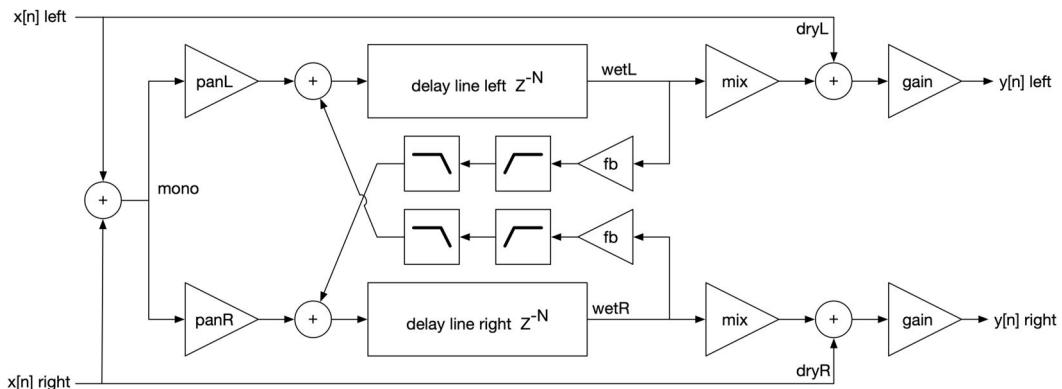


The frequency response of a low-pass filter with cutoff = 500 Hz and Q = 4

Notice the bump around the cutoff point. Anywhere the line in the frequency response plot becomes larger than 0 dB, those frequencies are made louder.

In our Delay plug-in we will not use the resonance feature of the filters. This is equivalent to setting the Q value to $1/\sqrt{2}$ or 0.707. The number 0.707 comes up a lot in audio programming — you've seen previously that this corresponds to a gain of -3 dB. For filters, using 0.707 for the Q setting means no resonance.

The new block diagram for the plug-in looks like the following. There are two filters in each feedback path now, a high-cut filter followed by a low-cut filter.



Block diagram with filters in the feedback path

JUCE filter classes

There are several ways to create a second-order low-pass filter. JUCE comes with a selection of filter objects:

- `juce::IIRFilter`
- `juce::dsp::IIR::Filter`
- `juce::dsp::FIR::Filter`
- `juce::dsp::StateVariableTPTFilter`
- `juce::dsp::StateVariable::Filter`
- `juce::dsp::LadderFilter`
- and others

It's a little messy as some of these classes do the exact same thing, such as `juce::IIRFilter` and `juce::dsp::IIRFilter`. This is because the `juce_dsp` module was introduced later and for completeness' sake it re-implements the DSP objects from other JUCE modules.

Some of the filter classes are deprecated, such as `juce::dsp::StateVariable::Filter`, which means you shouldn't use them in new code. It can be a bit confusing to figure out which filter class you actually should use.

In most DSP books you will find many a chapter dedicated to filter theory and the so-called biquadratic or **biquad** filter structure. The `juce::IIRFilter` is such a biquad. For many non-audio DSP filtering tasks, the biquad is a perfectly adequate filter. However, biquads have unfortunate side-effects when combined with the kind of thing music producers love to do: modulation.

The biquad is the workhorse of traditional DSP but for most audio-related purposes we want to use the **State Variable Filter** or SVF instead. The frequency response of the SVF is identical to that of the biquad, but what matters for audio purposes is that the SVF can be modulated without the filter blowing up.

When using a regular biquad, rapidly changing the filter's cutoff frequency may cause the internal state of the filter to go out of whack, which results in the filter's output becoming extremely large very quickly. Can you say screaming feedback? With the SVF we don't have to worry about this.

Our plug-in doesn't have any modulation facilities, but the user can still turn the knobs quickly or use their DAW's built-in automation and modulation features. In any case, for 99% of your audio filtering needs an SVF is preferred over the traditional biquad or `IIRFilter`, even if no modulation is happening.

Not all SVF implementations are equal and some older SVF code that you can find on the internet is horribly broken. JUCE uses an implementation based on Vadim Zavalishin's TPT structures that is known to be solid. To make a long story short: in our plug-in we will use `juce::dsp::StateVariableTPTFilter`.

Adding the parameters

We'll keep it simple and only add two new parameters: the cutoff frequencies for the low-cut filter and for the high-cut filter. The SVF is a second-order filter, which means we could add a resonance setting but that's left as an exercise for the reader.

In **Parameters.h**, add two new parameter IDs:

```
const juce::ParameterID lowCutParamID { "lowCut", 1 };
const juce::ParameterID highCutParamID { "highCut", 1 };
```

In the **public:** section of class **Parameters**, add two new variables:

```
float lowCut = 20.0f;
float highCut = 20000.0f;
```

These will hold the cutoff frequencies for the two filters. As usual, we give the variables some default values (the frequency in Hz) otherwise the compiler will complain they are uninitialized.

In the **private:** section of class **Parameters**, add:

```
juce::AudioParameterFloat* lowCutParam;
juce::LinearSmoothedValue<float> lowCutSmoother;

juce::AudioParameterFloat* highCutParam;
juce::LinearSmoothedValue<float> highCutSmoother;
```

Switch to **Parameters.cpp** and add these two new functions below the existing **stringFrom...** functions:

```
static juce::String stringFromHz(float value, int)
{
    if (value < 1000.0f) {
        return juce::String(int(value)) + " Hz";
    } else if (value < 10000.0f) {
        return juce::String(value / 1000.0f, 2) + " k";
    } else {
        return juce::String(value / 1000.0f, 1) + " k";
    }
}

static float hzFromString(const juce::String& str)
{
    float value = str.getFloatValue();
    if (value < 20.0f) {
        return value * 1000.0f;
    }
    return value;
}
```

The `stringFromHz` function will be used to display the cutoff frequency in Hz, or if larger than 1000, in kHz. `hzFromString` goes the other way around and will convert the value the user typed in, from a text string back to a float. Any values less than 20 will be interpreted as being in kHz.

In the Parameters constructor do:

```
castParameter(apvts, lowCutParamID, lowCutParam);
castParameter(apvts, highCutParamID, highCutParam);
```

In `createParameterLayout`, add the two new parameters:

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    lowCutParamID,
    "Low Cut",
    juce::NormalisableRange<float>(20.0f, 20000.0f, 1.0f, 0.3f),
    20.0f,
    juce::AudioParameterFloatAttributes()
        .withStringFromValueFunction(stringFromHz)
        .WithValueFromStringFunction(hzFromString)
));

layout.add(std::make_unique<juce::AudioParameterFloat>(
    highCutParamID,
    "High Cut",
    juce::NormalisableRange<float>(20.0f, 20000.0f, 1.0f, 0.3f),
    20000.0f,
    juce::AudioParameterFloatAttributes()
        .withStringFromValueFunction(stringFromHz)
        .WithValueFromStringFunction(hzFromString)
));
```

Both parameters range from 20 Hz to 20000 Hz (or 20 kHz). The default value for the low-cut filter is 20 Hz, which effectively turns it off. The default value for the high-cut filter is 20 kHz, which is high enough to be inaudible.

The `juce::NormalisableRange` objects have a skew factor of `0.3f`, giving the user more precise control over low frequencies than over high frequencies because the low ones are more important.

It is vital that the cutoff frequency of the filter does not exceed the Nyquist limit (or becomes less than zero). Recall from earlier in the book that the Nyquist limit describes the largest frequency that can be present in the signal and is always half the sample rate. Assuming that we'll never work with a sample rate lower than 44.1 kHz, the cutoff frequency must be at most 22050 Hz.

So we're good with our 20 kHz upper limit. If you ever use a cutoff that's larger than Nyquist, there's a big chance the filter will blow up and give screaming feedback. Fortunately `protectYourEars` will prevent such mistakes from doing any damage.

The smoothers are business as usual. In `prepareToPlay`:

```
lowCutSmoother.reset(sampleRate, duration);
highCutSmoother.reset(sampleRate, duration);
```

In `reset`:

```
lowCut = 20.0f;
lowCutSmoother.setCurrentAndTargetValue(lowCutParam->get());

highCut = 20000.0f;
highCutSmoother.setCurrentAndTargetValue(highCutParam->get());
```

In `update`:

```
lowCutSmoother.setTargetValue(lowCutParam->get());
highCutSmoother.setTargetValue(highCutParam->get());
```

And in `smoothen`:

```
lowCut = lowCutSmoother.getNextValue();
highCut = highCutSmoother.getNextValue();
```

Let's also add the knobs to the user interface. In `PluginEditor.h`, add the following lines:

```
RotaryKnob lowCutKnob { "Low Cut", audioProcessor.apvts, lowCutParamID };
RotaryKnob highCutKnob { "High Cut", audioProcessor.apvts, highCutParamID };
```

In `PluginEditor.cpp`, in the constructor, add the new knob components to the **Feedback** group:

```
feedbackGroup.addAndMakeVisible(lowCutKnob);
feedbackGroup.addAndMakeVisible(highCutKnob);
```

And in `resize`, add the following lines below the layout code for the feedback and stereo knobs.

```
lowCutKnob.setTopLeftPosition(feedbackKnob.getX(), feedbackKnob.getBottom() + 10);
highCutKnob.setTopLeftPosition(lowCutKnob.getRight() + 20, lowCutKnob.getY());
```

Try it out. The new editor looks like this:



The filter knobs in the editor

Notice how the Hz value below the knob changes to “k” (for kHz) for frequencies larger than 1000 Hz. Between 1 and 10 kHz the value has two digits behind the decimal point. Frequencies over 10 kHz only show a single decimal digit. Also try typing in some values.

I hope you’re getting the hang of adding parameters by now!

Filter that sound

Let’s add the actual filtering code. This is quite simple, thanks to JUCE’s DSP module.

Go to **PluginProcessor.h** and in the **private:** section add the following lines:

```
juce::dsp::StateVariableTPTFilter<float> lowCutFilter;
juce::dsp::StateVariableTPTFilter<float> highCutFilter;
```

This adds two new `juce::dsp::StateVariableTPTFilter<float>` objects to the class. Using these is very similar to the `juce::dsp::DelayLine` object. All the objects from the JUCE DSP module are designed to be used in much the same way.

The `StateVariableTPTFilter` can be configured to operate as a lowpass, bandpass, and highpass filter. Since this never has to change we can do this in the constructor of `DelayAudioProcessor`.

In **PluginProcessor.cpp**, add the following lines to the constructor:

```
lowCutFilter.setType(juce::dsp::StateVariableTPTFilterType::highpass);
highCutFilter.setType(juce::dsp::StateVariableTPTFilterType::lowpass);
```

I'm sure you figured out where to put it, but if not, this goes in between the constructor's { and } braces. Previously we didn't have anything to do for the `DelayAudioProcessor` constructor, except initialize the base class and the `params` variable in the member initializer list. Now it will also run some code.

Notice that the low-cut filter is a `highpass` type filter, and the high-cut filter is a `lowpass` filter. These names can be a bit confusing, but cutting out the low end is the same as keeping only the high end, and vice versa.

When using these two filters in series like we're doing here in the feedback path, I personally find it more logical to use the terms low-cut and high-cut, especially since their goal is to remove frequencies on the low end and high end.

The JUCE DSP objects must always be prepared before they can be used. This happens in `prepareToPlay`. Previously you created a `juce::dsp::ProcessSpec` object and used it to set up the `delayLine`. You can use the same “spec” for the filters.

Add these lines to `prepareToPlay` in **PluginProcessor.cpp**:

```
lowCutFilter.prepare(spec);
lowCutFilter.reset();

highCutFilter.prepare(spec);
highCutFilter.reset();
```

The filters must also be reset at this point. Filters, like many other DSP building blocks, will keep internal state. In the case of the filter, this state consists of several previous sample values.

When `prepareToPlay` is called, we must flush out that old state otherwise there might be a glitch when playback restarts. Such a glitch happens because the old state — the sample values from some audio that was playing previously — are no longer relevant.

Forgetting to `reset()` your DSP objects to clear out the old state is a common mistake. If you ever get glitches when you stop the audio and then start playing some new audio, make sure you're resetting all your DSP objects.

Now comes the fun part, the audio processing code. Add the following to the top of the processing loop in `processBlock`:

```
for (int sample = 0; sample < buffer.getNumSamples(); ++sample) {
    params.smoothen();

    float delayInSamples = params.delayTime/1000.0f * sampleRate;
    delayLine.setDelay(delayInSamples);

    // add these two lines:
    lowCutFilter.setCutoffFrequency(params.lowCut);
    highCutFilter.setCutoffFrequency(params.highCut);

    // ...the rest of the code...
```

This sets the cutoff frequencies for both filters. Since the `params.lowCut` and `params.highCut` variables are smoothed, we need to call `setCutoffFrequency()` on every sample timestep.

The filters will be applied to the feedback signal, so change the following lines,

```
feedbackL = wetL * params.feedback;
feedbackR = wetR * params.feedback;
```

into:

```
feedbackL = wetL * params.feedback;
feedbackL = lowCutFilter.processSample(0, feedbackL);
feedbackL = highCutFilter.processSample(0, feedbackL);

feedbackR = wetR * params.feedback;
feedbackR = lowCutFilter.processSample(1, feedbackR);
feedbackR = highCutFilter.processSample(1, feedbackR);
```

After reading the output from the delay line and applying the feedback gain, this sends the feedback signal through the low-cut filter and then through the high-cut filter. We do the same for the right channel, using channel index 1 instead of 0.

That's all you need to do to get a working filter! Try it out.

Use an audio file with a lot of high-end content. A drum track is great since it has lows for the kick and highs for the hihats and cymbals. Use a short **Delay Time**, say 50 ms. Turn up the **Feedback** to 90% and make sure **Gain** is low enough so that the sound doesn't distort.

Slowly reduce the **High Cut** knob. You should hear the echoes become less bright. Similarly, if you turn up the **Low Cut** knob, the bass should disappear from the echoes. It can be a subtle effect, so do this with headphones on and give those ears some exercise.

It's possible to put the **Low Cut** knob's frequency higher than the **High Cut** knob. This will effectively filter out the entire feedback signal and is equivalent to using no feedback at all.

Note: The order in which we apply the feedback gain and the two filters is not important. Each of these operations is LTI or **linear time invariant**. I won't go into the specifics here, but one of the properties of LTI systems is that the order in which you apply them does not matter. We could have put the high-cut filter before the low-cut filter, or the feedback gain at the end, and it would give the exact same results.

More efficient parameter updates

Up until now we've always recalculated all the plug-in parameter-based variables every time `smoothen()` is called, which is on every sample timestep. For `gain`, `mix`, and `feedback` this was just a matter of asking the smoother for the next value, and `delayTime` does its smoothing with a one-pole filter.

But for `panL` and `panR` it involves a `std::sin` and `std::cos` operation on every timestep, and for the filters it's even more work. Since most of the time the parameter values don't actually change — only when the user turns a knob or the host uses automation — we don't need to perform these calculations all the time.

We need some way to know if a parameter changed or not since the last block. The solution is to keep track of this ourselves with some extra variables. Here, I will show how to do this for the filters.

In `PluginProcessor.h`, add these lines to the `private:` section of the class:

```
float lastLowCut = -1.0f;
float lastHighCut = -1.0f;
```

We're using `-1.0f` as a special value that says, "no cutoff frequency set yet."

In **PluginProcessor.cpp**, in `prepareToPlay` set these variables back to their initial values:

```
lastLowCut = -1.0f;
lastHighCut = -1.0f;
```

In `processBlock`, change the lines that set the cutoff frequency of the filters from this,

```
lowCutFilter.setCutoffFrequency(params.lowCut);
highCutFilter.setCutoffFrequency(params.highCut);
```

into this:

```
if (params.lowCut != lastLowCut) {
    lowCutFilter.setCutoffFrequency(params.lowCut);
    lastLowCut = params.lowCut;
}
if (params.highCut != lastHighCut) {
    highCutFilter.setCutoffFrequency(params.highCut);
    lastHighCut = params.highCut;
}
```

This only calls `setCutoffFrequency` if `params.lowCut` or `params.highCut` changed from last time. This uses the `!=` operator, or **not equals**, to compare the values of two variables.

Note: Using lots of `if` statements inside the audio processing loop isn't ideal, as branching will slow down the processing and we want this to be as fast as possible. However, the CPU in your computer has a so-called branch predictor. If this predicts correctly whether `params.lowCut` is going to be unequal to `lastLowCut`, then these `if` statements will be very fast. Most of the time the filter frequencies won't be changing, so the CPU should generally predict that we can skip the `if` statements.

Xcode now gives a warning: "Comparing floating point with `==` or `!=` is unsafe." Visual Studio doesn't really care. What's that warning about?

Floating-point numbers are used to store values with a decimal point, but they are only an approximation of the actual number and don't always behave like regular numbers. For example, in math $1.1 \times 3 = 3.3$ but in code the expression `1.1f * 3.0f == 3.3f` evaluates to `false`.

That's because `1.1f * 3.0f` is `3.30000019f`, but `3.3f` is stored in memory as `3.29999995f`. This limitation of the precision of floating-point numbers is often surprising to new programmers, but it's something you'll need take into consideration when writing code that uses `floats`.

This is why the C++ compiler warns that using `==` or `!=` with floating-point numbers is unsafe. It's better to use `juce::approximatelyEqual` to compare two `floats`. However, for our purposes, where we simply compare the new value to the old value, the comparison is fine. The lesson is: compiler warnings can be helpful but they should not be blindly followed.

Build and run to make sure everything still works. All right, this plug-in is starting to sound good!

If you're in the mood for a challenge, add a new plug-in parameter for controlling the filter's resonance. Since the SVF is a second-order filter, it can add a boost to the frequencies surrounding the cutoff frequency. Higher resonance values give a higher, narrower peak.

You could have a single `Q` or `Reso` parameter that is used by both filters, or give each filter its own parameter. To change the resonance of the filter, do `filter.setResonance(resonance);` with `resonance` being a value between `0.5f` and `10.0f`. (You'll want to make the editor window larger to make the new knob fit.)

Besides the filters, you can add other DSP building blocks in the feedback path, such as a waveshaper that adds some distortion or saturation to the sound. This is a non-linear operation, which means the position of the waveshaper matters to the sound.

It's typical to add a filter before and after the waveshaper, so perhaps putting the waveshaper in between the two filters is a good place.

Here's your second challenge: Look up how to make a waveshaper on the internet or use the `juce::dsp::WaveShaper` class, and add this in between the two filters. You may need to add a new parameter that controls the amount of distortion, usually named `Drive`. Good luck!

Block-based processing

Before we conclude this chapter, I want to quickly mention an alternative approach to writing DSP code. The way we wrote the audio processing loop in `processBlock` is not the only way to do it. You may encounter the other way in online tutorials and videos, so I want you to at least have seen the differences between the two approaches.

There are two ways to write the code in `processBlock`:

1. using sample-by-sample processing
2. using block-based processing

What we've been doing in this book is sample-by-sample processing, where there is a big loop that steps through all the samples in the buffer one-by-one, like so:

```
for (int sample = 0; sample < buffer.getNumSamples(); ++sample) {
    // read the sample
    // do some processing on it
    // write the sample
}
```

Personally, I find this easiest to understand as you can see exactly what happens at every timestep. However, block-based processing can be more efficient.

You've already seen an example of block-based processing in an earlier chapter where we wrote:

```
buffer.applyGain(gain);
```

With a single command this changed all the samples in the `AudioBuffer`. There is still a loop but it's inside the `applyGain` function.

`juce::dsp::DelayLine` can also process the entire block. Instead of calling `pushSample` to write one sample at a time and `popSample` to read one sample, you do `delayLine.process(block)` to write the entire block into the delay line at once, and at the same time read a block's worth of delayed signal back out of the delay line.

Likewise for `juce::dsp::StateVariableFilterTPT`. By doing `filter.process(block)` instead of `processSample`, it applies the filter to all the samples in one go.

Writing our audio processing code in a block-based manner would look something like this (this is pseudo code, it will not actually compile):

```
// 1. smooth the parameter updates into their own buffers
smoothen(params.mixSmoothen, mixParamBuffer);
smoothen(params.gainSmoothen, gainParamBuffer);

// 2. convert the entire input buffer to mono, write result into a new monoBuffer
convertToMono(inputDataL, inputDataR, monoBuffer);

// 3. apply planning to mono signal, write to temporary buffer
multiply(monoBuffer, panLParamBuffer, wetBufferL);
multiply(monoBuffer, panRParamBuffer, wetBufferR);

// 4. write to delay line and read from it, overwrites the wet buffer
delayLine.process(wetBufferL);
delayLine.process(wetBufferR);

// 5. multiply wet signal with mix amount and add to input signal
dryWetMix(inputDataL, wetBufferL, mixParamBuffer);
dryWetMix(inputDataR, wetBufferR, mixParamBuffer);

// 6. multiply mixed signal with output gain and store in output buffer
multiply(wetBufferL, gainParamBuffer, outputDataL);
multiply(wetBufferR, gainParamBuffer, outputDataR);
```

There is no explicit loop anymore, although it would be more correct to say that each of these functions now performs its own loop through the sample data.

The reason block-based processing can be more efficient is because of vectorization. The CPU in your computer has special hardware that can process multiple floating-point numbers at once, typically 4 or 8.

For example, `buffer.applyGain` can modify 4 or 8 samples for the price of a single operation. In theory it is 4 – 8 times faster than our sample-by-sample loop.

This is known as **SIMD**, which stands for Single Instruction Multiple Data. Block-based processing is often done using SIMD instructions, hence the extra speed. It does require temporary `juce::AudioBuffer` objects to store the results of intermediate operations.

Unfortunately, not all audio processing code can be expressed as operations over the entire block. You may have noticed that feedback was missing from the above pseudo code. With feedback, every next sample depends on the previous sample. So we can't process multiple samples at the same time.

Likewise, the filter's state is updated based on the previous samples — the filter itself uses feedback internally. The `juce::dsp::StateVariableTPTFilter`'s `process` function that works on an entire block of audio simply calls `processSample` in a loop just like we did. We don't gain any efficiency there.

This is why we're using sample-by-sample processing in this book, as the feedback makes it tricky to do the entire processing loop in a block-based manner.

Also note that `delayLine.process(block)` assumes that the delay time is not going to change over the course of the block. That makes it impossible to smoothen the delay time parameter. So that's another disadvantage of block-based processing, at least for JUCE's delay line implementation. (You could write your own `DelayLine` class that doesn't have this limitation.)

I just wanted to point out that you may come across audio processing code that's written in a block-based manner, so now you at least have seen how and why.

In general, I recommend starting with the sample-by-sample approach, since it's simpler, and only convert the audio processing code to block-based once you're happy with your DSP — and if possible.

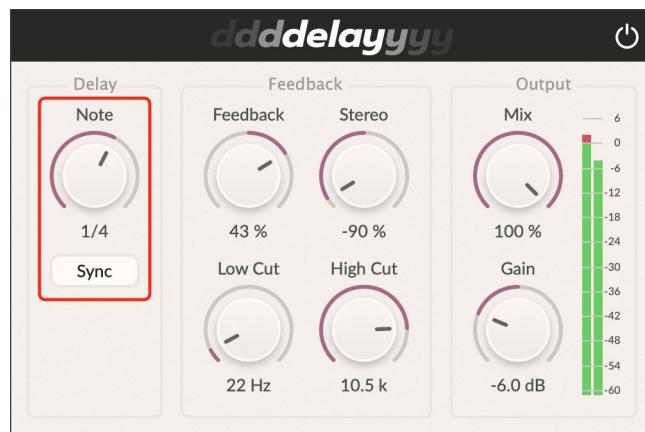
Besides, C++ compilers are pretty smart these days and will already try to convert your loops into vectorized form where they can. So it may not even be necessary to vectorize the code yourself, and you may actually do a worse job than the compiler if you try!

In the next chapter we'll add a cool new feature to the plug-in: syncing the delay time to the tempo of the song.

15: Tempo sync

The delay length is currently set as a time in milliseconds or seconds. In a musical context, it can be handy to set the delay length relative to the tempo of the project. This is known as tempo sync, beat sync, or BPM sync, as it's relative to the beats per minute setting in the DAW.

In this chapter you'll add tempo sync to the plug-in, which involves these controls:



Setting the note length

Instead of choosing the delay length as a time in seconds, the user can now set it as a note length, for example a quarter note or an eighth note. That way the delay length depends on the song's tempo. If the tempo changes — perhaps using automation — the delay length automatically changes with it.

For our plug-in we'll let the user choose between the following note lengths:

1/32	1/16	1/8	1/4	1/2	1/1
1/16 triplet	1/8 triplet	1/4 triplet	1/2 triplet	1/1 triplet	
1/32 dotted	1/16 dotted	1/8 dotted	1/4 dotted	1/2 dotted	

The possible note lengths

The shortest note is a 32nd note and the longest is a whole note, or one bar in 4/4 time. The list also includes triplets and dotted notes. A triplet is three notes in the space of one, a dotted note is one-and-a-half times the regular length.

Contrary to the other parameters that we've used, this one is not a continuous range between a minimum and maximum value but a set of distinct choices.

The `AudioParameterFloat` object is only suitable for continuous parameters. For a parameter that allows the user to select an option from a list of fixed choices, it's better to use an `AudioParameterChoice` object. We'll call this parameter **Delay Note**.

We also need a button to toggle between regular delay time and tempo sync modes. We'll call this parameter **Tempo Sync** and it's either off (use regular time) or on (use tempo sync).

If there are only two choices for a parameter, we can use the `AudioParameterBool` class. Recall that "bool" stands for boolean and the two choices are true for on, and false for off.

Let's add both these parameters. First in `Parameters.h`, add new parameter IDs as usual:

```
const juce::ParameterID tempoSyncParamID { "tempoSync", 1 };
const juce::ParameterID delayNoteParamID { "delayNote", 1 };
```

In the `private:` section of the class, add the following lines:

```
juce::AudioParameterBool* tempoSyncParam;
juce::AudioParameterChoice* delayNoteParam;
```

Notice that we're using `AudioParameterBool` and `AudioParameterChoice` now, not `AudioParameterFloat`.

There is no need to add a smoother for these parameters. After all, what would it mean to smoothen a change from true to false, or from a 1/4th note to a 1/2nd note? These new parameters offer discrete choices, and we only need to smoothen parameters that are continuous.

In **Parameters.cpp** in the constructor, add these lines:

```
castParameter(apvts, tempoSyncParamID, tempoSyncParam);
castParameter(apvts, delayNoteParamID, delayNoteParam);
```

Previously we used the `castParameter` function with `AudioParameterFloat` objects, but if you look at the source code for this function you'll see that it starts with the keywords `template<typename T>`. This `T` can stand for `AudioParameterFloat` but also `AudioParameterBool` or `AudioParameterChoice`.

Because this function is declared as being a template, the C++ compiler will actually write three different versions of `castParameter` with the following signatures:

```
static void castParameter(juce::AudioProcessorValueTreeState& apvts,
                           const juce::ParameterID& id,
                           AudioParameterFloat& destination)

static void castParameter(juce::AudioProcessorValueTreeState& apvts,
                           const juce::ParameterID& id,
                           AudioParameterBool& destination)

static void castParameter(juce::AudioProcessorValueTreeState& apvts,
                           const juce::ParameterID& id,
                           AudioParameterChoice& destination)
```

These functions all do the same thing but differ in the type of the destination argument, a technique known as **function overloading**. We could also have written these three variations by hand but it's easier (and less error-prone) to specify that the function is a template, so that the C++ compiler can write these variations for you.

In `createParameterLayout`, add the following to create the **Tempo Sync** parameter (you can put this all on one line but that didn't fit in the book):

```
layout.add(std::make_unique<juce::AudioParameterBool>(
    tempoSyncParamID, "Tempo Sync", false));
```

This has fewer arguments than `AudioParameterFloat`, since a boolean can only be true or false. We do need to specify the default value which we set to `false` here.

The other parameter is a bit more work. Add this code below:

```
juce::StringArray noteLengths = {
    "1/32",
    "1/16 trip",
    "1/32 dot",
    "1/16",
    "1/8 trip",
    "1/16 dot",
    "1/8",
    "1/4 trip",
    "1/8 dot",
    "1/4",
    "1/2 trip",
    "1/4 dot",
    "1/2",
    "1/1 trip",
    "1/2 dot",
    "1/1",
};

layout.add(std::make_unique<juce::AudioParameterChoice>(
    delayNoteParamID, "Delay Note", noteLengths, 9));
```

The `AudioParameterChoice` object takes a `juce::StringArray` object that lists all the possible rhythmic divisions. The last argument, 9, is the index of the default option, which here is "1/4" or a quarter note.

Next, we need to have some way to read the selected note length from the parameter. In **Parameters.h**, in the `public:` section of the class, add two new variables:

```
int delayNote = 0;
bool tempoSync = false;
```

Whereas the other smoothed parameter values are all of type `float`, here it needs to be an `int` and a `bool`. The `delayNote` variable will store the index of the selected element in the list. `tempoSync` is `true` when **Tempo Sync** is enabled and `false` when disabled.

In **Parameters.cpp**, in `update`, add the following lines:

```
delayNote = delayNoteParam->getIndex();
tempoSync = tempoSyncParam->get();
```

There is no need to smoothen these parameters on every sample timestep, so just reading the value once per block is enough. For `AudioParameterBool` and `AudioParameterFloat` objects, the `get()` function reads the parameter's value, but for `AudioParameterChoice` you need to use `getIndex()`.

Let's build the UI. The note length knob is easy, you've done this several times before already. In `PluginEditor.h`, create a new `RotaryKnob` object:

```
RotaryKnob delayNoteKnob { "Note", audioProcessor.apvts, delayNoteParamID };
```

In `PluginEditor.cpp`, in the constructor, add the knob into the **Delay** group:

```
delayGroup.addAndMakeVisible(delayNoteKnob);
```

We still need to move this knob into place in `resized`, but hold off on that for a minute.

First we'll add a UI component for the **Tempo Sync** parameter, so that the user can choose which of the **Time** and **Note** knobs is actually used for setting the delay length. We will make this a toggle button.

In `PluginEditor.h`, add the following line below the knobs, in the `private:` section of the class:

```
juce::TextButton tempoSyncButton;
```

As you may have guessed, the `juce::TextButton` object is a button with text on it. We need to attach this button component to the actual plug-in parameter. For the knobs this used an APVTS `SliderAttachment` object (see the `RotaryKnob` class). Here we will do something similar but using a different kind of attachment object, since we're not attaching the parameter to a slider but to a button.

Put the following somewhere below the `tempoSyncButton` line that you just added:

```
juce::AudioProcessorValueTreeState::ButtonAttachment tempoSyncAttachment {
    audioProcessor.apvts, tempoSyncParamID.getParamID(), tempoSyncButton
};
```

Recall that the attachment needs to have a reference to the APVTS object, which we can get from the `audioProcessor`. Note that `audioProcessor` is also a member variable of this class and since that member variable appears higher up in the header file, it is initialized before the attachment object gets created. That's a good thing.

If the `tempoSyncAttachment` variable was put above the line `DelayAudioProcessor& audioProcessor`, this code would crash, as in that case `audioProcessor` would not have been initialized yet and so `audioProcessor.apvts` would refer to junk memory.

Just something to keep in mind: the order in which you add variables to a class can be important.

The other two arguments for the attachment's constructor are the `juce::ParameterID` of the parameter and the `juce::Button` object to attach to. The `tempoSyncButton` is a `juce::TextButton` object but the `ButtonAttachment` expects a `juce::Button`. So why does this not give a compiler error?

`juce::TextButton` is a **subclass** of `juce::Button`, meaning that every `TextButton` is also a `Button`. This a fundamental property in object-oriented programming: anywhere a certain class is expected you can use a class that is based on it.

Recall that the attachment object must be placed in the header file below the component it's attached to, so that the attachment gets destroyed before its component when the user closes the editor window and the editor is deallocated.

Next, in `PluginEditor.cpp`, in the constructor, add the following lines (anywhere as long as it's before `setSize`):

```
tempoSyncButton.setButtonText("Sync");
tempoSyncButton.setClickingTogglesState(true);
tempoSyncButton.setBounds(0, 0, 70, 27);
tempoSyncButton.setColour(juce::TextButton::ColourIds::buttonOnColourId,
                         juce::Colours::red);
delayGroup.addAndMakeVisible(tempoSyncButton);
```

In `resized`, add the following lines below where we position the `delayTimeKnob`:

```
delayTimeKnob.setTopLeftPosition(20, 20);
tempoSyncButton.setTopLeftPosition(20, delayTimeKnob.getBottom() + 10);
delayNoteKnob.setTopLeftPosition(20, tempoSyncButton.getBottom() - 5);
```

Try it out, there is now a **Sync** button in between the **Time** and **Note** knobs.

The **Note** knob lets you change the note length. This knob doesn't smoothly turn but jumps between the different options. That's because the parameter that is attached to this knob is an `AudioParameterChoice`.



The Sync button that toggles between Time and Note modes

You can type in values as well: Try typing in **1/8** or **1/2 dot** and the knob will jump to that position. We didn't have to do anything special to make that happen, thanks to the `AudioParameterChoice`. Pretty neat.

Sync is a toggle button, and pressing it should draw the button as red. That's what the line `tempoSyncButton.setColour(...)` in the constructor was for, because by default it's really hard to see the difference between the on and off states.

Later in this chapter we'll properly style the button by making a look-and-feel for it. The UI looks a little cramped but we'll fix this later too.

Getting the tempo information

The plug-in communicates with the host in various ways. You've seen that the parameters expose the plug-in's controls to the host, the `get` and `setStateInformation` functions save and load the plug-in's state, `isBusesLayoutSupported` negotiates the number of audio channels, and of course `processBlock` is used by the host to tell the plug-in to render the next chunk of audio.

The host can also provide additional information about the current tempo, the time signature, whether the audio is currently playing, where the playhead is located, and more. This is done through the `juce::AudioPlayHead` object.

From within `processBlock` we can write `getPlayHead()` to obtain this object. We can inspect the playhead's properties to find out more about the status of the host, in particular the current tempo.

We'll write a new class to handle this playhead information, so that you can easily use this in other projects as well.

In Projucer, select the **Source** folder and from the right-click menu choose **Add New CPP & Header File...**. For the filename use **Tempo**. Save the project and open it in your IDE.

Replace the contents of **Tempo.h** with the following:

```
#pragma once

#include <JuceHeader.h>

class Tempo
{
public:
    void reset() noexcept;

    void update(const juce::AudioPlayHead* playhead) noexcept;

    double getMillisecondsForNoteLength(int index) const noexcept;

    double getTempo() const noexcept
    {
        return bpm;
    }

private:
    double bpm = 120.0;
};
```

This defines a new class **Tempo** with a few different functions and variables:

- **reset** – Like all DSP objects it needs to have a function to reset the internal state when audio playback starts or restarts.
- **update** – This function will be called from within `processBlock` to inspect the current state of the host's audio transport using the `juce::AudioPlayHead` object. Here we'll read the current tempo.
- **getMillisecondsForNoteLength** – We will use this function to convert a given note length, such as 1/4 or 1/2 dotted, into a time in milliseconds.
- **getTempo** – This is a “getter” function that returns the value of the private member variable `bpm`. The `bpm` variable, which stands for Beats Per Minute, describes the tempo in number of quarter notes per minute.

We're not putting the `bpm` variable directly in the `public:` section of the class because this variable should only be changed by the `Tempo` object itself, never by any outside code.

This is called **information hiding** and it's considered to be a good programming practice. The internal operation of an object should not be visible to, or accessible by, other objects. We didn't do this in the `Parameters` class as it was more convenient there to directly expose the variables, but you could move all its member variables into the `private:` section and write getter functions for them.

The `getTempo` function is really simple and can therefore be defined directly in the `.h` file. This also allows the C++ compiler to inline the function, which makes it just as fast as a direct variable access.

`getTempo` is declared `const` because it doesn't change the state of the `Tempo` object. In other words, it does not write to any of the variables, only reads from them. The function is also declared `noexcept` because we are 100% sure it won't throw any exceptions, allowing the C++ compiler to make this function as efficient as possible.

The `reset` and `update` functions are marked `noexcept` as well, but not `const` as they *do* change the state of the `Tempo` object by writing to the `bpm` variable. There is no explicitly defined constructor for `Tempo`, since we don't need to do anything special upon creation of this object.

Let's implement these functions. Open `Tempo.cpp` and replace its contents with the following code:

```
#include "Tempo.h"

void Tempo::reset() noexcept
{
    bpm = 120.0;
}
```

The `reset` function gives the `bpm` variable a reasonable default value. This is necessary because not all DAWs may provide tempo information. In that case, tempo sync will assume a tempo of 120 BPM.

Add the `update` function next. It's fairly complicated, so I've added lots of explanations.

```

void Tempo::update(const juce::AudioPlayHead* playhead) noexcept
{
    // 1
    reset();

    // 2
    if (playhead == nullptr) { return; }

    // 3
    const auto opt = playhead->getPosition();

    // 4
    if (!opt.hasValue()) { return; }

    // 5
    const auto& pos = *opt;

    // 6
    if (pos.getBpm().hasValue()) {
        bpm = *pos.getBpm();
    }
}

```

Step-by-step this is what it does:

1. Just to make sure, call `reset` so that `bpm` is 120 if any of the following steps fails.
2. The argument to this function is not a `juce::AudioPlayHead` object but a *pointer* to such an object. Pointers can have the special value `nullptr` to indicate they point to nothing at all. This happens when a host does not provide `playhead` information, in which case we immediately return from this function.
3. If we get here, then `playhead` points to a valid `juce::AudioPlayHead` object. To get information about the audio transport we call `getPosition()`. Note the usage of `->` instead of `.` because `playhead` is a pointer.
4. You'd think that `getPosition()` would return an object with position information but it doesn't. Instead, it returns a `juce::Optional` that may or may not contain the required information. We need to check this with `opt.hasValue()`. The optional is used here because even if we get a valid `AudioPlayHead` object, it may still not have position info inside it, in which case we immediately return.
5. To get the `juce::AudioPlayHead::PositionInfo` object we need to **dereference** the optional using the `*` operator. To avoid writing out the full typename we say `const auto` and let the C++ compiler figure out what the actual class is. The `&` avoids making a copy of the `PositionInfo` object.

6. From the `PositionInfo` object we can get the tempo with `pos.getBpm()`. This also returns an optional. If it has a value, we can dereference the optional using `*`, which gives the tempo in beats-per-minute as a double.

Phew, it was quite a bit of work to get there with all those pointers and `juce::Optional` objects. They are necessary because some hosts may not provide all this information and we need to check what pieces of info are actually available to us.

In addition to the BPM there is information about the elapsed playing time, the time signature, whether the host is correctly recording, whether the audio is looping, and so on. In this plug-in we only use the tempo.

It's OK if this code is not immediately clear to you. That's why I put it into a class of its own, so that when you need to use the tempo information for some new project, you can simply grab these **Tempo.h** and **Tempo.cpp** files and use them, without having to go through all that again. It works, it's tested, and it becomes a reusable module in your DSP toolbox.

Note: C++ often uses the same symbol for different purposes. The `*` is used for multiplication, to indicate something is a pointer, and to dereference or read from a pointer or other object such as `juce::Optional`. It depends on the context which meaning of `*` is intended. We've used `*this` a few times before, for instance in the APVTS constructor. There it's used to turn `this`, which is a pointer, into a reference.

Converting note lengths to delay time

The last function to add to the `Tempo` class is `getMillisecondsForNoteLength`. Somehow we must translate a note length such as $1/4$ into a time in milliseconds.

We have the tempo in beats-per-minute in the `bpm` variable, and we know that one beat is a quarter note, so in $4/4$ time there will be four beats per bar.

If `bpm` is 120, there are 120 quarter notes per minute, or $120 / 60 = 2$ quarter notes per second. However, we want to know it the other way around...

At 120 BPM each quarter note lasts $60 / 120 = 0.5$ seconds. In milliseconds that is $60 \times 1000 / 120 = 500$ milliseconds.

Suppose the chosen note length is a half note or $1/2$. We know one half note is two quarter notes, so the delay here should be twice as long: $60000 \times 2 / 120 = 1000$ milliseconds, or 1 second per note. For a note length that is shorter, say $1/8$, the delay length would be $60000 \times 0.5 / 120 = 250$ ms. And so on...

What we can do is make a table that contains these multipliers. For every entry in the list of note lengths used by the `AudioParameterChoice`, we will write down how many quarter notes this is made up of.

Add the following code to **Tempo.cpp**. It doesn't really matter where, but I prefer to put it at the top of the file, right below the `#include` statement.

```
static std::array<double, 16> noteLengthMultipliers =
{
    0.125,           // 0 = 1/32
    0.5 / 3.0,       // 1 = 1/16 triplet
    0.1875,          // 2 = 1/32 dotted
    0.25,            // 3 = 1/16
    1.0 / 3.0,        // 4 = 1/8 triplet
    0.375,           // 5 = 1/16 dotted
    0.5,              // 6 = 1/8
    2.0 / 3.0,        // 7 = 1/4 triplet
    0.75,             // 8 = 1/8 dotted
    1.0,               // 9 = 1/4
    4.0 / 3.0,        // 10 = 1/2 triplet
    1.5,              // 11 = 1/4 dotted
    2.0,               // 12 = 1/2
    8.0 / 3.0,        // 13 = 1/1 triplet
    3.0,              // 14 = 1/2 dotted
    4.0,              // 15 = 1/1
};
```

The comments on the right are the corresponding note length for each entry in this array. The multiplier for $1/4$ is 1.0 since that is a quarter note. The multipliers for longer notes are larger than one, the multipliers for shorter notes are smaller than one.

For example, a $1/32$ note is 8 times shorter than a quarter note, and so its multiplier is $1/8$ or 0.125. Dotted notes are 1.5 times larger than their note length, so $1/4$ dotted has a multiplier of 1.5. Triplets are a little weird, perhaps. A $1/4$ note played in a triplet has the length of a $1/2$ note divided by 3.

The data type of this variable is `std::array<double, 16>`. The `std::array` means it's a list of elements. Each element is of type `double` and there are 16 of them in total. Since these numbers are doubles, there is no `f` behind the numbers.

Note: We're using `double` instead of `float` because the `juce::AudioPlayHead` object gives us the tempo and any other info as `double`. The main difference is that `double` values are more precise than `float` values.

Now that we have this lookup table, we can write `getMillisecondsForNoteLength`. Add this to the bottom of **Tempo.cpp**:

```
double Tempo::getMillisecondsForNoteLength(int index) const noexcept
{
    return 60000.0 * noteLengthMultipliers[size_t(index)] / bpm;
}
```

This uses the same formula we just worked out: 60000 for the number of milliseconds in one minute, times the multiplier for the note length, divided by the tempo in BPM.

The `index` argument is the selected note length from the `AudioParameterChoice` parameter. The `size_t(index)` cast is done to silence a compiler warning. To index a `std::array` you need to provide the index as an unsigned number, which means it's either zero or positive. However, our `index` is an `int`, which can be negative as well as positive. By casting to `size_t` we make the compiler happy.

That's all for the `Tempo` source files. Next, we need to create an instance of `Tempo` and hook it up to the `juce::AudioPlayHead`.

Calculating the delay length

In `PluginProcessor.h`, put an include below the others so it knows what `Tempo` is.

```
#include "Tempo.h"
```

Then somewhere in the class's `private:` section create a new instance:

```
Tempo tempo;
```

In `PluginProcessor.cpp`, in `prepareToPlay`, reset the `tempo` object to give it a default tempo of 120 BPM.

```
tempo.reset();
```

In `processBlock` we need to read the playhead information into the `tempo` object. Add the following line right after `params.update()`:

```
tempo.update(getPlayHead());
```

Note that this function should *only* be called from `processBlock` or any functions that are called by `processBlock`. The `getPlayHead()` function will not give the correct results otherwise.

Next, we need to use the `tempo` information to calculate the corresponding delay time. For this, add the following lines right below the previous one:

```
float syncedTime = float(tempo.getMillisecondsForNoteLength(params.delayNote));
if (syncedTime > Parameters::maxDelayTime) {
    syncedTime = Parameters::maxDelayTime;
}
```

This calls the `getMillisecondsForNoteLength` function with the index of the selected note length from the `AudioParameterChoice` object, which we placed in `params.delayNote`. This returns the delay time in milliseconds as a double, so we cast it to a `float` as all our calculations are done in `floats` too.

Just to make sure, we limit `syncedTime` to the maximum delay time of 5 seconds. It's possible that certain combinations of note length and BPM would result in a delay length that is longer than the max. For example, a whole note at 48 BPM is $60000 \times 4 / 48 = 5000$ milliseconds. A BPM lower than 48 could therefore exceed the maximum delay length. That would be a pretty slow song, but it's always good to write code defensively like this, just in case.

Now that we have the delay length in milliseconds in the `syncedTime` variable, we should change the code that tells the delay line how long the delay is. Currently this is done in the audio processing loop like so:

```
float delayInSamples = params.delayTime / 1000.0f * sampleRate;
delayLine.setDelay(delayInSamples);
```

Replace those lines with the following logic:

```
float delayTime = params.tempoSync ? syncedTime : params.delayTime;
float delayInSamples = delayTime / 1000.0f * sampleRate;
delayLine.setDelay(delayInSamples);
```

This uses the conditional `? :` syntax again, which if you'll recall is like a mini `if` statement. If `params.tempoSync` is true, we use the `syncedTime` variable, otherwise we use the value from `params.delayTime`. Then we convert milliseconds to samples and use that to set the delay as before.

The reason I used `? :` here is that the alternative way of writing it takes up a lot more room:

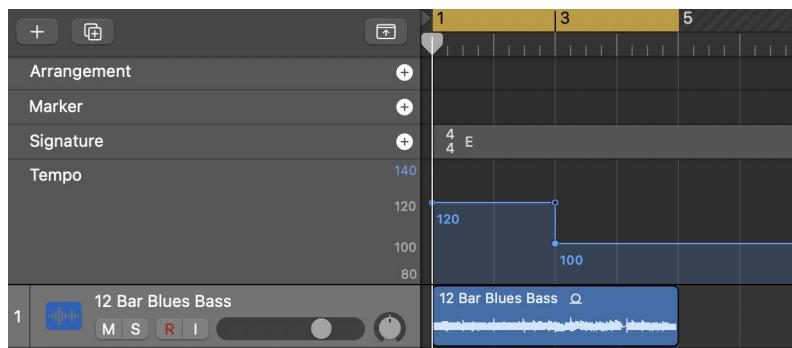
```
float delayTime;
if (params.tempoSync) {
    delayTime = syncedTime;
} else {
    delayTime = params.delayTime;
}
```

With the `? :` construct, we can do all of that in a single line.

All right, that should do it, give it a try! Toggle the **Sync** button and you'll now be able to set the delay time using the **Note** knob.

In `AudioPluginHost` there is no tempo setting, so the BPM will default to 120, but you can still test it: 1/4 note means a delay of half a second or 500 ms. Choosing a longer note value should make the delay longer and choosing a shorter note value should make it shorter.

Because we query the `juce::AudioPlayHead` at the start of every `processBlock`, the plug-in will be able to handle tempo automation just fine. For example, here I have automated the tempo in Logic Pro. As soon as the tempo changes, the delay length will change with it.



Tempo automation in Logic Pro

You may have noticed that turning the **Note** knob while sound is playing may give crackling sounds. This is zipper noise rearing its ugly head again. Because the `AudioParameterChoice` is discrete, changing its value will always jump from one setting to another.

This also happens when toggling **Sync** on and off, causing the delay time to suddenly switch to a value that may be very different.

These jumps between different delay times are immediate and therefore can create a small discontinuity in the sound, which sounds like a click or pop or crackle. For the same reason, glitches can happen with tempo automation in the DAW.

The way around this issue is to quickly crossfade between the previous delay length and the new one, similar how parameter smoothing quickly slides from the old value to the new one. We will investigate crossfading methods in a later chapter.

Styling the button

The tempo sync feature works — yay! — but the **Sync** button is rather unbecoming. Let's style this with our own look-and-feel. This is going to be similar to what you did previously for the `RotaryKnob`.

First, in `LookAndFeel.h`, add some new colors to the namespace:

```
namespace Colors
{
    // ...

    namespace Button
    {
        const juce::Colour text { 80, 80, 80 };
        const juce::Colour textToggled { 40, 40, 40 };
        const juce::Colour background { 245, 240, 235 };
        const juce::Colour backgroundToggled { 255, 250, 245 };
        const juce::Colour outline { 235, 230, 225 };
    }
}
```

At the bottom of the file, add a new ButtonLookAndFeel class:

```
class ButtonLookAndFeel : public juce::LookAndFeel_V4
{
public:
    ButtonLookAndFeel();

    static ButtonLookAndFeel* get()
    {
        static ButtonLookAndFeel instance;
        return &instance;
    }

    void drawButtonBackground(juce::Graphics& g, juce::Button& button,
                             const juce::Colour& backgroundColour,
                             bool shouldDrawButtonAsHighlighted,
                             bool shouldDrawButtonAsDown) override;

    void drawButtonText(juce::Graphics& g, juce::TextButton& button,
                        bool shouldDrawButtonAsHighlighted,
                        bool shouldDrawButtonAsDown) override;

private:
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(ButtonLookAndFeel)
};
```

This is similar to the RotaryKnobLookAndFeel class you added before, except this one overrides the functions specifically for drawing buttons.

In **LookAndFeel.cpp**, add the implementations of these functions. First, the constructor of the ButtonLookAndFeel class:

```
ButtonLookAndFeel::ButtonLookAndFeel()
{
    setColour(juce::TextButton::textColourOffId, Colors::Button::text);
    setColour(juce::TextButton::textColourOnId, Colors::Button::textToggled);
    setColour(juce::TextButton::buttonColourId, Colors::Button::background);
    setColour(juce::TextButton::buttonOnColourId,
              Colors::Button::backgroundToggled);
}
```

This assigns our colors to the color IDs used by JUCE internally. As before, this allows you to override these colors for specific buttons.

Next up is `drawButtonBackground`. This draws everything except the text:

```
void ButtonLookAndFeel::drawButtonBackground(
    juce::Graphics& g,
    juce::Button& button,
    const juce::Colour& backgroundColour,
    [[maybe_unused]] bool shouldDrawButtonAsHighlighted,
    bool shouldDrawButtonAsDown)
{
    auto bounds = button.getLocalBounds().toFloat();
    auto cornerSize = bounds.getHeight() * 0.25f;
    auto buttonRect = bounds.reduced(1.0f, 1.0f).withTrimmedBottom(1.0f);

    if (shouldDrawButtonAsDown) {
        buttonRect.translate(0.0f, 1.0f);
    }

    g.setColour(backgroundColour);
    g.fillRoundedRectangle(buttonRect, cornerSize);

    g.setColour(Colors::Button::outline);
    g.drawRoundedRectangle(buttonRect, cornerSize, 2.0f);
}
```

This creates a `buttonRect` rectangle that's the same as the button's bounds but inset by one pixel on all sides, so that there's room to draw a border around the button. It trims an additional pixel from the bottom.

When the button is clicked, `shouldDrawButtonAsDown` is true, and we shift the `buttonRect` one pixel downwards to give the impression that the button is literally being pressed down like a mechanical button would be. This is why we subtracted that one pixel from the bottom, so there is room for this.

Next, we fill the background with the `backgroundColor`. When the button is toggled, i.e. the **Tempo Sync** parameter is true, this corresponds to the color set in `juce::TextButton::buttonOnColourId`. When the button is not toggled, it's the color from `buttonColourId`. Finally, we draw a two-pixel wide outline around the button.

The other function to add is `drawButtonText`, which draws just the text.

```

void ButtonLookAndFeel::drawButtonText(
    juce::Graphics& g,
    juce::TextButton& button,
    [[maybe_unused]] bool shouldDrawButtonAsHighlighted,
    bool shouldDrawButtonAsDown)
{
    auto bounds = button.getLocalBounds().toFloat();
    auto buttonRect = bounds.reduced(1.0f, 1.0f).withTrimmedBottom(1.0f);

    if (shouldDrawButtonAsDown) {
        buttonRect.translate(0.0f, 1.0f);
    }

    if (button.getToggleState()) {
        g.setColour(button.findColour(juce::TextButton::textColourOnId));
    } else {
        g.setColour(button.findColour(juce::TextButton::textColourOffId));
    }

    g.setFont(FONTs::getFont());
    g.drawText(button.getText(), buttonRect, juce::Justification::centred);
}

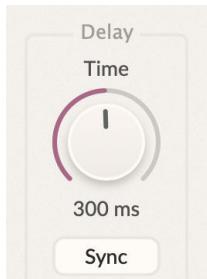
```

This does the same trick with shifting the `buttonRect` down by one pixel when the button is pressed down. Then it draws the button's text centered horizontally and vertically inside this rectangle. The color that is used depends on the button's toggle state.

In **PluginEditor.cpp**, in the constructor, replace the line that sets the red color on the button by the following, to use this new look-and-feel:

```
tempoSyncButton.setLookAndFeel(ButtonLookAndFeel::get());
```

Build and run to see what it looks like. Here, the button is shown in the “on” state.



The button with custom look-and-feel

It's a very basic button since I didn't want to fill this book with pages and pages of JUCE drawing code. If you feel like it, try to make this button look a bit fancier, perhaps by drawing the background as a subtle gradient like we did on the knobs.

Hiding the Time knob

It's a little strange to have both the **Delay Time** and **Note** knobs visible at the same time. When **Sync** is off, the **Note** knob isn't used. Conversely, when **Sync** is on the **Time** knob isn't used. It would be more appropriate to only show the knob that currently has an effect. This will also save some space and clean up the UI.

In **PluginEditor.cpp**, in the constructor, change the line that adds the `delayNoteKnob` to the `delayGroup` to the following:

```
delayGroup.addChildComponent(delayNoteKnob);
```

Previously this did `addAndMakeVisible`, now it does `addChildComponent`. This still adds the knob as a child of the delay `GroupComponent` but does not initially make it visible.

In `resized`, change the positioning code for the `delayNoteKnob` to place the **Note** knob directly on top of the **Time** knob:

```
delayNoteKnob.setTopLeftPosition(delayTimeKnob.getX(), delayTimeKnob.getY());
```

The improved UI looks like this, much cleaner!

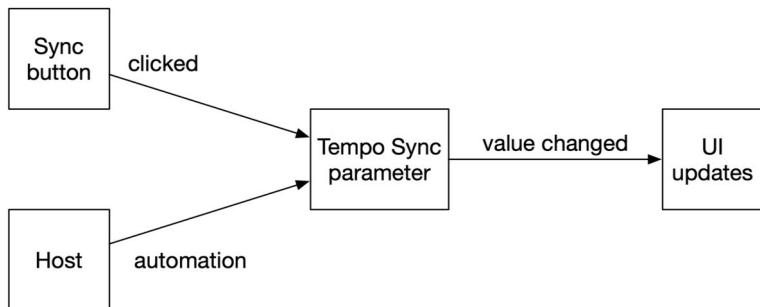


The Note knob is initially hidden

We will now write some code that listens to any changes in the **Tempo Sync** parameter and in response toggles the visibility of these two knobs — if **Tempo Sync** is on, we will hide the **Time** knob and show the **Note** knob. Both knobs will always exist and are both children of the **Delay** group, but only one will be visible at a time, depending on the state of the **Sync** button.

You might be tempted to write some code to intercept clicks on the **Sync** button. Click once, show the **Note knob**. Click again, show the **Time** knob. However, this won't work if the host changes the parameter through automation.

In plug-ins the UI is never the “source of truth”. We always have to go through the parameters. Pressing the **Sync** button will update the **Tempo Sync** parameter, and updates to the **Tempo Sync** parameter will decide which knob is visible. And also redraw the **Sync** button if that wasn't what triggered the update.



Everything revolves around the parameters

This means the editor will need to listen to changes on the **Tempo Sync** parameter somehow, to determine which knob to make visible.

We can use a `juce::AudioProcessorParameter::Listener` object for this. It's easiest if we turn the `DelayAudioProcessorEditor` object into such a listener.

In `PluginEditor.h` the class is currently declared as:

```
class DelayAudioProcessorEditor : public juce::AudioProcessorEditor
```

Change this to the following:

```
class DelayAudioProcessorEditor : public juce::AudioProcessorEditor,
                                 private juce::AudioProcessorParameter::Listener
```

Now the editor class inherits from two objects: the base JUCE editor as well as the parameter listener class. The `private` keyword means all the functions and variables of the listener get added to the private section of our editor class.

There are two functions we must implement to become a parameter listener. Put these two functions in the `private:` section in **PluginEditor.h**:

```
void parameterValueChanged(int, float) override;
void parameterGestureChanged(int, bool) override { }
```

Personally, I prefer putting functions above variables, so I put these two lines immediately after the `private:` keyword.

We won't be using `parameterGestureChanged` but must provide it anyway, so it has an empty implementation as given by the `{ }` behind the function. Notice that no semicolon ; is needed here.

Put the implementation of the other function in **PluginEditor.cpp**:

```
void DelayAudioProcessorEditor::parameterValueChanged(int, float value)
{
    DBG("parameter changed: " << value);
}
```

For now, this function just prints a message with the parameter's new value. The first argument is an integer containing the index of the parameter in the `AudioProcessor`'s list of parameters. We don't care about this index as we'll only be listening to one parameter, **Tempo Sync**.

Making the editor listen to the parameter happens in the constructor. Add this line at the bottom of the editor's constructor:

```
audioProcessor.params.tempoSyncParam->addListener(this);
```

This grabs the pointer to the **Tempo Sync** parameter and calls `addListener(this)` on it. The keyword `this` refers to the current object, which here is the editor.

From now on, the `AudioParameterBool` object that holds the **Tempo Sync** parameter will notify the editor when the parameter's value changes, by calling `parameterValueChanged` with the new value.

It's a good idea to remove the listener when the editor is destructed, otherwise the parameter might try to notify an object that no longer exists. Change the destructor to the following:

```
DelayAudioProcessorEditor::~DelayAudioProcessorEditor()
{
    audioProcessor.params.tempoSyncParam->removeListener(this);
    setLookAndFeel(nullptr);
}
```

One small problem: this won't compile... The `params` object is a private member of the `DelayAudioProcessor` and `tempoSyncParam` is a private member of `Parameters`.

To fix this, go to `PluginProcessor.h` and move the line `Parameters params;` into the `public:` section, but keep it below the line that creates the APVTS. The order in which these objects get initialized matters, and the APVTS must come first.

Next, go to `Parameters.h` and move the line `juce::AudioParameterBool* tempoSyncParam;` into the `public:` section. Everything should compile again.

Try it out. Run the plug-in and keep an eye on the debug output in the IDE. Whenever you click the **Sync** button, it will write `parameter changed: 0` (false, button is off) or `1` (true, button is on). So that works! Now all we need to do is show the corresponding knob while hiding the other one.

However, there is a little snag... The `parameterValueChanged` callback function will be called on the UI thread when you click the button, but it may also be called on some other thread — such as the audio thread — when automation is used. A host can decide to call this on any thread it wants, so inside `parameterValueChanged` we cannot assume that we're on the UI thread. That's a problem because doing something like `delayNoteKnob.setVisible(true)` is only allowed on the UI thread.

Quick recap: The audio processing is performed by a high priority thread called the audio thread. This runs independently from, and at the same time as, any other things your program does such as handling the user interface. The editor has its own thread, known as the UI thread. We must be careful about calling functions on the correct thread or unexpected things — and crashes! — may happen. Unfortunately, you can't simply look at the code to see which thread does what, and it's very easy to write code that ends up running on the wrong thread. Even experienced developers mess up this threads stuff!

One solution is to use `juce::MessageManager::callAsync` to schedule a piece of code to be run on the UI thread, or as JUCE calls it, the message thread. Let's put the code that we want to run into a new function, for clarity.

In **PluginEditor.h** add the following function declaration below the other private functions:

```
void updateDelayKnobs(bool tempoSyncActive);
```

In **PluginEditor.cpp**, implement this function as follows:

```
void DelayAudioProcessorEditor::updateDelayKnobs(bool tempoSyncActive)
{
    delayTimeKnob.setVisible(!tempoSyncActive);
    delayNoteKnob.setVisible(tempoSyncActive);
}
```

If the **Tempo Sync** parameter is on, `tempoSyncActive` is true. This hides the **Time** knob and shows the **Note** knob. If `tempoSyncActive` is false, it's the other way around. The `!` symbol is the **not** operator and inverts the meaning of the `bool` variable.

Now we can call this asynchronously from `parameterValueChanged`. Change that function as follows:

```
void DelayAudioProcessorEditor::parameterValueChanged(int, float value)
{
    if (juce::MessageManager::getInstance()->isThisTheMessageThread()) {
        updateDelayKnobs(value != 0.0f);
    } else {
        juce::MessageManager::callAsync([this, value]
        {
            updateDelayKnobs(value != 0.0f);
        });
    }
}
```

If `parameterValueChanged` is running on the UI thread, a.k.a. the message thread, we can directly call the `updateDelayKnobs` function. The argument `tempoSyncActive` is true if the new parameter value is not equal to `0.0f`, and false otherwise.

If we're not on the UI thread, this performs `callAsync` to do the same. The term **async** or **asynchronous** means that this code is passed off to the UI thread to be performed at some later point, but we won't wait for that to happen.

We've told the UI thread to call `updateDelayKnobs` and it will do so when it gets around to it, and in the meantime our thread continues working on the next thing.

Notice that the argument to `callAsync` is written like this:

```
[this, value]  
{  
    updateDelayKnobs(value != 0.0f);  
}
```

This is a so-called **lambda**, sometimes also called a **closure**. It defines a small function that has no name. Essentially we're passing a function (the lambda) as the argument to another function.

The `[this, value]` notation means that the values of `this` and the variable named `value` will be captured in the lambda. These values will be stored alongside this anonymous function, so that whenever the UI thread gets around to running it, it will use the correct data. Lambdas are an advanced C++ feature but very handy.

Try it out, clicking the **Sync** button will now switch between the two knobs. This will work even when you're changing this parameter using automation in your DAW. It's kind of silly to want to automate the **Tempo Sync** parameter, but we still should test whether it works.

By the way, even when a knob is hidden, it keeps getting parameter updates when the parameter is changed by the host. In `AudioPluginHost`, right-click the **Delay** block and choose **Show all parameters** to bring up the generic editor. Use this to change the **Delay Time** parameter when the **Time** knob is hidden in the actual editor. Turn off **Tempo Sync** and notice that **Time** has taken on the new value as expected.

Note: Using `juce::MessageManager::callAsync()` from the audio thread is not 100% kosher as it may block and we should never do anything from the audio thread that could potentially block it. However, sometimes you just need to break the rules. It's safe to assume that most of the time the callback *will* run on the UI thread since the user is unlikely to automate the **Tempo Sync** parameter. This is a parameter the user will typically change once, likely before audio is even playing. Since the chance is so small that this will cause a problem, I think we can get away with it here. The recommended method is to send a message using a lock-free FIFO that's periodically checked by the UI thread. (More about FIFOs in the next chapter.)

There is one situation we haven't handled yet and that is the initial state of the editor when it first opens. Do this:

- Start the plug-in in a session.
- Click **Tempo Sync** and set the **Note** to 1/8.
- Save the session and quit.
- Reopen the session.

What's going on here? The **Sync** button is correctly toggled on, but instead of the **Note** knob it is the **Time** knob that is visible. You have to click **Sync** once to turn it off, and again to re-enable it, and now the **Note** knob with a value of 1/8 will appear. Something's clearly broken here.

This happens because the editor's constructor always assumes the **Time** knob will be visible, since it does the following:

```
delayGroup.addAndMakeVisible(delayTimeKnob);
delayGroup.addChildComponent(delayNoteKnob);
```

As long as we don't get a notification in `parameterValueChanged`, the UI is never updated to reflect the true state. The fix is simple enough. Just before the line that adds the listener, write this:

```
updateDelayKnobs(audioProcessor.params.tempoSyncParam->get());
```

This will read the current setting of the **Tempo Sync** parameter and update the visibility of the knobs based on that. Try it out. Do the same test again and the **Note** knob should be visible right away.

Congrats! The Delay plug-in is pretty much feature complete now. But that doesn't mean we're done! In the next chapter we're going to dive deeper into DSP and write our own delay line class.

16: Diving deeper into DSP

The plug-in is almost complete but before we put the finishing touches to it, it will be instructive to look under the hood of one of the DSP building blocks we've been using: the delay line. The delay line is built on the concept of a **circular buffer**, which is something you'll find in many places in DSP and audio programming, so it's good to learn more about it.

If you've been able to follow along with the book so far, I'm sure you'll be able to understand how to write our own version of `juce::dsp::DelayLine` from scratch. It's not as scary as it may seem! Get ready to join the high priesthood of DSP developers...

The DelayLine class

We will write our own `DelayLine` class that is very similar to `juce::dsp::DelayLine` but there will also be some differences.

The public functions that our class `DelayLine` will have are:

- `setMaximumDelayInSamples` – Allocates the memory used to store the contents of the delay line. This should be called prior to any processing, from `prepareToPlay`.
- `reset` – Clears the delay line and resets all state. This should be called before first usage.
- `getBufferLength` – Returns the maximum delay length in samples.
- `write` – Places a new sample into the delay line, overwriting the previous oldest element. Does the same as JUCE's `pushSample`.
- `read` – Reads a sample from the delay line, similar to JUCE's `popSample`.

As you can see, the class behaves much the same as the JUCE delay line, even though some of the function names are different. The most important difference is that

there's no `setDelay` function. Rather, you specify the delay length when you call `read`. There is no `prepare` function as our `DelayLine` won't need it.

The heart of the `DelayLine` is a circular buffer. Because it is circular, writing a new sample into the delay line will always overwrite the oldest element in the buffer. But why a circular buffer? Let's look at some computer science theory first.

Why a circular buffer?

For the purposes of illustration suppose we have a delay line that is 5 samples long. We can implement this by allocating space in the computer's memory for 5 `float` values. This is usually called a **memory buffer** or an **array**.

Initially we fill this memory with zeros, like so:

0	1	2	3	4
0.0	0.0	0.0	0.0	0.0

An empty memory buffer

The numbers at the top are the indices that we can use to access the individual elements from this memory buffer. Recall that we always start counting at 0.

On each timestep, we will take the next input sample and write it into this memory buffer at index 0, while shifting the existing contents of the buffer to the right. Here's what happens for the first four timesteps:

	0	1	2	3	4
timestep 0	0.831	0.0	0.0	0.0	0.0
timestep 1	0.507	0.831	0.0	0.0	0.0
timestep 2	-0.142	0.507	0.831	0.0	0.0
timestep 3	-0.776	-0.142	0.507	0.831	0.0

Inserting new samples into the memory buffer

To read a delayed sample, we can simply read anywhere from within this buffer. The most recently inserted sample value is at index 0 in the buffer. The higher the index that we read at, the older the sample is.

After one more timestep the buffer is full:

timestep 4	0.253	-0.776	-0.142	0.507	0.831
------------	-------	--------	--------	-------	-------

All positions in the memory buffer have samples

If we keep going, the oldest sample values will shift out of the buffer and we no longer have access to them. Since this delay line has room for five samples, it only gives access to the five most recent sample values. Older samples will disappear forever.

In practice our delay lines will be much larger than 5 samples, but to understand how these things work I find it useful to draw out a small version on paper and manually go through a few steps, just like we did here. It's easier for me to think about 5 things than about 240-thousand things.

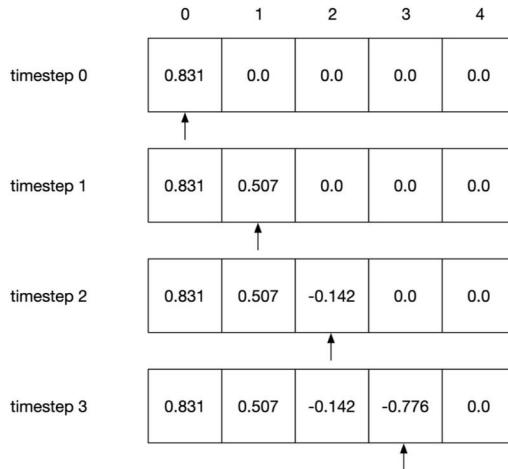
The above approach seems like it would work but it has one massive disadvantage: on every new timestep we have to shift the contents of the memory buffer to the right. This will be slow, especially if there are thousands of samples in the delay line.

Remember that a 5-second maximum delay at a sample rate of 48 kHz means there will be $5 \times 48000 = 240000$ float values in the buffer. Shifting these will be very slow, especially since we need to do this for every new input sample.

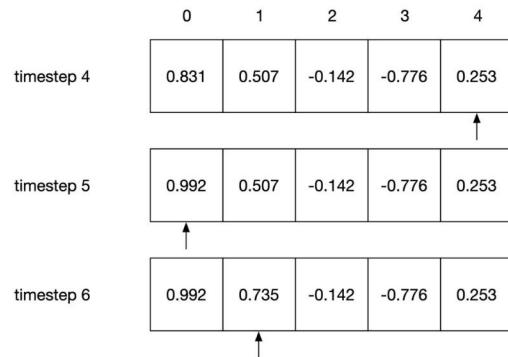
The solution is to convert this into a circular buffer, also known as a **ring buffer**, or a **FIFO queue** where FIFO stands for First In, First Out.

The circular buffer uses the same memory layout as before but it will write the samples into it in a different way. Instead of always writing into the first position and shifting the remaining samples, we'll first write into index 0, then into index 1, then into index 2, and so on.

The circular buffer keeps track of the place to write in a new variable `writeIndex`. This points to the place in memory where the most recent sample was written. In the illustration below, the arrow represents this write index.

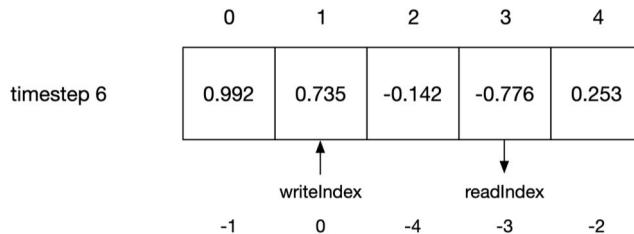
**Inserting samples into the circular buffer**

At each timestep, `writeIndex` is incremented and the incoming sample is written at that new index. When the write index gets to the end of the memory region, it will **wrap around** to the beginning and overwrites the sample at index 0 again:

**The circular buffer wraps around when it reaches the end**

Note that we always overwrite the oldest sample in the delay line. The sample at `writeIndex` is the newest, and the one to the right of `writeIndex` is the oldest.

To read from the delay line, we first calculate a `readIndex`, which is `writeIndex - delayInSamples`. Let's say the `delayInSamples` is 3. In the example, the `writeIndex` is at position 1. Then $\text{readIndex} = 1 - 3 = -2$. We cannot read from a negative index, so we'll have to wrap around the read position and make it read two indices from the right.



Reading with wrap-around

It's useful to count this out by hand to verify the numbers makes sense: Reading with delay 0 would mean reading at the `writeIndex` itself, which is the sample that was most recently written. A delay of 1 means reading at position `writeIndex - 1`, which is index 0. A delay of 2 means we need to wrap around to index 4, which is the last position in the buffer. A delay of 3 samples then lands at index 3. Looks correct to me!

The advantage of this circular buffer is that we never have to shift the contents of the memory area. We only write a single sample value to memory, which is very efficient. We just need to keep track of where we wrote the last sample, and possibly wrap around when writing and when reading.

One thing to consider is that with a 5-element circular buffer, the largest possible `delayInSamples` is 4, one less than the length of the memory area. Suppose `writeIndex` is 1, then a delay of 4 samples will read from index 2, which is the oldest sample in the buffer. However, a delay of 5 samples would attempt to read from index 1, which is actually the newest sample that we just wrote — if we go too far back in the past, we end up going in circles.

Note: In audio programming, FIFO queues are often used to communicate between different threads, such as the audio thread and the UI thread, for example to create an audio visualizer. A FIFO uses a circular buffer under the hood, but the write index and read index may move independently from each other. There will be one process writing into the FIFO and another process reading from the FIFO, and these processes may be working at different speeds. In that case it's important that the read index will never overtake the write index — the process that is reading must never try to read more than is written. In this book we won't do anything nearly as complicated, but since the term FIFO comes up in audio circles, now you know it's really just a fancy circular buffer.

Implementing the circular buffer

In Projucer, add new source files to the project named **DelayLine.h** and **DelayLine.cpp**.

In your IDE, replace the contents of **DelayLine.h** with the following:

```
#pragma once

#include <memory>

class DelayLine
{
public:
    void setMaximumDelayInSamples(int maxLengthInSamples);
    void reset() noexcept;

    void write(float input) noexcept;
    float read(float delayInSamples) const noexcept;

    int getBufferLength() const noexcept
    {
        return bufferLength;
    }

private:
    std::unique_ptr<float[]> buffer;
    int bufferLength = 0;
    int writeIndex = 0; // where the most recent value was written
};
```

It's perfectly OK for us to create a class named **DelayLine** even though the JUCE class is also named **DelayLine**, since the JUCE one lives in the namespace `juce::dsp`, and so their names won't conflict.

In the `public:` section are the functions described earlier. In the `private:` section are the internal variables that will be used by the delay line.

The `buffer` variable holds the memory region that will store the delayed samples. This is a `std::unique_ptr` for a `float[]` object.

`float[]` means a memory region or array that will hold multiple `float` values. We will allocate the memory for this array in `setMaximumDelayInSamples`, which returns a pointer to the first element in that memory region.

We keep track of this memory region using the `std::unique_ptr` object. This is a

so-called **smart pointer** (“ptr” is short for pointer). It will automatically manage the lifetime of the allocated memory, so that we don’t have to worry about de-allocating that memory later. In other words, it prevents memory leaks.

Initially the buffer smart pointer is not pointing to anything, its value is `nullptr`.

The `#include <memory>` line at the top of the file is necessary to tell the C++ compiler about the `std::unique_ptr` type.

Let’s implement the functions in **DelayLine.cpp**. First, `setMaximumDelayInSamples`:

```
#include <JuceHeader.h>
#include "DelayLine.h"

void DelayLine::setMaximumDelayInSamples(int maxLengthInSamples)
{
    // 1
    jassert(maxLengthInSamples > 0);

    // 2
    int paddedLength = maxLengthInSamples + 1;

    // 3
    if (bufferLength < paddedLength) {
        bufferLength = paddedLength;

        // 4
        buffer.reset(new float[size_t(bufferLength)]);
    }
}
```

The job of this function is to reserve enough memory to hold the requested number of samples, given by `maxLengthInSamples`.

Step-by-step this is what it does:

1. The `jassert` verifies that the requested maximum delay length is more than zero samples. I’m sure you wouldn’t write `setMaximumDelayInSamples(-100)` but since the maximum delay length is calculated using a formula involving the sample rate and other things, it’s conceivable a mistake somewhere else could result in `maxLengthInSamples` being 0 or less.

If that happens, the `jassert()` will save our butts and notify you that something is probably wrong somewhere. Sprinkling these kinds of assertion statements throughout your code is a good way to find bugs that otherwise would have gone unnoticed. The `<JuceHeader.h>` include is for the `jassert` function.

2. The amount of memory to allocate is actually `maxLengthInSamples` plus one extra sample. Remember I showed you that, if the length of the memory buffer was 5 samples, the maximum delay was 4 samples? To make sure there is enough memory to hold `maxLengthInSamples`, we must make room for one more sample. It's OK if the memory buffer is larger than what we need, it's only a problem if it's too small.
3. We want to avoid allocating memory if the existing buffer is the same length as before, or is already larger than what we need. `setMaximumDelayInSamples` will be called from `prepareToPlay`. The host may invoke `prepareToPlay` many times over the lifetime of the plug-in, but the maximum delay length only changes if the sampling rate changes. There is no reason to throw away the existing memory and allocate a new buffer, if we already have a perfectly good piece of memory to work with.
4. Finally, we allocate the new memory by writing `new float[...]`. This creates a new chunk of memory for the purpose of holding many float values. `new` returns a raw pointer, which we store in the `std::unique_ptr` using `buffer.reset(...)`. If `buffer` was already pointing to existing memory because this isn't the first time that `setMaximumDelayInSamples` is called, the `std::unique_ptr` will properly deallocate that memory and return it to the pool of free memory.

Note: In C++ there are several ways to allocate memory for storing a sequence of values, for example by using a `std::array` or `std::vector` or even a `juce::AudioBuffer`. I don't like using objects like `std::vector` in audio code since they may do things behind the scenes that are not real-time safe — such as resizing the underlying memory region — and we have no control over that. I'd rather allocate a raw chunk of memory using `new float[...]` and manage it using a `std::unique_ptr`. Now we're in complete control of the memory.

Next up is the `reset()` function:

```
void DelayLine::reset() noexcept
{
    writeIndex = bufferLength - 1;

    for (size_t i = 0; i < size_t(bufferLength); ++i) {
        buffer[i] = 0.0f;
    }
}
```

The goal of `reset` is to return the variables to good initial values and to clear out any old data from the delay line.

We set `writeIndex` to be the very last element of the memory buffer, at index `bufferLength - 1`. Then the first time `write()` is called, the index is advanced by one step, wraps around, and the first sample we write will be at index 0. It doesn't matter at all where in the buffer we start writing, but I like things to be neat.

To clear out the old contents of the delay line, we set the memory buffer to `0.0f` everywhere using a `for` loop. The loop starts at index `i = 0` and keeps incrementing this variable using `++i` until `i < bufferLength` is no longer true. In other words, `i` will go from 0 to `bufferLength - 1`.

Recall that in C++ and most other computer languages we start counting at 0, so `i = 0` means the first element of the memory buffer and `i = bufferLength - 1` means the last element. For every element we do `buffer[i] = 0.0f` to set the `i`-th sample value back to silence.

The next function to add is `write`, which puts a new sample into the circular buffer:

```
void DelayLine::write(float input) noexcept
{
    // 1
    jassert(bufferLength > 0);

    // 2
    writeIndex += 1;

    // 3
    if (writeIndex >= bufferLength) {
        writeIndex = 0;
    }

    // 4
    buffer[size_t(writeIndex)] = input;
}
```

Writing the new sample value, given by the argument `input`, is pretty easy. Step-by-step this is what the function does:

1. Just because I'm paranoid this starts with another assertion, which verifies that the length of the buffer is greater than zero, i.e. that memory has actually been allocated. The only time the condition `bufferLength > 0` would be false is when we forgot to call `setMaximumDelayInSamples` first. Seems dumb, but been there, done that.

2. Increment the `writeIndex` by doing `writeIndex += 1`; Another way to write this in C++ is `writeIndex++` or `++writeIndex`. You've seen this `++` operator used in the `for` loops. Outside of `for` loops I prefer to do `+= 1` to increment a variable but `++` would work too.
3. If the `writeIndex` is going beyond the very last element in the memory buffer, it needs to wrap around. We check for this by doing `if (writeIndex >= bufferLength)`, and if so, set `writeIndex` back to zero.

It would have been sufficient to do `if (writeIndex == bufferLength)`. The reason I use greater-than-or-equals `>=` rather than just equals `==` is again a type of defensive programming.

4. Lastly, we put the new sample into the memory buffer using `buffer[writeIndex] = input;` We cast the `writeIndex`, which is an `int`, to a `size_t` to avoid an unsigned-vs-signed compiler warning. To be honest, this particular compiler warning is a bit annoying and the code would work fine without the `size_t(...)` cast, but it's part of the JUCE recommended warnings.

Note: Assertions are also called **pre-conditions**. We can only perform the `write` function on the condition that memory was allocated first. If you're worried that these `jassert` checks will slow down the code, rest assured that they are exclusively enabled in debug builds. A debug build has loads of extra stuff in it, including these assertions, that will save us time while developing and debugging. Once you're ready to release the plug-in to end users, you will make a release build where the assertions are disabled. This is faster but we better be sure we've thoroughly tested the plug-in since the safety checks are gone now too.

Finally, there is the function for reading from the circular buffer. We'll come back to this function a few more times in the remainder of this chapter to improve it, but to get started we'll use a basic implementation.

```
float DelayLine::read(float delayInSamples) const noexcept
{
    // 1
    jassert(delayInSamples >= 0.0f);
    jassert(delayInSamples <= bufferLength - 1.0f);

    // 2
    int readIndex = int(std::round(writeIndex - delayInSamples));

    // 3
    if (readIndex < 0) {
        readIndex += bufferLength;
    }

    // 4
    return buffer[size_t(readIndex)];
}
```

From top to bottom, here's what it does:

1. As is getting to be our custom, first there are some assertions to catch potential mistakes. We want to make sure the requested `delayInSamples` is at least zero. It can't be less than zero since that would be looking into the future and our plug-in isn't *that* advanced. The other assertion checks that the requested delay time isn't too long either.
2. Calculate the read index. This is done by subtracting the requested delay length from the `writeIndex`. Notice that `delayInSamples` is a `float` but we need to index the array using an integer, so we'll round off the read position to the nearest whole number using `std::round`.
3. Depending on where the `writeIndex` currently points and the length of the requested delay, the `readIndex` may end up being a negative value. If that happens we need to wrap it around to the other end of the buffer. This is done by adding the `bufferLength`.

Is that correct? Let's work through an example again. Say the `writeIndex` is 1 and the delay is three samples, then `readIndex` is $1 - 3 = -2$. Now we add `bufferLength`, which was 5, to get the wrapped read position of $-2 + 5 = 3$. In the illustrations earlier in the chapter you can see that this is indeed the correct read index.

4. Finally, read an audio sample from the buffer at the read index and return the value. As before, we need to cast `readIndex` to a `size_t` or the C++ compiler will complain.

The `read` function is marked `const` because it doesn't change any of the `DelayLine` state. It only reads from the member variables, never writes to them. This lets the C++ compiler produce more optimal machine code, but it also helps other developers reading the code understand which functions will change the internal variables of this object and which ones don't.

Note: Some implementations of a circular buffer, such as `juce::dsp::DelayLine`, keep a separate variable for the `readIndex` that is updated on every timestep. However, we calculate the read index on-the-fly, which is simpler and more flexible. You may wonder why we didn't make `readIndex` a `size_t` variable to begin with instead of an `int`. Keep in mind that `readIndex` can become negative, in which case it needs to wrap around. This will not happen for `size_t` because that cannot be negative, and so we'd never realize we'd need to wrap around.

That's it for our `DelayLine` class for now. All the other functions are complete but `read()` will need more work as it does not handle fractional delay lengths very well. By doing `std::round` to snap the read index to the nearest sample value, we are introducing noise and distortion into the sound. You may not be able to consciously hear this but it's definitely there (and I'll prove it to you shortly). Later in this chapter we'll investigate methods of interpolating between sample values, which get rid of the distortions.

Replacing the JUCE DelayLine

Let's replace the `juce::dsp::DelayLine` with our new class. Go to **PluginProcessor.h** and add an include for the new header file that we just made:

```
#include "DelayLine.h"
```

Inside the `private:` section of the class, remove this line:

```
juce::dsp::DelayLine<float, juce::dsp::DelayLineInterpolationTypes::Linear> delayLine;
```

and replace it with:

```
DelayLine delayLineL, delayLineR;
```

This creates two instances of our new `DelayLine` class, one for the left channel and one for the right channel. When using JUCE's delay line we only needed a single instance because `juce::dsp::DelayLine` supports having multiple channels. Our `DelayLine` works on a single channel at a time, and so we need one instance for the left and one for the right.

In `PluginProcessor.cpp`, in `prepareToPlay`, remove the following line:

```
delayLine.prepare(spec);
```

Also change these lines,

```
delayLine.setMaximumDelayInSamples(maxDelayInSamples);
delayLine.reset();
```

into the following. Since there are two instances, we must do each thing twice:

```
delayLineL.setMaximumDelayInSamples(maxDelayInSamples);
delayLineR.setMaximumDelayInSamples(maxDelayInSamples);
delayLineL.reset();
delayLineR.reset();
```

The biggest changes are in `processBlock`. Remove the following line as our `DelayLine` object does not have a `setDelay` function:

```
delayLine.setDelay(delayInSamples);
```

Change the following two lines,

```
delayLine.pushSample(0, mono*params.panL + feedbackR);
delayLine.pushSample(1, mono*params.panR + feedbackL);
```

into:

```
delayLineL.write(mono*params.panL + feedbackR);
delayLineR.write(mono*params.panR + feedbackL);
```

Instead of the function `pushSample` we use `write`. Where `pushSample` needed to be told which channel to use (0 or 1), in our case we use a different `DelayLine` object for each channel.

Change the next two lines,

```
float wetL = delayLine.popSample(0);
float wetR = delayLine.popSample(1);
```

into this:

```
float wetL = delayLineL.read(delayInSamples);
float wetR = delayLineR.read(delayInSamples);
```

The `popSample` function used the delay length from `setDelay`. However, we pass the delay length directly into our `read` function.

All right, that should do it. Try it out and verify that everything works as it should. Make sure to test with the maximum delay time of 5000 milliseconds, to verify this doesn't trip any of the assertions.

Congrats! You've written your first DSP building block. It wasn't so bad, was it? Granted, a delay line is one of the easier building blocks, but now you're no longer just using the stuff that comes in the box with JUCE — you're a proper DSP programmer!

That said, we can make this delay line better still. It's time to put on your propeller hat as this involves doing some math.

Interpolation

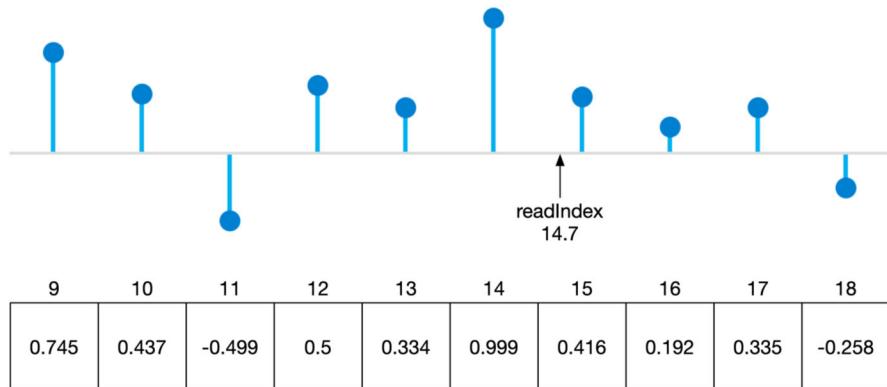
The specified delay length will usually correspond to a fractional number of samples. This means we may need to read in between two sample values somehow.

Currently `readIndex` is an integer but what if we made it a floating-point number? Perhaps like so:

```
float readIndex = writeIndex - delayInSamples;
```

Suppose `writeIndex` is 27 (this is always a whole number) and `delayInSamples` is 12.3. Then `readIndex` is $27 - 12.3 = 14.7$.

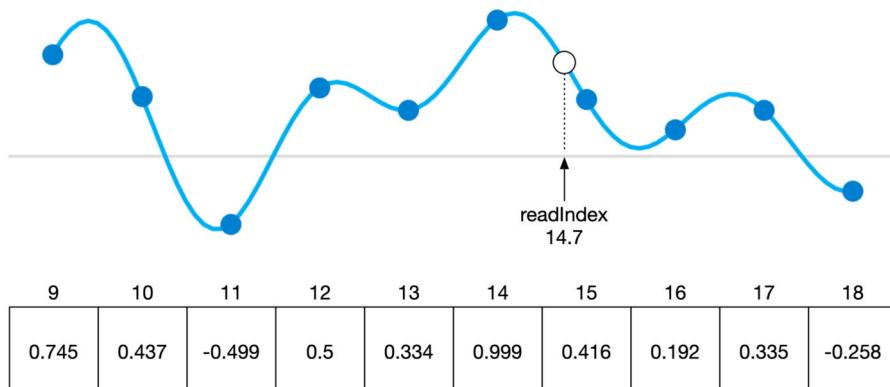
Let's use the following signal as an example:



This shows a portion of the delay line, from index 9 to index 18. The `readIndex` is at sample 14.7. How can we read from this fractional index?

The interesting thing about digital signals is that we can calculate exactly what the waveform would look like in between these samples. Even though only periodic readings were taken of the original continuous analog signal when it was converted into a digital signal, there is precisely one curve that goes through all these sample points.

This assumes the digital signal is bandlimited, meaning that it contains no frequencies that go over the Nyquist limit of half the sample rate. That will be true for the audio we're working with here.



The true signal going through the sample points

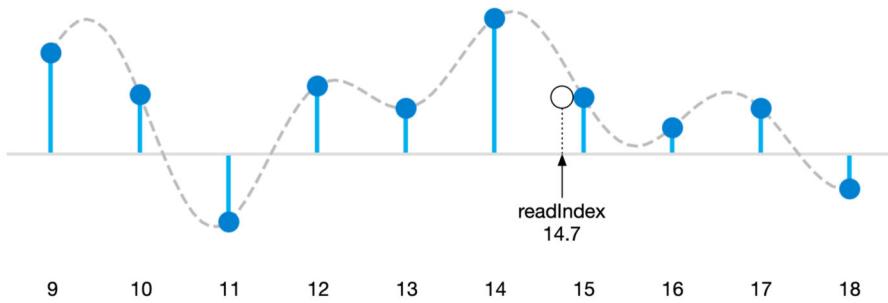
In theory we can therefore know what the sample value at index 14.7 is. In order to calculate the signal that goes in between the samples, we must convolve the digital signal with the sinc function. Convolution is an advanced DSP topic so I won't go into it here except to say that it involves blending two signals in a certain way.

The problem is that the sinc function is infinitely long, and so it will take infinite amount of time to perform this operation. The only reason I was able to make the above plot is that I have a time machine. Actually that's not true, I cheated a little.

Instead of an infinitely long sinc function, I used a windowed sinc — a sinc function that is chopped off and flattened at the ends — to make the computation tractable.

So even though the signal in that image looks convincing, it's not factually 100% correct. But we'll take that as the true signal anyway, since this is the closest we can get. Unfortunately, convolution with a windowed sinc is still kind of slow and not something we want to do in real-time.

Right now, `read()` rounds off the `readIndex` using `std::round`, so 14.7 becomes 15, and we'll read the sample value at index 15. But notice that this isn't really close to what the true value should have been (the gray dashed line).



The read index is rounded to the nearest sample

Since we do this rounding every time `read()` is called, the output from the delay line is rather different than the true signal. This difference can be audible if you have good ears, especially if the sound contains higher frequencies. It shows up as noise.

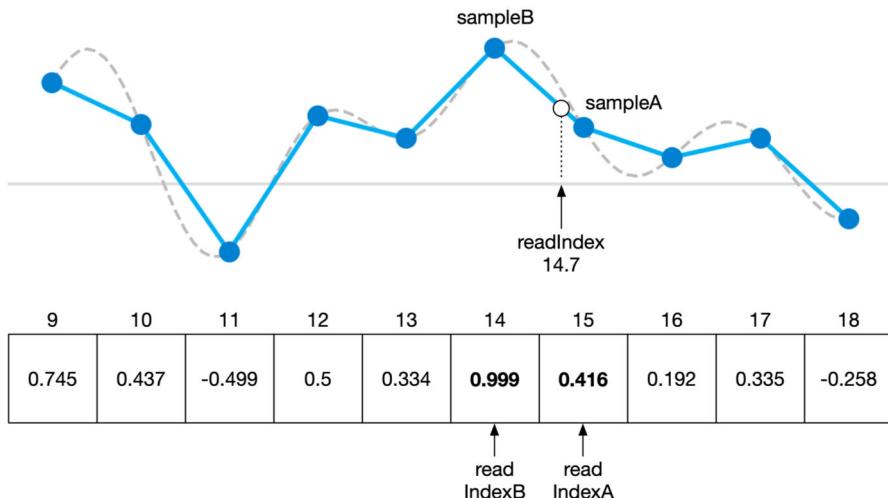
This method of rounding off the read index is also known as **nearest neighbors**. Maybe you've ever used a painting program to scale an image with the nearest neighbors setting and found that the result was very blocky. The same thing happens here, except with audio.

Try this: Play a pure sine wave as input audio, no feedback (use **Sine.wav** from the book's resources). While the sine wave is playing, gradually change the delay time. You should be able to hear a lot of noise in the high end as you move the **Time** knob back and forth. That noise happens because of `std::round`, isn't that wild?

Fortunately, with a bit of effort we can get much better results. Rather than rounding to the closest sample, we can make an educated guess as to what the intermediate value should be. We can't calculate it exactly as that would take too much time, but we can always estimate. This guessing is known as **interpolation**.

First we'll look at a basic interpolation method that is very common in all areas of audio programming, known as **linear interpolation** or sometimes **lerp**.

With linear interpolation, we draw straight lines between the sample points, like so:



Linear interpolation between two samples

This is a crude approximation of the true signal (plotted as a dashed gray line) but it's still a better approximation than rounding off.

To use linear interpolation in `DelayLine`, change the `read` function to the following.

Tip: You may want to keep the original version of `read` but put it between `/*` and `*/` symbols to comment it out. That way you can compare later how the two methods sound.

```

float DelayLine::read(float delayInSamples) const noexcept
{
    jassert(delayInSamples >= 0.0f);
    jassert(delayInSamples <= bufferLength - 1.0f);

    int integerDelay = int(delayInSamples);

    int readIndexA = writeIndex - integerDelay;
    if (readIndexA < 0) {
        readIndexA += bufferLength;
    }

    int readIndexB = readIndexA - 1;
    if (readIndexB < 0) {
        readIndexB += bufferLength;
    }

    float sampleA = buffer[size_t(readIndexA)];
    float sampleB = buffer[size_t(readIndexB)];

    float fraction = delayInSamples - float(integerDelay);
    return sampleA + fraction * (sampleB - sampleA);
}

```

To calculate the value of the in-between sample we need to read the values of the samples on the left and right, and find the average of these two.

For example, if our `readIndex` is 14.7, the sample we want sits in between samples 14 and 15. It is closer to 15, so that sample should contribute more to the average than sample 14.

We will read the samples at these two points into the variables `sampleA` and `sampleB`, where A is the sample on the right (index 15) and B is the sample on the left (index 14).

First we do `int(delayInSamples)` to strip off the fractional part of the delay. In our example, the `delayInSamples` was 12.3. Truncating that by casting the `float` to an `int` gives `integerDelay` the value 12.

Our write position was 27, so that makes `readIndexA = writeIndex - integerDelay` be $27 - 12 = 15$. Then `readIndexB` is one position earlier, $15 - 1 = 14$. We must make sure that both read index variables will properly wrap around if necessary.

We use `readIndexA` to read the value of `sampleA`, and `readIndexB` to read the value of `sampleB`. In our example, `sampleA` is 0.416 and `sampleB` is 0.999.

Since the interpolated sample lies closer to `sampleA` than to `sampleB`, the amount that `sampleA` should contribute is 0.7 and `sampleB` counts for 0.3.

The exact mixing amount is determined by the `fraction` variable, which in this example is `delayInSamples - float(integerDelay)` or $12.3 - 12 = 0.3$.

The final interpolated result is `sampleA*0.7 + sampleB*0.3 = 0.416*0.7 + 0.999*0.3 = 0.591`. Looking at the illustration, that seems about right. It's closer to `sampleA` than to `sampleB`.

The linear interpolation formula is:

```
interpolated = (1 - fraction)*sampleA + fraction*sampleB
```

The code in the `read` function doesn't use the above formula but a slight reformulation that saves a multiply:

```
interpolated = sampleA + fraction * (sampleB - sampleA)
```

You may find this looks very similar to the update formula for a one-pole filter. And you'd be right! The one-pole filter uses linear interpolation in its calculation. Likewise, a dry/wet mix that's done as `(1 - mix)*dry + mix*wet` is also a linear interpolation. Linear interpolation is everywhere.

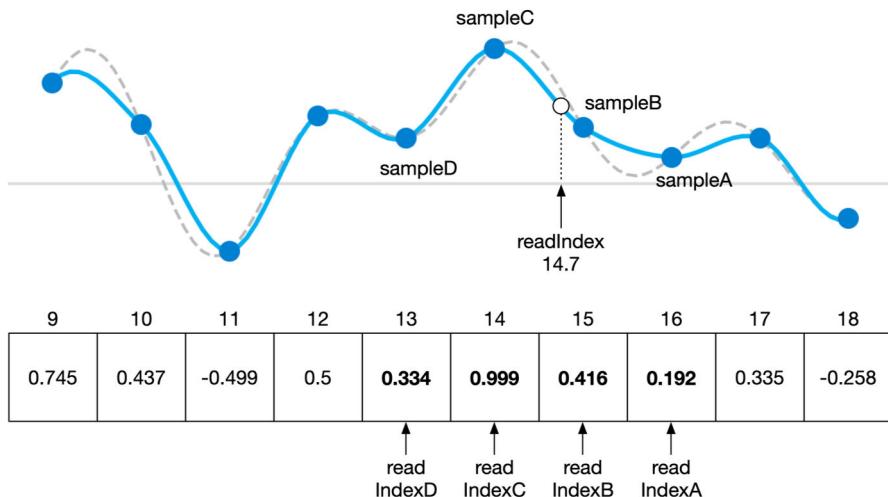
Try it out! Can you hear any difference with before? Doing the same test with the sine wave, the noise is mostly gone, or at least severely suppressed. The difference is quite stark.

It's easiest to hear with a clean signal such as a sine wave. On other program material such as drums or guitar, the effect is harder to notice because there's so much going on in the sound that noise artifacts get hidden below all the sound (which likely has a lot of noise of its own already).

Guess what? We can do even better than linear interpolation. Using a lerp is fast and better than nothing but with a bit of extra math we can improve on the results massively.

Hermite interpolation

Rather than draw straight lines between the points, we can calculate a rounded curve that goes through the points. It's likely that the true signal will be similar to these curves, although as the following illustration shows, it's still not perfect.



Interpolation using a cubic Hermite spline

To find the value for a given point, for example at `readIndex` 14.7 in our example, we will look at the four surrounding points and fit a curve through these four points. Then we read that curve at the “in between” position to get the interpolated value.

With linear interpolation we looked at only two points in total but here we'll look at the two points on the left and the two on the right.

There are several methods to draw a curve through four points but we'll use Hermite interpolation, also known as Catmull-Rom interpolation.

We'll write the new `read` function in two parts. The first part is:

```
float DelayLine::read(float delayInSamples) const noexcept
{
    jassert(delayInSamples >= 1.0f);
    jassert(delayInSamples <= bufferLength - 2.0f);

    int integerDelay = int(delayInSamples);
```

```

int readIndexA = writeIndex - integerDelay + 1;
int readIndexB = readIndexA - 1;
int readIndexC = readIndexA - 2;
int readIndexD = readIndexA - 3;

if (readIndexD < 0) {
    readIndexD += bufferLength;
    if (readIndexC < 0) {
        readIndexC += bufferLength;
        if (readIndexB < 0) {
            readIndexB += bufferLength;
            if (readIndexA < 0) {
                readIndexA += bufferLength;
            }
        }
    }
}

float sampleA = buffer[size_t(readIndexA)];
float sampleB = buffer[size_t(readIndexB)];
float sampleC = buffer[size_t(readIndexC)];
float sampleD = buffer[size_t(readIndexD)];

```

Instead of two points this reads four points: A, B, C, D. Note that B and C are the same points that were used by linear interpolation. A and D are two additional points.

The logic for wrapping the `readIndex` variables is a little different than before. The idea here is that point A has the largest index, so if index A is negative, then B, C, and D need to be wrapped too. Conversely, if point D (the smallest index) is not negative, we know that none of the points need to be wrapped around. By nesting the `if` statements like this, we avoid unnecessary checks.

One important gotcha is that the delay length cannot be less than 1 sample, which is why the `jassert` now checks for `delayInSamples >= 1.0f`.

With a delay length less than 1.0, `readIndexA` would be one beyond `writeIndex`, which would read the very oldest sample from the delay line into `sampleA`. Including that sample in the interpolation doesn't make any sense.

In practice having a minimum delay length of one sample isn't an issue. If necessary, there are other ways to delay the signal by only a fraction of a sample, such as using an all-pass filter.

Continue adding the rest of the `read` function. This is the part that creates the curve through the four samples we just read, and finds the interpolated value.

```

float fraction = delayInSamples - float(integerDelay);
float slope0 = (sampleC - sampleA) * 0.5f;
float slope1 = (sampleD - sampleB) * 0.5f;
float v = sampleB - sampleC;
float w = slope0 + v;
float a = w + v + slope1;
float b = w + a;
float stage1 = a * fraction - b;
float stage2 = stage1 * fraction + slope0;
return stage2 * fraction + sampleB;
}

```

Here's how this code works. Actually... want to hear a secret? I don't really know how this code works. I first heard about Hermite interpolation in a [talk given by Matt Tytel](#)³⁴, author of the Vital synth, at the Audio Developers Conference. He based his code on an [example from Laurent de Soras](#)³⁵.

I could probably figure out the math but I've never bothered — I tried running the code and it works great, and that's good enough for me. So you see, even authors of programming books don't know everything!

By the way, the maximum length of the delay is now `bufferLength - 2` because the interpolation reads two previous samples instead of just one. As a result, a delay time of 5000 ms should trigger the assertion that checks the maximum delay length. However, if you try this, the assertion might not actually do anything.

Recall that the **Delay Time** parameter is smoothed using a one-pole filter, to avoid zipper noise and to mimic an analog tape delay. One of the features of a one-pole filter is that it slows down the closer it gets to its target value. It can take a very long time for the **Delay Time** parameter to reach the value `5000.0f`. It might be "stuck" on something like `4998.56f`, slowly creeping towards 5000.

Just in case, to avoid any issues, in `setMaximumDelayInSamples` change the following line. Instead of adding 1 extra sample to the requested length, this adds 2 samples.

```
int paddedLength = maxLengthInSamples + 2;
```

Try it out! I personally can't really hear the difference between linear and Hermite interpolation in our plug-in, but perhaps you can. The difference is much smaller than with the nearest neighbor approach, which clearly added a lot of noise.

³⁴<https://www.youtube.com/watch?v=qlinVx60778>

³⁵<https://www.musicdsp.org/en/latest/Other/93-hermite-interpolation.html>

That said, it's possible to measure the difference between the various interpolation methods and, while none of them is perfect, Hermite does reduce interpolation artifacts much more than linear interpolation. For both approaches the noise levels are so low that they are hard to detect by ear.

Is the extra complexity of using Hermite interpolation worth it? That's something you have to decide on a case-by-case basis. For some plug-ins linear interpolation might be good enough, for others you want the best quality possible.

I just wanted to show you that we definitely need some kind of interpolation to estimate what the values in between samples are, as the nearest-neighbor rounding off method was audibly worse.

All forms of interpolation involve fitting some kind of curve through the points surrounding the value you're looking for. There are many other possibilities, such as third-order Lagrange interpolation, which is also available on JUCE's `juce::dsp::DelayLine`.

More about wrapping around

If you look at other people's delay line implementations, you may see that instead of doing the wrap-around like how we've been doing it,

```
writeIndex += 1;
if (writeIndex >= bufferLength) {
    writeIndex = 0;
}
```

they might write something like this:

```
writeIndex = (writeIndex + 1) % bufferLength;
```

The `%` operator in C++ is known as the **modulo** operator. It calculates the integer remainder of a division. For example, $6 \% 10 = 6$ but so is $16 \% 10$, because dividing 16 by 10 gives the answer 1 with a remainder of 6. The `/` operator on two integers gives the first part, the `%` operator gives the second part.

Going back to our example, if `bufferLength` is 5 and `writeIndex + 1` is less than 5, doing `% 5` will not change anything. But as soon as `writeIndex + 1` becomes 5, the modulo operation will automatically set it back to 0.

It's a neat trick but the modulo operator is problematic when applied to the `readIndex`. The read index should wrap around when it becomes negative, but $-2 \% 5$ gives -2 (you'd want it to be 3).

Sometimes you might see code like this:

```
writeIndex = (writeIndex + 1) & wrapMask;
```

This does conceptually the same thing as the `%` operator but more efficiently. The `&` operator is a so-called **bitwise and**. It combines the bits from two different integers. As `writeIndex + 1` becomes too large, the `&` operator sets the most significant bits in the variable to 0, which has the same effect as taking the remainder but is much faster.

However, there is a constraint to making the `&` approach work: the length of the delay line must be a power of two. A 5-element delay line is impossible — it would need to have 8 elements. For a five second delay line at 48 kHz, we now don't allocate 240000 numbers but 262144, since that is the next highest power of two. So, we're wasting some memory here. This memory waste will become worse the larger the delay line needs to be, since larger powers of two are further apart. Unlike `%`, the `&` trick works fine on negative numbers.

Whenever you're wondering which approach is fastest, it's a good idea to write a benchmarking program that tries each method under different circumstances and measures how long they take. Which I did!

So, which is fastest?

- using the `if` statements
- the `%` modulo operator
- bitwise masking with the `&` operator

The answer is: It Depends™. According to my tests, the speed is determined both by the delay time and the size of the circular buffer. The shorter the delay time, the less often wrapping happens, and expensive checks done on every sample start to matter.

Overall, the `%` version is slowest and the `&` version is fastest. However, in most realistic situations the `if` version performs just as well. The difference is that `%` and `&` are always being applied, even when wrapping isn't needed — which is most of the time (unless the delay time is relatively long).

The `if` statements only do any work when actual wrapping occurs, and the CPU's branch predictor makes these checks virtually free if the branch prediction is correct (which is most of the time). The `if`-based approach also does not have the requirement that the maximum length of the delay line is a power of two.

Note that I did my speed tests on a modern desktop class CPU. If you're planning to make audio software that runs on an embedded device with a less powerful CPU, the bitmasking version may turn out to perform way better. Again, if you're not sure: measure it, don't assume.

Unit tests

So far you've tested the plug-in by running it in a host and playing with it. For a comprehensive quality assurance test you'd need to try different sample rates and buffer sizes, all the possible combinations of parameters, rapidly changing the parameters to see if that blows up anything, adding/removing the plug-in while audio is playing, and so on. The goal is to put the plug-in under stress to see if it breaks.

Another method of testing we've used are the `jassert` assertion statements that verify function arguments and other variables have reasonable values. This helps to avoid mistakes that can easily be detected, so use these assertions everywhere you can.

There is an automated way to perform tests, known as **unit testing**. A unit test is a short program that tests the behavior of a single “unit”, such as the `DelayLine` class, in isolation.

Some useful tests for the `DelayLine` class could be:

- Test that `setMaximumDelayInSamples` indeed allocates the required amount of memory; this can be checked using `getBufferLength`.
- Test that `reset` indeed clears the contents of the delay line, i.e. writes silence into it. You can do this by doing `read` on all positions in the delay line, which should return zero everywhere.
- Test that `write` and `read` work properly, for example by allocating a small delay line, say 5 samples. Write the first sample and read at 0, 1, 2, 3 and 4 samples delay to make sure they are as expected. Write the next sample and read at every integer delay again, and so on.

- Test that `write` will wrap around when it reaches the end of the delay line. This can be done by allocating a small delay line, say 5 samples, and writing 6 numbers into it. After each write you check the value that is returned by `read(0)`.
- Test that interpolating gives the expected results. You do this by writing a known set of values into the delay line, and then doing `read` on a few in-between positions. This should match the expected values for those positions.
- And many more...

The `DelayLine` class is relatively simple, but for DSP building blocks with complex behavior, the unit tests also help to document what is considered correct operation for the code and what isn't.

Large projects have test suites of thousands of tests. Making permanent changes to the source code of such a project must first pass all the tests, so that you can be certain none of the existing functionality broke due to these changes.

Writing unit tests is outside the scope of this book but I wanted to mention them as they are very common in software engineering. You would put these tests in their own project, so they are independent of the plug-in.

[Catch2³⁶](#) is a popular unit testing framework for C++. JUCE itself comes with a set of unit tests. You can find the program that runs them in **JUCE/extras/UnitTestRunner**.

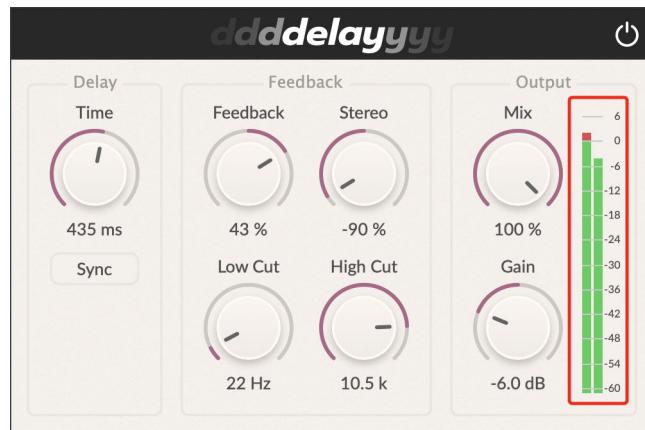
In the next chapter we'll add a level meter to the plug-in. The main challenge to solve there is how to communicate between the audio processing code and the plug-in's editor.

³⁶<https://github.com/catchorg/Catch2>

17: Adding a level meter

To allow the user to compensate for the increased amplitude from mixing the delayed signal into the original sound, the plug-in has an **Output Gain** knob. Adding a level meter to the plug-in will show the user how loud the combined output actually is, so they can determine how to set this gain knob.

In this chapter you'll build the following part of the UI:



Audio thread vs. UI thread

I've mentioned a few times now that the audio processing code and the plug-in's user interface should remain strictly separated. The way they communicate is through the plug-in parameters.

- When the user manipulates a knob in the editor, the parameter is changed, and the audio processing code will read the new value the next time it performs `params.update()`.

- The host may change the parameter through automation. This is detected by the parameter attachment object, which updates the UI in turn.
- It's also possible for the audio processing code to change the parameter, for example in response to receiving a MIDI CC message, but we're not doing this in our plug-in.

There are other situations where we want to communicate between the audio code and the UI, for example to draw a level meter. The level meter is a UI component that visualizes the current peak levels of the plug-in's output audio.

We need to measure these peak levels somewhere and the obvious place for this is in `processBlock`, which is part of the audio processing code. The UI component needs to be told about these peak levels somehow and draw them on the screen.

That is the problem we are going to solve in this chapter: How to measure something in `processBlock` and how to communicate this to the editor so it can draw that thing.

In case you were wondering, this is not a job for a plug-in parameter. We could certainly put the peak level in a parameter and use an attachment to communicate with the UI. However, it makes no sense for the host to try and automate the peak level reading. Besides, there is a more efficient method than using a parameter.

Note: You don't need to make everything into a parameter. For example, if the UI has tabs that reveal different panels, you wouldn't make the active tab index into a parameter. This isn't something you'd want to automate from the host, as it's not directly involved in the audio processing — it's just a UI thing. You do need to serialize such extra state when the session gets saved, so that next time the user opens the editor, the plug-in restores the tab that was last opened. To summarize: You only have to make things into parameters when they affect the audio processing and when it's something the host could possibly want to automate.

Measuring the peak levels

Measuring the audio level is simple enough: Inside the audio processing loop in `processBlock` we can keep track of what the largest output value is. Let's say we put this in member variables `levelL` and `levelR`, for the left and right channel, respectively.

The editor has a reference to the `DelayAudioProcessor` object, so if we put `levelL` and `levelR` in the `public:` section of `DelayAudioProcessor`, the editor can read these variables. However, there is a snag... the code in `processBlock` is performed on the audio thread, while the editor runs on the UI thread. It's a big no-no for one thread to access the variables that another thread may be using.

Weird stuff may happen if the audio thread is writing values into `levelL` and `levelR`, while at the same time the UI is trying to read these variables to draw the level meter. The C++ compiler does not have a concept of threads — those are handled by the operating system — and so it won't understand that two threads might be trying to use the same variables at the same time.

In normal programming you would fix this issue using a **lock** such as a **mutex** (which stands for mutual exclusion). When one thread wants to read or write a shared variable, it first takes a lock. If some other thread tries to obtain that lock, it will have to wait until the first thread is done. It's a "You can't use the toilet while I'm using the toilet!" kind of thing, so you're locking the toilet door until you're done.

Unfortunately for us, in audio programming these kinds of locks are taboo on the audio thread, since they could potentially block the audio process, making it miss its deadline and causing glitches in the sound.

An easy way around this problem is to use so-called **atomic variables**. An atomic variable can safely be accessed by multiple threads at the same time. Importantly, when you use an atomic, the C++ compiler knows that this variable may be used by more than one thread, and it will take this into consideration when compiling your code so that the weird stuff doesn't happen.

We'll use these atomics to communicate between the audio processor and the editor. In `PluginProcessor.h`, add the following line to the `public:` section of the class:

```
std::atomic<float> levelL, levelR;
```

The `std::atomic<float>` says, "This is not a plain `float` but one that is protected so it can be used by multiple threads at once."

In `PluginProcessor.cpp`, in `prepareToPlay`, set these variables to zero:

```
levelL.store(0.0f);
levelR.store(0.0f);
```

You could have written `levelL = 0.0f;` as if it were a regular `float`, which does the same thing, but the `store(...)` makes it clearer that you're dealing with an atomic.

Next, in `processBlock` just before the big `for` loop, add these two lines:

```
float maxL = 0.0f;
float maxR = 0.0f;
```

These local variables will be used to measure the peak level for the current block. These are regular `floats`, not atomics, as they'll only be used inside `processBlock`.

Inside the loop, change these two lines:

```
outputDataL[sample] = mixL * params.gain;
outputDataR[sample] = mixR * params.gain;
```

into:

```
float outL = mixL * params.gain;
float outR = mixR * params.gain;

outputDataL[sample] = outL;
outputDataR[sample] = outR;

maxL = std::max(maxL, std::abs(outL));
maxR = std::max(maxR, std::abs(outR));
```

We first put the output values into their own variables, `outL` and `outR`. To measure the peak level, we update `maxL` only if `std::abs(outL)` is greater than the current value of `maxL`, which is initially zero. Likewise for `maxR` on the right channel.

You could also have written it as follows:

```
if (std::abs(outL) > maxL) {
    maxL = std::abs(outL);
}
```

Or even:

```
maxL = (std::abs(outL) > maxL) ? std::abs(outL) : maxL;
```

But the `std::max()` function is just as easy to use. It takes two arguments and returns the one that is largest.

The reason for doing `std::abs()` is to take the **absolute value**. Recall that samples are numbers in the range `-1.0` to `+1.0`. We want to measure the sample with the largest magnitude, regardless of whether that sample is positive or negative. `std::abs()` simply makes negative values positive.

Below the `for` loop, add these lines to put the measurements into the atomic variables:

```
levelL.store(maxL);
levelR.store(maxR);
```

Excellent, that concludes the work the audio processing code has to do. Per block it keeps track of the largest sample value it has seen — positive or negative — and writes this into the atomic variable, so it can be safely read by the editor.

Next up, let's create the component that will draw the levels in the UI.

The LevelMeter component

In **Projucer**, add new files to the project using **Add New Component class (split between a CPP & header)**... and name the new component **LevelMeter**.

How do we know when the audio processor has measured new peak levels, so we can redraw this `LevelMeter` component? The easy answer is: we don't. Instead, we're going to check every so often what the current `levelL` and `levelR` values are and draw them. This is called **polling**.

The `LevelMeter` component will poll the `DelayAudioProcessor` by periodically reading the `levelL` and `levelR` values. This polling happens from a timer that will fire 60 times per second.

Polling may seem dumb... Why can't the `DelayAudioProcessor` send a message to the `LevelMeter` saying, "Hey I've got a new measurement"? It's not as silly as you may think: In order to get smooth animations, we want the `LevelMeter` component to redraw itself somewhere between 30 and 60 times per second.

The audio processor measures the peak levels once per block, so how often it delivers new values depends on the block size. Therefore, the timing of the audio process is not going to be the same as the timing of the drawing code.

Plus, the editor may not even exist, for example when the user hasn't opened the plug-in UI yet. The audio processing code should not assume there even is an editor. Therefore, it's best if the editor initiates this kind of communication rather than the other way around.

To add a timer to a component, you make it inherit from `juce::Timer`. In **LevelMeter.h**, change the class declaration to:

```
// 1
class LevelMeter : public juce::Component, private juce::Timer
{
public:
    // 2
    LevelMeter(std::atomic<float>& measurementL,
               std::atomic<float>& measurementR);
    ~LevelMeter() override;

    void paint (juce::Graphics&) override;
    void resized() override;

private:
    // 3
    void timerCallback() override;

    // 4
    std::atomic<float>& measurementL;
    std::atomic<float>& measurementR;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (LevelMeter)
};


```

The changes are as follows:

1. Add `private juce::Timer` to the list of classes `LevelMeter` inherits from. The `private` keyword means all the `juce::Timer` functions and member variables are added as private members to `LevelMeter`.
2. The constructor arguments are references to the two atomic variables.
3. This is the function that will be called 60 times per second by the timer. In here we'll read the peak levels and redraw the component.
4. These variables store **references** to the atomics that were passed into the constructor. The & means these variables are not `std::atomics` themselves, but they refer to the `std::atomic` variables from the audio processor, although `LevelMeter` doesn't really care where they come from.

Let's implement the bare minimum so we can compile the plug-in and try it out. In **LevelMeter.cpp**, change the constructor to:

```
LevelMeter::LevelMeter(std::atomic<float>& measurementL,
                      std::atomic<float>& measurementR)
    : measurementL(measurementL_), measurementR(measurementR_)
{
    setOpaque(true);
    startTimerHz(1);
}
```

We want to store the arguments `measurementL` and `measurementR` into member variables with the same names. Using the same name for an argument and a variable will give a compiler warning, so I've renamed the arguments to `measurementL_` and `measurementR_` here.

Using an underscore `_` for this is a personal preference. It's OK if the arguments have different names in the **.h** file than in the **.cpp** file. In the member initializer list, we copy these references into our member variables.

Inside the constructor's code we set the component to be opaque, which means it draws all its pixels with a non-transparent color. Any underlying UI components will not shine through. This is an optimization that potentially saves drawing things that wouldn't be visible anyway. Recall that the `RotaryKnob` does have transparent regions that allow the noise texture to shine through. We don't want that here.

Finally, `startTimerHz(1)` starts the timer with a firing rate of 1 Hz, or once per second. This is just for demonstrating that the timer works, later we'll change the rate to 60 Hz.

We'll leave the `paint` and `resized` functions unchanged for now.

Add the `timerCallback` function at the bottom of the source file:

```
void LevelMeter::timerCallback()
{
    DBG("left: " << measurementL.load() << ", right: " << measurementR.load());
}
```

This uses the `DBG` statement to write the current level measurements to the debug output. We use `load()` to read the value from the `std::atomic` variables.

We still need to add the `LevelMeter` component into the editor. Go to **PluginEditor.h** and add an include.

```
#include "LevelMeter.h"
```

Then in the `private:` section, create the `LevelMeter` object and put it in a member variable:

```
LevelMeter meter;
```

Since the `LevelMeter` constructor expects arguments, we must supply these in the member initializer list of `DelayAudioProcessorEditor`'s own constructor.

Go to **PluginEditor.cpp** and change the constructor to:

```
DelayAudioProcessorEditor::DelayAudioProcessorEditor (DelayAudioProcessor& p)
    : AudioProcessorEditor (&p),
      audioProcessor (p),
      meter(p.levelL, p.levelR)
{
    // ...existing code...
```

This grabs references to the audio processor object's `levelL` and `levelR` variables and passes them into the `LevelMeter` constructor. Note that in the `LevelMeter` source code these arguments are named `measurementL` and `measurementR` instead, because I want to use the name "level" for another variable in that class.

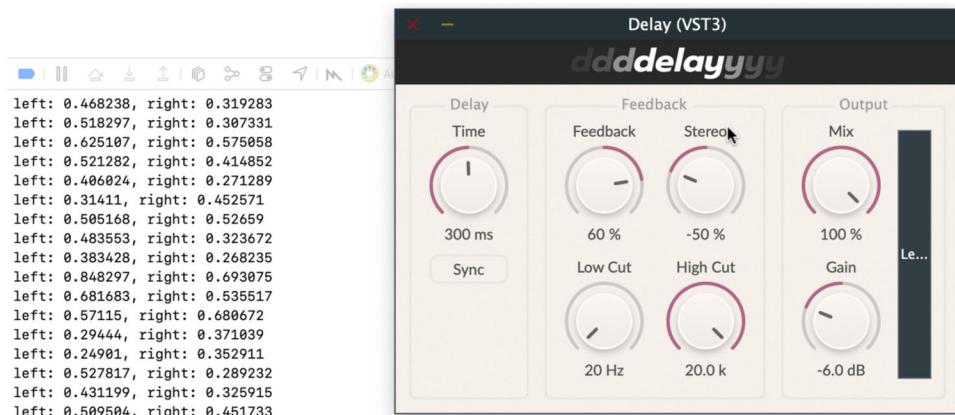
A bit lower in the constructor, write:

```
outputGroup.addAndMakeVisible(meter);
```

Finally, in `resized`, position the new component inside the `outputGroup`:

```
meter.setBounds(outputGroup.getWidth() - 45, 30, 30, gainKnob.getBottom() - 30);
```

That should be enough to get everything running again. Try it out, the editor now has a — rather ugly — level meter component on the right. When you play some audio, the IDE's debug pane will display the most recently measured peak levels once per second. Turn the **Output Gain** knob and you should see these levels become higher or lower.



The peak levels are printed to the IDE's debug output pane

Drawing the meter

The `levelL` and `levelR` variables give us a reading between 0.0 and 1.0, or possibly higher than 1.0 if the output signal is too loud. It's more common to display these measurements as decibels.

You'll remember that we used `juce::Decibels::decibelsToGain` to convert from a decibel value to a linear gain, which was used with the **Output Gain** parameter. Here we want to do it the other way around, from a linear gain to decibels, for which we can use `juce::Decibels::gainToDecibels`.

We need to map these decibel values to pixel coordinates. I've decided that the top of the `LevelMeter` component will correspond to +6 dB and the bottom will be -60 dB. If the output signal is quieter than -60 dB, it won't be drawn. I also want to put tick marks every 6 dB.

Exactly where these dB values and tick marks will be drawn depends on the height of the `LevelMeter` component. We will calculate these positions in `LevelMeter`'s `resized` function. This way you can resize the component to any height and it will draw everything correctly and proportionally.

This technique is known as **vector graphics**, and it's one of the advantages of doing the drawing with `juce::Graphics` rather than with images, since images always have fixed dimensions.

We will declare some new constants and variables to help with this. In **LevelMeter.h**, in the `private:` section of the class, add the following:

```
static constexpr float maxdB = 6.0f;
static constexpr float mindB = -60.0f;
static constexpr float stepdB = 6.0f;
```

The `maxdB` constant is to the largest decibel value we want to show, here +6 dB. `mindB` is the smallest decibel value, -60 dB. `stepdB` is the step size for the tick marks. By defining these as named constants it's easy to change them later. Perhaps you want the largest level to be +12 dB, in which case you'd simply change `maxdB` and the drawing code automatically adjusts.

Also add these private variables:

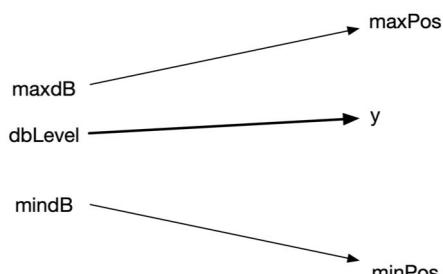
```
float maxPos = 0.0f;
float minPos = 0.0f;
```

These are the y-coordinates for the pixel positions corresponding to `maxdB` and `mindB`, respectively. We will fill these in inside the `resized` function.

Add a new function in the `private:` section of the class:

```
int positionForLevel(float dbLevel) const noexcept
{
    return int(std::round(juce::jmap(dbLevel, maxdB, mindB, maxPos, minPos)));
}
```

This function is simple enough that we can put it in the header file. Given a level in decibels, it converts the dB value to a pixel position.



Mapping decibel levels to pixel coordinates

The `juce::jmap` function takes its first argument, here `dbLevel`, and maps it from the range `[mindB, maxdB]` into the range `[minPos, maxPos]`.

- If `dbLevel` is `-60.0f`, which is equal to `mindB`, the result is `minPos`.
- If `dbLevel` is `6.0f`, which is equal to `maxdB`, the result is `maxPos`.
- For any decibel levels in between `-60.0f` and `6.0f`, the result is a pixel coordinate proportionally between `minPos` and `maxPos`.

The output from `juce::jmap` may be a fractional number, so we round it off with `std::round` and cast to an `int`, so that everything gets drawn exactly on whole number pixel coordinates, as drawing on a fractional coordinate creates a blurry mess.

Go to **LevelMeter.cpp** and change `resized` to the following:

```
void LevelMeter::resized()
{
    maxPos = 4.0f;
    minPos = float(getHeight()) - 4.0f;
}
```

This sets `maxPos` to 4 pixels from the top of the component and `minPos` to 4 pixels from the bottom, so that there is a little extra space to draw the tick labels. We now have all the information needed to draw the levels, so let's implement the painting logic.

Go to **LookAndFeel.h** and add some more color definitions to the `Colors` namespace:

```
namespace Colors
{
    // ...

    namespace LevelMeter
    {
        const juce::Colour background { 245, 240, 235 };
        const juce::Colour tickLine { 200, 200, 200 };
        const juce::Colour tickLabel { 80, 80, 80 };
        const juce::Colour tooLoud { 226, 74, 81 };
        const juce::Colour levelOK { 65, 206, 88 };
    }
}
```

In **LevelMeter.cpp**, add an import for the look-and-feel code so we can access these new colors:

```
#include "LookAndFeel.h"
```

First we'll draw the background and the tick marks, as these are static and will never move (unless the component gets resized, which doesn't happen in our plug-in).

Change the `paint` function to the following:

```
void LevelMeter::paint (juce::Graphics& g)
{
    const auto bounds = getLocalBounds();

    g.fillAll(Colors::LevelMeter::background);

    g.setFont(Fonts::getFont(10.0f));
    for (float db = maxdB; db >= mindB; db -= stepdB) {
        int y = positionForLevel(db);

        g.setColour(Colors::LevelMeter::tickLine);
        g.fillRect(0, y, 16, 1);

        g.setColour(Colors::LevelMeter::tickLabel);
        g.drawSingleLineText(juce::String(int(db)), bounds.getWidth(), y + 3,
                            juce::Justification::right);
    }
}
```

This has a `for` loop that goes from `maxdB` down to `mindB` in steps of `stepdB`. In other words, it starts at `6.0f` and decrements in steps of `6.0f` until it reaches `-60.0f`.

Previously when we had a `for` loop it always counted up using `++i`. Here we're counting down using the `-=` operator. The loop variable is named `db` and is a `float` that represents a certain level in decibels.

Every iteration of the loop we convert the level from `db` to a pixel coordinate using `positionForLevel` and draw a horizontal line using `g.fillRect`. The text label is drawn on the right.

Try it out. The level meter should now look like the following. It doesn't display the actual levels yet. That's the topic of the next section.



The tick marks and decibel levels

Drawing the levels

We're reading the level measurements in `timerCallback` but need to draw the levels in `paint`. How do we get those values from `timerCallback` into `paint`?

In `LevelMeter.h`, add two constants to the `private:` section:

```
static constexpr float clampdB = -120.0f;
static constexpr float clampLevel = 0.000001f; // -120 dB
```

The thing about decibels is that a linear gain of 0.0 corresponds to a decibel value of `-inf` or minus infinity. We don't have an infinite number of pixels at our disposal, so we will limit, or clamp, the decibel values to -120 dB.

Also add the following variables to the `private:` section:

```
float dbLevelL;
float dbLevelR;
```

Recall that member variables should be initialized. We could have written the following in the header file, similar to what we've been doing for the other member variables:

```
float dbLevelL = clampdB;
float dbLevelR = clampdB;
```

However, there are two other techniques that are commonly used in C++ to initialize variables. You've seen both before. We can use the member initializer list in the constructor in **LevelMeter.cpp**, like so:

```
LevelMeter::LevelMeter(std::atomic<float>& measurementL,
                      std::atomic<float>& measurementR)
    : measurementL(measurementL_), measurementR(measurementR_),
      dbLevelL(clampdB), dbLevelR(clampdB)
{
    setOpaque(true);
    startTimerHz(1);
}
```

Or put assignment statements inside the constructor's code:

```
LevelMeter::LevelMeter(std::atomic<float>& measurementL,
                      std::atomic<float>& measurementR)
    : measurementL(measurementL_), measurementR(measurementR_)
{
    dbLevelL = clampdB;
    dbLevelR = clampdB;

    setOpaque(true);
    startTimerHz(1);
}
```

All these approaches will give `dbLevelL` and `dbLevelR` the initial value `clampdB`. I wanted to show you the different ways to initialize member variables in C++ so that you'll recognize them in future code. The member initializer list is generally preferred, so I suggest you use that one.

Next, change `timerCallback` to the following:

```
void LevelMeter::timerCallback()
{
    dbLevelL = juce::Decibels::gainToDecibels(measurementL.load(), clampdB);
    dbLevelR = juce::Decibels::gainToDecibels(measurementR.load(), clampdB);

    repaint();
}
```

This grabs the value from the atomic variable `measurementL`, converts it to decibels and writes it into `dbLevelL`. Likewise for `measurementR` and `dbLevelR`.

If the gain is `0.0f`, then `gainToDecibels` will return the value from `clampdB`. (If you don't provide this second argument, it will use a default of `-100 dB`.)

Because `measurementL` and `measurementR` are atomic variables, the `load()` operation is thread-safe, meaning that it's OK if the audio processing code is trying to change these variables while we're reading them.

After writing the values from the measurements into our own `dbLevelL` and `dbLevelR` member variables, the timer callback function calls `repaint`, which tells JUCE that the component should be repainted.

Note: Sometimes developers new to JUCE will try to call `paint();` directly. This is not correct! JUCE, or more precisely, the operating system, will determine when the component gets redrawn. When we want to update the component, we use `repaint();` to tell JUCE that painting should happen. JUCE will then perform `paint` at its earliest convenience.

To draw the levels, we will add a new function, `drawLevel`. This paints a vertical bar representing the level for a single channel. Add the function declaration in the `private:` section in `LevelMeter.h`:

```
void drawLevel(juce::Graphics& g, float level, int x, int width);
```

The `x` and `width` arguments determine where this bar will be drawn and how wide it will be. We must also pass in a reference to the `juce::Graphics` object so that we have access to the drawing commands.

In `LevelMeter.cpp`, add the implementation of this function:

```
void LevelMeter::drawLevel(juce::Graphics& g, float level, int x, int width)
{
    int y = positionForLevel(level);
    if (level > 0.0f) {
        int y0 = positionForLevel(0.0f);
        g.setColour(Colors::LevelMeter::tooLoud);
        g.fillRect(x, y, width, y0 - y);
        g.setColour(Colors::LevelMeter::levelOK);
        g.fillRect(x, y0, width, getHeight() - y0);
    } else if (y < getHeight()) {
        g.setColour(Colors::LevelMeter::levelOK);
        g.fillRect(x, y, width, getHeight() - y);
    }
}
```

Using `positionForLevel` we convert the level value from decibels into the corresponding pixel y-coordinate. We draw a green bar from that y coordinate down to the bottom of the component. Where the level is higher than 0 dB, the bar is drawn in red to indicate the sound is too loud.

Note that we don't draw anything if the level is too low, which is when the y coordinate falls below the lower edge of the component. `g.fillRect` doesn't like it if you give it a rectangle with a negative height.

Nice, now all that's left to do is call this function from `paint`, once for the left channel and once for the right channel.

Add the following lines to the `paint` function, right after the `g.fillRect` line but before the loop that draws the tick marks and the labels:

```
drawLevel(g, dbLevelL, 0, 7);
drawLevel(g, dbLevelR, 9, 7);
```

Oh, before I forget, in the constructor change the timer frequency to 60 Hz instead of 1 Hz:

```
startTimerHz(60);
```

Try it out. Run the plug-in and play some audio and you'll see the level meter react to the sound. Nice! Boost the **Output Gain** until the sound is too loud and the level meter should turn red near the top. On a mono track both channels should always draw the same level.



The levels are drawn as green bars

One thing you may have noticed is that the animation can be a little choppy. This happens because we measure the peak levels over individual blocks. If the block size is fairly large, say, 1024 samples, the level measurements are only updated every $1024 / 48000 = 21$ milliseconds (at a sample rate of 48 kHz). The UI repaints faster than this, every 16 milliseconds.

Even with smaller block sizes, the results can still be quite choppy. For example, try it with **Beep.wav** and no feedback. As soon as the sound ends, the level bar immediately disappears. We can make this look nicer by smoothing the levels.

One-pole filter for smoothing

We can make the level meter look more professional by adding a subtle animation.

When a new measurement is higher than the currently displayed level, we'll immediately jump to that higher level. But when the new measurement is lower, the meter will smoothly decay rather than jump. This decay animation is performed using a one-pole filter.

In **LevelMeter.h**, add the following lines to the `private:` section:

```
static constexpr int refreshRate = 60;

float decay = 0.0f;
float levelL = clampLevel;
float levelR = clampLevel;
```

The `refreshRate` is how fast the component redraws, in other words how often `timerCallback` is invoked. We need this rate to calculate `decay`, the filter coefficient for the one-pole filter.

The `levelL` and `levelR` variables store the current level in linear units, before the dB calculation. Just like the `dbLevel` variables are initially `clampdB`, the `level` variables start out at `clampLevel`, which is `0.000001f`. That is the same as -120 dB.

We need a new function that will apply the filter, so put its declaration in the `private:` section as well:

```
void updateLevel(float newLevel, float& smoothedLevel, float& leveldB) const;
```

Switch to **LevelMeter.cpp** and add the following line to the constructor:

```
decay = 1.0f - std::exp(-1.0f / (float(refreshRate) * 0.2f));
```

This calculates the coefficient for the one-pole filter. You may recall this formula from when we first encountered this filter. The `0.2f` sets the decay time to 200 ms. Where previously this formula used `sampleRate`, here we use `refreshRate`, since that's essentially the “sample rate” of the `timerCallback` function.

In the constructor, also change the `startTimerHz` line to use the `refreshRate` constant instead of the hardcoded value. This way you can change the redraw rate in one place and both the filter coefficient and the timer will use the same value.

```
startTimerHz(refreshRate);
```

Next, add the implementation for the new `updateLevel` function:

```
void LevelMeter::updateLevel(float newLevel, float& smoothedLevel,
                           float& leveldB) const
{
    if (newLevel > smoothedLevel) {
        smoothedLevel = newLevel; // instantaneous attack
    } else {
        smoothedLevel += (newLevel - smoothedLevel) * decay;
    }

    if (smoothedLevel > clampLevel) {
        leveldB = juce::Decibels::gainToDecibels(smoothedLevel);
    } else {
        leveldB = clampdB;
    }
}
```

The `newLevel` argument is the value we've read from the atomic float. At this point it is still a linear gain, a value between `0.0f` and `1.0f` — or larger if the sound is too loud.

The `smoothedLevel` argument is a **reference** because of the `&`. It will refer to either the `levelL` or `levelR` member variable.

Making this argument a reference means we can write to `smoothedLevel` inside the function and it will overwrite the original variable. You've seen this technique before with the `panningEqualPower` function.

The first `if-else` block applies the animation:

- If the new level is larger than the currently displayed level, immediately jump to this new level. This is like having an instantaneous attack.
- If the new level is smaller, use the one-pole filter formula to do an exponential fade out of the meter.

Even though this changes the local variable `smoothedLevel`, because this variable is a reference, it's really the `levelL` and `levelR` variables that will get smoothed by the one-pole filter.

We convert the smoothed level to decibels and write it into the `leveldB` variable. This is also a reference and will refer to either `dbLevelL` or `dbLevelR`. Recall that `dbLevelL` and `dbLevelR` are used to actually draw the levels.

The reason we smoothen the linear gain values instead of the decibel values, is that the one-pole filter creates an exponential curve. The conversion to decibels involves a logarithmic operation, which is the inverse of the exponential. The logarithm will cancel out the exponent.

Because we draw in decibels, the smoothed animation will be linear — it moves with a constant speed. If we had instead smoothed the decibel levels, they would initially drop faster and slow down near the bottom of the level meter, which looks strange.

Finally, call `updateLevel` from the timer callback:

```
void LevelMeter::timerCallback()
{
    updateLevel(measurementL.load(), levelL, dbLevelL);
    updateLevel(measurementR.load(), levelR, dbLevelR);

    repaint();
}
```

Try it out. The animation is a lot smoother now. Whenever the level drops, it doesn't immediately jump to the lower value, but does so gradually.

When you stop the audio playback, you'll notice that the levels gently fall off, just like they do in your DAW.

Improving the measurements

The level meter already works quite well but we can still improve it. `processBlock` always overwrites the value that is stored in the atomic, regardless of how large or small it was. For example, it's possible we measure a level of 0.9 in one block and 0.2 in the next block. The atomic would be updated to 0.2, even though for our purposes it would make more sense to keep the 0.9.

Many blocks may be processed in between calls to `timerCallback`, especially if the audio buffer size is small. With a block size of 32 samples at 48 kHz, there will on average be 25 calls made to `processBlock` for every time the timer fires in the `LevelMeter`.

What `LevelMeter` sees is only the level measured in the latest block. It would be more appropriate to keep track of the highest level that was recorded since the last time we painted, no matter how many blocks get processed in the meantime.

As a general rule, the audio processor should just do its thing and not care about the editor at all. We don't want the audio code to worry about "When was the last time the editor drew the levels?"

Instead, we can use the atomic variable to let the `LevelMeter` communicate back to `processBlock` that it has read and displayed the measurement. Then `processBlock` can start a new series of measurements.

To make this a bit easier, we'll move the atomic variable into a helper object named `Measurement`.

In **Projucer**, right-click the **Source** group and choose **Add New Header File...**. Name it **Measurement.h** and save the project. The source code is small enough to fit in the header file, so we're not adding a **.cpp** file.

Replace the contents of **Measurement.h** with the following.

```
#pragma once

#include <atomic>

struct Measurement
{
    void reset() noexcept
    {
        value.store(0.0f);
    }
}
```

```

void updateIfGreater(float newValue) noexcept
{
    auto oldValue = value.load();
    while (newValue > oldValue &&
           !value.compare_exchange_weak(oldValue, newValue));
}

float readAndReset() noexcept
{
    return value.exchange(0.0f);
}

std::atomic<float> value;
};

```

This is a struct, not a class. In C++ the difference between a class and struct is very small. You generally use a struct when you want all the functions and variables to be public.

In **PluginProcessor.h**, add an include for this new file:

```
#include "Measurement.h"
```

Replace the **std::atomic<float>** variables with:

```
Measurement levelL, levelR;
```

In **PluginProcessor.cpp**, in `prepareToPlay` replace the lines,

```
levelL.store(0.0f);
levelR.store(0.0f);
```

with:

```
levelL.reset();
levelR.reset();
```

That does the same thing, setting the atomic variable to zero, but it happens inside a function from the `Measurement` object.

In `processBlock`, replace the lines,

```
levelL.store(maxL);
levelR.store(maxR);
```

with:

```
levelL.updateIfGreater(maxL);
levelR.updateIfGreater(maxR);
```

The `updateIfGreater` function is the key to making this work: It will only write `maxL` into `levelL` if the current value of `levelL` is smaller. This is the same as doing `levelL = std::max(levelL, maxL)` except the operation is atomic, keeping the variable safe to be used by multiple threads.

The code inside `updateIfGreater` doesn't do `value.store(newValue)`, but uses a so-called CAS loop (compare and swap) to make sure `newValue` is only placed into the atomic if it's larger than the atomic's current value. CAS loops are often used to build lock-free data structures. Explaining exactly how this CAS loop works is outside the scope of this book.

I do want to point out that the `while` statement in `updateIfGreater` is a so-called **while loop**, which repeats as long as the condition between the `()` parentheses is true.

Just like a `for` loop, `while` loops can have code between `{ }` braces. This particular `while` loop has no `{ }` braces, only a `;` to indicate there's no code in the loop itself. It simply keeps trying the condition over and over until it succeeds.

It's perfectly fine if the code in `updateIfGreater` makes no sense to you. Lock-free programming using atomics is an advanced topic. I first saw this technique in the source code of a plug-in called [Clean Machine³⁷](#). It looked interesting and so I copied it into my notes. A lot of software development consists of using tricks and techniques discovered by other people!

To make use of this new `Measurement` object we need to change a few things in `LevelMeter.h`. First, add the required include below the `<JuceHeader.h>` include:

```
#include "Measurement.h"
```

³⁷<https://amalgamatedsignals.com/clean-machine>

Change the constructor to use `Measurement` instead of `std::atomic<float>` for the arguments:

```
LevelMeter(Measurement& measurementL, Measurement& measurementR);
```

Likewise, the member variables are now references to `Measurement` objects:

```
Measurement& measurementL;
Measurement& measurementR;
```

In `LevelMeter.cpp`, also change the arguments for the constructor:

```
LevelMeter::LevelMeter(Measurement& measurementL_, Measurement& measurementR_)
```

Note that here again I've added an underscore _ behind the argument names, to avoid conflicts with the member variables with the same names.

Finally, change `timerCallback` to call `readAndReset()` on the measurement variables instead of `load()`:

```
updateLevel(measurementL.readAndReset(), levell, dbLevelL);
updateLevel(measurementR.readAndReset(), levelR, dbLevelR);
```

The `readAndReset` function reads the value from the atomic and immediately sets the atomic back to zero. This happens in a single operation, using `value.exchange(0.0f)`, so that it's thread safe.

The reason we set the atomic back to `0.0f` is that this signals to the audio thread it's OK to write smaller measurements again in the next `processBlock`, since any new measurement will be greater than `0.0f` — at least if sound is playing.

That will do it. `processBlock` will still measure the maximum sample value per block like it used to, but only overwrites the atomic if this new measurement is louder than before. Once the UI thread fires the timer, `LevelMeter` will read the atomic and set it back to zero to tell `processBlock` that this measurement has been consumed.

Try the plug-in with a few different block sizes. Recall that in `AudioPluginHost` you can double-click on the **Audio Input** or **Audio Output** blocks to change the audio buffer size. Notice that the level meter stays smooth, regardless of the block size.

Note: Communication between the audio processor and the editor can get much more complicated than what we did in this chapter. We used a `std::atomic<float>` instead of a plain `float`. To send an entire buffer of audio for a visualizer such as an oscilloscope or a spectrum analyzer, you'll need to use a more sophisticated method such as a FIFO. This is a circular buffer, just like the one you used to build the `DelayLine` class in the previous chapter, where the audio processor is writing new samples into the circular buffer and the UI is reading from it. To allow both threads access to the same data structure will require some extra bookkeeping, but as long as you're using a so-called lock-free FIFO, this will be real-time safe.

In the next chapter you'll put the finishing touches to the plug-in.

18: Sweating the details

In this chapter we'll tie up a few of the loose ends and really polish the plug-in. Often it's the little details that take most of the work!

Fixing the delay time knob artifacts

When you turn the **Delay Time** knob while audio is playing, there is an audible pitch-shifting effect. By itself this sounds kind of cool and mimics what happens on analog tape hardware. But if feedback is enabled, the pitch up/down effect gets caught in the feedback line and keeps repeating, which sounds bad to me.

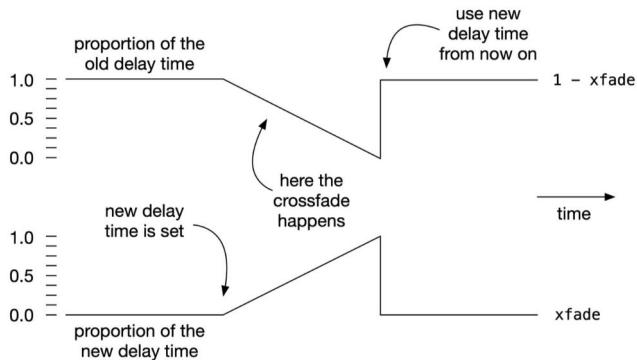
There are different ways to remove or avoid the pitch-shifting effect. In this book we'll look at two approaches: crossfading and ducking.

The first method is to perform a quick **crossfade** between the old delay time and the new delay time whenever the user — or the host — changes the **Delay Time** parameter. Crossfading is a common technique in audio programming and is often used to avoid glitches when there are sudden changes.

During the crossfade, we'll read the delay line in two places: at the old delay time and at the new delay time. Essentially this creates a multi-tap delay for a brief moment. The crossfade blends the two sample values in a proportion that varies over time.

At first, we can only hear the samples read at the old delay time, while the samples read at the new delay time are silent. But over time, the old delay time's influence drops and the samples from the new delay time start to get mixed in more and more, until eventually we're only hearing the sound from the new delay time.

The duration of the crossfade is fairly short, such as 50 milliseconds, but this is enough to create a smooth transition between the old and new delay times. When the crossfade is done, we'll switch to the new delay time and resume normal operation.



The crossfading curves

Go to **PluginProcessor.h** and add the following variables to the `private:` section:

```
float delayInSamples = 0.0f;
float targetDelay = 0.0f;
float xfade = 0.0f;
float xfadeInc = 0.0f;
```

The `delayInSamples` variable is the current delay time. We're already using a local variable with this name in `processBlock`. This will become a member variable now, so that it remembers its value between invocations to `processBlock`. Since we can't assume anything about the block size, the crossfade can easily start in one block and end in the next one, or even take place across several successive blocks.

The `targetDelay` variable will hold the new delay time. When the user changes the **Delay Time** parameter, we'll read the new value into `targetDelay`, and perform the crossfade between the tap at `delayInSamples` (the old time) and the tap at `targetDelay` (the new time).

`xfade` is a variable that will count from `0.0f` to `1.0f` in a linear fashion. This is the mix amount between the samples that we'll read from the two taps.

We want the crossfade to always take 50 ms, which means that the number of samples it spans depends on the sample rate. This is why we need the `xfadeInc` variable. This is the step size by which `xfade` counts up. The larger the sample rate, the smaller this step size will be.

In **PluginProcessor.cpp**, in `prepareToPlay`, add the following lines to reset the variables to their initial values.

```

delayInSamples = 0.0f;
targetDelay = 0.0f;
xfade = 0.0f;
xfadeInc = static_cast<float>(1.0 / (0.05 * sampleRate)); // 50 ms

```

When `xfade` has a value of `0.0f`, like it does here, it means we're currently not cross-fading. This is how the audio processing code will know whether to crossfade or not.

Since `prepareToPlay` receives the current sample rate as an argument, this is a good place to calculate the step size `xfadeInc`. The value `0.05` means 50 milliseconds. At a sample rate of 48 kHz, 50 ms corresponds to 2400 samples, meaning it will take 2400 timesteps to perform our crossfade, and so the step size is `1 / 2400` or `0.000417f`.

The calculation is performed using doubles since the `sampleRate` argument is a double. The `static_cast<float>` converts this result into a float. Previously we wrote `float(...)` to cast something into a float. Either way works fine, but C++ purists will prefer the `static_cast`.

We'll need to make some changes to `processBlock`. First, replace these two lines,

```

float delayTime = params.tempoSync ? syncedTime : params.delayTime;
float delayInSamples = delayTime / 1000.0f * sampleRate;

```

with the following:

```

// 1
if (xfade == 0.0f) {
    // 2
    float delayTime = params.tempoSync ? syncedTime : params.delayTime;
    targetDelay = delayTime / 1000.0f * sampleRate;

    // 3
    if (delayInSamples == 0.0f) { // first time
        delayInSamples = targetDelay;
    }
    // 4
    else if (targetDelay != delayInSamples) { // start crossfade
        xfade = xfadeInc;
    }
}

```

This code reads the **Delay Time** parameter value, or **Delay Note** when tempo sync is enabled, and decides whether to perform a crossfade.

How this works, step-by-step:

1. What happens when the **Delay Time** parameter changes while a crossfade is still taking place? That sounds like asking for trouble. To avoid this, we will only start a new crossfade if an existing crossfade is not already underway and simply ignore the new **Delay Time** value.
2. This is the same code as before, except now the new delay length is written into the `targetDelay` member variable.
3. The very first time `processBlock` is performed after a `prepareToPlay`, there is no need to crossfade and we can immediately switch to the new **Delay Time** value.
4. If `targetDelay` is the same as `delaySamples`, the **Delay Time** parameter did not change since last time and nothing needs to happen. This is going to be the case 99% of the time. However, if these two variables are not equal, we start a new crossfade by setting `xfade` to `xfadeInc` so it is no longer `0.0f`.

Let's write the code that performs the actual crossfade. Immediately after the following lines that read from the delay line,

```
float wetL = delayLineL.read(delayInSamples);
float wetR = delayLineR.read(delayInSamples);
```

add this logic:

```
// 1
if (xfade > 0.0f) { // crossfading?
    // 2
    float newL = delayLineL.read(targetDelay);
    float newR = delayLineR.read(targetDelay);

    // 3
    wetL = (1.0f - xfade) * wetL + xfade * newL;
    wetR = (1.0f - xfade) * wetR + xfade * newR;

    // 4
    xfade += xfadeInc;
    if (xfade >= 1.0f) {
        delayInSamples = targetDelay;
        xfade = 0.0f;
    }
}
```

What this does step-by-step:

1. We can skip the crossfading code if `xfade` equals `0.0f`. But if `xfade` is greater than zero, a crossfade is currently in progress.
2. Read the sample values at the new delay length given by `targetDelay`. This is easy with our own `DelayLine` class since we can pass any delay length into the `read()` function.
3. Perform the crossfade. The `wet` variable contains the sample at the old delay length, `delayInSamples`. We will mix this with the new variable, which has the sample at the new delay length. `xfade` is a value between `0.0f` and `1.0f` that determines how much each sample contributes to the total.

Initially, `xfade` is a small value close to zero and so `1.0f - xfade` is a value close to one, meaning that a large portion of the `wet` variable is used and only a small portion of the new variable. But as `xfade` gradually counts up to one, these proportions will change and the samples read at the old delay length will fade out while the ones from the new delay length will fade in.

Note that `xfade` and `(1.0f - xfade)` will always add up to one. This is yet another example of a linear interpolation.

4. Increment the `xfade` counter. When it reaches `1.0f`, the crossfade is done. We make `delayInSamples` equal to `targetDelay` to permanently switch to the new delay time, and set `xfade` back to `0.0f` to signal we're done with the crossfade.

Note that we check if `xfade >= 1.0f` rather than `xfade == 1.0f`. Due to floating-point imprecision, it's quite unlikely that `xfade` will land exactly on the value `1.0f`, but perhaps on something like `1.000000119f`. That's why we need to use `>=` to check for greater-than-or-equal.

Great, that adds a basic crossfade to the plug-in that happens whenever the **Delay Time** or **Delay Note** parameters changes value.

Run the plug-in to try it out! You may find that it still sounds a bit odd. Any ideas why?

To see what happens, we can output the crossfade envelope and look at it with an oscilloscope. The **envelope** of a sound describes the contours of the sound's amplitude. If you load an audio file into an audio editor and zoom out, it shows the overall shape of the waveform — that is the envelope.

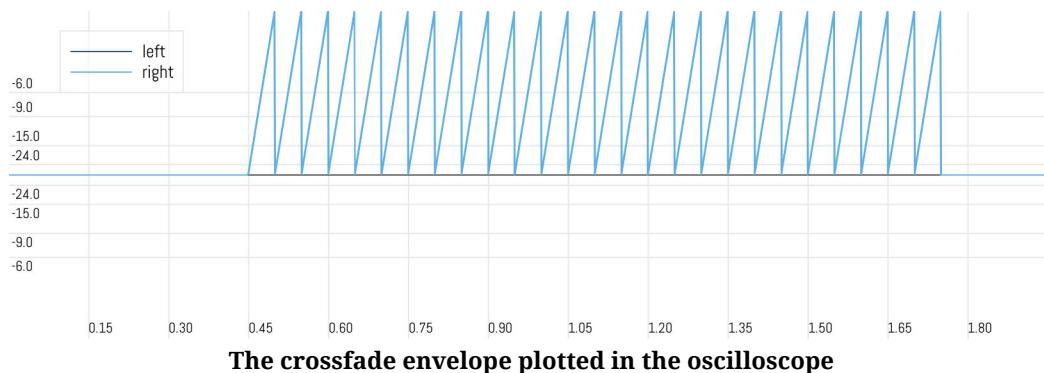
In `processBlock`, immediately before the following lines,

```
outputDataL[sample] = outL;
outputDataR[sample] = outR;
```

write:

```
outL = xfade;
outR = xfade;
```

Instead of outputting the processed sample data, this writes the `xfade` variable into the audio buffer. Run the plug-in in `AudioPluginHost` and disconnect the **Audio Output** from the **Delay** block. Turn the **Time** knob and look at what happens in the oscilloscope:



For every turn of the knob, you'll see a series of spikes. Each of these is a crossfade happening, since `xfade` starts at 0.0 and over the course of 50 ms increments towards 1.0 and then is reset to zero again.

The reason there is more than one spike is that the **Delay Time** parameter is smoothed. Turning the knob doesn't just give one new delay time value, it gradually ramps between the old and new values. The result is that we'll perform a bunch of additional crossfades because the parameter keeps changing. This is what makes it sound a little odd.

From this investigation we can conclude that the parameter smoothing is conflicting with the crossfade. Since we're doing the crossfade, we don't actually need to smoothen this parameter anymore.

In **Parameters.cpp**, in `smoothen()`, replace the following line:

```
delayTime += (targetDelayTime - delayTime) * coeff;
```

with:

```
delayTime = targetDelayTime;
```

Try it again and you'll see fewer spikes when you turn the knob. There may still be more than one crossfade happening. When you turn the **Time** knob, say from 100 ms to 800 ms, JUCE doesn't wait with changing the parameter until you're done moving the knob. The plug-in will see some of the intermediate knob positions as well, for example the values at 200 ms, 400 ms, and 600 ms.

Since the crossfade is short, and the user turns the knob relatively slowly, the audio processor may perform several of these crossfades in a row before it arrives at the final parameter value. This is to be expected. If you type in a new value rather than turning the knob, you'll see only one crossfade happen.

Remove the lines we added for outputting the envelope, and see if you can hear the difference. It should be better now.

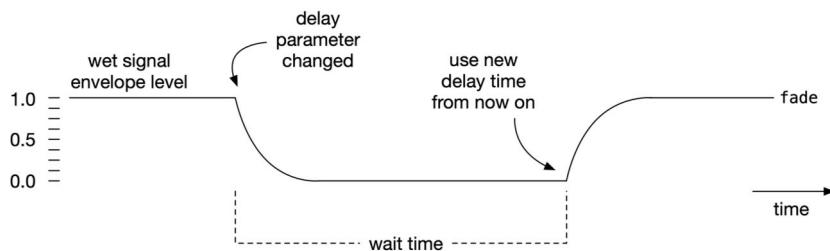
Switching between time and tempo sync mode should be glitch free too, just as toggling between different note lengths. Previously this would glitch because of the sudden jump in time, but now we always crossfade between the two different delays. This works even if you "jump" from a short delay time, such as 5ms, to a long delay time such as 5000 ms. No more glitches!

Is this a great solution? I think it sounds pretty good but it's not perfect. When using a pure sine wave as input and turning the knob, you can hear it modulates the amplitude of sine wave because of the repeated crossfades. If you use feedback, you may still get "weird" results when modulating the delay time, but that's also due to the nature of feedback — at least the pitching issue is gone. Be sure to experiment with different crossfade durations to see what kind of effect they have.

Note: There are other ways you can implement this kind of crossfade. You could read the **Delay Time** parameter every 256 samples, for example. In between reading successive parameter reads, do a crossfade that exactly lasts 256 samples. This is simpler to implement but it does make the fade time depend on the sample rate.

Ducking

Another way to avoid the pitch-shifting effect is to use **ducking**, where the delay effect is temporarily turned off while the user is manipulating the **Delay Time** knob. When done well, this feels very natural and it's hardly noticeable that the delay briefly disappears.



Ducking temporarily fades out the wet signal

Similar to the previous section, we'll keep a variable `fade` that describes the current envelope level of the signal. Initially this is 1.0, so that the sound plays at its regular volume.

As soon as the **Delay Time** parameter changes, a one-pole filter will exponentially decay `fade` towards zero, gradually fading out the echoes from the wet signal. The dry signal is passed through unchanged. After a short waiting period, the delay time is changed to the new time, and `fade` will exponentially rise until it reaches 1.0, fading in the wet signal again.

If the user manipulates the **Delay Time** knob while the ducking is in progress, it will simply lengthen the waiting time before the sound fades back in. This way we'll only resume playing the delayed signal once the user has let go of the knob.

To implement the ducking approach, start from the previous version of the plug-in without the crossfading. These two methods don't work well together, so it's an either-or situation.

You can comment out the changes you made in the previous section by adding `//` in front of those lines or by putting them in between `/*` and `*/` symbols. Or use the code from the previous chapter from the book's resources as the starting point.

In **PluginProcessor.h**, add the following variables to the `private:` section:

```
float delayInSamples = 0.0f;
float targetDelay = 0.0f;
float fade = 0.0f;
float fadeTarget = 0.0f;
float coeff = 0.0f;
float wait = 0.0f;
float waitInc = 0.0f;
```

When the user changes the **Delay Time** parameter, we don't immediately switch to the new delay time. Instead, we'll put it in `targetDelay` and then start fading out the wet signal. Once we're ready to fade in again, we'll switch `delayInSamples` to the value from `targetDelay`.

In **PluginProcessor.cpp**, in `prepareToPlay`, add the following lines to give the variables their initial values:

```
delayInSamples = 0.0f;
targetDelay = 0.0f;

fade = 1.0f;
fadeTarget = 1.0f;

coeff = 1.0f - std::exp(-1.0f / (0.05f * float(sampleRate)));

wait = 0.0f;
waitInc = 1.0f / (0.3f * float(sampleRate)); // 300 ms
```

The `fade` variable is the current envelope level, `fadeTarget` is the level that the one-pole filter is trying to reach. Initially both are `1.0f`, meaning that the filter isn't going to do anything — it will only spring into action once we set `fadeTarget` to `0.0f` in `processBlock`.

`coeff` is the coefficient for the one-pole filter. You've seen this formula before. The `0.05f` means it takes 50 ms for the filter to reach 63.2% of its target value.

We'll hold for a minimum amount of time until fading back in. This is what the `wait` variable is for: It counts from `0.0f` up to `1.0f`.

I want this waiting period to last for 300 ms, and so `waitInc` is the step size by which `wait` is incremented on each sample timestep. This is essentially the same as the `xfadeInc` variable from the previous section.

Next up we'll need to make some changes to `processBlock`. First, replace these two lines,

```
float delayTime = params.tempoSync ? syncedTime : params.delayTime;
float delayInSamples = delayTime / 1000.0f * sampleRate;
```

with the following:

```
// 1
float delayTime = params.tempoSync ? syncedTime : params.delayTime;
float newTargetDelay = delayTime / 1000.0f * sampleRate;

// 2
if (newTargetDelay != targetDelay) {
    // 3
    targetDelay = newTargetDelay;
    // 4
    if (delayInSamples == 0.0f) { // first time
        delayInSamples = targetDelay;
    }
    // 5
    else {
        wait = waitInc;      // start counter
        fadeTarget = 0.0f;   // fade out
    }
}
```

This code reads the **Delay Time** parameter value, or **Delay Note** when tempo sync is enabled, and decides whether to perform ducking.

How this works, step-by-step:

1. This is the same code as before, except now the new delay length is written into the `newTargetDelay` variable.
2. If the delay time has not changed, we'll keep going as normal. However, if the delay time did change, we'll need to start fading out.
3. Remember the new delay time by putting it in `targetDelay`.
4. The very first time `processBlock` is performed after a `prepareToPlay`, there is no need to duck and we can immediately switch to the new **Delay Time** value.
5. Start the `wait` counter to begin ducking. By setting `fadeTarget` to `0.0f`, we initiate a fade-out and activate the one-pole filter.

Let's write the code that performs the actual ducking. Still in `processBlock`, immediately after the following lines that read from the delay line,

```
float wetL = delayLineL.read(delayInSamples);
float wetR = delayLineR.read(delayInSamples);
```

add this logic:

```
// 1
fade += (fadeTarget - fade) * coeff;

// 2
wetL *= fade;
wetR *= fade;

// 3
if (wait > 0.0f) {
    wait += waitInc;
    if (wait >= 1.0f) {
        // 4
        delayInSamples = targetDelay;
        wait = 0.0f;
        fadeTarget = 1.0f; // fade in
    }
}
```

What this does step-by-step:

1. The famous one-pole filter formula. It slowly and smoothly moves the value of `fade` towards `fadeTarget`. This is always happening, whether we're actually ducking or not. In the latter case, `fade` and `fadeTarget` are identical and so `fade` will stay at its current value.
2. Apply the `fade` value as the envelope of the wet signal. Most of the time `fade` will be `1.0f` and nothing happens to the delayed sound. However, when we're ducking, `fade` will be `0.0f` — or at least be decaying towards zero — and the wet signal is suppressed.
3. If `wait` is larger than zero, it means we're ducking. Increment `wait` by `waitInc` until it has reached `1.0f`. Recall that `wait` gets reset to the (small) value of `waitInc` whenever the user manipulates the knob. So, when `wait >= 1.0f`, it means the user hasn't moved the knob for 300 ms or roughly a third of a second.
4. When the holding period is over, switch over to the new delay length and start fading in by setting the filter's target value to `1.0f` again.

Note that the fading happens independently of the waiting logic. The waiting is purely there to throttle the number of parameter updates.

Just like with the crossfading approach, we don't need to smoothen the **Delay Time** parameter now. That would keep resetting the hold period, making it much longer than it should be.

In **Parameters.cpp**, in `smoothen`, replace the following line:

```
delayTime += (targetDelayTime - delayTime) * coeff;
```

with:

```
delayTime = targetDelayTime;
```

Great, that's all you need to do to add ducking whenever the **Delay Time** or **Delay Note** parameters changes value.

Run the plug-in to try it out! I think that works really well. Even with lots of feedback, there is no weird sound at all in the feedback loop, since we turn off the delay while interacting with the knob.

Again, we can see what happens by outputting the envelope rather than the audio. In `processBlock`, immediately before the following lines,

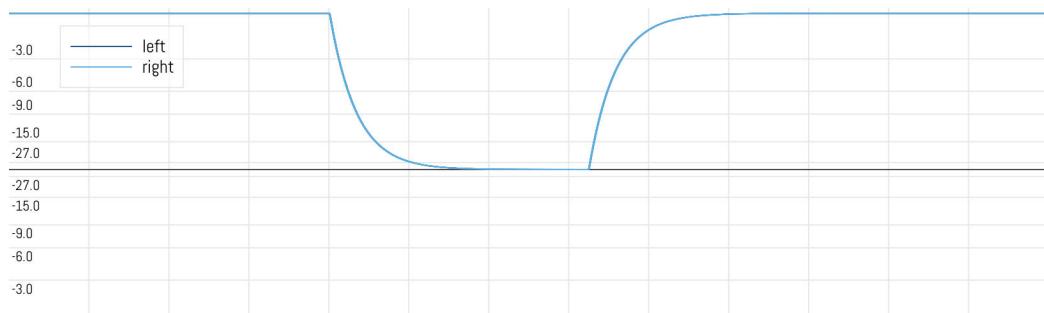
```
outputDataL[sample] = outL;
outputDataR[sample] = outR;
```

write:

```
outL = fade;
outR = fade;
```

Instead of outputting the processed sample data, this writes the `fade` variable into the audio buffer. Run the plug-in in `AudioPluginHost` and disconnect the **Audio Output** from the **Delay** block.

Turn the **Time** knob and look at what happens in the oscilloscope.



The ducking envelope plotted in the oscilloscope

For every turn of the knob, you'll see an exponential decay where the wet signal gets faded out, followed by a short pause, and an exponential fade-in to bring the wet signal back into the mix but with the new delay time.

What happens when **Delay Time** parameter changes while we're already ducking or fading in/out? Whenever the user manipulates the knob, the `wait` variable immediately takes on the value `waitInc` again, essentially resetting the hold timer, and `fadeTarget` is made `0.0f` to begin a new fade out.

As a result, additional parameter changes simply prolong the time we stay ducked. Try it: As long as you keep moving the knob, the envelope level stays at zero and the wet signal stays suppressed.

Note: In the previous section we used a linear crossfade while here we used the exponential fade from a one-pole filter. We could have used the one-pole filter in the previous section too, or done a linear fade while ducking. There are also other formulas, such as `sqrt(1 - xfade)` for fading out and `sqrt(xfade)` for fading in, where `xfade` is a counter that goes from 0 to 1. However, these fade times are so short they are pretty much inaudible — which is the whole point — and so it doesn't really matter what formula we use.

Adding a bypass button

Most DAWs have a button that lets the user bypass the plug-in to temporarily skip its audio processing, without having to remove the plug-in from the track. This is useful for A/B testing the difference the plug-in makes to the sound.

JUCE lets us add a special **bypass parameter** that many hosts will recognize and connect to their own bypass button. Unfortunately, this feature does not work in all DAWs. Some will completely stop sending the `processBlock` command while the plug-in is bypassed.

The downside is that in such DAWs the plug-in has no opportunity to gracefully switch to bypassed mode. Instead, on one block the plug-in is still active and on the next block it's not. This sudden change creates glitches in the sound, which makes A/B tests harder because the pop or click you hear from toggling the bypass button is very distracting.

This is why many plug-ins have their own bypass button that is independent of the DAW. In this section you'll add such a bypass button to the plug-in.

As always, it starts by adding a new parameter ID in **Parameters.h**:

```
const juce::ParameterID bypassParamID { "bypass", 1 };
```

Add a new variable to the `public:` section:

```
bool bypassed = false;
```

Also add a pointer to a `juce::AudioParameterBool` object to the `public:` section, so it can be accessed by the audio processor later.

```
juce::AudioParameterBool* bypassParam;
```

In **Parameters.cpp** add to the constructor:

```
castParameter(apvts, bypassParamID, bypassParam);
```

Create the new `AudioParameterBool` object in `createParameterLayout`:

```
layout.add(std::make_unique<juce::AudioParameterBool>(
    bypassParamID, "Bypass", false));
```

And read its value in `update`:

```
bypassed = bypassParam->get();
```

So far nothing new here. Let's also create a button in our plug-in's editor. First, you'll need to add an image file to the project's binary data.

In **Projucer**, in the **File Explorer**, right-click the **Assets** folder and choose **Add Existing Files....** From the book's resources choose the **Bypass.png** image file. Save the project from Projucer.

In **PluginEditor.h**, add these lines to the `private:` section:

```
juce::ImageButton bypassButton;

juce::AudioProcessorValueTreeState::ButtonAttachment bypassAttachment {
    audioProcessor.apvts, bypassParamID.getParamID(), bypassButton
};
```

This is the same as what you did for the tempo sync button, except this time it's a `juce::ImageButton` object.

In **PluginEditor.cpp**, in the constructor, add the following code, as always before the `setSize(...)` line:

```
auto bypassIcon = juce::ImageCache::getFromMemory(BinaryData::Bypass_png,
                                                 BinaryData::Bypass_pngSize);
bypassButton.setClickingTogglesState(true);
bypassButton.setBounds(0, 0, 20, 20);
bypassButton.setImages(
    false, true, true,
    bypassIcon, 1.0f, juce::Colours::white,
    bypassIcon, 1.0f, juce::Colours::white,
    bypassIcon, 1.0f, juce::Colours::grey,
    0.0f);
addAndMakeVisible(bypassButton);
```

This creates a 20×20-pixel button that draws the **Bypass.png** image. This image is a black icon on top of a transparent background. The `setImages` function lets you

specify that it should be drawn in a different color. We're using `white` for the button's normal and highlighted states and `grey` for when the image is pressed down.

In `resized` add the following line to position the button in the top-right corner of the window:

```
bypassButton.setTopLeftPosition(
    bounds.getRight() - bypassButton.getWidth() - 10, 10);
```

To actually make this bypass button do something, we'll have to write some code in the audio processor. First, go to **PluginProcessor.h** and add the following function signature to the `public:` section:

```
juce::AudioProcessorParameter* getBypassParameter() const override;
```

The `getBypassParameter` function is part of `juce::AudioProcessor`. That function normally returns `nullptr`, which tells JUCE the plug-in doesn't have a bypass parameter. We'll have to override this function and return a pointer to our new parameter.

In **PluginProcessor.cpp**, add the implementation of this function:

```
juce::AudioProcessorParameter* DelayAudioProcessor::getBypassParameter() const
{
    return params.bypassParam;
}
```

This is why you made `bypassParam` public in the `Parameters` object, so that `getBypassParameter` can tell JUCE that one of our plug-in parameters acts as the bypass toggle. JUCE will now attempt to hook this up to the DAW's own bypass button, at least for DAWs that support this feature.

All right, one more thing to do. In `processBlock` we can read `params.bypassed` to determine whether the effect should be bypassed or not. We could put the following somewhere at the top of `processBlock`, right below the call to `params.update()`:

```
if (params.bypassed) { return; }
```

This would immediately bail out of `processBlock` if the plug-in is in bypass mode. However, if you turn the plug-in back on a couple of seconds later, it will have old contents in the delay line that no longer make sense at this point in the song. One way to fix this is to clear out the delay line when bypassed.

Another solution is to keep processing the audio like normal but only output the dry signal. That makes sense if you're treating the bypass as a way to A/B test the sound.

We'll keep feeding the input audio into the delay line and calculating the wet signal. When the user turns bypass off again, the wet signal is immediately ready to get mixed into the output. Let's implement this option.

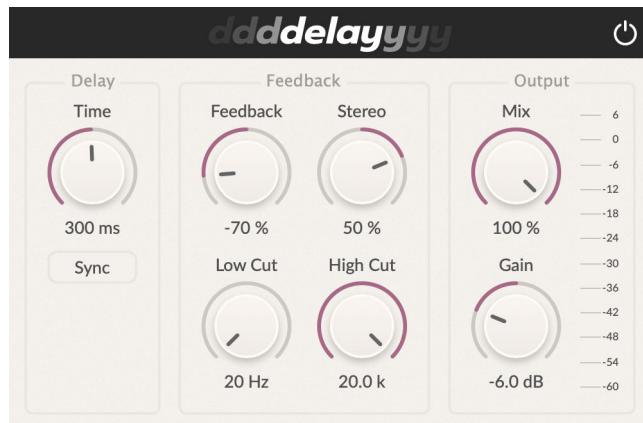
In `processBlock`, immediately below the lines,

```
float outL = mixL * params.gain;
float outR = mixR * params.gain;
```

insert the following:

```
if (params.bypassed) {
    outL = dryL;
    outR = dryR;
}
```

When in bypassed mode, we'll still do all the computations but we'll simply output only the dry signal. Run the plug-in and it will now have a bypass button in the top-right corner.



The plug-in with the bypass button

Try toggling the bypass button on and off and verify that the plug-in keeps sounding good. If you move the **Time** or **Feedback** knobs as the plug-in is bypassed, it should immediately pick up these changes once you turn off the bypass.

Also try it out in your DAW. As mentioned, not all DAWs support this bypass parameter, which is why we gave the plug-in its own bypass button too.

One thing that typically happens with an on/off button like bypass is that the sound can glitch. We always immediately disable or re-enable the wet signal, and there is bound to be a small discontinuity when that happens.

As an exercise for the reader, try fading out the wet signal over a few milliseconds when bypass is turned on, and fading it back in once bypass is disabled again. That should stop the glitches from happening. You've already seen how to do (cross)fading in the previous sections, so I'm sure you'll be able to get this to work!

Notice how taking care of these small details, such as preventing glitches when turning the **Time** knob or when toggling the **Bypass** button, can end up taking up a lot of development time. As they say, the devil is in the details...

About presets and programs

The IDE still complains about a few “unused parameter” warnings, in the following functions in **PluginProcessor.cpp**:

These functions exist to enable so-called **factory presets** for the plug-in, or **programs** in JUCE terminology. A preset is a particular set of plug-in parameter values.

If you wanted to, you could implement these functions to support factory presets, so that they show up in your DAW's presets menu. However, the functionality offered by these programs is rather limited and not all DAWs fully support them. In practice this means most plug-ins do not implement these program functions and instead have their own preset manager.

The advantage of building your own preset manager is that it allows users to create presets that are independent of the DAW, letting a Logic Pro user share presets with an Ableton Live user, for example.

How to build your own preset manager is outside the scope of this book but I can heartily recommend [this excellent video by Akash Murthy³⁸](#) that explains step-by-step how to add presets to your plug-ins. I suggest that you watch his video and try to implement a preset manager for the Delay plug-in yourself.

To get rid of the “unused parameter” warning messages in your IDE, simply remove the argument names, like so:

```
void DelayAudioProcessor::setCurrentProgram (int)
{
}

const juce::String DelayAudioProcessor::getProgramName (int)
{
    return {};
}

void DelayAudioProcessor::changeProgramName (int, const juce::String&)
```

Using the debugger

I've mentioned a few times that your IDE comes with a debugger and that this tool can help track down errors and programming mistakes. In this section I want to show the rudimentary principles of operating the debugger.

³⁸<https://www.youtube.com/watch?v=YwAtWuGA4Cg>

You've already seen what happens when we hit an assertion such as `jassert`. This means the program has reached a state it shouldn't be in. Let's trigger such an assertion on purpose just to have an excuse to play with the debugger.

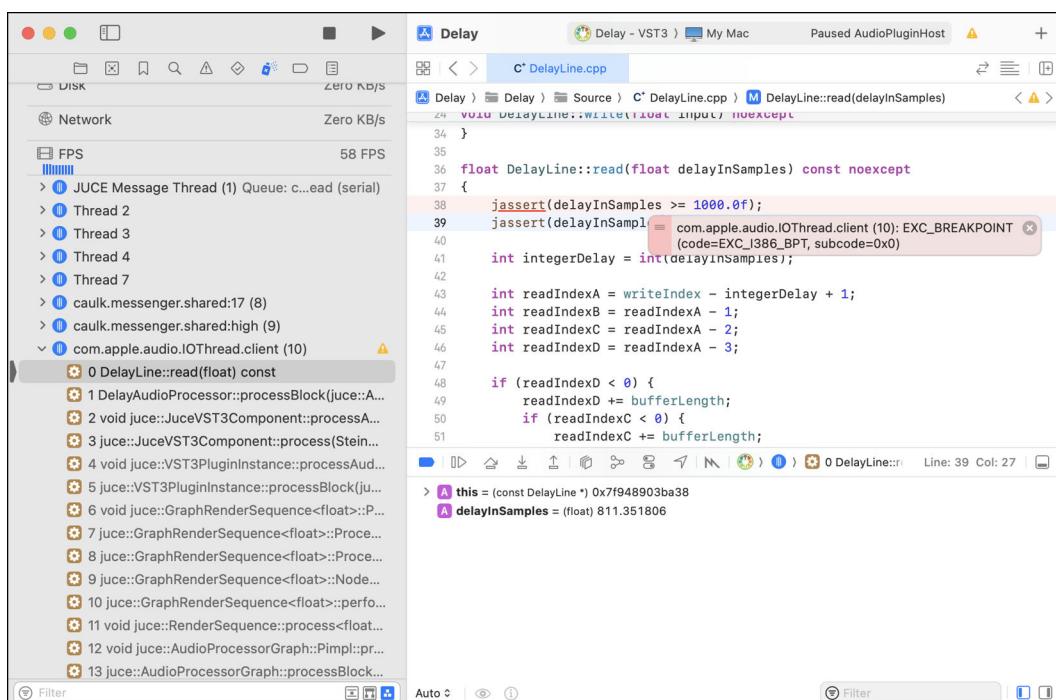
In `DelayLine.cpp`, in the `read` function, change the first `jassert` line to:

```
jassert(delayInSamples >= 1000.0f);
```

The assertion should now trigger when you choose a delay time less than 20 or so milliseconds. Rebuild and run the plug-in. Xcode will automatically run the plug-in in the debugger. In Visual Studio you'll need to press **F5** or the dark green run button that says **Local Windows Debugger**.

Slowly turn the **Delay Time** knob to the left. When it drops below 20 ms, the plug-in will pause and the IDE will drop into the debugger.

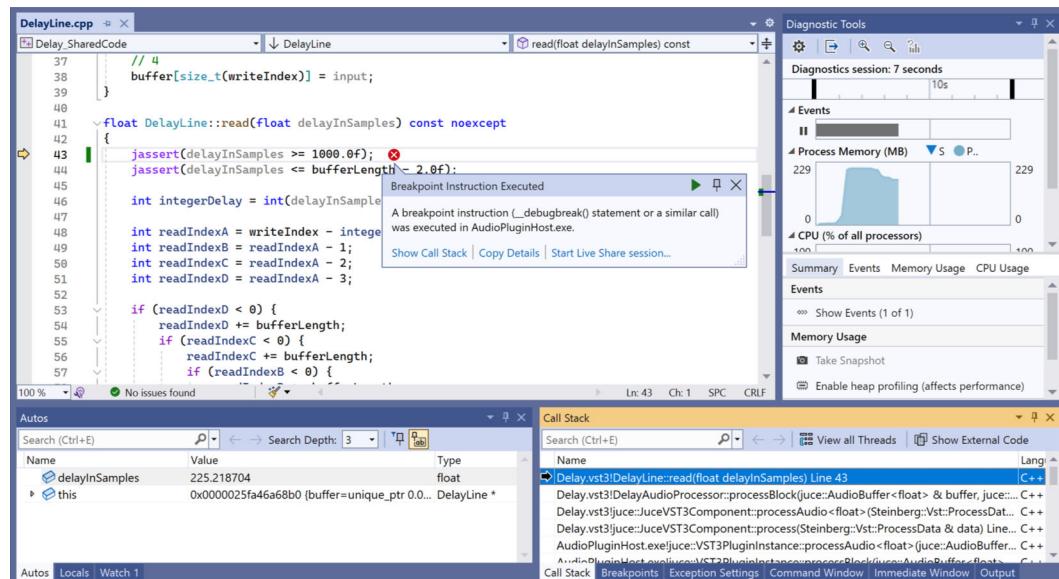
In Xcode it will look like this:



The Xcode debugger

In the source code editor, there is a red error pointing at the `jassert` line. The text in the error message is something like, “com.apple.audio.IOThread.client (10): EXC_BREAKPOINT (code=EXC_I386_BPT, subcode=0x0)”. This message is a bit cryptic. It says which thread the error occurred on, `com.apple.audio.IOThread.client`, and what type of crash it is: `EXC_BREAKPOINT`, which is typical for an assertion.

In Visual Studio, an error message box is pointing at the `jassert` line. It says a breakpoint instruction was executed, which is something that happens in an assertion.



The Visual Studio debugger

An important region of the debugger window to pay attention to is the **call stack**, which shows what functions were being called to get to this point. In Xcode, the call stack is in the pane on the left, in Visual Studio it's in the pane at the bottom right.

The most recent function is at the top, `DelayLine::read(float)` const. This is where the assertion occurred. The line below it says `DelayAudioProcessor::processBlock` since that's where we called `read` from.

Below that, the call stack lists functions that are part of the JUCE framework, such as `juce::JuceVST3Component::processAudio`, then functions from the host, and ultimately functions from the operating system. And so we can go down the call stack to see which function called what.

You can click (or double-click) on any of these functions to look inside. A function name in gray means there is no source code available for that function. You can still click it but the IDE will only show the low-level assembly code that the computer will actually execute. Eventually, if you're serious about audio programming, you'll learn how to read such assembly code.

Knowing how to use the call stack is very important when debugging, because it gives you some context for when and how the function that crashed was invoked. Often, the bug is not in the function that crashed but it happened earlier on in the program. The crash is the effect — the puzzle is finding the cause.

You've already seen the debug output pane where the DBG messages show up, but there are other useful panes in the debugger window, in particular the **variables view** that shows the values of all variables from the current context.

In Xcode, the variables view is revealed using the small buttons at the bottom of the window (next to Filter). In Visual Studio it's the pane in the bottom-left corner of the debugger screen, using the **Autos** or **Locals** tabs.

In the above screenshots the variables view shows two entries: **this** and **delayInSamples**:

- The **this** entry refers to the class instance that we're in, here the `DelayLine` object. Expand it to have a look at the current values of all the member variables.
- This pane also shows the local variables, all the variables that only exist in this function. Here, **delayInSamples** has a value that's indeed less than `1000.0f`, so that explains why the assertion was triggered.

The debugger is handy because it gives you a snapshot of the state of your program at the moment it crashed. With the call stack you can find out which function crashed and where it was being used. From the variables view you can learn what the current values of the variables are. This is often enough to find out why the program crashed, although sometimes it's merely the starting point in your investigation.

Exit the debugger by pressing the **stop button** in the toolbar. This will terminate `AudioPluginHost` and the plug-in. Put the `jassert` line back to what it was (use `1.0f` instead of `1000.0f`).

Not all bugs are crashes. Sometimes stuff simply doesn't work correctly and you have to figure out why not. Let's introduce a new bug into our program.

In **Parameters.cpp**, in `update`, comment out the following line:

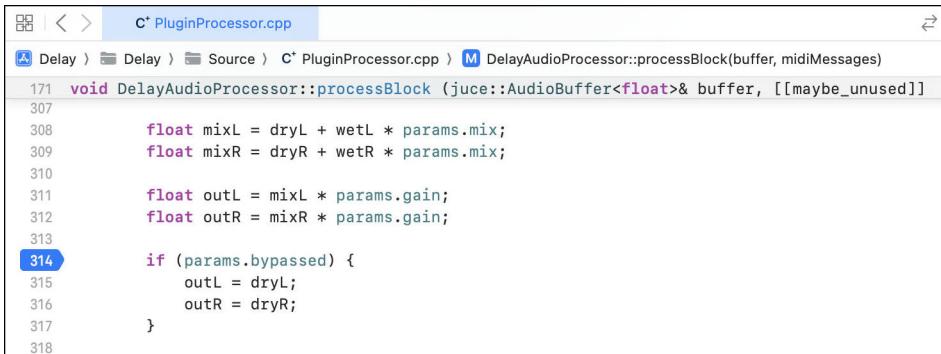
```
// bypassed = bypassParam->get();
```

We'll pretend that we forgot to read the value of the bypass parameter into the `bypassed` variable. Now when you run the plug-in and toggle the **Bypass** button, nothing happens. If you encounter such a bug in your program, how do you track it down?

A good tool for this is the **breakpoint**. You may have a hunch that the bypass mode doesn't work because perhaps something is wrong with the parameter. So you go to **PluginProcessor.cpp**, scroll to `processBlock`, and put a breakpoint on the following line:

```
if (params.bypassed) {
```

In Xcode, you create a breakpoint by clicking on the line number. This places a blue arrow in the margin. Click again to disable the breakpoint, which makes it light blue. To completely remove the breakpoint, drag it out of the margin.



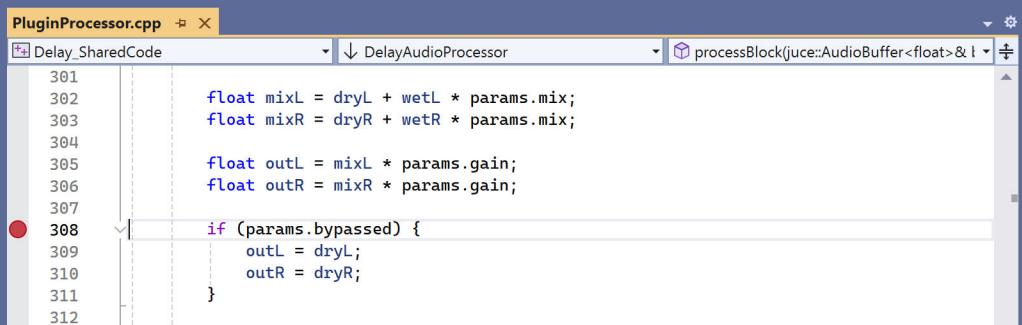
The screenshot shows a code editor window for `C* PluginProcessor.cpp`. The file is part of a project named `Delay`. The code is a C++ implementation of a plugin processor. A blue arrow-shaped breakpoint is visible in the margin next to the line number 314. The line contains an `if` statement checking if `params.bypassed` is true. The code snippet is as follows:

```
171 void DelayAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer, [[maybe_unused]]
```

- 307
- 308 float mixL = dryL + wetL * params.mix;
- 309 float mixR = dryR + wetR * params.mix;
- 310
- 311 float outL = mixL * params.gain;
- 312 float outR = mixR * params.gain;
- 313
- 314 if (params.bypassed) {**
- 315 outL = dryL;
- 316 outR = dryR;
- 317 }
- 318

A breakpoint in Xcode

In Visual Studio, click in the margin next to the line number. This places a red circle in the margin. Click the red circle a second time to remove the breakpoint. There is also a right-click menu that lets you disable the breakpoint temporarily.



The screenshot shows a code editor window for 'PluginProcessor.cpp'. The file is named 'Delay_SharedCode' and contains C++ code for a 'DelayAudioProcessor'. A red circular breakpoint icon is visible on the left margin next to line 308. The code on line 308 is an if-statement that checks if 'params.bypassed' is true, and if so, sets 'outL' and 'outR' to 'dryL' and 'dryR' respectively.

```

301     float mixL = dryL + wetL * params.mix;
302     float mixR = dryR + wetR * params.mix;
303
304     float outL = mixL * params.gain;
305     float outR = mixR * params.gain;
306
307
308 if (params.bypassed) {
309     outL = dryL;
310     outR = dryR;
311 }
312

```

A breakpoint in Visual Studio

The cool thing is that you can create breakpoints while the plug-in is already running inside the debugger.

Try this out: First remove the breakpoint and then run the plug-in and play some audio through it. This should work as usual. While the plug-in is active, switch back to your IDE and put the breakpoint on the `if (params.bypassed)` line.

You'll find that the plug-in immediately is paused and the debugger becomes active. This happens because the next time JUCE performed `processBlock`, it saw the breakpoint and jumped into the debugger.

Note: Putting a breakpoint in the audio processing code can cause your computer to stop playing all audio, even if you were playing other sounds or music in the background. This is normal since you're interrupting the audio thread. The sound will continue once the plug-in resumes running.

Again, the call stack shows which functions were called to get to `processBlock`, and the variables view shows all member variables and local variables.

In the variables view, expand **this**, and then expand **params**, to verify that **bypassed** is currently **false**. This is as expected, as the plug-in is not bypassed yet.

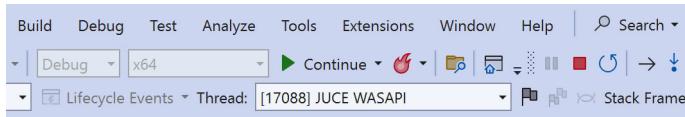
Disable the breakpoint by clicking the arrow (Xcode) or by hovering over it to get a popup with a Disable option (Visual Studio).

The host program and the plug-in inside it are currently suspended in the debugger. To get them to resume running, you'll need to click the **Continue** button. In Xcode this is a button inside the debugger toolbar.



The Continue button in Xcode

In Visual Studio this is the green run button in the toolbar that says **Continue**.



The Continue button in Visual Studio

After pressing this **Continue** button, the host will resume running the plug-in. This is why it's important that you disabled the breakpoint, otherwise you'd immediately drop back into the debugger.

Toggle the **Bypass** button and then activate the breakpoint again. The plug-in gets suspended and the debugger takes over. In the variables view you would have expected to see that the `params.bypassed` variable is now `true` but it's still `false`.

Hypothesis confirmed: We must have “forgotten” to read the plug-in parameter into the bypassed variable. Stop the debugger, fix the code, and try again. `params.bypassed` should have the correct value after toggling the **Bypass** button.

Of course, this was only a simple example of how to use the debugger, but I hope it gives you some idea of how we approach finding and fixing bugs in our programs. It's worth learning more about how your IDE's debugger works!

One of the biggest skills you need to obtain as a software developer is how to problem solve. I've observed many times that new developers, when confronted with a bug, will randomly try things hoping that one of these stabs in the dark will somehow fix the problem.

It's better to think the situation through and check any assumptions you're making that might not actually be true. The debugger can help with that.

Now that the plug-in is finished, in the next chapter we'll talk about how to get it in the hands of users.

19: Releasing your plug-in

The plug-in is finished but you can't upload it to your website or send it to other people just yet. There's more work to be done!

Making a release build

So far you've been making debug builds of the plug-in. A debug build includes a lot of extra information such as special symbols that allow the debugger to pinpoint exactly where in the source code a crash happened.

Debug builds are very useful during development because they make it easier to track down and fix errors. But you don't want to send a debug build to your users. Instead, you'll need to make a release build.

Some of the differences between debug and release builds:

- A debug build is much larger because of all the extra information that gets included. The debug build for the VST3 version of the Delay plug-in on my Mac is about 30 MB, while the release build is only 6 MB. That's a big difference!
- Debug builds have optimizations disabled, since optimizations can produce machine code that is harder to understand and debug. No optimizations makes the debug build a lot slower than a release build.
- Debug builds have assertions enabled to perform additional safety checks. Likewise, we've added `protectYourEars`, which only is enabled if `JUCE_DEBUG` is set. In release builds, assertions are stripped out of the program, so all the safety checks are gone.
- Some compilers try to be helpful and set variables to zero in debug builds if you forget to initialize them. However, it won't do this for release builds and uninitialized variables will get arbitrary values. Switching from debug to release can actually highlight bugs you didn't know were there!

- Often developers will strip all symbols from release builds, not just the debug symbols, to prevent crackers and other peeping toms from easily being able to reverse engineer the machine code. This is a form of obfuscation, hiding useful information to make it harder to interpret how the plug-in works.

In **Projucer** the exporters have Debug and Release configurations. Go into the **Debug** configuration for your IDE's exporter and notice that the **Debug Mode** option is enabled and **Optimisation** is disabled. For the **Release** configuration, it's the other way around: debug mode is turned off and optimizations are turned on.

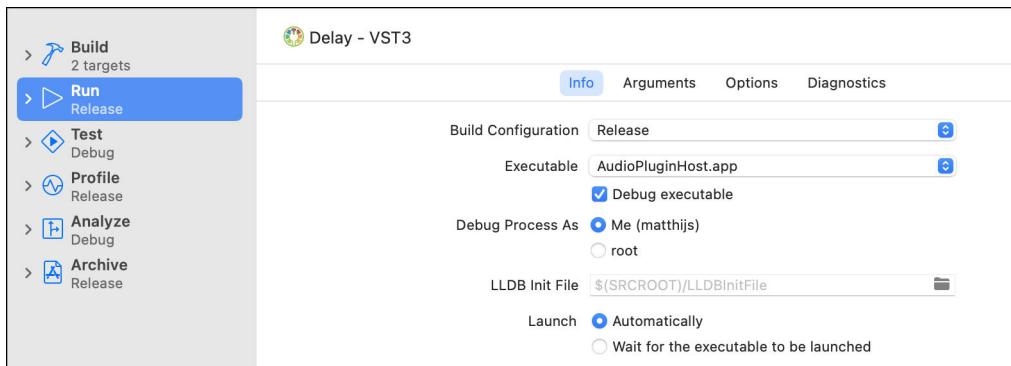
To make a release build in Xcode, first go to **Projucer** and under the **Xcode (macOS)** exporter, select the **Release** configuration. Scroll down to the option **macOS Architecture**. Set this to **Standard 64-bit**. That way Xcode will build a universal binary that works on both Intel and ARM-based Macs.

Additionally, set the **Custom Xcode Flags** option to:

```
GCC_GENERATE_DEBUGGING_SYMBOLS=YES, DEBUG_INFORMATION_FORMAT=dwarf-with-dsym
```

This will make Xcode output a **.dSYM** file containing the archived debug symbols. This is useful for when a user sends a crash report. Since release builds do not contain debug symbols, such a crash report is useless unless you can “symbolicate” it, which adds the debug symbols back in. That’s what the **dSYM** file is for.

In Xcode, click the **Delay - VST3** target at the top of the window and choose **Edit Scheme...** from the popup. In the window that appears, select **Run** on the left. Go to the **Info** tab and for **Build Configuration** choose **Release**. Now when you build the project, it will use the release configuration.



Enabling release builds in Xcode

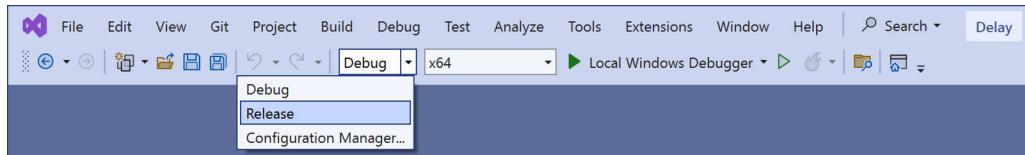
After building, the plug-in can be found in the **Builds/MacOSX/build/Release** folder. On Mac the release build is automatically copied to the system's plug-in folder, overwriting any previous debug build that was there.

You'll want to store the dSYM file along with the VST3 or AU file in a safe place, so that you can symbolicate any crash reports that you receive from users.

Note: Another method for making a release build is to use the Xcode menu option **Product > Archive**. However, this wasn't really designed for JUCE plug-ins and is awkward to use. It also doesn't include the dSYM file in the archive.

To make a release build with Visual Studio, first go to **Projucer** and under the **Visual Studio 2022** exporter, select the **Release** configuration. Scroll down to the option **Runtime Library**. Set this to **Use static runtime**. The default option, Use DLL runtime, requires that the user installs a separate package to run the plug-in, the Microsoft Visual C++ Redistributable, which is not very user-friendly.

In Visual Studio, the toolbar has a box that says **Debug**. Click this and choose **Release** from the popup. Then build the project again (**Ctrl+B**).



Enabling release builds in Visual Studio

The plug-in can now be found in the **Builds/VisualStudio2022/x64/Release/VST3** folder. On Windows the release build is not automatically copied to the system's plug-in folder, so if you want to try out the release build you'll need to copy it over manually.

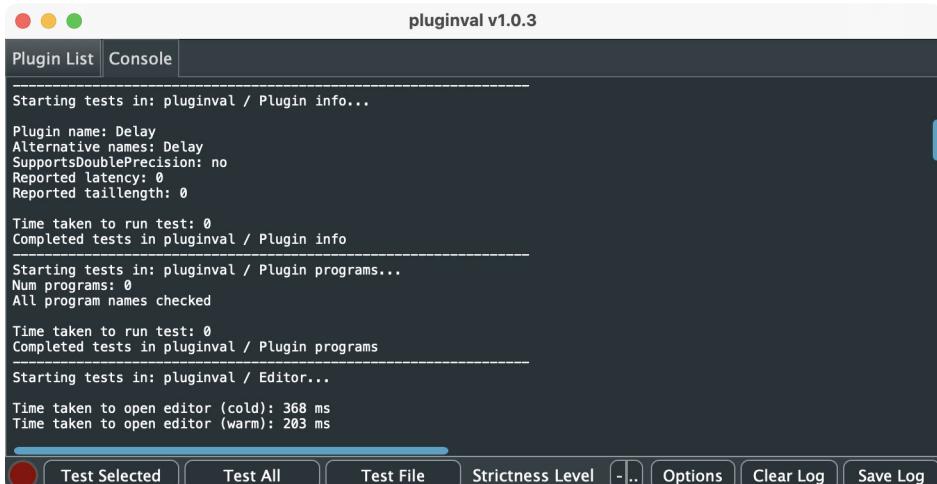
You will want to keep everything that's inside this **Release** folder, in particular the **Delay.pdb** file, which stands for Program Debug Database. This contains the debug symbols Visual Studio will use to relate any crash reports to your original source code. Without this **.pdb** file it will be nearly impossible to determine where the crash happened.

Note: If the plug-in suddenly behaves weird or crashes when you try the release build, chances are that there are uninitialized variables in your program or there is a timing problem. The latter often happens when multiple threads are involved. Because release builds are made with optimizations enabled, the timing of events is different than in a debug build, and any mistakes in the threading code that may previously have been hidden, will suddenly show up in the release build.

Testing with pluginval

Naturally, before releasing any new product to the public, you'll need to give it a proper round of testing. No matter how good a programmer you are, there will probably be bugs. It is better if it's you that finds these bugs and not your users.

The first thing you should do is run the plug-in through [pluginval](#)³⁹. This is a handy tool that performs a suite of tests to make sure your plug-in behaves properly. The document [Testing plugins with pluginval](#)⁴⁰ explains how to install pluginval and how to use it.



Testing the plug-in with pluginval

Give pluginval a go on the Delay plug-in!

³⁹<https://github.com/Tracktion/pluginval>

⁴⁰<https://github.com/Tracktion/pluginval/blob/develop/docs/Testing%20plugins%20with%20pluginval.md>

Specifically for Audio Units there is also `auval`, which performs many of the same tests as `pluginval`. This utility is already installed on your Mac. You run it from the Terminal.

```
$ auval -v aufx Dlay Manu
```

The first argument after `auval -v` is the type of the plug-in. `aufx` means it's an effect. For a synth you'd use `aumu`. The next argument is the 4-character code for the plug-in, `Dlay`. The last argument is the 4-character code for the manufacturer, `Manu`. These values are defined in Projucer in the main project settings, under **Plugin Manufacturer Code** and **Plugin Code**.

`auval` should finish with the message `AU VALIDATION SUCCEEDED`. If not, you've got some debugging to do.

Testing on different hosts

A previous chapter mentioned unit tests, which are automated tests that check each building block of your plug-in in isolation. Unit tests are great but they won't catch problems that may happen because of disagreements between the plug-in and the host program.

Tools like `pluginval` and `auval` help to minimize the risk of running into such issues, but you should still test your plug-in in a DAW. Or even better, in multiple DAWs. Just because the plug-in works OK in your favorite DAW doesn't mean it will work in all of them — they all have their own quirks. Ideally, you should test on all the major DAWs, both on Mac and Windows.

Here's a checklist of things to test:

- Try the plug-in at different sample rates: 44100, 48000, 96000, etc.
- Try the plug-in with different buffer sizes, from smallest to largest.
- Try the plug-in on mono and stereo tracks.
- Test that automation of all parameters works. The controls should change values when you play back recorded automation.
- Make sure the plug-in's state is saved and restored correctly when the project is closed and reopened.

- Put the plug-in on multiple tracks at the same time.
- Add / remove the plug-in repeatedly while sound is and isn't playing.
- Verify that exporting / bouncing / offline rendering works.

Make sure to do these tests with the release build of your plug-in, not only the debug build!

Tip: Make a number of test projects in the DAW for different parts of the plug-in to test. Create a track with an audio file on it, put your plug-in on the track with a particular set of parameter settings, and bounce the project to a WAV file.

When you've made changes to the plug-in and you want to verify it still works OK, bounce the project again and compare the WAV files by listening to them. Or use your DAW to subtract the second WAV from the first one, to see if they null out. If not, then something might be wrong with the plug-in.

Performance testing

Users like their plug-ins to not eat up too much CPU time. After all, the less CPU usage, the more plug-ins they can run at the same time! As developers we don't have to obsess over using as little CPU as possible — tempting as it is — but at the very least it's good to get an idea of how efficient or inefficient the plug-in is.

For a rough indication of the CPU usage of your plug-in, you can look at the Activity Monitor (Mac) or Task Manager (Windows). For more detailed results, use the profiling tools that come with your IDE, such as Xcode's Instruments or Visual Studio's Diagnostic Tools.

The downside of these approaches is that they measure the CPU usage of the entire host application, not just of your plug-in. Make sure to disable all other plug-ins, then attach the profiling tool to the host application and compare how much CPU it uses with the plug-in enabled vs. disabled.

Tip: Always do these tests with the plug-in compiled in release mode, not debug mode. It's easy to forget to switch to a release build, but performance measurements on a debug build are meaningless and you may end up optimizing the wrong thing.

The DAW may have its own CPU meter. Logic Pro has a CPU meter in the **Custom** mode of the time display at the top. REAPER has a detailed meter that you can access by choosing **Performance Meter** from the **View** menu. The **FX** window in REAPER also shows CPU usage numbers: the first is the CPU usage of the selected plug-in, the second is for the entire plug-in chain. Other DAWs will have their own ways to measure performance.

Make sure to test this while the plug-in is actually producing sound — in some hosts if the track isn't playing, the plug-in may be paused.

You can also add timing code to the plug-in itself to measure how long each call to `processBlock` takes, for example using `std::chrono::high_resolution_clock`. The question is how to report this elapsed time, since printing to the debug console from `processBlock` will affect the timing. One solution is to add up the measurements over the course of one second, then print out how much time was taken up during that second. JUCE has a class for this, `AudioProcessLoadMeasurer`.

For automated performance testing, create your own lightweight host program that loads the plug-in and then calls `processBlock` in a continuous loop for a certain amount of time. Creating such a test host is easy enough with JUCE since you can borrow the source code from `AudioPluginHost`.

Running the plug-in in a loop is similar to what a DAW does when it bounces the project in offline mode, and it lets you measure exactly how efficient your code is. If you make your test program produce 100 seconds of audio and it takes the loop 1 second to do this, then the plug-in runs at $100\times$ real-time speed and you can expect it to use only 1% CPU time in real-world usage.

Another useful test is the stress test: In your DAW, put the plug-in on 10 tracks at the same time so that all the plug-in instances will be rendering audio simultaneously. Does this use $10\times$ the CPU time or is it suddenly much worse? What if you use 20 tracks, or more?

A downside of doing the performance test in a “how fast can this run?” loop is that your plug-in may benefit from all its data being in the CPU cache. But when you use ten or more instances at the same time, they may start getting cache misses, which is a more realistic usage pattern.

There are special applications that can be used to analyze and benchmark plug-ins, such as [DSP Testbench⁴¹](#) and [Plugindocor⁴²](#).

⁴¹<https://github.com/AndrewJJ/DSP-Testbench>

⁴²<https://ddmf.eu/plugindocor/>

Improving real-time performance

What if your plug-in is too slow and burns more CPU time than you'd like? If your plug-in is a CPU hog, you'll probably want to try optimizing it. Here are some suggestions:

- Before you do anything else, use your IDE's time profiler to determine what part of the plug-in is responsible for making it slow. Don't guess, measure. Create a test environment that runs your `processBlock` in a loop and use a time profiler tool to measure how long everything takes.
- If the profiler shows you're calling a lot of expensive mathematical functions, you might consider replacing these with approximations or look-up tables. There are several fast math approximation libraries available.
- You may have a memory access problem. When a modern CPU loads a variable from RAM, it actually grabs a larger chunk of memory to put into the cache. Key to getting good performance is taking advantage of this caching mechanism because cache memory is much faster than regular RAM. Usually this means being smart about how you're setting up your loops. Code that has a lot of "cache misses" will be unnecessarily slow.
- Use SIMD, which stands for Single Instruction Multiple Data. Your CPU comes with a set of instructions that can perform multiple floating-point operations at once. On Intel chips these are SSE/AVX instructions, on ARM chips it is known as Neon. These so-called vector instructions let you perform operations such as addition and multiplication on four or eight floats at once, which gives a 4× to 8× speed up.

There is some overhead involved in using SIMD, since data needs to be copied into and out of these special registers, but it's the premier way to speed up floating-point stuff. You can use SIMD from C++ using so-called compiler intrinsics, which are functions with names like `_mm256_add_ps`. JUCE has `SIMDRegister` and `FloatVectorOperations` classes to make this easier.

Vectorizing your audio code using SIMD is great for performance, but it does require you to rethink how the code is structured, as now operations need to be done in groups. You can vectorize code "in time", meaning that instead of processing just the current sample, you will process the next four samples at once. This won't work for a filter, for example, since the next sample depends on the previous one, but you could vectorize this by processing the left and right channels together.

To enable SIMD, you'll need to switch to block-based processing rather than sample-by-sample processing, as was explained at the end of chapter 14. Another benefit of this approach is that it can make better use of the cache, since the code is not jumping between different tasks all the time.

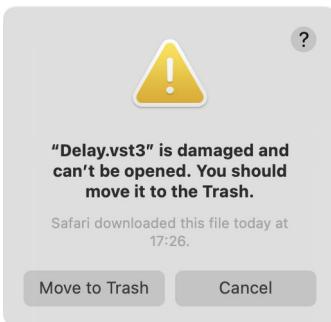
The good news is that modern C++ compilers already can apply auto-vectorization if a loop is simple enough, so you might not even have to write the SIMD code by hand.

Tip: If you care about performance, learn to read assembly code (also known as machine code). Your IDE can show the assembly it generates from the C++ code. By inspecting the assembly, you can already see if a loop was auto-vectorized by the compiler, and if not, you can experiment to see how to rewrite the loop so that the compiler does the right thing. You should still always measure the performance because longer assembly code doesn't mean it runs slower, but getting some insight into what the compiler does under the hood is really useful for writing high performance code.

Signing the plug-in (Mac)

The latest versions of macOS will only run software that is signed using a special certificate that identifies the developer of the software. Until now we have not done anything to set this up, which means the plug-in was signed using a so-called adhoc certificate. That allows you to run the plug-in on your own machine but not on other computers.

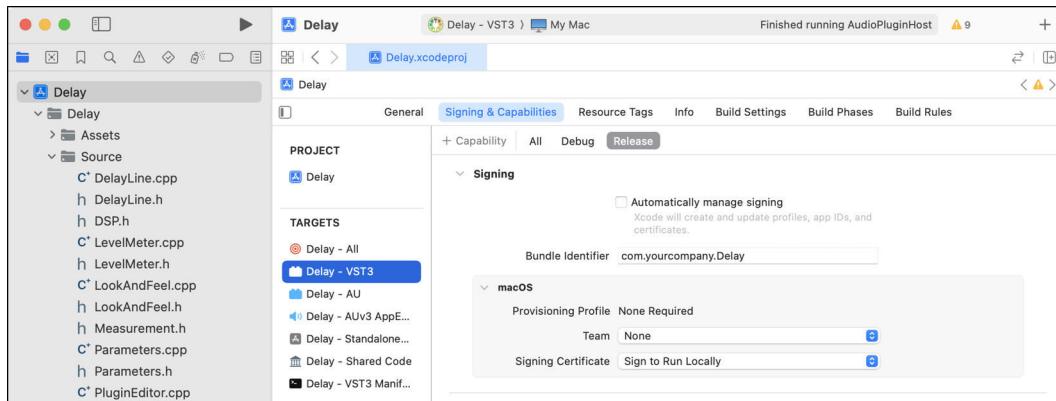
If you give the **Delay.vst3** or **Delay.component** files to a friend and they try to load the plug-in on their Mac, they will get the following error message:



macOS will not let users load unsigned plug-ins

To stop this from happening, the plug-in will need to be signed. This requires you to become a member of the Apple Developer Program, which costs \$99 per year.

You can find out what signing certificate Xcode used by clicking on the **Delay** project at the top of the **Project Navigator**. Then under **TARGETS** select **Delay - VST3**. Next, go to the **Signing & Capabilities** tab.



Xcode is signing the plug-in to run locally

If under **Signing Certificate** it says **Sign to Run Locally**, the plug-in is only allowed to run on your own computer but not on anyone else's computer.

Enrolling in the Apple Developer Program

You'll need to become a member of the paid Apple Developer Program in order to distribute your plug-ins — or any other software you write — whether that's directly from your own website or from the Mac App Store or iOS App Store.

Technically speaking, you can distribute apps and plug-ins from your website without being a member of the paid Developer Program, but users will get that scary error message and will have to jump through hoops to get your software to run. That doesn't give a very professional impression.

To join the Apple Developer Program, go to developer.apple.com/programs⁴³ and click **Enroll** in the top-right corner.

You can sign up as an Individual or as an Organization. The latter option is only possible if you own or work for a company that is its own legal entity. You'll have to

⁴³<https://developer.apple.com/programs>

provide Apple with a D-U-N-S Number for your company. The website explains how to get such a number.

If you don't have a company or if it's a sole proprietorship, you'll need to sign up as an Individual. In that case the plug-in will be signed with your personal name instead of a company name.

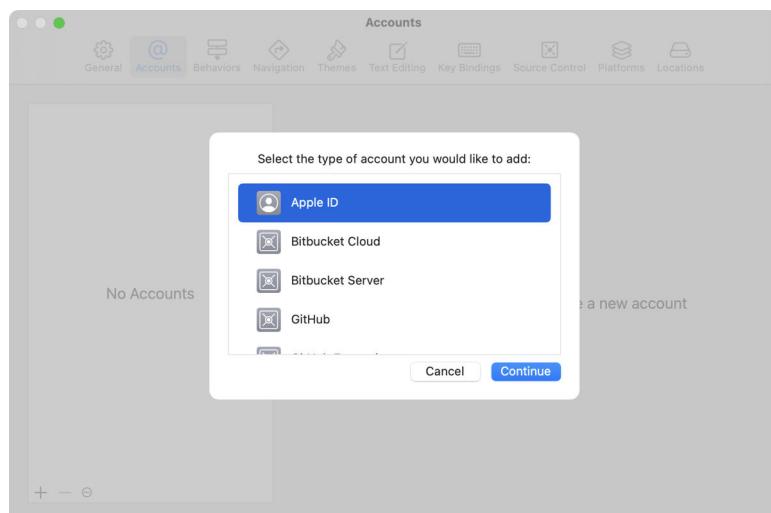
Both programs cost \$99 per year. This includes two free tickets per year for code level support from Apple engineers, which can be a godsend in case you get stuck on some problem and can't find the solution anywhere. (Although they probably won't help with JUCE questions.)

Signing up to the Apple Developer Program requires an Apple ID. You can use your personal Apple ID but it makes sense to create a new one specifically for this purpose.

Codesigning

Once your Apple Developer Program membership is in the bag, you can set up Xcode and Projucer to automatically sign the compiled plug-in binaries. Be warned: Sometimes the signing process simply won't cooperate and you'll waste a bunch of time getting it to work. Unfortunately, that's just how it is...

In Xcode, open the **Settings** screen (called Preferences in older Xcode versions) and switch to the **Accounts** tab. Click the + icon at the bottom and choose **Apple ID**.



Adding a new account to Xcode

Next, enter your Apple ID and password. You may be asked to turn on two-factor authentication (requires sending a text message to your phone). After a short while, the panel shows your account information.

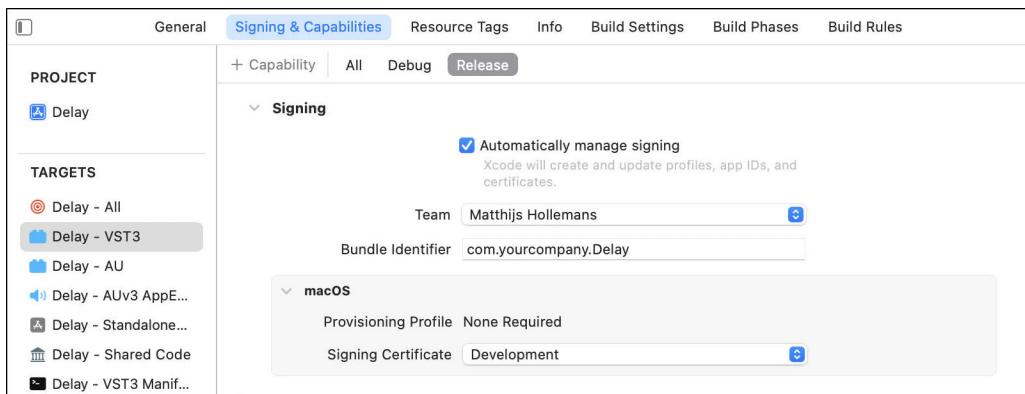
You'll need to find the **Organization ID** for the account, as this Projucer needs this. Unfortunately, there is no easy place to get this from the Xcode accounts panel.

Open **Keychain Access** (can be found in **Applications/Utilities**) and go to **My Certificates**. There should be an entry for **Apple Development** followed by your Apple ID email address. Double click this entry and a window pops up. Find the **Organizational Unit**. This is a 10-character code such as UEZ5249C7M. Make a note of this and close the window.

In **Projucer**, in the **Exporters, Xcode (macOS)** settings, fill in that 10-character code under **Development Team ID** (near the bottom).

In Xcode, click the **Delay** project at the top of the **Project Navigator** to bring up the project settings again. Under **TARGETS** select **Delay - VST3**. In the **Signing & Capabilities** tab it will have enabled **Automatically manage signing**.

The **Team** should list your account name and the **Signing Certificate** says **Development** (instead of Sign to Run Locally).



Xcode will sign the plug-in

In this screenshot, the team name is my personal name because I'm signed up to the Apple Developer Program as an individual, not as a company.

To see the signature that the plug-in is signed with, run the following command from the **Terminal**:

```
codesign -dvv ~/Library/Audio/Plug-Ins/VST3/Delay.vst3
```

Look for the line `Signature=adhoc`. If this is present, the plug-in has been signed to run on your local computer only. If instead there is a line starting with `Authority=`, a valid signing certificate was used. It's always smart to double-check that the plug-in really has been signed properly!

After rebuilding the plug-in, running the `codesign -dvv` command on my computer prints the following lines (among several other lines you can ignore):

```
Authority=Apple Development: Matthijs Hollemans (UEZ5249C7M)
Authority=Apple Worldwide Developer Relations Certification Authority
Authority=Apple Root CA
```

The text `Apple Development:` means that the plug-in was signed for development. This still only allows you to run the plug-in on any Macs that have this certificate installed, but we're a step closer already.

Note: I found that sometimes the plug-in was signed as “adhoc” even though Xcode has the correct Signing Certificate selected. The solution was to simply press **Command+B** again. Afterwards, the plug-in was signed OK. So, you may need to rebuild an extra time to apply the certificate.

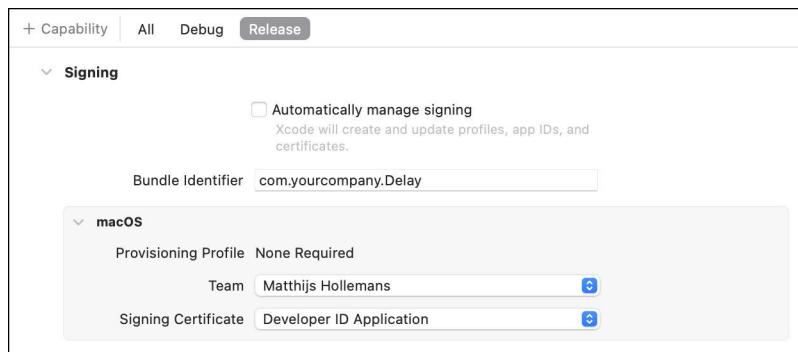
To distribute the plug-in to other people, the Development certificate is not the right choice. It's a little confusing but there are three different distribution certificates to choose from:

- **Developer ID Application.** This certificate is used to sign the plug-in.
- **Developer ID Installer.** If you want to distribute a `.pkg` installer through your own website, you will need to sign the installer with a certificate of this type, in addition to signing the plug-in itself.
- **Mac Installer Distribution Certificate.** This is for distributing through the Mac App Store. You can only distribute AUv3 plug-ins through the Mac App Store, not AU or VST3 or other formats.

It's easy to create the required certificates from within Xcode in the **Settings... > Accounts** panel. Select your Apple ID on the left and click **Manage Certificates...** In the popup that appears, click the + button at the bottom and choose the certificate type you need, for example **Developer ID Application**. This will add the new certificate to your computer.

Go back to the **Signing & Capabilities** settings for the **Delay - VST3** target, and switch to the **Release** tab. Uncheck the **Automatically manage signing** box. Next, choose your account for **Team** and the **Developer ID Application** signing certificate.

When you make a release build now, it will automatically sign the **Delay.vst3** file in the **Release** folder using the Developer ID Application certificate.



Signing with the Developer ID Application certificate

Build the plug-in again. When you check with `codesign -dvv` it should indicate the plug-in has been signed properly. The first `Authority=` line should say `Developer ID Application` instead of `Apple Development`, like so:

```
Authority=Developer ID Application: Matthijs Hollemans (UEZ5249C7M)
```

Excellent, other people will now be able to use the plug-in without problems on their Macs.

If you want to make an installer, [this tutorial⁴⁴](#) has good instructions. Just remember that a **.pkg** installer needs to be signed too, with a Developer ID Installer certificate. It also needs to be notarized, which lets Apple verify your package is not malware. There is a [great blog post by Sudara⁴⁵](#) that explains how to get notarization to work.

⁴⁴https://docs.juce.com/master/tutorial_app_plugin_packaging.html

⁴⁵<https://melatonin.dev/blog/how-to-code-sign-and-notarize-macos-audio-plugins-in-ci/>

Signing the installer (Windows)

Windows may give a warning if you attempt to run software that does not have a digital signature. Unlike macOS, however, it allows users to continue using the software.

As of this moment, there is no need to sign the plug-in itself with a certificate. However, if you plan on providing an installer for Windows, users may be confronted with a nasty SmartScreen malware warning when they launch the installer. This doesn't come across as very professional and may even scare off potential users, so you might want to consider signing the installer.

There are two types of certificates:

- OV (Organization Validation)
- EV (Extended Validation)

The OV certificate is cheaper but Windows can still show malware warnings even for software that is signed with an OV certificate. Only after your software has gained a certain amount of reputation with Microsoft's malware checkers will those warnings disappear for new users. Not great. There is a way to manually [submit your software⁴⁶](#) to Microsoft for malware scanning, which apparently helps speed up the reputation process.

The EV certificate sidesteps these issues and completely avoids the Windows malware warnings. However, it's a lot more expensive and you may need a registered business to even be able to buy one. The certificate will need to be renewed every few years for \$++. Microsoft does not sell these certificates themselves, so you'll need to buy from a third-party vendor. The whole thing is kind of a scam, really.

The signing process is rather tricky and I cannot cover it all in this book. My suggestion is to ignore signing on Windows for now and come back to it when you're ready to start selling plug-ins. Or don't have an installer and simply put the VST3 in a ZIP file and explain to the user where to copy it.

To learn more about making an installer for Windows, [see this tutorial⁴⁷](#).

⁴⁶<https://learn.microsoft.com/en-us/microsoft-365/security/defender/submission-guide>

⁴⁷https://docs.juce.com/master/tutorial_app_plugin_packaging.html

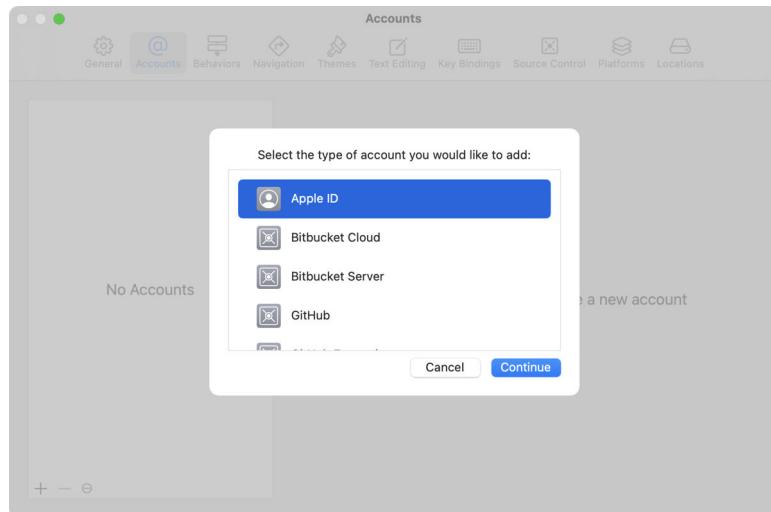
Building for iOS

To make an iOS version of the plug-in, you need a Mac with Xcode. Membership in the paid Apple Developer Program is optional but recommended.

For testing on your own devices, you can use your existing Apple ID to codesign the plug-in. The downside is that the plug-in will only function for a limited time and only on your own device. After that, you'll have to remove it from the device and use Xcode again to install it. Kind of annoying but good enough to play around with this for free.

To distribute the plug-in to other people on the iOS App Store, you will need to join the paid Apple Developer Program, which costs \$99 per year. This is the same Developer Program as for macOS, so you don't need to join twice.

If you haven't yet signed into Xcode using your Apple ID, follow these steps. In Xcode, open the **Settings** screen (called Preferences in older Xcode versions) and switch to the **Accounts** tab. Click the + icon at the bottom and choose **Apple ID**.



Adding a new account to Xcode

Next, enter your Apple ID and password. You may be asked to turn on two-factor authentication (requires sending a text message to your phone). After a short while, the panel shows your account information.

For the free account, the team name is followed by “(Personal Team)”. That means it’s not a full Developer Program account, but only something you can use on your own devices.

Making an iOS build is fairly straightforward, as JUCE already does all the work for you. The major difference with plug-ins on Mac or Windows is that on iOS the plug-in always lives inside an app.

VST3 and AU plug-ins are shared libraries or DLLs that are loaded into memory by the host. On iOS this works somewhat differently. The plug-in exists as an app extension that is embedded inside a regular app. When users download a plug-in from the App Store, they are really downloading an app that contains the plug-in as an app extension.

The format of this plug-in is AUv3, which is a slightly more modern version of the Audio Unit format used on the Mac. When the app is installed, the AUv3 is automatically registered with the system and can be used by host apps such as GarageBand. By the way, AUv3 can also be used on macOS and is delivered the same way: embedded inside an app.

In **Projucer**, go to the project settings screen (gear icon at the top). Under **Plugin Formats**, make sure both **AUv3** and **Standalone** are enabled.

Next, go to the **Exporters** pane and add a new exporter for **Xcode (iOS)** if you didn’t add this previously. In the settings for this exporter, scroll all the way down and under **Development Team ID** fill in your Apple Developer account’s 10-letter identifier. This is necessary for code signing to work when installing the app to your device. You can find this identifier in Keychain Access.

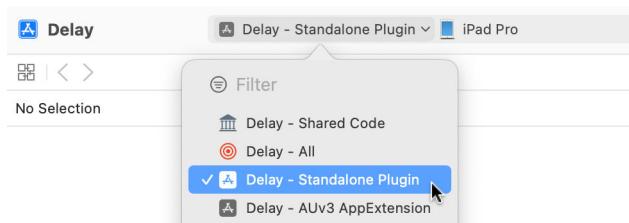
Open **Keychain Access** (can be found in **Applications/Utilities**) and go to **My Certificates**. There should be an entry for **Apple Development** followed by your Apple ID email address. Double click this entry and a window pops up. Find the **Organizational Unit**. This is a 10-character code such as UEZ5249C7M. Copy this into Projucer.

The iOS exporter settings also have entries for icons, screen orientation, and other typical iOS app configuration options. Scroll to the **Microphone Access** option and set it to **Enabled**. Without this, the plug-in’s app will crash right after launching for trying to use the microphone without having asked permission first.

At the top of the Projucer window, under **Selected exporter**, choose **Xcode (iOS)** and press the export button. Now the iOS version of the plug-in will open in Xcode.

Connect your iOS device to the Mac. On your device, open the **Settings** app and go to the **Privacy & Security** options. At the bottom is an option for **Developer Mode**. Turn this on. This may require restarting the device.

In the Xcode scheme picker at the top of the window, select your device. The target to build is **Delay - Standalone Plugin**. Since AUV3 are always delivered inside an app, you need to build the standalone app and install it on your device, just as if someone downloaded the app from the App Store. (If you don't see this target, try restarting Xcode or export from Projucer again.)



Selecting the target and device for building on iOS

Press the run button to build and install the app. If all goes well, the app is running on your device and showing the plug-in's UI in full screen. On first launch it asks for access to the microphone. It's smart to deny this to avoid feedback.

The app is very barebones, it doesn't even have an icon. For a real plug-in that you plan to release on the App Store, you need to do a bit of work to make it more appealing, by writing a customized standalone app to host the plug-in.

Try using the plug-in in a DAW such as GarageBand. In GarageBand's track setting panel, go to the **PLUG-INS & EQ** section and tap the arrow to bring up the list of plugins for this track. Click **Edit** and then tap on an empty effect slot. In the popup that appears tap on **Audio Unit Extensions**. In the list should be an icon for **Delay**. Select that and you can use the Delay plug-in inside GarageBand, sweet!

Note: If you get the error message “Unable to install Delay” when building the app, it means you didn’t set up the code signing options correctly in Projucer.

Licensing and legal stuff

Before you can send the finished plug-in out into the world there are some legal hoops to jump through.

In this book we've been using JUCE under the Educational or Personal plans. We enabled GPLv3 mode, which allowed us to hide the JUCE splash screen. This is fine for just using the plug-in on your own computer, but when it comes to distributing the plug-in to others, these are your options:

- License your plug-in under the terms of [GPLv3⁴⁸](#) for JUCE 7 or the [AGPLv3⁴⁹](#) for JUCE 8. This means making your plug-in open source. If you're a fan of open source and think the GPL is an acceptable license, then this is a great option. The awesome [Surge XT synthesizer⁵⁰](#), for example, uses JUCE and is GPLv3.
- Use JUCE for free using a Personal license. The plug-in is required to show a "Made with JUCE" splash screen when it starts up. (JUCE 7 only, no splash screen needed with JUCE 8.)
- For commercial products you must pay a monthly license fee, although there is an option for a one-time fee. See the [available plans⁵¹](#) on the JUCE website.

Besides the JUCE license there are other legalities to take care of. Steinberg, who owns the VST3 SDK and trademark, requires that developers sign a licensing agreement to distribute commercial VST3 plug-ins. The VST3 SDK is included in JUCE but this does not automatically give you the right to distribute VST3 plug-ins. Once you decide to make some money from your creations, you'll need to [sign the agreement with Steinberg⁵²](#).

No such agreement is needed for Audio Units. It is already covered by the agreement you sign with Apple to be part of their developer program. However, there is a separate licensing agreement for the use of Audio Unit logos on your website.

JUCE can also build AAX plug-ins but only after you have installed the Pro Tools SDK. Creating and distributing AAX plug-ins for Pro Tools requires entering into a legal agreement with Avid. You can find out more at [developer.avid.com⁵³](#).

⁴⁸<https://www.gnu.org/licenses/gpl-3.0.html>

⁴⁹<https://www.gnu.org/licenses/agpl-3.0.en.html>

⁵⁰<https://surge-synthesizer.github.io>

⁵¹<https://juce.com/get-juce>

⁵²https://github.com/steinbergmedia/vst3sdk/blob/master/VST3_License_Agreement.pdf

⁵³<https://developer.avid.com>

20: Where to go from here

Congrats, you've made it all the way to the end of the book! Thanks for joining me on this journey — I hope you had fun writing the delay plug-in. In this chapter I'm going to give some ideas for things you can try next, and resources you can use to learn more about audio programming.

Ideas for new features

Here are some suggestions for new things you could add to the Delay plug-in:

- Put the low-cut and high-cut filters after the delay line output rather than in the feedback path, so that the first echo is also filtered.
- Have separate knobs for the delay time of the left and right channels. Similarly, you could use different amounts of feedback for left and right.
- Add new effects into the feedback path, such as white noise, waveshaping, bit crushing, or pitch shifting. The filters used are second-order, which means you could add a resonance setting.
- Use an LFO (low frequency oscillator) to automatically modulate the delay time. This requires two new parameters: the frequency of the LFO (typically between 0.1 Hz and 20 Hz) and the modulation amount, which determines by how much the delay time is varied.
- Disable the Stereo knob in the editor when the plug-in is placed on a mono track. The audio processor somehow needs to inform the editor that the bus is mono or stereo.
- Make the editor resizable.
- Add a preset manager.

Currently there's a lot going on inside the audio processing loop in `processBlock`. It's a big monolithic function. Try to extract this logic into a class `PingPongDelay`. Moving the code into a separate class will simplify what happens in `processBlock`, but also gives you a new DSP building block that you can re-use in other projects.

The header file could look something like this:

```
class PingPongDelay
{
public:
    void prepareToPlay(float sampleRate, float maxDelayInMilliseconds);

    void setDelayTime(float delayInMilliseconds);
    void setStereoWidth(float value);
    void setFeedback(float value);
    void setLowCut(float freq);
    void setHighCut(float freq);
    void setDryWetMix(float value);

    void processSample(float inL, float outL, float& outL, float& outR);

private:
    // ...all the variables...
};
```

I strongly recommend that you try implementing some of the above ideas. It's good practice and you'll learn a lot from it, even if you can't quite get it to work. Reading a book is one thing but you won't really start to learn until you begin to implement features by yourself.

Once you start making this plug-in your own, I'm sure you'll come up with lots more creative ideas!

Accessibility

Before concluding this book, we should explore the topic of accessibility. In terms of software design, accessibility means that the software can also be used by people with special needs. With a few small tweaks the UI can become a lot more convenient to use, not just for visually impaired users or people with mobility issues, but for everyone. A good product is designed for people of all abilities.

The JUCE UI components have the following functions to support accessibility:

- `setTooltip`. A tooltip is a small bubble that is displayed when the mouse hovers over the component. You can use this to show a short help text of what the UI control is for, making the features of the plug-in more discoverable. (Note: to make this work, add a `TooltipWindow` object to the plug-in editor.)
- `setTitle`, `setDescription`, `setHelpText`. These functions provide text descriptions of the purpose of the component to screen reader software, such as VoiceOver on macOS and Narrator on Windows. Good descriptions here make it possible for visually impaired users to navigate the UI.
- `setExplicitFocusOrder`. This determines the order in which UI elements are given keyboard focus when you're using the **tab** key to move between them. If you don't set this order, JUCE may not always choose something that is logical. The focus order is used by screen readers but also for keyboard navigation. Making it possible to use the plug-in from the keyboard is a good accessibility feature as not everyone is able to use a mouse or trackpad.
- `setFocusContainerType`, `createFocusTraverser`. These are more advanced functions for defining parent/child relationships between the UI components from a navigation point of view, and are useful for UIs with complex hierarchies.

These same accessibility features can be added to custom components with a little bit of work. It's worth doing! A large part of user interface development is about polishing the product until it looks good and feels good to use, and adding accessibility support is one of those things.

The key thing in making a UI usable for people who may not have 100% functional bodies is to make the navigation between the different UI components as clear and simple as possible.

For more info about JUCE's accessibility features, see [this note in the JUCE repo⁵⁴](#).

A common form of visual impairment is color blindness. When designing your UI and choosing colors, it's a good idea to use a color blindness simulation tool that will show what the UI looks like for people with different forms of this condition. Two colors that may look great together to you may blend into the same color for someone else, but fortunately with these tools it's an easy mistake to avoid.

Tip: If you're on macOS, enable VoiceOver. If you're on Windows, enable Narrator. Be sure to read the instructions on how to disable this again first! Now try using the plug-in with your eyes closed. It's tough! However, it gives you a good idea of how hard — or hopefully how easy! — it is to navigate through your UI for someone who doesn't have full use of their eyes. Be sure to have actual visually impaired users test your plug-in too, their perspective on how they interact with the world can be very illuminating.

Further reading

There aren't that many approachable books on the market that give practical advice about building audio plug-ins but I've found a few that I can recommend. There are also a number of good books on DSP in general.

Not all of these books are about writing plug-ins and the source examples are not always in C++. Many DSP books use MATLAB, Python, or even BASIC. If you don't have any experience with MATLAB, it's not that hard to convert MATLAB code to Python since many of the Python numerical programming libraries such as NumPy, SciPy and matplotlib are inspired by MATLAB.

If you don't know Python, I can recommend learning it. Using Python together with Jupyter notebooks is an excellent way to prototype audio algorithms. In the end, all programming languages are pretty much the same and if you learn the basics of MATLAB or Python syntax, you shouldn't have too much trouble converting the code to C++ or your language of choice.

⁵⁴<https://github.com/juce-framework/JUCE/blob/develop/docs/Accessibility.md>

DSP books

Audio programming is a form of DSP, so it's a good idea to get a solid grounding in how digital signal processing works. Here is a short selection of quality DSP books:

Think DSP by Allen B. Downey. Available as a printed book and [online as a free PDF⁵⁵](#). This is a great introduction to the basic principles of digital signal processing. It's written for beginners and doesn't have lots of math. Especially nice is that most of the examples use audio signals. If you haven't studied any DSP at all, I highly recommend starting with this book. It uses Python and Jupyter notebooks, so it's also a good place to learn about those.

The Scientist and Engineer's Guide to Digital Signal Processing by Steven W. Smith. This is a fantastic book that is written for people who are new to DSP. While I recommend that you start with Think DSP because it's more relevant to audio and is a shorter read, this is also a must-read book. It is all about being practical so there is not too much math, and it goes deeply into the subject.

This book is about DSP in general, not audio, but it's all stuff you'll need to know anyway. The code examples are in BASIC, but you can simply treat this as pseudocode. This book can be read for free online, although I suggest that you [download the PDFs⁵⁶](#) and read those.

Understanding Digital Signal Processing by Richard G. Lyons. Another well-regarded DSP beginner book. This has more math than the other two books I listed but at some point you'll have to get dirty with the math, it's unavoidable. A good book to read after the above two. Available on O'Reilly online.

A Digital Signal Processing Primer by Ken Steiglitz. A solid introduction to DSP with a focus on audio applications and computer music. Like the most DSP books, it jumps straight into the math so this might not be for you if you have an aversion to equations and Greek symbols.

Introduction to Signal Processing by Sophocles J. Orfanidis. Another good [free online textbook⁵⁷](#) that covers all the basics of DSP. Written for university students, so it has more math than the others.

⁵⁵<https://greenteapress.com/wp/think-dsp/>

⁵⁶<http://www.dspsguide.com/pdfbook.htm>

⁵⁷<http://eceweb1.rutgers.edu/~orfanidi/intro2sp/2e/>

Audio programming books

Books that specifically cover audio processing, creating effects, and sound synthesis:

Designing Audio Effect Plugins in C++ and **Designing Software Synthesizer Plugins in C++** by Will C. Pirkle. These books contain a wealth of relevant information, although the explanations are sometimes confusing (you may want to read a regular DSP book first) and I'm not a fan of the coding style. There are two editions of these books. Personally, I found the first edition of the synth book to be easier to read, but both editions contain useful information.

Pirkle's books are among the few that talk about making plug-ins, although they do not use JUCE. The first editions use the AU and VST3 SDKs directly while the second editions use Pirkle's own framework, RackAFX / ASPiK, which is suitable for educational purposes but not for making real plug-ins. The first editions are available on O'Reilly online. I learned a lot from Pirkle's books and they're definitely worth getting.

The Computer Music Tutorial by Curtis Roads. This is a massive book that discusses all facets of computer music. This is not a programming book but it does describe effects and synthesis algorithms in a fair amount of detail. A great reference for anything related to computer audio and very readable. Every time I read it, I learn something new.

Hack Audio by Eric Tarr. An accessible introduction to audio processing algorithms. The code examples are in MATLAB but Python versions can be found on GitHub. This book is available on O'Reilly online. You can also find useful video tutorials on [Eric's website⁵⁸](#).

Audio Effects: Theory, Implementation and Application by Joshua D. Reiss and Andrew McPherson. A good practical book that explains a variety of audio effects. The code is in C++. The final chapter of the book explains how to make plug-ins in JUCE, but as the book is almost a decade old it may no longer be completely up-to-date with the latest versions of JUCE. Available at O'Reilly online.

Creating Synthesizer Plug-Ins with C++ and JUCE by Matthijs Hollemans (that's me). This book explains step-by-step how to create a complete software synthesizer plug-in. You'll learn the theory of how synthesizers work and how to implement the underlying DSP algorithms in C++. The book is [available from The Audio Programmer⁵⁹](#).

⁵⁸<https://www.hackaudio.com/>

⁵⁹<https://theaudioprogrammer.com/synth-plugin-book>

DAFX: Digital Audio Effects, Second Edition by Udo Zölzer. Another excellent book for learning about audio programming. The examples are in MATLAB. Expensive but worth getting for your collection or read it at O'Reilly online.

The Art of VA Filter Design by Vadim Zavalishin. This is a [free book⁶⁰](#) about designing filters using the Topology Preserving Transform (TPT), which is a mathematical method for taking an analog filter design and making it work in software. I'm not going to lie: this is a tough book to get into. The first chapters are a terse summary of many decades worth of DSP theory and are math heavy. If you can look past the math, this book is a fascinating look at how filters can be designed. Not a book I would recommend to a beginner but worth a read if you're curious about filters.

Julius O. Smith's publications: The [JOS publications⁶¹](#) at Stanford's CCRMA (Center for Computer Research in Music and Acoustics) are legendary. There are several online books about topics such as filters and physical modeling synthesis, and many other relevant publications. You need to be comfortable with the math, but there's enough here to keep you coming back for a long time.

The Theory and Technique of Electronic Music by Miller Puckette. This is an online textbook that covers many synthesis techniques. The language used is Pure Data (Pd), which is more of a prototyping environment than a real language but that should not matter. It's free and [worth checking out⁶²](#).

There is a book about JUCE, *Getting Started with JUCE* by Martin Robinson. Much as I like to support my fellow authors, I cannot recommend getting this book. It's old, it does not describe latest JUCE best practices, and doesn't cover much about audio programming anyway. If you have an account, read it at O'Reilly online and make up your own mind.

C++ books

Learning more C++ never hurts. There are lots of books and online tutorials on C++, some better than others. [The Definitive C++ Book Guide and List⁶³](#) has an overview of the best available books.

⁶⁰https://www.native-instruments.com/fileadmin/ni_media/downloads/pdf/VAFilterDesign_2.1.0.pdf

⁶¹<https://ccrma.stanford.edu/~jos/pubs.html>

⁶²<http://msp.ucsd.edu/techniques/v0.11/book.pdf>

⁶³<https://stackoverflow.com/questions/388242/the-definitive-c-book-guide-and-list/>

Here are my tips for learning C++:

C++ Primer (5th Edition) by Lippman, Lajoie & Moo. If you have no programming experience at all, this is usually the recommended book to start with.

A Tour of C++ by Bjarne Stroustrup, the inventor of C++. A good starting point for intermediate programmers who are new to C++. Also worth reading if you programmed in C++ ages ago and want a refresher on all the new features in the language.

The Cherno on [YouTube](#)⁶⁴. If you're more into watching videos than reading books, this is a very good series of videos on how to learn C++.

cppreference.com This is not a tutorial but an indispensable reference guide for when you need to look up something in the language or the standard library. [Bookmark this!](#)⁶⁵

Mathematics

You can't avoid it, DSP and audio programming involves math. You don't have to be a math wizard to be able to write plug-ins, but it's nice not to panic when you encounter some equations.

Math Overboard! (Basic Math for Adults) by Colin W. Clark. This is a set of two books that explain how math works from the ground up, written for adults who want to learn math, even if you flunked it in school. These are the books I used to brush up on my math skills.

An Introduction to the Mathematics of Digital Signal Processing by F. R. Moore. This is a set of two tutorials published in the Computer Music Journal in 1978 but they're still relevant. The PDFs are behind a paywall but you can easily find them with a bit of googling.

Musimathics by Gareth Loy. This is a wonderful set of two books that looks at the math behind music and sound in the real world, as well as in electronic and digital systems. Great stuff, especially if you've built up an appreciation for mathematics.

Khan Academy has excellent [free online courses](#)⁶⁶ on high school and college math, and pretty much everything else.

⁶⁴<https://www.youtube.com/playlist?list=PLlrATfBNZ98dudnM48yfGUldqGD0S4FFb>

⁶⁵<http://www.cppreference.com>

⁶⁶<https://www.khanacademy.org>

Online materials

The JUCE website has [tutorials⁶⁷](#) and API documentation. If you haven't looked at the tutorials yet, you definitely should. They don't just explain how to use JUCE, but also how to do practical audio programming. Make sure to check out the [JUCE forums⁶⁸](#). And don't forget the example projects that come with JUCE in the examples folder (also accessible from Projucer).

The Audio Programmer If you learn best by watching videos, I can heartily recommend [Joshua Hodge's YouTube channel⁶⁹](#). He explains how to use JUCE to make audio effects and synths. Here you can also find the videos and live streams of monthly meetups where Josh and his co-hosts interview experts from the audio development world.

GitHub There is a massive amount of audio code on GitHub and some of it is very high quality. The [MDA plug-in suite⁷⁰](#) of classic effects is a good place to get started. Also check out the source code of the [Surge XT⁷¹](#) synthesizer. This is a top-notch synth and you can see exactly how it's made, how awesome is that! Reading other people's source code is great practice, and after reading this book I'm sure you'll be able to decipher how these GitHub projects work too!

Musicdsp.org A collection of algorithms, thoughts, and snippets, gathered from the now defunct Music-DSP mailing list. There are a lot of [useful source code snippets⁷²](#) here. Just keep in mind that some of these techniques are considered outdated.

KVR Forums The KVR website is all about audio gear and software, but they also have an [audio development forum⁷³](#). As is typical for forums, not everyone is always on their best behavior, but you can find great discussions here and insights shared by well-known audio developers. Worth keeping an eye on.

Blogs There are plenty of great blogs with in-depth audio programming information. Too many to mention them all here, so I will just link to my own [audiodev.blog⁷⁴](#), Nigel Redmon's [earlevel.com⁷⁵](#), and Geraint Luff's [Signalsmith Audio blog⁷⁶](#).

⁶⁷<https://juce.com/learn/tutorials>

⁶⁸<https://forum.juce.com/>

⁶⁹<https://www.youtube.com/c/TheAudioProgrammer>

⁷⁰<https://github.com/hollance/mda-plugins-juce>

⁷¹<https://github.com/surge-synthesizer/surge>

⁷²<https://www.musicdsp.org/en/latest/>

⁷³<https://www.kvraudio.com/forum/viewforum.php?f=33>

⁷⁴<https://audiodev.blog>

⁷⁵<https://www.earlevel.com/>

⁷⁶<https://signalsmith-audio.co.uk/writing/>

Journals and organizations

A lot of the knowledge of the field is in the form of academic papers. You can often find these papers in PDF format online by googling their title. Sometimes the papers are published by a journal and are behind a paywall.

Journals and organizations for audio development and computer music:

- [AES, the Audio Engineering Society⁷⁷](https://aes.org). This is a professional society of audio engineers, artists, scientists, producers, and anything else having to do with audio and sound. It covers a lot more than software development. Not only can you learn a lot from being in an organization like this but you can also connect with other members, who might just turn out to be the future users of your plug-ins.
- [DAFX conference⁷⁸](http://www.dafx.de): An international conference that specializes in digital audio effects. The DAFX book mentioned above is a collection of works that originated here. You can read the papers from previous conferences at this website.
- [Computer Music Journal⁷⁹](http://direct.mit.edu/comj/). A quarterly publication that goes back all the way to the 1970s.
- [International Computer Music Association⁸⁰](http://computermusic.org)
- [Journal of the Acoustical Society of America⁸¹](http://asa.scitation.org/journal/jas)
- [IEEE signal processing society⁸²](http://signalprocessingociety.org). This covers all DSP, not just audio.

The community

A lot of conferences and organizations swing towards the academic side of things more than the practitioner's side. However, there is a [great conference⁸³](#), the **Audio Developer Conference** or ADC, that focuses on the people writing the code. It is organized by the people behind JUCE. You can find videos of previous ADC talks on [their YouTube channel⁸⁴](#).

⁷⁷<https://aes.org>

⁷⁸<http://www.dafx.de>

⁷⁹[https://direct.mit.edu/comj/](http://direct.mit.edu/comj/)

⁸⁰<http://computermusic.org>

⁸¹<https://asa.scitation.org/journal/jas>

⁸²<https://signalprocessingociety.org>

⁸³<https://audio.dev>

⁸⁴<https://www.youtube.com/c/JUCELibrary/videos>

The Audio Programmer Community This is a Discord server with lots of smart and fun people (and me) who love to chat about audio development all day long. This Discord is very welcome to newbies but we also have a number of people who work for audio companies that you know and love. [Come and join⁸⁵](#), it's free and you will learn tons.

The end beginning

All stories must end and so does this book. Hopefully you were able to more or less follow what we did and how everything works — without breaking your brain in process!

Now how do you get started making your own plug-ins?

My advice: Start as simple as you can. In this book we began by adding a really basic **Output Gain** parameter. That wasn't purely for educational purposes: I usually start my projects with a gain control because most plug-ins need one. It's pretty straightforward to implement and it already lets you add a lot of the plumbing that you'll need later on — the APVTS, a Parameters class, parameter smoothing, and so on.

There's often a mental barrier to getting started. Having an empty canvas, so to speak, can be extremely intimidating. One way to overcome this kind of "getting started paralysis" is to begin with something simple, like a gain knob. Once you have that, you keep adding stuff until the plug-in is finished.

Make no mistake: You still have lots of things to learn, and if your own plug-in ideas are very ambitious, they may be out of reach for a while. There's no way a single book like this one can teach all you need to know — the topic of C++ and audio programming is too vast for that. But hopefully you are now armed with at least the basic knowledge to get started, even if it's just with a gain parameter.

I also encourage you to read the book again in a couple of weeks. You will likely pick up some new insights on a second or even third reading.

But most importantly: Stop reading for a while and start writing some basic plug-ins of your own, no matter how simple or silly. You need to run into strange compiler errors, unexpected crashes, the sound going haywire, and all the other things that can go wrong — and then learn how to get yourself out of such scrapes. The only way to get better at development is to write code. Always keep learning!

⁸⁵<https://theaudioprogrammer.com/community>