

# Tremolo VST Plugin

Bachelors Degree Senior Project 2003  
Chris Larsen

## Table of Contents

Introduction	3
Design	4
Equation Translation	6
Algorithm	7
Tools	8
Example Code Walkthrough	8
AGain.hpp	9
AGain.cpp	11
Tremolo Code Walkthrough	16
Tremolo.hpp	16
Tremolo.cpp	17
Results	21
Conclusion	24
End Notes	25
Appendix A	
Futher Reading and Resources	
Appendix B	
AGain.hpp	
AGain.cpp	
Appendix C	
Tremolo.hpp	
Tremolo.cpp	
Appendix D	
Audio CD Track list	

## Introduction

Computers and digital processing have become an inextricable part of music recording and production. Digital technology began replacing analog technologies with the advent of digital signal processing. Delays that used an infinite looping tape with the playback head positioned before the record head were replaced with digital rack mount boxes that sampled the audio and stored each sample until the proper delay time was reached and it could send the sample to the output of the device. As DSP techniques advanced more analog processes were converted to digital such as optical based compressors and acoustic room reverbs. Eventually every portion of the audio signal path was engineered with digital processing except for the acoustic to electronic transduction at the microphone pickup which was then fed immediately through an analog to digital converter within the microphone itself.

Digital systems were often smaller in size, offered greater precision with their controls, and settings could be stored and recalled quickly. Early digital sound quality was somewhat lacking when compared to traditional analog techniques but with faster processing speeds and greater storage space, quality improved a great deal.

Today engineers can replace almost the entire studio with a speedy computer, a sound card, and some microphones. The computer can act as a mixer, processor, and music storage space to allow the user to record, mix, and master a project with incredible ease. Computer based systems are often less expensive than their dedicated counterparts. A variety of audio input/output cards are available with a wide range of sampling frequencies and digital or analog options. Additional storage space is simply a matter of plugging in a hard drive or two and a finished product can be burned to a CD and sent to the replicating plant.

However a computer can not perform any of these tasks unless it is provided with the instructions necessary to accomplish them in the form of software. The software code is loaded into the computers memory when a program is launched and it behaves according to the directions of the user. Multiple audio host applications can be installed on a machine for different purposes. The host application is responsible for retrieving audio from the sound card, performing processing such as changing the gain or mixing it with other audio sources, storing the audio on the hard drive, and sending audio back to the sound card for monitoring. The host also must provide an interface for the engineer to edit the sampled audio and perform any functions necessary to reach a final product.

Most host applications handle the important audio input, editing, and output functions but they leave effects and signal processing up to another program that can be incorporated into the host in the form of a plugin. An audio plugin is a small program that performs only one digital signal processing function such as reverberation, delay, or compression. The host application loads the plugin and then passes audio samples to it. The plugin then performs its calculations on the received samples and sends the audio back to the host. With a dedicated outboard processor you only have one unit to work with. However in a host application you can run as many copies of a particular plugin as your computer processor will allow. This flexibility offers the engineer much more freedom to be creative and achieve the desired effect.

As the culmination of a senior project to fulfill a bachelors degree program in audio engineering, this paper will present the design process used to create an audio processing plugin that creates a tremolo sound effect when it is used by a host application on some form of signal. Our design parameters require that the tremolo plugin accept a stereo input signal from the host and return a stereo output signal. The plugin must also provide an interface that allows the user to edit the frequency and depth of the effect. The tremolo effect should return the source signal when the user configures the settings for zero effect and it should not introduce any unwanted artifacts into the audio signal at any time. Finally the plugin should sound “good” when it is performing its tasks. Our ultimate goal is that the plugin is of such a quality that it could be used in a professional mix.

## Design

The classic tremolo effect is accomplished by varying the volume (or amplitude) of a sound over time periodically. For example a guitarist might use a tremolo pedal that increases the volume of their guitar as they step on it and decreases it as they let pressure off of the pedal. If they apply pressure and let up quickly and repeat the process, they will produce a tremolo effect. Vibraphones have a small rotating disc at the top of each note pipe that opens and closes an air path very quickly depending on how fast the motor is set to spin. As the top of the pipe is closed off, the volume of the note decreases. When the pipe starts to open again, the volume increases. In each of the previous examples the periodic variation of the amplitude is sinusoidal.

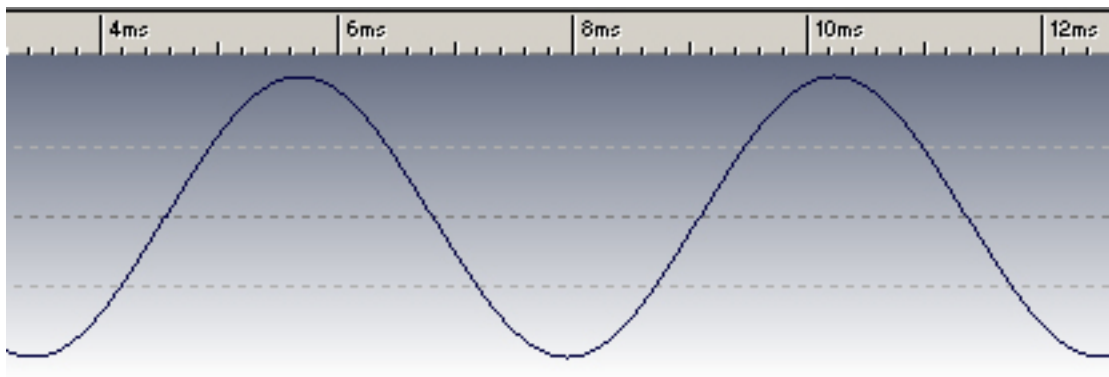


Figure 1

Because the amplitude of the signal is changing periodically over time we have a type of amplitude modulation. AM is usually used for communications allowing equipment to transmit telemetry data or radio stations to broadcast music to a wide audience. An AM signal can be produced by using the instantaneous amplitude of the information signal (the baseband or modulating signal) to vary the peak amplitude of a higher-frequency signal.<sup>1</sup> The higher-frequency signal is called the carrier frequency and it is usually a constant frequency sinusoidal signal. The modulating signal is often a complex waveform, in radio an audio signal. When the peaks of the carrier and modulating signal are joined, a symmetrical envelope is produced that “looks” like the

modulating signal with the carrier contained within the envelope when the signal is viewed on an oscilloscope.

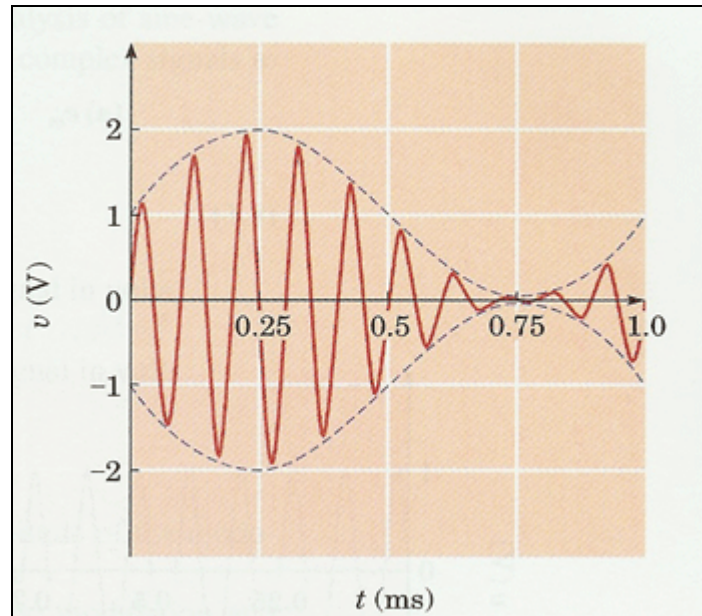


Figure 2<sup>ii</sup>

AM is not the linear addition of the carrier and modulating signals. Rather the instantaneous baseband amplitude is added to the peak carrier amplitude to produce the envelope<sup>iii</sup>.

In mathematical terms using signal voltages we can express the AM process as:

$$v(t) = (E_c + e_m) \sin \omega_c t \quad (\text{Eq. 1})$$

Where  $v(t)$  is the instantaneous amplitude of the modulated signal in volts,  $E_c$  is the peak amplitude of the carrier in volts,  $e_m$  is the instantaneous amplitude of the modulating signal in volts,  $\omega_c$  is the radian frequency of the carrier, and  $t$  is the time in seconds. If we assume that the modulating signal is a sine wave then we can use Equation 2.

$$v(t) = (E_c + E_m \sin \omega_m t) \sin \omega_c t \quad (\text{Eq. 2})$$

Where  $E_m$  is the peak amplitude of the modulating signal in volts and  $\omega_m$  is the radian frequency of the modulating signal.

Equation 2 provides us with an amplitude modulated signal modulated at 100%. This means that at negative peaks of the envelope the carrier is completely suppressed and we have 0% signal output. In audio processing it may not be desirable to completely suppress the signal. Instead a user may simply want to vary the output amplitude from 100% of the source signal to 50%. Introducing the modulation index,  $m$ , will allow for the adjustment of depth of modulation. The modulation index is the ratio between the modulating and carrier signals, expressed mathematically as:

$$m = E_m / E_c \quad (\text{Eq. 3})$$

Substituting  $m$  into Equation 2 returns:

$$v(t) = E_c (1 + m \sin \omega_m t) \sin \omega_c t \quad (\text{Eq. 4})$$

$m$  must never be greater than 1 or else the envelope will not represent the modulating signal and the output signal will become distorted. Equation 4 is the algebraic equation useful for creating a tremolo effect.

## Equation Translation

The next step in creating the tremolo plugin involves translating Equation 4 into C++ source code. C++ functions require specific operand and variables that may be implied in an algebraic equation or used for a different purpose. Functions also require variables to store and retrieve data and actual values must be used in the place of constants. The first step of translation is to resolve any implicit operations. We will have to break  $\omega$  into its component parts.

$$\omega = 2\pi f \quad (\text{Eq. 5})$$

Where  $f$  is the frequency in hertz. Substituting  $\omega$  into equation 4 gives us:

$$v(t) = E_c (1 + m \sin (2\pi f)_m t) \sin (2\pi f)_c t \quad (\text{Eq. 6})$$

Next we need to insert arithmetic operators where they are implied and substitute numeric values for our constants, in this case  $\pi$ .

$$v(t) = E_c * (1 + m * (\sin (2 * 3.14 * f_m) * t)) * (\sin (2 * 3.14 * f_c) * t) \quad (\text{Eq. 7})$$

The  $*$  operand stands for multiplication in C++ and operations within parentheses are calculated first as they are in algebra. The `sin` keyword cannot be used in our function unless the standard `Cmath` library header file is included in the source code. `Cmath` contains a number of useful functions that we can call to perform calculations without having to code the implicit steps by hand. Finally we can substitute descriptive variable names that will make the code easier to read.

$$\text{Output} = \text{CarrierAmplitude} * (1 + \text{Depth} * (\sin (2 * 3.14 * \text{ModulationFrequency} * \text{Time})) * (\sin (2 * 3.14 * \text{CarrierFrequency} * \text{Time})) \quad (\text{Eq. 8})$$

Equation 8 is now ready to be plugged into C++ source code. However for our final tremolo equation we will need to make a few adjustments to compensate for overmodulation and complex waveforms. Our final tremolo equation appears as:

$$\text{modifier} = (1.0 - \text{fDepth}) + \text{fDepth} * (\sin((2 * 3.14 * (\text{fFrequency} * 10)) * (\text{counter}/\text{sampleRate}))) * (\sin((2 * 3.14 * (\text{fFrequency} * 10)) * (\text{counter}/\text{sampleRate})));$$

(Eq.9)

Equation 9 is missing the Carrier Amplitude because in our actual code we will be calculating a modifying value and applying that to the audio samples. This saves processing time so that the computer doesn't have to perform this large calculation twice for a stereo audio signal. The sample data is the carrier amplitude in this situation. With traditional AM broadcasting the carrier is a set frequency and it is modulated with the audio signal. In our plugin the audio is the carrier with a complex waveform and no predefined frequency and we are modulating it with a fixed frequency. Since we don't know the carrier frequency and we're not interested in encoding for broadcast we can simply use the modulation frequency in place of the carrier for equation 8. The depth is still expressed as a percentage as in equation 8 however, because the Steinberg VST standard states that our output amplitude cannot be over 1, we cannot add 1 to the total product or we will overmodulate our output signal and our host will receive a distorted signal. Thus we can modify it by subtracting the modulation index from 1 and we are left with a modifier that will not overmodulate our amplitude. In effect we are left with the equation:

$$\text{modifier} = (1 - \text{depth}) + \text{depth} (\sin \omega t)^2$$

(Eq. 10)

## Algorithm

Now that we know the theory behind the tremolo effect, we need to design a program that will apply the mathematics to a real signal. In order to design our plugin program we need to come up with a general algorithm. An algorithm is nothing more than a set of instructions that solve a problem in a certain amount of time.<sup>iv</sup> Our plugin algorithm appears as follows:

1. Create an instance of our program when the host application calls for the plugin by reading input and output parameters from the host and returning information about our programs name and abilities to the host.
2. Display a graphical user interface (GUI) that allows the user to set the depth and frequency of the tremolo effect.
3. Read the depth and frequency values set by the user.
4. Read the sample rate of the current audio data
5. Calculate the time for the sample
6. Calculate our amplitude modifying value

7. Read a sample of audio data from the host program
8. Apply the modifying value to the audio sample
9. Return the audio sample to the host program
10. Repeat steps 3 through 9 until audio playback is stopped.

This is the high-level algorithm that defines the operation of the tremolo plugin. We start out by loading our program in memory and defining various parameters that allow the plugin to communicate with the host properly. Then we provide an interface for our users to adjust given parameters and then we read the parameter values into our program. After reading the data we perform our tremolo modification calculation, retrieve a sample of audio from the host, apply the modifier to the audio sample and return the modified sample to the host. Finally we repeat the read, apply, return process for each sample that the host program provides the plugin until samples stop coming. It is important to define a stopping point otherwise the plugin would continue processing invalid data forever creating an infinite loop that could eventually cause the computer to crash. Once playback ceases the plugin simply waits for more data.

## Tools

Creating the tremolo plugin requires the proper set of software tools. Steinberg Inc. (now a division of Pinnacle Systems, a producer of multimedia production software) designed an open called Virtual Studio Technology that acts as an interface between a processing plugin and a host application so that the plugin can be used with any audio program that adheres to the standard. Steinberg has posted a free developers kit (SDK) on their website with documentation and a couple of plugin examples. The VST interface is coded with the C++ programming language so it is best to write plugins in C++. Microsoft's Visual Studio .NET was used to code and compile the tremolo plugin. Compilation consists of taking the user readable source code and converting it into a machine readable binary file that can be called by the host application. Two host applications that can run the VST based plugin are Steinberg's Wavelab and Nuendo. From this point on it is recommended that the reader have a basic knowledge of C++ coding and understand what functions, data types, classes, and variables are and how a program is compiled.

## Example Code Walkthrough

With the software developers kit, Steinberg includes the source code for a fully functional gain changing plugin called AGain. This simple plugin reads a sample of audio and adjusts the volume from unity (no adjustments are made) to infinity (no output) depending on the value of the single user adjustable parameter called "Gain". AGain is the basis for the tremolo plugin, however where tremolo adjusts the volume periodically with time, AGain does not change the volume over time unless the user manually adjusts the settings during playback. We will walk through the AGain C++ code example to

become familiar with the way the plugin works and then we will modify the program to create the tremolo plugin.

AGain consists of three source files that contain the C++ code that allows the compiling software to create a binary executable. The files also include references to each other, standard C++ libraries, and Steinberg's VST libraries. The three files are named AGainMain.cpp, AGain.hpp, and AGain.cpp. AGainMain.cpp is responsible for executing the correct code when the plugin is loaded into the host. Most plugins will not need to have any changes made to this file so we will not be viewing it in depth. Just remember that this file is necessary to run a plugin.

## AGain.hpp<sup>v</sup>

The first file that we will look at is called the header file and Steinberg calls it AGain.hpp. This file is responsible for defining the plugin object so that other programs know how to interface with the plugin without having to know how the plugin accomplishes its tasks. Header files contain function declarations (called the prototype) which include the names of the function, parameters, and the return data type. Headers also contain public variables and their data types.

```
1  //-----  
2  // VST Plug-Ins SDK  
3  // Example AGain (VST 1.0)  
4  // Stereo plugin which applies a Gain [-∞, 0dB]  
5  // © 2003, Steinberg Media Technologies, All Rights Reserved  
6  //-----  
7  
8  #ifndef __AGAIN_H  
9  #define __AGAIN_H  
10  
11 #include "audioeffectx.h"
```

The first six lines are comments. The double slash (//) in C++ informs the compiler that anything following those slashes are programmer notes that do not have an operational effect on the code so they can safely be ignored. Comments usually include helpful information about the program that is not obvious in the code itself. Line 8 checks to see if the AGain class has been declared to the preprocessor which is responsible for gathering all object, function, variable, and class names at the beginning of compilation. If AGain has not been declared then line 9 will declare it. Line 47 ends the “if end if” declaration test. Line 11 tells the compiler that it needs to include the AudioEffectX class in our program because we will need to access functions and variables from AudioEffectX. When the compiler reads this line of code it will grab the header file for audioeffectx and allow the plugin to use its functions.

The remainder of AGain.hpp declares our class name, our functions and variables.

```
14 class AGain : public AudioEffectX
15 {
16 public:
17     AGain (audioMasterCallback audioMaster);
18     ~AGain ();
19
20     // Processes
21     virtual void process (float **inputs, float **outputs, long
        sampleFrames);
22     virtual void processReplacing (float **inputs, float
        **outputs, long sampleFrames);
23
24     // Program
25     virtual void setProgramName (char *name);
26     virtual void getProgramName (char *name);
27
28     // Parameters
29     virtual void setParameter (long index, float value);
30     virtual float getParameter (long index);
31     virtual void getParameterLabel (long index, char *label);
32     virtual void getParameterDisplay (long index, char *text);
33     virtual void getParameterName (long index, char *text);
34
35     virtual bool getEffectName (char* name);
36     virtual bool getVendorString (char* text);
37     virtual bool getProductString (char* text);
38     virtual long getVendorVersion () { return 1000; }
39
40     virtual VstPlugCategory getPlugCategory () { return
        kPlugCategEffect; }
41
42 protected:
43     float fGain;
44     char programName[32];
45 };
46
47 #endif
```

Line 14 defines the name for our AGain class and tells us that it is derived from the AudioEffectx class (remember the header file from line 11). This means that our new AGain class can access variables and functions that are in the AudioEffectx class without having to rewrite the AudioEffectX implementation on our own. A large part of the power of object oriented programming lies in the ability to assimilate other objects/classes into your own object. You can view the AudioEffectX.hpp header file in the SDK to find out what variables and functions are provided. However do not edit the AudioEffectX.hpp file or you will be breaking the VST standard and run the risk of you plugin failing to function properly in any host. The SDK documentation contains a list of AudioEffectX functions, their parameters and purpose.

The bracket on line 15 denotes the beginning of the definition of the AGain class and line 45 denotes the ending. The keyword “Public:” on line 16 tells the compiler that all of the following functions and data are available for other objects and classes to read

and change. For instance if another plugin wanted to include our AGain class, it would have access to any function or variable that follows the “Public:” declaration until another keyword such as “Private:” or “Protected” is encountered. Everything that follows “Public:” is simply a declaration of a variable or function and does not contain specific values or implementation code.

In order to initialize the variables and functions of our AGain class, we need a constructor which is a function that is invoked when our plugin is called up by the host program. Line 17 shows the constructor for our program and it initializes an object of type `audioMasterCallback` as a parameter. This object is passed from the host to our plugin as a part of the VST specification in order to handle the actual interaction between the two programs and is responsible for providing our audio samples. When the plugin is closed we need to clean out the memory used by our plugin and that job is accomplished by the destructor on line 18.

Lines 21 through 40 are our public function declarations. We will discuss each function when we walk through the implementation code but for now know that each declaration follows the format of: *binding-type return-data-type function-name (parameters)*. All of these functions have a binding type of “virtual” which means that the host application can use our AGain class to derive another class and still use AGains virtual functions as its own without any problems.

Keyword “Protected:” on line 42 declares that the following variables are able to be read by the host application but not changed directly. Line 43 declares “fGain” of data type `float` which will be used by the end user and the GUI to adjust the overall gain of our sample audio. Line 44 declares a simple variable that will contain the name of the program as a string of 32 characters. Note that neither of these variables have data stored in them. The constructor will initialize each variable in the implementation file.

Line 47 concludes the test to see if our class was defined to the preprocessor. And that’s the end of the AGain header file. Next we move on to the implementation file containing code that actually performs calculations and work in the functions that we just declared.

## AGain.cpp<sup>vi</sup>

```
1  //-----  
2  // VST Plug-Ins SDK  
3  // Example AGain (VST 1.0)  
4  // Stereo plugin which applies a Gain [-∞, 0dB]  
5  // © 2003, Steinberg Media Technologies, All Rights Reserved  
6  //-----  
7  
8  #ifndef __AGAIN_H  
9  #include "AGain.hpp"  
10 #endif
```

Our implementation file begins in much the same way as our header file with comments about the file and tests to see if the AGain class has been declared to the preprocessor yet. This time line 9 includes the AGain header file that we were looking at previously so that the compiler can verify that we are not defining any functions or

variables that we have not declared in the headers. Each function definition is in the format of *return-data-type namespace::function-name (parameters)*. The namespace refers to what class or object the function we are defining belongs. Because our object is called AGain, every function we define will start with “AGain” as the namespace. Then we provide the function name that we used in the header declaration and provide the same parameters.

```
13 AGain::AGain (audioMasterCallback audioMaster)
14     : AudioEffectX (audioMaster, 1, 1) // 1 program, 1
      parameter only
15 {
16     fGain = 1.; // default to 0 dB
17     setNumInputs (2); // stereo in
18     setNumOutputs (2); // stereo out
19     setUniqueID ('Gain'); // identify
20     canMono (); // makes sense to feed both
      inputs with the same signal
21     canProcessReplacing (); // supports both accumulating and
      replacing output
22     strcpy (programName, "Default"); // default program name
23 }
```

This piece of code is the implementation of the AGain constructor that we declared in the header file on line 17. Line 13 names the constructor (it has to be the same name of the class) and creates an object of type audioMasterCallback called audioMaster as it’s parameter. Because AGain also inherits the AudioEffectX class, we also have to supply three parameter values in order to initialize the variables that we inherited from AudioEffectX. In the first parameter we pass the audioMaster object we just initialized. The second parameter, the number 1, tells the host application that we have only one preset for our plugin. A plugin can have many presets that a user clicks to adjust the plugin for a certain overall setting, for example a reverb might have preset to make it sound like a large hall and another to represent a small auditorium. Because the AGain plugin only has one adjustable parameter we do not have much of a use for multiple presets. So we will have the plugin open to an initial value. The third parameter states that we only have one user adjustable parameter, in our case “fGain”.

Line 16 initializes the variable fGain (that we declared in the header file) to 1 so that when the plugin is opened by the host application it will not modify the amplitude of the processed audio until the user adjusts the parameter.

Line 17 and 18 are function calls from AudioEffectx responsible for setttin a private variable that tells the host how many inputs and outputs our plugin can process. In this case we are telling the host that our plgin can process two channels of audio.

Line 19 calls a function to provide our plugin with a unique ID that the host can use to identify our plugin. If another plugin uses the same ID and is loaded into the host then the computer may crash. The end user may never see this ID unless it is programmed into the GUI to display it however it is very important to the host application.

Line 20 tells the host that we are applying the same type of processing to both channels so we could use this plugin in mono. If we wanted to implement a true stereo

effect, such as an autopanner, we would not call this function and the plugin could not be used by the host to process mono signals.

Line 21 deals with the way our plugin is used in a host application. The two most important functions in the plugin are *process()* and *processReplacing()* because they handle the actual editing of the audio sample. If the plugin is to be used as a send effect in the host application, *process()* is called. If the plugin is used as an insert effect, *processReplacing()* is called. That way you can have two algorithms that operate differently depending on the situation.

Line 22 simply labels our default program “Default” in the host GUI.

```
26 AGain::~AGain ()
27 {
28     // nothing to do here
29 }
```

Our destructor for the AGain plugin has nothing special to do other than clean out the simple variables that we were using. If our program was more complicated, including link lists for example, then we would have to step through the lists and clean out the memory so that the computer could use it again.

```
32 void AGain::setProgramName (char *name)
33 {
34     strcpy (programName, name);
35 }
36
37 //-----
38 void AGain::getProgramName (char *name)
39 {
40     strcpy (name, programName);
41 }
```

The next two functions aren’t even used in this example. *setProgramName* provides a means for the plugin to change the name of the preset displayed in the host and *getProgramName* reads the name from the host. In both cases the parameter passed in or out of the function is a pointer to a character string.

```
44 void AGain::setParameter (long index, float value)
45 {
46     fGain = value;
47 }
48
49 //-----
50 float AGain::getParameter (long index)
51 {
52     return fGain;
53 }
```

The next two functions deal with the gain parameter that the user can adjust. *setParameter* on line 44 accepts two parameters, a long and a float. The index parameter refers to the user definable parameter index number. If a plugin has more than one user definable parameter, each one must have a unique index number in order to differentiate

them. The second function parameter is the value that the host provides to be stored in the fGain variable. The setParameter function allows for the host to grab the current fGain setting to display in the GUI.

```
56 void AGain::getParameterName (long index, char *label)
57 {
58     strcpy (label, "Gain");
59 }
```

The getParameterName function provides names to the GUI for each of the user definable parameters. Our only parameter is called “Gain”.

```
62 void AGain::getParameterDisplay (long index, char *text)
63 {
64     dB2string (fGain, text);
65 }
```

Because all user parameter values in the VST standard range from 0.0 to 1.0, we have to convert and display them in a format that is a little more readable in the GUI. The audioEffectX class includes functions to convert the float values into strings that the user can understand. In this case the function takes the parameter (fGain) value and passes it through the dB2string function which returns a character string in decibels that the GUI can display to the user.

```
68 void AGain::getParameterLabel(long index, char *label)
69 {
70     strcpy (label, "dB");
71 }
```

The next function simply labels the GUI with a character string. In this case we are displaying “dB” but it could be anything up to 8 characters length.

```
74 bool AGain::getEffectName (char* name)
75 {
76     strcpy (name, "Gain");
77     return true;
78 }
```

Next, the getEffectName function returns a name for the plugin that the GUI can display. The ‘return true’ statement simply says that this function has completed it’s goal.

```
81 bool AGain::getProductString (char* text)
82 {
83     strcpy (text, "Gain");
84     return true;
85 }
```

Steinberg asks for each plugin to be registered with their web site and when you fill out the registration form you get a short identifying string to make your plugin unique. If you receive one of these strings you insert it here in the place of “Gain”. It does

not affect the operation of your plugin but having a product string is a nice finishing touch once your plugin is working.

```
88 bool AGain::getVendorString (char* text)
89 {
90     strcpy (text, "Steinberg Media Technologies");
91     return true;
92 }
```

For more identification you can provide your name or the name of your company using this function. Some hosts will check the product string and vendor string when cataloging and organizing plugins so the end user may see this information.

```
95 void AGain::process (float **inputs, float **outputs, long
    sampleFrames)
96 {
97     float *in1 = inputs[0];
98     float *in2 = inputs[1];
99     float *out1 = outputs[0];
100    float *out2 = outputs[1];
101
102    while (--sampleFrames >= 0)
103    {
104        (*out1++) += (*in1++) * fGain;    // accumulating
105        (*out2++) += (*in2++) * fGain;
106    }
107 }
```

Finally we come to the heart of the plugin where the actual audio processing occurs. Line 95 defines the process function and accepts input and output array pointers and a long value called “sampleFrames”. The two arrays are our buffers of audio samples and their size (how many samples the buffers hold) is determined by the “sampleFrames” value. We can use the “sampleFrames” value to determine when we’ve run out of samples and when to stop processing. Lines 97 through 100 create local pointer variables that point to the memory address of the left and right input and output buffers. These local pointers will be used to step through the buffer arrays.

Line 102 sets up a While loop. Any processes within the loop will repeat until the conditions set in the parameter of the loop are met. In this case each iteration of the loop decrements the local “sampleFrames” value until it becomes -1. At this point the loop quits and processing continues to the next step which is to exit the function.

Each iteration of the loop alters the actual values held in the audio buffers. Line 104 and 105 are similar, only they act on different audio channels. Each line is processed using the order of precedence defined by C++. First the pointer “\*out” is resolved to read its value which is usually a 0 or null on the first iteration. Next we resolve the pointer “\*in” to read its value which is a floating point number representing the amplitude of the current audio sample. Then the value represented by “\*in” is multiplied by our “fGain” value (the floating point value set by our user in the GUI). Next the “\*out” value is added to the modified “\*in” value. Then the “\*out” value is replaced by the sum of the “\*out” value and the modified “\*in” value. Then the “\*out” and “\*in” pointer addresses are incremented to read the next buffer sample and the iteration is finished. Once

“sampleFrames” reaches -1, all of the iterations are finished and the output buffer is returned to the host via “\*out” and our function is called again to read the next buffer.

Following the process function is the *processReplacing* function that performs exactly the same steps as *process()* except that instead of adding the value of the “\*out” pointer to the “\*in” pointer in each iteration it simply replaces the “\*out” value with the modified “\*in” value. This will allow the plugin to be used as an insert effect.

Now that you have seen how the AGain example plugin works, lets take a look at the modifications we made to create the tremolo plugin.

## Tremolo Code Walkthrough

### Tremolo.hpp

```
1  //-----  
2  // Chris Larsen October 2003 Senior Project University of  
   // Hartford Ward College  
3  // EA Tremolo  
4  // A stereo plugin that takes audio samples and adjusts their  
   // amplitude periodically  
5  // with a user adjustable rate and depth to provide the  
   // classic tremolo sound  
6  //-----  
7  
8  #ifndef __TREMLO_H  
9  #define __TREMLO_H  
10  
11 #include "audioeffectx.h"  
12  
13 enum  
14 {  
15     // Parameters identifiers  
16     kDepth,  
17     kFrequency,  
18     kNumParams  
19 };
```

Looking at our new tremolo.hpp header file we can see that the comments were changed to reflect the nature of the new plugin. Also the preprocessor declaration names were change to reflect the filename changes. Again the audioeffectx.hpp header file is included so that we can inherit its class.

New to the code is the enum statement and our parameter identifiers. An enumeration data type is an ordered list of names (enumerators) that allow for easier reading and code organizing. Each enumerator corresponds to an integer value. In our case “kDepth” = 0, “kFrequency” = 1 and “kNumParams” = 2. If we wanted to add more parameters to our plugin we could insert them just before “kNumParams”. That way “kNumParams” will always contain the actual number of parameters by default. By using the enum data type we can reference “kDepth” and “kFrequency” as the index values in our code instead of having to remember the integer numbers. Paramter names makes it much easier to understand the code.

The rest of the function declarations in our header file are the same as the AGain example. The only other difference in the code is where we declare our protected variables after line 50.

```
50 protected:
51     float fDepth;
52     float fFrequency;
53     // global counter used to calculate the time
54     long counter;
55     char programName[32];
```

We declared two float variables to hold the values that our user sets via the GUI for our depth and frequency parameters. We also declared a counter to hold a value we will use to calculate timing information as required by our tremolo equation 8. We will have more information about timing under the *process()* function.

## Tremolo.cpp

```
1  //-----
2  // Chris Larsen October 2003 Senior Project University of
   Hartford Ward College
3  // EA Tremolo
4  // A stereo plugin that takes audio samples and adjusts their
   amplitude periodically
5  // with a user adjustable rate and depth to provide the
   classic tremolo sound
6  //-----
7
8  #ifndef __TREMLOLO_H
9  #include "TREMLOLO.hpp"
10 #include <cmath>
11 #endif
```

Our Tremolo.cpp implementation file also starts out like the AGain example however again the preprocessor declaration is different to reflect the filename and so is the first include statement on line 9. Line 10 includes a standard C++ library called *cmath* that we will need to access in order to use the ‘sin’ function for our periodic calculations.

```
15 Tremolo::Tremolo (audioMasterCallback audioMaster)
16     : AudioEffectX (audioMaster, 1, kNumParams)    // 1
   program, 2 parameters only
17 {
18     counter = 1;                                // global counter to keep
   time
19
20     fDepth = 0.5;                                // default to minimum depth
21     fFrequency = 0.1;                            // set the frequency
22     setNumInputs (2);                            // stereo in
23     setNumOutputs (2);                          // stereo out
24     setUniqueID ('Trem');                        // identify
```

```

25     canMono (); // makes sense to feed both
        inputs with the same signal
26     canProcessReplacing (); // supports both accumulating and
        replacing output
27     strcpy (programName, "Default"); // default program name
28 }

```

The tremolo constructor is similar to the AGain constructor except that it passes “kNumParams” on line 16 to tell the audioEffectX object how many user definable parameters we will have for our plugin. We declared “kNumParams” as an enum datatype in the header file and the order that it was declared assigned the integer 2 to “kNumParams”, therefore 2 will be passed from the constructor to notify the compiler that we have two user definable parameters. Line 18 initializes our global counter to 1. Line 20 initializes our depth parameter to .5 which translates to 50% modulation. Our initial frequency is set to 1 hertz on line 21. Remember that we can only provide parameter values between 1.0 and 0.0 so our depth and frequency cannot be read one for one. We have to alter the values to make them useful in our equation and we will do that in the *process()* function and *processReplacing()*. The only other constructor difference is the “UniqueID” which is set to “Trem”.

As in the AGain example we have nothing to do in the destructor but it is still defined in the implementation file on line 32. Also, the setProgramName and getProgramName functions are identical to the AGain code.

```

49 void Tremolo::setParameter (long index, float value)
50 {
51     switch (index)
52     {
53         case kDepth :     fDepth = value;         break;
54         case kFrequency : fFrequency = value;     break;
55     }
56 }

```

Because we now have two user parameters, our next difference naturally involves the parameter functions. In AGain we didn’t need to worry about the index for our user parameters because we only had one. This time we have two parameters and the host program will pass the integer of the parameter that is being changed as well as the value of that change to the setParameter function so that we can adjust the value of the correct variable. When the switch function receives the enumerator name of the index it compares the value to the enum value of the first case statement and if it matches it executes the code to the right of the colon and exits the function. If the enumerator index and the case do not match it proceeds down each case until it finds a match or runs out of cases. In this function when the correct parameter index is matched it will write the value to the “fDepth” or “fFrequency” variable.

```

59 float Tremolo::getParameter (long index)
60 {
61     float v = 0;
62
63     switch (index)
64     {
65         case kDepth :    v = fDepth; break;
66         case kFrequency : v = fFrequency; break;
67     }
68     return v;
69 }

```

getParameter uses an internal variable called v to store a value provided by the switch statement for the parameter that the host program requests. Again the switch statement cycles through the cases until a match is found and this time instead of assigning a value to the user definable parameter, it reads the values to be displayed in the host GUI.

```

72 void Tremolo::getParameterName (long index, char *label)
73 {
74     switch (index)
75     {
76         case kDepth :    strcpy (label, "Depth");
77                         break;
78         case kFrequency : strcpy (label, "Frequency");
79                         break;
80     }
81 }

```

Naming the parameters is similar to AGain, we just use the switch function again to cycle through the parameters.

```

82 void Tremolo::getParameterDisplay (long index, char *text)
83 {
84     switch (index)
85     {
86         case kDepth :    float2string (fDepth * 100, text);
87                         break;
88         case kFrequency : float2string (fFrequency * 10,
89     text);    break;
90     }
91 }

```

In order to display the float values correctly in the GUI we have to apply adjustments so that they represent what they actually do in the tremolo equations and then convert them to strings (using the float2string function provided by audioeffectx) to be displayed on screen. Our depth is multiplied by 100 on line 86 to display the float as a percentage. The frequency is multiplied by a factor of 10 to be displayed as hertz.

```

92 void Tremolo::getParameterLabel(long index, char *label)
93 {
94     switch (index)
95     {
96         case kDepth :    strcpy (label, "%");    break;
97         case kFrequency : strcpy (label, "Hz");  break;
98     }
99 }

```

To label our parameters we use the same switch statement as before and copy a % sign for the depth and Hz for the frequency.

getEffectName, getProductString, and getVendorString are the same functions as in the AGain example but we have provided different strings for the values that you can see in the appendix on lines 100 through 120.

```

123 void Tremolo::process (float **inputs, float **outputs, long
        sampleFrames)
124 {
125     float *in1 = inputs[0];
126     float *in2 = inputs[1];
127     float *out1 = outputs[0];
128     float *out2 = outputs[1];
129     float sampleRate = updateSampleRate();
130     float modifier;
131
132     while (--sampleFrames >= 0)
133     {
134         // increment our counter
135         if(counter < ((1/fFrequency * 10) * sampleRate))
136         {
137             counter++;
138         }
139         else
140         {
141             counter = 0;
142         }
143
144         // calculate the multiplier
145         modifier = (1.0 - fDepth) + fDepth * (sin((2 * 3.14 *
            (fFrequency * 10)) * (counter/sampleRate))) * (sin((2 *
            3.14 * (fFrequency * 10)) * (counter/sampleRate)));
146
147         (*out1++) += (*in1++) * modifier;    // replacing
148         (*out2++) += (*in2++) * modifier;
149     }
150 }

```

Our process function begins like the gain example. However at line 129 we have a new local variable called “sampleRate” that grabs a value from the function *updateSampleRate()*, provided by the audioEffectX class. We will need the sample rate in conjunction with our global counter to calculate our time beyond the audio buffer provided by the host application. The audio buffer may be shorter than one cycle of our frequency setting so having a sample counter outside of the *process()* function allows us

to maintain our position in the cycle once we have run out of audio in the buffer and have to restart the *process()* function. Line 130 declares a temporary modifier float variable to be used to store the calculation of our tremolo modifier.

The while loop still operates as it does in the AGain example by stopping when we run out of audio buffer samples. Our first step within the loop is to check and or increment our counter variable. Line 135 checks to see if the counter value has exceeded one cycle as defined by our user in the “fFrequency” variable. The equation to the right of the < symbol calculates how many samples there should be in one cycle for the given frequency parameter. The fFrequency value is multiplied by 10 to reach a hertz value. Then we take the inverse of the resultant value to get the period of the cycle in seconds. Because the sample rate represents the number of samples provided in one second of audio we can multiply the period by the sample rate to find out how many samples there are in one cycle.

For example if we use the default parameter value of .1 we convert it to 1 hertz by multiplying it by 10. Then inverting 1 hertz gives us a period of 1 second and if we multiply that by a typical sample rate of 44100 samples/second we get 44100 samples per cycle which seems intuitively correct.

Each time the loop runs through an iteration we have processed one sample of audio and the counter will hold how many samples we have processed. If the value of the counter is less than the number of samples per cycle calculated on line 135 then we can move to line 137 and increment the counter (add 1 to the current value). If the counter value is greater than the number of samples per cycle then we skip to line 141 and reset the counter to 0 to start the count over.

On line 145 we calculate the tremolo modifying value using equation 9 that we solved for in the design phase. After calculating the modifier then we can apply it on lines 147 and 148 by multiplying the value of the “\*in” sample by the modifier and assigning the resultant to the “\*out” pointer which returns the value to the host.

The function for *processReplacing()* is identical to *process()* except that we do not add the output buffer to the input buffer before assigning it to the output.

## Results

After the source code is finished we compile the code into a binary executable that can be used in a host application. In our case the compiler creates eatremolo.dll. To use the plugin we move the binary file into the host applications plugin directory, launch the host, and load the plugin into the appropriate slot.

Eatremolo.dll was compile on a Dell Inspiron 7500 laptop with a 750MHz Pentium III processor and 392MB of RAM. The plugin was initially tested on this machine using Steinbergs Wavelab 4.01 and Steinbergs Nuendo 2.0.

The tremolo plugin uses the default GUI provided by the host application. In Wavelab and other Steinberg applications the graphical user interface looks like the following.



Figure 3

Our depth parameter is displayed on the left as a string of text just above the percent sign that we coded into the program on line 96. The frequency parameter appears just to the right of the depth parameter. Steinberg's interface has room for three parameters on each page with multiple pages allowed and accessed by the parameter plus and minus page buttons on the right. When a parameter is selected it is highlighted with a green box and it can be adjusted with either the plus and minus buttons or the jog wheel in the center. The preset button allows the user to load and save different programs but our plugin only has one default program. You can bypass the plugin easily by clicking the bypass button which will simply take it out of the virtual circuit. The user can also mute any audio from the plugin by clicking the mute button.

Visually we should expect to see an AM envelope like that of Figure 2 if we pass a waveform through our plugin and this is exactly what we get. Here is a 220Hz sinewave with an amplitude of 100% as displayed in Steinberg's Wavelab before processing with the tremolo plugin.

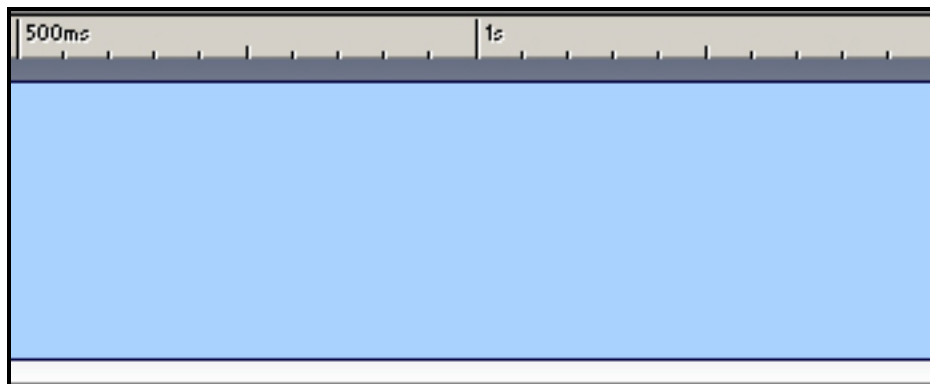


Figure 4

Next we show the same 220Hz sinewave processed with the tremolo plugin set at 50% depth and a frequency of 1Hz. The image is similar to the AM envelope and you can see that the nulls do not meet in the center proving that we are operating at 50% modulation.

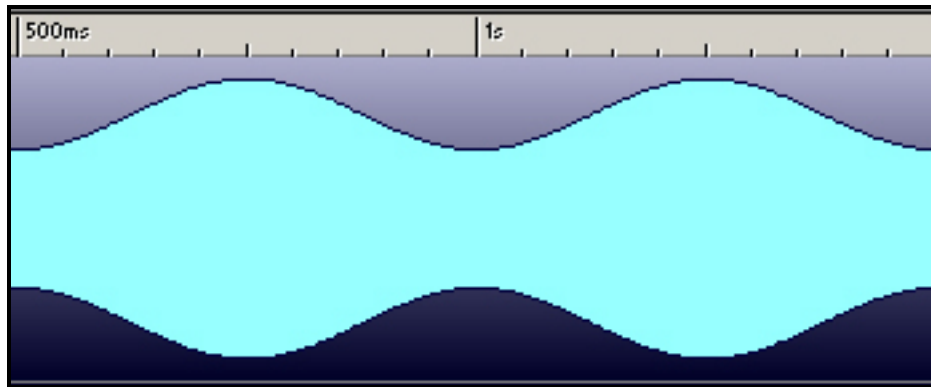


Figure 5

We can zoom in and see that the carrier frequency is a steady 220Hz while the amplitude changes according to our modulation frequency. The white dashed line represents 50% of amplitude in the positive and negative directions.

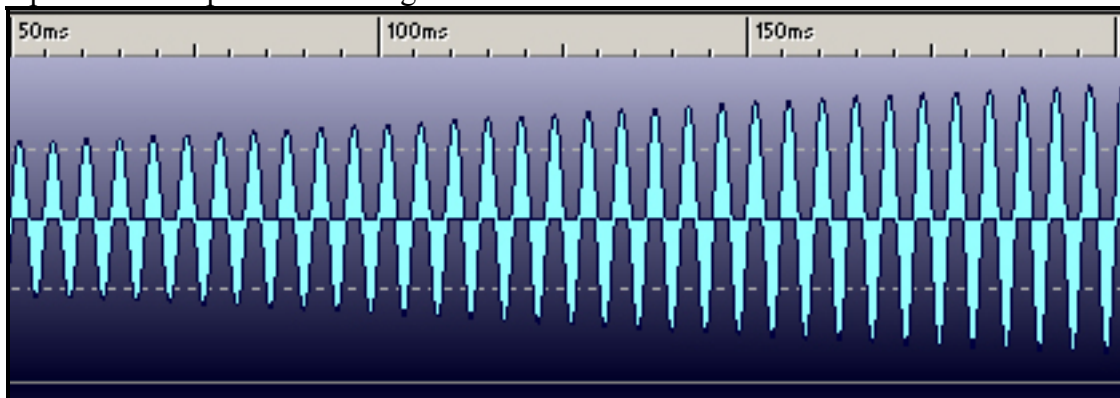


Figure 6

If we increase our depth to 100% we observe the following waveform. Notice that the nulls meet in the center.

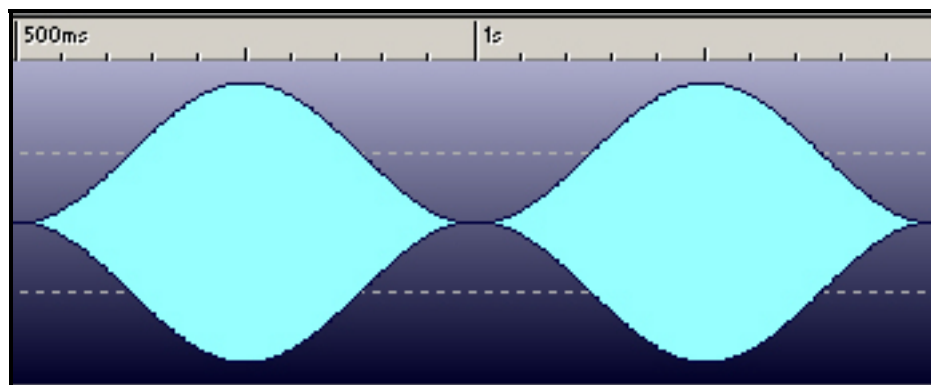


Figure 7

Setting the depth back to 50% and adjusting the modulation frequency to 2Hz we obtain the following waveform.

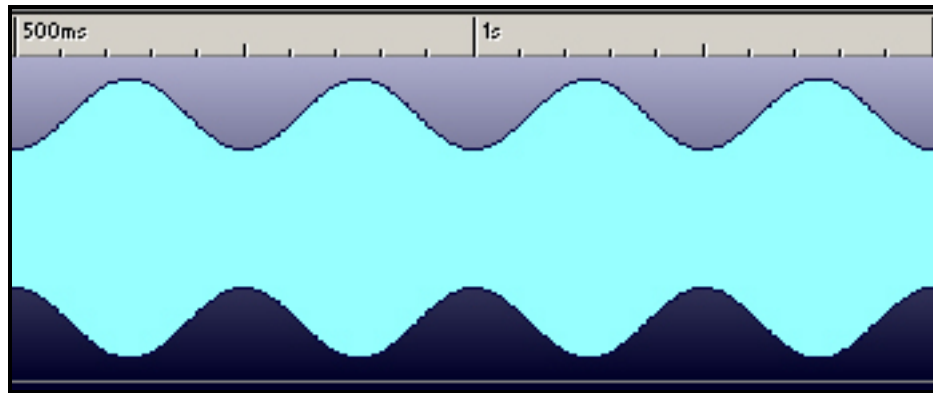


Figure 8

Please listen to the accompanying audio CD in appendix D to hear the tremolo effect on the signals displayed in figures 4 through 8. You can also hear the effect on a single acoustic guitar and on a guitar submix.

## Conclusion

In the end we were able to create a functional audio processing plugin for the VST interface that applied a tremolo effect to audio passed through its interface. We were able to meet our design criteria of processing a stereo audio signal, having two user adjustable parameters, and providing a clean output that sounded good and did not introduce artifacts. The plugin can be used with any host application that adheres to the VST standard. After manipulating the amplitude modulation equation we were able to come up with an algorithm that we could convert into C++ code and plug into the Steinberg AGain audio processor example. Compiling and running the tremolo plugin in an actual VST host application produced waveforms and audio that we predicted. Using the tremolo on a guitar with a low depth and frequency setting provided a nice mellow sound that fit well with many styles of music. Using a high depth and frequency setting provided a shimmering sound that works well as an extra effect used sparingly.

One feature that we could add to improve this plugin is to allow the user to select additional waveforms to modulate the signal including triangle waves, square waves, and sawtooth waves. Fans of analog synthesizers would appreciate the ability to use these different waveforms to modify sample signals. Another advanced feature would be to synchronize the timing with the host application. Currently when playback begins, modulation always starts at zero and moves towards the positive in the cycle. For extremely long tremolo frequencies it is desirable to control the sweep so that peaks and nulls occur in time with the audio. In order to work around that problem with the current plugin, the user can shift the audio and begin playback from the same position every time to land peaks in the appropriate places.

With a solid background in digital signal processing and computer programming, many wonderful advanced plugins can be created with the VST interface standard.

## End Notes

---

- <sup>i</sup> Blake, Roy Electronic Communication Systems, 2002 Thomson Learning P. 102
- <sup>ii</sup> Blake, Roy Electronic Communication Systems, 2002 Thomson Learning P. 103
- <sup>iii</sup> Blake, Roy Electronic Communication Systems, 2002 Thomson Learning P. 105
- <sup>iv</sup> Dale, Weems, Headington Programming and Problem Solving with C++, 2000 Jones and Bartlett Publishers P. 5
- <sup>v</sup> Steinberg VST Software Developers Kit
- <sup>vi</sup> Steinberg VST Software Developers Kit