Presentation Title: Advanced VisualAudio®

Presenter Name: Paul Beckmann

Chapter 1: Introduction

Subchapter 1a: Overview

Hello and welcome. My name is Paul Beckmann. I'm an engineer with Analog Devices, and today I'm going to be talking about advanced features of VisualAudio.

This presentation provides training on the advanced features of VisualAudio. This is a tool for accelerating the development of audio products. Examples and demonstrations today will be based upon the Blackfin 533 EZ-KIT, although VisualAudio supports a number of Blackfin and SHARC processors. You will learn about advanced tool features such as high and low-level variables, the expression language, and using presets. You'll learn how to use the external interface to control VisualAudio from other applications such as MATLAB and you will also learn the basics of writing audio modules. The target audience for this presentation is audio algorithm developers. You should be comfortable writing C code and have some familiarity with the Blackfin processors and the VisualDSP++ development environment.

The outline is as follows: First of all we're going to talk about advanced features of the VisualAudio Designer application. This includes high- and low-level parameters, the expression language and presets. We're going to talk about using the external interface with particular emphasis on MATLAB, then we're going to talk about writing custom audio modules, and then we'll finally conclude.

Chapter 2: Advanced Features

Sub-chapter 2a: Module Variables

Let's start with the advanced features of the VisualAudio Designer application. First of all, there are high-level and low-level module variables. The high-level variables are the ones which appear on a module's inspectors. We call that an inspector, is an interface for adjusting the parameters of an audio module. What's shown here is the inspector for a bass tone control. And you can adjust the smoothing time, the gain in DB, and also the frequency.

Now, shown on the inspector, the three variables, those are the high-level variables. If you look more closely at the data structure associated with the audio module, you'll see that there are a number of other parameters. These are the low-level ones, or also called the render variables, and these appear within the module's data structures. So on the one hand we have the high-level variables shown on the inspector, and then we have the low-level variables which are in the data structure on the DSP.

Sub-chapter 2b: Expression Language

Converting between the high-level and the low-level parameters is the expression language. Expression language is a simple scripting language that takes the high-level parameters, does some basic mathematical operations on them and uses those results to set the low-level parameters. Expression language is typically used for things like converting from a smoothing time in milliseconds, to a smoothing coefficient, converting from dB units to linear units, or possibly to convert a balance control, a single control into two separate gains. So again, there's the inspector, which contains the high-level variables, goes through the expression language and then gets turned into low-level parameters in the audio module structures. In some cases, you could, in fact, have high-level parameters that map directly to low-level parameters without any expression language in between.

Sub-chapter 2c: Presets

Another mechanism within VisualAudio is what's referred to as presets. Presets are a convenient mechanism for managing audio-module parameter sets. Steps in using a preset are as follows: First of all, you tune your system to a desired state using either the inspectors or the external interface. So shown here is one of the inspectors. Shown here is an external interface designed using MATLAB. So you tune up the system the way you want.

The next thing you do is tell VisualAudio to capture the preset. It will display a list of all the audio modules that you have in your layout. You can click on the checkboxes to select those, and then you name the preset and everything that gets selected here gets saved with this preset.

What you can do then after you've created a few presets, is you can select the presets from the tool. There's a drop list here that gives you a list of all the presets you've created, and you just select one and release the mouse button, and that sends all the audio module parameters associated with those presets down to the DSP. What you can also do, is after you've created some presets, maybe you've determined the ones that sound right, you can also select presets and optionally compile them with the executable. So presets live both in the PC application itself (you can have a large number of presets) and then you can select the presets you want to compile down onto the DSP. Once they're compiled on the DSP, you can enable those presets from the DSP code.

Some other things about presets. Presets are written in Intel hex format, so we use a text file format for that. You can store them on the host and download them to the DSP, or you could download or store them on the DSP itself. Typical uses for presets are, for example, dealing with multiple sample rates, so your system may have to operate at, for example, 32 kHz, 44.1 and 48 kHz, so you'd create a preset for each of the sample rates that your system needs to operate on. You might use them for preserving default EQ settings, or even to make A/B comparisons between different parameter sets in order to fine tune the audio performance of a system.

Chapter 3: The External Interface

Sub-chapter 3a: The External Interface

Now, let's talk about the external interface. The external interface works both in design mode and tuning mode. Recall that design mode is where you instantiate audio modules, wire them together, and set audio module parameters. Tuning mode, on the other hand, is when the executable is running on the target hardware and the changes you make on the application get sent down to the DSP in real-time. The external interface works both in both design mode and tuning mode.

When in design mode, the changes happen to the audio module data structures residing on the PC. And in tuning mode, the changes happen to the audio module data structures on the PC and they're also sent to the DSP in real-time. Capabilities of the external interface include manipulating audio module parameters so you can get and set audio module parameters on the external interface. There's basic control of the system, such as loading, saving, building, capturing presets and so forth. There's also advanced control, such as instantiating audio modules, naming them and wiring them together. And there's also a mode that enables you to exchange audio data with the target processor. This process is very useful for regression testing. Audio is exchanged block-by-block between the PC and the running DSP. It happens in non-real-time and the speed of the data exchange is determined by the speed of the tuning interface. The external interface is implemented as a local COM server, housed in an .EXE. And it's accessible by any COM-compliant language: C, C++, Excel, VisualBasic and so forth. A total of 53 APIs and the program ID is VisualAudio Designer.

Typical uses of the external interface are creating custom audio module design functions. You might have some detailed filter design calculations. You could also use it for creating custom GUIs, so maybe you want a simple control panel that controls multiple module parameters, or you may want to provide full or restricted access to certain audio module parameters. That can be done through the external interface.

You can also leverage existing design tools and methodologies that you might have. So instead of starting over with VisualAudio, you can use your existing tools and interface them to VisualAudio using external interface. External interface is also useful for automating system design and tuning and also for regression testing of audio modules and systems. So, for example, you could design an audio module, test it using external interface to make sure it's working properly, embed that into a larger system and then test the larger system using the regression testing capabilities of the external interface.

The expression language is included in the external interface. For example, here you have a COM-aware application, it interfaces via the external interface, and the changes you make, in fact, flow through the expression language. So you can access the high-level variables, and there's also another way to access the low-level render variables directly on the DSP.

When you make a change to a high-level variable, if there's an expression associated with it, the expression language is invoked and the low-level render variables are updated as well. The direct access to low-level render variables is also very useful. For example, you could reset state variables on a filter and so forth. And that allows you to manipulate the DSP variables directly.

Sub-chapter 3b: MATLAB Interface

What we've also done is that we know that a number of customers use MATLAB as a preferred design environment. So what we've done is we've created a special layer that simplifies usage of VisualAudio with MATLAB. When you use this, each audio module appears as a MATLAB object, and objects can be manipulated as if they were MATLAB structures. Again, using this approach, the expression language is included as well when you make changes through MATLAB.

Let me give you some examples here. Here's the audio processing layout that's a simple stereo system. There's tone controls bass control, treble control, there's a volume control with built in loudness compensation and then these three modules here form a peak limiter. So the command here, "va_module", and the first argument you give it the instance name of the module. In this case, it's VolumeFletcherMunson_S1. That's the name of this audio module. When you execute this command, it queries VisualAudio for information regarding this audio module, the variables, what data types, sizes and so forth. And then it generates and returns a MATLAB object that represents the volume control.

Let me demonstrate this in MATLAB. So I'm going to start and I'm going to query VisualAudio for a single audio module. So we're going to query the volume control. The name of the audio module is VolumeFletcherMunson_S1. What this command does is it creates a MATLAB object with the same name as the volume control.

Let's take a look at the fields of this data structure. The data structure has four fields: the smoothing time gain, low frequency, and low Q. And if I switch back to VisualAudio, and open up the inspector for the volume control, you'll see that there's a one-on-one correspondence between the elements shown on the inspector, and the fields of that data structure. What I can do then is I can treat this object or data structure simply as if it were a MATLAB structure and get and set module variables. For example, I'm going to set the gain of the volume control from 0 dB, that's what it's set to right now. It's the FM gain field. I'm going to set it from 0 dB to -20 dB. It changes the value, and if I switch back VisualAudio, you'll see that the gain here has been set to -20 dB.

What's really handy about this is it's good for configuring the parameters of audio modules in a script and doing this in a repeatable fashion. So, in fact, I can access any of the parameters here. I can get and set them. I can use the full capabilities of MATLAB to design filters and so forth.

What's also nice about this is MATLAB also continues to work in tuning mode. Let me demonstrate that for you. I'll switch back to VisualAudio. I'm going to generate code for this layout. I'm going to go to VisualDSP++. I have the associated project for this platform loaded, and I'm going to halt the processor, and I'm going to build this executable. So VisualDSP++ is now compiling, linking the project. It's going to download it to the DSP and we're going to get this running in real-time now. I'm going to switch back to VisualAudio and go into tuning mode. So now in tuning mode, any changes that I make are going to happen in real-time on the DSP as well.

Let's switch back to MATLAB and I'm going to start the music here. You can hear the music here. Let's see what the volume control is at -20 dB. I'm going to set this to -10 dB, so it's going to get louder. So all the way to full volume 0 dB and then back down to, going to mute it, go to a -100 dB. So you can see all the changes that happen in MATLAB also happen in real-time on the DSP.

So in this example, I got the high-level module parameters of this volume control. VisualAudio also provides the ability to get to the low-level render variables through the external interface. I'm going to reissue this command, and I'm going to give it a second argument. Okay, that says either high level or low level mode. By default it gives me high-level mode. I'm going to request it to do low-level mode.

And let's take a look at what this data structure looks like now. Instead of having the four fields shown on the high-level inspector, what we have now is all of the variables from the low-level data structure on the DSP. So you can see that the four high-level variables map to about ten different DSP variables. You'll also see that there are these state variables, and you'll have to refer to the volume control documentation to understand exactly what each of these parameters does.

But what I want to point out is that these state variables are, in fact, being grabbed from the DSP in real-time as it's running. So if I query that module structure again, you'll see that the state variables are changing, and they're changing, in fact, in response to the audio as it's being processed. So MATLAB provides you access to the high-level inspector variables, and also to the low-level render variables.

Sub-chapter 3c: Regression Testing

And let me go back to my presentation. VisualAudio also provides regression testing capabilities. What happens is you place a platform in demand render mode. At that point, an external application generates data, passes the data via the external interface block by block through the tuning interface. Basically, you specify the audio inputs, the data is passed through the audio processing layout, and then the audio outputs are returned to the external application. And the external application analyzes the data for correctness. All the regression testing capabilities are also available through MATLAB and we provide MATLAB examples on how to do this.

This is what it looks like to use the testing API through MATLAB. First of all you issued a command, va_demandrender and you tell it to begin. When you issue that command, it halts the real-time flow of audio and the platform waits for you to send data from the PC down to the DSP. What you do next is you issue the va_demandrender('process') command, and you give it an input argument which is the data in. DATA_IN is a matrix of the input data. There's one column per input channel. And the size of it is TickSize, so it's the size of the block times the number of inputs.

This data is sent down to the DSP. The DSP processes it and then returns a matrix of output data. Again, there's one output channel per column of the matrix. And you can repeat this for multiple blocks or large blocks. When you're finally done with demand render mode, you issue – va_demandrender('end'); – and that resumes the real-time processing.

Here's an example using tone control. First of all, you place all the modules into bypass mode except the treble tone control. In bypass mode, audio flows through a module unchanged. What you do next is you generate input data in MATLAB and the example here is I'm generating a logarithmic chirp starting from 20 Hz to 24 kHz. Next thing I do is I start demandrender mode. I pass the data through the DSP, and then I halt demandrender mode. So with this setup, the only module which should be affecting the audio is the treble tone control. And in the example here, I have the treble tone control set to 6 dB of gain. So this is what the output of the tone control looks like.

For low frequencies, we have a gain of 1 or 0 dB, so it doesn't affect the low frequencies. And as we go higher and higher in frequency, at some point the treble tone control kicks in and it starts boosting the high frequencies. And it boosts it all the way up to a total of 6 dB or a factor of two. So using this method, you can essentially verify the operation of a single audio module or the operation of an overall audio layout.

Chapter 4: Custom Audio Modules

Sub-chapter 4a: Standard vs. Custom

Next we're going to look at writing custom audio modules within VisualAudio. We have standard audio modules and custom audio modules. Standard audio modules are supplied with VisualAudio with separate libraries for SHARC and Blackfin. Custom modules, on the other hand, are written by the user. Standard and custom modules appear on separate tabs within the module palette. Here I'm showing the standard modules. There's another tab here for custom audio modules. And, in fact, the only distinction between standard and custom modules is which tab it appears on. There's no limitation or cost overhead associated with writing custom modules. We also provide source code for all the standard modules. What's nice is that this source code serves as a starting point for writing your own customer audio modules.

Sub-chapter 4b: Module Components

There are three components of an audio module. There's a header file that contains a module's run-time interface and the description of the associated data structure. There's a module's run-time function, or we use the term "render function". This can be written in C, in Assembly code, or provided as object or as a library. Lastly, there's an XML file that describes a module in detail to VisualAudio. It contains, for example, elements of its data structure, it describes the inspector interface, containing both the high- and low-level variables, and any expressions. And there's also memory allocation rules – should an array be allocated in internal memory, external memory, and so forth.

Let's look at the instance data structure. Each instance of an audio module has an associated C data structure. All data structures start with the same set of fields. These fields contain elements common to all audio modules. And it, for example, describes the base class of the audio object. After the common header, this is followed by module-specific fields. For example, this is what you'd find in the header file for the ScalerSmoothed control. This is a smoothly varying single input, single output gain control. What you have here is you have the common header, followed by the specific parameters for this module. There's ampSmoothing, oneOverTickSize, and so forth. These are 32-bit fractional values. This is a 16-bit fractional value. Finally, there's a typedef name. So this defines the type for this ScalerSmoothed.

Sub-chapter 4c: Render Function

Next you have an associated render function. Again, they can be written in C or Assembly code and most of the modules provided with VisualAudio are written in Assembly code. The example here is in C code just for readability, although the actual code provided with VisualAudio is in Assembly. So let's start out with the function arguments.

Each audio module takes three input arguments. First of all, there's a pointer to the instance data structure. There's an array of buffer pointers. And there's an integer which specifics the TickSize: how many samples this processing function is receiving as its input. What's happening here in the code is I'm pulling out some of the instance variables from the data structure into local variables. I then get a pointer to the input data and a pointer to the output data. What you need to remember is the input and output buffers, you get a single array of buffer pointers. The buffers are ordered as input buffers followed by output buffers, followed by any scratch buffers. In this module, there's only a single input buffer, one output buffer and no scratch buffers.

So, the first buffer pointer gives you the input, second one gives you the output data. What you do then you apply the processing, and in this case the smoothing control updates using a first order filter on a block-by-block basis. And then internally within each block, it applies the gain difference as a linear amp. The function calls you see here are also intrinsic operations on the Blackfin which do fractional arithmetic. And here it loops over an entire block to each process

TickSize elements performing the processing, taking the input, processing it, and deriving the output. This is, in fact, how all audio modules are written. There's three input arguments: a pointer to the instance structure, an array of pointers, and the TickSize.

Sub-chapter 4d: Class Structure

There's also a class structure. So in addition to an instance structure, all audio modules of the same type share a single class structure. For example, if I had multiple volume controls within my system, each volume control would have its own instance structure, and then there would be a single class structure. The class structure describes the behavior of the audio module to VisualAudio's run-time interface. It contains, for example, the number of inputs and outputs, specifies is it a mono input or a stereo input, and so forth.

In this example, it says it's a mono input and a mono output, gives the name of the render function, and also discusses the bypass behavior. In VisualAudio you can bypass a module, which case the inputs are copied directly to the outputs. This class structure is typically declared within the module's C file. If the module's render function is written in Assembly, you would have two files. There would be an Assembly file containing the render function, and a C file with the class structure declaration.

Take a look at the figure down here. In this case, I have three instances of the ScalerSmoothed. Each of those has a separate instance structure and they all point to a single class structure for the ScalerSmoothed. And if I had delays in the system, there's two instances of delay, there'd be separate instance structures, and they would both point to a separate delay class structure. So that's another part of an audio module.

This is what the class structure looks like for the particular ScalerSmoothed that we've been using. Again, there's a pointer to the render function. There's a field here that says to use the default bypass behavior. You can also specify a custom bypass function here. And then there's a descriptor for the inputs and outputs. And this refers to a single input and it's mono, a single output, and it's mono.

Sub-chapter 4e: XML File

And lastly, there's audio module XML. This XML file describes the audio module to the VisualAudio Designer application. It contains, for example, the name of the audio module, where it should appear within the tree view in the module palette. It describes the input and output pins, what data types they are, how many pins it has, lists which processors the module is compatible with. For example, is it a SHARC, is it a Blackfin? If it's a Blackfin, which Blackfin processors are supported. It describes the instance data structure, what are the fields of the data structure. Lists out the high-level variables, and any expressions. It talks about memory allocation rules, and also finally, a few other miscellaneous usage rules.

Chapter 5: Conclusion

Sub-chapter 5a: Summary

So this concludes the presentation on VisualAudio's advanced design features. These design features simplify the development of advanced audio features. We discussed high- and low-level variables, we discussed the expression language, talked about how to use presets. There's also an open API that allows VisualAudio's capabilities to be extended by interfacing to external COM-compliant applications. There's a special layer for simplifying use with MATLAB, and finally you can also write custom audio modules.

Sub-chapter 5b: Additional Information

You can get more information on VisualAudio from a number of places. First of all, a free download is available at the VisualAudio product page. This includes a VisualAudio Designer application, EZ Kit platforms, and audio module libraries for the SHARC and Blackfin. The download also includes full product help, and that's another good source of information.

You can get additional examples and tutorials at the VisualAudio Developers website, shown here. You can email specific technical questions to the VisualAudio Support email address, or you can click the "Ask A Question" button. This concludes my presentation on VisualAudio advanced features. I'd like to thank you for your time and attention.