REAL-TIME SIGNAL PROCESSING SYSTEM PROPOSAL AND IMPLEMENTATION

Carlos Gómez Moreno

Audio Department
Sinixtek ADTS, s.l.
carlos@sinixtek.com

ABSTRACT

This paper intends to describe the desirable features of a complete, powerful and highly customizable real-time audio algorithm implementation system, and to provide the guidelines to its implementation. The goal is to design a platform by means of which new sophisticated audio algorithms can be developed, tested and used with the minimum effort. The idea is to build large complex processing systems based on elemental building blocks which may interact in any possible manner. This way, by connecting existing, proved modules, such as filters, noise gates, or any new specific module, complex processes can be achieved and tested in a real-time environment with the minimum possible effort. The building-block philosophy would also make such a system very suitable for educational purposes, as it would make possible to 'hear' in real-time a particular complex algorithm with and without one of its blocks (a filter, for example), thus showing its importance.

1. SPECIFICATIONS

The main desirable features for a flexible and powerful audio processing system as described above are the following:

1.1. Flexibility

Connections between the different building-blocks must be arbitrary and unlimited, no topology pattern or block count limit should be imposed by the system. The connections must at least include signal connections and control connections. The latter may be defined as special connections that allow one building-block to change the parameters of other in real-time. Optionally, one block may have multiple signal input, output and control output ports.
It must be possible to change the parameters of any building block in real-time. This is done with two purposes: he first is to allow the user to make changes to the audio process and hear the effects immediately; the second is to allow automatic parameter modification. This way one building-block can be designed to change other block's parameters sinusoidally, and be reused both in a 'Flanger' or 'Tremolo' application.

1.2. Dynamic loading – Development community

The user must be capable of loading the unit with newly developed or third-party algorithms, and the developer must be able to hot-replace and debug new code. All this must be achieved without re-compiling the system core. The main objective is to promote an open code interchange community, by means of which new, complex algorithms can be developed very easily by building up on basic, tested blocks.

1.3. Portability

The unit must be able to function autonomously. This is due to the need to test algorithms in particular environments and conditions, with real-world signals. However, extra functionality may be provided when connected to an external device, such as a computer.

1.4. Programming simplicity

When the development of a new, specific building-block is needed, it must be programmed. The developer does not need to know the intrinsics of low-level programming. The system must provide him with a plain, standard interface where all architecture-dependant operations (such as I/O) are handled by the system. Ideally, the only knowledge required by the programmer is Digital Signal Processing.

1.5. Soft transitions upon process change

When the user requests a process change, the transitions between the old and the new process should be smooth and instant. This is particularly important when the process to be replaced has memory, because the discontinuity is more evident and unpleasant in such cases.

1.6. Efficiency

The intelligence and the code size in the unit must kept at a minimum in the real-time (audio processing) thread, leaving the as much calculations as possible to other low-priority threads. The system must be speed-optimized for patch execution.

2. IMPLEMENTATION

This section discusses the implementation details of the different elements that must be present in the designed real-time audio algorithm system.

2.1. Audio algorithms

An audio algorithm is basically a 'black-box' which is capable of producing outputs, either as a result of an operation on its inputs, or in an input-independent way (this would be the case of generators). These operations may depend on parameters the algorithm

may have. It is the building-block for complex processes described in the previous section. Instances of algorithms are called 'Effects'. We will discuss now several considerations which will provide us with guidelines to its implementation.

2.1.1. Considerations as a Black-box

- Algorithms may have any number of signal inputs and outputs.
- Algorithms may have any number of parameters, these parameters may be of integer or floating-point type, and may have different allowed ranges and quantization steps. These parameters must be allowed to be changed in real-time.
- Algorithms may have additional characteristics that the system must be aware of, such as natural extinction request.
- It must be possible to load and unload algorithms from the system.
- It must be possible to modify the parameters of an effect in real-time.
- It must be possible for an effect to modify parameters of other effects in real-time.

2.1.2. Implementation Considerations

- Algorithms must be implemented in 'C' for efficiency reasons.
- It must be possible to load and unload effects.
- It must be possible to instantiate algorithms as many times as needed.
- Effects must be able to allocate resources (memory) upon creation and liberate them upon destruction.
- Algorithms may veto parameter changes (for example, depending on the values of other parameters).
- Effects may have to recalculate internal variables upon parameter changes.

2.1.3. Implementation

Taking into account these considerations, a 'Code + Descriptor' approach is suggested.

2.1.3.1 Descriptor

The descriptor provides system with the I/O capabilities, parameter number and type, and extra information, of an algorithm. It exports the 'Black-box' interface description, and allows for parameter modification from the outside.

2.1.3.2 Code

The Algorithm code must be written as a dynamic-load module and delivered in a binary format compatible with the implementation platform's architecture. This is must in order to promote an algorithm interchange forum.
This module must have the following entries. For simplicity reasons, any of this methods may be missing. In that case, the system must provide a default implementation, which in most cases will be useful.

- *Initialization* – Called when the effect is initialized. It is used to allocate memory and initialize variables.
- *De-Allocation* – Called when the effect is terminated. It is used to free allocated memory.
- *Process* – Called every block to process/generate audio data. This is where the algorithm is coded. Input and output buffers are provided by the system to this function.
- *Parameter change* – Called to request a parameter change. This would be mainly used to veto or limit a change in a parameter depending on the value of other parameters or variables. Note that, although parameter range is respected by the system, the user might want to limit the range depending on other values (for instance, to impose a slew-rate control to a sine-wave generator, the amplitude must be limited taken the frequency into account).
- *Internal variables recalculation* – Called upon a successful parameter change. This would be used to recalculate the internal variables when a parameter changes.
- *External parameter change* – Called to request a parameter change in other effect. This is used to produce automation values, which will be routed and scaled by the system in order to produce a parameter change in other effect.

In all these methods, an instance context object must be passed. Parameter values, block size and sampling frequency information should be placed by the system n this object. Additionally, the user should be allowed to add to this object as many variables as he might need to make his calculations.

To minimize code size, the system should provide the dynamic modules with the basic mathematic and DSP functions, which would be linked in algorithm load-time.

2.2. Patches

Patches are defined as a set of effects and their relationships (i.e. connections). They constitute the process applied to the input signal, as described in the Specifications section.

2.2.1. Considerations

- Patches may be loaded and unloaded.
- Connection between effects must be arbitrary.
- Connections must include control connections (virtual connections by means of which effects can automate the parameters of others).
- If multiple signal connections are made to a single input of a given effect, the signals are added.
- If multiple signal connections are made to a single output of a given effect, the signals are copied.
- It must be possible to partially unload a patch in order to let only the signal paths with memory fade naturally.

2.2.2. Implementation

For efficiency reasons, the patch must be arranged as an array of effect descriptors, which must contain the algorithm id, a buffer id for each input or output, the set of parameter values, and a set of option flags.
Because effects are executed in order in which they appear in the array, sorting must be performed to ensure that all effects are executed after all their inputs are available and processed by the

preceding effect chain. In this scenario, a signal connection between different effects means that they share a buffer, which is written by the effect located upstream and read by the downstream effect. This way, multiple readers share the same input data. When there are multiple writers, all but the first one in the order of execution must add their result to the buffer. This may be achieved by the system inserting a special effect, which would be capable of adding its input to an output buffer, in series with each of them.
Now, the need to achieve partial patch de-allocation (in order to keep the memory section of an unloaded patch sounding to produce a smooth transition), must be taken into account. The information that a particular effect has memory is one of the options (we will call it the 'Keep' option) that must be present in the descriptor, and must be set whenever the effect is an instance of an algorithm that has memory (a Delay, for example), or when there is one of such effects earlier in the effect's signal path. This way, when time is come to unload the patch, only the effects marked as 'Keep' are kept, and all the inputs in this sub-patch that are not connected to other effects in the sub-patch (i.e., they were connected to effects which have been unloaded) are zeroed-out. This makes these sub-chains extinguish naturally (due to the fact that they are fed with zeros), while their outputs are added to those of the new patch, thus achieving the desired transition softness.
Control connections should treated separately as a list of automation entries. Such entries should have origin effect id, destination effect id and destination parameter index (inside the effect). After the execution of each effect, for each list entry for which the effect is the automation origin, the destination parameter must be updated. This is done by calling once the 'External parameter change' method. The effect is responsible of generating a value between 0 and 1 to automate the destination effect, or –1 if it won't be generating output this block. Then, the system must change all affected destination parameters. It is up to the system to scale the value to each destination parameter's range and to do the requested quantization. This is specified in each destination parameter's descriptor.

2.3. Proposed user interfaces

2.3.1. Rich user interface

In order to make patch design friendly to the user, a rich user interface based on graphical black-box interconnection is proposed. This way, the user places effects (represented by a box or other graphical element) on a canvas, and then interconnects them (the connections may be represented by lines). The user must have a way of choosing which kind connection he wants (a signal connection or an automation connection) and then the connection parameters (to which signal port, in the case of a signal connection, and which parameter to automate, in the case of an automation connection). Physical inputs and outputs the unit may have can be represented in the same way the effects are.
Once the design is complete, the graph must be translated into the described patch structure, and it is ready for execution. This interface will typically reside in an external device (a computer), so a way of communicating the external device and the system must be provided.

2.3.2. Autonomous user interface

As the resources for the rich user interface described may not be practical to include in an autonomous unit (a big graphical display would be needed), a minimum user interface is proposed. This reduced interface would not allow the user to create a new patch or to make structural modifications to an existing one, but would allow the user to adjust effect parameters in real-time, and to load/unload previously stored patches. The proposal consists on a basic text display and keyboard interface (easy to include in a portable unit). Keys may be used to change between patches, to edit them, and to save them. Once in edit mode, the patch name may be allowed to be changed, and then, for example, parameters can be traversed and changed in real time.

3. CONCLUSIONS

The proposed system may be a very powerful signal processing workbench for both professional and didactical purposes. The possibilities offered by an algorithm exchange community may be very interesting to promote the development of new advanced audio algorithms.
An implementation of the described system has already been done based on a Texas Instruments TMS320C6713 DSP, with very good results.