Audio Engineering Society
Convention Paper

Presented at the 119th Convention
2005 October 7–10 New York, New York USA

VISUALAUDIO - AN ENVIRONMENT FOR DESIGNING, TUNING, AND TESTING EMBEDDED AUDIO APPLICATIONS

David A. Jaffe[1], Paul Beckmann[2], Britton Peddie[3], Timothy Stilson[4], and Scott Van Duyne[5]

Audio Rendering Technology Center (ARTC), Analog Devices, Inc., 1741 Technology Drive, Suite 400, San Jose, CA. 95110

[1]David.Jaffe@analog.com, [2]Paul.Beckmann@analog.com, [3]Britton.Peddie@analog.com, [4]Tim.Stilson@analog.com, [5]Scott.VanDuyne@analog.com

ABSTRACT

Different hardware configurations and applications suggest different audio system design trade-offs. VisualAudio is focused on embedded processor applications, and currently works with Analog Devices, Inc. SHARC and Blackfin processors. VisualAudio is appropriate for a wide range of applications, including general purpose audio, pro audio, music "stomp" boxes, consumer electronics (such as audio-visual receiver (AVR) systems), and automotive audio systems. This article describes the decisions that were made in the design of VisualAudio and how they are tailored to the embedded processing environment. It contrasts VisualAudio with previous systems created by the authors, particularly Staccato Systems' "SynthCore," currently known as Analog Devices' "SoundMAX."

1. OVERVIEW OF VISUALAUDIO

VisualAudio[1] is a development environment that includes a framework, audio modules, and audio decoders, all of which run on an embedded processor, as well as PC tools for creating audio applications.

1.1. SHARC and Blackfin

VisualAudio was originally written for the SHARC and its coverage has recently been expanded to include the Blackfin processor. These two processors are quite different. The SHARC is a 32-bit SIMD (single instruction multiple data) floating point DSP with a relatively large internal memory (for a DSP). The Blackfin is a high-clock-rate 2x16-bit SIMD fixed-point DSP with 32-bit embedded microcontroller capabilities (such as instruction and data caches, high-performance external SDRAM interface, and significant general-purpose extensions to the instruction set). The Blackfin also has extensive power management features and is thus appropriate for portable battery-powered devices. Both processors come in a variety of models, many with integrated audio peripherals such as asynchronous sampling rate converters, S/PDIF transceivers, etc.

Despite their differences, both processors are supported by a similar framework. In addition, complementary audio module sets allow processing networks to be easily moved between processors. In the SHARC implementation, the normal signal type is 32-bit floating-point; on the Blackfin, it is 32-bit (double precision) fixed-point.

1.2. Components of VisualAudio

The VisualAudio tool chain consists of the VisualAudio tool, which is a plug-in to VisualDSP++ (Analog Devices' integrated development and debugging environment), as well as a software framework which may be configured for various uses, a set of audio modules (commonly used audio processing functions), a set of platform support packages, a set of decoders, an external COM interface, a protocol for communicating with a host micro-controller (if any), and a MATLAB interface package. Three XML file formats are defined to ease integration of diverse components: a module description file, a target platform description file and a decoder description file.

The VisualAudio tool includes a GUI block diagram editor for designing audio signal flow graphs (also called "post-processing"), real-time adjustment of audio module and decoder parameters, transparent switching between hardware platforms, and other features. The framework supports audio I/O, identification of the S/PDIF content type, dynamic decoder instantiation, asynchronous sampling rate conversion, post-processing, a host protocol, a preset mechanism (for compactly storing and applying sets of parameters, doing A/B comparisons, storing multiple sample rate coefficient sets, etc.), and user control code. In addition, VisualAudio can generate post-processing systems independent of the VisualAudio framework. These can be integrated into customers' frameworks and tuned (as can VisualAudio frameworks) via the Background Telemetry Channel (BTC), RS232, or other method.

2. WHY YET ANOTHER AUDIO SYSTEM? – HISTORICAL CONTEXT

VisualAudio is the latest generation of a series of audio software environments developed by the authors in collaboration with others. Some historical context may clarify the terrain that has been explored and that informed the design of VisualAudio.

2.1. The PDP10/Samson Box

In the late 1970s, David Jaffe and Julius Smith worked as gradudate students at Stanford University's Center for Computer Research in Music and Acoustics (CCRMA) with an architecture consisting of a Digital Equipment Corporation mainframe PDP10 and a special-purpose audio processor, the Systems Concept Digital Synthesizer (known as the "Samson Box"). The Samson Box provided a fixed bank of basic audio functions that could be configured in various ways via registers, but were not fully programmable. The PDP10 was responsible for feeding commands to the Samson Box, and the Samson Box rendered those commands musically. The MUSBOX [1] (later, SAMBOX) software took a dynamic approach similar to MUSIC-N languages, with each "note" causing Samson Box resources to be dynamically allocated, then deallocated. Reflecting the raison d'etre of CCRMA, the focus was entirely on music.

[1] "VisualAudio," "SHARC," "Blackfin," "VisualDSP," and "SoundMAX" are registered as trademarks owned by Analog Devices, Inc.

2.2. The NeXT Music Kit, SynthBuilder and the Frankenstein Box

Beginning in 1987, Jaffe and Smith were hired by Steve Jobs of NeXT, Inc. to develop a music system for a new computer. This computer was the first personal computer to contain an embedded DSP for audio. The resulting software was the NeXT Music Kit [2,3]. As in the PDP10/SamsonBox environment, the main processor (a Motorola 68030, then later a 68040, and finally an Intel x86) was responsible for sending commands to the DSP and the DSP was responsible for rendering the audio. However, it differed from the PDP10/SamsonBox in a number of respects. The DSP was fully programmable and the goal was not just music but also sound effects for games, compression/decompression, etc. [4]. Most importantly, because this was a personal computer, the number of users was potentially much larger than those who had access to specialized hardware at research institutions. The software reflected this reality.

Like the PDP10/SamsonBox software, the Music Kit was entirely dynamic. In fact, it was more dynamic in that the primitive functions themselves could be downloaded on the fly while audio was processed. The system for allocating memory and processing cycles (MIPS) was extremely general [5].

The object-oriented design of the Music Kit made it easy to generalize to a multiple DSP architecture. This was first accomplished with the Ariel QuintProcessor [6], which contained five 56001s, then later with the NextSTEP version of the MusicKit [7], which ran on the Intel/PC architecture, using PC add-on cards. Each of the PC cards contained a 56001 and used the DSP's serial port for I/O. The most elaborate implementation of this was the prototype "Frankenstein box" which used a bank of Motorola 56002 evaluation boards [8].

In view of the growing importance of MIDI at that time, a design was developed that merged MIDI with the Music-N generality, without sacrificing any of the power of either of these. The Music Kit was the first to tackle this problem in its full theoretical implications, using a system of 32-bit "note tags," to which MIDI channels and key numbers were mapped. (In retrospect, the solution was perhaps too academically correct, and a simpler more practical solution might have sufficed.)

A need was seen for a graphical environment for configuring the Music Kit and interacting with it in performance. SynthBuilder [10] was developed by Nick Porcaro[2], some of the authors (Jaffe/Van Duyne/Stilson), and others at CCRMA, as part of the "Sondius" project, supported by the Stanford Office of Technology Licensing (OTL) as part of its program to maximize the value of its audio physical modeling patents. SynthBuilder provided the user a graphical user interface (GUI) to draw a block diagram of both the signal processing (running on the DSP) and the event processing (running on the PC.)

SynthBuilder took a somewhat static approach to using the MusicKit. The user drew a block diagram that was then downloaded to the DSP. At that point, it could be played via real-time commands sent to the DSP. For example, MIDI was received by the PC via a sound card and sent to the PC's main CPU, where software objects would process the MIDI via a graphical network of Music Kit "note filters," then convert the event to a series of DSP commands. SynthBuilder and the Music Kit were especially useful in creating physical modeling instruments, reflecting an interest of its authors that dates back to the early 1980's. Note that the MusicKit is still supported and available for MacOS, Linux and Windows [11].

2.3. SynthCore and SynthScript

In 1996, the Stanford OTL spun off the Sondius group to form Staccato Systems, Inc., with the charter of commercializing physical models for musical purposes. The technology chosen was SynthBuilder, but this time divorced from DSPs. Instead, a new audio engine called "SynthCore" was developed. SynthCore was designed as a server that would receive commands from a client such as SynthBuilder via a simple language called "SynthScript" [12]. SynthBuilder was modified to communicate with SynthCore via SynthScript or, alternatively, to save algorithms as SynthScript to be rendered by SynthCore in a stand-alone manner (e.g. in response to MIDI commands.)

SynthCore ran entirely on a PC, with both note filters and signal processing in a single thread of execution. It dynamically instantiated SynthScript algorithms, in

[2] SynthBuilder evolved from a prototype graphical application at NeXT, Inc. [9] and a student project by Eric Jordan at Princeton University.

response to incoming MIDI or API[3] calls. Custom audio functions could be dynamically loaded as plug-ins without needing to restart SynthCore. Client applications communicated with it via a COM interface.

In addition to basic audio processing functions, SynthCore included a highly-optimized wavetable engine that supported Yamaha XG format, as well as DLS2[4]. The wavetable engine could be used together with the basic audio processing functions to design more complicated systems. For example, a MIDI file could thus be rendered with a combination of physical modeling algorithms and wavetable voices. SynthCore was eventually shipped in millions of PCs as Analog Devices' SoundMAX [13].

Meanwhile, a significant market emerged for sound effects for computer games, where the responsiveness and realism of physical models is attractive. This lead to the development of the SynthCore SDK (later, Analog Devices' SoundMAX SDK), based on the SynthCore/SoundMAX engine, and shipped in a number of leading computer games. These effects were also used in a contemporary musical context in David Jaffe's "Racing Against Time," scored for two violins, two saxophones, piano and Mathews Radio Drum [14] which used a six-dimensional input device [15] to control SynthCore's rendering of physical models of automobiles, jet planes and electric guitars, as well as samples of the acoustic instruments.

2.4. From SynthCore to VisualAudio

In 2001, soon after Analog Devices acquired Staccato Systems, the authors demonstrated a prototype system running SynthCore on a SHARC ADSP-21161 SIMD (single instruction, multiple data) processor. The PC communicated with the SHARC via RS-232, sending SynthScript commands. The SHARC itself contained a SynthScript interpreter, and ran the note filters and signal processing network.

However, this design was not suitable for commercial products based on embedded processors, such as the SHARC or Blackfin, for a number of reasons. Embedded applications have different requirements than PC-based applications. First and foremost, MIPS and memory usage translate directly to cost which is carefully controlled down to the penny. Thus, external RAM is avoided when possible leaving only internal memory. MIPS are also used to the limit of the chip and must always be figured for worst case, rather than average case.

In late 2002, Paul Beckmann proposed a new tool, VisualAudio, which addresses these issues in a variety of ways, including the following: VisualAudio moves memory allocation and parameter initialization from the DSP to the PC (or optional host processor) wherever possible. It provides an automatic routing and allocation algorithm that heavily reuses audio buffers. It supports multiple interrupt levels for different block processing sizes (larger sizes at lower interrupt levels). Finally, it supports user control code written in C, instead of a network of event processing modules.

In addition to general purpose audio support, VisualAudio supports complex applications such as AVR (audio-visual receivers) and automotive audio, which contain elaborate I/O beyond what the general-purpose audio model typically requires. For example, in automotive audio, multiple sampling rates are simultaneously received, including encrypted compressed streams. This implies that content decryption, identification of the content type, dynamic instantiation of decoders, decoding and asynchronous sampling rate conversion are required in one or more places in the system.

For musical applications, VisualAudio is especially well-suited to hardware designs that are intended to be portable, low cost, and/or require low power consumption or heat dissipation. Typical applications include dedicated effects units ("stomp boxes"), reverberators, synthesizers, performance controllers, etc.

VisualAudio addresses each phase of the typical product development cycle, from design, through initial tuning, porting to target hardware, final tuning, and automated testing. Note that there is a fair degree of parallelism in this process. One engineer may be developing the processing algorithms on an evaluation board, while another is developing the hardware, then the algorithms are ported to the target hardware. VisualAudio takes this work flow into account, with the goal of reducing product development time, scheduling risk and project cost for audio applications on embedded processors.

[3] API = "Application programmer interface," i.e. a set of function calls that provide a programming interface to SynthCore.

[4] DLS2 = "Down-loadable sounds," a format for wavetable synthesizer data

3. THE VISUALAUDIO DSP ARCHITECTURE

The VisualAudio DSP application is a combination of libraries, auto-generated source files, and manually-created source files. The software can be factored into a number of separate functions. The audio post-processing "layout" contains the real-time audio processing functions and all required parameters and state variables. The data portion of the layout is generated based upon a graphical block diagram designed by the user while the layout code consists of pre-written processing functions. The "layout support" library is a thin layer that allows post-processing to run and be tuned, and supports "presets," which are collections of module parameters. The "framework" is a platform independent layer that encapsulates audio functions such as buffering, stream detection, instantiating and executing audio decoders. The "platform" contains hardware specific functions such as processor initialization and audio I/O. Finally, the "user control code" contains the product's internal control logic.

The first step in using VisualAudio is to select a "platform file". This XML file describes the capabilities of the target hardware to the VisualAudio tool. It lists the libraries to link against, the sources to copy into the new project, and characteristics of the target hardware such as processor type and clock speed, and the number and types of audio inputs and outputs. After selecting a platform, the user enables decoders and audio inputs and outputs, and this in turn is used to derive the memory requirements of the system.

Once these preliminaries are completed, the principal VisualAudio window appears as shown in Figure 1. To its left is the "module palette", a tree-structured browser of audio modules. This is derived by searching a set of user-settable file paths and finding all XML files that specify the target processor as supported. To its right is the "layout design window". The user designs his post-processing by dragging audio modules from the module palette, dropping them onto the layout design window, and connecting them together. Audio module parameters can be set using "inspectors". Once all modules have been connected and configured, the user pushes the "generate code" button. This runs the routing algorithm and writes out a source file that represents the post-processing network. The user can then edit the control code to set module parameters and apply presets in response to notifications from the framework or decoders, or in response to host messages. For example, if two presets in flash each represent filter coefficients for a different sampling rate, then when a sampling rate change notification is received, the user control code would apply the appropriate preset. The user control code can also be used to periodically poll the hardware, ramp coefficients, etc.

Finally, the user presses the "build" button, at which time the application is compiled/assembled and linked and down-loaded to the target hardware. If any presets are specified, these must be loaded to flash at this time. Finally, the program is started and the BTC channel (or other method) is used to communicate tuning information. If the user likes the adjustments to the parameters, he can save one or more sets as presets to be loaded to flash the next time the program is built. Tuning information is saved in a simple text format, allowing for easy editing, comparing, etc.

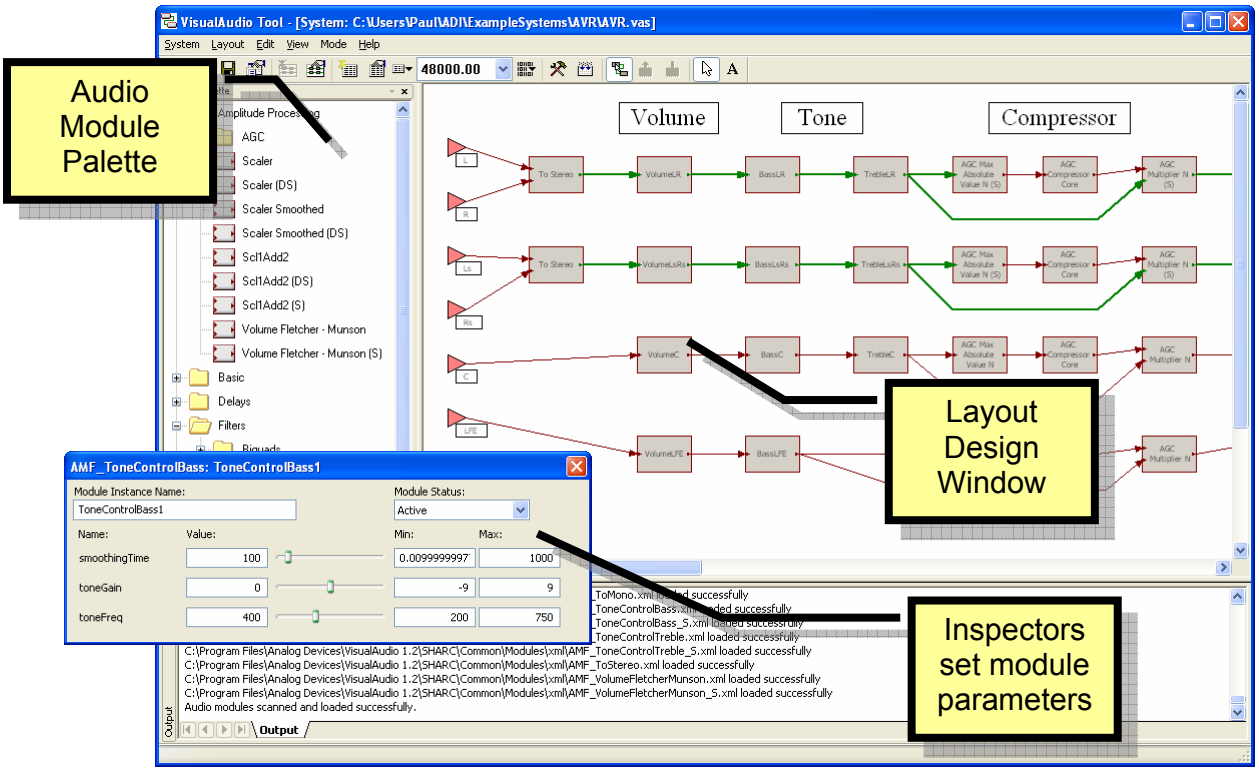

Figure 1. The principal VisualAudio window. The Audio Module Palette contains a list of available audio processing modules. Modules are dragged, dropped, and wired together on the Layout Design Window. Each module has an associated inspector that can be used to configure module parameters.

4. THE VISUALAUDIO FRAMEWORK

The VisualAudio framework is a light-weight interrupt driven OS that handles audio I/O, communications with a host micro-controller (if any), audio processing and user control code. VisualAudio provides two variants of this framework: a general purpose/AVR ("GP/AVR") framework and an automotive framework.

4.1. General Purpose/AVR Framework

The general purpose framework is shown in Figure 2 and assumes a single input stream that may be switched between different modes, such as digital (e.g. S/PDIF) and multi-channel analog. It runs the decoder, post-processing and output at the same rate as the input. The stream detection (if the digital input is active) selects and instantiates the appropriate decoder. When enough data has come in, a decoder is triggered at a lower interrupt level. The decoder determines its own block size. The only requirement is that the post-processing buffer size evenly divide the decoder block size. Meanwhile data continues to flow through the system via a double-buffering scheme. The post-processing is run immediately after the decoding. If the post-processing buffer size is smaller than the decoder buffer size, the post-processing is run several times in a row. At user level (non-interrupt level), any host commands are processed, and any control code is executed. Note that since the decoding and post-processing are in the same thread of execution, they can share scratch memory, another memory optimization. Post-processing is done in-place on the decoder output buffer. Intermediate buffers in the post-processing are automatically assigned using a routing algorithm (described below), resulting in a high level of memory reuse.

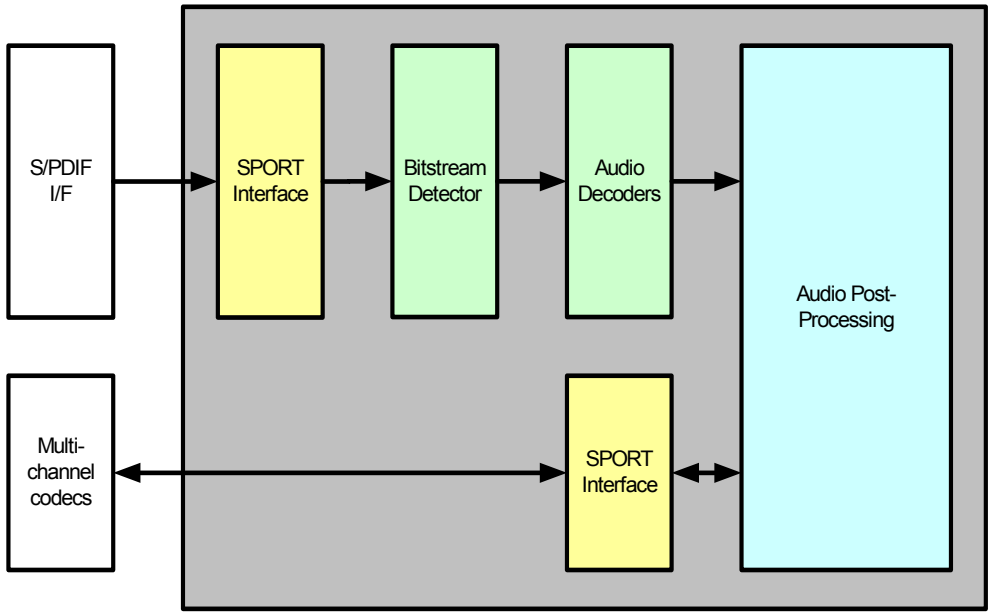

Figure 2. Audio flow within the VisualAudio general purpose framework.

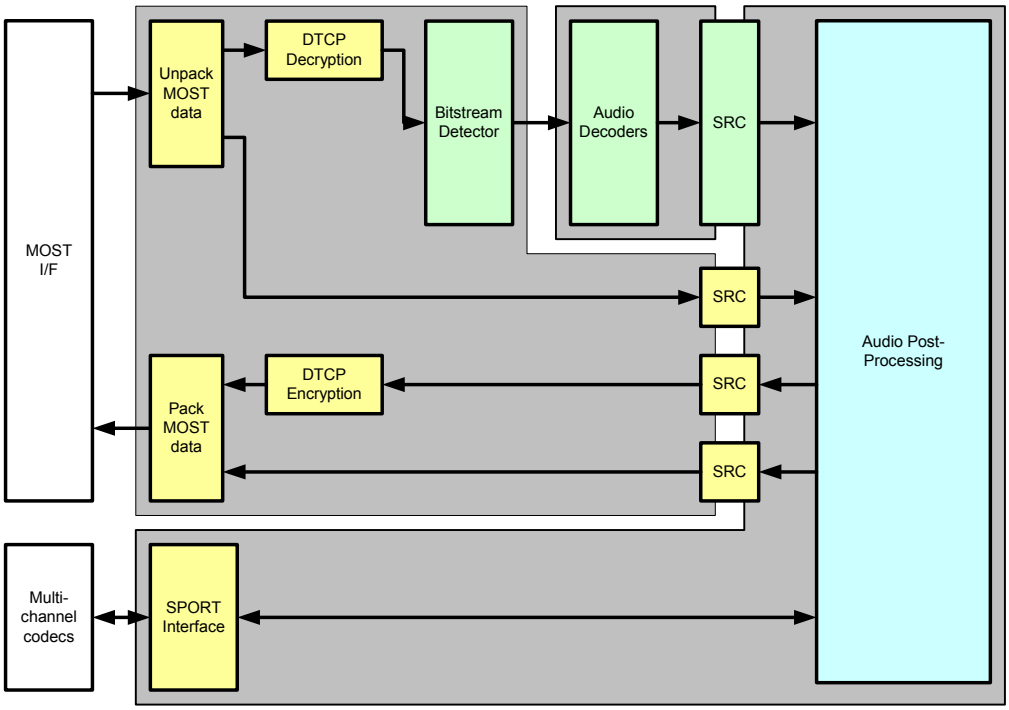

Figure 3. Audio flow within the VisualAudio automotive framework.

4.2. Automotive Framework with Asynchronous Sample Rate Conversion

The automotive framework is designed especially for high-end digital automotive amplifiers and is shown in Figure 3. It differs from the GP/AVR framework in that there are multiple simultaneous input streams which may be running at different sampling rates. There may be several PCM digital streams from sources such as a MOST[5] bus (the network of choice in automotive applications), as well as several analog streams, and an encoded content stream (via MOST or S/PDIF.) The content stream may be encrypted with DTCP[6] , a new encrypting scheme for copyrighted entertainment material sent over digital buses. The outputs may include both analog channels and digital channels via MOST. DTCP encryption may be required. To accommodate this heterogeneous set of inputs and outputs, the post-processing is run at a fixed sampling rate and any inputs or outputs not synched to that rate are converted using an asynchronous sampling rate converter with an integrated jitter buffer. An asynchronous sample rate converter has been developed that is extremely efficient for multi-channel data (the more channels, the more efficient the per-channel cost). A servo algorithm makes subtle adjustments to the sampling rate ratio to prevent underrun or overrun. Since the block size of the decoding can vary, the post-processing runs at a smaller block size and at a higher interrupt level than the decoding. [REF]

4.3. Multiple Block Sizes without MIPS Spikes

If large block processing is required, such as large FFT-based processing, this may be done (in either the automotive or GP/AVR framework) at yet another interrupt level and double-buffered to/from the post-processing. The interrupt level depends on the block size: the smaller the block size, the higher the interrupt level. This ensures that all MIPS are used and there are no bursts. Note that it is not adequate to compute the FFTs in the post-processing thread, and buffer up the output, as this would result in bursts of MIPS and would require the whole system to leave room for these bursts, thus wasting the processor much of the time.[7]

4.4. Bit Stream Detection Issues

The audio system may have a digital input, such as S/PDIF or MOST, for receiving encoded digital content. The content may arrive in a number of formats and it is the job of the bit stream detector to identify the format of the content. The implementation and fine-tuning of the bit stream detector was tricky, for several reasons. There is ambiguity in the IEC[8] specifications for S/PDIF streams. In addition, the detector must support various non-IEC streams, such as DTS-CD[9] , which appears as an IEC PCM stream. Finally, many different compressed stream types must be handled without producing artifacts resulting from switching or misidentification. The problem is exacerbated by variations in DVD and CD players, and various out-of-spec short-cuts in low-end players.

VisualAudio provides a robust heuristic algorithm that does very well in preventing unwanted artifacts. However, since there is no single ideal solution, it also includes a set of parameters that give the customer the ability to select his own trade-offs.

4.5. Memory Allocation

To avoid fragmentation and speed up memory allocation, a standard general-purpose heap memory allocation, such as the C library malloc(), is not

[5] MOST stands for "Media Oriented Systems Transport", a multimedia fiber-optic (low overhead, low cost) point-to-point network implemented in a ring, star or daisy-chain topology over Plastic optical fibers. The MOST bus specifications define the Physical Layer as well as the Application Layer, Network Layer, and Medium Access Control.

[6] DTCP stands for "Digital Transfer Content Protocol Specification," managed by Digital Transfer Licensing Administrator (DTLA). It provides a cryptographic protocol intended to protecting copyrighted material from unauthorized copying, tampering, etc. Despite the fact that an automobile is seemingly a closed system, DTCP is legally required, perhaps forseeing a time where cars use wireless technology to communicate with a home network on the Internet.

[7] Currently the VisualAudio tool supports only two block sizes: that of the decoder and that of the post-processing. However, there is nothing in the architecture that prohibits this generalization.

[8] IEC stands for "International Electrotechnical Commission," a standards organization. The S/PDIF specification is document IEC 61937.

[9] Also called "unformatted DTS," DTS-CD does not follow the IEC specification. Thus, heuristics are required to distinguish it from true PCM.

required. Instead, memory is allocated statically by the VisualAudio tool wherever possible. When dynamic memory is required, it is allocated in a fixed sequence from a stack. This ensures no fragmentation. Post-processing state is allocated first, then post-processing scratch, then decoder scratch (which can overlay with post-processing scratch in the GP/AVR framework), then decoder state, and finally user memory. User memory is automatically freed when a decoder is destroyed by the bit stream detector. Synchronization of memory allocation and freeing is done via notification messages to the user control code.

Since different decoders require different amounts of memory, it is difficult to have a single post-processing network that will support all decoders. In particular, DTS 24/96, which produces data at 96 Khz., uses much more memory than Dolby Digital (AC3) and DTS 48. Thus, it is often desirable to have a smaller post-processing layout for DTS 24/96, and a larger one for Dolby Digital and DTS 48. This can be implemented using VisualDSP++'s "overlay" mechanism. This allows multiple data/code sets to share the same "run" memory space, while each has its own "live" memory space in flash or external memory. The code and data for the new layout are DMA'ed as needed when the stream is detected, into the same memory as the previous layout. VisualDSP++ supports the ability for the same piece of code or data to have two different addresses, one where the code or data lives and the other where it runs. This enables the linker to properly fix up addresses and the loader to properly place the code.

Finally, it is worth mentioning that the audio I/O DMA size is typically smaller than the decoder block size. This is because the DMA is double-buffered in both directions. Thus, by using a smaller DMA size, less overall memory is required. In addition, in the automotive framework, a smaller DMA size allows for lower latency for non-encoded data streams such as announcements from collision avoidance systems.

4.6. Host Protocol and Notifications

VisualAudio supports a simple extensible host protocol for communication between a host micro-controller and the DSP. Each message begins with a 32-bit header word containing an op-code and a word count. This is followed by the data payload. The message ends with a 32-bit checksum word.

The protocol is always initiated by the host micro-controller. The message is first sent to the framework. If the framework does not recognize the op-code, the message is forwarded to the user control code with the high bit set. The receiver of the message constructs a reply, which is then forwarded to the host micro-controller.

The method of transmitting the messages and replies is up to the platform-specific implementation. Typically SPI[10] is used, but other methods, such as flag pins, are possible as well. Note that host communication can occur in parallel with tuning via the BTC.

In addition to host messages, the framework and decoders generate asynchronous notifications. These are distinguished from host messages by the high bit being zero. After the user control code handles the notification, it can optionally forward it to the host micro-processor. In this case, the notification gets put in a queue of messages for the host micro-controller. When the queue goes from empty to non-empty, a platform-specific function is called to raise an interrupt on the host micro-controller. If no such interrupt exists, the host micro-controller should poll periodically for notifications.

As an example, user control code might respond to notifications by enabling Dolby Pro Logic II when the input is stereo and disabling it when the input is 5.1.

5. AUDIO MODULES

VisualAudio provides a library of audio modules sufficient to develop most products. Audio modules operate on blocks of audio data rather than on individual samples. This design choice is a good match for products containing audio decoders which output blocks of audio data. Also, by operating on blocks of data, the overhead of a function call can be amortized over several samples. This yields efficient execution while supporting a modular software architecture.

VisualAudio provides a library of audio modules for a variety of common functions ranging from primitives such as scalers, multipliers and delays, to moderately complex modules such as cascaded biquad filters, to modules with 3rd party intellectual property, such as Dolby Pro Logic IIx and DTS Neo:6. Source code is

[10] SPI stands for "Serial Peripheral Interface" a standard serial bus.

provided to many of the modules and these can serve as templates for custom modules.

Each audio module consists of three elements: (1) an XML file that describes the audio module to the PC application, (2) a real-time audio processing function, and (3) a header file that describes the module's data structure and render function to the C compiler.

The XML file contains memory allocation rules, a list of audio inputs and outputs, and lists high-level and low-level parameters. High-level parameters are to be set by the user directly. The XML file also contains a C-like expression language for converting the high-level parameters to low-level parameters that reside only on the DSP. For example, an oscillator's frequency may be specified in Hertz using a high-level parameter, but would be mapped to a low-level parameter with units of radians/sample by the expression language. This mechanism is adequate for (e.g.) simple filter design techniques, but more involved design algorithms are best handled in an external application, such as MATLAB, communicating via the COM interface.

The audio module's real-time processing function – called the render function – follows a C calling convention for ease of development and integration. Most of the audio modules supplied with VisualAudio are written in hand-coded assembly language, though they can be written in C if desired. Each module's render function is called with three arguments: a pointer to the module's data structure, an array of pointers to input, output and scratch buffers, and the block size. An example C prototype for a render function appears as follows:

```
void Biquad_Render(AMF_Biquad *instance,
                   float **buffers,
                   int tickSize);
```

Each module stores a pointer to its render function; hence the render function itself can swap-in an alternative function to be run the next time the module executes. Any module can be muted, bypassed or inactivated simply by modifying its render function pointer.

Audio data is presented by 32-bit data types on both the SHARC and Blackfin. The SHARC utilizes single precision floating-point while the Blackfin utilizes a fractional representation. VisualAudio supports both mono and stereo audio connections. Stereo streams are represented by interleaved data and facilitate use of SIMD. Audio modules have any number of input and output "pins," any of which may be mono or stereo. The module's I/O is specified in the module XML file.

Each module class is represented by a single C structure typedef, where the fields of this structure represent the low-level parameters of the module. The module format also supports indirect arrays whose size is determined by an expression. For example, a delay buffer size could be specified in seconds or the size of a filter coefficient array could depend on the filter order.

VisualAudio provides inspectors for configuring audio module parameters as shown in Figure 1. The inspector contains sliders, fields, or other controls for the high-level variables specified in the module XML file. It also supports read-back variables that display the state on the DSP. VisualAudio is able to create this inspector based solely on the information in the XML file. Alternatively, the XML file can specify a DLL to load to provide a custom inspector and communicate with VisualAudio via a standard API.

The audio module library was designed to provide significant overlap between the SHARC and Blackfin processors. When possible, the same high-level API is provided on both processors. Thus, it is possible to open a SHARC processing layout, change to a Blackfin platform, and run the layout on the Blackfin.

The library modules are highly-optimized assembly language and examples of inner loops are shown in Figure 4. The examples are from modules that implement a cascade of biquad filters that operate on interleaved stereo data.

```
  lcntr=r1, do filtering until lce;
    f12=f2*f4,  f8=f8+f12,   f10=dm(i3,m4);
    f12=f3*f5,  f10=f10+f12, dm(i4,m4)=f8;
    f12=f2*f6,  f8=f10+f12,  f2=f3;
  filtering:
    f12=f3*f7,  f8=f8+f12,   f3=f8;
```

SHARC

```
  LSETUP (sampleLoopStart , sampleLoopEnd)
  LC0=P3;
  sampleLoopStart:
    A0 += R0.H*R4.H, A1 += R0.H*R4.L (M) || R1=[I1]      || NOP;
    A1 += R4.H*R0.L (M);
    A0 += R1.H*R6.H, A1 += R1.H*R6.L (M) || [I1]=R5      || NOP;
    A1 += R6.H*R1.L (M) || R1=[I0++M2] || NOP;
    A0 += R1.H*R7.H, A1 += R1.H*R7.L (M) || R0=[I0]      || NOP;
                     A1 += R7.H*R1.L (M);
    A0 += R0.H*R3.H, A1 += R0.H*R3.L (M) || R5=[I2++M2] || NOP;
                     A1 += R3.H*R0.L (M);
    A1 = A1 >>> 15;
    A0 += A1;
    R0 = A0 (ISS2);
    A0  = R5.H*R2.H, A1  = R5.H*R2.L (M) || [I0]=R0      || NOP;
sampleLoopEnd:
    A1 += R2.H*R5.L (M) || R0=[I1++M2] || [I3++M2]=R0;
```

Blackfin

Figure 4. Optimized inner loops for stereo biquad filters on the SHARC (top) and Blackfin (bottom). Each loop operates on stereo interleaved data and iterates over a block of audio samples. An outer loop (not shown) iterates over multiple stages of biquads. On the SHARC, SIMD execution is enabled, and the arithmetic occurs twice – once each in the primary and secondary register sets. Each biquad has 4 coefficients and implements a Direct Form II structure. A total of 8 floating-point multiply-accumulates occur in 4 clock cycles. On the Blackfin, each biquad has 5 coefficients and implements a Direct Form I structure. A total of 10 32-bit double precision multiply-accumulates require 13 cycles.

6. AUTOMATIC ALGORITHM FOR DETERMINING ROUTING, RUN-TIME ORDER AND MEMORY ALLOCATION

VisualAudio supports arbitrary graphs for post-processing. The only restriction is that only one graphical connection (called a "wire") may connect to each input pin of a module. Output pins may fan out to multiple modules. Feedback is allowed. This allows reverberators and physical models to be easily created. A module may even feedback to itself.

To allocate the modules and determine their run order, an automated routing algorithm was devised. This algorithm has several stages. First, it checks for circularities. It does this by traversing the network recursively, beginning with each audio input pin (that is, the outputs of the decoder or the direct analog or PCM input), and then beginning again with each module that was not reached in the first search (because it has no connected inputs or no inputs at all). Any detected circularity results in a static buffer allocation, as this buffer is needed to break the circularity and represents a buffer-length delay.

The next step is to traverse the network again, this time doing a topological sort [16] to determine run order. This leads to a run order with no hidden delays, except where a circularity was detected. As each module is visited, input and output buffers are allocated from a scratch pool. As soon as a buffer's contents is no longer needed, it is returned to the scratch pool. This typically results in a large degree of buffer reuse. Note that some modules are coded (and specified in the XML) such that inputs and outputs may share the same memory buffers,

while others require that their inputs and outputs be distinct from one another. These restrictions are taken into account by the routing and allocation algorithm. In addition, modules can use arbitrary amounts of scratch memory, which is allocated from the same pool as the inter-connection buffers. Finally, all of this memory is available as scratch to the decoders in the GP/AVR framework, where decoding and post-processing occur in the same thread. Thus, only a small amount of memory is needed to run the post-processing network above and beyond that required by the audio module parameters and state variables themselves.

Inter-module connection buffer pointers are not stored directly, but as indices into an array of pointers. This allows double buffering of the post-processing buffer itself and reuse of this pair of buffers as inter-module connection buffers. Before a module is called by the framework, its indices are resolved to pointers, and these are passed to the module render code. In applications where the double-buffering is not needed, the indices are resolved at initialization time, thus saving the MIPS needed to continually resolve them on each render call. (This optimization is possible only if the post-processing buffer size is the same as the I/O DMA size.)

Note that since the post-processing is done in-place, it is possible for the routing algorithm to reach a point where it needs to write into an output buffer that is unavailable, as its contents haven't been fully consumed yet. In this case, the routing algorithm allocates a temporary buffer and inserts a copy module to copy the data to the output buffer. This copy is inserted at the end of the list of modules so that by the time it runs, the output buffer is available.

7. TUNING

Tuning of audio modules can be done in real-time using the VisualAudio tool's inspectors or via the external COM interface. The tuning is implemented with a subset of the host protocol that allows individual variables and arrays (including sparse arrays) to be set or read. The tuning commands are executed on the DSP at non-interrupt level, while audio executes at interrupt level. This allows audio to proceed uninterrupted, regardless of the bandwidth of the tuning communication. However, there are times when a set of coefficients must be applied as a unit, with no intervening audio processing, to avoid an undesirable effect, such as a filter going unstable. For this reason, an "atomic" setting command is provided, which masks interrupts while the array transfer occurs. Similarly, an atomic read guarantees that all read values in a packet come from the same instant in time.

8. AUTOMATED TESTING

The COM interface is useful for implementing automated test scripts. In addition to its support for reading and writing parameters, the COM interface also provides a "demand render" mode. Here, the audio processing is halted and input buffers are written to the DSP from the external application (typically, MATLAB). The audio processing is then triggered and the resulting data is read back. This allows regression testing to be completely automated.

9. CONCLUSION

VisualAudio has proven to be a viable tool for speeding up deployment of audio products on Analog Devices SHARC processors, and is beginning to be used for Blackfin processors. It was designed from the beginning with these applications in mind and this focus guided the design trade-offs. In addition the VisualAudio tool, framework, modules and decoders were developed in parallel, and are thus well integrated with one another, while also maintaining sufficient independence to allow them to be easily used separately. For information about obtaining VisualAudio, visit www.analog.com and search for "VisualAudio."

10. ACKNOWLEDGEMENTS

This work was supported by Analog Devices, Inc. Thanks to everyone who helped with the VisualAudio project, in no particular order: Thanks to key software developers, Eric Liang, Steve Goeckler, Vathsa Duglapura, Min Guo; to algorithm developers, Sean Costello and Joe Anderson; to quality engineers, Alicia Nachman, Chris Owens, Sachin Nanda, Liza Khodel, Kishore Dasari, Madan Mohan, Kranti Kumar, Tarun Kumar; to audio decoder development team, Laxminarayana Parayitam, Padmavathi Ramanathan, Subrahmanyam KVVD, Sailaja Mahendrakar; to the program management and engineering support team, Mauricio Greene, Fernando Martinez, and Atom Ellis; to Melissa Fahey for excellent documentation, graphics and developer support; to Matt Walsh and Dan Ledger for automotive platform integration and design engineering; to Brad Lee, Adam Shelly and Alan

Gerrard, who integrated the SoundMAX wavetable synthesizer into VisualAudio; to Vincent Fung, Jason Herskowitz, Lori Berensen, Tom Cote on the product side, and Dennis Labrecque marketing guy par excellence; to Dan Harn for legal support; and to Debbie Davis, who kept us organized. Thanks to Colin Duggan, who steadfastly promoted the business justification for VisualAudio; to Catherine Stevens who kept our team resourced and motivated, and to John Croteau who championed the acquisition of Staccato Systems and formed ARTC at Analog Devices.

11. REFERENCES

[1] Loy, G. D. 1981. "Notes on the implementation of MUSBOX: A compiler for the Systems Concepts digital synthesizer." *Computer Music Journal 5, 1, 34-50*.

[2] Jaffe, D. and L. Boynton. "An Overview of the Sound and Music Kits for the NeXT Computer", *Computer Music Journal, 14(2):48-55*. Reprinted in The Well-Tempered Object, ed. Stephen Pope, 1991, MIT Press.

[3] J. O. Smith, D. Jaffe and L. Boynton. "Music System Architecture on the NeXT Computer", *Proceedings 1989 Audio Engineering Society Conference*, L.A., CA.

[4] Jaffe, D. "Musical and Extramusical Applications of the NeXT Music Kit", *Proceedings 1991 International Computer Music Conference*, pp. 521-524, Montreal, Canada.

[5] Jaffe, D. "Efficient Dynamic Resource Management on Multiple DSPs, as Implemented in the NeXT Music Kit", *Proceedings 1990 International Computer Music Conference*, pp. 188-190, Eastman, NY.

[6] Jaffe, D. and J. O. Smith. "Real Time Sound Processing and Synthesis on Multiple DSPs Using the Music Kit and the Ariel QuintProcessor." *Proceedings 1993 International Computer Music Conference,* Tokyo, Japan.

[7] Jaffe, D., J. O. Smith, N. Porcaro. "The Music Kit on a PC", *Proceedings of the first Brazilian Symposium on Computation and Music, XIV Congress of the Brazilian Society of Computation,* 1995, Caxambu, MG. pg. 63-69.

[8] Poracro, N., W. Putnam, P. Scandalis, D. Jaffe, J. Smith, T. Stilson, and S. Van Duyne, "SynthBuilder and Frankenstein, Tools for the Creation of Musical Physical Models." *International Conference on Auditory Display,* 1996, Palo Alto, Santa Fe Institute and Xerox Parc.

[9] Minnick, M. "A Graphical Editor for Building Unit Generator Patches", *Proceedings 1990 International Computer Music Conference.*

[10] Porcaro, N., D. Jaffe, P. Scandalis, J. Smith, and T. Stilson, "SynthBuilder – a Graphical Rapid-Prototyping Tool for the Development of Musical Synthesis and Effects Patches on Multiple Platforms," *Computer Music Journal, MIT Press,* 22(2):35-43

[11] Brandon, S. and L. Smith, "Next Steps from NeXTSTEP: Music Kit and Sound Kit in a New World," *Proceedings 2000 International Computer Music Conference.*

[12] Staccato Systems, Inc. demo presentation at International Computer Music Conference, Thesoliniki, Greece. Further info at: http://www.http://www.elecdesign.com/Globals/PlanetEE/Content/4386.html

[13] SoundMAX web site at the following URL: http://www.soundmax.com

[14] Jaffe, D. "Racing Against Time," for two violins, two saxophones, piano and radio drum. Commissioned by "Quarks!". Published by *Terra Non Firma Press.* Released on DVD with *Proceedings 2004 International Computer Music Conference*, Miami. For further info, visit: http://www.jaffe.com

[15] Boie, R. A., L.W. Ruedisueli and E.R. Wagner "Gesture Sensing via Capacitive Moments" Work Project No. 311401-(2099,2399) AT&T Bell Laboratories, 1989. Also, Mathews, M. V. and W. A. Schloss "The Radio Drum as a Synthesizer Controller." *Proceedings 1989 International Computer Music Conference*, Columbus Ohio.

[16] Cormen, T., Charles.Leiserson, Ronald Rivest. *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990, pgs. 485-488. Editorial Acme, Barcelona, 2005.