Creating LV2 Plugins with Faust

Albert Gräf
Dept. of Computer Music, Institute of Musicology
Johannes Gutenberg University (JGU) Mainz, Germany
Dr.Graef@t-online.de

Abstract

The faust-lv2 project aims to provide a complete set of LV2 plugin architectures for the Faust programming language. It currently implements generic audio and MIDI plugins with some interesting features such as Faust MIDI controller mapping, polyphonic instruments with automatic voice allocation and support for the MIDI tuning standard. You can use these architectures to quickly turn Faust programs into working LV2 audio effects and instrument plugins, ready to be run with LV2-capable DAWs such as Ardour and Qtractor. The plugin architectures and some helper scripts are now also available in the Faust distribution, and the Faust online compiler supports these as well.

Keywords

Faust, LV2, plugins, audio, MIDI.

1 Introduction

Most Linux audio users will be familiar with David Robillard's LV2 [4], the successor of the venerable LADSPA plugin standard. LV2 has been supported by major Linux DAWs such as Ardour and Qtractor for quite some time, and version 1.0 of the standard has been released in 2012, so that LV2 host and plugin authors now have a stable specification to base their work on. LV2 is much more complex than LADSPA, but it is also much more capable. In particular, it supports both audio and MIDI plugins and can thus be used to develop audio effects as well as software instruments. One of LV2's strong points is that it is extensible, so that new extensions for various special needs can be developed and deployed in LV2 hosts with (relative) ease. This makes LV2 very flexible. A number of both open source and proprietary suites of LV2 plugins have been developed or ported over to LV2, such as Calf, CAPS, TAL, drowAudio, Loomer and linuxDSP, so that Linux audio users now have a variety of high-quality plugins available to them. Nevertheless, compared to other plugin standards such as Steinberg's VST, the number of available plugins is still quite small.

The goal of the faust-lv2 project is to bring LV2 to Faust, Grame's funtional DSP programming language [3], so that LV2 plugins can be developed more easily. The interface is implemented in terms of corresponding LV2 architectures for Faust. At present two architecture (C++) files are provided, one for ordinary audio (effect type) plugins and one for polyphonic MIDI (instrument type) plugins.

We should note here that this is not the first time that LV2 has been targeted by Faust developers; projects such as Sampo Savolainen's Foo YC-20 organ emulation [5] or the Guitarix tube amplifier simulation by Hermann Meyer and others [1] utilize Faust as well. However, the goal of faust-lv2 is different. The architectures provided by faust-lv2 are completely generic and thus allow you to compile *any* Faust source and get a working LV2 plugin from it. There is a growing collection of Faust programs available, ranging from simple routing and panning plugins to sophisticated sound effects and instruments such as the Faust Synthesis Toolkit [2]. faust-lv2 enables you to use all of these in your favourite LV2 host without any further ado. Many sources will work out of the box, while others may require a few edits to make the Faust program behave nicely as an LV2 plugin. And of course faust-lv2 now also provides a convenient way to develop new sound modules and instruments in Faust and deploy them as LV2 plugins (in fact, recent posts to the Linux audio mailing lists seem to indicate that faust-lv2 is already being used that way in some projects).

This paper gives a brief overview of faust-lv2 and how you can use it to compile your own plugins. We also discuss major features and current limitations of the software and give an outlook on future work. We don't go into all the gory details here, however, so the interested reader should refer to the extended version of this paper at the faust-lv2 website for more information:

`http://faust-lv2.googlecode.com`

2 Installation and basic usage

Chances are that if you are running a recent Faust version (Faust 0.9.58 will do) then faust-lv2 is already included, so you don't have to install anything extra. faust-lv2 is also fully supported by the Faust online compiler, so you can just drop your dsp sources there and, after a few clicks, grab your ready-made LV2 plugin bundles, even without installing Faust on your computer.

Another option is to install faust-lv2 from the source distribution tarball available at the faust-lv2 project website. As a bonus this also gives you a few plugin examples you can start playing with. The package also demonstrates how you can put together your own LV2 plugin collections ready to be compiled from source. faust-lv2 is distributed as free and open-source software, licensed under the LGPL. More detailed information about the source package can be found at the project website. Briefly, if you go this route then you can compile and install faust-lv2 as follows:

```
./waf configure && ./waf && sudo ./waf
install
```

This will install both the Faust architecture files and the sample plugins under `/usr/local`, so that you can compile your own plugins and try the sample plugins in your favourite DAW. You can also just drop your Faust dsp files into the `effects` and `synths` subfolders and have them compiled and installed when running `waf`.

Alternatively, if you already have Faust installed, you can also employ two convenience scripts `faust2lv2` and `faust2lv2synth` distributed with recent Faust versions, which make the creation of LV2 bundles very easy; this is also the approach shown in the remainder of this paper.

If you want to learn exactly how this works, you should note that compiling LV2 plugins using Faust is a bit more involved than usual. This is because LV2 plugins aren't mere shared library (`.so`) files, but collections of libraries and RDF description files in Turtle syntax (`.ttl`) in their own directory. This is also known as an *LV2 bundle*. The precise steps needed to create plugin bundles with Faust are described in the extended version of this paper and in the faust-lv2 online documentation, both available at `http://faust-lv2.googlecode.com`. Developers may want to study these if they want to come up with their own build systems for compiling Faust LV2 plugins.

3 Supported plugin types

At present, faust-lv2 supports two types of plugins: the usual audio processing plugins as well as MIDI-driven software synthesizer plugins. Together these should cover most common uses in Linux audio software.

3.1 Audio plugins

Audio plugins can be added to the signal pathway in a DAW in order to realize audio effects such as amplification, panning, filtering, distortion, chorus, reverbation, etc. They are implemented by the `lv2` architecture. Please check the `lv2.cpp` file in the faust-lv2 distribution or the Faust library directory (`/usr/local/lib/faust` or `/usr/lib/faust` in most installations) if you are interested in how exactly these plugins are implemented. Plugins created with the `lv2` architecture provide the following basic features:

- Audio inputs and outputs of the Faust dsp are made available as LV2 audio input and output ports.
- Faust controls are made available as LV2 control ports with the proper label, initial value, range and (if supported by the host) step size. Both "active" (input) and "passive" (output) Faust controls are supported and mapped to the corresponding LV2 input and output ports, but note that most LV2 hosts don't provide access to LV2 control output ports (a.k.a. Faust passive controls) at this time.
- If the dsp defines any controls with corresponding MIDI mappings (`midi:ctrl` attributes in the Faust source), the plugin also provides an LV2 MIDI input port and interprets incoming MIDI controller messages accordingly.
- Plugin name, description, author and license information provided as metadata in the Faust source are translated to the corresponding fields in the LV2 manifest of the plugin.

The architectures also recognize the following Faust control metadata and set up the LV2 control port properties accordingly. Note that some of these properties rely on extensions which may not be supported by all LV2 hosts. Please refer to the LV2 documentation for a closer description of these options.

- The `unit` attribute (e.g., `[unit:Hz]`) in the Faust source is translated to a corresponding LV2 `unit` attribute. The host may then display this information in its GUI rendering of the plugin controls.
- LV2 scale points can be set with the `lv2:scalePoint` (or `lv2:scalepoint`) attribute on the Faust side. The value of this attribute in the Faust source takes the form of a list of pairs of descriptive labels and corresponding values, for instance:

  ```
  toggle = button(
  "trigger [lv2:scalepoint on 1 off 0]");
  ```

  The host may then display the given scale points with a descriptive label in its GUI.
- The `lv2:integer` attribute in the Faust source is translated to the `lv2:integer` LV2 port property, so that the control may be shown as an integer-only field in the host's GUI.
- The `lv2:hidden` or `lv2:notOnGUI` attribute maps to the LV2 `notOnGUI` port property, so that hosts honoring this property may suppress the display of this control in their GUI.

It is worth noting here that the special treatment of MIDI controllers and metadata in the Faust source can also be turned off, either with corresponding waf configure options (when using the faust-lv2 source package) or by disabling corresponding conditional compilation symbols in the `lv2.cpp` file.

For instance, consider the `chorus.dsp` example in the faust-lv2 source (cf. Fig. 1).

Compiling this program to an LV2 bundle can be done conveniently with the `faust2lv2` helper script included in recent Faust versions:

```
faust2lv2 chorus.dsp
```

This leaves a subfolder named `chorus.lv2` with the LV2 plugin (`.so` file) itself and the requisite `.ttl` files in the current directory. You can just copy this folder to `/usr/lib/lv2`, `/usr/local/lib/lv2` or any other directory on your `LV2_PATH` to have the plugin recognized by your DAW or other LV2 host program.

Besides the usual options supported by Faust compilation scripts, `faust2lv2` also understands the following target-specific options:

- `-nometa`: Normally, metadata in the Faust program (plugin description, author information, etc., as shown in the chorus example) will be translated to corresponding LV2 properties so that this data becomes available in the LV2 plugin host. When using the `-nometa` option, the metadata from the Faust source is ignored, which may be useful if you prefer to specify the corresponding information by manually editing the `manifest.ttl` file in the plugin bundle.
- `-nomidicc`: If you specify this, the plugin will not process any MIDI control data. This might be useful if the built-in MIDI control processing of the plugin gets in the way of the plugin host's own MIDI controller and automation features.
- `-uri-prefix` *URI*: This option specifies the URI prefix of the plugin. The argument must be a valid URI designation which, together with the name of the plugin uniquely identifies the plugin; please check the LV2 documentation for details. By, default, the URI prefix `http://faust-lv2.googlecode.com` will be used. You may want to replace this with the URL of the website where your plugins can be downloaded, or any other (possibly abstract) URI prefix which uniquely identifies your plugins so that they don't clash with other LV2 plugins installed on your system.
- `-dyn-manifest`: This enables dynamic manifests in the plugin, see Section 4.2 below for details. Note that to make this work, your LV2 host must support dynamic manifests. (For hosts like Ardour and Qtractor which are based on David Robillard's `lilv` library, you'll have to make sure that `lilv` was built with the `--dyn-manifest` waf configure option.)

Note that the faust-lv2 source package supports similar (as well as a bunch of other) options when configuring the package; run `./waf`

```
declare name "chorus";
declare description "stereo␣chorus␣effect";
declare author "Albert␣Graef";
declare version "1.0";

import("music.lib");

level   = hslider("level", 0.5, 0, 1, 0.01);
freq    = hslider("freq", 3, 0, 10, 0.01);
dtime   = hslider("delay", 0.025, 0, 0.2, 0.001);
depth   = hslider("depth", 0.02, 0, 1, 0.001);

tblosc(n,f,freq,mod)    = (1-d)*rdtable(n,waveform,i&(n-1)) +
                          d*rdtable(n,waveform,(i+1)&(n-1))
with {
        waveform        = time*(2.0*PI)/n : f;
        phase           = freq/SR : (+ : decimal) ~ _;
        modphase        = decimal(phase+mod/(2*PI))*n;
        i               = int(floor(modphase));
        d               = decimal(modphase);
};

chorus(dtime,freq,depth,phase,x)
                        = x+level*fdelay(1<<16, t, x)
with {
        t               = SR*dtime/2*(1+depth*tblosc(1<<16, sin, freq, phase));
};

process                 = vgroup("chorus", (left, right))
with {
        left            = chorus(dtime,freq,depth,0);
        right           = chorus(dtime,freq,depth,PI/2);
};
```

Figure 1: Faust program `chorus.dsp`.

`configure --help` in the faust-lv2 source directory to get a list of these.

3.2 MIDI plugins

faust-lv2 also fully supports instrument plugins a.k.a. software synthesizers, which can be employed as the head of the synth-effects chain in a MIDI track of your DAW. These are implemented by a separate `lv2synth` architecture.

Besides all of the features of the audio plugins described above, plugins created with the `lv2synth` architecture also provide the necessary logic to drive a polyphonic synth with automatic voice allocation. To make this work, the Faust dsp must be able to function as a monophonic synth which provides controls named `freq`, `gain` and `gate` to set the pitch (as a frequency in Hz), velocity (as a normalized value in the range 0...1) and gate (as a binary 0 or 1 value) of a note, respectively; the example below illustrates how this is done. The desired maximum number of voices can be configured with the `--nvoices` option (when using the faust-lv2 source package) or by setting the `NVOICES` macro in the `lv2synth.cpp` file accordingly. The plugin will manage at most that many instances of the Faust dsp. The actual number of voices can be changed dynamically from 1 to `NVOICES` with a special `Polyphony` control provided by the plugin.

This kind of plugin always provides a MIDI input port and interprets incoming MIDI note and pitch bend messages, as well as a number of General MIDI standard controller and system exclusive (sysex) messages, as detailed below. By default, the synth units have a pitch bend range of ±2 semitones (General MIDI default) and are tuned in equal temperament with A4 at 440 Hz. These defaults can be adjusted as needed using some of the controller and sysex messages described below.

- The “all notes off” (123) and “all sounds off” (120) MIDI controllers stop sounding notes on the corresponding MIDI channel.
- The “all controllers off” (121) MIDI controller resets the current RPN (“registered parameter number”) and data entry controllers on the corresponding MIDI channel (see below).
- The registered parameters (RPNs) 0 (pitch bend range), 1 (channel fine tuning) and 2 (channel coarse tuning) can be used to set the pitch bend range and fine/coarse master tuning on the corresponding MIDI channel in the usual way, employing a combination of the RPN (101, 100) and data entry controller pairs (6 and 38, as well as 96 and 97). Please check the MIDI specification for details.
- Universal realtime and non-realtime scale/octave tuning messages following the MIDI Tuning Standard (MTS), Section MIDI Tuning Scale/Octave Extensions, can be used to set the synth to a given octave-based tuning specified as cent offsets relative to equal temperament, which is repeated in every octave of the MIDI note range 0...127. Please check Section 4.1 below for further details.

For instance, consider the `organ.dsp` example from the faust-lv2 distribution (cf. Fig. 2).

Note the `freq`, `gain` and `gate` controls which turn this Faust dsp into a monophonic synthesizer. Polyphony with automatic allocation of up to `NVOICES` voices is implemented in the plugin architecture. Also note the `midi:ctrl 10` attribute in the label of the `pan` control. This is Faust control metadata which denotes that MIDI controller 10 (the MIDI pan position controller) should be associated with this control value. The plugin architecture will add a MIDI input port and the required MIDI controller processing to the plugin in order to implement this. (Whether your LV2 host actually passes such MIDI controller messages to the plugin depends on the host, though.)

Compiling the plugin works as with audio plugins, using `faust2lv2synth` in lieu of `faust2lv2`:

```
faust2lv2synth organ.dsp
```

You’ll get an `organ.lv2` folder which you simply copy to your LV2 library directory to have the plugin recognized. In addition to the target-specific options recognized by `faust2lv2`, `faust2lv2synth` also lets you specify the desired maximum number of voices with the `-nvoices` option which takes the desired number of voices as its argument (the default is 16). In principle, any positive integer can be specified here, but the feasible range will of course depend on how much cpu power you have to spare.

Figure 3 shows the `organ.lv2` instrument along with some other Faust-generated LV2 plugins running in Qtractor.

4 Special features and limitations

In this section we discuss some notable features and limitations of the Faust LV2 implementation. The generated plugins should work with any LV2 1.0 compatible host which supports either the `urid` or the older `uri-map` extension (most if not all LV2 hosts will have this). MIDI input requires a host capable of delivering MIDI events through LV2’s `event` extension. faust-lv2 also supports the `dynmanifest` extension (see Section 4.2 below), but this is an optional feature which is by no means required for proper operation of the plugins.

4.1 MIDI tunings

The MTS support of instrument plugins mentioned in the previous section calls for a more detailed explanation. The general format of the supported MTS messages is as follows (using hexadecimal notation):

```
f0 7f/7e id 08 08/09 bb bb bb tt ... tt f7
```

Note that the `f0 7f` and `f0 7e` headers are used to denote a universal realtime and non-realtime sysex message, respectively, and the final `f7` byte terminates the message. Both types of messages will take effect immediately, but the realtime form will also change the frequencies of already sounding notes. The device id can be any 7-bit value from `00` to `7f` and will be ignored, so that the unit will always respond to these messages, no matter which device id is specified. The following `08` id denotes an MTS message, followed either by the `08` subid to denote 1-byte, or the `09` subid to denote 2-byte encoding (see below).

The `lv2synth` architecture keeps track of separate tunings for different MIDI channels. The three `bb` bytes together specify the bitmask of MIDI channels the message applies to, most significant byte first; the bitmask `03 7f 7f` thus sets the tuning for all MIDI channels, while the

```
declare name "organ";
declare description "a␣simple␣additive␣synth";
declare author "Albert␣Graef";
declare version "1.0";

import("music.lib");

// control variables

vol     = hslider("vol", 0.3, 0, 10, 0.01);     // %
pan     = hslider("pan␣[midi:ctrl␣10]", 0.5, 0, 1, 0.01); // %
attack  = hslider("attack", 0.01, 0, 1, 0.001); // sec
decay   = hslider("decay", 0.3, 0, 1, 0.001);   // sec
sustain = hslider("sustain", 0.5, 0, 1, 0.01);  // %
release = hslider("release", 0.2, 0, 1, 0.001); // sec
freq    = nentry("freq", 440, 20, 20000, 1);    // Hz
gain    = nentry("gain", 0.3, 0, 10, 0.01);     // %
gate    = button("gate");                       // 0/1

// relative amplitudes of the different partials

amp(1)  = hslider("amp1", 1.0, 0, 3, 0.01);
amp(2)  = hslider("amp2", 0.5, 0, 3, 0.01);
amp(3)  = hslider("amp3", 0.25, 0, 3, 0.01);

// additive synth: 3 sine oscillators with adsr envelop

partial(i) = amp(i+1)*osc((i+1)*freq);

process = sum(i, 3, partial(i))
  * (gate : vgroup("1-adsr", adsr(attack, decay, sustain, release)))
  * gain : vgroup("2-master", *(vol) : panner(pan));
```

Figure 2: Faust program `organ.dsp`.

bitmask `00 00 01` only affects the tuning of the first MIDI channel.

The `tt` bytes specify the tuning itself, as a sequence of 12 tuning offsets for the notes C, C♯, D, etc., thru B. In the one-byte encoding (subid `08`), each tuning offset is a 7 bit value in the range `00`...`7f`, with `00`, `40` and `7f` denoting -64, 0 and +63 cents, respectively. Thus equal temperament is specified using twelve `40` bytes, and a quarter comma meantone tuning could be denoted, e.g., as `4a 32 43 55 3d 4e 36 47 2f 40 51 39`. The two-byte encoding (subid `09`) works in a similar fashion, but provides both an extended range and better resolution. Here each tuning offset is specified as a 14 bit value encoded as two data bytes (most significant byte first), mapping the range 0...16384 to -100..+100 cents with the center value 8192 (`40 00`) denoting 0 cents. Please check the MMA's MIDI Tuning Standard document for details.

Using these messages you can tune a Faust synth in any octave-based temperament you like, provided that your DAW supports sending sysex messages to LV2 instrument plugins. (Qtractor allows you to enter the sysex messages in its "Buses" dialog. Ardour 3 doesn't support editing sysex messages yet, but it is still under development, so there is hope that this will be fixed before the final release.) A large repository of historical and contemporary microtonal tunings is available on the website of the Scala program; writing a little script to convert the Scala tuning files to binary sysex files in one of the formats described above should be a fun exercise for Linux audio developers.

4.2 Dynamic manifests

Plugins created with faust-lv2 support the LV2 dynamic manifest extension, so that all requisite information about the plugin's name, author, ports, etc. can also be included in the plugin module (`.so` file) itself. This also cuts down the compilation time since the manifest doesn't have to be generated from the plugin executable

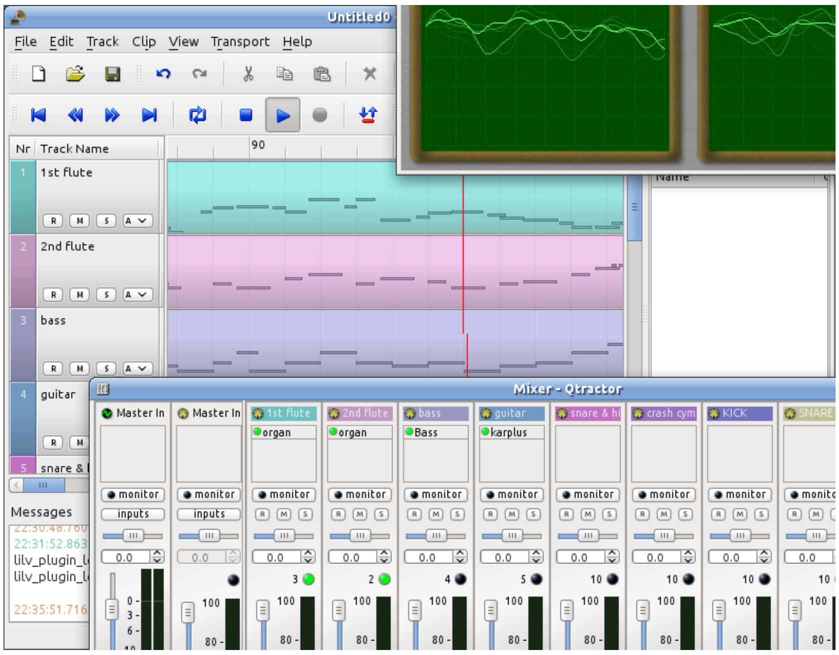

Figure 3: faust-lv2 plugins running in Qtractor.

beforehand.

Note that in order to provide better compatibility with current LV2 hosts, which usually don't have this extension enabled, this feature isn't used by default in the provided build scripts. But you can select it by configuring faust-lv2 with the `--dyn-manifest` option, when using the faust-lv2 source package, or with the `-dyn-manifest` option of the `faust2lv2` and `faust2lv2synth` scripts included in recent Faust versions.

4.3 GUIs

One major limitation of faust-lv2 is that it does not support custom plugin GUIs in the current version. This might be added in the future, but for the time being you'll have to rely on the LV2 host to display a GUI for the control elements. Both Ardour and Qtractor do a reasonably good job at this. (However, the hierarchical layout of GUI controls prescribed by the Faust source is lost in the generic plugin GUIs provided by LV2 hosts.)

5 Future work

While the LV2 plugin implementation of the Faust LV2 architectures is fully functional and reasonably complete already, there are ways in which they could be further improved. Some items which are worth further consideration are listed below.

- Add improvements for smoother playback. In particular, the polyphony control provided by `lv2synth.cpp` is fairly disruptive right now, as it simply resets all voices each time the control changes.

- Add custom plugin GUIs which honor the hierarchical GUI layout defined in the Faust source. Corresponding code is readily available in other Faust architectures such as jack-gtk and jack-qt, but would need to be integrated with the LV2 architectures and the LV2 GUI extension.

- Update the architectures so that they employ the new atom-based interface for MIDI input instead of the older (and now deprecated) LV2 Event extension.

- Add support for the new LV2 Time extension, which provides transport information such as the current position, tempo and time signature to a plugin.

- Implement MIDI output for passive Faust controls. It's unclear if and how existing LV2 hosts would process such data, however, so there's still some research to be done there.

Besides these, LV2's extensible nature might call for completely new plugin types in the future. While the audio and instrument plugin types implemented by faust-lv2 seem to cover the requirements of the current generation of DAWs, it is good to know that Faust's and LV2's modular nature will make it easy to support new types of audio applications when they emerge.

References

H. Meyer, A. Degert, and P. Shorthose. Guitarix tube amplifier simulation for Jack/Linux. `http://guitarix.sourceforge.net`, 2013.

R. Michon and J. O. Smith. Faust-STK: a set of linear and nonlinear physical models for the Faust programming language. In G. Peeters, editor, *Proceedings of the 11th International Conference on Digital Audio Effects (DAFx-11)*, pages 199–204, Paris, 2011. IRCAM.

Y. Orlarey, D. Fober, and S. Letz. FAUST : an efficient functional approach to DSP programming. In G. Assayag and A. Gerzso, editors, *New Computational Paradigms for Computer Music*. Editions Delatour France, 2009.

D. Robillard. LV2 1.2.0 Specifications. `http://lv2plug.in/ns/`, 2013.

S. Savolainen. Emulating a combo organ using Faust. In *Proceedings of the 9th International Linux Audio Conference*, pages 21–29, Utrecht, 2010. Hogeschool voor de Kunsten.