Steinberg

Virtual Studio Technology Plug-In Specification 2.0 Software Development Kit

Documentation Release #1

Steinberg VST 2.0 SDK
Hyperlinks & Contents

VST Plug-Ins Formal issues

What is a VST Plug-in?

In the widest possible sense a VST-Plug-in is an audio process. A VST Plug-in is not an application. It needs a host application that handles the audio streams and makes use of the process the VST plug-in supplies.

Generally speaking, it can take a stream of audio data, apply a process to the audio and send the result back the host application. A VST Plug-in performs its process normally using the processor of the computer, it does not necessarily need dedicated digital signal processors. The audio stream is broken down into a series of blocks. The host supplies the blocks in sequence. The host and its current environment control the block-size. The VST Plug-in maintains all parameters & statuses that refer to the running process: The host does not maintain any information about what the plug-in did with the last block of data it processed.

From the host application's point of view, a VST Plug-In is a black box with an arbitrary number of inputs, outputs, and associated parameters. The Host needs no knowledge of the plug-in process to be able to use it.

The plug-in process can use whatever parameters it wishes, internally to the process, but it can choose to allow the changes to user parameters to be automated by the host, if the host application wishes to do so.

The source code of a VST plug-in is platform independent, but the delivery system depends on the platform architecture.

On the Windows platform, a VST plug-in is a multi-threaded DLL (Dynamic Link Library).

On Apple Macintosh, it's a raw Code resource. Note that the plug's name as seen by the user is the resource name (not the file name). A folder labeled 'VstPlugIns' is searched for plugs, this should preferably be installed in the System Folder. Applications should additionally search the application folder.

On BeOS, a VST plug-in is a Library.

On SGI (MOTIF), a VST plug-in is a Library.

Processing Options

Audio processing in the plug is accomplished by one of 2 methods, namely process(), and processReplacing(). The first one must be provided, while the second one is optional (but it is

highly recommended to always implement both methods). While process() takes input data, applies its pocessing algorithm, and then adds the result to the output (accumulating), processReplacing() overwrites the output buffer. The host provides both input and output buffers for either process method.

The reasoning behind the accumulating version (process()) is that it is much more efficient in the case where many processors operate on the same output (aux-send bus), while for simple chained connection schemes (inserts), the replacing method is more efficient.

All parameters - the user parameters, acting directly or indirectly on that data, as automated by the host, are 32 bit floating-point numbers. They must always range from 0.0 to 1.0 inclusive, regardless of their internal or external representation.

Audio data processed by VST Pug-ins are 32 bit floating-point numbers. They must always range from -1.0 to +1.0 inclusive.

Plug-in implementation

If you want to develop a VST plug-in, you may prefer to go straight to the code examples now. These are very simple examples in which you will learn most of the important basic concepts just by reading a few lines of code.

As a plug-in developer you actually need to know very little about hosting a plug-in. You should concentrate on the AudioEffect and AudioEffectX base classes.

AudioEffect is the base class, which represents VST 1.0 plugs-ins, and has its declarations in **audioeffect.hpp**, and the implementation in **audioeffect.cpp**. You will also need to be aware of the structure definitions in **aeffect.h**.

The files **aeffectx.h** (more structures), **audioeffectx.h**, and **audioeffectx.cpp** are similar to the ones above, and extend them to the version 2.0 specification.

X marks the Spot

The files relating to the 2.0 spec have their counterparts in the original 1.0 specification. The original files are expanded upon by versions representing the 2.0 specification. Basically the new classes & files all have a 1.0 predecessor – but the new ones have an X in the name.

aeffectx.h includes **aeffect.h**.
audioeffectx.h includes **audioeffect.hpp**; and
audioeffectx.cpp extends the implementation in **audioeffect.cpp** through the
AudioEffectX class, which inherits from **AudioEffect** and thus is compatible to the 1.0 specs.

Don't Edit. Inherit

Never edit any of these files. Never ever. The host application relies on them being used as they are provided. Anything can be added or changed by overriding in your private classes derived from AudioEffectX.

User Interfaces

All user-interface issues are entirely separated from the audio processing issues. At its simplest there is an option where you can avoid providing a user interface at all. In this case the host requests character strings from the plug-in representing each of the parameters. The host can use the separate ASCII strings for the value, the label, and the units of the parameters to construct its own user interface. This is how the simple code-examples, AGain & ADelay, work. This is also often a good way to develop a VST plug-in, it offers a very short development cycle to start to test the algorithm. The proper interface can come later.

The next user interface level is provided when the plug-in defines its own editor. This allows practically any user interface to be defined. A negative aspect is that then you can quickly land up in platform specifics when dealing with the nuts an bolts of the interface issues, even though the audio process, the concepts and methodology remain platform independent. The ADelayEdit example project works through the generic issues and then deals with the platform specifics for both Windows and the Macintosh operating systems.

The final option is to use the new portable framework for creating sophisticated user interfaces. This framework takes the form of the VSTGUI Library files that are available for all supported VST platforms. Apart from converting a few pictures from one resource format to another, for which there are tools available, and setting up the initial project files, writing a VST 2.0 Plug-in can really be completely cross-platform. VSTGUI Library files and their usage is completely described in the HMTL files that accompany this SDK. After reading about the underlying issues in this document and associated code examples, the VSTGUI comes very highly recommended.

Basic Programming Examples
A simple gain-change plug-in.

You may ask why we are diving right into a programming example. Well the answer is clear, creating a VST plug-in is easy and we don't want to cloud the issue. If the SDK is standing in the way of understanding something relatively uncomplicated, then something has gone wrong.

This example is very simple and the resulting plug-in does not really do anything interesting, but it does show the how uncomplicated creating a VST plug-in can be. The example source files can be found in the accompanying 'examples' folder along with project files for the various platforms.

Please note that when you start experimenting with these examples we have marked the places where you should make your changes if these examples were to be used as templates for your own plug-ins.

Before getting our programming feet wet, maybe we should cover just a couple of background points that will help with understanding the following code fragments.

• The core of your VST plug-in code, and this 'AGain' example too, is built around a C++ class derived from a Steinberg supplied base class called `AudioEffectX`.

• The constructor of your class is passed a parameter of the type `audioMasterCallback`. The actual mechanism in which your class gets constructed is not important right now, but effectively your class is constructed by the hosting application, and the host passes an object of type `audioMasterCallback` that handles the interaction with the plug-in. You pass this on to the base class's constructor and then can forget about it.

• A few flags, and identifiers must be set and you declare your class's input & output requirements at construction time.

• Finally your class gets to over-ride both of two possible member functions that actually do the work. These are repeatedly called by the host application, each time with a new block of data. These member functions have only three parameters: a pointer to the data, a pointer to where you can write modified data to, and how big the block is.

Example Code : AGain

Basic Source Files Required

AGain.cpp

AGain.hpp

AGainMain.cpp

```cpp
#include "AGain.hpp"

//-------------------------------------------------------------------------

AGain::AGain(audioMasterCallback    audioMaster)
        : AudioEffectX(audioMaster, 1, 1)          // 1 program, 1 parameter  only
{
        fGain = 1.;                // default to 0 dB
        setNumInputs(2);           // stereo in
        setNumOutputs(2);          // stereo out
        setUniqueID('Gain');       // identify (you must change this for other plugs!)
        canMono();                 // makes sense to feed both inputs with the same signal
        canProcessReplacing();     // supports both accumulating and replacing output
        strcpy(programName, "Default"); // default program name
}

AGain::~AGain()
{
        // nothing to do here
}

void AGain::setProgramName(char *name)
{
        strcpy(programName, name);
}

void AGain::getProgramName(char *name)
{
        strcpy(name, programName);
}

void AGain::setParameter(long index, float value)
{
        fGain = value;
}

float AGain::getParameter(long index)
{
        return fGain;
}

void AGain::getParameterName(long index, char *label)
{
```

```
                strcpy(label, "  Gain  ");
}

void AGain::getParameterDisplay(long index, char *text)
{
        dB2string(fGain, text);
}

void AGain::getParameterLabel(long index, char *label)
{
        strcpy(label, "   dB   ");
}


//-------------------------------------------------------------------------------
// process - where the action is...

void AGain::process(float **inputs, float **outputs, long   sampleFrames)
{
        float *in1 = inputs[0];
        float *in2 = inputs[1];
        float *out1 = outputs[0];
        float *out2 = outputs[1];

        while(--sampleFrames >= 0)
        {
                (*out1++) += (*in1++) * gain;     // accumulating
                (*out2++) += (*in2++) * gain;
        }
}

void AGain::processReplacing(float **inputs, float **outputs, long sampleFrames)
{
        float *in1 = inputs[0];
        float *in2 = inputs[1];
        float *out1 = outputs[0];
        float *out2 = outputs[1];

        while(--sampleFrames >= 0)
        {
                (*out1++) = (*in1++) * gain;     // replacing
                (*out2++) = (*in2++) * gain;
        }
}
```

That's all there is to it! That cannot have hurt too much. So let's walk through all that again and explain what's where.

To recap, apart from a very small 'main' function that resides in another file, which is used by the host to construct your plug-in, you really have seen the complete code for an VST plug-in, admittedly it does not really do much.

Your plug-in class is derived from `AudioEffectX` and provides is own processing methods, and that's all you need to create a VST plug-in that uses the host's default method for displaying its parameters.

How does it work?

Before proceeding with the description of the plug-in code fragment, we will take a short look at a skeleton code fragment from the host-side. That is, the application code associated with supporting VST-Plug-ins in an application. This SDK does not handle the host-side implementation of VST Plug-ins and the code here is purely for understanding purposes.

First, when the Application instanciates a plug-in, it calls the plug-ins main () function somewhat like this:

```
typedef AEffect *(*mainCall)(audioMasterCallback cb);
audioMasterCallback audioMaster;

void instanciatePlug (mainCall plugsMain)
{
        AEffect *ce = plugsMain (&audioMaster);
        if (ce && ce->magic == AEffectMagic)
        {
        ....
                }
}
```

It's all pretty elementary stuff, our instanciatePlug() function takes a typedef'd pointer to the `main()` function in our plug-in code. Our plug's `Main()` is then called with the audioMaster data which we have already established is:

• Data, controlling the interchange between the host and the plug-in
• Passed on to the constructor of the base-class `AudioEffectX`
• and is nothing to worry about.

The host then gets a pointer to the plug-in, if all is well, and then can check if it was actually a VST plug-in that was actually created. This is magic.

Returning now to the plug-in code for this example, let's have a look at the `main()` function itself. This you can find in AGainMain.cpp.

```
AEffect *main(audioMasterCallback audioMaster)
{
        // get vst version
        if (!audioMaster (0, audioMasterVersion, 0, 0, 0, 0))
```

```
        {
                DEBUGGERMESSAGE ("old vst version");
                return 0;
        }
        effect = new AGain (audioMaster);
        if (!effect)
        {
                return 0;
        }
        return effect->getAeffect ();
}
```

Firstly you can see that the `audioMasterCallback audioMaster` is called with the `audioMasterVersion` flag just to check that our plug-in is being opened in a valid VST host.

Next the constructor of our gain plug in is called. Unless something goes wrong (out of memory etc), and we return null to the VST host, then the resulting `effect` object has one of it's member functions called `effect->getAeffect ()`, and the result of this call is returned to the host.

What's actually being returned to the host is the C interface of our plug-in: We may be writing code in C++ but deep down there is a C structure filled with fields we need to know nothing about.

Whenever the Host instanciates a plug-in, after the `main()` call, it also immediately informs the plug-in about important system parameters like sample rate, and sample block size. Because the audio effect object is constructed in our plug-in's main(), before the host gets any information about the created object, you need to be careful what functions are called within the constructor of the plug-in. You may be talking but no-one is listening.

Let's look at some details of the gain example. First, the constructor:

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

Our plug-in's constructor has the callback to the host as its parameter. This is passed onto the `AudioEffectX` base-class where it is stored, but the constructor to the base-class also has two extra parameters that set the number of programs the effect has, and the number of parameters the host should 'see' as user parameters for the plug-in.

> Parameters are the individual parameter settings the user can adjust.
> A VST host can automate these parameters.
>
> Programs are a complete set of parameters that refer to the current state of the plug-in.
>
> Banks are a collection of Program objects.

Our plug-in only has one parameter, namely the gain. This is a member variable of our AGain class, and get initialized to 1.0, which translates to 0dB. This value like all VST parameters is declared as a `float` with a inclusive range of 0.0 to 1.0. How data is presented to the user is merely in the user-interface handling.

> **Parameter** values are limited to the range 0.0 to 1.0
> This is a convention, but still worth regarding. Maybe the VST-host's automation system depends on this range.

Next the plug-in states the number of inputs and outputs it can process with `setNumInputs(2)` & `setNumOutputs(2);`

The plug-in next declares its unique identifier. The host uses this to identify the plug-in, for instance when it is loading effect programs and banks. Steinberg will keep a record of all these IDs so if you want to, before releasing a plug-in ask us if anyone else has already said they use to the ID. Otherwise be inventive; you can be pretty certain that someone already has used 'DLAY'.

The call `canMono()` tells the host that it makes sense to use this plug-in as a 1 in / 2 out device. When the host has mono effects sends and has a stereo buss for the effects returns, then the host can check this flag to add it to the list of plug-ins that could be used in this way.

In this instance the host can check to see if a stereo plug is selected, and can decide to feed both inputs with the same signal.

> For the gain plug, this makes sense (although it would be much more efficient to have a separate 1 in / 2 out version), but a stereo width enhancer, for instance, really doesn't make sense to be used with both inputs receiving the same signal; such a plug should definitely not call `canMono()`.

We then tell host that we support replacing processing with the call `canProcessReplacing()` (see 'Basic Concepts' for more information).

Finally, the one and only program name gets set to "Default" with `strcpy(programName, "Default")`. The program name is displayed in the rack, and can be edited by the user.

This gain plug-in only has one parameter so its handling of the hosts requests for parameter information are very simple, in that the index to the parameter required is simply ignored, and either the current value of our fGain member is set or returned.

```
void AGain::setParameter(long index, float value)
{
        fGain = value;
}

float AGain::getParameter(long index)
{
        return fGain;
}
```

In this case the parameter directly reflects the 'value' used in our processing method. When the user 'turns the dial' on a generic plug interface it is this value that gets swept between 0.0 & 1.0 as the `setParameter()` function gets called. If you want a different relationship between the parameters that are automated by the host and what is actually used in your processing method, then that's up to you to do the mapping.

When using the default host interface to provide an interface for a plug-in, as we are doing here, it is necessary that effect object overrides the following functions to allow the default interface a chance to display useful values to the user. Note, here again we are ignoring the parameter 'index' while we only have one parameter.

The Default User-Interface?
The a VST Host provides a method of displaying parameter values. The plug-in only needs to provide character strings representing its paramter values, labels & names. The Host then is free to display and edit these parameters as it wishes.

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

void AGain::getParameterDisplay(long index, char *text)
{
        dB2string(fGain, text);
}

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

Please be aware that the string lengths supported by the default VST interface are normally limited to 24 characters. If you copy too much data into the buffers provided, you will break the host application. Don't do it.

As you can see, the `AudioEffectX` base-class has some default methods that are useful for parameter to value and parameter to string conversions, for instance here to display a dB value string. `dB2string(fGain, text)` . Please check the `audioeffectX.hpp` & `AudioEffect.h` files for the full compliment.

So that's it ? !

Yes, that's all you need to know to create a fully working VST plug-in. If you already have an idea for an algorithm or audio process you could test it out right now.

Actually there are two different kinds of processes that you can support, so it's probably a good idea to keep on reading a little further.

Anyway that's all there is to initialization, parameter handling, and user interfacing in its simplest form.

Now you should proceed to the ADelay example, which is similar to AGain but has more parameters, and provides the basis for a plug-in with its own graphical user interface.

Basic Programming Examples
A simple delay plug-in.

This next coding example builds on what was learnt from the previous example, the simple gain plug-in. Please read this first if you have not already done so, because only what's new is explained. Apart from offering a somewhat more meaningful audio process, the main reason for this example is to learn more about how multiple parameters are handled and how a plug manages more than one snapshot of user settings.

Example Code : ADelay

Basic Source Files Required
ADelay.cpp
ADelay.hpp
ADelayMain.cpp

First, let's look at the simple and short declaration of the ADelay class and it's companion class ADelayProgram:

```
#include "AudioEffect.hpp"
#include <string.h>

enum
{
    kDelay,
    kFeedBack,
    kOut,

    kNumParams
};




class ADelayProgram
{
public:
    ADelayProgram();
    ~ADelayProgram() {}

private:
    friend class ADelay;
    float fDelay, fFeedBack, fOut;
    char name[24];
};
```

```
class ADelay : public AudioEffect
{
public:
    ADelay(audioMasterCallback audioMaster);
    ~ADelay();

    virtual void process(float **inputs, float **outputs, long sampleframes);
    virtual void processReplacing(float **inputs, float **outputs, long sampleFrames);
    virtual void setProgram(long program);
    virtual void setProgramName(char *name);
    virtual void getProgramName(char *name);
    virtual void setParameter(long index, float value);
    virtual float getParameter(long index);
    virtual void getParameterLabel(long index, char *label);
    virtual void getParameterDisplay(long index, char *text);
    virtual void getParameterName(long index, char *text);
    virtual float getVu();
    virtual void suspend();

private:
    void setDelay(float delay);
    void subProcess(float *in, float *out1, float *out2,
        float *dest, float *del, long sampleframes);
    void subProcessReplacing(float *in, float *out1, float *out2,
        float *dest, float *del, long sampleframes);

    ADelayProgram *programs;
    float *buffer;
    float fDelay, fFeedBack, fOut;
    float vu;
    long delay;
    long size;
    long inPos;
    long outPos;
};
```

So what's new?

So after scanning through that you probably can see what's familiar from the original gain plug-in but also what's new. Now one more time, step by step…

```
enum
{
    kDelay,
    kFeedBack,
    kOut,

    kNumParams
};
```

This enumerates all parameters that are accessible to the user. It's a good habit to use such an enumerator for any plug-in. If our final linked plug-in was dropped into the host's VST Plug-in folder, then the

parameters we would see would be representations of these, namely the delay time, the output to input feedback amount, and the output level as parameters.

> While we are on the topic of good ideas, we would recommend having at least one representation of these parameters in your plug-ins main class that corresponds to the generic parameter format. In other words, for each of these parameters declared in the enumeration there should a (similarly named) parameter that holds it's value, where the value is a float type between the values 0.0 and 1.0 inclusive.

The handling of programs, the snapshots of user settings, is going to need a container for each set of parameters. We define a class to handle this.

```
class ADelayProgram
{
public:
    ADelayProgram();
    ~ADelayProgram() {}

private:
    friend class ADelay;
    float fDelay, fFeedBack, fOut;
    char name[24];
};
```

The members, `fDelay`, `fFeedBack`, and `fOut` are the generic representations of the parameters as stored in this container. They are not themselves the parameters used in the audio process but places where they will be copied from when the user selects another program. At this point the user can edit these values using the controls provided by the host using `setParameter()` & `getParameter()`.

The same is true of the name, if this program was to be selected by the user, after the plug-in has copied it from the container, the user can use the host application to edit the name, which is achieved with `getProgramName()` and `setProgramName()`.

We will omit the method declarations and have a look at their implementation instead, firstly the ADelayProgram constructor:

```
ADelayProgram::ADelayProgram()
{
    fDelay = 0.5;
    fFeedBack = 0.5;
    fOut = 0.75;
    strcpy(name, "Init");
}
```

This simply sets the default parameter values for all programs. You might want to assign different values (for instance, from a table or array) for each program, in order to provide some useful presets. That's all there is to the ADelayProgram class.

So, back to the constructor of the ADelay class…

```
ADelay::ADelay(audioMasterCallback audioMaster) : AudioEffectX(audioMaster, 2, kNumParams)
{
    size = 44100;
    buffer = new float[size];
    programs = new ADelayProgram[numPrograms];
    fDelay = fFeedBack = fOut = vu = 0;
    delay = inPos = outPos = 0;
    if(programs)
        setProgram(0);
    setNumInputs(1);
    setNumOutputs(2);
    hasVu();
    canProcessReplacing();
    setUniqueID('ADly');
    suspend();       // flush buffer
}
```

When the ADelay class is constructed, firstly the base-class AudioEffectX is constructed. As you will remember from the AGain example, one of AudioEffectX's arguments is the number of programs the plug-in has (here set to 2). Once the base class is constructed, during the construction of the ADelay object, the member variable numPrograms is valid and is used to construct a set of ADelayPrograms.

Because this is a delay process we are going to need some memory to store samples. The parameter 'size' is set to 44100 to provide a maximum of one second of delay, and then a buffer of floats is allocated.

Probably a statement such as size = getSampleRate() would be better, but as we don't react to changes in sample rate in this simple example, and then anyway we would have to re-allocate the buffer accordingly etc. That's not the point of this example.

Next all the parameters that are used in the audio process are set to zero as a matter of good behavior, but actually they will get set to an initial value when setProgram() gets called if delay programs were successfully created.

Just as in the AGain example, `setUniqueID()` is very important, you must set this to a value as unique as possible as the host uses this to identify the plug-in, for instance when it is loading effect programs and banks.

Finally, we state that we support vu meters, and in-place processing as with the gain example. The destructor just takes care of cleaning up (remember, we're in the Hosts' context):

```
ADelay::~ADelay()
{
    if(buffer)
        delete buffer;
    if(programs)
        delete[] programs;
}
```

Changing Programs

Next we should have a look at the `setProgram()` method. This is called whenever the user, or the host as a result of automation activity, wants to change the current active program in the plug-in. This (re-) initializes both generic, internal parameters, and variables:

```
void ADelay::setProgram(long program)
{
    ADelayProgram * ap = &programs[program];

    curProgram = program;
    setParameter(kDelay, ap->fDelay);
    setParameter(kFeedBack, ap->fFeedBack);
    setParameter(kOut, ap->fOut);
}
```

It reads the parameters from the selected program, makes that the current program, and calls `setParameter()` for each of the indexed parameters, with the new values, just as if the user would have set them by hand. It doesn't matter why the parameters are being changed, user selection, automation, program change, the same `setParameter()` method gets called.

Changing Parameters

Well, while we are on the subject of `setParameter()` we should look at its implementation.

```
void ADelay::setParameter(long index, float value)
{
    ADelayProgram * ap = &programs[curProgram];

    switch(index)
    {
```

```
        case kDelay:    setDelay(value);                       break;
        case kFeedBack: fFeedBack = ap->fFeedBack = value;     break;
        case kOut:      fOut = ap->fOut = value;               break;
    }
    if(editor)
        editor->postUpdate();
}
```

We'll get back to `setDelay()` later. The `setParameter()` function is also called from the Host, it's important that we store the parameters back into the current program, to maintain the currently edited state of any particular program.

Both `getParameter()` and `setParameter()` always refer to the current program.

> But notice that last section referring to the editor. When a parameter gets changed you want the user's display to be updated as well. And here is exactly where you should NOT do it. The reason is that `setParameter` can & will get called from interrupt and drawing graphics from an interrupt (on the Mac!) is basically impossible. And we will leave it at that. There are actually performance reasons why deferring graphic updates to the user-interface thread is a good idea. What you actually do is post a message to the user-interface that something or everything needs updating.

To complete the picture, here's the implementation of `getParameter()`...

```
float ADelay::getParameter(long index)
{
    float v = 0;

    switch(index)
    {
        case kDelay:    v = fDelay;         break;
        case kFeedBack: v = fFeedBack;      break;
        case kOut:      v = fOut;           break;
    }
    return v;
}
```

That's simple enough. Compared with the version in AGain where we had only one parameter, we use the index value to return the appropriate parameter. Alternatively we could have returned values that referred to the program index by `curProgram`, but because we were so diligent in how `setProgram()` also updated the current program, there was no need.

There's also `getProgramName()` and `setProgramName()`: Remember the size of the character arrays, and the consequences of overfilling them?

```
void ADelay::setProgramName(char *name)
{
    strcpy(programs[curProgram].name, name);
}

void ADelay::getProgramName(char *name)
{
    strcpy(name, programs[curProgram].name);
}
```

Fairly straightforward as well. And as we have more than one parameter, we use switch statements where the Host asks us for strings. We still don't have our own interface yet and must support the host's efforts in providing us with the generic interface.

> **Even if you do provide your own editor you should still have a fully set of 'string' interfaces, so the host application can graphically edit your automated parameters, with their names and labels.**

```
void ADelay::getParameterName(long index, char *label)
{
    switch(index)
    {
        case kDelay:    strcpy(label, " Delay  ");  break;
        case kFeedBack: strcpy(label, "FeedBack");  break;
        case kOut:      strcpy(label, " Volume ");  break;
    }
}

void ADelay::getParameterDisplay(long index, char *text)
{
    switch(index)
    {
        case kDelay:    long2string(delay, text);        break;
        case kFeedBack: float2string(fFeedBack, text);   break;
        case kOut:      dB2string(fOut, text);           break;
    }
}

void ADelay::getParameterLabel(long index, char *label)
{
    switch(index)
    {
        case kDelay:    strcpy(label, "samples ");  break;
        case kFeedBack: strcpy(label, " amount ");  break;
        case kOut:      strcpy(label, "   dB   ");  break;
    }
}
```

This just a small detail really but the `AudioEffectX` base class provides a number of conversion functions, we have already used `dB2string()` in the gain plug-in, but here `long2string()` and `float2string()` are used as well.

We are getting close the to end of the new stuff here. Have a look at the method `suspend()`.

```
void ADelay::suspend()
{
    memset(buffer, 0, size * sizeof(float));
}
```

This method `suspend()` is called when the effect is turned off by the user; the buffer gets flushed because otherwise pending delays would sound again when the effect is switched on next time. It's not essential to do this but it would be rather strange not to, but there's no accounting for taste.

You could also flush your 'buffers' in the `resume()` method, for completeness.

One more. While ADelay was being constructed `hasVu()` was called.

> The method `hasVu()` tells the host that this plug-in can supply data about the current VU level. That is, your plug-in can calculate VU values while it's processing, and if the host asks about the current VU level with `getVU()`, then it's returned from the plug-in to the host.

```
float ADelay::getVu()
{
    float cvu = vu;

    vu = 0;
    return cvu;
}
```

And Finally, the Process.

```
void ADelay::process(float **inputs, float **outputs, long sampleframes)
{
    float *in = *inputs;
    float *out1 = outputs[0];
    float *out2 = outputs[1];
    long frames, remain;
    remain = sampleframes;
    while(remain > 0)
    {
```

```
            if(size - inPos < remain || size - outPos < remain)
            {
                if(size - inPos < size - outPos)
                    frames = size - inPos;
                else
                    frames = size - outPos;
            }
            else
                frames = remain;

            subProcess(in, out1, out2, buffer + inPos, buffer + outPos, frames);
            in += frames;
            out1 += frames;
            out2 += frames;
            inPos += frames;
            if(inPos >= size)
                inPos -= size;
            outPos += frames;
            if(outPos >= size)
                outPos -= size;
            remain -= frames;
        }
}


//-------------------------------------------------------------------------------
// replacing
void ADelay::subProcessReplacing(float *in, float *out1, float *out2,
    float *dest, float *del, long sampleframes)
{
    float d, feed = fFeedBack, vol = fOut, xn;
    float cvu = vu;
    in--;
    out1--;
    out2--;
    dest--;
    del--;
    while(--sampleframes >= 0)
    {
        xn = *++in;        // pre-increment is fast on ppcs
        d = *++del * vol;
        if(d > cvu)        // positive only...
            cvu = d;
        xn += d * feed;
        *++dest = xn;
        *++out1 = d;       // store to buss
        *++out2 = d;
    }
    vu = cvu;
}
```

There is a buffer of (at least) the maximum number of samples to be delayed, reflected by the size member variable. In order to get the buffer wrap correctly, and because of efficiency reasons, the audio process is divided into the actual process, and a subProcess(). The idea is that the

subProcess() is called for each contiguous part in the buffer that can be processed at a time, in order to keep the number of cycles in the inner loop small.

The details of the processes and are left for you to figure out by yourself. All in all, this is a simple delay with feedback.

At this point, we have seen enough to understand the fundamental mechanisms of writing VST plug-ins: Now complete with multiple parameters, that can be correctly automated and saved as programs, and displayed with labels and value units by the host. Not bad for a few lines of code.

Both the gain plug-in and this simple delay plug-in have used the host's default parameter handling for changing and displaying values. Nothing wrong with that, but VST plug-ins can also provide their own user interface, so that they can do customize their look and feel. In the next example code we will look at how the delay plug-in can be extended to do just that.

Basic Programming Examples
A delay plug-in with its own user interface
Doing it the hard way.

Background information
How Cubase VST handles Plug-ins with their own interface.

The sequence is as follows: first, when VST gets to know that your plug-in supports its own editor as the user selects it in the Effects Rack instead of displaying the standard text based interface using the alpha-dial approach, it installs a special 'Rack Expander' panel. An Edit button is displayed as part of the panel. If the user hits this button the plug-in's own editor is opened. Exactly how that happens is explained in this section.

Doing it yourself

For far we have had it really easy, we have written plug-ins that can be compiled for different platforms, but the concepts have remained exactly the same. But as soon as we want to start providing our own user interface we are confronted with the platform specifics of different windows handling and event handing models.

For this very reason we created the VSTGUI libraries. These are available for all supported platforms and make the handling of user interaction with a graphic-rich user interface also cross platform. With only one proviso: You need to be able to convert your source graphics from one platform format to another. But for this there are tools that someone else has already debugged, which is how we like it. Now you could jump straight into the next chapter, where the easy method is presented. But you won't know what the others will know.

Still here? Good. This discussion will actually concentrate on the issues raised by the Mac & Windows systems just to keep things under control. You should have read about the Gain Plug-In and you must know how the ADelay example works, as ADelayEdit is based on the latter example.

Defining Steps.

ADelayEdit is an example of a plug-in which maintains its own editor window, as opposed to the gain and delay examples discussed previously, which work on the built-in 'string interface' provided by a VST host.

> The concept is as follows. Just as all the plug-ins are derived from `AudioEffectX`, all the editors are derived from the class `AEffEditor`. If a plug-in has its own editor, it instanciates one when it itself is constructed. The editor gets passed the plug-in object as a parameter to its constructor, and the editor uses this to notify the effect object of its presence.

Sounds simple: So let's see how that mechanism works. Firstly look at the following class declaration. This is `AEffEditor`, from which all editors are derived.

```
class AEffEditor
{
friend class AudioEffect;

public:
            virtual void update() {}
            virtual void postUpdate() {updateFlag = 1;}

protected:
            AEffEditor(AudioEffect *effect) {this->effect = effect;}
            virtual ~AEffEditor() {}

            virtual long getRect(ERect **rect) {*rect = 0; return 0;}
            virtual long open(void *ptr) {systemWindow = ptr; return 0;}
            virtual void close() {}
            virtual void idle() {if(updateFlag) {updateFlag = 0; update();}}

            #if MAC
            virtual void draw(ERect *rect) {rect = rect;}
            virtual long mouse(long x, long y) {x = x; y = y; return 0;}
            virtual long key(long keyCode) {keyCode = keyCode; return 0;}
            virtual void top() {}
            virtual void sleep() {}
            #endif

            AudioEffect *effect;
            void *systemWindow;
            long updateFlag;
};
```

#if MAC ???
Isn't this meant to the cross platform?

On the Macintosh the window your plug-in receives is created by the host, and the GUI events your plug-in receives are received globally by the host application and posted on to the plug-in with these #if MAC methods.

On the PC, your Plug-in registers its own window class, opens its own window and the then collects its events from the `WndProc` it defined as part of its window class. Sure you need a bit of windows programming knowledge to do this, but it's really elementary stuff. And it's really stuff you could totally ignore if you used the VSTGUI Libraries.

Back to the plot, we were describing a delay plug-in with its own editor. As usual we define a class for the new plug-in. This ADelayEdit class inherits directly from ADelay, thus both the declaration and constructor methods are very simple…

```
class ADelayEdit : public ADelay
{
public:
            ADelayEdit(audioMasterCallback audioMaster);
        ~ADelayEdit();
};

ADelayEdit::ADelayEdit(audioMasterCallback audioMaster) : ADelay(audioMaster)
{
            setUniqueID('ADlE');
            editor = new ADEditor(this);
            if(!editor)
                  oome = true;
}
```

As you can see from the constructor, ADelayEdit merely adds an editor ADEditor. This editor is derived from the base class AEffEditor as described above. Looking at the constructor of the editor, you can see the Audio Effect (our plug-in) being used as a parameter, and the plug-in being formally informed about the editor.

```
ADEditor::ADEditor (AudioEffect *effect)
        : AEffEditor (effect)
{
            effect->setEditor (this);        // notify effect that 'this is the editor'
}
```

The ADelayEdit destructor has nothing to do, as AudioEffect takes care of deleting the editor when the plug-in is removed.

```
ADelayEdit::~ADelayEdit()
{
            // the editor gets deleted by the
            // AudioEffect base class
}
```

All that's left now is the ADEditor class, which provides a simple 3 fader panel to allow the user to adjust delay, feedback, and output level in a the Host window. This will be fully described in the VSTGUI library version that follows shortly.

Macintosh Outline

As was said before, VST hosts on the Mac create a window for the plug-in's editor and then passes all the events on to the editor. There are a few functions that negotiate this process. Here they are:

`long getRect(ERect **rect)`
Host is asking the editor how big it wants its window to be.

`long open(void *ptr)`
Host is about to open a window for the editor

`void close()`
Host is about to close the window for the editor.

`void idle()`
Host has 'idling' events for the editor.

`void draw(ERect *rect)`
Host says this area of the editor window needs updating.

`long mouse(long x, long y)`
Host says the mouse was clicked here.

`long key(long keyCode)`
Host says what key was pressed.

`void postUpdate()`
Here you can flag any updates that will later be actioned in update()

`void update()`
Update your graphics during this call if you have flagged any updates.
This gets called during idle().

Windows Outline

The handling is a little different for Windows. To recap; the reason is that the window the plug-in's editor will appear in is created by the plug-in, when the host sends the `open()` command to the plug-ins editor. From this moment on, until the host sends the plug-in's editor the `close()` command, the plug-in editor manages its own event queue associated with the window. What follows is an outline of how that is done...

Here comes another code fragment.

```
extern HINSTANCE hInstance;
int useCount = 0;

HWND CreateFader (HWND parent, char* title, int x, int y, int w, int h, int min, int max);
LONG WINAPI WindowProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);
```

```
//-------------------------------------------------------------------------------
ADEditor::ADEditor (AudioEffect *effect)
:       AEffEditor (effect)
{
        effect->setEditor (this);
}

//-------------------------------------------------------------------------------
ADEditor::~ADEditor ()
{
}

//-------------------------------------------------------------------------------
long ADEditor::getRect (ERect **erect)
{
        static ERect r = {0, 0, kEditorHeight, kEditorWidth};
        *erect = &r;
        return true;
}

//-------------------------------------------------------------------------------
long ADEditor::open (void *ptr)
{
        // Remember the parent window
        systemWindow = ptr;

        // Create window class, if we are called the first time
        useCount++;
        if (useCount == 1)
        {
                WNDCLASS windowClass;
                windowClass.style = 0;
                windowClass.lpfnWndProc = WindowProc;
                windowClass.cbClsExtra = 0;
                windowClass.cbWndExtra = 0;
                windowClass.hInstance = hInstance;
                windowClass.hIcon = 0;
                windowClass.hCursor = 0;
                windowClass.hbrBackground = GetSysColorBrush (COLOR_BTNFACE);
                windowClass.lpszMenuName = 0;
                windowClass.lpszClassName = "DelayWindowClass";
                RegisterClass (&windowClass);
        }

        // Create our base window
        HWND hwnd = CreateWindowEx (0, "DelayWindowClass", "Window",
                WS_CHILD | WS_VISIBLE,
                0, 0, kEditorHeight, kEditorWidth,
                (HWND)systemWindow, NULL, hInstance, NULL);

        SetWindowLong (hwnd, GWL_USERDATA, (long)this);

        // Create three fader controls
        delayFader =    CreateFader (hwnd, "Delay",    x,    y,    w,    h,    0, 100);
        feedbackFader = CreateFader (hwnd, "Feedback", x,    y,    w,    h,    0, 100);
```

```
            volumeFader =   CreateFader (hwnd, "Volume",   x,     y,     w,     h,    0, 100);


            return true;
}


void ADEditor::close ()
{
            useCount--;
            if (useCount == 0)
            {
                    UnregisterClass ("DelayWindowClass", hInstance);
            }
}
void ADEditor::idle ()
{
            AEffEditor::idle ();
}


void ADEditor::update()
{
        SendMessage (delayFader,    TBM_SETPOS, (WPARAM) TRUE, (LPARAM) effect->getParameter (kDelay)
        SendMessage (feedbackFader, TBM_SETPOS, (WPARAM) TRUE, (LPARAM) effect->getParameter
(kFeedBack));
        SendMessage (volumeFader,   TBM_SETPOS, (WPARAM) TRUE, (LPARAM) effect->getParameter (kOut));
}


void ADEditor::setValue(void* fader, int value)
{
            if (fader == delayFader)
                    effect->setParameterAutomated (kDelay, (float)value / 100.f);
            else if (fader == feedbackFader)
                    effect->setParameterAutomated (kFeedBack, (float)value / 100.f);
            else if (fader == volumeFader)
                    effect->setParameterAutomated (kOut, (float)value / 100.f);
}


HWND CreateFader (HWND parent, char* title, int x, int y, int w, int h, int min, int max)
{
        HWND hwndTrack = CreateWindowEx (0, TRACKBAR_CLASS, title,
            WS_CHILD | WS_VISIBLE |
                    TBS_NOTICKS | TBS_ENABLESELRANGE | TBS_VERT,
          x, y, w, h, parent, NULL, hInstance, NULL);

        SendMessage (hwndTrack, TBM_SETRANGE, (WPARAM ) TRUE, (LPARAM) MAKELONG (min, max));
        SendMessage (hwndTrack, TBM_SETPAGESIZE, 0, (LPARAM) 4);
        SendMessage (hwndTrack, TBM_SETPOS, (WPARAM) TRUE, (LPARAM) min);
            return hwndTrack;
}


LONG WINAPI WindowProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
            switch (message)
            {
            case WM_VSCROLL:
                    {
                            int newValue = SendMessage ((HWND)lParam, TBM_GETPOS, 0, 0);
```

```
                        ADEditor* editor = (ADEditor*)GetWindowLong (hwnd, GWL_USERDATA);
                        if (editor)
                                editor->setValue ((void*)lParam, newValue);
                }
                break;
        }
        return DefWindowProc (hwnd, message, wParam, lParam);
}
```

Walking on through

That may seem like a lot of code compared with the Mac example but most of it is either
elementary windows stuff or deals with the handling associated with the 3 windows stock objects -
the scroll bars used as controls. Let's annotate the highlights and ignore the lowlights.

```
//-------------------------------------------------------------------------
ADEditor::ADEditor (AudioEffect *effect)
:       AEffEditor (effect)
{
        effect->setEditor (this);
}

//-------------------------------------------------------------------------
ADEditor::~ADEditor ()
{
}
```

OK, that's just as we described earlier: The Plug-in instanciates an editor which is derived from
`AeffEditor`. The `AudioEffectX` is passed as a parameter to the editor's constructor, and the effect
(our plug-in class) is notified about the editor. The destructor of the editor class does nothing.

```
//-------------------------------------------------------------------------
long ADEditor::getRect (ERect **erect)
{
        static ERect r = {0, 0, kEditorHeight, kEditorWidth};
        *erect = &r;
        return true;
}
```

Here the host requests the size of the editor. The static rectangle structure is filled with value from
enumerated constants from our editor sources.

Now comes the big window thing.

```
//-------------------------------------------------------------------------
long ADEditor::open (void *ptr)
{
        // Remember the parent window
        systemWindow = ptr;

        // Create window class, if we are called the first time
```

```
        useCount++;
        if (useCount == 1)
        {
                WNDCLASS windowClass;
                windowClass.style = 0;
                windowClass.lpfnWndProc = WindowProc;
                windowClass.cbClsExtra = 0;
                windowClass.cbWndExtra = 0;
                windowClass.hInstance = hInstance;
                windowClass.hIcon = 0;
                windowClass.hCursor = 0;
                windowClass.hbrBackground = GetSysColorBrush (COLOR_BTNFACE);
                windowClass.lpszMenuName = 0;
                windowClass.lpszClassName = "DelayWindowClass";
                RegisterClass (&windowClass);
        }

        // Create our base window
        HWND hwnd = CreateWindowEx (0, "DelayWindowClass", "Window",
                WS_CHILD | WS_VISIBLE,
                0, 0, kEditorHeight, kEditorWidth,
                (HWND)systemWindow, NULL, hInstance, NULL);

        SetWindowLong (hwnd, GWL_USERDATA, (long)this);

        // Create three fader controls
        delayFader =    CreateFader (hwnd, "Delay",    x,    y,    w,    h,    0, 100);
        feedbackFader = CreateFader (hwnd, "Feedback", x,    y,    w,    h,    0, 100);
        volumeFader =   CreateFader (hwnd, "Volume",   x,    y,    w,    h,    0, 100);

        return true;
}
```

Notice first, the useCount variable, which is in file scope and is initialized to 0. When the function open() is called and the pre-incremented useCount is equal to 1, then we register the window class, create a window from this class. Open() can get called more than once as multiple effects are instanciated but the window class gets registered just once.

Notice the parent window for our window was passed as a parameter to the open() function. Other important details are the lpfnWndProc member of the window class. We take control of the event handling for this window so we define the window procedure, more on this shortly. The 'this' pointer for this editor is stuffed into the user-data area for the instanciated window.

Finally we call our own CreateFader() function three times to make the controls for the editor. This function is just making controls of the TRACKBAR_CLASS and returning the window handles of these objects that are stored as members of our editor class.

```
void ADEditor::close ()
{
        useCount--;
        if (useCount == 0)
```

```
        {
                UnregisterClass ("DelayWindowClass", hInstance);
        }
}
```

Here is the reverse process, when the pre-decremented variable useCount equals 0 then the last of any windows created by this editor has been closed and the window class can be unregistered.

```
void ADEditor::idle ()
{
        AEffEditor::idle ();
}
```

The host will call the idling method of our editor regularly. Here our have an opportunity to update any displays or respond to pending user interface issues, such as meters etc. You must still call the base class function however. What is important is that this is purely a user-interface issue: It has nothing to do with the audio process. Making part of your audio process dependent on the GUI idling events is asking for trouble.

```
void ADEditor::update()
{
    SendMessage (delayFader,    TBM_SETPOS, (WPARAM) TRUE, (LPARAM) effect->getParameter (kDelay));
    SendMessage (feedbackFader, TBM_SETPOS, (WPARAM) TRUE, (LPARAM) effect->getParameter (kFeedBack));
    SendMessage (volumeFader,   TBM_SETPOS, (WPARAM) TRUE, (LPARAM) effect->getParameter (kOut));
}
```
When the host wants to update the appearance of the controls, it calls the update method of the editor. Here the faders we created and stored away are just updated with the parameters that are fetched properly from the effect's getParameter() method.

Now to round up the process lets look at the WindowProc.

```
LONG WINAPI WindowProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
        switch (message)
        {
        case WM_VSCROLL:
            {
                    int newValue = SendMessage ((HWND)lParam, TBM_GETPOS, 0, 0);
                    ADEditor* editor = (ADEditor*)GetWindowLong (hwnd, GWL_USERDATA);
                    if (editor)
                            editor->setValue ((void*)lParam, newValue);
            }
            break;
        }
        return DefWindowProc (hwnd, message, wParam, lParam);
}
```

What's important here is the fact that the Window Procedure belongs to all instances of windows that we create with our registered window class, and there could be multiple instances of our plug-in and it's editor.

In this case our Window procedure is only handling WM_VSCROLL messages. Because we have only used windows stock objects, calling DefWindowProc does everything we need to do, but this is precisely the point where you would intercept any other WM_X messages that are relevant to your plug-in.

So the Window Procedure receives WM_VSCROLL, but to whom do they belong? Well remember that we stuffed the 'this' pointer to the editor object into the user data area of the window when we created it. This is simply recovered…

```
ADEditor* editor = (ADEditor*)GetWindowLong (hwnd, GWL_USERDATA);
```

…and then used to reference our editor. A member function `setValue()` is called with the pointer to the scrollbar that was touched, and the new value of the scroll bar that was recovered before we knew to whom it belonged. The pointer to the scroll bar is compared with the values we stored when creating the faders, and the effects is properly informed about the changes with the effect's `setParameterAutomated()` method.

```
void ADEditor::setValue(void* fader, int value)
{
       if (fader == delayFader)
             effect->setParameterAutomated (kDelay, (float)value / 100.f);
       else if (fader == feedbackFader)
             effect->setParameterAutomated (kFeedBack, (float)value / 100.f);
       else if (fader == volumeFader)
             effect->setParameterAutomated (kOut, (float)value / 100.f);
}
```

An important thing to notice is that if the user changes a parameter in your editor (which is out of the host's control –because we are not using the default string based interface), you should call…

```
setParameterAutomated(index, (float)newValue);
```

…which is an `AEffEditor` method. This ensures that the host is notified of the parameter change, which allows it to record these changes for automation. Note that `setParameterAutomated()` calls setParameter().

No Pain No Gain.

That completes the description of how a plug-ins editor manages its own window. OK, we only used Windows stock-objects, and no fancy user interface controls and displays that make VST plug-ins so attractive to the end user. But that's only an issue of scale.

Now we have looked to the bare issues surrounding the platform specific implementation custom editors for VST Plug-ins for both Macintosh and Windows, we can now move on to the VSTGUI Libraries. These libraries, that are available for all four supported platforms, demonstrate how providing a custom user interface can also be as cross-platform as writing a VST plug-in the first place.

Basic Programming Examples
A delay plug-in with its own user interface
Taking it Easy – Using the VSTGUI Libraries.

This section covers the implementation of a Plug-ins own user-interface but this time using the VSTGUI Libraries. These are a set of cross-platform tools that encapsulate actually what we have already seen in the previous example. Before any of the following will make any sense at all you must be familiar with the previous example files and concepts therein. No excuses.

This section only forms an introduction to the complete VSTGUI documentation that is included as part of this SDK. The VSTGUI documentation is provided as a full set of html files where you can point and click your way through the ways and means of the VSTGUI Libraries. Thoroughly recommended browsing.

As we said, before you read this document you should have read about the Gain Plug-In, and you must know how the ADelay example works, as this new ADelayEdit example is based on the latter example (it actually inherits from the ADelay class, which you should be familiar with).

This new ADelayEdit implementation is an example of a plug-in which maintains its own editor window, as opposed to the gain and delay examples discussed previously, which work on the built-in 'Multi-effects' string interface. The user interface is entirely de-coupled from the audio processing, and implemented using the VST GUI.

The `ADelayEdit` class inherits directly from `ADelay`, thus both the declaration and constructor methods are very simple (ADelayEdit.hpp and ADelayEdit.cpp):

```
class ADelayEdit : public ADelay
{
public:
        ADelayEdit(audioMasterCallback audioMaster);
        ~ADelayEdit();
};

ADelayEdit::ADelayEdit(audioMasterCallback audioMaster) : ADelay(audioMaster)
{
        setUniqueID('ADlE');
        editor = new ADEditor(this);
}

ADelayEdit::~ADelayEdit()
{
        // the editor gets deleted by the
        // AudioEffect base class
}
```

```
void ADelayEdit::setParameter (long index, float value)
{
    ADelay::setParameter (index, value);
    if (editor)
        ((AEffGUIEditor*)editor)->setParameter (index, value);
}
```

As you can see from the constructor, ADelayEdit 'merely' adds an editor (ADEditor), and overrides the setParameter() method so to distribute changes therefore to the 'value' objects. It's a bit out of the scope of this brief description but the VSTGUI makes use of the Model-Value-Controller principle. There are value objects that hold the value itself and controller objects that act upon the value object. We will have to leave that to the main VSTGUI documentation.

The ADelayEditor destructor has nothing to do, as AudioEffect takes care of deleting the editor when the plug-in is removed.

All that's left now is the ADEditor class, which provides a simple 3 fader panel to allow the user to adjust delay, feedback, and output level in the host window. It is based on the VSTGUI. The ADEditor class inherits from 2 classes, namely, AEffGUIEditor and CControlListener (both in vstgui.h).

```
class AEffGUIEditor : public AEffEditor
{
public :

    AEffGUIEditor (AudioEffect *effect);
    virtual ~AEffGUIEditor ();

    virtual void setParameter (long index, float value) { postUpdate (); }
    virtual long getRect (ERect **rect);
    virtual long open (void *ptr);
    virtual void idle ();
    virtual void draw (ERect *rect);
#if MAC
    virtual long mouse (long x, long y);
#endif

    // get the current time (in ms)
    long getTicks ();

    // feedback to appli.
    void doIdleStuff ();

    // get the effect attached to this editor
    AudioEffect *getEffect () { return effect; }
};
```

AEffGUIEditor extends AEffEditor (the VST plug-ins version 1.0 class, in AEffEditor.hpp). When the application gets to know that your plug supports its own editor, it will implement a method to call your editor (like the 'Edit' button on a Cubase rack display), which:

• calls the AEffGUIEditor method getRect() to get the Rect from the plug which states where the window should appear, and how large it should be.

• creates and opens a window at the requested position, and with the requested dimensions

• calls the AEffGUIEditor open() method

• calls the AEffGUIEditor draw() method for MacOS

As time goes by, Mac OS events for this window are passed to the according AEffGUIEditor methods (`mouse()`, `key()`,`top()`, `sleep()`) with the VSTGUI, simply override the relevant methods). All other platforms use a dispatcher class to collect and dispatch similar GUI based events to your plug-in's editor.

An important thing to notice is that if the user changes a parameter in your editor (which is out of the application's control), namely in the `valueChanged()` method, you should call…

```
setParameterAutomated(index, (float)newValue);
```

…which is a AEffEditor method. This ensures that the application is notified of the parameter change, which allows it to record these changes for automation purposes. Note also that setParameterAutomated() calls setParameter().

Referring now to the Source files for this project…

The ADEditor class is a very simple example based on `AEffGUIEditor`. A background bitmap is created, and 3 faders with their background and handle bitmaps are installed. Also all 3 parameters get a control for text display of these values. The functions `setParameter()`, and `valueChanged()`, show how the update mechanism works; and that was it again. Have a look at the example code. It shows some basic concepts of how to use the VSTGUI.

This was the final example for the basis plug-in concepts. The powerful VSTGUI Library makes your plug entirely portable, so you should compile it on all platforms (or find somebody who can). Good Luck.

What's New in VST 2.0
Introduction – finding your way around

If you already are familiar with the VST 1.0 specification here is an overview of what's new in the 2.0 SDK and where you should look for further details.

> **Note: It's important to keep in mind that the VST 2.0 specification is completely downwardly compatible with the VST 1.0 specification.**

The plug-ins created under the VST 1.0 specification are all derived from the class AudioEffect. The VST 2.0 specification defines a new class AudioEffectX upon which all plug-ins are derived. The new class AudioEffectX inherits from the existing AudioEffect class, so that any 1.0 plug-in can be re-complied using the AudioEffectX class without modification.

General new features of version 2.0

VST Instruments
> Plug-ins that can receive MIDI events from the host application and then 'render' audio however they please.

VstTimeInfo
> A protocol defining how a plug-in can obtain time & tempo information from the host application in a variety of formats.

VstOffline Processing
> A complete interface definition between host and plug-in defining off-line audio processes.

Apart from these new capabilities, there is a whole batch of new methods on both the plug-in and application side of the VST Plug-in interface.

> A host application cannot make assumptions about the presence of the new 2.0 features of a plug-in. Similarly, a plug-in cannot assume a 2.0 feature is available from the host.
>
> There is a full set of property inquiry methods for both hosts and plug-ins to use to ascertain the environment in which either finds itself. Ignoring these enquiry methods and trying to access a 2.0 feature from a 1.0 host, or vice versa, will mean your plug-in or host application will break.
>
> It is not the end-users job to pick and choose which plug-ins can be supported by which host.

Where to find details of the 2.0 implementation

The new VST parameters, structures and constants can be found in the file **aeffectx.h** it makes reference to and expands upon the existing file **aeffect.h**

The VST 2.0 Audio Effect class can be seen in the file **audioeffectx.h** that similarly makes reference to and expand upon the existing file **audioeffect.hpp**

New Universal Basic Methods

There is a basic set of methods that your plug should support, even if you have a 1.0 version working already. These are all found in the new AudioEffectX class.

```
virtual bool getInputProperties (long index, VstPinProperties* properties);
virtual bool getOutputProperties (long index, VstPinProperties* properties);
virtual bool getProgramNameIndexed (long category, long index, char* text);
virtual bool copyProgram (long destination);
virtual bool getEffectName (char* name);
virtual bool getVendorString (char* text);
virtual bool getProductString (char* text);
virtual long getVendorVersion () {return 1;}
virtual long canDo (char* text);
```

You may copy the following code and modify it to fit your implementation:

Note: VstPinProperties is a structure defining connection properties of plug-ins. Details are to be seen in **aeffectx.h.**

```
//-------------------------------------------------------------------------
bool MyPlug::getInputProperties (long index, VstPinProperties* properties)
{
    if (index < kNumInputs)
    {
        sprintf (properties->label, "My %ld In", index + 1);
        properties->flags = kVstPinIsStereo | kVstPinIsActive;
        return true;
    }
    return false;
}

//-------------------------------------------------------------------------
bool MyPlug::getOutputProperties (long index, VstPinProperties* properties)
{
    if (index < kNumOutputs)
    {
        sprintf (properties->label, "My %ld Out", index + 1);
        properties->flags = kVstPinIsStereo | kVstPinIsActive;
        return true;
    }
    return false;
}
```

```
//----------------------------------------------------------------------
bool MyPlug::getProgramNameIndexed (long category, long index, char* text)
{
    if (index < kNumPrograms)
    {
        strcpy (text, programs[index].name);
        return true;
    }
    return false;
}


//----------------------------------------------------------------------
bool MyPlug::copyProgram (long destination)
{
    if (destination < kNumPrograms)
    {
        programs[destination] = programs[curProgram];
        return true;
    }
    return false;
}


//----------------------------------------------------------------------
bool MyPlug::getEffectName (char* name)
{
    strcpy (name, "MyPlug");
    return true;
}


//----------------------------------------------------------------------
bool MyPlug::getVendorString (char* text)
{
    strcpy (text, "My Company");
    return true;
}


//----------------------------------------------------------------------
bool MyPlug::getProductString (char* text)
{
    strcpy (text, "My Plug Soft Synth");
    return true;
}




//----------------------------------------------------------------------
long MyPlug::canDo (char* text)
{
    if (!strcmp (text, "receiveVstEvents"))
        return 1;
    if (!strcmp (text, "receiveVstMidiEvent"))
        return 1;
    return -1;
}
```

This set of canDo's (receiveVstEvents and receiveVstMidiEvent) is how Virtual Instruments (synths based on VST2.0) declare to Cubase VST what they are capable of, and what events they need to be notified of. We recommend having a look at audioeffectx.cpp for other 'canDo's'.

For a complete discussion of all new VST 2.0 features and detailed documentation, look at the new AudioEffectX class included in the source code files in this SDK.

What's new in VST 2.0 - VST Event Information
Sending MIDI information from the Host to a Plug-in

VST Events form a mechanism that allows a host application to send a stream of events to a VST 2.0 plug-in. VST Events contain information about the relationship between the position of the event in relation to the currently processed buffer received by process() or processReplacing().

Basically when a VST host is capable of sending VST Events, then a VST 2.0 Plug-in could use this information to control the audio processing that is undertaken.

Lets start by looking at the primary data structure, the VstEvent.

```
struct VstEvent          // a generic timestamped event
{
    long type;           // see enum below
    long byteSize;       // of this event, excl. type and byteSize
    long deltaFrames;    // sample frames related to the current block start sample position
    long flags;          // generic flags, none defined yet (0)
    char data[16];       // size may vary but is usually 16
};

enum                     // VstEvent types
{
    kVstMidiType = 1,    // midi event, can be cast as VstMidiEvent (see below)
    kVstAudioType,       // audio
    kVstVideoType,       // video
    kVstParameterType,   // parameter
    kVstTriggerType      // trigger
    // ...etc
};
```

There is an array of such events too:

```
struct VstEvents                // a block of events for the current audio block
{
    long numEvents;
    long reserved;              // zero
    VstEvent* events[2];        // variable
};
```

A VstEvent could be anything and be of any size, this allows for future extension and introduction of new event types. The current enumerated types are just examples; at the time of writing this only kVstMidiType was supported.

```
struct VstMidiEvent      // to be casted from a VstEvent
{
    long type;           // kVstMidiType
    long byteSize;       // 24
    long deltaFrames;    // sample frames related to the current block start sample position
    long flags;          // none defined yet
    long noteLength;     // (in sample frames) of entire note, if available, else 0
    long noteOffset;     // offset into note from note start if available, else 0
    char midiData[4];    // 1 thru 3 midi bytes; midiData[3] is reserved (zero)
    char detune;         // -64 to +63 cents; for scales other than 'well-tempered' ('microtuning'
    char noteOffVelocity;
    char reserved1;      // zero
    char reserved2;      // zero
};
```

A VstMidiEvent can safely be cast from a given VstEvent as follows.

```
void doSomeEvent (VstEvent* e)
{
    if (e->type == kVstMidiType)
    {
        VstMidiEvent* me = (VstMidiEvent*)e;
        doSomeMidi (me);
    }
}
```

Making Use of VST Events

So after making such a series of hints, let's look at the process of receiving and processing VST events in general terms.

First and foremost it should be pretty obvious that your plug-in class should be derived from the new AudioEffectX class and not from the older AudioEffect class. In your class you must override the method…

```
long processEvents(VstEvents* ev);
```

Then you must call wantEvents() in the resume() method of your plug-in. The method wantEvents() can not be called at construction time, as the host has no instanciated plug-in yet that it can relate such requests to. See the example later for details.

Handling Events

If you have met the above conditions, the host will call the processEvents() just before calling either process() or processReplacing(). The method processEvents() has a block of VstEvents as its parameter. Each of these events in the block has a delta-time, in samples, that refers to positions into the audio block about to be processed with process() or processReplacing(). It's your job to

gather up all the VstEvents given to you, then evaluate them during the next call to process() or processReplacing(). That's it. That's how it works. So now onto some details.

Handling Events – the Detailed View

Returning for a moment to the VstEvents block passed as a parameter to processEvents()

```
struct VstEvents           // a block of events for the current audio block
{
    long numEvents;
    long reserved;         // zero
    VstEvent* events[2];   // variable
};
```

This is really straightforward; The field numEvents specifies the number of pointers to VstEvents the events[ ] array holds. What is very important to understand is that this array of VstEvent* is only valid during this call to your processEvents() method, and the following process () or processReplacing() call. It's up to you to evaluate the events and maintain any status information you need between calls to processEvents().

The incoming events need to be examined. Maybe the following code fragment will help.

```
long myPlug::processEvents (VstEvents* ev)
{
    for (long i = 0; i < ev->numEvents; i++)
    {
        if (events[i]->type == kVstMidiType)
        {
            VstMidiEvent* me = (VstMidiEvent*)events[i];
            doDaMidi (me->data);  // may also want to examine noteLength etc
        }
    }
}
```

Details of VstMidiEvent

Now we should examine the fields of the VstMidiEvent type

The non- programming comments are in right justified blue italics

```
struct VstMidiEvent           // cast from a VstEvent
{
    long type;        // kVstMidiType
```
obviously, is always kVstMidiType (else it is not a midi event at all).
```
    long byteSize;    // 24
```

fixed to 24. this is supposed to be the 'default' byteSize for all 'regular' VstEvent's (with the option for future expansion).

```
    long deltaFrames;    // sample frames related to the current block start sample position
```
see below.

```
    long flags;          // none defined yet
```
so don't touch...and set to zero when sending.

```
    long noteLength;     // (in sample frames) of entire note, if available, else 0
```
on sequencer playback, the note length (in sample frames) may be present (but doesn't have to). a synth may use this for instance to automatically adjust envelope paramters (decay / release) to this parameter.

```
    long noteOffset;     // offset into note from note start if available, else 0
```
when the sequencer is started from within a note, it may (but doesn't have to) tell how far into this note it was located (in sample frames). a synth may use this for instance to shift envelope processing accordingly (i.e. set the envelopes as if the note had sounded as long as noteOffset).

```
    char midiData[4];    // 1 thru 3 midi bytes; midiData[3] is reserved (zero)
```
currently only midi *channel* data are supported, that is,
midiData[0] ranges from 0x80 to 0xef (note mtc and/or midi clock may also appear later)
midiData[1] and
midiData[2] hold the according midi bytes (if any) as defined in MIDI.
midiData[3] may hold further information but isn't supported yet (must set to 0 when sending).

```
    char detune;         // -64 to +63 cents; for scales other than 'well-tempered' ('microtuning'
```
you may want to implement per-note microtuning. detune tells the offset from the regular (well-tempered scale) frequency, in cents of a semitone.

```
    char noteOffVelocity;
```
if any.

```
    char reserved1;      // zero
    char reserved2;      // zero
};
```

Notes

Obviously, noteLength, noteOffset, detune, and noteOffVelocity, are only of concern if midiData[0] == 0x9x, that is, a midi Note On status. Also be aware, despite a given noteLength (if any), a Note Off (or Note On with velocity 0) will be sent when the note sounds off.

It is a matter of convention, not law, that the time order of the events in the array is ascending.

Delta Times

The field deltaFrames is the offset where the event starts, in sample frames, into the following audio block to be processed by process () or processReplacing (). In other words, if you had a block size of 1024 samples, and the first event in song that should be seen by this plug-in is as sample position 2000, and the song was started at sample position 0, then the first call to processEvents()

before process() was called, would contain no events in the array. The reason is that the first events sample start time doesn't fall into the scope of our first audio block of 1024 samples.

However in the second call to processEvents() the intended time of the event (sample position 2000) falls into the scope of the related process() call. That is, it's somewhere between 1024-2048 samples. The Delta time of the event is relative to the block within which it falls. Meaning our event that should fall at the absolute sample position of 2000 is sent with a delta time of 976 (2000-1024).

Therefore, assuming the event is a note and must be played for more than a few samples, the event is started in this second block at a relative sample position 976, and continues until the end of the block, before processing would continue in subsequent processEvent() / process() cycles.

This mechanism allows for sample-accurate playback of the notes.

Just as a side issue, the MIDI events resulting from sequencer playback are actually scheduled into the future, or pre-fetched, to accommodate for audio device latencies. This actually means you get events some time ahead of the actual playback time in pure MIDI terms. It's a fact, but one that you as a plug-in developer shouldn't need to bother about.

Where to go from here

This section has covered only the bare essentials for VstEvents, their Delta-times, and the relationship between them and the processEvents() and your process() calls. If you are interested in developing virtual synths then move on the next chapter that extends some of the concepts introduced here.

What's new in VST 2.0
A Soft-Synth Plug-In Example

There is an example 'synth' included in this SDK which is intended to show how VstEvents are handled from the plug-in's side. Note that the actual algorithm is really very bad. It is monophonic, it's not anti-aliasing, and not even efficient. You will have to solve these issues yourself, but you may use the example as a template when dealing with VstEvents.

This example assumes that you are entirely familiar with the VST 1.0 specification or have worked through this document. Hopefully you have found the time to read the ADelay example or better still the ADelayEdit.

The 'synth' has no interface so it uses the default 'Multi-Fx' type provided by the host. It has 2 'oscillators', each with waveform, frequency, and volume controls, each of which sounds on its own channel.

As with the other examples, we'll start looking at the code right away. The file vstxsynth.h should look familiar, it's just like in the ADelay example where we have several parameters. Also vstxsynth.cpp looks almost like ADelay.cpp, with some notable exceptions in that the constructor calls some interesting AudioEffectX methods:

```
VstXSynth::VstXSynth (audioMasterCallback audioMaster)
      : AudioEffectX (audioMaster, kNumPrograms, kNumParams)
{
    programs = new VstXSynthProgram[kNumPrograms];
    if (programs)
        setProgram (0);
    if (audioMaster)
    {
        setNumInputs (0);               // no inputs
        setNumOutputs (kNumOutputs);    // 2 outputs, 1 for each osciallator
        canProcessReplacing ();
        hasVu (false);
        hasClip (false);
        isSynth ();
        setUniqueID ('VxSy');           // <<<! *must* change this!!!!
    }
    initProcess ();
    suspend ();
}
```

Notice the call to setNumOutputs (). You may create an arbitrary number of output channels for a plug-in but resources may be limited on the host application side, and also too many channels will

confuse the user, so you should make a reasonable decision about the number of output channels to choose. Also, see below how to declare a pair of channels stereo; and please don't choose an odd number of channels, as the convention is that a stereo channel pair *always* starts on an even indexed (or odd numbered) channel.

The call to isSynth() is very important. This indicates to the host application that we want our own channels. In Cubase, this plug will appear in the Vst Instruments rack, and new channels will be created for each output (as far as available).

The call initProcess() is part of the algorithm which will not be discussed here. All of this can be found in vstxsynthproc.cpp, where also the very important resume() method can be found:

```
void VstXSynth::resume ()
{
    wantEvents ();
}
```

If you don't call this, the synth will not receive any 'MIDI' data from the host application. The reason why this will have no effect when called at construction time is that the constructor is called during the 'main' call, that is, the application has not yet received a pointer to the 'C' interface, in other words, the plug has no object yet (it doesn't yet 'exist'). A VST plug-in's resume() method is called when the plug is activated (switched 'on'), before the process() or processReplacing() method is called for the first time, or after having been de-activated (likewise, suspend() is called when the process is de-activated).

Except for isSynth() and wantEvents() there is nothing really new when compared to ADelay. Now we should receive some midi from the application, which comes in the form of VstEvents, which are discussed in detail in the previous section. Scanning VstEvents is done in the processEvents method that you must override:

```
long VstXSynth::processEvents (VstEvents* ev)
{
    for (long i = 0; i < ev->numEvents; i++)
    {
        if ((ev->events[i])->type != kVstMidiType)
            continue;
        VstMidiEvent* event = (VstMidiEvent*)ev->events[i];
        char* midiData = event->midiData;
        long status = midiData[0] & 0xf0;         // ignoring channel
        if (status == 0x90 || status == 0x80)   // we only look at notes
        {
            long note = midiData[1] & 0x7f;
            long velocity = midiData[2] & 0x7f;
            if (status == 0x80)
                velocity = 0;
            if (!velocity && (note == currentNote))
                noteIsOn = false;    // note off by velocity 0
            else
                noteOn (note, velocity, event->deltaFrames);
        }
        else if (status == 0xb0 && midiData[1] == 0x7e) // all notes off
            noteIsOn = false;
        event++;
    }
    return 1;   // want more
}
```

As with everything else that deals with the actual 'functionality' of this example, this is very raw, and you will most probably implement much more sophisticated midi data handling. The rest of this example is can be seen in the accompanying source file. Build it and have a look through, then go and do it properly.

But as described in the section on VST Events, the host will call your processEvents() method with the events that are valid for the next call to your process() or processReplacing() call. You can assume the data given to you in the processEvents() function is valid until the point your processing function returns. It is the host's job to handle this correctly.

The most important feature perhaps of VstEvents is the deltaFrames field. As this defines where precisely an event occurs, it's possible to render 'MIDI' events to audio with sample accuracy.

What's New in VST 2.0 – Reference Section
AudioEffectX Class – Getting to know the new core class

The following information is intended for those people who have already produced Plug-ins conforming to the VST 1.0 or are at least fully familiar with the VST 1.0 Specification. Diving in here as a short cut to learning about VST Plug-ins in general is not going to be very helpful.

Just as the VST 1.0 specification was built around a class `AudioEffect`, the VST 2.0 specification is built around a new core class `AudioEffectX`. We start with a complete discussion of the `AudioEffectX` class and its methods, which declares all new VST 2.0 features as seen from plug-in point of view.

The non- programming comments are in italics blue and right justified.

class AudioEffectX : public AudioEffect

so we are 100% compatible with version 1.0, as we derive from AudioEffect.

```
{
public:
    AudioEffectX (audioMasterCallback audioMaster, long numPrograms, long numParams);
```
same parameters as AudioEffect
```
    virtual ~AudioEffectX ();
    virtual long dispatcher (long opCode, long index, long value, void *ptr, float opt);
```
overriding the AudioEffect dispatcher. don't worry, nothing that should concern you.
```
    // 'host' are methods which go from plug to host, and are usually not overridden
    // 'plug' are methods which you may override to implement the according functionality (to ho
```
also note that a return value of 0 of either methods almost always indicates 'not implemented'.
```
//-------------------------------------------------------------------------------
// events + time
//-------------------------------------------------------------------------------
    // host
    virtual void wantEvents (long filter = 1);
    // filter is currently ignored, midi channel data only (default)
```
you can't issue this call at construction time,

do so in your resume() method instead if you want to receive VstEvents .
```
    virtual VstTimeInfo* getTimeInfo (long filter);
    // returns const VstTimeInfo* (or 0 if not supported)
    // filter should contain a mask indicating which fields are requested
    // (see valid masks in aeffectx.h), as some items may require extensive  conversions
```
see discussion of VstTimeInfo .
```
    virtual long tempoAt (long pos);
    // returns tempo (in bpm * 10000) at sample frame location <pos>
```
note that this call may cause heavy calculations on the application side.

Tempo is returned in Beats Per Minute * 10000.

Sample position and other time info can be obtained though VstTimeInfo.

```
    bool sendVstEventsToHost (VstEvents* events);    // true:success
```
see discussion of VstEvents, and Steinberg products implementation issues.

```
    // plug
    virtual long processEvents (VstEvents* events) {return 0;}
    // wants no more...else return 1!
    // VstEvents and VstMidiEvents are declared in aeffectx.h
```
override this to evaluate VstEvents that may be sent to your plug due to a wantsEvent() request.

```
//-------------------------------------------------------------------------
// parameters and programs
//-------------------------------------------------------------------------
    // host
    virtual long getNumAutomatableParameters ();
```
depending on the application's implementation, the number of automatable parameters may be limited.

a return value of 0 indicates 'don't know'.

```
    virtual long getParameterQuantization ();
    // returns the integer value for +1.0 representation,
    // or 1 if full single float precision is maintained
    // in automation. parameter index in <value> (-1: all, any)
```
when the application stores and restores automation data, or when a user control (like a fader or knob) is quantized to integer values, there is some quantization applied related to the floating point value. a return value of 0 indicates 'don't know' (not implemented).

```
    // plug
    virtual bool canParameterBeAutomated (long index) { return true; }
```
indicate if a parameter can be automated. obviously only useful when the application supports this.
```
    virtual bool string2parameter (long index, char* text) {return false;}
    // note: implies setParameter. text==0 is to be
    // expected to check the capability (returns true).
```
especially useful for 'multieffect' type of plug-ins (without user interface). The application can then implement a text edit field for the user to set a parameter by entering text.
```
    virtual float getChannelParameter (long channel, long index) {return 0;}
```
for internal use of soft synth voices, as a voice may be channel dependant.
```
    virtual long getNumCategories () {return 1L;}
```
if you support of more than 1 category (see aeffectx.h), override this.
```
    virtual bool getProgramNameIndexed (long category, long index, char* text) {return false;}
```
allows a host application to list your programs (presets).
```
    virtual bool copyProgram (long destination) {return false;}
```
copy currently selected program to program with index <destination>.
```
//-------------------------------------------------------------------------
// connections, configuration
//-------------------------------------------------------------------------
    // host
    virtual bool ioChanged ();        // tell host numInputs and/or numOutputs has changed
```
indicate a change of the number of inputs and/or outputs.

again, only useful if the hosting application supports this (returns true).
```
    virtual bool needIdle ();          // plug needs idle calls (outside its editor window)
```

some plugs need idle calls even when their editor is closed.

```
    virtual bool sizeWindow (long width, long height);
```

request to resize the editor window.

```
    virtual double updateSampleRate ();
    // gets and returns sample rate from host (may issue setSampleRate() )
    virtual long updateBlockSize ();
    // same for block size
```

will cause application to call AudioEffect's setSampleRate() and setBlockSize() methods to be called (when implemented).

```
    virtual long getInputLatency ();
    virtual long getOutputLatency ();
```

get the ASIO latency values. while inputLatency is probably not of concern, outputLatency may be used in conjunction with getTimeInfo() . samplePos of VstTimeInfo is ahead of the 'visual' sequencer play time by the output latency, such that when outputLatency samples have passed by, our processing result becomes audible.

```
    virtual AEffect* getPreviousPlug (long input);
    // input can be -1 in which case the first found is returned
    virtual AEffect* getNextPlug (long output);
    // output can be -1 in which case the first found is returned
```

for future expansion. the argument is an index to a pin, as several plugs may be connected on several pins.

```
    // plug
    virtual void inputConnected (long index, bool state) {}
    // input at <index> has been (dis-)connected,
    virtual void outputConnected (long index, bool state) {}
    // same as input; state == true: connected
```

application may issue these calls when implemented.

```
    virtual bool getInputProperties (long index, VstPinProperties* properties) {return false;}
    virtual bool getOutputProperties (long index, VstPinProperties* properties) {return false;}
```

these you should support (see 'general 2.0 additions').

all relevant structures like VstPinProperties can be found in aeffectx.h.

```
    virtual VstPlugCategory getPlugCategory()
      {
            if (cEffect.flags & effFlagsIsSynth)
                  return kPlugCategSynth;

            return kPlugCategUnknown;
      }
```

specify a category that fits your plug. see VstPlugCategory enumerator in aeffectx.h.

```
//-------------------------------------------------------------------------------
// realtime
//-------------------------------------------------------------------------------
    // host
    virtual long willProcessReplacing ();
    // returns 0: not implemented, 1: replacing, 2: accumulating
```

if host returns 0, we don't know whether process() or processreplacing() is called. remember all plugs should really support both methods, it's a matter of copy and paste but may significantly enhance audio performance. a return value of 1 indicates that host will call processReplacing(), otherwise process() will be used.

```
    virtual long getCurrentProcessLevel ();
```

```
                           // returns: 0: not supported,
                           // 1: currently in user thread (gui)
                           // 2: currently in audio thread or irq (where process is called)
                           // 3: currently in 'sequencer' thread or irq (midi, timer etc)
                           // 4: currently offline processing and thus in user thread
                           // other: not defined, but probably pre-empting user thread.
```

a plug is like black box processing some audio coming in on some inputs (if any) and going out of some

outputs (if any). this may be used to do offline or real-time processing, and sometimes it may be desirable

to know the current context.

```
    virtual long getAutomationState ();
              // returns 0: not supported, 1: off, 2:read, 3:write, 4:read/write   sic.
    virtual void wantAsyncOperation (bool state = true);
              // notify host that we want to operate asynchronously.
              // process() will return immedeately; host will poll getCurrentPosition
          // to see if data are available in time.
```

support for plug-ins whose processing is actually carried out on external resources (dsp). the idea is to

use asynchronous processing here; host will call the process() method as always, but the plug will only

pass input data to its dsp, and return immediately. now while the external processor does its work, the

application may call other (native) audio processes which are not depending on the output of this plug.

when host is done, it will poll the plug's getCurrentPosition() method to learn if the external dsp is done,

and then take data from the plugs output buffer (reportDestinationBuffer()), or the 'regular' output buffers

if none is provided. use canHostDo("asyncProcessing") to find out if this functionality is supported by the

application.

```
    virtual void hasExternalBuffer (bool state = true);
              // external dsp, may have their own output buffe (32 bit float)
              // host then requests this via effGetDestinationBuffer
```

for dma access it may be more efficient for the plug with external dsp to provide its own buffers.

otherwise, host expects output data in the output buffers passed to process() as usual.

```
    // plug
    virtual long reportCurrentPosition () {return 0;}
              // for external dsp, see wantAsyncOperation ()
    virtual float* reportDestinationBuffer () {return 0;}
              // for external dsp (dma option)
```

see wantAsyncOperation.

```
//-------------------------------------------------------------------------------
// offline
//-------------------------------------------------------------------------------
    // host
    virtual bool offlineRead (VstOfflineTask* offline, VstOfflineOption option, bool readSource =
true);
    virtual bool offlineWrite (VstOfflineTask* offline, VstOfflineOption option);
    virtual bool offlineStart (VstAudioFile* ptr, long numAudioFiles, long numNewAudioFiles);
    virtual long offlineGetCurrentPass ();
    virtual long offlineGetCurrentMetaPass ();
    // plug
    virtual bool offlineNotify (VstAudioFile* ptr, long numAudioFiles, bool start) { return false
    virtual bool offlinePrepare (VstOfflineTask* offline, long count) {return false;}
    virtual bool offlineRun (VstOfflineTask* offline, long count) {return false;}
    virtual long offlineGetNumPasses () {return 0;}
    virtual long offlineGetNumMetaPasses () {return 0;}
```

for a complete discussion of these, see Vst Offline.

```
//-----------------------------------------------------------------------
// other
//-----------------------------------------------------------------------
    // host
    virtual void setOutputSamplerate (float samplerate);
```

used for variableIo processing (see below).

```
    virtual bool getSpeakerArrangement (VstSpeakerArrangement* plug-inInput, VstSpeakerArrangement*
plug-inOutput);
```

see Vst Surround.

```
    virtual bool getHostVendorString (char* text);
       // fills <text> with a string identifying the vendor (max 64 char)

    virtual bool getHostProductString (char* text);
       // fills <text> with a string with product name (max 64 char)

    virtual long getHostVendorVersion ();
       // returns vendor-specific version

    virtual long hostVendorSpecific (long lArg1, long lArg2, void* ptrArg, float floatArg);
       // no definition

    virtual long canHostDo (char* text);
       // see 'hostCanDos' in audioeffectx.cpp
       // returns 0: don't know (default), 1: yes, -1: no

    virtual void isSynth (bool state = true);
       // will call wantEvents if true

    virtual void noTail (bool state = true);
       // true: tells host we produce no output when silence comes in

    virtual long getHostLanguage ();
       // returns VstHostLanguage

     ***** there's a function missing here charlie *****
       // enables host to omit process() when no data are present on any one input.

    virtual void* openWindow (VstWindow*);                    // create new window
    virtual bool closeWindow (VstWindow*);                    // close a newly created window
      // plug
    virtual bool processVariableIo (VstVariableIo* varIo) {return false;}
```

used for variableIo processing (see below).

```
    virtual bool setSpeakerArrangement (VstSpeakerArrangement* plug-inInput,
                      VstSpeakerArrangement* plug-inOutput) {return false;}
```

see Vst Surround.

```
    virtual void setBlockSizeAndSampleRate (long blockSize, float sampleRate)
        {this->blockSize = blockSize; this->sampleRate = sampleRate;}
```

more handy than the seperate methods setBlockSize() and setSampleRate()..

```
    virtual bool setBypass(bool onOff) {return false;}
    // for 'soft-bypass; process() still called
```

some plugs need to stay 'alive' even when bypassed; an example is a surround decoder which has more

inputs than outputs and must maintain some reasonable signal distribution even when being bypassed.

```
virtual bool getEffectName (char* name) {return false;}      // name max 32 char
virtual bool getErrorText (char* text) {return false;}       // max 256 char
virtual bool getVendorString (char* text) {return false;}
  // fill text with a string identifying the vendor (max 64 char)
virtual bool getProductString (char* text) {return false;}
  // fill text with a string identifying the product name (max 64 char)

virtual long getVendorVersion () {return 0;}
  // return vendor-specific version
virtual long vendorSpecific (long lArg, long lArg2, void* ptrArg, float floatArg) {return 0;}
  // no definition, vendor specific handling
```

see the documentation and examples of general features of version 2.0 that you should support.

```
virtual long canDo (char* text) {return 0;}
  // see 'plugCanDos' in audioeffectx.cpp. return values:
  // 0: don't know (default), 1: yes, -1: no

  virtual void* getIcon () {return 0;}          // not yet defined
```

could be a CBitmap (see Vst GUI).*

```
virtual bool setViewPosition (long x, long y) {return false;}
virtual long getGetTailSize () {return 0; }
```

this should return, in samples, the time it 'sounds out', that is, after what time the response to an input

sample will have dies to zero. for instance, a reverb will have a tail somewhat similar to its reverbaration

time; an IIR will 'never' die, and an FIR will usually die after a short amount of samples. many plugs will

have no tail at all (those should return 1, as a return value of 0 indicates 'not supported'. the application

may decide to not call the plug's process method after it sounded out, and no input data are applied

(silence). you should return 0 if continuous calls to your process methods are required.

```
virtual long fxIdle () {return 0;}
```

in return to a needIdle() request, the application may issue this call once in its idle loop.

```
virtual bool getParameterProperties (long index, VstParameterProperties* p) {return false;}


};
```

What's New in VST 2.0 – Reference Section
VST Time Info - Letting the plug-in know what time it is.

Under the VST 2.0 specification a plug may request detailed time information at runtime. If the hosting application supports this call, it will return a pointer to a VstTimeInfo structure. Note that all time information at the time of the call is related to the current time slice.

From the plug's point of view, a new time slice starts when either processEvents(), or process() or processreplacing() is called. Let's start by looking at the basic VstTimeInfo structure.

```
struct VstTimeInfo
{
    double  samplePos;            // current location
    double  sampleRate;
    double  nanoSeconds;          // system time
    double  ppqPos;               // 1 ppq
    double  tempo;                // in bpm
    double  barStartPos;          // last bar start, in 1 ppq
    double  cycleStartPos;        // 1 ppq
    double  cycleEndPos;          // 1 ppq
    long  timeSigNumerator;       // time signature
    long  timeSigDenominator;
    long  smpteOffset;
    long  smpteFrameRate;         // 0:24, 1:25, 2:29.97, 3:30, 4:29.97 df, 5:30 df
    long  samplesToNextClock;     // midi clock resolution (24 ppq), can be negative
    long  flags;                  // see below
};

enum
{
    kVstTransportChanged       = 1,
    kVstTransportPlaying       = 1 << 1,
    kVstTransportCycleActive   = 1 << 2,
    kVstAutomationWriting      = 1 << 6,
    kVstAutomationReading      = 1 << 7,
    // flags which indicate which of the fields in this VstTimeInfo
    //  are valid; samplePos and sampleRate are always valid
    kVstNanosValid             = 1 << 8,
    kVstPpqPosValid            = 1 << 9,
    kVstTempoValid             = 1 << 10,
    kVstBarsValid              = 1 << 11,
    kVstCyclePosValid          = 1 << 12,  // start and end
    kVstTimeSigValid           = 1 << 13,
    kVstSmpteValid             = 1 << 14,
    kVstClockValid             = 1 << 15
};
```

A Plug will request time info by calling the function getTimeInfo(long mask) which returns a VstTimeInfo pointer (or NULL if not implemented by the host). The mask parameter is composed of the same flags which will be found in the <flags> field of VstTimeInfo when returned, that is, if you need information about tempo, the parameter passed to getTimeInfo() should have the kVstTempoValid flag set.

This request and delivery system is important, as a request like this may cause significant calculations at the application's end, which may take a lot of our precious time. This obviously means you should only set those flags that are required to get the information you need. Also please be aware that requesting information does not necessarily mean that that information is provided in return. Check the flag field in the VstTimeInfo structure to see if your request was actually met.

So looking at the VstTimeInfo structure more closely:

struct VstTimeInfo

```
{
    double samplePos;          // current location
    double sampleRate;
```
these must always be valid, and should not cost a lot to ask for.
Note that samplePos may jump backwards in cycle mode.
```
    double nanoSeconds;        // system time
```
System Time related to samplePos (which is related to the first sample in the buffers passed to the process() methods). system time is derived from timeGetTime() on WINDOWS platforms, Microseconds() on MAC platform, UST for MOTIF, and BTimeSource::RealTime () for BEOS.
```
    double ppqPos;             // 1 ppq
```
the quarter position related to samplePos. as this is a floating point value, it can be easily converted to any scale. provided that the tempo is also given, any linear positions (like samples, smpte) can also be calculated precisely.
```
    double tempo;              // in bpm
```
tempo in Beats Per Minute (no scaling).
```
    double barStartPos;        // last bar start, in 1 ppq
```
useful in conjunction with Time Signature, or to calculate a precise sample position of a beat or other measure.
```
    double cycleStartPos;      // 1 ppq
    double cycleEndPos;        // 1 ppq
```
locator positions in quarter notes. note the kVstTransportCycleActive flag.
```
    long timeSigNumerator;     // time signature
    long timeSigDenominator;
```
Time Signature; 1/4 has timeSigNumerator == 1 and timeSigDenominator == 4.
```
    long smpteOffset;
```
in SMPTE subframes (bits; 1/80 of a frame). the current smpte position can be calculated using samplePos, sampleRate, and smpteFrameRate.

```
long smpteFrameRate;        // 0:24, 1:25, 2:29.97, 3:30, 4:29.97 df, 5:30 df
```
SMPTE format.
```
long samplesToNextClock;    // midi clock resolution (24 ppq), can be negative
```
the distance to the next midi clock (24 ppq, pulses per quarter) in samples. unless samplePos falls precicely on a midi clock, this will either be negative such that the previous midi clock is addressed, or positive when referencing the following (future) midi clock.
```
long flags;
```
discussion of flags:

```
kVstTransportChanged
```
something has changed. this is not restricted to start/sop, or location changes but may also be set (once!) to indicate other changes like tempo, cycle state or positions etc.
```
kVstTransportPlaying
```
Sequencer is started (running).
```
kVstTransportCycleActive
```
Cycle mode is active.
```
kVstAutomationWriting
```
Automation Write mode is activated.
```
kVstAutomationReading
```
Automation Write mode is activated.
```
kVstNanosValid
```
the nanoSeconds field in this VstTimeInfo is valid (or requested from getTimeInfo ( )).
```
kVstPpqPosValid
```
the ppqPos field in this VstTimeInfo is valid (or requested from getTimeInfo ()).
```
kVstTempoValid
```
the tempo field in this VstTimeInfo is valid (or requested from getTimeInfo ()).
```
kVstBarsValid
```
the barStartPos field in this VstTimeInfo is valid (or requested from getTimeInfo ()).
```
kVstCyclePosValid
```
the cycleStartPos and cycleEndPos fields in this VstTimeInfo are valid (or requested from getTimeInfo ()).

```
kVstTimeSigValid
```
the timeSigNumerator and timeSigDenominator fields in this VstTimeInfo are valid (or requested from getTimeInfo ()).
```
kVstSmpteValid
```
the smpteOffset and smpteFrameRate fields in this VstTimeInfo are valid (or requested from getTimeInfo ()).
```
kVstClockValid
```
the samplesToNextClock field in this VstTimeInfo is valid (or requested from getTimeInfo ()).

```
};
```

What's New in VST 2.0 – Reference Section
VST Offline Processing

Introduction

The VST offline interface is a powerful API that allows a plug-in to freely read audio files open in the host, to transform them or to generate new audio files. The main features are:

* The plug-in controls reading/writing of audio samples by sending commands to the host
(this approach is the reverse of the mechanism used in the real-time interface).

* A plug-in can read, simultaneously and with random-access, any number of files open in the host.

* A plug-in can overwrite part or all of any open file in the host, with random access.

* A plug-in can create one or more files (temporary ones or not).

* Any file created by the plug-in can be freely re-read and overwritten, with random access, during the process.

* Thanks to "delayed overwriting", original samples in source files are preserved in all cases and can be read again, at will and at any time during a process.

* Not only audio can be read/written in files: markers can be created; sample selection can be set; edit cursor can be moved. Moreover, custom parameters can be written along a file, with a time stamp, and read again later, by the same plug-in or another one.

* No stereo assumption: could be used for multi-channel files, if the host supports it.

* An offline plug-in can be used as a file transformer, as a file analyzer or as a file generator.

* An offline plug-in does not deal directly with file i/o and file formats: it just sends read and write commands with float buffers, resulting in great simplification.

* An offline-capable plug-in is notified by the host anytime a change occurs in the set of files available for editing (new open file, transformed file, new marker created in a file, etc.). This allows the plug-in, if necessary, to update its user interface according to the context (e.g. new file with the focus; or to show a list of files to pick from; etc.).

Declaring an offline plug-in

The host knows if a given plug-in supports the offline interface through the canDo function, as follows:

   If canDo ("offline") is true, the plug-in supports the offline interface

   If canDo ("noRealTime") is true, the plug-in only supports the offline interface

Overview of the interface

Three structures are dedicated to the offline interface:

```
VstOfflineTask
VstAudioFile
VstAudioFileMarker
```

There are also three enums:

```
VstOfflineTaskFlags
VstAudioFileFlags
VstOfflineOption
```

Three host opcodes are defined:

```
audioMasterOfflineStart
audioMasterOfflineRead
audioMasterOfflineWrite
```

with the corresponding functions in AudioEffectX:

```
bool offlineStart(VstAudioFile* ptr, long numAudioFiles, long numNewAudioFiles);
bool offlineRead(VstOfflineTask* offline, VstOfflineOption option, bool readSource = true);
bool offlineWrite(VstOfflineTask* offline, VstOfflineOption option);
```

Three plug-in opcodes are defined:

```
effOfflineNotify,
effOfflinePrepare,
effOfflineRun,
```

with the corresponding functions in AudioEffectX:

```
void offlineNotify(VstAudioFile* ptr, long numAudioFiles, bool start);
bool offlinePrepare(VstOfflineTask* offline, long count);
bool offlineRun(VstOfflineTask* offline, long count);
```

An offline process results from a nested sequence of calls, as follows:

1) The host calls offlineNotify, passing an array of VstAudioFile structures that describe all files that can be read and written. There is also a "start" argument that indicates to the plug-in whether

the process should begin or not. The "start" argument is true e.g. after the user presses the "Process" button (which should be under the host control). The "start" argument is false if the call results from any change in the file environment of the host.

2) In its implementation of offlineNotify, the plug-in states which file(s) it wants to read and write, by setting flags in the VstAudioFile structures. Then the plug-in calls the function offlineStart. The last argument of offlineStart allows the plug-in to create one or more new files.

3) In its implementation of offlineStart, the host initializes an array of VstOfflineTask structures, one for each file to read or/and write. Then the host calls: offlinePrepare.

4) In its implementation of offlinePrepare, the plug-in continues the initialization of the VstOfflineTask structures (eg. set the sample rate and number of channels in new files).

5) If offlinePrepare returns true, the host finalizes the preparation of the VstOfflineTask structures (eg. allocate i/o audio buffers), then calls offlineRun.

6) In its implementation of offlineRun, the plug-in can call the host functions offlineRead and offlineWrite at will, until its audio processing is completed. The approach is therefore opposite to the realtime interface: the plug-in here instructs the host about which files to read and write, in which order, at which rate.

A small diagram can illustrate the nested sequence of calls. Functions in red are called by the host and implemented by the plug-in. Functions in blue are called by the plug-in and implemented by the host.

```
offlineNotify
{
     offlineStart
    {
        offlinePrepare
        offlineRun
        {
            ...
            offlineRead
            offlineWrite
            ...
        }
    }
}
```

Details of the interface

`struct VstAudioFile`

This structure describes an audio file already open in the host. This description is system-independent: no file path is specified. A plug-in does not query the host about available files; instead, it gets informed about the available files when the host calls the function offlineNotify. Note: there is an option, however, to force the host to send a notification (see kVstOfflineQueryFiles).

The host sets all the members of this structure, unless notified otherwise.

`long flags`

See enum VstAudioFileFlags. Both host and plug-in can set flags.

`void* hostOwned`

Any data private to host.

`void* plugOwned`

Any data private to plug-in. This value is (optionally) set by the plug-in in its implementation of offlineNotify. This value is then copied by the host into the VstOfflineTask structure (plugOwned member), when offlineStart is called (this allows the plug-in, if necessary, to make a link between the offlineNotify call and the offlinePrepare/offlineRun calls).

`char name[100]`

Name of the file (no path, just name without any file extension). This can be used by the plug-in to display in its user interface the list of all files available for processing. This is useful if the plug-in requires the user to select more than one file to perform its job (eg. mixer, file comparer, etc...).

`long uniqueId`

This value identifies a file instance in the host, during a session. Not two file instances must ever get the same ID during a session. If a file gets closed and is reopen later, it must be attributed an ID that was never attributed so far during the session. From the host side, this can be easily implemented with a simple global counter.
This ID can be a useful reference for a plug-in, if its user interface maintains a list of files that the user can select.

`double sampleRate`

sample rate of the file

`long numChannels`

number of channels: 1 for mono, 2 for stereo, etc...

`double numFrames`

number of frames in the audio file

`long format`
reserved for future. Currently 0.

`double editCursorPosition`
frame index of the edit cursor, or -1 if no such cursor exists. This member represents the "edit-cursor" position, if any, but never the "playback-cursor" position.

`double selectionStart`
frame index of first selected frame, or -1 if there is no selection

`double selectionSize`
number of frames in selection. Zero if no selection.

`long selectedChannelsMask`
Bit mask describing which channels are selected. One bit per channel. Eg. value 3 means that both channels of a stereo file are selected.

`long numMarkers`
number of markers in the file. The plug-in can use offlineRead to get details about the markers.

`long timeRulerUnit`
If the plug-in needs to display time values, it might be nice to match the unit system selected by the user for the file in the host. This is the reason why this member, and the following ones, are provided. Possible values:
0: undefined
1: samples units
2: hours/minutes/seconds/milliseconds
3: smpte
4: measures and beats

`double timeRulerOffset`
frame offset of the time ruler for the window that displays the audio file. Usually 0 but could be negative or positive.

`double tempo`
-1 if not used by the file's time ruler, else BPM units.

`long timeSigNumerator`
-1 if not used by the file's time ruler, else number of beats per measure

`long timeSigDenominator`
-1 if not used by the file's time ruler, else number of beats per whole note

`long ticksPerBlackNote`
-1 if not used by the file's time ruler, else sequencer resolution

`long smpteFrameRate`
-1 if not used by the file's time ruler, else refers to VstTimeInfo for the meaning.

`enum VstAudioFileFlags`
This refers to the possible flags for the member flag of the structure VstAudioFile:
The host sets its flags before calling offlineNotify.
The plug-in sets its flags in its implementation of offlineNotify, before calling offlineStart.

`kVstOfflineReadOnly`
Set by host. Means that the file can't be modified, it can only be read. If the plug-in tries to write it later, the host should return false from offlineWrite.

`kVstOfflineNoRateConversion`
Set by host. Means that the file can't have its sample rate modified.

`kVstOfflineNoChannelChange`
Set by host. Means that the number of channels can't be changed (eg. the host might not allow an in-place mono to stereo conversion).

`kVstOfflineCanProcessSelection`
Set by the plug-in if its process can be applied to a limited part of a file. If no selection exists, the entire file range is used. The host checks this flag and, accordingly, initializes the members positionToProcessFrom and numFramesToProcess in the structure VstOfflineTask.
Setting this flag is a common case, but a counter example is e.g. a sample rate converter (the sample rate is global to a file and can't be applied to a limited part of a file).

`kVstOfflineNoCrossfade`
Consider the case of a plug-in transforming only a part of a file. To avoid a click at the edges (between the processed part and the non-processed part) the host might perform a short crossfade with the old samples, according to user preferences. However, a plug-in might want to reject this option for some reasons (eg. the plug-in performs a local glitch restoration and wants to perform the crossfade itself). In that case, the plug-in should set this flag to instruct the host not to perform any crossfade.

`kVstOfflineWantRead`
If the plug-in wants to read the file, it should set this flag. E.g. a signal-generator plug-in would never set that flag. If this flag is not set and the plug-in tries to read the file later, the host should return false from offlineRead.

`kVstOfflineWantWrite`

If the plug-in wants to overwrite part or the entire file, it should set this flag. E.g. an analyzer plug-in would never set that flag.

Note: as an alternative, the plug-in can choose to create a new file, rather than overwriting the source file (see offlineStart).

If this flag is not set and the plug-in tries to write the file later, the host should return false from offlineWrite.

`kVstOfflineWantWriteMarker`

If the plug-in wants to modify or create markers in the file, it should set this flag. If this flag is not set and the plug-in tries to move or create a marker later, the host should return false from offlineWrite.

`kVstOfflineWantMoveCursor`

If the plug-in wants to move the edit-cursor of the file, it should set this flag. If this flag is not set and the plug-in tries to move the edit-cursor later, the host should return false from offlineWrite.

`kVstOfflineWantSelect`

If the plug-in wants to select samples in the file, it should set this flag. If this flag is not set and the plug-in tries to select samples later, the host should return false from offlineWrite.

`struct VstOfflineTask`

This structure is used in offlinePrepare, offlineRun, offlineRead and offlineWrite functions. Its main purpose is to be a parameter-holder to instruct the host to read/write an existing file, or to write a new file. However, it can also be used as a parameter-holder for other purposes, as we shall see later (see VstOfflineOption). Thus, certain members of this structure have a different meaning according to the option selected when calling offlineRead and offlineWrite. For the sake of simplicity, we now mainly cover the common case of reading/writing audio samples.

An important principle to understand from the beginning, is that each file which is read or/and written is associated with a single VstOfflineTask structure.

`char processName[96]`

Set by plug-in in offlinePrepare. The host to label the process can use this name. E.g. the host might display that name in a menu entry called "Undo ....".

If the process uses multiple VstOfflineTask structures, only the first one needs to have this field set (or all other VstOfflineTask structures should have the same label).

This field might be erased later during the process, therefore the host should make a copy of it.

`double readPosition`

Position, as frames, of the read "head" in the audio file. Set both by plug-in and host:

This value should be set by the plug-in, before calling offlineRead, to instruct the host.

On the other hand, the host updates the value to reflect the read position after the call to offlineRead.

`double writePosition`

Position, as frames, of the write "head" in the audio file. Set both by plug-in and host:

This value should be set by the plug-in, before calling offlineWrite, to instruct the host.

On the other hand, the host updates the value to reflect the write position after the call to offlineWrite.

`long readCount`

Number of audio frames to read.

Set by plug-in, before calling offlineRead.

When returning from offlineRead, readCount contains the number of actually read frames. If the plug-in tries to read beyond the end of the file (not considered as an error), the float buffers are completed with blank frames by the host. In that case, the number of blank frames is returned in the member value. In other words, the sum (readCount + value) after the call is equal to the value of readCount before the call.

`long writeCount`

Number of audio frames to write.

Set by plug-in, before calling offlineWrite.

Never set by the host. If the host can't write the samples because of a disk-full situation, the host should return false from offlineWrite.

`long sizeInputBuffer`

Size, as frames, of the audio input buffer. Set by host before calling offlineRun. This value remains unchanged during the whole process. A plug-in can't read more than this number of samples in a single call to offlineRead.

`long sizeOutputBuffer`

Size, as frames, of the audio output buffer. Set by host before calling offlineRun. This value remains unchanged during the whole process. A plug-in can't write more than this number of samples in a single call to offlineWrite.

`void* inputBuffer`
`void* outputBuffer`

Both set by host, before calling offlineRun. The actual type of the pointer depends on the channel mode: if the plug-in has set the flag kVstOfflineInterleavedAudio, then the type is float* (array of interleaved samples). In the other case, the type is float** (array of array of samples). The latter is the standard case.

`double positionToProcessFrom`
`double numFramesToProcess`

Set by host, according to the flags set in enum VstAudioFileFlags. This defines the frame range that the plug-in should read for its process.

If required for its algorithm, the plug-in is allowed to read before and after this range (if the range is a subset of the file), but only that range of samples should be transformed.

`double maxFramesToWrite`

Set by plug-in in offlinePrepare. This value could be used by the host for optimization purposes (to select a proper write algorithm), and also to check if the disk space is sufficient before starting the process.

If the plug-in writes no audio, this value should be 0.

If the number of written samples is the same as the number of read samples, this value should be equal to numFramesToProcess.

If the plug-ins does not know exactly the number of frames, this value should be an approximate value, large enough for sure, but as small as possible (if the plug-in later tries to write more frames than this number, an error would be issued by the host).

If the plug-ins does not know at all, this value should be -1 (this is the default value of this member).

`void* extraBuffer`

This is set by the plug-in. This is a buffer which is used to read/write other data than audio. Meaning depends on the offlineRead/offlineWrite option (see VstOfflineOption).

`long value`

Set by plug-in or host. Meaning depends on the offlineRead/offlineWrite option (see VstOfflineOption).

`long index`

Set by plug-in or host. Meaning depends on the offlineRead/offlineWrite option (see VstOfflineOption).

This value is also optionally set by the plug-in during offlinePrepare, as follows:

If the plug-in generates a new file out of an existing file, then it should initialize this value with the index of the VstOfflineTask structure corresponding to the source file. This is not mandatory, but this info could be of interest for the host. Be default, index is -1 when offlinePrepare is called.

`double numFramesInSourceFile`

Number of frames in source file. This is set by the host in offlineStart. This value is only set for existing source files.

If the VstOfflineTask structure refers to a file created by the host on behalf of the plug-in, this value is 0.

`double sourceSampleRate`

Sample rate of the source file. Set by host.

If the VstOfflineTask structure refers to a file created by the host on behalf of the plug-in, this value is 0.

In that case, the plug-in must initialize this value when offlinePrepare is called (in that case, same value as destinationSampleRate).

`double destinationSampleRate`

Sample rate of the destination file. Set by plug-in in offlinePrepare (but previously initialized by host as sourceSampleRate).

If the VstOfflineTask structure refers to a file created by the host on behalf of the plug-in, this value is 0.

In that case, the plug-in must initialize this value when offlinePrepare is called (in that case, same value as sourceSampleRate).

`long numSourceChannels`

Number of channels in the source file. Set by host.

Note: if the mode kVstOfflineCanProcessSelection is active, and if only one channel of a stereo file is selected, then numSourceChannels should be set to 1. In that case, the file appears as a mono file from the plug-in point of view.

If the VstOfflineTask structure refers to a file created by the host on behalf of the plug-in, this value is 0. In that case, the plug-in must initialize this value when offlinePrepare is called (in that case, same value as numDestinationChannels).

`long numDestinationChannels`

Number of channels in the destination file. Set by plug-in in offlinePrepare (but previously initialized by host as numSourceChannels). This value is required for the host to allocate the proper outputBuffer.

If the VstOfflineTask structure refers to a file created by the host on behalf of the plug-in, this value is 0. In that case, the plug-in must initialize this value when offlinePrepare is called (in that case, same value as numSourceChannels).

`long sourceFormat`

Reserved for future. Set by host.

`long destinationFormat`

Reserved for future. Set by plug-in.

`char outputText [512]`

There are three uses for this member:

If the plug-in has instructed the host to create a new file (see offlineStart), then the plug-in can optionally provide its name in this member, with a fully qualified path (this file name must be selected by the user from the plug-in user interface). In that case, the file is saved by the host in the default audio file format for the platform (this could also be a host specific option). This name has to be initialized when offlinePrepare is called.

Note: the host, if a demo version, might reject this option!

If outputText is empty (common case), then the host creates the file in a folder dedicated to temporary files. Later, it's up to the user to save the file from the host.

Before returning from a function with false, the plug-in can set the flag kVstOfflinePlugError and then (over)write the outputText member. In that case, the host should display the message to the user.

If the host sets the flag kVstOfflineUnvalidParameter, then the host might as well fill up the outputText member, to give a hint to the plug-in, for pure debugging purposes.

`double progress`

Set by plug-in to inform the host about the current progress of the whole operation. The value must be in the range 0 to 1. If the value is not in this range (e.g. -1), the host must ignore it.

The plug-in should, if possible, update this value each time before calling offlineRead and offlineWrite (this would give the host enough occasions to update a progress indicator to give feedback to the user).

If the process has to perform several "passes", the progress value is allowed to go from 0 to 1 several times. However, ideally, a single 0 to 1 pass is better for the user's feedback.

The progress value is meant to be global: if there are several VstOfflineTask involved, the progress value should be "independent" from each task (yet, this global progress should be set in each VstOfflineTask structure passed to VstOfflineTask and offlineWrite calls).

`long progressMode`
Reserved for the future.

`char progressText[100]`
Set by plug-in, to describe what's going on. Can be updated any time. Optional.

`long flags`
Set by host and plug-in. See enum VstOfflineTaskFlags.

`long returnValue`
Reserved for the future.

`void* hostOwned`
any data private to host

`void* plugOwned`
Any data private to plug-in. This value is firstly initialized by the host, in offlineStart, with the value of the member plugOwned from the structure VstAudioFile (if the VstOfflineTask corresponds to an existing file).

enum VstOfflineTaskFlags

`kVstOfflineUnvalidParameter`
Sets by host if the plug-in passes an unvalid parameter. In that case, the host might fill up the member outputText, to give a hint to the plug-in, for debugging purposes.

`kVstOfflineNewFile`
Set by the host to indicate that this VstOfflineTask represents a task that creates/reads/writes a new file.

`kVstOfflinePlugError`
If an error happens in the plug-in itself (not an error notified by the host), then the plug-in could optionally set this flag before returning false from its function to indicate to the host that the member outputText (now) contains a description of the error. The host is then in charge of displaying the message to the user. The plug-in should never try to display itself an error message from the offlineRun function, since offlineRun could happen in a background task.

kVstOfflineInterleavedAudio

The plug-in should set this flag if it wants to get data in interleaved format. By default, this is not the case.

kVstOfflineTempOutputFile

The plug-in should set this flag in offlinePrepare, if the file to create must be a temporary one. In that case, the file is deleted at the end of the process (else, the file is usually open by the host at the end of the process).

This flag can obviously be set only for a new file, not for an existing file.

kVstOfflineFloatOutputFile

If the plug-in needs creating a file made of float samples, this flag should be set. Else, the default file format is dependant on the host (could be 16 bit, 24 bit, float...). This can be useful if the plug-in needs to store temporary results to disk, without fear of clipping.

kVstOfflineRandomWrite

If the plug-in needs to write randomly (not sequentially) a file, it should set this flag. This flag should also be set if the file is to be written sequentially more than once. This is a hint for the host to select a proper writing procedure. If this flag is not set, the host could return false from offlineWrite, if the plug-in attempts a non-sequential writing.

kVstOfflineStretch

If the plug-in time-stretches part or all of a file (eg. resampling), it should set this flag. This instructs the host to move and stretch the relative file markers, if any, to match the change. This also is of great importance in mode "process-selection" (see kVstOfflineCanProcessSelection), as it instructs the host to replace only the selection, whatever the number of written samples. Let's take an example: if there are 10000 selected input samples (from 0 to 9999) and 20000 samples are output by the plug-in, then:

1) if the flag kVstOfflineStretch is set: the host simply replaces the samples 0-9999 with the new 20000 samples, and also moves/stretches the file markers as required. Note that this requires the host to "push" the samples above position 20000 (to insert the 10000 new samples).

2) if the flag kVstOfflineStretch is not set: the host replaces the samples 0-19999 with the new 20000 samples (eg. echo plug-in that reads samples beyond the end of the selection, to merge the tail of the echo).

kVstOfflineNoThread

The host might either create a background thread to run the process, or run it inside the main application thread. The plug-in does not decide about this. However, it can happen that a process is so short that creating a thread would be a waste of time. In that case, the plug-in can set this flag as a hint to the host.

struct VstAudioFileMarker

`double position`
Position of the marker

`char name[32]`
Name of the marker

`long type`
The host might not support all types. We currently define:
0: undefined
1: generic marker
2: temporary marker (not saved with the file)
3: loop start marker
4: loop end marker
5: section start (whatever "section" might mean for the host)
6: section end

`long id`
This value is set by the host to identify a marker in a file. It can be any value but 0, which is reserved to indicate a new marker (see option kVstOfflineMarker). Not two markers can ever get the same ID for a given file.

enum VstOfflineOption

The functions offlineRead and offlineWrite have an argument (VstOfflineOption) that allows to read/write different types of data. Let's see what these options are:

`kVstOfflineAudio`
Use this option to read/write audio samples. See also description of VstOfflineTask.
Reading can happen randomly. This means that any time during the process, a plug-in is allowed to jump at any frame position and read from that point.
Random reading can occur either in a read-only file or in a file currently being written.
If a plug-in tries to read beyond the end of the file (not to be considered as an error by the host), the buffers are completed with blank samples by the host. See comments about readCount on that subject.
Writing can happen randomly. This means that a file can be (over)written any number of times and in any order. See kVstOfflineRandomWrite.
If writing is to happen at a position beyond the end of the file, the host must extend the file as required and fill the gap with zeroes.
Delayed overwriting. When a plug-in tries to overwrite part of an existing source file, the host should in fact write the samples in a separate file. When the process is finished, it's up to the host to actually replace the source samples. This feature is required to let to the plug-in have the possibility to read the original samples at any time during a process.

One important consequence of the above feature is that any writing, whatever the situation, always occur in a new - possibly temporary - file. This is why all write positions that a plug-in ever specifies, should always relate to the origin Zero.

E.g. if a plug-in wants to overwrite samples [10000-19999], it should specify write position in the range [0-9999]. It's up to the host to do the rest, later, to achieve the desired effect.

A plug-in can never "overwrite" before the position given by the member positionToProcessFrom; it is only possible to overwrite a continuous block starting from positionToProcessFrom. If the plug-in starts overwriting after positionToProcessFrom, the gap if filled up with blank samples.

To ease the undo/redo handling of the host (usually based on audio blocks), there is a rule to obey when "overwriting" a source file:

Only one continuous segment of a source file can be overwritten during a single task. E.g. it is not allowed to overwrite samples 0 to 10000 then samples 50000 to 60000, hoping that intermediary samples are preserved. If a plug-in does so, the result is undefined. However, if a plug-in really needs to do so, it must simply transfer itself the intermediary samples.

`kVstOfflinePeaks`

The plug-in UI might need to display part or all of a file. Reading a whole file (for display purposes) is a time consuming operation; this is why most hosts maintain a "peak file" that stores a "summary" of the audio file. With the kVstOfflinePeaks option, a plug-in can benefit from this host functionality. This option is only to be used with offlineRead, not offlineWrite. The required parameters of VstOfflineTask are the following ones:

positionToProcessFrom: set by plug-in. The frame index to display from.

numFramesToProcess: set by plug-in. The number of frames to display.

writeCount: set by host. This represents how many elements of pixel information have been stored by the host in the buffer.

value: set by plug-in. This is the zoom factor: it represents the desired number of frames to display per screen pixel.

index: set by host, see further.

inputBuffer: set by host. The elements of the array are not 32 bit float values, but pairs of 16 bit integers. An element of the array could be represented as follows:

struct { int16 y1; int16 y2; }

There are two ways to interpret the data written by the host into the buffer:

If the member index is set to 0 by the host, then the sound view is much "compressed". In that case, a peak at a given position is represented by a vertical line below and above the horizontal axis of the display. The value y1 represents the positive coordinate, above the axis, and the y2 coordinate, the negative value below the axis.

y1 is always in the range 0 to 32767. It has to be scaled by the plug-in, according to its display's height.

y2 is always in the range -32767 to 0. It has to be scaled by the plug-in, according to its display's height.

If the member index is set to 1 by the host, then the sound view is less "compressed" and should be displayed as a continuous curve.

In this case, y1 is always in the range -32767 to 32767. It has to be scaled by the plug-in, according to its display's height.

y2 is always 0 and should be ignored.

It is only possible to call this function on a source file, not on a file being written during a process.

If the host does not implement this function, offlineRead returns false. The plug-in could then read itself the audio (kVstOfflineAudio option) to display the wave.

kVstOfflineParameter

If the host supports this option, the plug-in can read and write parameters along the audio-file. A "parameter" is a float value or byte array. Each parameter is attributed an index (e.g. if there are 3 float values and 2 arrays, the indexes go from 0 to 4).

Examples of use: parameter automation; storage of meta-information (e.g. pitch) usable by the same plug-in, later during the same process, or in another process by another plug-in, etc.

The host is free to implement the underlying parameter storage, as it likes. However, it is easy to understand that parameters should be stored in a sorted vector, each one attached to a frame index.

The parameters are usually maintained in RAM by the host, therefore the plug-in should not over-use this feature and for instance write one parameter per sample!

The host might choose to save the parameters into a project-file, or to embed them into the audio file header, or not to save them at all (ie. all parameters get erased when the audio file closes).

Writing parameters with offlineWrite:

processName: name of the parameter family.

If a plug-in X writes parameters to a file, then a plug-in Y can retrieve the parameters only if it provides the right family name.

The name must be made unique enough, to prevent any clash.

This member only needs to be set for the first parameter to record during the process.

If this first parameter belongs to a new family, the host destroys all previously stored parameters.

value: version of the parameter family. Freely under the control of the plug-in. This member only needs to be set for the first parameter to record during the process.

index: index of parameter to write.

writeCount: 0 if writing a float parameter, else byte size of the parameter array.

extraBuffer: buffer allocated by the plug-in to pass the parameter.

For writing a float, this pointer is actually a float* pointer, else this is a pointer to the array.

If this pointer is NULL, then the parameter at that position, if any, is deleted.

If this pointer is NULL and the writePosition is negative, all parameters are erased.

writePosition: position where to write the parameter.

Since parameters are not stored "inside" the audio samples, it does not matter if the write position is temporarily beyond the end of the audio file.

For the sake of simplicity (when reading back later), it is not possible to write more than one parameter at a given position. If this happens, the old parameter gets erased.

If this parameter is negative and extraBuffer is NULL, all parameters get erased.

Reading parameters with offlineRead:

At the beginning, the plug-in is usually ignorant about what parameters are stored, and where. The first call to offlineRead is therefore a special one, as follows:

The plug-in initializes the member extraBuffer to 0 and the member readPosition to the position it wants to get informed about (usually, 0). When returning from offlineRead, the host has initialized the following parameters:

processName: name of the parameter family, or nothing if no recorded parameter. If the name of this parameter family is not supported by the plug-in, the plug-in should not try to read the parameters.

value: version of the recorded parameter family. Might be useful for the plug-in.

readPosition: the frame index at which the next parameter is found (this value was unchanged by the host if there was already a parameter there).

readCount: if the parameter is an array, this is its size (as bytes), else the value is 0.

index: the index of the parameter, or -1 if no parameter was found

In order to retrieve the parameters one by one, the plug-in can then use offlineRead in the following way:

Input parameters, as set by the plug-in before calling offlineRead:

readCount: should be 0 when retrieving a float parameter, else indicates the size of the buffer, as bytes, to receive the array.

extraBuffer: buffer allocated by the plug-in to receive the parameter. If the parameter is a float, this pointer is actually a float* pointer, else this is a pointer to an array.

readPosition: position where to read the parameter. If there is no parameter at this position, the host returns false.

index: index of the parameter to retrieve.

Output parameters, as set by the host after calling offlineRead:

index: index of the next parameter, else -1.

readPosition: position of next recorded parameter, or -1 if no more parameter. This is an useful hint for the plug-in, to make it easy and fast to retrieve sequentially all recorded parameters.

readCount: if the next parameter is a float, this value is 0. If it is an array, this value is the byte-size of this array.

`kVstOfflineMarker`

With this option, the plug-in can create one or more markers in the file, or move existing ones.

To know which markers currently exist in a given file, the plug-in can use offlineRead, with the following parameters:

extraBuffer: buffer allocated by the plug-in, to receive a copy of the markers. If this value is NULL, then offlineRead sets the number of markers into the member readCount. This is a way for the plug-in to know how many markers exist in the file, and therefore to allocate a proper buffer size (and call again offlineRead).

readCount: the size of the buffer (number of VstAudioFileMarker elements). If this size is not equal to the current number of markers, the offlineRead function should return false.

To write new markers:

extraBuffer: a buffer allocated by the plug-in that holds the markers to create, and only them.

writeCount: the number of markers in the buffer.

Important: the member id of a marker to create must be 0. When returning from the offlineWrite function, the id of the marker has been initialized by the host (it could be reused by the plug-in to move the marker later).

To move existing markers:

extraBuffer: a buffer allocated by the plug-in that holds the markers to move.

These markers must have been previously retrieved through offlineRead.

The host identifies the markers to move by checking the id member of the markers.

The position member of a marker structure represents the new position that a marker should adopt.

If position is -1, then the host deletes the markers.

writeCount: the number of markers in the buffer.

To copy markers from one file to another:

If the plug-in creates a new file out of a source file, it might be convenient to copy the source file markers into the new file. In this case, the plug-in can call the offlineWrite function with the following parameters:

extraBuffer: NULL

index: index of the VstOfflineTask structure that corresponds to the source file.

`kVstOfflineCursor`

By calling offlineRead with this option, the plug-in retrieves the file's edit-cursor position:

readPosition: position of the cursor, or -1 if no edit-cursor

index: bit mask describing on which channel(s) lies the edit-cursor.

To move the edit cursor to a certain location, the plug-in should initialize the following members:

writePosition: position of the cursor

index: bit mask describing on which channel(s) should lie the edit-cursor. -1 means all channels. If the host does not support the placement of the edit-cursor on individual channels, it should ignore this parameter.

It's worth noting that "edit-cursor" position does not mean "playback-cursor" position.

`kVstOfflineSelection`

By calling offlineRead with this option, the plug-in retrieves the current sample selection in the file:

positionToProcessFrom: the first selected frame, or -1

numFramesToProcess: the size of the selection, or 0

index: bit mask describing which channel(s) are selected

To set a selection in the file, the plug-in should initialize the above members.

If the host does not support the selection of individual channels, it should ignore index and select all channels.

If the host does not support sample selections, offlineWrite should return false.

`kVstOfflineQueryFiles`

If the plug-in desires to get notified about which files are available in the host, it should call offlineRead with this option.

The first parameter of offlineRead (VstOfflineTask*) should be NULL. On receiving this call, the host should call, immedialty or later, offlineNotify, with the start parameter set to false. In other words, the kVstOfflineQueryFiles option is a way to force the host to call offlineNotify, in order to get informed about open files (normally, the host only calls offlineNotify if a change happens in the set of open files). It is important to insist on the fact that the host is free to call kVstOfflineQueryFiles asynchronously, ie. not immediatly when kVstOfflineQueryFiles is called.

Normally, this option is only used if the plug-in needs to be notified about the file information, in order to update its user interface.

Functions

`bool offlineNotify(VstAudioFile* ptr, long numAudioFiles, bool start)`

The host calls this plug-in function in two cases:

When the plug-in's process is to be executed. E.g. the user has pressed the "Process" button. In that case, the "start" parameter is true.

Anytime a change happens in the set of files open in the host. In that case, the "start" parameter is false. The purpose of this notification is to give the plug-in a chance to update its user interface according to the host environment. For example:

The plug-in might display the list of all files available for processing; this list needs to be updated if a new file is open or closed in the host.

The plug-in might display some information about the file with the focus: this needs to change if a new file gains the focus.

Etc...

Tip: since the VstAudioFile structure contains parameters that are likely to often change, such as cursor position or sample selection, the offlineNotify function might be called often. Therefore, a good design for a plug-in that needs to update its user interface would be to cache the VstAudioFile settings, so as to actually update its user interface only when really required (eg. if the plug-in does not care about the edit-cursor position in a file. It should not update its user-interface only if the edit-cursor position happens to move in a file).

The host as aparameter passes an array of VstAudioFile structures.

The number of elements in this array is given by the parameter numAudioFiles.

numAudioFiles is 0 if there is no open file in the host, and in that case, the parameter "ptr" is NULL.

The first element of the array always represents the file with the focus.

If the "start" argument is true, the plug-in should start the process by calling offlineStart. Else, the plug-in might, or might not, starts a read-only process, to update its user-interface. See offlineStart.

Whatever the state of the start argument, the plug-in should return false from the function if it can't process the file(s) open in the host. E.g. if the plug-in only works with stereo files, and the file with the focus is mono, the plug-in should return false from this function. This allows the host, for instance, to disable the process button of the user interface.

Important: the plug-in should not initialize anything internally at this stage. All internal initialization and cleanup required for the process should happen inside offlineRun, and only there.

`bool offlinePrepare(VstOfflineTask* offline, long count)`

The host calls this function so that the plug-in complements the VstOfflineTask structure(s). If everything is fine, the function should return true.

Important: the plug-in should not initialize anything internally at this stage. All internal initialization and cleanup required for the process should happen inside offlineRun, and only there.

`bool offlineRun(VstOfflineTask* offline, long count)`

This function is called by the host once the VstOfflineTask structure(s) is(are) ready. Within this function, the plug-in does its audio processing and calls offlineRead and offlineWrite at will. If any error is detected during this procedure, the function should return false.

Important: all internal initialization and cleanup required for the process should happen inside offlineRun, and only there. E.g. if the plug-in should allocate some memory, this should be done inside this function (as well as the deallocation).

`bool offlineStart(VstAudioFile* ptr, long numAudioFiles, long numNewAudioFiles)`

When the function offlineNotify is called, the plug-in might decide to start a process. For this purpose, the plug-in has to decide which file(s) to process, and also if some new file(s) should be created.

By setting the member flag of each VstAudioFile structure, the plug-in instructs the host about which file(s) should be processed. In many cases, only the first VstAudioFile element (the focused file) is concerned.

The parameter numAudioFiles is simply the one passed from offlineNotify.

The parameter numNewAudioFiles is the number of files that the plug-in want to create.

E.g. if the plug-in selects one file from the VstAudioFile array and sets the value of numNewAudioFiles to 1, the host will create two VstOfflineTask structures. By convention, all VstOfflineTask structures corresponding to new files are placed by the host at the end of the array passed to offlineRun (ie, the front of the array corresponds to already existing files).

It is not allowed for a plug-in to call offlineStart if the plug-in is not itself called with offlineNotify. This is to ensure a synchronous protocol with the host. If the plug-in would call offlineStart asynchronously, maybe the VstAudioFile structures would not be valid anymore at that time, resulting in an undefined behaviour.

`bool offlineRead(VstOfflineTask* offline, VstOfflineOption option, bool readSource = true)`

This function is called by the plug-in to read data. See enum VstOfflineOption to see what kind of data can be read, apart audio samples.

About the parameter readSource:

As already seen, a single VstOfflineTask structure can be used both to read an existing file, and to overwrite it. Moreover, the offline specification states that it is possible, at any time, to read both the original samples and the new ones (the "overwritten" samples). This is the reason for the readSource parameter: set it to true to read the original samples and to false to read the recently written samples.

```
bool offlineWrite(VstOfflineTask* offline, VstOfflineOption option)
```
This function is called by the plug-in to write data. See enum VstOfflineOption to see what kind of data can be written, apart audio samples.

What's New in VST 2.0 – Reference Section
Surround Sound Support

The VST 2.0 specification provides structure and methods for hosts and plug-ins to communicate their surround sound capabilities.

In order to realize surround processing, some means of definitions are required to describe the target listening environment. In VST this is accomplished thru VstSpeakerProperties:

```
struct VstSpeakerProperties
{                        // units:    range:         except:
    float azimuth;       // rad       -PI...PI        10.f for LFE channel
    float elevation;     // rad       -PI/2...PI/2    10.f for LFE channel
    float radius;        // meter                     0.f for LFE channel
    float reserved;      // 0.
    char  name[64];      // for new setups, new names should be given (L/R/C... won't do)
    char  future[32];
};
// note: the origin for azimuth is right (as by math conventions dealing with radians);
// the elevation origin is also right, visualizing a rotation of a circle across the
// -pi/pi axis of the horizontal circle. thus, an elevation of -pi/2 corresponds
// to bottom, and a speaker standing on the left, and 'beaming' upwards would have
// an azimuth of -pi, and an elevation of pi/2.
// for user interface representation, grads are more likely to be used, and the
// origins will obviously 'shift' accordingly.

struct VstSpeakerArrangement
{
    float lfeGain;              // LFE channel gain is adjusted [dB] higher than other channels
    long numChannels;          // number of channels in this speaker arrangement
    VstSpeakerProperties speakers[8];   // variable
};
```

Finally there are methods in the AudioEffectX class that allows the host to inquire what the plug-in is capable of.

```
virtual bool getSpeakerArrangement (VstSpeakerArrangement* plug-inInput,
                    VstSpeakerArrangement* plug-inOutput);

virtual bool setSpeakerArrangement (VstSpeakerArrangement* plug-inInput,
                    VstSpeakerArrangement* plug-inOutput) {return false;}
```
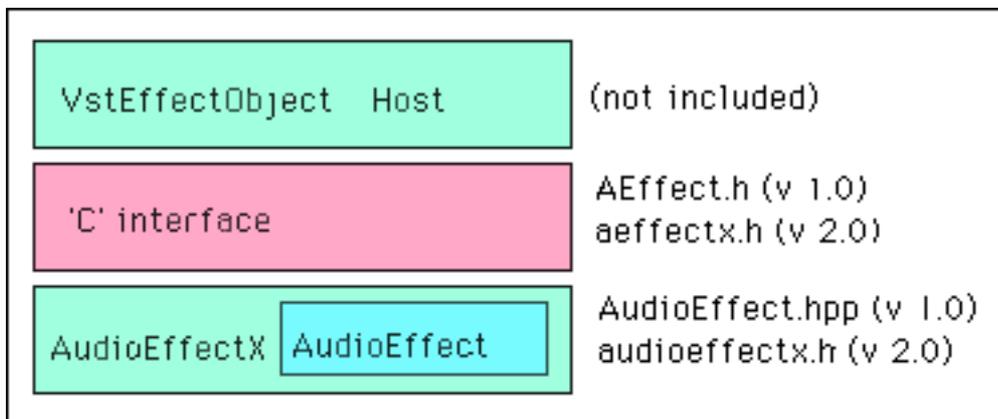
Host & Plug-in Communication

This documentation is primarily concerned with the implementation of the plug-in side of the VST interfaces. More details about hosting VST Plug-ins will be available in another SDK.

The hosting application will gather all available plugs from the registry, or dedicated folder. Important information about the properties of VST plug-ins may be stored externally, depending on the platform (registry, resource file etc).

Communication between a VST plug-in and the host is based around two 'C' header files. These are aeffectx.h (representing version SDK 2.0 features), which includes AEffect.h (version SDK 1.0).

AEffect.h has been left unchanged so to guarantee version 1 compatibility.

When developing an application which hosts VST plug-ins, you will have to implement the callback passed as an argument to the plug's main function, and implement as many of the audioMasterXX selectors as possible. The header files provide detailed information about the parameter usage.

Steinberg Products supporting VST 2.0

Please Note, this information will be probably be outdated by the time you read this because of the number of people eagerly waiting for this SDK. We definitely recommend joining the VST Plug-Ins mailing list for up-to-date information about who is supporting the VST2.0 interface, the products already available for VST 2.0 and share the experiences other vendors have when implementing VST 2.0 Plug-Ins and VST 2.0 Host software.

Join up by sending email to **Majordomo@steinberg.net**

with either of the following messages in the BODY of the email message:

subscribe vst-plugins

subscribe vst-plugins-digest

Current VST2.0 compatible Programs from Steinberg.

As this document was being prepared (June 1999) the following Steinberg products support parts of the 2.0 specification:

• Cubase 3.7 for Windows

• Cubase 4.1 for Apple Macintosh

• Wavelab 3.0 for Windows

VST 2.0 Support in Cubase 3.7 Windows and Cubase 4.1 Mac

Cubase 3.7 Windows and Cubase 4.1 Mac support the same set of functionality:

• Vst Instruments will receive MIDI channel data*

• Cubase will receive and record MIDI channel data from any plug-in*

• Cubase will provide full VstTime info*

Notes: Specific to Cubase 3.7 Windows and Cubase 4.1 Mac
The Cubase host will be able to send MIDI events with the valid status bytes of 0x80 through to 0xEF to a plug-in.

Additionally a VST 2.0 Plug-in will we able to send MIDI data to the Host application with the sendVstEventsToHost()

When VstTime information is passed from the Cubase Host to a plug-in please note that the nanoSeconds field is not aligned to the samplePos, but rather the current system time when getTimeInfo() was issued. You should not use this field other than for approximate calculations. ppqPos, tempo, cycleStartPos, cycleEndPos, smpteOffset, and smpteFrameRate are always valid regardless of the request mask. samplePos will follow cycle mode when cycle is activated (that is, it will jump backwards when the right locator is passed).

Cubase 4.0 Windows may have additional vst 2.0 support when released.

Wavelab 3.0 for Windows supports the Vst Offline interface.

And obviously, many of our (and other vendor's) plug-ins will support 2.0 functionality.

Licensing issues & Trademark Usage
Steinberg VST Plug-ins SDK Licensing Agreement

The Steinberg VST Plug-in Software Development Kit can be used freely subject to certain licensing conditions by anyone interested in developing Plug-ins, or to extend an application so that it's capable of hosting VST Plug-Ins. When you chose to receive the VST Plug-in SDK you declared yourself to be in agreement with the Licensing conditions.

These notes don't replace the licensing agreement in any way, but quickly explain what you can and cannot do with the Steinberg VST Plug-in Software Development Kit.

The License:
• is not time limited.
• is subject to the laws of the Federal Republic of Germany only.
• & its remaining conditions remain valid even if one of the conditions become invalid.

This SDK:
• is created with the sole aim that you can create or host VST Plug-ins.
• is offered 'AS IS' and we make no claims about how suitable it is for your application.

Steinberg:
• still holds the copyright for VST Plug-in specification.
• are not responsible for anything that happens because you chose to use the VST Plug-In Interface.
• cannot be held liable if your use of the VST Plug-In Interfaces causes damage or loss to anyone.
• may release improved versions of the Licensed Software Developer Kit
• offer no commitment that updates will occur at anytime or for anybody.
• are not aware of VST Plug-in technology infringing the intellectual copyright of anyone else,
• cannot accept any responsibility for any claims made against you.

You cannot:
• transfer your License to anyone
• license the information contained in this SDK to anyone else.
• sell the information contained in this SDK to anyone.
• re-work or otherwise pass this technology off as your own.
• give it away or in any other way distribute it, or cause it to be become distributed.
• use the VST logo or other marks on promotional merchandise. So no VST t-shirts or baseball caps.
• claim to be a partner of Steinberg or be acting on our behalf.
• make any statements on Steinberg's behalf.

You should not:
• bring the VST technology into disrepute, or damage its reputation in any way.
• use VST technology in connection with products that are obscene, pornographic .
• use VST technology in connection with products that are excessively violent, or in poor taste.
• break the rules of the license, or we have the right terminate the License immediately.

You have to:
• include references to Steinberg's copyrights and trademarks in a product that uses this SDK.
• ensure "VST is a trademark of Steinberg Soft- und Hardware GmbH" appears on any packaging
• place the VST Logo on packages and promotional material. We provide the artwork.
• add our copyright notice to your about box. "VST Plug-In Technology by Steinberg."
• agree that we hold your details on file for our internal purposes.
• inform Steinberg immediately if any one makes a claim against you in regard to VST-Plug-ins.
• make sure the end-publisher of your work is also a VST license holder.

You can:
• make vst plug-ins or vst-host applications and distribute them worldwide.
• release products without having to pay to use the VST Plug-in Interface technology
• use VST technology in demo-versions of your products.

Please read the License Agreement!

Thanks & Acknowledgements

The VSTGUI interfaces & Libraries were created by Wolfgang Kundrus and Yvan Grabit. The time they have invested has really made the huge inroads into cross-platform code compatibility.

Thanks too, to Philippe Goutier who provided the VST Offline Interface.

And to Jens Osbahr from Spectral Design who provided the VST Surround Interfaces.

Not forgetting Dave Nicholson who did some additional typing.

There were many more people involved in the creation of this interface, not all of whom can be mentioned here. We extend our thanks not just to the Steinberg Staff and Associates that contributed but also to the many other developers from other parties have helped with their input - mainly all the nice people on the VST PLUG-IN DEVELOPER MAILING LIST

Thanks to all of you for having made this possible.
Charlie Steinberg 1999