

# How I coded a JUCE plug-in using ChatGPT

By: Patrick Lacey

Recently I've been working and learning with JUCE. For those that don't know, [JUCE](#) is a great piece of software, it is an open-source cross-platform C++ application framework for audio application and plug-in development. The documentation for JUCE is extensive and well laid out and I wondered if I could use ChatGPT to reduce the time necessary to develop the plug-in and significantly boost my efficiency, especially for the DSP portion of the code.



Instead of calculating filter coefficients or compensating for latency between dry and wet samples, I wanted to focus on a higher level of processing and also the more creative with my projects. Throughout this writeup, I will be going over my process with using AI as a tool to write a Phaser and Chorus plugin where the GUI switches between effect specific parameters.

Initially, I knew that ChatGPT could write C++ code, but I was unsure that it had access to the JUCE documentation. So the first thing to do was to ask:



Do you have access to the juce documentation?



As an AI language model, I do not have direct access to the JUCE documentation. However, I   can provide you with general information about JUCE and help answer any questions you may have about it. Additionally, I can assist with understanding programming concepts and provide guidance on how to approach certain tasks.

After that assurance, I had to choose what my plugin would do. I chose to start out by creating a simple Phaser and then build off that later:



Could you code an implementation of `juce::dsp::Phaser` in C++?



Certainly! Here's an example implementation of `juce::dsp::Phaser`` in C++:



C++

Copy code

```
#include <JuceHeader.h>

// Declare a `juce::dsp::Phaser` object as a member variable in your class
class MyClass
{
public:
    MyClass()
    {
        // Initialize the phaser with the desired parameters
        phaser.setRate(0.5);
        phaser.setDepth(0.2);
        phaser.setCentreFrequency(1000.0);
        phaser.setFeedback(0.9);
    }

    // Process audio using the phaser
    void processAudio(juce::AudioBuffer<float>& buffer)
    {
        juce::dsp::AudioBlock<float> block(buffer);
        juce::dsp::ProcessContextReplacing<float> context(block);
        phaser.process(context);
    }

private:
    juce::dsp::Phaser<float> phaser;
};
```

The AI spit out this code in easily under a minute and is pretty much ready to use. It even wrote out a bit

In this example, we declare a `juce::dsp::Phaser`` object as a member variable of a hypothetical class `MyClass``. We then initialize the phaser with the desired parameters in the constructor of the class.

To process audio with the phaser, we create a `juce::dsp::AudioBlock`` from the audio buffer and a `juce::dsp::ProcessContextReplacing`` from the audio block. We then call `phaser.process()`` on the context to apply the phaser effect to the audio.

This is just an example implementation, and the specific details of how you use the `juce::dsp::Phaser`` class may depend on your specific needs and the context in which you're using it.

of an explanation of the code it made. When it came to adding it to my project, I created a header file called MyClass.h within the jucer and copied the code in there. To be able to access the header file within the PluginProcessor, I needed to add simple set functions to the header file for the parameters of the Phaser. I also had to add a prepare function for the Phaser, so I asked ChatGPT to do that for me:



Can you add a .prepare function to the phaser class?

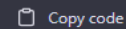


As an AI language model, I am not able to modify the actual JUCE library code. However, I can provide guidance on how you can implement a `.prepare()` function for the `juce::dsp::Phaser` class in your own code.`



The purpose of a `.prepare()` function in a JUCE `dsp` effect class is to initialize the internal state of the effect and configure it for processing a specific block size and sample rate. Here's an example implementation of a .prepare() function for the juce::dsp::Phaser` class:`

c++



```
class MyPhaser : public juce::dsp::Phaser<float>
{
public:
    void prepare(const juce::dsp::ProcessSpec& spec)
    {
        setRate(0.5f);
        setDepth(0.2f);
        setCentreFrequency(1000.0f);
        setFeedback(0.9f);
        reset();


        juce::dsp::Phaser<float>::prepare(spec);
    }
};
```




In this example, we define a new class `MyPhaser` that inherits from juce::dsp::Phaser<float>`. We define a new .prepare() function that takes a juce::dsp::ProcessSpec` object as its argument.`

In the `.prepare()` function, we set the phaser parameters to their desired values, reset the internal state of the phaser, and then call the base class's `.prepare()` function to configure the phaser for the specified block size and sample rate.

This is just an example implementation, and the specific details of how you implement a `.prepare()` function for the `juce::dsp::Phaser` class may depend on your specific needs and the context in which you're using it.`

With those few and simple additions, I have a ready to use Phaser within this header file. The next step is to implement this Phaser in my PluginProcessor. To do that, I needed to create adjustable parameters with proper ranges within a ValueTreeState. So, I asked ChatGPT once again.

 What are good ranges for the parameters of the Phaser

 The optimal ranges for the parameters of a ``juce::dsp::Phaser`` effect can vary depending on the desired effect and the characteristics of the input signal. Here are some general guidelines for setting the parameters:  

- **Rate:** The rate parameter determines the speed of the phase shifting effect, and is usually set between 0.1 and 10 Hz. Lower rates create a slower and more subtle effect, while higher rates create a faster and more pronounced effect.
- **Depth:** The depth parameter controls the amount of phase shifting applied to the input signal, and is typically set between 0.1 and 0.9. Lower values create a more subtle effect, while higher values create a more pronounced effect.
- **Centre frequency:** The centre frequency parameter determines the frequency at which the phase shift is centred, and is typically set between 100 Hz and 2 kHz. The optimal frequency range will depend on the input signal, but frequencies in the mid-range tend to work well for most applications.
- **Feedback:** The feedback parameter controls the amount of feedback applied to the signal, and is typically set between 0.1 and 0.9. Higher feedback values create a more pronounced effect, while lower values create a more subtle effect.

It's worth noting that the optimal ranges for these parameters will vary depending on the specific use case and the characteristics of the input signal. Experimenting with different parameter values can help you find the settings that work best for your specific application.

The next page is my code for “MyClass.h”, and a condensed version of the code inside PluginProcessor where you can see the connection between files and how the Phaser object parameters are linked to the ValueTreeState adjustable parameters.

```

86
87 MyClass myClass;
88
89 auto pPRate = std::make_unique<juce::AudioParameterFloat>("phaser rate", "Phaser Rate", 0.1, 10.0, 0.5);
90 auto pPDepth = std::make_unique<juce::AudioParameterFloat>("phaser depth", "Phaser Depth", 0.1, 0.9, 0.2);
91 auto pPFreq = std::make_unique<juce::AudioParameterFloat>("phaser frequency", "Phaser Frequency", 100.0, 2000.0, 1000.0);
92 auto pPFeedback = std::make_unique<juce::AudioParameterFloat>("phaser feedback", "Phaser Feedback", 0.1, 0.9, 0.9);
93 auto pMix = std::make_unique<juce::AudioParameterFloat>("mix", "Mix", 0.0, 100.0, 100.0);
94
95
96 if (parameterID == "phaser rate")
97 {
98     myClass.setRate(newValue);
99 }
100 if (parameterID == "phaser depth")
101 {
102     myClass.setDepth(newValue);
103 }
104 if (parameterID == "phaser frequency")
105 {
106     myClass.setFreq(newValue);
107 }
108 if (parameterID == "phaser feedback")
109 {
110     myClass.setFeedback(newValue);
111 }
112 if (parameterID == "mix")
113 {
114     myClass.setMix(newValue / 100.0);
115 }
116
117 float prate = (*treestate.getRawParameterValue("phaser rate"));
118 float pdepth = (*treestate.getRawParameterValue("phaser depth"));
119 float pfreq = (*treestate.getRawParameterValue("phaser frequency"));
120 float pfeedback = (*treestate.getRawParameterValue("phaser feedback"));
121 float mix = (*treestate.getRawParameterValue("mix") / 100.0);
122
123
124 myClass.setPhaserRate(prate);
125 myClass.setPhaserDepth(pdepth);
126 myClass.setPhaserFreq(pfreq);
127 myClass.setPhaserFeedback(pfeedback);
128 myClass.setMix(mix);
129
130

```

```

2
3 #include <JuceHeader.h>
4
5 // Declare a `juce::dsp::Phaser` object as a member variable in your class
6 class MyClass
7 {
8 public:
9     MyClass()
10     {
11         // Initialize the phaser with the desired parameters
12         phaser.setRate(0.5);
13         phaser.setDepth(0.2);
14         phaser.setCentreFrequency(1000.0);
15         phaser.setFeedback(0.9);
16     }
17
18     void prepare(const juce::dsp::ProcessSpec& spec)
19     {
20         phaser.reset();
21         phaser.prepare(spec);
22     }
23
24     // Process audio using the phaser
25     void processAudio(juce::AudioBuffer<float>& buffer)
26     {
27         juce::dsp::AudioBlock<float> block(buffer);
28         juce::dsp::ProcessContextReplacing<float> context(block);
29         phaser.process(context);
30     }
31
32     void setRate(float rate) { phaser.setRate(rate); }
33
34     void setDepth(float depth) { phaser.setDepth(depth); }
35
36     void setFreq(float freq) { phaser.setCentreFrequency(freq); }
37
38     void setFeedback(float feedback) { phaser.setFeedback(feedback); }
39
40     void setMix(float mix) { phaser.setMix(mix); }
41
42 private:
43     juce::dsp::Phaser<float> phaser;
44 };

```

Once I built the VST3 plugin and tested it out in Ableton Live, I checked the time and was astonished how fast I was able to get that done. It is a very simple plugin, but regardless ChatGPT took away a lot of the annoying grunt work of typing out basic header files, and got me right into the editing, and debugging of the project. I decided with my extra time that I saved that I was going to add some creativity to this by adding a Chorus effect and making a basic GUI where a ComboBox let you select between effects and show only the effect specific parameters.



Can you rewrite the header file where it uses an array to switch between processing the audio using `juce::dsp::Chorus` and `juce::dsp::Phaser`?

```
12 #pragma once
13 #include <JuceHeader.h>
14
15 class ChorusPhaserProcessor
16 {
17 public:
18     ChorusPhaserProcessor() : currentEffectIndex(0)
19     {
20         chorus.setRate(1.0f);
21         chorus.setDepth(0.2f);
22         chorus.setCentreDelay(20.0f);
23         chorus.setFeedback(0.1f);
24
25         phaser.setRate(0.5f);
26         phaser.setDepth(0.2f);
27         phaser.setCentreFrequency(1000.0f);
28         phaser.setFeedback(0.9f);
29     }
30
31     void prepareToPlay(double sampleRate, int samplesPerBlock)
32     {
33         juce::dsp::ProcessSpec spec;
34         spec.sampleRate = sampleRate;
35         spec.maximumBlockSize = samplesPerBlock;
36         spec.numChannels = 2;
37
38         chorus.prepare(spec);
39         phaser.prepare(spec);
40     }
41
42     void processAudio(juce::AudioBuffer<float>& buffer)
43     {
44         juce::dsp::AudioBlock<float> block(buffer);
45         juce::dsp::ProcessContextReplacing<float> context(block);
46
47         // Use an array to switch between the effects
48         juce::dsp::ProcessorChain<juce::dsp::Chorus<float>, juce::dsp::Phaser<float>> chain;
49         chain.get<0>().setRate(chorus.getRate());
50         chain.get<0>().setDepth(chorus.getDepth());
51         chain.get<0>().setCentreDelay(chorus.getCentreDelay());
52         chain.get<0>().setFeedback(chorus.getFeedback());
53         chain.get<1>().setRate(phaser.getRate());
54         chain.get<1>().setDepth(phaser.getDepth());
55         chain.get<1>().setCentreFrequency(phaser.getCentreFrequency());
56         chain.get<1>().setFeedback(phaser.getFeedback());
57
58         // Process the audio with the current effect
59         if (currentEffectIndex == 0)
60             chain.template process<0>(context);
61         else
62             chain.template process<1>(context);
63     }
64
65     void setEffect(int effectIndex)
66     {
67         currentEffectIndex = effectIndex;
68     }
69
70 private:
71     juce::dsp::Chorus<float> chorus;
72     juce::dsp::Phaser<float> phaser;
73     int currentEffectIndex;
74 };
```

In this example, we define a `ChorusPhaserProcessor` class that contains both a `juce::dsp::Chorus` and a `juce::dsp::Phaser` object, as well as an integer `currentEffectIndex` to keep track of which effect is currently selected.

In the constructor, we initialize the parameters of both effects to their desired values. In the `prepareToPlay()` function, we configure both effects for the specified sample rate and block size.

In the `processAudio()` function, we create a `juce::dsp::ProcessorChain` object that contains both the chorus and phaser effects. We set the parameters of each effect in the chain to match the current values of the corresponding objects. We then use the `currentEffectIndex` to determine which effect to process the audio with.

Finally, we define a `setEffect()` function that allows the caller to switch between the two effects by updating the `currentEffectIndex` member variable.

This is just an example implementation, and the specific details of how you implement a class that switches between `juce::dsp::Chorus` and `juce::dsp::Phaser` may

ChatGPT responded with another header file that uses a ProcessorChain with a Chorus and Phaser within and an integer value chooses which effect is processed. Now same as the Phaser, I need to create adjustable Chorus parameters with decent ranges in regard to the parameter within the PluginProcessor. I could easily do a simple internet search for common chorus parameter ranges or look into JUCE's documentation online, but instead I asked ChatGPT:



what are good parameter ranges for the `juce::dsp::Chorus` object?



Here are some good parameter ranges for the `juce::dsp::Chorus` object:



- `juce::dsp::Chorus<float>::setRate(float rate)`: This controls the rate of modulation of the delay time, in Hz. A good range for this parameter is typically between 0.1 and 10 Hz.
- `juce::dsp::Chorus<float>::setDepth(float depth)`: This controls the depth of modulation, or the maximum amount of delay time shift applied to the signal. A good range for this parameter is typically between 0.1 and 1.
- `juce::dsp::Chorus<float>::setCentreDelay(float delayMs)`: This controls the center delay time of the chorus, in milliseconds. A good range for this parameter is typically between 10 and 50 ms.
- `juce::dsp::Chorus<float>::setFeedback(float feedback)`: This controls the amount of feedback applied to the delayed signal. A good range for this parameter is typically between 0 and 0.9.

Note that the optimal parameter values can vary depending on the specific application and the desired sound. These are just general guidelines to get started with. Experimenting with different parameter values and listening to the result is often the best way to find the optimal settings for a particular use case.



Once these parameters were added and connected to the header file within the PluginProcessor, my focus shifted towards the GUI creation. So, I switched my GUI from the GenericAudioProcessorEditor to the projects PluginEditor, and asked ChatGPT to write up a compiler file for it:



Write me a PluginEditor.cpp file using a combobox to switch between a Chorus and Phaser while having only the effect specific controls showing in the GUI.

```
1  #include "PluginEditor.h"
2
3  PluginEditor::PluginEditor(PluginProcessor& p)
4      : AudioProcessorEditor(&p), processor(p)
5  {
6      // Create the ComboBox to switch between effects
7      effectSelector.addItem("Chorus", 1);
8      effectSelector.addItem("Phaser", 2);
9      effectSelector.setSelectedId(1);
10     effectSelector.addListener(this);
11     addAndMakeVisible(effectSelector);
12
13     // Set up the GUI for the Chorus effect
14     chorusGui = new ChorusGUI(processor.getChorusParameters());
15     addAndMakeVisible(chorusGui);
16     setChorusGuiVisible(true);
17
18     // Set up the GUI for the Phaser effect
19     phaserGui = new PhaserGUI(processor.getPhaserParameters());
20     addAndMakeVisible(phaserGui);
21     setPhaserGuiVisible(false);
22
23     setSize(400, 300);
24 }
25
26 PluginEditor::~PluginEditor()
27 {
28 }
29
30 void PluginEditor::paint(Graphics& g)
31 {
32     g.fillAll(getLookAndFeel().findColour(ResizableWindow::backgroundColourId));
33 }
34
35 void PluginEditor::resized()
36 {
37     effectSelector.setBounds(10, 10, 100, 20);
38     chorusGui->setBounds(10, 50, getWidth() - 20, getHeight() - 60);
39     phaserGui->setBounds(10, 50, getWidth() - 20, getHeight() - 60);
40 }
41
42 void PluginEditor::comboBoxChanged(ComboBox* comboBoxThatHasChanged)
43 {
44     // Show the Chorus GUI if Chorus is selected
45     if (comboBoxThatHasChanged == &effectSelector && effectSelector.getSelectedId() == 1)
46     {
47         setChorusGuiVisible(true);
48         setPhaserGuiVisible(false);
49     }
50     // Show the Phaser GUI if Phaser is selected
51     else if (comboBoxThatHasChanged == &effectSelector && effectSelector.getSelectedId() == 2)
52     {
53         setChorusGuiVisible(false);
54         setPhaserGuiVisible(true);
55     }
56 }
57
58 void PluginEditor::setChorusGuiVisible(bool isVisible)
59 {
60     chorusGui->setVisible(isVisible);
61 }
62
63 void PluginEditor::setPhaserGuiVisible(bool isVisible)
64 {
65     phaserGui->setVisible(isVisible);
66 }
```

In this example, the `ComboBox` is created in the `PluginEditor` constructor and assigned an ID of 1 for the `Chorus` effect and an ID of 2 for the `Phaser` effect. When the user selects an effect from the `ComboBox`, the `comboBoxChanged()` function is called, which shows the appropriate GUI and hides the other one.

The `ChorusGUI` and `PhaserGUI` classes represent the GUI components for the `Chorus` and `Phaser` effects, respectively. These classes are responsible for creating and managing the individual controls specific to each effect.

Note that this is just one possible implementation of this feature. The exact details may vary depending on the specific needs of the project.

With this new header file, I did a decent amount of editing within this file, but the logic still stayed the same. I decided not to create the `ChorusGUI` and `PhaserGUI` classes and just individually define the sliders and the attachments in the `PluginEditor`, but still toggle their visibility within the change of the `ComboBox` component.

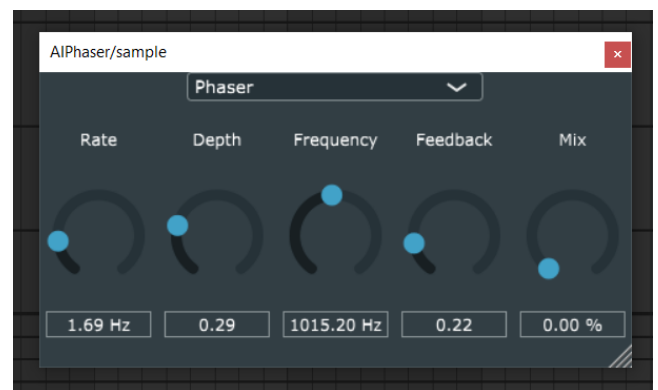
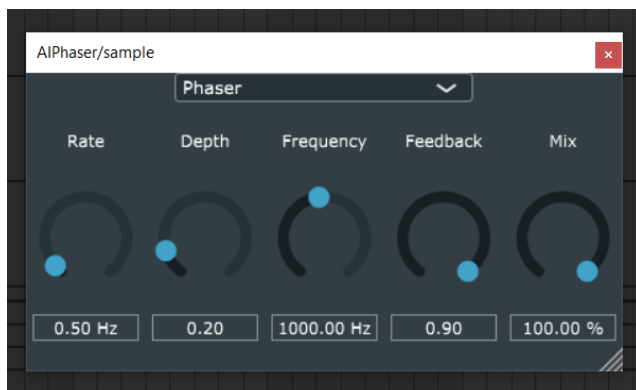
```
26 |  
27 |  
28 |     addAndMakeVisible(chorusrateSlider);  
29 |     chorusrateSlider.setTextValueSuffix(" Hz");  
30 |     chorusrateSlider.setTextBoxIsEditable(true);  
31 |  
32 |     addAndMakeVisible(chorusdepthSlider);  
33 |     chorusdepthSlider.setTextBoxIsEditable(true);  
34 |  
35 |     addAndMakeVisible(chorusfeedbackSlider);  
36 |     chorusfeedbackSlider.setTextBoxIsEditable(true);  
37 |  
38 |     addAndMakeVisible(chorusdelaySlider);  
39 |     chorusdelaySlider.setTextValueSuffix(" ms");  
40 |     chorusdelaySlider.setTextBoxIsEditable(true);  
41 |  
42 |     addAndMakeVisible(phaserrateSlider);  
43 |     phaserrateSlider.setTextValueSuffix(" Hz");  
44 |     phaserrateSlider.setTextBoxIsEditable(true);  
45 |  
46 |     addAndMakeVisible(phaserdepthSlider);  
47 |     phaserdepthSlider.setTextBoxIsEditable(true);  
48 |  
49 |     addAndMakeVisible(phaserfeedbackSlider);  
50 |     phaserfeedbackSlider.setTextBoxIsEditable(true);  
51 |  
52 |     addAndMakeVisible(phaserfreqSlider);  
53 |     phaserfreqSlider.setTextValueSuffix(" Hz");  
54 |     phaserfreqSlider.setTextBoxIsEditable(true);  
55 |  
56 |     addAndMakeVisible(effectbox);  
57 |     effectbox.addItem("Chorus", 1);  
58 |     effectbox.addItem("Phaser", 2);  
59 |
```

```

59 effectbox.onChange = [this]
60 {
61     if (effectbox.getSelectedId() == 1)
62     {
63         chorusrateSlider.setVisible(true);
64         phaserrateSlider.setVisible(false);
65         chorusdepthSlider.setVisible(true);
66         phaserdepthSlider.setVisible(false);
67         chorusdelaySlider.setVisible(true);
68         phaserfreqSlider.setVisible(false);
69         chorusfeedbackSlider.setVisible(true);
70         phaserfeedbackSlider.setVisible(false);
71     }
72
73     if (effectbox.getSelectedId() == 2)
74     {
75         chorusrateSlider.setVisible(false);
76         phaserrateSlider.setVisible(true);
77         chorusdepthSlider.setVisible(false);
78         phaserdepthSlider.setVisible(true);
79         chorusdelaySlider.setVisible(false);
80         phaserfreqSlider.setVisible(true);
81         chorusfeedbackSlider.setVisible(false);
82         phaserfeedbackSlider.setVisible(true);
83     }
84 }
85
86 effectbox.setSelectedId(1);
87 effectAttachment = std::make_unique<juce::AudioProcessorValueTreeState::ComboBoxAttachment>
88 (audioProcessor.treestate, "effect", effectbox);
89
90 addAndMakeVisible(mixSlider);
91 mixSlider.setTextValueSuffix(" %");
92 mixSlider.setTextBoxIsEditable(true);
93
94 setResizable(true, true);
95 setResizeLimits(300, 225, 600, 400);
96 getConstrainer()->setFixedAspectRatio(2.0);
97 setSize(400, 300);
98 }

```

The only parameter that stayed the same between effects was the mix, so I created a singular mix knob that just switches what effect it is a part of via the ComboBox index. Finally, my plugin was fully coded and operational. Here are a few screenshots of the plugin displaying its effect specific parameters. The main difference between a Chorus and a Phaser is that the Chorus modulates a delay time (ms) and a Phaser modulates a frequency (Hz). You can easily see how the middle knob parameter switches its units, but for the other parameters, the switch between effects doesn't change too much visually other than changing the ranges or changing the initial values.



The main objective of this project was to test out the AI's capability and efficiency when it comes to coding, and I believe that ChatGPT performed pretty well. I was able to create a simple Phaser in at least half the time it normally would take for me to do it, and it provided be the basic code to which I could add some more complexity and refinements. So with minimal editing and debugging, I got to my end goal faster and was gifted the ability to spend more time being creative within my projects. Overall, I had a blast making this plugin, and I definitely will be revisiting ChatGPT with future coding projects. All the code and the VST3 file are on my [GitHub account](#), if you are interested in using it or for a further dive into the code.

## Patrick Lacey

[Website](#) | [GitHub](#) | [Linktree](#)