Audio Process Design & Implementation
Practical 3 - Introducing MyEffect

1 Purpose

This practical will introduce you to the *MyEffect* code project, and the process of developing audio plugins. We will explore plugins using the APDI "mini-plugin" development platform, and produce *Audio Unit (AU)* or *VST* plugins that will run in almost any Mac or Windows music program, such as Logic Pro X and Cubase. Through *MyEffect*, the low-level, platform-dependent code has been abstracted, allowing you to focus on the audio and signal processing functionality. The concepts explored are universal to all plugin architectures, as well as analogue (hardware) approaches. A working knowledge of basic C++ programming is the only pre-requisite.

2 The MyEffect plugin

A plugin is a piece of compiled code used to extend the functionality of a larger (host) program (such as Logic, Cubase, Pro Tools, Premiere, Amadeus Pro, etc.).

The main benefits of plugins are:

- Third-parties can add extra features to a program without access to its source code. This also reduces the onus on developers to include every conceivable feature in the original program.
- The host handles low-level communication with the hardware, providing a basic interface (for audio and MIDI), so that the plugin developer can focus on audio and music functionality.
- Users need load only the features they need, rather than working in a workspace cluttered with unused components. They can likewise replace features of the host with alternatives.
- Through a single body of code, programmers can develop audio and music functionality for multiple host programs and/or operating systems - *write once, run anywhere*.

The challenges are:

- Plugin functionality must provide a consistent code-level interface to the host, to ensure compatibility. The plugin specification may impose limits on what is possible (e.g. user interaction or audio connectivity). Outside realtime audio and MIDI, other user or musical data stored in the host is usually not accessible to a plugin.
- Debugging can occasionally be more complex than in a self-contained, standalone application, as plugin developers don't have access to the code for the full system (i.e. the host).
- The plugin is at the mercy of the host. While a plugin specification defines what should be possible, not all hosts implement every feature - nor provide the functionality consistently.

A good way to think of a plugin is as a software equivlent to a hardware audio device. You can add it or remove it from your system or studio, but it has to use standard interconnects that all the manufacturers have agreed on, so that the various black boxes can communicate with each other.

2.1 Mini-Plugin Architecture

The `MyEffect` code project uses a JUCE-based *Audio Unit (AU)* and *VST* mini-plugin system specially developed at UWE to expose only the core components of the plugin architecture, to help students learn, explore, and develop effects processing plugins.

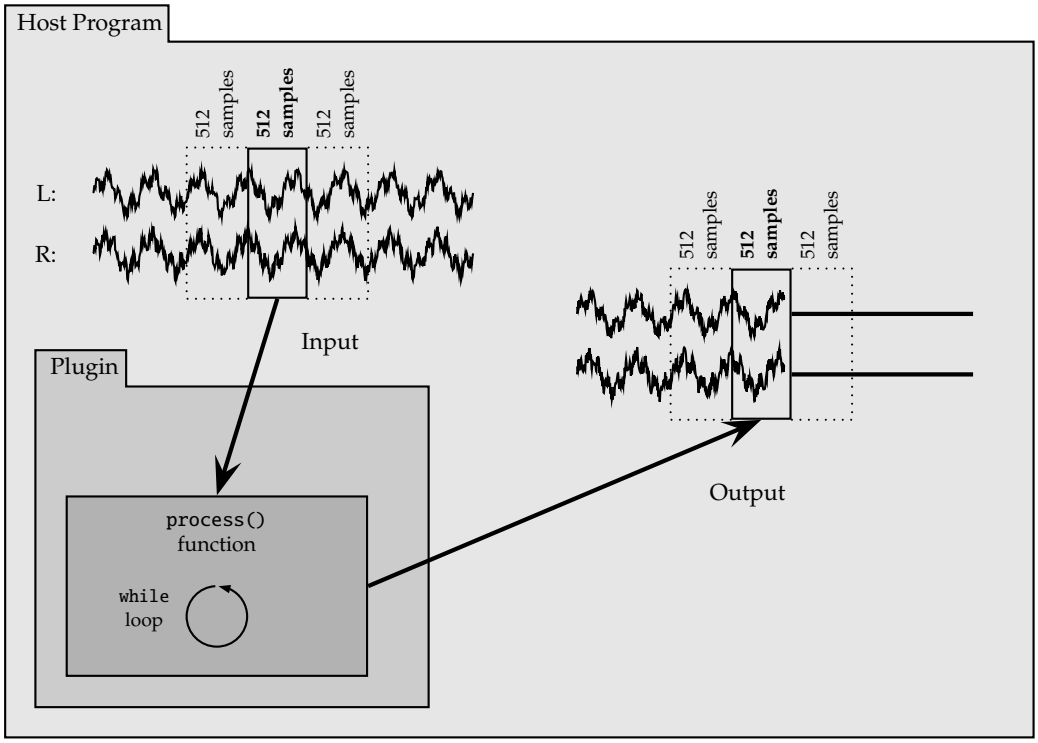
In combination with the provided *Effect Plugin* container, your code can run in any audio program that supports AU or VST plugins, including nearly all professional audio packages on Mac or Windows - Logic, Cubase, Pro Tools, Amadeus, Premiere, etc. A customisable UI is also supported, through which you, your users, and the host (through automation) can control the effects process, also providing basic realtime analysis of your plugin's output (waveform, spectrum, sonogram).

This allows you to ignore platform-specific details, such as OS (Mac or Windows), audio API (CoreAudio, WASAPI, WDM, DirectX, MME, etc.) or plugin SDK (AU, VST), which are handle for you by the *Effect Plugin*, enabling you to focus on digital signal processing (DSP). Beyond this, you will need little more than a knowledge of C/C++, which we will further develop over the year.

2.2 Audio Processing using Buffers

The central part of any audio plugin is the code used to process audio data, which is usually contained within a single function inside the plugin. The host calls this regularly, supplying input and output buffers that the plugin reads from and writes to, respectively.

A **buffer** is simply an *array* of floating-point numbers representing the audio signal. The input and output buffers are stored in the host, but it provides *pointers* to the array, so that the plugin can access the data. We process data in buffers, rather than individual samples, to improve performance - since a lot of the tasks required to prepare and process multiple samples are similar, we look to do them as few times as possible. Using tricks such as iteration (i.e. a *while* loop), we are able to write fast code to process multiple samples in one pass, as illiustrated below.



The disadvantage of using buffers is that it introduces **latency**. If we have a buffer of 512 samples, then the host has to wait until it has captured 512 samples from the audio input, before it can deliver them to the plugin for processing. Similarly, the host then has to wait until all 512 samples are processed and returned by the plugin, before it can play them.

This leads to both an input and output latency, proportional to the size of the buffer - e.g. 23ms for a 512 sample buffer, using a 44,100Hz sample rate (`(512+512)/44100 = 0.023s`) - though for improved stability the system might have additional safety buffers, further increasing the latency. Latency can obviously be reduced by reducing the audio buffer size, but this comes at the cost of reducing the efficiency of the processing, leading to higher CPU load.

2.3 Getting Started

1. From Blackboard, download the *MyEffect* project template (under "Learning Materials"). Copy the `MyEffect.zip` file to a local folder and double-click it to expand its contents.

   Note: Always be sure to copy work to safe, permanent storage - your APDI folder plus an external drive and/or cloud-based storage (e.g. OneDrive). See Section 2.4.3 for tips.

   The resulting folder contains two sub-folders:

   - `Build` - containing the executable binaries (including *Effect Plugin*, *Plugin Host*, and your compiled *MyEffect* mini-plugin), plus any temporary build files.
   - `MyEffect` - containing all the source code, resource files, and code projects. This folder contains **all your work**, and thus the one you need to back up or submit for assignments.

2. In the `MyEffect` folder, open the appropriate project file `MyEffect.xcodeproj` (Mac/Xcode) or `MyEffect.vcxproj` (Windows/Visual Studio), by double clicking it.

   - The project consists of three main code files: `EffectPlugin.cpp/h` and `EffectExtra.h`.
   - To test our plugin, a dedicated program called "Plugin Host" is provided. The program should install and run automatically when you build and run your code. It should also handle configuration and installation of the *Effect Plugin* and your *MyEffect* mini-plugin.

3. Build and run the project (⌘R in Xcode; F5 in Visual Studio). After a short delay, you should see your plugin appear, running in *Plugin Host*. Click Play to check the audio is working.

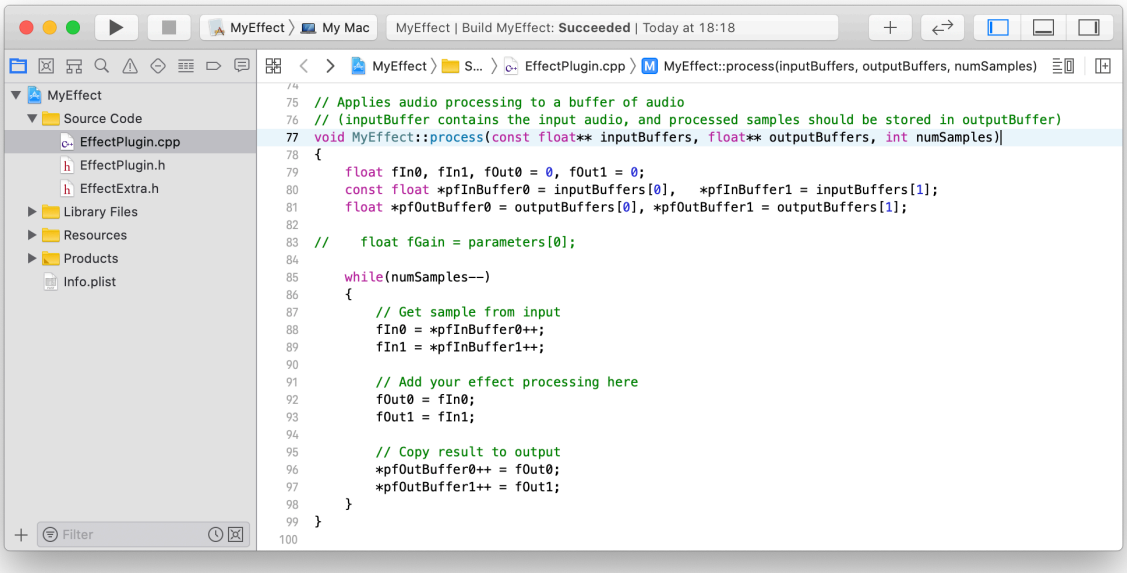
4. If your plugin does not appear, try the following troubleshooting steps:

   - After compiling, check in the `Build` folder to make sure the `MyEffect.bundle` (Mac) or `MyEffect.dll` (Windows) files have been produced. This is the mini-plugin that links with the `Effect Plugin.component` (AU) or `Effect Plugin.dll` (VST) plugins.
   - Running the project installs / launches the *Plugin Host*, which in turn automatically installs and/or runs your plugin. On Windows, everything runs from the `Build` folder. On Mac, plugins need to be installed to the system folder, which *should* happen automatically when you first run your code, but you might see errors (see inset).
   - Otherwise check the *Discussion Board* on Blackboard and/or report any problems (always including details of your system and any error messages) to the module leader.

***Mac only**: To check the install in Finder, press Shift-⌘G and enter: `~/Library/Audio/Plug-Ins/Components` (note the ~ tilde). Verify that `MyEffect.bundle` and `Effect Plugin.component` are both present. If they are, you might need to either re-run the project or restart the system to refresh your audio plugin cache. If `Effect Plugin.component` is missing, you can try copying it manually from the `Build` folder. If you have an LLDB error about `Plugin Host.app`, try manually copying it from the `Build` folder to your main `Applications` folder, then re-run the project from Xcode.*

If shown a security warning(s), open your Security & Privacy preferences and click "Allow Anyway" to unblock `Effect Plugin.component`. Then, the next time you run it, you'll be able to click "Open". Note that because the OS interrupted the program, it will initially crash, but should run flawlessly on future runs.

2.4 The `process(...)` function



In the *MyEffect* project, the function used to process audio is called `process(...)`. In practicals, it will be up to you to provide the code for this function, using a standard template, which you should understand and follow, as explained in this section. This function is called regularly, by the host, when it is time to process a buffer of audio samples. The default code simply passes data from the input buffer to the output buffer without changing it. Expand the project tree, click the *EffectPlugin.cpp* file and locate the `process()` function (as pictured above).

```
void MyEffect::process(const float** inputBuffers, float** outputBuffers, int numSamples)
```

Through the arguments to the function, the host supplies pointers to two stereo buffers, for input and output audio. Notice that the arguments are pointers to pointers (**) representing arrays of arrays - that is, a stereo buffer is an array of two audio buffers, each of which is an array of floats.

- `**inputBuffers` points to a *stereo* input buffer, containing read-only audio to be processed.
- `**outputBuffers` points to a *stereo* output buffer, to place audio in after processing.
- `numSamples` indicates how many samples each buffer contains.

Each audio buffer contains 32-bit floating-point sample values representing the changing amplitude of the audio, which has a **maximum range of -1.0 to 1.0**. Amplitudes beyond +/-1.0 may lead to unexpected clipping and distortion, and should be avoided.

2.4.1 How it works: line-by-line

The role of the `process()` function is to go through each input sample and process it to yield an output sample, which is placed in the output buffer. Samples from each channel have to be processed separately, but we tend to do their calculations at similar points in the function, using similar code. A pair of left and right samples is called a sample **frame**.

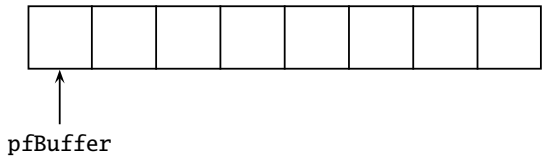
At the start of the function, we create four pointers, addressing each buffer of each stereo channel (0=left, 1=right). For example, inputBuffers[0] points to the left input buffer:

```
float *pfInBuffer0  = inputBuffers[0], *pfInBuffer1  = inputBuffers[1];
float *pfOutBuffer0 = outputBuffers[0],*pfOutBuffer1 = outputBuffers[1];
```

To iterate over the individual samples of each buffer, we use a **while loop**. As in the last practical, the while loop and decrement (--) operator are used to count down the number of iterations. When `numSamples` reaches 0, all frames will have been processed and the loop (and function) ends:

```
while (numSamples--)
{
    ...
}
```

Unlike a for loop, an iterator variable is not used to access the array. Instead, we access the data by *dereferencing* a moving pointer (using the * operator), in the first two lines of the loop. The diagram below illustrates how the pointers are simply a signpost to the current position in the array:



Inside the loop, the current input sample value is copied into a local variable for each channel (fIn0, fIn1), declared before the loop, before the pointer is moved on (ready for the next read). This local variable offers us a temporary copy we can modify, without touch the (read-only) input buffer:

```
fIn0 = *pfInBuffer0++;
```

We then place code to process sample values in the subsequent lines of code. However, in the default template, no modification is made. Instead, the input is simply copied to a temporary output variable...

```
fOut0 = fIn0;
```

When you add your own code, you will replace the line(s) above, using fIn0/1 to calculate fOut0/1. Lastly, in the final lines of the loop, we copy the temporary output variable to the output buffer (by dereferencing the output pointer), and also increment the pointers to each output buffer (*after* copying the values). Note how we are incrementing the addresses in memory, not the values they are pointing to:

```
*pfOutBuffer0++ = fOut0;
```

2.4.2 Performance considerations

It is important to think about where you place your code, to ensure your plugin runs efficiently in realtime. If your code is inefficient, it can lead to glitches in the audio output. Hosts will give your plugin a fixed window of time to process and return the samples. If they are not returned in time (e.g. before the host moves to the next buffer), problems arise - audio overloads and dropouts will produce glitches and crackle in the output, since the host has no audio (only silence) to output.

Code inside the while loop is executed for *each sample* in the buffer - at ***audio rate***. For example, in a system with a 44,100 Hz sample rate, the while loop repeats 44100 times every second. As such, if you add inefficient or unnecessary code to this section, the time it takes will be magnified thousands of times.

By contrast, the code outside (e.g. before) the loop is only executed once *per buffer* - e.g. once every 512 samples, or 512 times slower than sample rate (about 86 Hz). Therefore, calculations that do not have to be calculated at audio rate, including user input handling and other slower-evolving processes that control audio processing rather than apply it, should be calculated outside the while loop - at ***control rate***.

2.4.3 Archiving your work

Over the semester, you will make many plugins, and put a lot of effort into coding for the practical exercises and assignments. It's **extremely important** to save your progress, as well as organise the code for different exercises and projects. Save (and back up) often, for each exercise, as follows:

1. Select the `MyEffect` subfolder and **compress it to a .zip archive**:
   - **Mac**: Right click the folder in Finder, then choose `Compress X items`. A new file appears, named `Archive.zip` or `MyEffect.zip`.
   - **Windows**: Right click the folder in Explorer, then choose `Send to` → `Compressed (zipped) folder`. A new file appears and prompts you to enter a name for it.
   - You do not need to include the `Build` folder, which is much bigger (several MBs) - and contains programs that can either be replaced from a freshly downloaded project or are automatically regenerated using Xcode or Visual Studio.
   - Generally, the most important files are those containing your source code:

     EffectPlugin.cpp, EffectPlugin.h, and EffectExtra.h

     You might chose to only archive these files, but be sure not to overlook any important supplementary files (e.g. custom audio files or background art). These three files are also the ones you should attach when emailing tutors with questions about your code.
2. Be sure to name the archive to tell you what it contains (e.g. *Practical 3 - Ex 3.1.zip*), and store it somewhere sensible (easy to find) and safe (hard to lose). It should only be a few KBs.
3. To retrieve your work later, simply double click the archive to extract the folder, and use it in place of the `MyEffect` folder in a newly downloaded code project.

3 Gain, Mixing and Panning [BASIC]

- **Only change code inside the `process()` function, unless otherwise indicated.**
- **You should never use "Save As..." or rename a file inside the project, or change the configuration settings. Doing so may cause the project to stop working.**
- **After each exercise, archive your code, as described in the previous section.**

3.1 Coding your first plugin

Your first task is to add an amplitude gain control based on "Param 0" (i.e. the first dial). To do this, you will use the commented-out line of code, in the `process()` function, which reads a value from the user interface (UI):

```
// float fGain = parameter[0];
```

The `parameter` array references the current settings for each control in the UI, using the subscript in square brackets to identify which. You can both read from and write to these settings, which also correspond to the plugin's automation parameters, controllable from the host. Each parameter (and UI control) is defined in the `UI_CONTROLS` array, passed to the plugin on startup, which will be explained in a later practical. By default, you have 10 rotary sliders ([0] to [9]).

The = operator assigns the current value of the first parameter / control (a value **between 0.0 and 1.0**) to `fGain`, which is declared as a temporary local variable.

1. Work out how to use the `fGain` variable inside the while loop to change the sample values being fed to each output buffer, to apply a change in gain. That is, at 1.0 there should be full amplitude; at 0.0, silence.

   You should remember how to do this from lectures (or the printed course notes) - but as a hint, one of the following mathematical operators should be used: `+` `-` `*` or `/`

2. Test your code:

   (a) Build and run the plugin, and test your code with the built-in test sounds.

   (b) Use `leftrightEP.aif` to ensure that the effect works on both left and right channels controlled from a **single** dial. If not, return to your code and try to address the issue.

3.1.1 Testing in Another Host

Congratulations, you have created your first interactive effects plugin! To demonstrate that the plugin is not restricted to working in *Plugin Host*, we can test it in another host. On Mac, *Effect Plugin* should already be available to other programs, like *Logic Pro* or *Amadeus Pro*, and will load your *MyEffect* code automatically. On Windows, you will need to copy `Effect Plugin.dll` and `MyEffect.dll` to your `vstplugins` folder, before opening the VST host.

Load your favourite program, and test the plugin works on a 2-channel stereo track (note: the host might not load the plugin unless the track is stereo). If the program organises its effects plugins by manufacturer, look under `UWE` → `Effect Plugin`.

3.2 Basic Mixing [BASIC]

In this exercise, we seek to mix the stereo input channels together to produce a mono mix, and control the mix with separate dials. The plugin will still output in stereo, but the contents of each channel will be the same: a mix of the two input channels.

1. We need a place to store the mix of both channels. Declare a suitable variable to contain a single sample value, before the `while()` loop, and give it a suitable (informative) name.

2. Inside the loop, use it to create a simple mix of the left and right input sample values. Because we are combining two signals together, they might peak louder. Work out how to halve the amplitude of the mix, to guard against this.

3. Send the result to **both** outputs to create a stereo-to-mono effect. Test this by running it in *Plugin Host* (from your IDE) to ensure that the previously stereo effect of `leftrightEP.aif` is now mono.

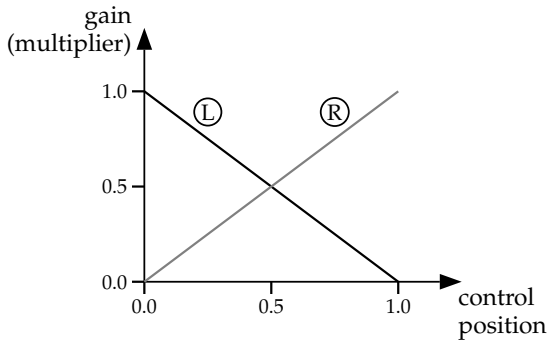
4. Now imagine the left and right inputs were two separate mono channels on a mixer, each with independent gain controls (e.g. faders). Using the following tips, work out how to change your code to simulate a simple 2-channel mixer:

   - Use the original dial to set the level of the left channel, and a second to set the right. You will have to modify and extend your fGain code to handle each channel and dial.
   - Mix the channels in proportion to their levels, sending the mix to **both** output channels.

   Test that it is possible to create a mix of just the left input signal, or just the right input signal, or a combination of both, and that the result is **mono**. A good test file is `leftrightsynth.aif`. Remember to archive your work before proceeding.

3.3 Basic Panning [BASIC]

In this exercise, we will use only the first dial to achieve a panning effect for a mono mix. This is done by having the dial directly control the gain of the mix in the right channel, and inverting the gain in the left. If line "L" is the gain value for the left output, and "R" is the gain value for the right output, then the control mapping is as follows:



Remove the individual gain code from the previous section, but keep a simple mono mix `fMix` of the two inputs (which will be panned left or right). To achieve the panning effect, use the control mapping $y = 1 - x$ to change the right gain values (x) to those needed for the left (y). Implement the effect. Test it carefully to ensure that the result is exactly as expected.

4 Beyond the `process()` function. [CORE]

In this section, we will move beyond the confines of the `process()` function, investigating some of the other functions and code associated with the MyEffect plugin, whilst exploring the difference between audio and control rate processing.

We will revisit the wavetable array from the previous practical (Practical 2, Section 6), and integrate the code with the plugin to explore how the audio effect can change, depending on where code is placed. This exercise will also introduce different variable scopes: local and shared variables.

If you have not completed the previous practical's exercise, go back and do so before continuing. Be sure to check your solution with a member of staff.

4.1 Shared Variables

We are going to use the wavetable as an oscillator to control the gain of our audio process. However, if we declare the array locally, in the `process()` function, it will be cease to exist when the function exits. We'd thus have to recreate and recalculate it every time the function is called, which isn't efficient, but would also lose track of where we were in the wavetable.

Instead, we must place the array not only where we can access it from `process()`, but also where it will survive, so that we only have to calculate it once. To achieve this, we will declare the variable as part of the `MyEffect` class, so that it is available throughout our plugin object:

1. Start with a clean copy of the `MyEffect` code project.
2. Declare the wavetable array as a **shared variable**. To do this, place the line of code that declares `fArray` inside the `MyEffect` class declaration (in the header file, `EffectPlugin.h`). There is a comment to highlight where shared variables should go - add the code immediately below, making sure it appears before `MyEffect`'s closing braces.

3. As before, we must now fill the array with the values of the sine wave. To do this, find the `MyEffect::initialise()` function definition, in `EffectPlugin.cpp`. This function is called once, when the plugin is first created, and is useful for initialising variables, allocating extra memory, or otherwise preparing the plugin before audio processing starts.

   Identify the code in the last practical that you used to iterate over the array and calculate its values. Copy and paste the relevant parts into the `initialise()` function. Do not include the array declaration; the code will use the one you just declared for the `MyEffect` class.

4. As we want to control amplitude, the arrays values must be in the range 0.0 to 1.0. Edit the code to **scale** (*) and **offset** (+) the sin values so they map to the range 0.0 to 1.0, instead of -1.0 to 1.0 (i.e. halve the range, then make it so that instead of starting at -0.5, it begins at 0.0).

4.2 A Wavetable Oscillator

We will use the array values to control the gain applied to the input audio, stepping through the wavetable to vary the gain over time. To do this, we also need to keep track of our current position within the array, moving it along as necessary.

1. Like the array itself, the position variable needs to survive beyond the end of the `process()` function. Declare another shared variable, a single 32-bit integer called `iArrayPos`, as before (i.e. in the `MyEffect` class declaration, in `EffectPlugin.h`).

2. In the `MyEffect::initialise()` function (in `EffectPlugin.cpp`), initialise the variable to a default value of 0, indicating the beginning of the array.

3. Now, back in `process()`, we need to add code to step through the array, and use the values in the array to control the gain. Add the following code, *before* the while loop:

```
iArrayPos++;
if(iArrayPos == 16)
    iArrayPos = 0;

float fGain = fArray[iArrayPos];
```

4. Modify the code in the while loop to use `fGain` to attenuate the amplitude of both channels.

5. Build and run the plugin to test your code. Use `electricguitar.aif` to audition the effect. If successful, it should bear resemblance to a common guitar effect.

   Note, however, that the audio will be noisy and glitchy. This is because there are so few steps in our wavetable that the jumps in amplitude are large enough to produce audible clicks. One way to address this is to increase the size and resolution of the wavetable. Try to work out how to do this in the code - you'll need to make at least four changes.

   Similar issues frequently arise whenever control variables have a low resolution (as in MIDI) and audibly step between values, producing abrupt changes in the sound. This is known as *parameter* or *zipper noise*. We will look at another way to address this in a future practical.

6. Now take the code inserted in Step 3 and move it to the start of the while loop. Audition the plugin again. The effect should sound very different. Can you work out what is happenning?

Finishing up

Remember to comment and archive your work, to allow you refer back to it later on.

If you get unexpected results, check your code / answers with a member of staff.