Designing Side-chaining PlugIns with RackAFX
Will Pirkle

The ability to design and port side-chaining plugins is new in RackAFX v6.6.1. This document explains how to enable and use side-chaining in your projects, either old pre-v6.6.1 or new projects. RackAFX will also port the side-chaining capability to VST3 and AU projects using the <Make VST> and <Make AU> features. And, the side-chaining is also implemented in the RAFX-DLL-as-Native-VST paradigm.

The RackAFX Aux Input Bus

In order to implement side-chaining, I added a second audio input bus to the plugin object, known as an Aux Input Bus. This is the same way the VST3 and AU APIs implement side-chaining. The side-chaining Aux Input Bus is indexed with the number 1. Aux Input 0 is reserved (it is the main input bus in RackAFX) and in the future I may add more input busses as needed. RackAFX will call a new function named ***processAuxInputBus( )*** prior to calling *processAudioFrame( )* or *processRackAFXAudioBuffer( )* or *processVSTAudioBuffer()* depending on which process method you prefer — most will be using *processAudioFrame( )*.

- RackAFX will call this new function once for each aux input bus
- you will grab pointers to the aux input buffers in *processAuxInputBuffer( )*
- you will use the pointers in your *process…( )* function to apply the side-chain to your plugin
- currently side-chaining is only available for FX plugins, and only using the WAV file input source

In order to implement side-chaining, you need to do a couple of things:

1. for older projects, you need to add *processAuxInputBus( )* to your .h and .cpp files — for new projects, the function is already there for you to populate with your code

2. you will need to declare pointers to the aux input bus buffer and the type of pointer you declare depends on which process( ) function you are using in RackAFX

Step 1: Declare your Aux Input buffer pointers

The type of pointer you need to declare depends on which *process…( )* method you are using.

processAudioFrame( )
In this function, your FX audio input is a frame of data (one sample-period's worth) and arrives via a floating point buffer, *pInputBuffer* and you pick up either one or two samples from it (mono or stereo) by accessing pInputBuffer[0] and pInputBuffer[1] respectively. You declare your Aux Input buffer pointers the same way as *pInputBuffer* is declared, for example:

float* m_pSidechainFrameBuffer;

You then acquire the side-chain samples the same way as the audio input, by accessing m_pSidechainFrameBuffer[0] and m_pSidechainFrameBuffer[1] in your processAudioFrame function.

processRackAFXAudioBuffer( )
In this function, your FX audio input is a complete audio buffer delivered as-is from RackAFX and arrives via a floating point buffer, *pInputBuffer.* For stereo inputs, the samples are interleaved L/R/L/R and it is up to you to de-interleave the samples for stereo inputs. You declare your Aux Input buffer pointers the same way as *pInputBuffer* is declared, for example:

float* m_pSidechainRAFXBuffer;

You then acquire the side-chain samples the same way as the audio input, de-interleaving the samples for stereo side-chain inputs.

processVSTAudioBuffer( )
In this function, your FX audio input consists of a float double-pointer *ppInputs* — a pointer to a buffer of pointers. Each of the sub-pointers is a buffer for one channel. For stereo inputs, the first pointer is the left channel and the second pointer is the right channel. In this case, RackAFX (or your VST/AU host) de-interleaves the samples for you and places each in its own buffer. You declare your Aux Input buffer pointers the same way as *ppInputs* is declared, for example:

float m_ppSidechainVSTBuffer;**

You then acquire the side-chain samples the same way as the audio input, each channel arrives in its own buffer.

More helper variables
You also need to declare a couple of extra helper variables so your plugin knows the channel count of the side-chain input and whether the bus is enabled or not. Add these variables to your plugin's .h file along with the buffer pointers above:

bool m_bSidechainEnabled;
UINT m_uSidechainChannelCount;

So, that's it - you need three extra variables in your .h file so go ahead and initialize them in your constructor:

- set the side-chain pointer to NULL
- set the side-chain channel count to 0
- disable the sods-chain flag

For the accompanying plugin example, I show how to use all three types, so my constructor looks like this:

```
CVolumeWithSidechain::CVolumeWithSidechain()
{
	<SNIP SNIP SNIP>

	// Finish initializations here
	m_pSidechainFrameBuffer = NULL;
	m_pSidechainRAFXBuffer = NULL;
	m_ppSidechainVSTBuffer = NULL;

	m_bSidechainEnabled = false;
	m_uSidechainChannelCount = 0;
}
```

Step 2: Understanding/Implementing the *processAuxInputBus()* method

For older projects, you need to add the function prototype to your .h file and implement it in your .cpp file. The prototype is:

```
// --- process aux inputs
virtual bool __stdcall processAuxInputBus(audioProcessData* pAudioProcessData);
```

The function's input variable is a pointer to an *audioProcessData* structure. This structure is defined in *pluginconstants.h.*

```
typedef struct
{
	// --- bus index
	UINT uInputBus; // sidechain input = 1; others may follow

	// --- pointers for the three process types
	float* pFrameInputBuffer;
	float* pRAFXInputBuffer;
	float** ppVSTInputBuffer;
	UINT uNumInputChannels; // number of input channels, 1 or 2
	UINT uBufferSize; // RAFX buffer only, generally you won't need this

	bool bInputEnabled;

}audioProcessData;
```

The structure's member variables are easy to decode and all but two of them will map to the variables you just declared in the .h file. When you implement the processAuxInputBus( ) function you will need to:

- check the input bus index value - we only respond to Bus 1 for side-chain input
- pick up and store the particular pointer you need, for example if you use processAudioFrame( ) you would need to pick up the *pFrameInputBuffer* ; the others are easy to decode
- store the channel count
- store the input-enabled flag
- the uBufferSize variable is for processRackAFXAudioBuffer( ) users - however, it is guaranteed to be identical to the buffer size of the input/output buffers for this function
- **if the side-chain is disabled, the buffer pointers will be NULL and the input channel count will be 0 so it is always important to check them!**

My example plugin looks like this:

```
// --- process aux inputs (currently sidechain only)
bool __stdcall CVolumeWithSidechain::processAuxInputBus(audioProcessData*
                                                        pAudioProcessData)
{
	// --- store sidechain buffer pointers for use in process( ) methods
	//
	//    Input Bus 1 = Sidechain
	//
	if(pAudioProcessData->uInputBus == 1)
	{
		// --- save various pointers, in practice you only save the
		//     pointer you need for your particular process( ) function
```

```
            m_pSidechainFrameBuffer = pAudioProcessData->pFrameInputBuffer;
            m_pSidechainRAFXBuffer = pAudioProcessData->pRAFXInputBuffer;
            m_ppSidechainVSTBuffer = pAudioProcessData->ppVSTInputBuffer;

            // --- sidechain input activation/channels
            m_bSidechainEnabled = pAudioProcessData->bInputEnabled;
            m_uSidechainChannelCount = pAudioProcessData->uNumInputChannels;
        }

        return true;
}
```

Step 3: Use the pointers in your *process…( )* method

Lastly, you need to use the pointers and that will depend on the type of process( ) method you are using.

processAudioFrame( )

You acquire the side-chain samples the same way as the audio input, by accessing m_pSidechainFrameBuffer[0] and m_pSidechainFrameBuffer[1] in your processAudioFrame function, for example:

```
        float fSideChainLeft = 0;
        float fSideChainRight = 0;

        // do we have an active side-chain and valid buffer?
        if(m_bSidechainEnabled && m_pSidechainFrameBuffer)
        {
            // left sample
            fSideChainLeft  = m_pSidechainFrameBuffer[0];

            // right sample
            if(m_uSidechainChannelCount == 2)
                fSideChainRight  = m_pSidechainFrameBuffer[1];

            // use the side-chain…
        }
```

processRackAFXAudioBuffer( )

You acquire the side-chain samples the same way as the audio input, de-interleaving the samples for stereo side-chain inputs, if your plugin requires it. In this snippet, you can see how to de-interleave the buffer for a stereo side-chain:

```
        float fSideChainLeft = 0;
        float fSideChainRight = 0;

        // do we have an active side-chain and valid buffer?
        if(m_bSidechainEnabled && m_pSidechainRAFXBuffer)
        {
            // left sample
            if(i%2 == 0)
            {
                fSideChainLeft = m_pSidechainRAFXBuffer[i];
```

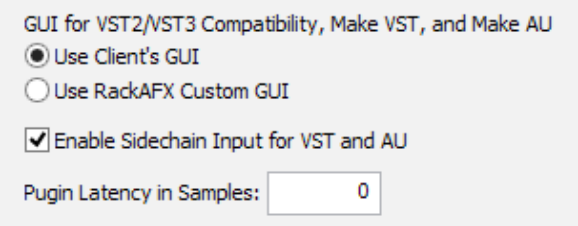
```
        }
        else
            fSideChainRight = m_pSidechainRAFXBuffer[i];

        // use the side-chain…
}
```

processVSTAudioBuffer( )
You acquire the side-chain samples the same way as the audio input, each channel arrives in its own buffer and you are delivered the double-pointer like the input and outputs.

```
// --- sidechain pointers
float* inSidechain1 = NULL;
float* inSidechain2 = NULL;

// --- sidechain samples
float fLeftSidechainSample = 0 ;
float fRightSidechainSample = 0 ;

// --- grab the buffer pointers
if(m_bSidechainEnabled && m_ppSidechainVSTBuffer)
{
    inSidechain1 = m_ppSidechainVSTBuffer[0];
    if(m_uSidechainChannelCount == 2)
        inSidechain2 = m_ppSidechainVSTBuffer[1];
}

// then use them in the processing loop
while (--inFramesToProcess >= 0)
{
    if(m_bSidechainEnabled && m_ppSidechainVSTBuffer)
    {
        // --- get left side-chain sample
        fLeftSidechainSample = *inSidechain1++;

        // --- get left side-chain sample
        if(m_uSidechainChannelCount == 2)
            fRightSidechainSample = *inSidechain2++;

        // use the side-chain…
    }

etc…
```

Step 4: Optional: add side-chaining to your VST and AU Ports

You don't need to add any additional code in your *Make VST* and *Make AU* ports. However, you do need to enable the aux input and this is done with *File->Edit Project in* RackAFX - just check the box to enable the side-chain in the ports:

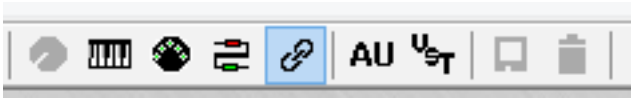


For VST3 ports, you will see either a side-chain button on the GUI toolbar (Cubase) or you will see additional audio input channels (Reaper), or you will see the side-chain input show up in the track routing box (Ableton Live). For AU ports, you will see either a side-chain drop-down list on the GUI toolbar (Logic) or you will see the side-chain input show up in the track routing box (Ableton Live).

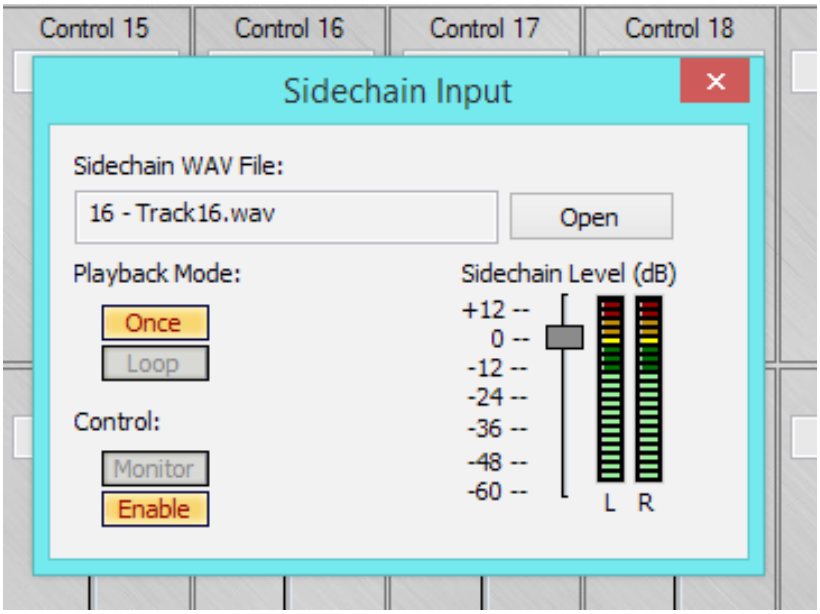
NOTE: side-chaining is not available in the VST2 version of the *Make VST* projects; this is because the built-in Stienberg VST3-to-VST2 wrapper *does not support side-chaining*!

Using the Side-Chain Input in RackAFX

Using the side-chain input is really easy in RackAFX. Open the *Sidechain Input* dialog with either *View->Sidechain Control* or with the toolbar button that looks like chain-links:



In the dialog box (which is mode-less and floats), you choose your WAV file to use as the side-chain input and the playback options (one-shot or loop). You also enable and disable the side-chain bus from this panel. Finally, the monitor button lets you route the side-chain bus to the output bus (without any processing) so you can hear the side-chain if needed. The level control can be used to tweak the volume of the side-chain.



You can turn the side-chain on and off, and you can also change the playback mode during plugin operation with audio streaming!

The VolumeWithSidechain Sample Project

I made a really simple demo plugin to show you how to use the side-chain input regardless of which *process…( )* method you choose to use. This project is a volume-in-dB plugin with a side-chain input. When you activate the side-chain, the operation changes and the side-chain signal is routed to the outputs, without any volume control. This was the easiest way for me to demo how to pick up the left and right side-chain samples. You can comment and un-comment the last view lines of the constructor to change the *process…( )* functions. The code for picking up the side-chain samples is purposefully verbose to show exactly how to get at those input samples.