The World Leader in High Performance Signal Processing Solutions

VisualAudio
Advanced Features

Presented by:
Paul Beckmann
Analog Devices CPSG
About this Module

- This module provides advanced training on VisualAudio. Examples and demonstrations will be based on the ADSP-BF533 EZ-KIT. You will learn about:
  - Advanced tool features such as high and low-level variables, the expression language, and presets.
  - How to use the external interface to control VisualAudio from other applications, such as MATLAB.
  - The basics of writing audio modules.

- Target Audience
  - Audio algorithm developers
  - Comfortable writing C code
  - Some familiarity with Blackfin processors and the VisualDSP++ development environment
Module Outline

- VisualAudio Designer advanced features
  - High and low-level parameters
  - The expression language
  - Presets
- Using the external interface
- Writing custom audio modules
- Conclusion
VisualAudio Designer
Advanced Features
High and Low-Level Module Variables

High-level variables appear on a module’s inspector.

Low-level (or render) variables appear within the module’s data structure.

```c
typedef struct {
    AMF_Module b;
    float ampSmoothing, ampTarget, b0;
    float amp, state;
} AMF_ToneControlBass;
```
Expression Language

The expression language converts between high-level and low-level parameters

Expression language examples:

1. Convert from smoothing time (msecs) to coefficient
2. Convert from dB to linear units.
3. Convert a balance control setting into 2 separate gains
Presets

- Convenient mechanism for managing audio module parameter sets
- Step 1 – Tune the system to a desired state
  - Inspectors
  - External interface
- Step 2 – Capture the preset
- Step 3 – Apply the preset from the Tool
- Step 4 – Optionally compile the preset with the application.
- Presets are written in Intel hex format
- Can be stored on host and downloaded to the DSP
- Typical uses
  - Dealing with multiple sample rates
  - Preserving default EQ settings
  - Making A/B comparisons to fine tune system performance
Presets

- Convenient mechanism for managing audio module parameter sets
- Step 1 – Tune the system to a desired state
  - Inspectors
  - External interface
- Step 2 – Capture the preset
- Step 3 – Apply the preset from the Tool
- Step 4 – Optionally compile the preset with the application.
- Presets are written in Intel hex format
- Can be stored on host and downloaded to the DSP
- Typical uses
  - Dealing with multiple sample rates
  - Preserving default EQ settings
  - Making A/B comparisons to fine tune system performance
Presets

- Convenient mechanism for managing audio module parameter sets
- **Step 1** – Tune the system to a desired state
  - Inspectors
  - External interface
- **Step 2** – Capture the preset
- **Step 3** – Apply the preset from the Tool
- **Step 4** – Optionally compile the preset with the application.
- Presets are written in Intel hex format
- Can be stored on host and downloaded to the DSP
- Typical uses
  - Dealing with multiple sample rates
  - Preserving default EQ settings
  - Making A/B comparisons to fine tune system performance
Presets

- Convenient mechanism for managing audio module parameter sets
- Step 1 – Tune the system to a desired state
  - Inspectors
  - External interface
- Step 2 – Capture the preset
- Step 3 – Apply the preset from the Tool
- Step 4 – Optionally compile the preset with the application.
- Presets are written in Intel hex format
- Can be stored on host and downloaded to the DSP
- Typical uses
  - Dealing with multiple sample rates
  - Preserving default EQ settings
  - Making A/B comparisons to fine tune system performance
Presets

- Convenient mechanism for managing audio module parameter sets
- Step 1 – Tune the system to a desired state
  - Inspectors
  - External interface
- Step 2 – Capture the preset
- Step 3 – Apply the preset from the Tool
- Step 4 – Optionally compile the preset with the executable.
- Presets are written in Intel hex format
- Can be stored on the host and downloaded to the DSP
- Typical uses
  - Dealing with multiple sample rates
  - Preserving default EQ settings
  - Making A/B comparisons to fine tune system performance
The External Interface
External Interface

- **Works in both Design Mode and Tuning Mode**
  - Design mode ➔ module data structures
  - Tuning mode ➔ module data structures AND sent to the DSP in real-time

- **Capabilities**
  - Manipulating audio module parameters
  - Basic control of the system (loading, saving, building, capturing presets, etc.) is also supported
  - Advanced control (instantiating and wiring modules)
  - Exchanging audio data with the target processor.
    - Block-by-block
    - Non-real-time
    - Speed is determined by the speed of the tuning interface

- **Implemented as a local COM server (housed in an EXE)**
  - Accessible by any COM compliant language/application (C/C++, Excel, VisualBasic, etc.)
  - Total of 53 APIs supported
  - Prog-ID is ‘VisualAudioDesigner’
Uses of the External Interface

- Creating custom audio module design functions
- Creating custom GUIs
  - Control panels
  - Full or restricted functionality
- Leveraging existing design tools and methodologies
- Automating system design and tuning
- Regression testing of audio modules and systems
Expression Language is Included

External applications can access the high-level and low-level render variables.

Changes to high-level variables invoke the expression language.

Low-level accesses bypass the expression language and manipulate DSP variables directly.
MATLAB Interface Layer

- Simplifies usage with MATLAB
- Each audio module appears as a MATLAB object
- Objects can be manipulated as if they were MATLAB structures

MATLAB scripts

MATLAB Interface Layer

External Interface

Expression Language

Render variables in audio module structure
Querying a Single Audio Module

Queries VisualAudio for information regarding this audio module
- Variables names
- Data types
- Sizes

Generates and returns a MATLAB object

>> S = va_module('VolumeFletcherMunson_S1');
Comparison with the Inspector

\[ S = \]

- smoothingTime: [100]
- fmGain: [0]
- lowFreq: [30]
- lowQ: [1]

One-to-one correspondence between MATLAB structure members and interface variables shown on the inspector
Manipulating Module Parameters

- Treat them as if they were standard MATLAB structures:
  ```matlab
  VolumeFletcherMunson_S1.smoothingTime=50;
  VolumeFletcherMunson_S1.fmGain=-3;
  VolumeFletcherMunson_S1.lowFreq=40;
  VolumeFletcherMunson_S1.lowQ=1.1;
  ```

- Often used to initialize parameters in a repeatable method using a script file.
Accessing Low-Level Render Variables

- The previous example demonstrated how to access the high-level variables shown on the inspector.
- To access low-level render variables in Tuning Mode, issue the command:

  ```
  va_module('VolumeFletcherMunson_S1',0)
  ```

  ```
  VolumeFletcherMunson_S1 =
  ```

  ```
  ampSmoothing: [ 0.002081]
  ampTarget: [ 1.000000]
  lowAmpTarget: [ 1.000000]
  b0: [ 0.996081]
  b1: [ -1.996061]
  amp: [ 1.000000]
  lowAmp: [ 1.000000]
  aux_state1: [ 777.399841
  -438.739838]
  aux_state2: [ -21.297104
  177.371170]
  ```
Regression Testing Capabilities

Platform operates in “demand render mode”

External application generates data.

Audio passed block-by-block through the tuning interface.

External application analyzes data for correctness.

MATLAB examples are provided.
MATLAB Testing API

- Place platform into “demand render” mode
  
  ```matlab
  va_demandrender('begin');
  ```

- Send and receive individual blocks of data
  
  ```matlab
  DATA_OUT=va_demandrender('process', DATA_IN)
  DATA_IN=TickSize x NumberOfInputs
  DATA_OUT=TickSize x NumberOfOutputs
  ```

- Repeat for multiple blocks

- Exit demand render mode and resume real-time processing
  
  ```matlab
  va_demandrender('end');
  ```
Place all modules into bypass mode except the treble tone control
Generate input data in MATLAB
\[
t=((0:2047)/48000).';
\]
\[
\text{DATA}_\text{IN}=\text{chirp}(t, 20, t(\text{end}), 24000, 'logarithmic', -90)*\text{ones}(1,2);
\]
Process the data
\[
\text{va}_\text{demandrender('begin');}
\]
\[
\text{DATA}_\text{OUT}=\text{va}_\text{demandrender('process', DATA}_\text{IN});
\]
\[
\text{va}_\text{demandrender('end');}
\]
Tone Control Results
Writing Custom Audio Modules
Standard vs. Custom Modules

- Standard modules are supplied with VisualAudio
- Custom modules are written by the user
- Standard and custom modules appear on separate tabs within VisualAudio Designer. This is the only distinction between standard and custom modules – no limitations or cost overhead associated with custom modules
- Source code is provided for all standard modules. This serves as a starting point for creating custom modules
3 Components of an Audio Module

- A header file which contains the module’s run-time interface and a description of the associated data structure
- The module’s run-time function (“render function”). This can be:
  - C code
  - ASM code
  - Object or library
- An XML file that describes the module in detail to VisualAudio
  - Elements of its data structure
  - Inspector interface
  - Memory allocation rules
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 contain elements common to all audio modules
  - Describe the “base class” of the “object”
- This is followed by module specific fields
AMF_ScalerSmoothed.h

Defines the data structure associated with the audio module

```c
#include "AudioProcessing.h"

typedef struct
{
    AMF_Module b;

    // parameters
    fract32 ampSmoothing;
    fract32 oneOverTickSize;

    // state
    fract32 amp;

    // parameters
    fract16 ampTarget;
} AMF_ScalerSmoothed;
```

Common header

Module specific variables

typedef name
SEG_MOD_FAST_CODE void AMF_ScalerSmoothed_Render(AMF_ScalerSmoothed * instance,
    AMF_Signal ** buffers, int tickSize) {
    int i;
    fract32 amp = instance->amp;
    fract16 ampTarget = instance->ampTarget;
    fract32 ampSmoothing = instance->ampSmoothing;
    AMF_Signal *in = buffers[0];
    AMF_Signal *out = buffers[1];
    fract32 diff, slew;

    /* compute smoothing filter only once per tick, and derive a linear
     * slew for the per-sample update */
    diff = sub_fr1x32(mult_fr1x32x32NS(ampSmoothing, L_deposit_h(ampTarget)),
        mult_fr1x32x32NS(ampSmoothing, amp));
    instance->amp = add_fr1x32(amp, diff);
    slew = mult_fr1x32x32NS(diff, instance->oneOverTickSize);

    for (i=0; i<tickSize; i++) {
        out[i] = mult_fr1x32x32NS(in[i], amp);
        amp = add_fr1x32(amp, slew);
    }
}

- Same arguments passed to all render functions:
  - Pointer to instance structure
  - Array of buffer pointers ordered as inputs, outputs, and scratch
  - tickSize = block size
Class Structure Declaration

- All audio modules of the same type share a single “Class Structure”
- Describes the behavior of the module to VisualAudio’s run-time interface
  - Number of inputs and outputs
  - Mono input and mono output
  - Name of render function
  - Bypass behavior
- Typically declared within the module’s C file
  - If the module’s render function is in assembly, there will be two files: an .ASM file containing the render function, and a C file with the class structure declaration
SEG_MOD_SLOW_CONST const AMF_ModuleClass AMFClassScalerSmoothed =
{
    /** Flags. */
    0,

    /** Reference to render function. */
    (AMF_RenderFunction) AMF_ScalerSmoothed_Render,

    /* Default bypass */
    (void *)0,

    /* Input descriptor - 1 input, and it is mono. */
    1, 0,

    /* Output descriptor - 1 output, and it is mono. */
    1, 0,
};
Audio Module XML

- Describes the audio module to VisualAudio Designer
  - Module name and palette location
  - Input and output pins
  - Compatible processors
  - Instance data structure
  - High-level variables and expressions
  - Memory allocation rules
  - Other usage rules
Conclusion

- VisualAudio’s design features simplify the development of advanced audio features
  - High and Low-level variables
  - Expression language
  - Presets
- Open API’s allow VisualAudio’s capabilities to be extended by
  - Interfacing to external COM compliant applications
  - Interfacing to MATLAB
  - Writing custom audio modules
For Additional Information

- A free download is available at the VisualAudio product page

- Additional examples and tutorials can be found at the VisualAudio Developer’s Web site:
  - [www.visualaudiodeveloper.com](http://www.visualaudiodeveloper.com)

- Specific technical questions can be sent to:
  - [visualaudio.support@analog.com](mailto:visualaudio.support@analog.com)

- Click the “Ask A Question” button