KRONOS VST – THE PROGRAMMABLE EFFECT PLUGIN

Vesa Norilo

Department of Music Technology
Sibelius Academy
Helsinki, Finland
vnorilo@siba.fi

ABSTRACT

This paper introduces Kronos VST, an audio effect plugin conforming to the VST 3 standard that can be programmed on the fly by the user, allowing entire signal processors to be defined in real time. A brief survey of existing programmable plugins or development aids for audio effect plugins is given. Kronos VST includes a functional just in time compiler that produces high performance native machine code from high level source code. The features of the Kronos programming language are briefly covered, followed by the special considerations of integrating user programs into the VST infrastructure. Finally, introductory example programs are provided.

1. INTRODUCTION

There are several callback-architecture oriented standards which allow third parties to extend conformant audio software packages. These extensions are colloquially called *plugins*. The plugin concept was popularized by early standards such as VST by Steinberg. This paper discusses a plugin implementation that conforms to VST 3. Other widely used plugin standards include Microsoft DirectX, Apple Audio Unit and the open source LADSPA. Pure Data[1] extensions could also be considered plugins.

As customizability and varied use cases are always encountered in audio software, it is no suprise that the plugin concept is highly popular. Compared to a complete audio processing software package, developing a plugin requires less resources, allowing small developers to produce specialized signal processors. The same benefit is relevant for academic researchers as well, who often demonstrate a novel signal processing concept in context in the form of a plugin.

The canonical way of developing a plugin is via C or C++. Since musical domain expertise is highly critical in developing digital audio effects, there is often a shortage of developers who have both the requisite skill set and are able to implement audio effects in C++. One way to address this problem is to develop a meta-plugin that implements some of the requisite infrastructure while leaving the actual algorithm to the end user, with the aim of simplifying the development process and bringing it within the reach of domain experts who are not necessarily professional programmers.

This paper presents Kronos VST, an implementation of the programmable plugin concept utilizing the Kronos signal processing language and compiler[2]. The plugin integrates the entire compiler package, and produces native machine code from textual source code while running inside a VST host, without an edit-compile-debug cycle that is required for C/C++ development.

The rest of the paper is organized as follows; Section 2, *Programmable Plugins and Use Cases*, discusses the existing implementations of the concept. Section 3, *Kronos Compiler Technology Overview*, briefly discusses the language supported by the plugin. Section 4, *Interfacing User Code and VST*, discusses the interface between the VST environment and user code. Section 5, *Conclusions*, summarizes and wraps up the paper, while some example programs are shown in Appendix A.

2. PROGRAMMABLE PLUGINS AND USE CASES

2.1. Survey of Programmable Plugins

2.1.1. Modular Synthesizers

Modular synthesizer plugins are arguably programmable, much as their analog predecessors. In this case, the user is presented with a set of synthesis units that can be connected in different configurations. A notable example of such a plugin is the *Arturia Moog Modular*.

Native Instruments Reaktor represents a plugin more flexible and somewhat harder to learn. It offers a selection of modular synthesis components but also ones that resemble programming language constructs rather than analog synthesis modules.

A step further is the Max/MSP environment by Cycling'74 in its various plugin forms. The discontinued *Pluggo* allowed Max/MSP programs to be used as plugins, while *Max for Live* is its contemporary sibling, although available exclusively for the Ableton Live software.

2.1.2. Specialist Programming Environments

In addition to modular synthesizers, several musical programming environments have been adapted for plugins. CSoundVST is a CSound[3] frontend that allows one to embed the entire CSound language into a VST plugin. More recently, Cabbage[4] is a toolset for compiling CSound programs into plugin format.

Faust[5], the functional signal processing language, can be compiled into several plugin formats. It has traditionally relied in part on a C/C++ toolchain, but the recent development of libfaust can potentially remove this dependency and enable a faster development cycle.

Cerny and Menzer report an interesting application of the commercial Simulink signal processing environment to VST Plugin generation[6].

2.2. Use Cases for Programmable Plugins

The main differences between developing a plugin with an external tool versus supplying an user program to the plugin itself boil down to development workflow. The compilation cycle required for a developer to obtain audio feedback from a code change is particularily burdensome in the case of plugin development. In addition to the traditional edit-compile-run cycle, where compilation can take minutes, plugin development often requires the host program to be shut down and restarted, or at least forced to rescan and reload the modified plugin file.

In contrast, if changes can be made on the fly, while the plugin is running, the feedback cycle is almost instantaneous. This is what the KronosVST plugin aims to do. Several use cases motivate such a scheme;

2.2.1. Rapid Prototyping and Development

Rapid prototyping traditionally means that the program is initially developed in a language or an environment that focuses primarily on developer productivity. In traditional software design, this can mean a scripting language that is developer friendly but perhaps not as performant or capable as C/C++. In the case of audio processors, rapid prototyping can take place in, for example, a graphical synthesis environment or a programmable plugin. Once the prototyping is complete, the product can be rewritten in C/C++ for final polish and performance.

2.2.2. Live Coding

Live coding is programming as a performance art. In the audio context, the audience can see the process of programming as well as hear the output in real time. The main technical requirement for successful live coding is that code changes are relatively instantaneous. Also, the environment should be robust to deal with programming errors in a way that doesn't bring the performance to a halt. KronosVST aims to support live coding, although the main focus of this article is rapid development.

2.3. Motivating Kronos VST

As programming languages evolve, it becomes more conceivable that the final rewrite in a low level language like C may no longer be necessary. This is one of the main purposes of the Kronos project. Ideally, the language should strike a correct balance of completeness, capability and performance to eliminate the need to drop down to C++ for any of these reasons.

The benefit of this approach is a radical improvement in developer productivity – but the threat is, as always, that the specialist language may not be good enough for every eventuality and that C++ might still be needed.

The main disincentive for developers to learn a new programming language is the perception that the time invested might not yield sufficient benefits. Kronos VST aims to present the language in a manner where interested parties can quickly evaluate the system and its relative merit, look at example programs and audition them in context.

3. KRONOS TECHNOLOGY OVERVIEW

This section presents a brief overview of the technology behind KronosVST. For detailed discussion on the programming language, the reader is referred to previous work [7] [8] [2].

3.1. Programming Language

Kronos as a programming language is a functional language[9] that deals with signals. From existing systems, Faust[5] is likely the one that it resembles the most. Both systems feature an expressive syntax and compilation to high performance native code. In the recent developments, both have converged on the LLVM[10] backend which provides just in time and optimization capabilities.

As the main differentiators, Kronos aims to offer a type system that extends the metaprogramming capabilities considerably[8]. Also, Kronos offers an unified signal model[7] that allows the user to deal with signals other than audio. Recent developments to Faust enhance its multirate model[**?**], but event-based streams remain second class. Kronos is also designed, from the ground up, for compatibility with visual programming.

On the other hand, Faust is a mature and widely used system, successfully employed in many research projects. In comparison, Kronos is still quite obscure and untested.

3.2. Libraries

The principle behind Kronos is that there are no built-in unit generators. The signal processing library that it comes with is in source form and user editable. By extension, it means that the library components cannot rely on any "magic tricks" with special compiler support. User programs are first class citizens, and can supplant or completely replace the built-in library.

Also due to the nature of the optimizing compiler built into Kronos, the library can remain simpler than most competing solutions. Functional polymorphism is employed so that signal processing components can adapt to their context. It supports generic programming, which enables a single processor implementation to adapt and optimize itself to various channel configurations and sample formats. With a little imagination this mechanism can be used to achieve various sophisticated techniques – facilities such as *currying* and *closures* in the standard library are realized by employing the generic capabilities of the compiler.

As Kronos is relatively early in its development, the standard library is continuously evolving. At the moment it provides functional programming support for the map-reduce paradigm as well as fundamentals such as oscillators, filters, delay elements and interpolators.

3.3. Code Generation

Kronos is a Just in Time compiler[11] that performs the conversion of textual source code to native machine code, to be immediately executed. In the case of a plugin version, the plugin acts as the compiler driver, feeding in the source code entered via the plugin user interface and connecting the resulting native code object to the VST infrasturcture.

3.3.1. Recent Compiler Developments

The standalone Kronos compiler is currently freely available in its beta version. This version features compilation and optimization of source code to native x86 machine code or alternatively translation into C++.

Currently, the compiler is being rewritten, with focus on compile time performance. The major improvement is in the case of

extended multirate DSP, where various buffering techniques can be employed. The language semantics seamlessly support cases where a signal frame is anything from a single sample to a large buffer of sound, but the compile time could become unacceptable as frame size was increased. The major enhancement in the new compiler version is the decoupling of vector size and compilation time, resulting from a novel redundancy algorithm in the polymorphic function specializer.

The design of the compiler is also revised and simplified, aiming to an eventual release of the source code under a free software license. As the code generator backend, the new version relies on LLVM[10] for code generation instead of a custom x86 solution; greatly increasing the number of available compile targets.

3.3.2. Optimization and Signal Rate Factorization

The aim of the Kronos project is to have the source code to look like the way humans think about signal processing, and the generated machine code to perform like that written by a decent developer. This is the goal of most compiler systems, but very hard to accomplish, as in most systems the developer needs to intervene on relatively low level of abstraction to enforce that the generated machine code is close to optimal.

Kronos aims to combine high level source code with high performance native code. The programs should be higher level than C++ to make the language easier for musicians, as well as faster to write. However, if the generated code is significantly slower, a final rewrite in C++ might still be required, defeating the purpose of rapid development.

The proposed solution is to narrow down the capabilities of the language to fulfill the requirements of signal processor development as narrowly as possible. The Kronos language is by design statically typed, strictly side effect free and deterministic, which is well suited for signal processing. This allows the compiler to make a broad range of assumptions about the code, and apply transformations that are far more radical than the ones a C++ compiler can safely do.

A further important example of DSP-specific optimization is the multirate problem. Languages such as C++ require the developer to specify a chronological order in which the program executes. In the case of multirate signal processing, this requires manual and detailed handling of various signal processors that update synchronously or in different orders.

As a result, many frameworks gloss over the multirate problem by offering a certain set of signal rates from which the user may – and has to – choose from. Traditionally, this manifests as similar-but-different processing units geared either for control or audio rate processing, or maybe handling discrete events such as MIDI. This increases the *vocabulary* an user has to learn, and makes signal processing libraries harder to maintain.

Kronos aims to solve the multirate problem, combined with the event handling problem, by defining the user programs as having no chronology. This is inherent to the functional programming model. Instead of time, the programs model data flow; data flow between processing blocks is essentially everything that signal processing boils down to.

Each data flow is semantically synchronous. Updates to the inputs of the system trigger recomputation of the results that depend on them, with the signal graph being updated accordingly. Special delay primitives in the language allow signal flow graphs to connect to previous update frames and provide for recursive loops and delay effects.

Since the inputs and the data flows that depend on them are known, the compiler is able to factorize user programs by their inputs. It can produce update entry points that respond to a certain set of system inputs, and optimize away everything that depends on inputs outside of the chosen set. Each system input then becomes an entry point that activates a certain subset of the user program – essentially, a clock source.

Because the code is generated on the fly, this data flow factorization has no performance impact, which renders it suitable to use at extreme signal rates such as high definition audio as well as sparse event streams such as MIDI. Both signal types become simple entry points that correspond to either an audio sample frame or a MIDI event.

3.3.3. Alternative Integration Strategies

In addition to plugin format, the Kronos compiler is available as a C++-callable library. There is also a command line compile server that responds to OSC[12] commands and is capable of audio i/o. The main purpose for the compile server is to act as a back end for a visual patching environment.

The compiler generates code modules that implement an object oriented interface. The user program is compiled into a code module with a set of C functions, covering allocation and initialization of a signal processor instance, as well as callbacks for plugging data into its external inputs and triggering various update routines. It is also possible to export this module as either LLVM[10] intermediate representation or C-callable object code.

4. INTERFACING USER CODE AND VST

To facilitate easy interaction between user code and the VST host application, various VST inputs and outputs are exposed as user-visible Kronos functions. These functions appear in a special package called *IO*, which the plugin generates according to the current processing context. The user entry point is a function called *Main*, which is called by the base plugin to obtain a frame of audio output.

4.1. Audio I/O

A VST plugin can be used in a variety of different audio I/O contexts. The VST3 standard allows for any number of input and output buses to and from the plugin. Each of these buses is labeled for semantic meaning and can contain an arbitrary number of channels.

The typical use for multiple input buses is to allow for sidechain input to a plugin. Multiple output buses, on the other hand, can be used to inform the host that multiple mixer channels could be allocated for the plugin output. The latter is mostly used in the context of instrument plugins.

The Kronos VST plugin exposes the main input bus as a function called *IO:Audio-In*. The return type of this function is a tuple containing all the input channels to the main bus of the plugin. The sidechain bus is exposed as *IO:Audio-Sidechain*. Both functions act as *external inputs* to the user program, propagating updates at the current VST sample rate.

Currently, only a single output bus is supported. The channel count of the output is automatically inferred from the *Main* function.

4.1.1. Audio Bus Metadata

Programs that need to know the update interval of a given data flow can interrogate it with the Kronos reactive metadata system. The sample rate of the data flow in question becomes another external input to the program with its own update context.

The Kronos VST plugin supplies update interval data for the audio buses to the user program. On sample rate changes, the reactive system can automatically update any computation results that depend on it.

4.1.2. Multichannel Datatype

Polymorphism within the Kronos VST plugin allows user programs to be flexible in their channel configuration. Many signal processors have implementations that do not vary significantly on the channel count. A processor such as an equalizer would just contain a set of identical filter instances to process a bundle of channels.

For such cases, the Kronos VST library comes with a data type that represents a multichannel sample frame. An atom of this type can be constructed from a tuple of samples by a call to *Frame:Cons*, which packages any number of channels into a single frame. These frames have arithmetic with typical vector semantics; operations are carried out for each element pair for matching multichannel frames.

The *Frame* type also has an upgrade coercion semantic. There is a specialization of the *Implicit-Coerce* function that can promote a scalar number into a multichannel duplicate. The Kronos runtime library widely calls the implicit coercion function to resolve type mismatches. This means that the compiler is able to automatically promote a scalar to a multichannel type. For example, whenever a multichannel frame is multiplied by a gain coefficient, each channel of the frame is processed without explicit instructions from the user program.

Because Kronos programs have only implicit state, this extension carries over to filter- and delay-like operations. The compiler sees a delay operation on a multichannel frame and allocates state accordingly for each channel. Therefore, the vast majority of algorithms can operate on both monophonic samples and multichannel frames without any changes to the user code.

4.2. User Interface

The VST user interface is connected to the user program via calls to *IO:Parameter*. This function receives the parameter label and range. The *IO* package constructs external inputs and triggers that are uniquely identified by all the parameter metadata, which allows for the base plugin infrastructure to read back both parameter labels and ranges.

Any external input in the user code that has the correct label and range metadata attached is considered a parameter by the base plugin. At the moment, each parameter is assigned a slider in the graphical user interface. In the future, further metadata may be added to support customizing the user interface with various widgets such as knobs or XY-pads. An example user interface is shown in Figure 1.

The parameters appear as external inputs from the user code perspective, and work just like the audio input, automatically propagating a signal clock that ticks whenever a user interaction or sequencer automation causes the parameter value to be updated.

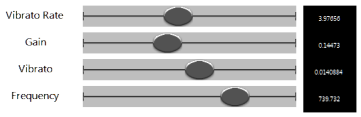

Figure 1: *An Example of a Generated VST Plugin Interface*

However, the parameters are assigned a lower reactive priority than the audio. Any computations that depend on both audio and parameter updates ignore the parameters and lock solely to audio clock.

This prevents parameter updates from causing additional output clock ticks – in effect, the user interface parameters terminate inside the audio processor at the point where their data flow merges with the audio path. This is analogous to how manually factored programs tend to cache intermediate results such as filter coefficients that result from the user interface and are consumed by the audio processor.

4.3. MIDI

MIDI input is expressed as an event stream, with a priority between parameters and audio. Thus, MIDI updates will override parameter updates but submit to audio updates.

The MIDI stream is expressed as a 32-bit integer that packs the three MIDI bytes. Accessor functions *MIDI:Event:Status()*, *MIDI:Event:A()* and *MIDI:Event:B()* can be called to retrieve the relevant MIDI bytes.

MIDI brings up a relevant feature in the Kronos multirate system; dynamic clock. MIDI filtering can be implemented by inhibiting updates that do not conform to the desired MIDI event pattern. The relevant function is *Reactive:Gate(filter sig)* which propagates the signal *sig* updates if and only if *filter* is true. A series of Gates can be used to deploy different signal paths to deal with note on, note off and continuous controller events. Later, the updates can be merged with *Reactive:Merge()*.

5. CONCLUSIONS

This paper presented an usage scenario for *Kronos*, a signal processing language. Recent developments in Kronos include a compiler rewrite from scratch. Kronos VST, a programmable plugin, is the first public release powered by the new version.

The programmable plugin allows an user to deploy and modify a signal processing program inside a digital audio workstation while it is running. It is of interest to programmers and researchers attracted to rapid prototyping or development of audio processor plugins. The instant feedback is also useful to live coders.

The Kronos VST plugin is designed to stimulate interest in the Kronos programming language. As such, it is offered free of charge to interested parties. The plugin can be used as is, or as a development tool – a finished module may be exported as C-callable object code, to be integrated in any development project.

A potential further development is an Apple Audio Unit version. The host compatibility of the plugin will be enhanced in extended field tests. Pertaining to the mainline Kronos Project, the libraries shipped with the plugin as well as the learning materials are under continued development. As the plugin and the compiler technology are very recent, the program examples at this point are

introductory. More sophisticated applications are forthcoming, to better demonstrate the capabilities of the compiler.

A. EXAMPLE PROGRAMS

A.1. Tremolo Effect

Listing 1: *Tremolo Source*

```
Saw(freq) {
    inc = IO:Audio-Clock(freq / IO:Audio-Rate())
    next = z-1(0 wrap + inc)
    wrap = next - Floor(next)
    Saw = 2 * wrap - 1
}

Main() {
  freq = IO:Parameter("Tremolo Freq" #0.1 #5 #20)
  (l r) = IO:Audio-In()
  gain  = Abs(Saw(freq))
  Main  = (l * gain r * gain)
}
```

A.2. Parametric EQ

Listing 2: *Parametric EQ Source*

```
/* Coefficient computation routine 'EQ-Coefs' omitted
    for brevity */
EQ-Band(x0 a0 a1 a2 b1 b2) {
  y1 = z-1(init(x0 #0) y0)
  y2 = z-1(init(x0 #0) y1)
  y0 = x0 - b1 * y1 - b2 * y2
  EQ-Band = a0 * y0 + a1 * y1 + a2 * y2
}

EQ-Params(num) {
  EQ-Params = (
    IO:Parameter(String:Concat("Gain " num) #-12 #0 #12)
    IO:Parameter(String:Concat("Freq " num) #20 #2000
        #20000)
    IO:Parameter(String:Concat("Q " num) #0.3 #3 #10))
}

Main() {
  input  = Frame:Cons(IO:Audio-In())
  params = Algorithm:Map(band => EQ-Coefs(EQ-Params(band
      )) [#1 #2 #3 #4])
  Main   = Algorithm:Cascade(Fitler:Biquad input params)
}
```

A.3. Reverberator

Listing 3: *Simple Mono Reverb*

```
RT60-Fb(delay rt60) {
  RT60-Fb = Crt:pow(0.001 delay / rt60)
}

Main() {
  Use Algorithm  /* for Map and Reduce */
  /* simplification: input is mono sum */
  input = Reduce(Add IO:Audio-In())
  rt60 = IO:Parameter("Reverb Time" #0.1 #3 #10) * IO:
      Audio-Rate()
  mix  = IO:Parameter("Mix" #0 #0.5 #1)

  /* settings adapted from the Schroeder paper */
  allpass-params = [(0.7 #221) (0.7 #75)]
  delay-times = [#1310 #1636 #1813 #1927]

  /* compute feedbacks and arrange delay line params */
  delay-params = Map(d => (d RT60-Fb(d rt60))
                     delay-times)

  /* compute parallel comb section */
  comb-sec  = Map((dl fb) => Delay(input dl fb)
                delay-params)

  /* mono sum comb filters and mix into input */
  sig = (1 - mix) * input +
        mix * Reduce(Add comb-sec) / 4

  Main = (sig sig)
}
```

B. REFERENCES

[1] M Puckette, "Pure data: another integrated computer music environment," in *Proceedings of the 1996 International Computer Music Conference*, 1996, pp. 269–272.

[2] Vesa Norilo, "Introducing Kronos - A Novel Approach to Signal Processing Languages," in *Proceedings of the Linux Audio Conference*, Frank Neumann and Victor Lazzarini, Eds., Maynooth, Ireland, 2011, pp. 9–16, NUIM.

[3] Richard Boulanger, *The Csound Book*, vol. 309, MIT Press, 2000.

[4] Rory Walsh, "Audio Plugin development with Cabbage," in *Proceedings of the Linux Audio Conference*, Maynooth, Ireland, 2011, pp. 47–53, Linuxaudio.org.

[5] Y Orlarey, D Fober, and S Letz, "Syntactical and semantical aspects of Faust," *Soft Computing*, vol. 8, no. 9, pp. 623–632, 2004.

[6] Robert Cerny and Fritz Menzer, "Convention e-Brief The Audio Plugin Generator: Rapid Prototyping of Audio DSP Algorithms," in *Audio Engineering Society Convention*, 2012, vol. 132, pp. 3–6.

[7] Vesa Norilo and Mikael Laurson, "Unified Model for Audio and Control Signals," in *Proceedings of ICMC*, Belfast, Northern Ireland, 2008.

[8] Vesa Norilo and Mikael Laurson, "A Method of Generic Programming for High Performance DSP," in *DAFx-10 Proceedings*, Graz, Austria, 2010, pp. 65–68.

[9] Paul Hudak, "Conception, evolution, and application of functional programming languages," *ACM Computing Surveys*, vol. 21, no. 3, pp. 359–411, 1989.

[10] C Lattner and V Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," *International Symposium on Code Generation and Optimization 2004 CGO 2004*, vol. 57, no. c, pp. 75–86, 2004.

[11] J Aycock, "A brief history of just-in-time," *ACM Computing Surveys*, vol. 35, no. 2, pp. 97–113, 2003.

[12] Matthew Wright, Adrian Freed, and Ali Momeni, "OpenSound Control: State of the Art 2003," in *Proceedings of NIME*, Montreal, 2003, pp. 153–159.