Generic DSP system thoughts

This will progress from a series of bullet points and somewhat random notes to a proper document over time.

Motivation

Originally, I wanted to use AUV3 as a way to create DSP blocks, wire them together using a patch bay model, combine them with other AUV3 components in third party hosts such as Logic Pro etc. Even though the granularity of blocks would be coarser than desirable, and platform independence would be non-existent, the benefits would outweigh these disadvantages. However, this dream has gone down in flames because of these factors:

- Crappy Apple example code
- Non-existent AUV3 documentation in general and on important subjects such as device discovery.
- No third party Mac host with a patchbay model.
- Poor AUV3 support in Logic Pro.
- Multiple problems creating and using AUV3 AUs on the Mac OS platform
- No indication of a recognition of any of these problems by Apple.

Having given up on AUV3 as a lingua franca for DSP blocks and realizing that something else would have to be built, I was no longer willing to deal with platform dependence, coarse granularity, and other AUV3 issues.

Central Tenets and Design Goals

- Granularity is far smaller than systems such as AUs and VSTs. Whereas a synth or complex effect would be *delivered* as an AU, it would be *comprised of* Generic DSP blocks. If for instance it were delivered as a VST, a different host wrapper would be used.
- Core processing path code is platform independent, should run on Apple, Linux, bare metal, and Windows. Therefore is written in C++.
- Block Based for efficiency. This enables leveraging SIMD and other vector processing feature
- Heirarchy is supported. A *graph* can itself be a *block*, ad infinitum. This will be important to keep a complex design such as a synth manageable.
- Optimized blocks will not be platform independent if they leverage features such as the Apple Accelerate library directly. But at the block level they can be platform independent. Even library calls used by multiple blocks can be made device independent (i.e. VectorAdd).
- Buffer size can be determined by the graph. Buffer size will not change during a “run” of processing. External “host adapter” code can deal with external contingencies such as variable number of sample per render call and unknown latency.
- Based on the concept of Pins (aka Busses). Each DSP block can have N number of input and output pins, though in most cases N will be 1 for both input and output pins.

- Pins are multichannel, so a stereo in stereo out effect will still have one input and one output pins.
- Multiple sample rates are supported for over and undersampling, so each pin has a sample rate. It is expected that all downsampling and upsampling factor will be integers.
- The model is that connection parameters propagate from the outputs of blocks to the inputs of other blocks. So the act of connecting blocks together sets up things like sample rate, buffer size, and number of channels. This greatly simplifies the code to describe a graph.
- Multiple channel signal paths are implemented as multiple buffer pointers, rather than interleaved buffers. This makes it compatible with AUs and also VST AFAIK.
- Pan out of blocks is free, since multiple input pins can point at the same output buffers

Inspiration

Many ideas are lifted from DSP Concepts' Audio Weaver:

https://dspconcepts.com/solutions/audio-weaver

I would encourage watching some of the youtube videos and looking at the docs to get a feel for it.

Important differences from AudioWeaver

- Fixed point data paths are not supported. 32 bit floating point data is assumed.
- Implementation in C++ 11 rather than C. Audioweaver was started earlier and designed to run in even tinier platforms. C++ was not as universally supported as it is today.
- It is assumed that all target platforms will support a C++ runtime including sufficient resources to support dynamic memory allocation during initialization and reconfiguration.
- The design tool will not be driven by Matlab. It is highly desirable that it be platform independent, or even web based.

Visual Design Tool

The visual design tool superficially resembles other tools used for schematic design, etc. DSP *blocks* from a *library* are assembled and connected with "*wires*". Parameters for these blocks are specified using pop up design dialogs. Heirarchy is supported with the idea of subsystems. Subsystems are displayed as "superblocks", but can be dived into visually to look inside them. Again, similar to a schematic design package. When the design is finalized, either the design is run "live" on a target system, or a design output file is generated.

In addition to this fairly generic functionality, the tool is responsible for supporting the following domain specific tasks (either by itself or by issuing command to a target app, described in a subsequent section below):

- Propagating wire properties, mostly in the direction of signal flow but with some exceptions. It should display the wire properties if the display option is set. Wire properties include sample rate, number of channels, and buffer size
- Specifying the allocation of buffers used to pass signals between blocks. This will include a facility to reuse buffers to save memory, when probing intermediate signals is not necessary.

- Specifying the processing order of the blocks in the diagram. This is by nature tightly connected with buffer use and is arranged for minimum latency.

Parameter Interface

The subject of parameters is complicated and multifaceted. A few things to keep in mind as design progresses:

- There is a difference between design time only parameters and runtime parameters. Most parameters are runtime parameters, but an example of a design time only parameter would be maximum delay of a delay line, since this involves memory allocation.
- The design tool should be able to adjust both types of parameters, but adjusting the runtime parameter would set the *initial* value of the parameter
- Parameter addressing/discovery and parameter types are big subjects. They are particularly important subject for creating automatic “generic” UIs, both for the design tool and for runtime UIs.
- Parameter addressing must support hierarchical graphs with subsystems. You might have two different subsystems called “High Frequency” and “Low Frequency”, each with a compressor block. There needs to be a way to distinguish these blocks, either by using a tree structure or a pathname type scheme.
- There is overlap between the idea of signals and parameters. A gain block and a multiplier are the same thing, except for the fact that one has one input signal and one parameter, and the other has two input signals. If parameters aren’t treated distinctly, everything becomes a signal. This distinction needs to be formalized and defined.
- The parameter adjustment scheme should support operation over a remote link given the appropriate transport protocol. This would enable for example wireless control of a hardware appliance running Generic DSP graph.

Buffer Ownership and Processing Order

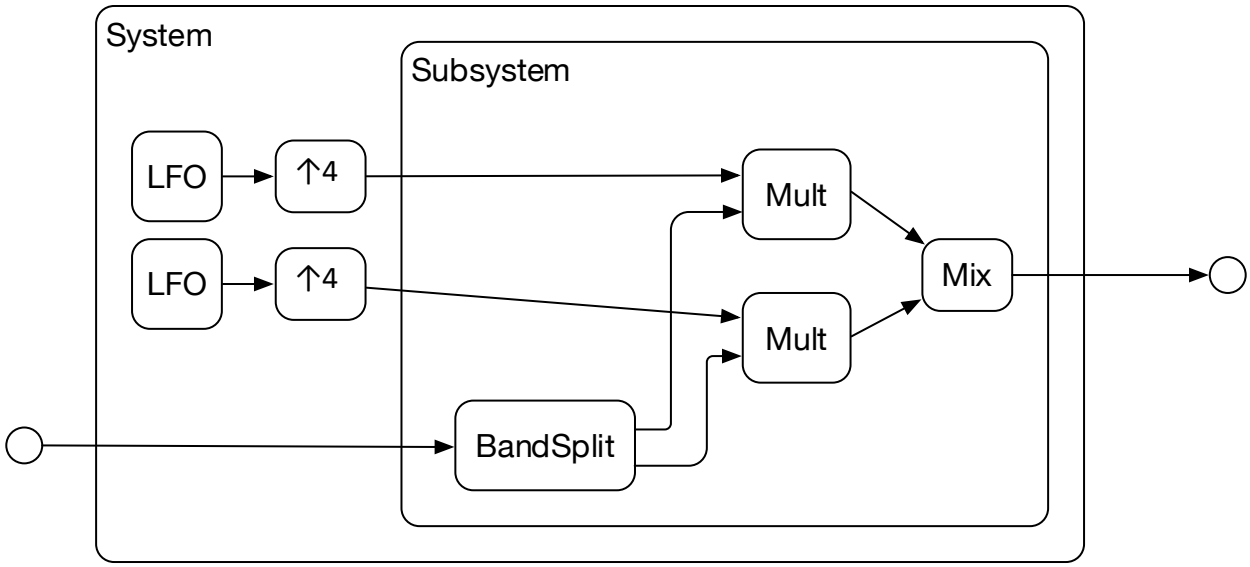
Buffer ownership and block processing order are worked out by the Visual Design Tool, as described above. A couple things are working noting here though:

- There is no need in this system to ask individual DSP blocks to do “in place processing”. In place processing complicates the design of the DSP blocks, and all it does is save buffer memory. Buffer memory can be just as easily conserved by reusing buffers.
- If it is desired to “probe” a signal for diagnostic purposes, then obviously its buffer cannot be reused. We therefore need a flag to specify this. We also need to think about “probe discovery” in addition to parameter discovery

Example Graph

An example graph for a fictitious and probably fairless useless “dual band tremolo”. This will be used a test mule to explore the algorithms for propagation of wire properties, determination of processing order, and allocation of wire buffers.

Note that this is not what the graph would look like in the visual design tool. There are no custom block shapes, and because the graph has hierarchy, the “Subsystem” block would look like a single block, and would only expand to a new pane if double clicked or opened



Object model

Blocks

Blocks are usually leaves of the object tree, unless they are also Graph Blocks (see below). They are the agents which process audio. When a process() method is called on them, they take data from input buffers corresponding to input Pins that they own, and produce output that is put in output buffers corresponding to their output Pins.

Blocks can have Parameters in a Parameter Set. Parameters can be design time parameters only, or they can be parameters which can either be set at design time of when a Graph is running.

Pins

(Alternate name suggestions: ConnectionPoint or ??)

Pins are elements within a block, specifying the connection points of the block. Each Pin contains these important member properties:

- Number of Channels
- Sample Rate
- Buffer Size
- Buffer Pointers (array of pointers)

Connections

(Alternate name suggestions: Wires or Busses or Connectors or ??)

Connections are also leaves of the object tree. Connections are objects mainly useful at design time. They contain source and destination properties to specify the Pins and Blocks that they are connected to, as well as some housekeeping properties. A connection can only specify one

source pin, but can have multiple destination pins. This allows “pan out” of one block output to multiple inputs on other blocks. The four pin properties mention in the section above must be identical for all Pins attached to a Connection. You might say that the Connection has the same properties, but the actual storage for the Pin properties is stored within the pins, for runtime convenience.

Graphs

Graphs are a collection of Blocks and Connections which together specify a processing algorithm. The intent is that Graphs can also be Blocks, to support hierarchy. Of course there must be one top level Graph.

Lifetime of a Graph

Composition

During the composition phase, Blocks are instantiated and connected by Connect operations which instantiate Connections. The eventual goal is that composition will be performed using. GUI design tool.

Signal Propagation

During the signal propagation phase, the characteristics of the Connections are propagated through a Graph, in order to make the specification of Connections types practical and convenient.

For instance, in a simple graph, if any Connection in the Graph is specified to be 2 channels, 44100 Khz, with a buffer size of 128, then this specification will propagate through the graph when the propagation algorithm is run, and all the Connections will acquire that specification.

It gets more complicated in a Graph with multiple sample rates or with blocks which don’t have the same number of channels for each pin. In this cases, the blocks have rules for signal propagation with the Graph’s propagation algorithm uses.

In some cases, there will be conflicts or situations where a signal type cannot be determined. The UI for dealing with these situations is TBD.

The logic for signal propagation can in principle run in either the GUI or in a server component, at present it runs in a server component closely coupled with the code for the processing blocks.

If the Signal Propagation algorithm is to run in a GUI, it must be able to determine the signal propagation rules for each Block.

Determining Processing Order and Buffer Allocation

A Block cannot process data until all the data at its input pins has been computed. Using this simple principle, an algorithm chooses a processing order for the blocks. This same algorithm can arrange for buffer reuse. The principle there is also simple. After all the blocks which have their inputs connected to a given Connection have processed their data, the buffers associated

with that connection can be freed and reused. At present, buffer will only be reused if the signal types match (SampleRate/NumberOfChannels/BufferSize). More complicated schemes are probably possible but also probably unnecessary.

The logic for signal propagation can in principle run in either the GUI or in a server component, at present it runs in a server component closely coupled with the code for the processing blocks.

Loading the Graph

On some systems, the blocks may not be loaded even after the steps above have run. In such systems the block need to be loaded at this point in time.

Initialization of Blocks

Once the signal types of all the Pins on a block have been determined, the blocks can be initialized. Often there is nothing or not much to this, but certain types of blocks will need to make initialization decision based either on signal types or on the setting of Design Time Parameters. Examples would be filter which need to calculate coefficients or a delay line which has a maximum delay set at design time.

Initialization Blocks has to run on the target system or target process.

Connecting to inputs and outputs

This really may be not a step per se, but at some point the host environment must be set up so that the Graph can be supplied with audio data to process, and it can harvest the data being processed.

Running the Graph

The Host environment repeatedly:

1. fills the Graph's input buffers
2. run the Process() function on the Graph
3. harvests the Graph output buffers.

Stopping and Starting

Stopping a graph can be as simple that host not calling the Process() function for the graph. In the stopped state, the Graph would even be reinitialized.

Tear Down

Free up render time resources

UI API

The UI will be implemented using web based technologies. The target system (where the DSP and important business logic will run) will be implemented in C++, for platform independence

and runtime efficiency. For this reason, and also so that the UI can run on a different machine than the target (e.g. over a network), the communication between the UI and the target will use a remote procedure call type of interface with support for object based feature (e.g. JSON over HTTP, etc).

The use case for operation described below envisages that the target will run a Host program suitable for the target platform and that the system will be capable of "live operation", where a Graph can be built and run interactively from the GUI. But an alternate and extremerly important mode of operation is to generate a Graph configuration which can be subsequently packaged for standalone operation, such as within a plug-in such as AUV3, VST, etc. Most of the steps are similar except the ones supporting live operation.

Target Code Organization

The target program will be a complete executable running in its own processed, comprised of several parts:

- Base functionality used for discovering Libraries and Blocks, instantiating and initializing Blocks and Graphs, and running graphs. It will also likely be capable of connecting Blocks, running algorithms for signal type propagation, determination of processing order and buffer allocation.
- Common libraries used for all DSP operations. This will include but not be limited to a set of generic vector operation which will leverage the capability of the SIMD processor founds on so many platforms today. Other examples would be standard filter operations, FFTs, coefficient generation, etc.
- Block Libraries. These are sets of available processing blocks. Initially, there will be just one block library, but this will probably become unmanageable, so a capability for multiple libraries must be anticipated.

Library Discovery

The target app must be able to report available libraries to the UI. This will likely happen by reading some local configuration files to find out that the available libraries are. These will almost certainly be dynamic libraries such as dylib on Mac/Linux and dll on Windows.

Block discovery

Once the target app loads a Block Library under command by the UI, that library must support Block Discovery. This includes both a list of blocks and information about each block including:

- Name
- List of Pins
- Perhaps some constraints on the types of signals supported by each Pin.

- A parameter tree, specifying each parameter and the type and legal ranges of that parameter. Note that the legal ranges of a parameter might become more restricted once the graph has been processed and the signal types of all Connections is known.
- A specification of the type of visual representation for the Block.

Support for hierarchy

Although this is a bit TBD at this point, the must be a capability to create a Graph out of Blocks, which will itself be a Block within a containing Graph.

Connection of Blocks

The target app must support commands to connect Blocks via Connections

Design time parameter initialization

Parameters have different types. Some parameters can only be adjusted at design time, others can be set at design time or runtime. In either case, the target app must provide commands to set parameters in the target graph.

Processing the Graph

Processing the graph consists of signal type propagation, determination of block processing order, and buffer allocation. Depending on design decisions, this could take place either in the UI or on the target. If this is implemented on the target, the target app must provide a command for this, as well as ways of conveying errors that can occur during signal type propagation.

Initializing the Graph

This is much like AllocateRenderResources in AUV3 and will take place in the target app, which will furnish a command to perform this task and report any errors that might occur

Running and Stopping the Graph

The target app must provide command stop and start the graph.

Host configuration and test modes

The host logic built into the target app must provide support for connecting and streaming to and from audio HW on the target. It will also support test modes such as reading audio data from a file or saving audio data to a file.

Graph Tear Down

This is much like DeallocateRenderResources in AUV3, and the target app must support this command as well.