Electronic Design

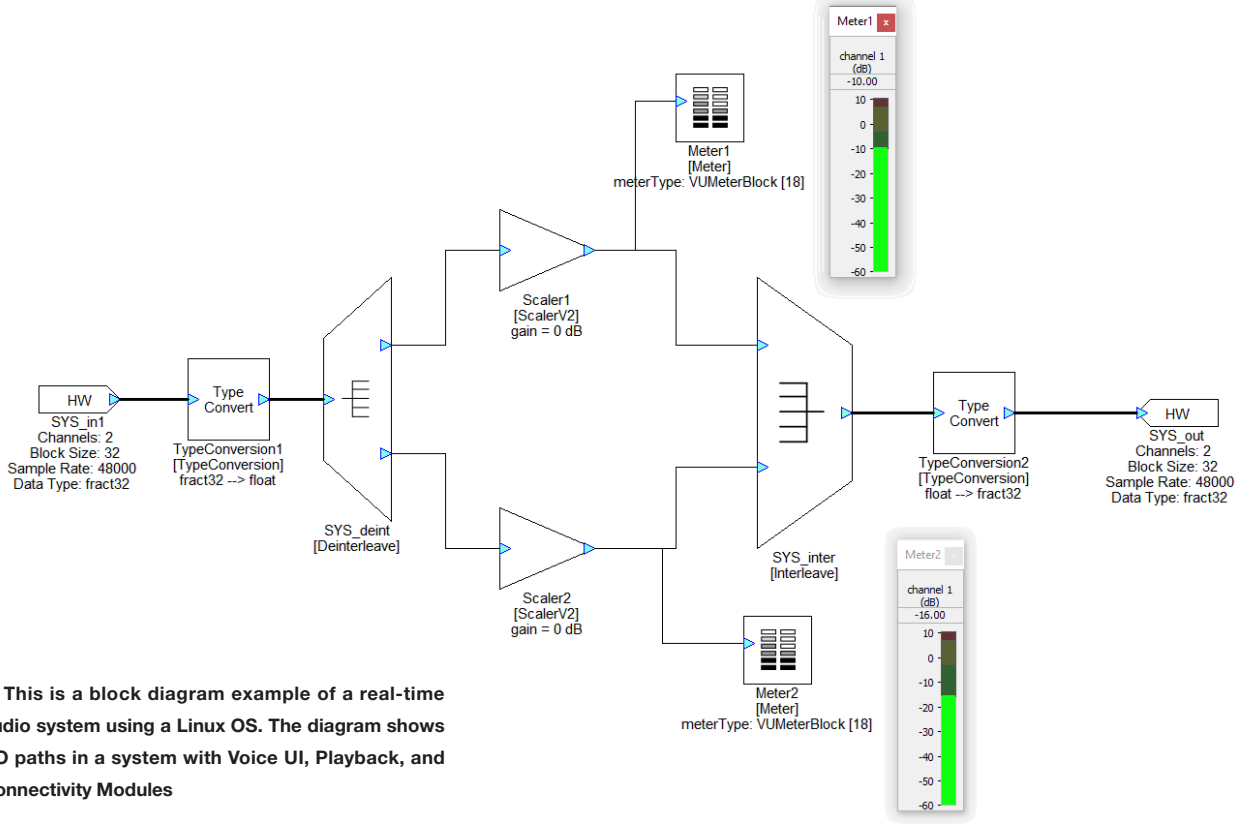
By CHIN BECKMANN, CEO, DSP Concepts,
http://www.dspconcepts.com

5 Development Platform Keys for Easy Real-Time Audio System Design

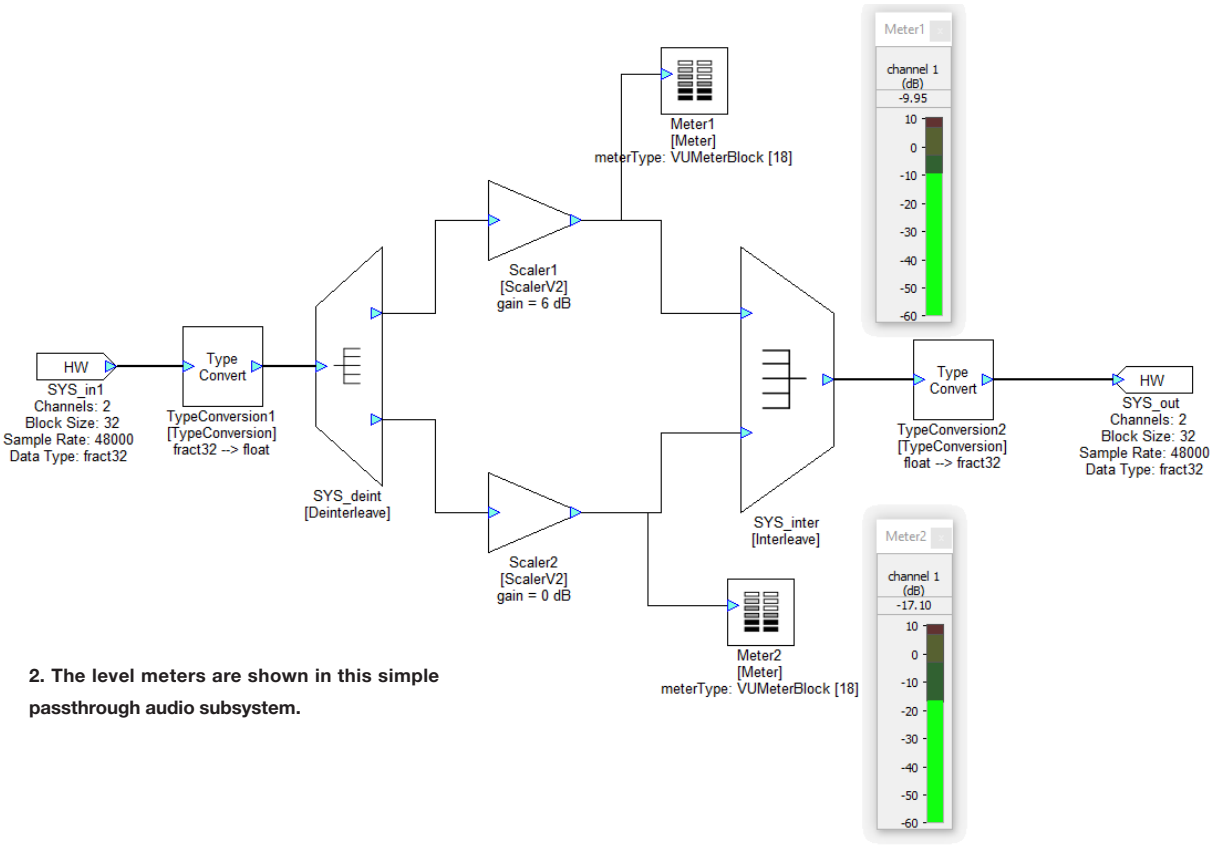
This article explores exactly what features to look for when selecting a development tool for building real-time embedded audio systems.

Audio systems in consumer electronics and automotive infotainment systems have become increasingly complex because of consumers' rising demand for premium audio experiences. From perfectly tuned sound to flawless voice recognition, consumer expectations have forced brands to develop new product-differentiating audio features at a pace and scale we've never seen before. And that includes real-time audio subsystem design, where all of the audio processing must be completed within a fixed time period to avoid drop-outs and audio packet loss.

Developing real-time embedded audio systems involves a combination of software development and porting onto one or more hardware components, such as a system-on-a-chip



1. This is a block diagram example of a real-time audio system using a Linux OS. The diagram shows I/O paths in a system with Voice UI, Playback, and Connectivity Modules



2. The level meters are shown in this simple passthrough audio subsystem.

(SoC), microcontroller, and specialized audio chips. It's a complex process that requires an audio development platform with several mission-critical features. Here, we'll explore exactly what features to look for when selecting a development tool for building real-time embedded audio systems.

1. A Flexible, Cross-Platform Design Environment

Audio development is a multidisciplinary process that involves software engineering, algorithm building, digital signal processing, and hardware engineering. Therefore, it's important to select a flexible development platform that allows you to separate hardware design from software development in the early prototyping phase, and then re-integrate them seamlessly in the final design phase with minimal changes to the code and optimal resource efficiency.

For example, designers should be able to develop real-time audio algorithms on a PC and seamlessly switch to evaluation hardware before switching to the end-target hardware. This cross-platform approach enables developers to independently prototype multiple features and designs on a PC before committing to a specific design or hardware component for the end product.

A traditional code-based approach to audio development is tedious and slow, which is why real-time system development must be flexible enough to develop, measure, tune, and optimize new features without having to constantly rewrite low-level code. Flexibility is also key for improving both productivity and scalability of new audio features across different platforms or product lines.

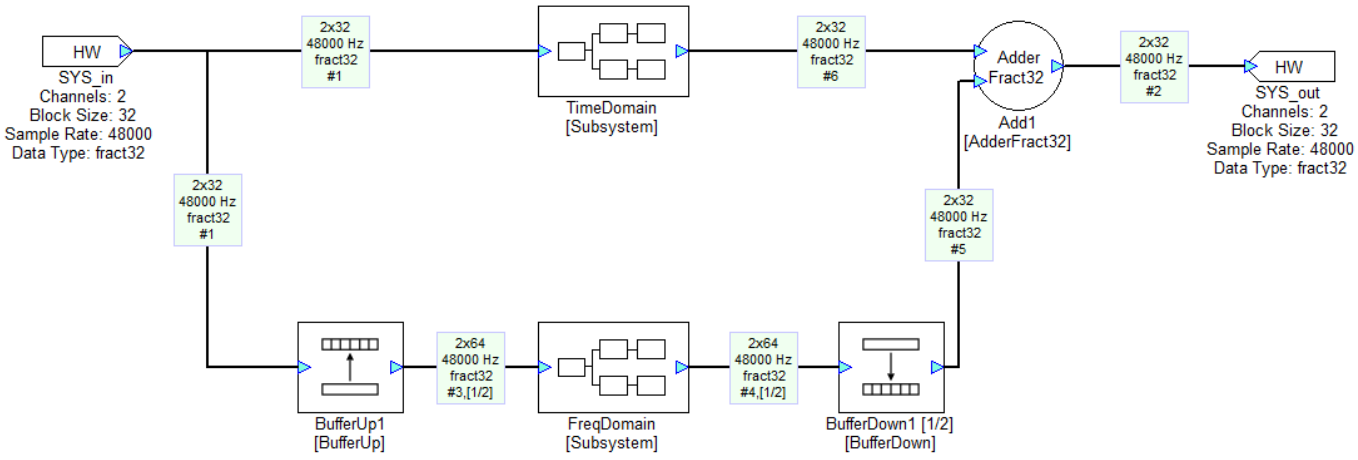
A great example of a flexible development platform is one that offers a graphical user interface (GUI) with built-in audio modules to enable rapid prototyping and facilitate testing new features, because you don't have to rewrite hundreds of lines of code for each module. Every audio product will have checklist features that the developers could use across multiple product lines. Thus, developers should be able to reuse these features where necessary, which unlocks more time to spend on creating product-differentiating features.

This level of flexibility combined with a cross-platform development environment, allows product makers to test multiple features and designs through a GUI on a PC before porting it to any hardware. It also allows multiple teams to collaborate efficiently, which is critical given the multidisciplinary nature of audio development. For instance, you can have one person designing an equalizer component while another team member works on speaker playback processing, and everything will come together seamlessly.

2. Supports Efficient I/O Signal-Routing Management

The very first step in audio-system development is configuring the signal pipeline so that the incoming audio stream is routed correctly to the output stage. Before implementing the system's core audio-processing blocks, you should develop an audio framework that can efficiently handle audio input/output (I/O) and signal routing through the entire audio signal chain within the constraints of the board-support package (BSP) code provided by the hardware vendor.

Such a framework should define the real-time I/O



3. This is an example of a real-time audio system with multi-rate scheduling priorities.

initialization, memory allocation and instantiation, tuning interface for debugging, and transport protocol to communicate with the host interface. This framework is critical to ensure a high-fidelity audio system, and it can vary from platform to platform.

An example of different components in the framework is provided for a Linux-based system with DSP Concepts' Audio Weaver platform as the development environment for a system with TalkTo Voice UI and playback modules *(Fig. 1)*. The diagram shows the I/O routing for the different design blocks in the system. Note that the Linux system shown in the block diagram uses Advanced Linux Sound Architecture (ALSA), whereas an embedded target might utilize the hardware's audio I/O digital memory access (DMA).

The audio framework makes it easy to verify the audio signal path for appropriate sample rates, block sizes, signal routing, etc., before building complex audio modules to be integrated into the framework. The resulting audio system should be inspected for signal-routing accuracy by testing it at the passthrough stage. This passthrough test could just use a simple sine wave as a test signal to identify any distortions or dropouts in the system. The audio development platform should provide the necessary tools such as inspectors, level meters, etc., to easily debug the audio framework and identify any routing or BSP integration issues in the early stages of development.

An example screenshot of a simple passthrough system during run-time is shown in *Figure 2*. Assuming that all the I/O parameters are set correctly to passthrough a 1-kHz tone at −10 dBFs input level, the level meters on the top and bottom paths are expected to show no gain difference since the scaler on each path is also set to 0 dB. The 6-dB offset between the top and the bottom level meters (Meter1 shows −10 dB and Meter2 shows −16 dB) indicates that there's something wrong in the BSP code or hardware integration setting.

Traditionally, developers debug this with breakpoints, write code to measure output levels, or listen to recorded data at intermediate steps to troubleshoot the issue—all of which contribute significantly to design time and risk of project failures. Instead, a sophisticated development tool should provide the necessary level meters or inspectors to be able to visually identify the issue immediately at the block level *(Fig. 2, again)*. Visually debugging is crucial for complex real-time audio systems where multiple independent paths or cross-functional components can easily complicate the debug process.

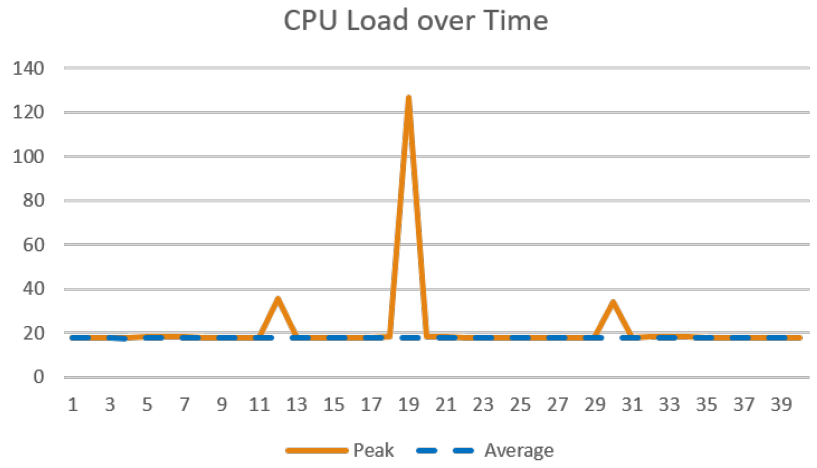
3. Manages Real-Time Audio Priorities

Every audio product has two different functions happening concurrently. The first is real-time audio processing, and the second is the control interfaces working to enable this processing. For example, a hands-free telephony module will be running in real-time on an automotive system while the control interface will simultaneously be updating the tuning parameters of an automatic-gain-control (AGC) block within the module.

Those two functions are very different, with real-time processes needing to run at a sample rate—e.g., 8, 16, or 48 kHz—to avoid any audio pops or clicks while control processes can run at much lower rates, say around 10 Hz to 100 Hz. For this reason, real-time and control functions are able to run on separate threads, so long as your audio development platform can automatically prioritize tasks.

Similarly, real-time tasks that must run at a faster rate should be prioritized over real-time tasks with a slower run rate. *Figure 3* shows an example of such an audio system. The two sub-layouts run in their own processing path but have different block sizes. The top layout runs a time-domain subsystem at a 32-sample block size while the bottom layout runs a frequency domain subsystem at a 64-sample block size, but at half the rate of time domain subsystem.

BufferUp and BufferDown modules combine the two sub-layouts with different block sizes. Real-time constraint requires that the 64-sample block should also complete processing at the same time as the 32-sample block. But in practice, the 64-sample block needn't complete execution



4. The peak versus average CPU plot shows load over time on an audio system with a Bluetooth application incorrectly set to a higher priority than real-time audio.

until the next 64-sample block data is available. Therefore, the processing time of the 64-sample block should be split between two 32-block processes.

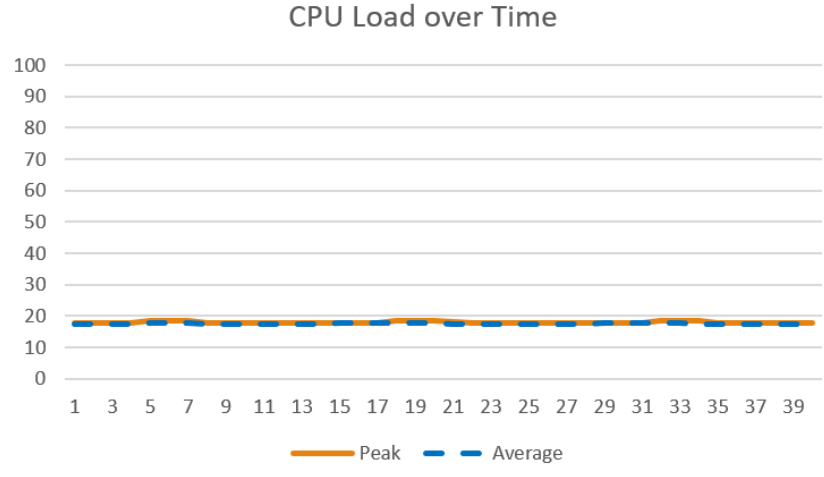
Splitting that time must be achieved by using priority settings in the development environment, essentially setting the sub-block with the shorter block size to have a higher priority than the longer block size. That's why it's imperative your audio development platform can effectively prioritize real-time tasks.

In general, the prerequisites to design a real-time audio product design are:

- Audio threads should have a higher priority and it's recommended that audio processing isn't preempted.
- Latency should be fixed in the audio path; any change in latency will result in poor performance.
- Clocks for audio blocks should be synchronized; any drift or jitter will lead to decreased performance over time.

CPU load over time could provide a good indication of incorrect thread priorities between concurrent tasks. *Figure 4* shows peak CPU load over time of an audio system where the Bluetooth thread priority is set higher than the real-time audio thread. The spike in CPU load resulted in a loss in real-time operation.

When thread priorities are reversed, the average CPU load remains constant over time *(Fig. 5)*.



5. Here, CPU load remains constant over time on a real-time audio system with real-time audio set to a higher priority than the Bluetooth application.

4. Provides Multicore Support

In many audio products, concurrent tasks must run on different processor cores, or multiple cores within the same processor. This requires using a development platform that supports multithreaded software development to support audio I/O, processing, control, or host communication events running on their own individual threads. Each of these threads could be running at its own sample rate and priority level. The available central-processing-unit (CPU) clock speed and memory on the processor, as well as the audio-processing requirements, determine whether to use a single core or multiple cores.

For example, automotive OEMs traditionally rely on a multichip architecture that includes a main SoC for application processing along with an external digital signal processor (DSP) to implement audio-system processing. Historically, SoC designs have lacked the resources required to implement complex audio processing, so they had to be combined with external DSPs.

The problem with relying on external DSPs is that it requires significant development expertise and coding proficiency in the respective architecture. External DSPs also often need additional supporting components like random access memory (RAM), flash memory, or a microcontroller, which makes the development process more complex, time-consuming, and expensive.

Luckily for developers today, modern SoCs such as the Arm Cortex-A and Cortex-M series processors have started to offer significant improvements in processing power to enable embedded development at lower costs. We put this to the test here at DSP Concepts by running some performance assessments, and

the results indicate that even a high-end, complex audio-processing system can fit comfortably on a single-core SoC with Linux OS.

5. Optimizes Performance and Debugs Errors in Real-Time

Audio-system tuning is a significant investment for development teams from both cost and time perspectives. Two key areas of tuning are required to optimize design performance.

The first is performance tuning to meet acoustic benchmark requirements for a particular real-world use case. For example, a smart speaker with Alexa built-in must meet certification requirements set by Amazon for basic or premium voice-activation performance before it ever goes to market. In this instance, developers should use a pre-qualified reference design from a vendor with ample experience in that space to reduce time-to-market. Once they determine what design to use, they need to make sure their development platform provides a real-time tuning interface that lets them quickly view and adjust tuning parameters to cut down on performance tuning resources.

The other element of performance tuning is optimizing the CPU and memory resource usage on the target platform. This can be accomplished most efficiently by using a development platform that has built-in, real-time profiling capabilities to provide profiling information down to the block level. Therefore, the resource can be optimized selectively.

And because of the complex nature of audio systems, it›s critical to use a development platform that has real-time debugging features built-in at the audio module level as well as the overall system level. At the module level, it's critical to simultaneously view outputs of each function using built-in tools such as inspectors and level meters so that the output can be easily verified at each stage. Similarly, it's important to verify that debugging tools are automatically running against both software and hardware. Therefore, you can immediately identify any flaws in CPU clock synchronization, latency, mechanical design, microphone isolation, sensitivity matching, etc., which could all impact real-time audio performance. Building with tools that have robust debugging capabilities is essential to avoid massive production delays and unexpected costs.

Your Dev Platform Can Make or Break Your Product Launch

It might seem daunting to not only look for all of these features in a development platform for real-time audio-system design, but also verify that they function as they should. However, the decision you make with regard to what development platform to use can absolutely make or break your product launch, so it's important to choose wisely. Keep the aforementioned five capabilities in mind and you'll be well on your way to building a successful audio system that delivers an exceptional user experience.

Chin Beckmann co-founded and has led DSP Concepts since its inception in 2003. Her leadership has resulted in Audio Weaver, a platform disrupting audio product development, and she has raised more than $25 million of venture financing. Under her direction, Audio Weaver has continued to solve the most difficult audio processing problems, resulting in design wins among Tier 1 Automotive and Consumer OEMs globally.

Prior to DSP Concepts, Chin held software engineering roles at Bose, Proteon, and Data General. Chin holds a BS degree in Electrical Engineering from Boston University and an MBA from Northeastern University. She is also pianist for the California Pops Orchestra and is fluent in English, Spanish and Chinese.