Generate a plug-in with Project Generator

This tutorial explains how to create a new audio plug-in by using the ***VST 3 Project Generator*** included in the **VST 3 SDK** and how to add some basic features.

The artifact will be an audio plug-in that can compute a gain to an audio signal and can be loaded into VST3 hosts like **Cubase**, **WaveLab**, ...

On this page:

Part 1: Getting and installing the VST 3 SDK

For downloading the SDK, see the section " How to set up my system for VST 3".

You have the following possibilities to start a new project:

- You can use the **helloworld** template included in the **VST SDK** and duplicate the folder into a new folder. Adapt each file where the comment mentions it.
- Or, which is **easier** and **recommended**, you can use the **VST3 Project Generator** application included in the **VST SDK.** The following steps show how to use it.

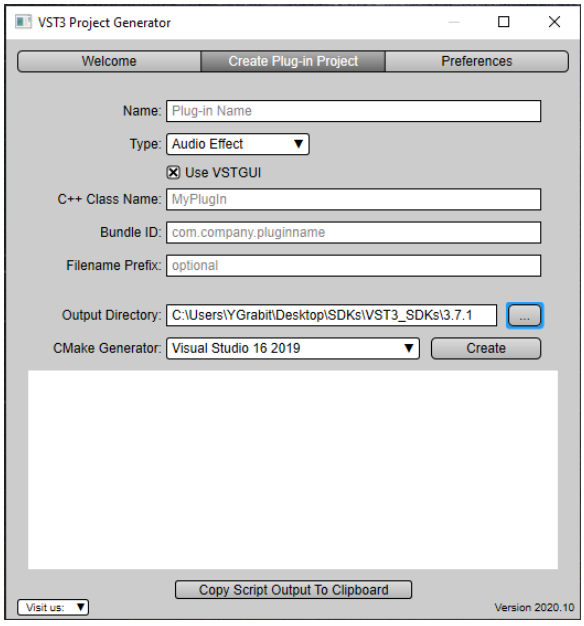
Part 2: Using the VST 3 plug-in **Project Generator** application

The **VST3 Project Generator** application included in the **VST SDK** is available for Windows and for macOS.

Start the application located in the *VST3_Project_Generator* folder of the **VST SDK**.

Check that the **Preferences** tab has the required information: see Setting the Preferences.

In the **Create Plug-in Project** tab you have to enter information about the plug-in that you want create:



Check the Create Plug-in Project tab of the VST 3 Project Generator dialog for more detailed documentation.

Once you have entered all information, click **Create.** A script is started which creates a project with updated files in the Output directory. After this step, the IDE (Visual Studio or XCode) is launched.

Compile the project and test your new plug-in. The plug-in is created in the *Output Directory*, in order to make it visible to a **VST 3** host you may have to copy or symbolic-link it to the official VST 3 Locations / Format.

For example, if you chose **Audio Effect** as Type, a simple StereoStereo plug-in is created.

A good way to understand how a **VST 3** plug-in works is to add breakpoints in each function in the processor and controller files:

```
tresult PLUGIN_API MyPluginController::initialize (FUnknown* context);
tresult PLUGIN_API MyPluginController::terminate ();
//...
tresult PLUGIN_API MyPluginProcessor::initialize (FUnknown* context);
//...
```

and start a **VST 3** host from the debugger.

Part 3: Coding your Plug-in

Now you have an automatically generated frame for your plug-in. The following sections explain how to add a new parameter, its associated processing algorithm, and other specific features like saving/loading project or presets, creating a dedicated user interface, etc.

A **VST 3** plug-in contains two main classes: its *PlugProcessor* (performing the processing and persistence) and its *PlugController* (taking care of communication with the DAW, handling parameters and the UI).

Add a parameter: Gain

In this basic plug-in example, we will add a Gain parameter which modifies the volume of the audio going through the plug-in.

For this, each **VST 3** parameter requires a unique identifier (a number).

1. Open the file plugids.h and enter a new id *kParamGainId.* In this example, assign the unique number 102.

plugids.h

```
#include "pluginterfaces/vst/vsttypes.h"

enum GainParams : Steinberg::Vst::ParamID
{
    kParamGainId = 102, // should be an unique id...
};
```

2. Open the plugcontroller.cpp file, and add the gain parameter with the *parameters.addParameter*

plugcontroller.cpp

```
#include "myplugincids.h"

//------------------------------------------------------------------------
----
tresult PLUGIN_API PlugController::initialize (FUnknown* context)
{
    tresult result = EditController::initialize (context);
    if (result != kResultOk)
        {
                return result;
        }

           //---Create Parameters------------
    parameters.addParameter (STR16 ("Gain"), STR16 ("dB"), 0, .5, Vst::
ParameterInfo::kCanAutomate, GainParams::kParamGainId, 0);

    return kResultTrue;
}
```

Note

- We add the flag *kCanAutomate* which informs the DAW/host that this parameter can be automated.
- A **VST 3** parameter is always normalized (its value is a floating point value between [0, 1]), here its default value is set to 0.5.

3. Now adapt the processor part for this new parameter. Open the file *plugprocessor.h* and add a gain value **Vst::ParamValue mGain**. This value is used for the processing to apply the gain.

plugprocessor.h

```
// ...
static FUnknown* createInstance (void*)
{
        return (Steinberg::Vst::IAudioProcessor*)new PlugProcessor ();
}
protected:
        Steinberg::Vst::ParamValue mGain = 1.;
// ...
```

Add the process applying the gain

1. We need to set our internal **mGain** with its required value from the host. This is the first step of the process method. Parse the parameter change coming from the host in the structure *data. inputParameterChanges* for the current audio block to process.

plugprocessor.cpp

```
#include "pluginterfaces/vst/ivstparameterchanges.h"
#include "public.sdk/source/vst/vstaudioprocessoralgo.h"

//------------------------------------------------------------------------
----
tresult PLUGIN_API PlugProcessor::process (Vst::ProcessData& data)
{
    //--- First : Read inputs parameter changes-----------
    if (data.inputParameterChanges)
    {
                // for each parameter defined by its ID
                int32 numParamsChanged = data.inputParameterChanges-
>getParameterCount ();
        for (int32 index = 0; index < numParamsChanged; index++)
        {
            // for this parameter we could iterate the list of value
changes (could 1 per audio block or more!)
                        // in this example we get only the last value
(getPointCount - 1)
                        Vst::IParamValueQueue* paramQueue = data.
inputParameterChanges->getParameterData (index);
            if (paramQueue)
            {
                Vst::ParamValue value;
                int32 sampleOffset;
                int32 numPoints = paramQueue->getPointCount ();
                switch (paramQueue->getParameterId ())
                {
                    case GainParams::kParamGainId:
                        if (paramQueue->getPoint (numPoints - 1,
sampleOffset, value) == kResultTrue)
                            mGain = value;
                        break;
                                }
                 }
                }
        }
        // ....
}
```

data.inputParameterChanges can include more than **1** change for the same parameter inside a processing audio block. Here we take only the last change in the list and apply it our **m Gain**.

2. The real processing part:

plugprocessor.cpp

```
	// ...

	//-- Flush case: we only need to update parameter, no processing
possible
	if (data.numInputs == 0 || data.numSamples == 0)
		return kResultOk;

	//--- Here you have to implement your processing
	int32 numChannels = data.inputs[0].numChannels;

	//---get audio buffers----------------
	void** in = getChannelBuffersPointer (processSetup, data.inputs
[0]);
	void** out = getChannelBuffersPointer (processSetup, data.outputs
[0]);

	// Here could check the silent flags
	// now we will produce the output
	// mark our outputs has not silent
	data.outputs[0].silenceFlags = 0;

	float gain = mGain;
	// for each channel (left and right)
	for (int32 i = 0; i < numChannels; i++)
	{
		int32 samples = data.numSamples;
		Vst::Sample32* ptrIn = (Vst::Sample32*)in[i];
		Vst::Sample32* ptrOut = (Vst::Sample32*)out[i];
		Vst::Sample32 tmp;
		// for each sample in this channel
		while (--samples >= 0)
		{
			// apply gain
			tmp = (*ptrIn++) * gain;
			(*ptrOut++) = tmp;
		}
	}
	//...
```

3. **VST 3** includes a way for the host to inform the plug-in that its inputs are silent (using the **VST 3** silence flags):

plugprocessor.cpp

```
        // Here could check the silent flags
    //---check if silence---------------
        // normally we have to check each channel (simplification)
        if (data.inputs[0].silenceFlags != 0)
        {
                // mark output silence too
                data.outputs[0].silenceFlags = data.inputs[0].silenceFlags;

                // the Plug-in has to be sure that if it sets the flags
silence that the output buffer are clear
                for (int32 i = 0; i < numChannels; i++)
                {
                         // do not need to be cleared if the buffers are
the same (in this case input buffer are
                        // already cleared by the host)
                        if (in[i] != out[i])
                        {
                                memset (out[i], 0, sampleFramesSize);
                        }
                }
                // nothing to do at this point
                return kResultOk;
    }
```

Add store/restore state

The *Processor* part represents the state of the plug-in, so it is its job to implement the **getState**/**setState** method used by the host to save/load projects and presets.

1. In the file *plugprocessor.cpp,* add the **mGain** value to the state stream given by the host in the **getState** method which will save it as a project or preset.

plugprocessor.cpp

```
//------------------------------------------------------------------------
tresult PLUGIN_API PlugProcessor::getState (IBStream* state)
{
        // here we need to save the model (preset or project)
        float toSaveParam1 = mGain;
        IBStreamer streamer (state, kLittleEndian);
        streamer.writeFloat (toSaveParam1);
        return kResultOk;
}
```

2. In the **setState** () method, the plug-in receives a new state (called after a project or preset is loaded) from the host.

plugprocessor.cpp

```
//------------------------------------------------------------------------
tresult PLUGIN_API PlugProcessor::setState (IBStream* state)
{
	if (!state)
		return kResultFalse;
	// called when we load a preset or project, the model has to be
reloaded
	IBStreamer streamer (state, kLittleEndian);
	float savedParam1 = 0.f;
	if (streamer.readFloat (savedParam1) == false)
		return kResultFalse;
	mGain = savedParam1;

	return kResultOk;
}
```

Part 4: Advanced Steps

Add an Event Input

In our example we want to modify our current Gain factor with the velocity of a played "MIDI" event (noteOn).

1. If you need in your plug-in to receive not only audio but events (like MIDI), you have to add an Event input. For this you just have to add in , in order to do this call in the **initialize()** method of the processor addEventInput:

plugprocessor.cpp

```
//------------------------------------------------------------------------
tresult PLUGIN_API PlugProcessor::initialize (FUnknown* context)
{
	//---always initialize the parent-------
	tresult result = AudioEffect::initialize (context);
	// if everything Ok, continue
	if (result != kResultOk)
	{
		return result;
	}

	//....

	//---create Event In/Out busses (1 bus with only 1 channel)------
	addEventInput (STR16 ("Event In"), 1);

	return kResultOk;
}
```

In this example we add 1 input event bus, receiving only on 1 channel. If you need to receive differentiated events, for example, from different channels, just change it like this:

addEventInput (STR16 ("Event In"), 4); // here 4 channels

2. We create a new internal value mGainReduction (not exported to the host) which is changed by the velocity of a played noteOn, so that the harder you hit the note, the higher is the gain reduction (this is what we want here):

plugprocessor.h

```
// ...
static FUnknown* createInstance (void*)
{
        return (Steinberg::Vst::IAudioProcessor*)new PlugProcessor ();
}
protected:
        Steinberg::Vst::ParamValue mGain= 1.;
        Steinberg::Vst::ParamValue mGainReduction = 0.;

// ...
```

3. Now we have to receive the event changes in the process method:

plugprocessor.cpp

```
//------------------------------------------------------------------------
tresult PLUGIN_API PlugProcessor::process (ProcessData& data)
{
        //--- First : Read inputs parameter changes-----------
        //...

        //---Second : Read input events-------------
        // get the list of all event changes
        Vst::IEventList* eventList = data.inputEvents;
        if (eventList)
        {
                int32 numEvent = eventList->getEventCount ();
                for (int32 i = 0; i < numEvent; i++)
                {
                        Vst::Event event;
                        if (eventList->getEvent (i, event) == kResultOk)
                        {
                                // here we do not take care of the channel
info of the event
                                switch (event.type)
                                {
                                        //----------------------
                                        case Vst::Event::kNoteOnEvent:
                                                // use the velocity as
gain modifier: a velocity max (1) will lead to silent audio
                                                mGainReduction = event.
noteOn.velocity; // value between 0 and 1
                                                break;

                                        //----------------------
                                        case Vst::Event::kNoteOffEvent:
                                                // noteOff reset the gain
modifier
                                                mGainReduction = 0.f;
                                                break;
                                }
                        }
                }
        }
```

4. Make use of this mGainReduction in our real processing part:

plugprocessor.cpp

```
//------------------------------------------------------------------------
----
tresult PLUGIN_API PlugProcessor::process (Vst::ProcessData& data)
{
        //....

        float gain = mGain - mGainReduction;
        if (gain < 0.f)        // gain should always positive or zero
                gain = 0.f;

        // for each channel (left and right)
        for (int32 i = 0; i < numChannels; i++)
        {
                int32 samples = data.numSamples;
                Vst::Sample32* ptrIn = (Vst::Sample32*)in[i];
                Vst::Sample32* ptrOut = (Vst::Sample32*)out[i];
                Vst::Sample32 tmp;
                // for each sample in this channel
                while (--samples >= 0)
                {
                        // apply gain
                        tmp = (*ptrIn++) * gain;
                        (*ptrOut++) = tmp;
                }
        }
        //...
}
```

Add a mono audio Side-chain

In our example we want to modulate our main audio input with a side-chain audio input.

1. First add a new side-chain audio input (busType: ***kAux***) in the initialize call of our processor:

plugprocessor.cpp

```
//------------------------------------------------------------------------
tresult PLUGIN_API PlugProcessor::initialize (FUnknown* context)
{
        //---always initialize the parent-------
        tresult result = AudioEffect::initialize (context);
        // if everything Ok, continue
        if (result != kResultOk)
        {
                return result;
        }

        //....

        //---create Event In/Out busses (1 bus with only 1 channel)------
        addEventInput (STR16 ("Event In"), 1);

        // create a Mono SideChain input bus
        addAudioInput (STR16 ("Mono Aux In"), Steinberg::Vst::SpeakerArr::
kMono, Steinberg::Vst::kAux, 0);

        return kResultOk;
}
```

2. We want to be sure that our side-chain is handled as mono input. For this we need to overwrite the AudioEffect::setBusArrangements function:

plugprocessor.h

```
//------------------------------------------------------------------------
class PlugProcessor: public AudioEffect
{
public:
        PlugProcessor();

        //...
        // overwrite this function
        Steinberg::tresult PLUGIN_API setBusArrangements (Steinberg::Vst::
SpeakerArrangement* inputs,
                                                          Steinberg::int32
numIns,
                                                          Steinberg::Vst::
SpeakerArrangement* outputs,
                                                          Steinberg::int32
numOuts) SMTG_OVERRIDE;
        //...
};
```

plugprocessor.cpp

```
//------------------------------------------------------------------------
tresult PLUGIN_API PlugProcessor::setBusArrangements (Vst::
SpeakerArrangement* inputs, int32 numIns,
                                                      Vst::
SpeakerArrangement* outputs,
                                                      int32 numOuts)
{
        // the first input is the Main Input and the second is the
SideChain Input
        // be sure that we have 2 inputs and 1 output
        if (numIns == 2 && numOuts == 1)
        {
                // we support only when Main input has the same number of
channel than the output
                if (Vst::SpeakerArr::getChannelCount (inputs[0]) != Vst::
SpeakerArr::getChannelCount (outputs[0]))
                        return kResultFalse;

                // we are agree with all arrangement for Main Input and
output
                // then apply them
                getAudioInput (0)->setArrangement (inputs[0]);
                getAudioOutput (0)->setArrangement (outputs[0]);

                // Now check if sidechain is mono (we support in our
example only mono Side-chain)
                if (Vst::SpeakerArr::getChannelCount (inputs[1]) != 1)
                        return kResultFalse;

                // OK the Side-chain is mono, we accept this by returning
kResultTrue
                return kResultTrue;
        }

        // we do not accept what the host wants : return kResultFalse !
        return kResultFalse;
}
```

3. Adapt our process using the side-chain input as modulation:

```
//------------------------------------------------------------------------
tresult PLUGIN_API PlugProcessor::process (ProcessData& data)
{
	//--- First : Read inputs parameter changes-----------
	//...

	//---Second : Read input events-------------
	//...

	float gain = mGain - mGainReduction;
	if (gain < 0.f)		// gain should always positive or zero
		gain = 0.f;

	void** auxIn = nullptr;

	// Check if the Side-chain input is activated
	bool auxActive = false;
	if (getAudioInput (1)->isActive ())
	{
		auxIn = getChannelBuffersPointer (processSetup, data.inputs
[1]);
		auxActive = true;
	}
	if (auxActive)
	{
		// for each channel (left and right)
		for (int32 i = 0; i < numChannels; i++)
		{
			int32 samples = data.numSamples;
			Vst::Sample32* ptrIn = (Vst::Sample32*)in[i];
			Vst::Sample32* ptrOut = (Vst::Sample32*)out[i];
			// Side-chain is mono, so take auxIn[0]: index 0
			Vst::Sample32* ptrAux = (Vst::Sample32*)auxIn[0];
			Vst::Sample32 tmp;

			// for each sample in this channel
			while (--samples >= 0)
			{
				// apply modulation and gain
				tmp = (*ptrIn++) * (*ptrAux++) * gain;
				(*ptrOut++) = tmp;
			}
		}
	}
	else
	{
		// for each channel (left and right)
		for (int32 i = 0; i < numChannels; i++)
		{
			int32 samples = data.numSamples;
			Vst::Sample32* ptrIn = (Vst::Sample32*)in[i];
			Vst::Sample32* ptrOut = (Vst::Sample32*)out[i];
			Vst::Sample32 tmp;
			// for each sample in this channel
			while (--samples >= 0)
			{
				// apply gain
				tmp = (*ptrIn++) * gain;
				(*ptrOut++) = tmp;
			}
		}
	}
```

That´s it!