# A COMPONENT-BASED SOFTWARE ARCHITECTURE FOR REALTIME AUDIO PROCESSING SYSTEMS

Jarmo Hiipakka
*Nokia Research Center, Multimedia Technologies Laboratory,*
*P.O. Box 407, 00045 Nokia Group, Finland*

Keywords: Audio processing, Software architecture, Realtime systems.

Abstract: This paper describes a new software architecture for audio signal processing. The architecture was specifically designed low-latency, low-delay realtime applications in mind. Additionally, the frequently used paradigm of dividing the functionality into components all sharing the same interface, was adopted. The paper presents a systematic approach into structuring the processing inside the components by dividing the functionality into two groups of functions: realtime and control functions. The implementation options are also outlined with short descriptions of two existing implementations of the architecture. An algorithm example highlighting the benefits of the architecture concludes the paper.

## 1 INTRODUCTION

Recent years have significantly increased the role of software in the field of audio processing. While dedicated hardware solutions still dominate certain applications, more and more processing is moving to general purpose processors or programmable digital signal processors (DSPs). Software systems for audio processing have been widely, though not very systematically, described in the literature. Naturally, software implementations in the context of specific algorithms have been presented, but more generic architectures are seldom described in public.

Many of the software frameworks available for audio processing, are based on dividing the processing tasks into components that all share the same interface towards the rest of the system. The component paradigm encourages fewer dependencies between different audio processing algorithms. This enables separating also the development work into more easily manageable entities.

Audio processing software often needs to run in real time, and with low interaction latency and audio delay. This sets strict requirements to the implementation. Additionally, for best performance, the system parameters such as the audio processing block length need to be carefully tuned, finding an optimal compromise between computational efficiency and latency and delay behaviour.

This paper presents an audio processing software architecture that can be used, when the processing platforms efficiently supports multiple simultaneous processing contexts or threads. The system described here is beneficial both when running and audio processing system on a general-purpose operating system and when, e.g., using a dedicated DSP to accelerate the processing. The architecture separates audio delay optimization from the interaction latency optimization, and allows an optimal processing load distribution by separating the realtime and control parts of the processing inside logically integrated components.

The rest of the paper is organized as follows: section 2 presents the motivation and background for component based processing architecture. Section 3 describes how the current architecture has been adapted for realtime usage. Section 4 details a few implementation alternatives, and section 5 describes a use case and an algorithm example that highlights the advantages of the architecture design. Finally, section 6 concludes the paper.

Hiipakka J. (2006).
A COMPONENT-BASED SOFTWARE ARCHITECTURE FOR REALTIME AUDIO PROCESSING SYSTEMS.
In *Proceedings of the International Conference on Signal Processing and Multimedia Applications*, pages 289-294
DOI: 10.5220/0001573002890294
Copyright © SciTePress

## 2 COMPONENT ARCHITECTURE

There are several audio processing software solutions, where at least part of the processing is split into units that conform to a unified interface, irrespective of the nature of the processing. These building blocks for the software system framework are called *components* in this paper. Component is defined as a logically unitary audio processing entity.

### 2.1 Existing Component Designs for Audio Processing

In the desktop computing environments, many audio application vendors have their proprietary mechanisms of including new audio processing algorithms into processing chains as plugins. Examples of such applications include Steinberg's Cubase (www.steinberg.de), Nullsoft's Winamp (www.winamp.com), and Microsoft's Media Player (www.microsoft.com/windows/windowsmedia/). However, some of these plugin interfaces have become popular outside their original applications, and some have originally been designed to be used by multiple different applications. Currently, popular audio plugin formats include VST by Steinberg (2006) and LADSPA (Furse, 2006; Phillips, 2001) by the Linux audio community, among others.

In addition to the audio plugin interfaces, there are more complete component based multimedia frameworks. Examples include the DirectShow media streaming architecture by Microsoft (2006), and the open source GStreamer (2006) framework. Both of these systems include media encoders, decoders, and processing components as plugins that can be connected together for complete multimedia applications.

Component based audio software architectures are also fairly common in more restricted, embedded environments. In the system described by Datta et al. (1999) all post processing audio effects applied to a decoded multi-channel audio stream share a common interface. On a more generic level, the XDAIS algorithm interface specification developed by Texas Instruments (2002) provides a unified way for integrating signal processing algorithms into their specific operating system (OS) environment.

A system that falls between the desktop environment and dedicated systems has been described by Lohan and Defée (2001) for a media terminal architecture. Their system is technically a PC, but developed with a dedicated use case in mind, and equipped with a dedicated user interface.

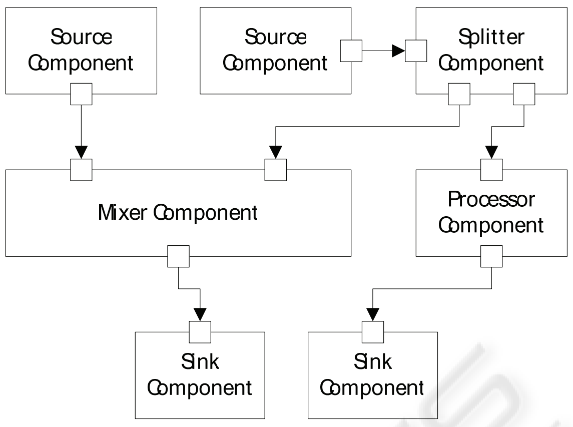

Figure 1: A component system with connections between components' input and output ports.

An interesting emerging standard in the field of component based multimedia is the OpenMAX standard, defined by the Khronos Group (2005). Specifically, the OpenMAX Integration Layer (IL) API provides applications or OS multimedia frameworks a standard interface to commonly used multimedia components such as encoders, decoders, and various effect processing algorithms.

### 2.2 Component Characteristics

A component typically implements an audio processing feature, such as a mixer, a sample rate converter, or an audio effect such as reverberation. One component encapsulates an algorithm that may contain several basic signal processing blocks such as delays and filters. From software design point of view, components are implemented using a single logical building block of the environment, for example using a C++ class linked as a dynamically loadable library.

Components have a number of data inputs and outputs—often called ports—that can be used to connect components together. In audio software systems, any audio data representation format can be used for the connections. Very often a linear PCM data format is used. When all the inputs and outputs have the same representation, it is possible to freely route audio data from one component to another. Figure 1 depicts a component system with several different component types and connections between the components.

Components can be either statically built, i.e., defined and integrated at system development time, or they can be dynamically loadable *plugins* that can be connected to a previously compiled and linked (i.e., executable) software.

## 2.3 Filter Graphs

An audio processing engine can connect components together to form processing networks or *filter graphs*. Within the graph, components are run sequentially in the order defined by the needs of audio processing functionality. It should be noted that there can also be special components that contain several individual components in a hierarchical manner for grouping, e.g., according to the audio sample rate.

When a software package is used as a complete audio mixing and effects engine in a multitasking OS, it is desirable to be able to add new audio applications, streams, and effects while keeping the current ones running. Adding a new audio feature should happen without causing any problems to the currently playing streams. This calls for the ability to modify the filter graph while the system is running. If the graph is modified one component at a time, it may easily become invalid during a transition from one larger configuration to another. Therefore, it is important that related changes to the configuration are all taken into use simultaneously.

Simultaneous update is easy to achieve, when the graph configuration is kept separate from the components themselves. Furthermore, at least two graphs are always stored at a time: one is active (used in processing), the other one can be modified. When all modifications to the graph configuration have been made, the filter graphs are swapped atomically so that the one that was modified will become the active graph. This mechanism is similar to the double buffering scheme used in computer graphics for avoiding flickering effects caused by modifying an image that is currently been drawn on the screen.

Many existing component frameworks distribute the connections between the components to the nodes of the filter graph. This may be beneficial for optimizing the connections, but updating the graph becomes problematic, especially if the changes should happen without drop-outs in the audio signal.

## 2.3 Framework Functionality

An important benefit in a component based architecture is the possibility of delegating common functionality to the framework around the components. This can considerably reduce the effort when implementing individual components. Additionally, the total executable binary size of the audio subsystem is smaller, when functionality is implemented only once.

In the our architecture, the framework takes care of scheduling the signal processing, providing memory buffers for component inputs and outputs, scheduling the control events for processing, and providing processing functions temporary memory buffers that can be shared between components (so called *scratch memory*).

## 3 REALTIME SYSTEM DESIGN

Audio processing is very time-critical by nature. A realtime audio subsystem typically produces a block of audio samples while simultaneously playing out the previous block. If the processing or generation of one block takes more time than is the duration of one block, a gap (“drop-out”) will result. Naturally, it is possible to queue up more than one block for playback in memory, but this will add to the audio *signal delay* through the system and lead to added *interaction latency* between user input and the corresponding change in the output audio.

Interactive audio applications typically require low interaction latency. This can be achieved with short audio blocks. On the other hand, interaction events can happen at any time during the lifetime of the application and they often require additional processing. This may result in a rather uneven processing load distribution as a function of time, depending on the amount of the additional processing. Therefore, there can be a significant difference in the average and worst-case processing times. It then becomes a difficult task to find the smallest possible buffer size that allows low latency but never produces audible gaps in different usage situations.

## 3.1 Splitting the Components

A unique feature in the architecture described in this paper is splitting the methods or functions in each component into two groups: realtime and control methods (see Figure 2). Realtime or DSP methods take care of the actual audio processing and run continuously. Control methods are executed only when needed, i.e., when there are interaction or other parameter change events pending.

In this architecture, the control methods take parameter change events from upper layers of the software system. They process the parameters to a

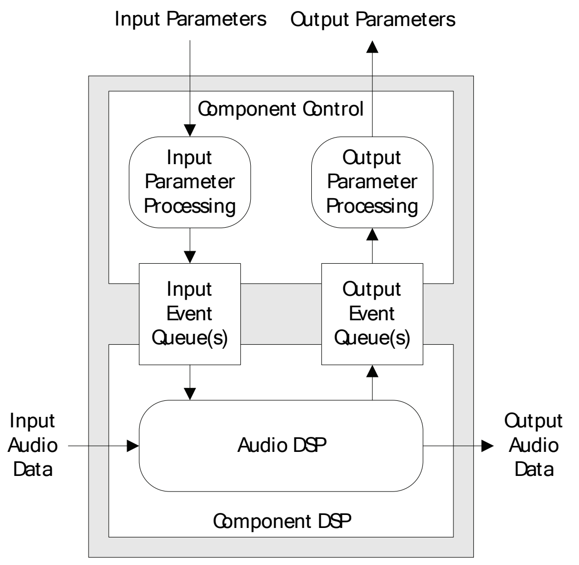

Figure 2: An audio processing component structure with separate control and signal processing parts.

format that, subsequently, can be easily taken into use in the signal processing methods. For example, calculating a linear gain value from a level value in decibels could be done on the control side of an audio mixer. The control methods may also be used for parameter processing in the opposite direction: for events that need to be transmitted from the processing to the control side, the control methods can convert DSP parameters into values more useful at the higher software layers.

The realtime methods implement the actual audio signal processing tasks. They also take the precalculated parameter values into use. Because the complex parameter processing tasks can be delegated to the control methods, the signal processing can more easily be implemented so that the computational load from these methods is substantially constant or at least strictly bounded.

## 3.2 Separate Execution Threads

The split between realtime and control methods is necessary but not enough to achieve the best performance. Additionally, the realtime and control methods need to be run in different threads of execution, the realtime thread needs to run on a higher execution priority, and the communication between the threads needs to be carefully considered.

The realtime signal processing thread can be run at a constant processing load independent of the control thread, provided that the realtime methods are written in a way that really takes a constant amount of processing power per each block of audio samples. It is also required that the realtime thread is never blocked for anything else than data transfer either between components or between software and audio hardware. When these requirements are satisfied, the signal processing thread can always run when it has data to process, and can always produce its output in a timely manner.

The control thread performs a varying amount of processing depending on the user interaction and application controls. The control code is executed in a lower priority context than the realtime methods. This ensures that the realtime thread can always interrupt control processing, but not vice versa.

The communication between the control and DSP parts of a component are organized by providing *event queues* either in a component template or base class, or in the overall processing framework. This makes implementing components with the desired split of functionality easy, and makes the consideration for the best realtime performance a standard part of algorithm development. The event queues need to be implemented using lock-free data structures to ensure that the realtime thread is never blocked because the control part is writing to the queue.

Running the control and realtime methods in separate threads effectively separates the optimization of the signal delay from the optimization of the interaction latency. The audio buffering parameters, such as block length and buffer count, can now be selected according to the constant computational load from the realtime thread. Generally, the block size can be set to a lower value than would be possible if all parameter processing happened inside the processing thread. The interaction latency, on the other hand, can be different from an algorithm to another and is subject to optimization in the context of other interaction events in the system.

The earlier systems provide limited support for this sort of a functionality split. For example, the LADSPA plugin API (Furse, 2006) accesses the control parameter values through pointers to floating point values. Therefore, it is impossible to know, if a control has changed its value without comparing the current value to the previous value. Additionally, the call to the processing function should use the parameter values as they are when the call occurs. Therefore, the only practical thing to do with these plugins is to run the control calculations within the signal processing thread.

While not a component based audio system itself, the Structured Audio part of the MPEG-4 standard (ISO, 1999) can also be used as an example of how previous systems have dealt with control events and their processing. The Structured Audio Orchestra Language (SAOL) divides its variables into three classes, initialization (i-rate), control (k-rate, and audio (a-rate) variables. The control rate variables are processed at a predefined rate that is typically significantly lower than the audio sampling rate of the algorithm. All processing still occurs synchronously from one thread. The control events that are described using the accompanying Structured Audio Score Language (SASL) are just used to change the values of parameters exposed by the sound synthesis or processing algorithm.

If control events are sent to the component using a specific function call, as is the case with the OpenMAX IL standard (Khronos, 2005), it is easier to implement the algorithm according to the architecture presented in this paper. But, as the event queues are not part of the standard, each component provider has to develop their own implementation for the queues.

### 3.3 Time-stamped Control Events

An important feature in the current architecture is time-stamping of the events in the event queues. All events can be time-stamped sample accurately when they enter the parameter processing. This is a convenient way of controlling the control event timing accuracy and jitter under normal usage conditions.

Control event time-stamping is highly useful in this architecture that is based on running control and realtime code in separate threads. If the control events are properly time-stamped and delivered to the parameter processing early enough, the fact that parameters are pre-processed by the control methods has no effect on interaction latency. Also, two events requiring different amounts of processing can easily have the same latency, if desired.

Generally, control event time-stamping cannot necessarily guarantee event timing, when the control methods are run in a separate thread of execution. However, under normal conditions the behaviour is easy to control and optimize, and even when a control event deadline is missed, it does not introduce a drop-out in the audio output signal.

## 4 IMPLEMENTATION

We have implemented the architecture described in this report on two different platforms. The first version runs on a single processor system (or a symmetric multi-processor system) in several threads. Control methods are run in one or more threads and the realtime methods in their own high-priority thread. The second runs on a heterogeneous two-core processor system.

The first version can be run on a general purpose OS, such as the typical PC operating systems or the Symbian OS for smartphones. In this implementation, all components inherit a common C++ base class that defines the component interface, separates control methods from signal processing methods, and implements the event queues described in section 3.2. The framework has been implemented such that the events in the event queues are automatically dispatched in the signal processing thread. This means that sample accurate timing is easily achieved with all components.

The second implementation has been designed for an embedded system using a processor chip that contains both a general purpose ARM processor core and a separate DSP core. Both processors run their own operating systems with a vendor-specific communication interface between the processor cores. The signal processing functionality is run on the DSP processor, and the control methods of each algorithm are executed on the ARM. In this implementation, the control and signal processing parts of an algorithm are, naturally, more separate than in the first implementation. Still an important feature prevails: each DSP side algorithm is accompanied by an ARM side control processing instance, and the details of the messages sent between the control and the signal processing parts are considered internal to the algorithm.

Both of the implementations described above have been successfully used for applications requiring low delay and latency with several different audio processing components, such as mixers, sampling rate conversions, and audio effects. The constant load from the processing methods has been found a significant benefit, when optimizing a complete system for reliable low-latency operation. Especially, the second version can now fully utilize the potential of the separate DSP core, as the varying processing from the control calculations is delegated to the general purpose ARM processor core.

## 5 ALGORITHM EXAMPLE

Consider an algorithm, where an FIR digital filter is used for filtering according to a frequency domain specification given while the processing is running. For example, a user controlled audio equalization algorithm could be implemented so that the user controls are first converted to a frequency domain target response. Then, a time domain impulse response can be calculated using the inverse discrete Fourier transform (IDFT). This time domain response will then be used for actually filtering the audio signal going through the equalizer.

This algorithm is characterized by the significant amount of processing needed for transforming the user controls to the response that can be used in filtering. The target response typically needs smoothing before the IDFT is calculated. After moving to time domain, the response needs windowing and truncation before it is ready to be used as an FIR filter. On the other hand, FIR filtering is an operation that is very efficiently implemented on modern processors, especially on dedicated signal processors. All this combined means that the filter design phase easily takes roughly the same number of processor cycles as processing a short block of audio samples.

When this algorithm is implemented according to the current architecture, the audio signal delay through the system can be kept very low. There is no need to queue up more audio buffers, even if it takes a considerable amount of time to process the parameter changes. Instead, the efficiency of the standard FIR filtering can be leveraged fully, when the parameter processing happens in an execution thread separate from and parallel to the signal processing thread. On the other hand, the interaction latency is still determined by the time it takes to transform the user controls to the FIR coefficients; this time cannot be considerably shortened.

## 6 CONCLUSION

A component based audio software architecture for an efficient realtime audio system was described in this paper. The key feature that differentiates this architecture from previous work is the systematic division of the component functionality into two groups of functions or methods. The benefit that this division brings is that the processing load of the realtime part can be kept substantially constant regardless of the amount of interaction.

Constant processing load is a significant improvement for DSP resource management, because earlier systems have had to prepare for the worst-case estimates (or take the risk for drop-outs). The worst cases happen relatively seldom, thus leading to the situation in which the best potential is wasted.

In addition to the basic architecture, this paper shortly described two different concrete implementations. A digital filtering component with a sophisticated on-line filter design algorithm was also used to highlight the benefits of the architecture.

## REFERENCES

Datta, J., Karley, B., Chen, T., Longley, L., Baudendistel, K., and Dulanski, T., 1999. “Architecting a Versatile Multi-Channel Multi-Decoder System on a DSP,” Presented at the AES 106th Convention, Munich, Germany, May 8-11, 1999.

Furse, R., 2006. “Linux Audio Developer's Simple Plugin API (LADSPA),” available at <http://www.ladspa.org/>, referenced June-12, 2006.

GStreamer, 2006. “GStreamer: open source multimedia framework,” available at <http://www.gstreamer.org/> referenced June-12, 2006.

ISO, 1999. Coding of multimedia objects (MPEG-4). International Standard ISO/IEC 14496:1999, Geneva, ISO.

Khronos Group, 2005. “OpenMAX Integration Layer Application Programming Interface Specification,” Version 1.0. available at <http://www.khronos.org/openmax/>.

Lohan, F. and Defée, I., 2001. “Modularity in Open Media Terminal System Architecture,” In Proc. 2001 IEEE Int. Conf. on Multimedia and Expo, Tokyo, Japan, August 22–25, 2001. pp. 708-711.

Microsoft, 2006. “DirectShow,” available at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directshow/htm/directshow.asp>, referenced June-12, 2006.

Phillips, D., 2001. “Linux Audio Plug-Ins: A Look Into LADSPA,” available at <http://www.linuxdevcenter.com/pub/a/linux/2001/02/02/ladspa.html>, referenced June-12, 2006.

Steinberg Media Technologies GmbH, 2006. “Our technologies,” available at <http://www.steinberg.de/325_1.html>, referenced June-12, 2006.

Texas Instruments, 2002. “TMS320 DSP Algorithm Standard API Reference,” TI Literature Number SPRU360C, 2002, available at <http://focus.ti.com/lit/ug/spru360c/spru360c.pdf>, referenced June-12, 2006.