

C++ Plugin Frameworks

Thomas Klambauer*

June 23, 2009

Abstract

Continuing from the short introduction paper *Generic image processing* we delve into the further analysis of plugin frameworks, here FxEngine and LADSPA for C/C++, both with a focus on signal processing. A specific look is taken at the solutions to resource management and the overall architecture. Furthermore we present available tools for plugin interconnection and composition.

1 Introduction

In the quest for important plugin environment properties and solutions to common issues we analyze the frameworks of two products freely available. The first free as in “no cost”, the second free as in “freedom” concerning open source software.

2 FxEngine Framework

The FxEngine Framework[1] is a closed-source C++ plugin framework licensed with an attribution clause and required author’s permission for bundled distribution. It can be used on Linux and Windows platforms with DLLs and shared objects.

Signal processing is an important application area of the framework and the underlying concept of plugins reflects this fact, as they have multiple “Pins”: for parameters and for the main input/output data. The architecture of the *engine* which connects and drives the plugins - which are here also called “Fx” - augments this impression of an signal processing system:

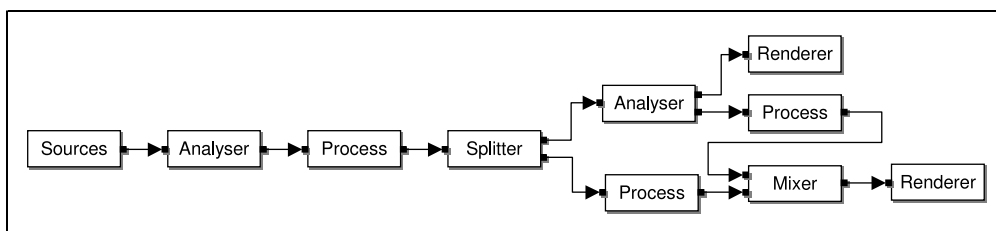


Figure 1: FxEngine architecture (FxEngine Doc)

*Thomas@Klambauer.info

The principal different plugin categories and their interconnection in one Fx-Engine instance are shown in figure 1 on the preceding page with Renderer being a *sink*. Based on this network, arbitrarily large systems can be developed using the multiplicity of FxEngine objects illustrated in figure 2.

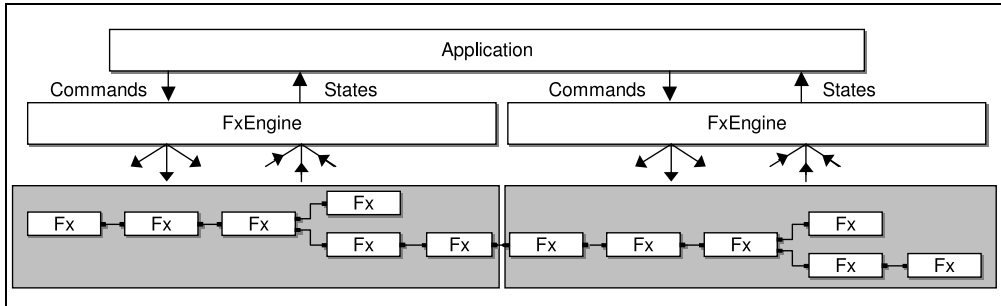


Figure 2: FxEngine multiplicity (FxEngine Doc)

FxEngine Framework uses an `enum`-based *runtime type information* for the media that is passed between plugins (by itself untyped). An interesting point hereby is, that multiple media types which are supported can be specified per pin. A wide variety of types in the field of audio and video as well as text are predefined and implementations of plugin types using that media are available.

Media data itself is intended to be transferred in data chunks allocated and managed by the `IFxMedia` class. The chunk size can be adjusted and filled with user-defined data, so that the passing of preallocated data is possible although it doesn't seem intended by the framework.

The basic work-flow is to instantiate an FxEngine, load plugins through its interface and then to establish *plugin interconnection* through that same interface. Then by running the engine the data processing starts. A visual editor, "FxEngineEditor" comes with the software package which performs this tasks and is able to store and load the network information in a proprietary file format. In figure 3 on the following page the connected input and output port of two plugins can be seen in the editor with "Hello World" as processed text data.

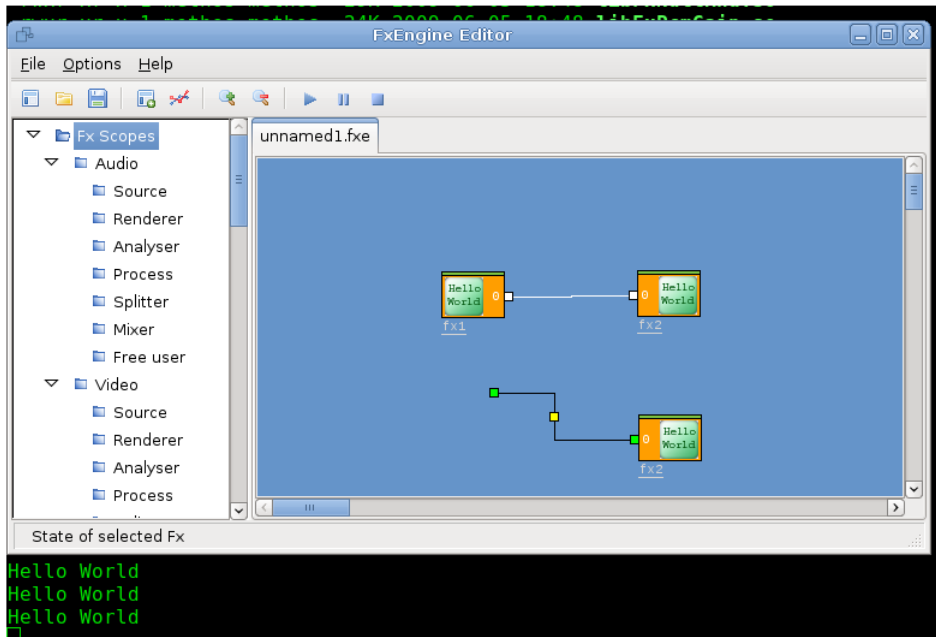


Figure 3: FxEngineEditor

Another interesting aspect of the Framework is the propagation of the processed data: the *synchronization of the plugins*. This can occur either by clock signals or a media request signal - a pushing versus pulling approach. The *error handling* and interface requesting bears strong similarity with the COM error handling of checking return values with macros and predefined error values.

Resource management, especially of memory shared between the plugins, which is another big concern for C++ module systems is managed by *allocation and release functions*. It can be observed, as with many other plugin and library systems that the problems with cross-module memory-management and possible inconsistencies with type definitions of objects crossing the boundary, due to the separate compilation of modules, drives or forces many developers into the direction of a C-style interface rather than an object-oriented one.

Amongst the available plugins also a LADSPA wrapper can be found enabling the integration of a wide variety of Audio plugins, which brings us to the next framework.

3 LADSPA

The “Linux Audio Developer’s Simple Plugin API”[2], in short LADSPA, defines a simple C interface (`ladspa.h`) for audio-processing plugins under Linux. The framework only consists of this header file, which is licensed under LGPL[3]. It is settled in an open source environment, aimed at providing an VST - Virtual Studio Technology by Steinberg - replacement for Linux. A large number of plugins are available implementing this interface and a number of prominent hosts like “audacious” can make use of them. LADSPA does itself not introduce any dependencies.

A module can contain multiple plugins, whose functionality is accessed by requesting *descriptors*. Plugins have input/output *ports* for control and audio

data, which is passed simply as `float` arrays.

Plugins are controlled by functions specified by function pointers in the descriptor. See the listing below for the definitions.

```
1  /* only export */
2  const LADSPA_Descriptor * ladspa_descriptor(unsigned long Index);
3
4  typedef struct _LADSPA_Descriptor {
5      /* author info etc. */
6
7      /* port data */
8
9      LADSPA_Handle (*instantiate)(const struct _LADSPA_Descriptor *
10         Descriptor,
11                                     unsigned long
12                                     SampleRate);
13
14     void (*connect_port)(LADSPA_Handle Instance,
15                           unsigned long Port,
16                           LADSPA_Data * DataLocation);
17
18     void (*activate)(LADSPA_Handle Instance);
19
20     void (*run)(LADSPA_Handle Instance,
21                 unsigned long SampleCount);
22
23     void (*run_adding)(LADSPA_Handle Instance,
24                        unsigned long SampleCount);
25
26     void (*set_run_adding_gain)(LADSPA_Handle Instance,
27                                 LADSPA_Data Gain);
28
29     void (*deactivate)(LADSPA_Handle Instance);
30
31     void (*cleanup)(LADSPA_Handle Instance);
32 } LADSPA_Descriptor;
```

An issue mentioned also in the documentation is the limited *error handling* capability. As solely the user's language C is assumed, no exceptions are expected or defined and no return values are designated to this. However this keeps the design very simple which is one of the stated goals. *Resource management* happens again with an allocation and release function pair.

For *module interconnection*, the LGPL'ed mixing software package GDAM - "Geoff & Dave's Audio Mixer" [4] contains functionality to chain LADSPA plugins visually by defining a flow graph. This mini-network can then be saved to XML whereof an encapsulating LADSPA plugin can be generated automatically, allowing for easy chaining and composing of plugins. In figure 4 on the next page an instance of this editor can be seen.

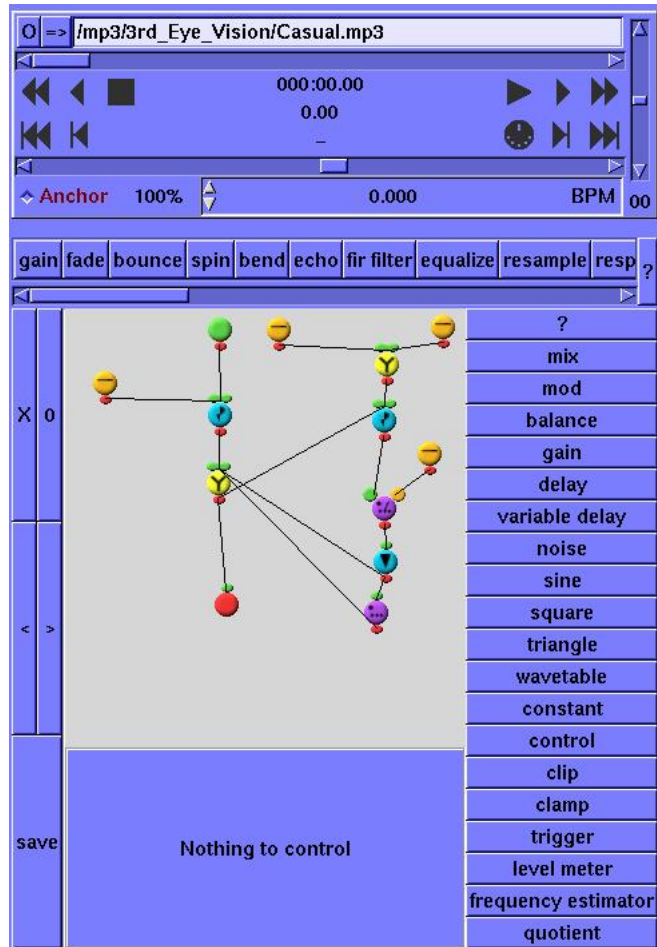


Figure 4: GDAM flow graph editor

4 Cross-Module Error Handling

In general, error handling in C++ is performed via *return values*, *exceptions* or a state information of some kind, where a module-level state and a function level result does not replace each other and can be complementary. Within modules - applications and libraries alike - often exceptions are preferred, as they blend in naturally with the RAII pattern[5].

However with module boundaries, problems arise. As the C++ Standard does not concern itself with modules - they being an operating system concept - no ABI for them is specified. And also it is not specified how exception handling is implemented. Sutter and Alexandrescu in [6] even define as C++ coding standard rule #62: *Don't allow exceptions to propagate across module boundaries*.

This results from the fact the the exception handling code that is generated may be compiler/compiler-version and even compiler-flag dependent and can thus not be reliably guaranteed to the same across different modules.

With this in mind, it seems we best resort to error handling with return values.

5 Conclusion

We analyzed the FxEngine and LADSPA plugin environment with regards to *Resource Management*, *Module Interconnection*, *Error handling*, the interface and available tools in general. It was observed that error handling has a significant role and needs to be considered during plugin framework design. We also discussed *synchronization* and plugin composition as part of plugin frameworks, which may be required but the former introducing some additional complexity.

The work on plugin frameworks as part of “Generic Image Processing” will be continued in the same-titled bachelor thesis ¹.

References

- [1] <http://www.smprocess.com>.
- [2] <http://www.ladspa.org/>.
- [3] <http://www.gnu.org/licenses/lgpl.html>.
- [4] <http://gdam.ffem.org/>.
- [5] http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization.
- [6] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards*. Addison-Wesley, 2004.

Nomenclature

ABI	Application Binary Interface
Fx	Plugin
Host	A plugin hosting environment (application)
LADSPA	Linux Audio Developer’s Simple Plugin API
RAII	Resource Acquisition Is Initialization
VST	Virtual Studio Technology

¹To be available at <http://klambauer.info> by August 2009