Polyphony, sample-accurate control and MIDI support for FAUST DSP using combinable architecture files

Stéphane Letz, Yann Orlarey, Dominique Fober, Romain Michon

Polyphony, sample-accurate control and MIDI support for FAUST DSP using combinable architecture files

Stéphane LETZ, Yann ORLAREY, Dominique FOBER
GRAME
Centre National de Création Musicale
11 Cours de Verdun (Gensoul)
69002, Lyon
France
{letz,orlarey,fober}@grame.fr

Romain MICHON
CCRMA
Stanford University
Stanford, CA 94305-8180
USA
rmichon@ccrma.stanford.edu

Abstract

The FAUST architecture files ecosystem is regularly enriched with new targets to deploy Digital Signal Processing (DSP) programs. This paper presents recently developed techniques to expand the standard *one DSP source, one program or plugin* model, and to better control parameter changes during the audio computation. Sample accurate control and polyphonic instruments definition have been introduced, and will be explained particularly in the context of MIDI control.

Keywords

FAUST, DSP programming, audio, MIDI

1 Introduction

FAUST is a functional programming language specifically designed for real-time signal processing and synthesis. From a high-level specification, its compiler typically generates the DSP computation as a C++ class[1] to be wrapped by so-called *architecture files* and connected to the external world.

1.1 Audio and UI Architecture Files

Native audio drivers are developed as subclasses of a base *audio* class, controllers as subclasses of a base *UI* class. Typical Graphical User Interface architectures are based on well established frameworks like QT[2] or JUCE[3], and allow to display a ready to use window with sliders, text zones and buttons. Audio and UI parts are finally combined with the actual DSP computation to produce the final audio application or plugin (see Figure 1).

Non graphical controllers can also be defined as subclasses of UI, simply by ignoring the layout description[4], and just keeping the actual controls definition (with their name, default value, value range etc.). OSCUI and httpdUI classes [1] typically follow this strategy.

New architecture files have been regularly added to the already rich FAUST ecosystem, to expand the variety of possible targets for the DSP code.

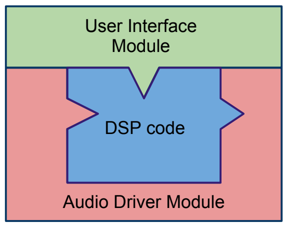

Figure 1: DSP code is generated by the compiler, audio and UI codes are added from the generic architecture files.

1.2 Macro Construction of DSP Components

The FAUST program specification is usually entirely done in the language itself. But in some specific cases it may be useful to develop *separated DSP components* and *combine* them in a more complex setup.

Since taking advantage of the huge number of already available UI and audio architecture files is important, keeping the same `dsp` API is preferable[5], so that more complex DSP can be controlled and audio rendered the usual way:

```
class dsp {

   public:
       .....
       virtual int getNumInputs() {}
       virtual int getNumOutputs() {}
       virtual void buildUserInterface(UI* ui) {}
       virtual void init(int samplingRate) {}
```

[1]The faust2 development branch can also generate C, LLVM IR, WebAssemby etc. target languages.

[2]`http://doc.qt.io`

[3]`https://www.juce.com/doc/classes`

[4]Typically done using hgroup, vgroup or tgroup in the DSP source code.

[5]Only part of the complete DSP API is presented here.

```
        virtual void compute(int count,
                 FAUSTFLOAT** inputs,
                 FAUSTFLOAT** outputs) {}
        .....
};
```

Extended DSP classes will typically subclass the `dsp` root class and override part of its API.

This paper shows how this approach can be used to define new extended and combinable `dsp` classes. Section 2 describes tools to *combine* separately developed DSP. Section 3 explains how *sample accurate* parameter control of a given DSP can be done using the new `timed_dsp` class, and when it needs to be used.

Section 4 presents the model used to deploy polyphonic instruments, section 5 presents how the previously presented components can be used together in the context of MIDI control, and finally the conclusion tries to enlarge this work in a more general analysis of the FAUST compiler generated code.

2 Combining DSP

2.1 Dsp Decorator Pattern

A `dsp_decorator` class, subclass of the root `dsp` class has first been defined. Following the decorator design pattern[6], it allows behavior to be added to an individual object, either statically or dynamically.

The extended DSP class hierarchy is shown in Figure 2. As an example of the decorator pattern, the `timed_dsp` class allows to decorate a given DSP with sample accurate control capability as explained in section 3.

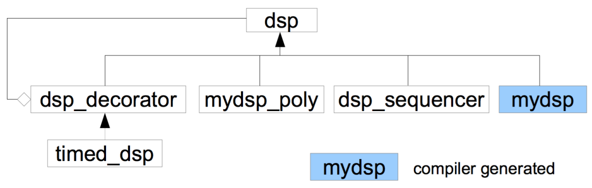

Figure 2: DSP classes diagram

2.2 Combining DSP Components

A few additional *macro construction* classes, subclasses of the root `dsp` class have been defined in the public *faust/dsp/dsp-combiner.h* header file:

[6]`https://en.wikipedia.org/wiki/Decorator_pattern`

- the `dsp_sequencer` class combines two DSP in sequence, assuming that the number of outputs of the first DSP equals the number of input of the second one. Its `buildUserInterface` method is overloaded to group the two DSP in a tabgroup, so that control parameters of both DSPs can be individually controlled[7]. Its `compute` method is overloaded to call each DSP compute in sequence, using an intermediate output buffer produced by first DSP as the input one given to the second DSP.

- the `dsp_parallelizer` class combines two DSP in parallel. Its `getNumInputs/getNumOutputs` methods are overloaded by correctly reflecting the input/output of the resulting DSP as the sum of the two combined ones. Its `buildUserInterface` method is overloaded to group the two DSP in a tabgroup, so that control parameters of both DSP can be individually controlled. Its `compute` method is overloaded to call each DSP compute, where each DSP consuming and producing its own number of input/output audio buffers taken from the method parameters.

3 Sample Accurate Control

DSP audio languages usually deal with several timing dimensions when treating control events and generating audio samples. For performance reasons, systems maintain separated audio rate for samples generation and control rate for asynchronous messages handling.

The audio stream is most often computed by blocks, and control is updated between blocks. To smooth control parameter changes, some language chose to interpolate parameter values [7] between blocks.

In some cases control may be more finely interleaved with audio rendering [8], and some languages [9] simply choose to interleave control and sample computation at sample level.

Although the FAUST language permits the description of sample level algorithms (like recursive filters etc.), FAUST generated DSP are usually computed by blocks. Underlying audio architectures usually give a fixed size buffer over and over to the DSP `compute` method which consumes and produces audio samples.

[7]Typically using any UI object.

3.1 Control to DSP Link

In the current version of the FAUST generated code, the primary connection point between the control interface and the DSP code is simply a memory zone. For control inputs, the architecture layer continuously write values in this zone, which is then *sampled* by the DSP code at the beginning of the `compute` method, and used with the same values during the entire call. Because of this simple control/DSP connexion mechanism, the *most recent value* is seen by the DSP code.

Similarly for control outputs[8], the DSP code inside the `compute` method possibly write several values at the same memory zone, and the *last value* only will be seen by the control architecture layer when the method finishes.

Although this behaviour is satisfactory for most use-cases, some specific usages need to handle the *complete* stream of control values with *sample accurate* timing. For instance keeping all control messages and handling them at their exact position in time is critical for proper MIDI clock synchronisation.

3.2 Time-Stamped Control

The first step consists in extending the architecture control mechanism to deal with *time-stamped* control events. Note that this requires the underlying event control layer to support this capability. The native MIDI API for instance is usually able to deliver time-stamped MIDI messages.

The next step is to keep all time-stamped events in a *time ordered* data structure to be continuously written by the control side, and read by the audio side.

Finally the sample computation has to take account of all queued control events, and correctly change the DSP control state at successive points in time.

3.3 Slices Based DSP Computation

With time-stamped control messages, changing control values at precise sample indexes on the audio stream becomes possible. A generic *slices based* DSP rendering strategy has been implemented in the `timed_dsp` class.

A ring-buffer is used to transmit the stream of time-stamped events from the control layer to the DSP one. In the case of MIDI control case for instance, the ring-buffer is written with a pair containing the time-stamp expressed in samples and the actual MIDI message each time one is received. In the DSP `compute` method, the ring-buffer will be read to handle all messages received during the previous audio block.

Since control values can change several times inside the same audio block, the DSP `compute` cannot be called only once with the total number of frames and the complete inputs/outputs audio buffers. The following strategy has to be used:

- several slices are defined with control values changing between consecutive slices.
- all control values having the same time-stamp are handled together, and change the DSP control internal state. The slice is computed up to the next control parameters time-stamp until the end of the given audio block is reached.
- in the Figure 3 example, four slices with the sequence of c1, c2, c3, c4 frames are successively given to the DSP `compute` method, with the appropriate part of the audio input/output buffers. Control values (appearing here as [v1,v2,v3], then [v1,v3], then [v1], then [v1,v2,v3] sets) are changed between slices.

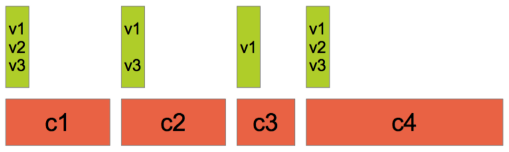

Figure 3: Audio block slice-based computation

Since time-stamped control messages from the previous audio block are used in the current block, control messages are aways handled with one audio buffer latency.

4 Polyphonic Instruments

Directly programing polyphonic instruments in FAUST is perfectly possible. It is also needed if very complex signal interaction between the different voices have to be described[9].

But since all voices would always be computed, this approach could be too CPU costly for simpler or more limited needs. In this case

[8]Using *bargraph* kind of UI elements.

[9]Like sympathetic strings resonance in a physical model of a piano for instance.

describing a single voice in a FAUST DSP program and externally combining several of them with a special *polyphonic instrument aware* architecture file is a better solution. Moreover, this special architecture file takes care of dynamic voice allocations and control MIDI messages decoding and mapping.

4.1 Polyphonic Ready DSP Code

By convention FAUST architecture files with polyphonic capabilities expect to find control parameters named *freq*, *gain* and *gate*. The metadata `declare nvoices "8";` kind of line with a desired value of voices can be added in the source code.

In the case of MIDI control, the *freq* parameter (which should be a frequency) will be automatically computed from MIDI note numbers, *gain* (which should be a value between 0 and 1) from velocity and *gate* from *keyon/keyoff* events. Thus, gate can be used as a trigger signal for any envelope generator, etc.

4.2 Using the mydsp_poly class

The single voice has to be described by a FAUST DSP program, the `mydsp_poly` class is then used to combine several voices and create a polyphonic ready DSP:

- the *faust/dsp/poly-dsp.h* file contains the definition of the `mydsp_poly` class used to wrap the DSP voice into the polyphonic architecture. This class maintains an array of `dsp` type of objects, manage dynamic voice allocations, control MIDI messages decoding and mapping, mixing of all running voices, and stopping a voice when its output level decreases below a given threshold.
- as a sub-class of DSP, the `mydsp_poly` class redefines the `buildUserInterface` method. By convention all allocated voices are grouped in a global *Polyphonic* tabgroup. The first tab contains a *Voices* group, a master like component used to change parameters on all voices at the same time, with a *Panic* button to be used to stop running voices[10], followed by one tab for each voice. Graphical User Interface components will then reflect the multi-voices structure of the new polyphonic DSP (Figure 4).

[10]An internal control grouping mechanism has been defined to automatically dispatch a user interface action done on the master component on all linked voices.

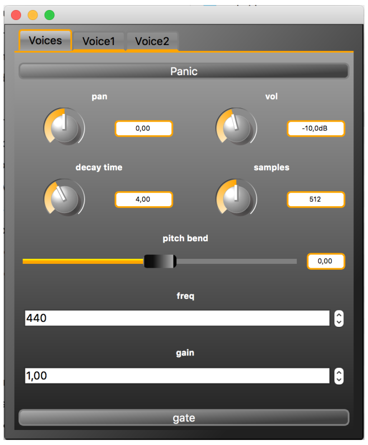

Figure 4: Extended multi-voices GUI interface

The resulting polyphonic DSP object can be used as usual, connected with the needed audio driver, and possibly other UI control objects like OSCUI, httpdUI, etc. Having this new UI hierarchical view allows complete OSC control of each single voice and their control parameters, but also all voices using the master component.

The following OSC messages reflect the same DSP code either compiled normally, or in polyphonic mode (only part of the OSC hierarchies are displayed here):

```
// Mono mode

/0x00/0x00/vol f -10.0
/0x00/0x00/pan f 0.0

// Polyphonic mode

/Polyphonic/Voices/0x00/0x00/pan f 0.0
/Polyphonic/Voices/0x00/0x00/vol f -10.0
...
/Polyphonic/Voice1/0x00/0x00/vol f -10.0
/Polyphonic/Voice1/0x00/0x00/pan f 0.0
...
/Polyphonic/Voice2/0x00/0x00/vol f -10.0
/Polyphonic/Voice2/0x00/0x00/pan f 0.0
...
```

The polyphonic instrument allocation takes the DSP to be used for one voice[11], the desired number of voices, the *dynamic voice allocation* state[12], and the *group* state which controls if separated voices are displayed or not (Figure 4):

```
    DSP = new mydsp_poly(dsp, 2, true, true);
```

[11]The DSP object will be automatically cloned in the mydsp_poly class to create all needed voices.

[12]Voices may be always running, or dynamically started/stopped in case of MIDI control.

With the following code, note that a polyphonic instrument may be used outside of a MIDI control context, so that all voices will be always running and possibly controlled with OSC messages for instance:

```
DSP = new mydsp_poly(dsp, 8, false, true);
```

4.3 Controlling the Polyphonic Instrument

The `mydsp_poly` class is also ready for MIDI control and can react to *keyon/keyoff* and *pitchwheel* messages. Other MIDI control parameters can directly be added in the DSP source code.

4.4 Deploying the Polyphonic Instrument

Several architecture files and associated scripts have been updated to handle polyphonic instruments:

As an example on OSX, the script `faust2caqt foo.dsp` can be used to create a polyphonic CoreAudio/QT application. The desired number of voices is either declared in a `nvoices` metadata or changed with the `-nvoices num` additional parameter[13]. MIDI control is activated using the `-midi` parameter.

The number of allocated voices can possibly be changed at runtime using the `-nvoices` parameter to change the default value (so using `./foo -nvoices 16` for instance).

Several other scripts have been adapted using the same conventions.

4.5 Polyphonic Instrument with a Global Output Effect

Polyphonic instruments may be used with an output effect. Putting that effect in the main Faust code is not a good idea since it would be instantiated for each voice which would be very inefficient. This is a typical use case for the `dsp_sequencer` class previously presented with the polyphonic DSP connected in sequence with a unique global effect (Figure 5).

`faustcaqt inst.dsp -effect effect.dsp` with inst.dsp and effect.dsp in the same folder, and the number of outputs of the instrument matching the number of inputs of the effect, has to be used. A `dsp_sequencer` object will be created to combine the polyphonic instrument in sequence with the single output effect.

[13]-nvoices parameter takes precedence over the metadata value.

Polyphonic ready *faust2xx* scripts will then compile the polyphonic instrument and the effect, combine them in sequence, and create a ready to use DSP.

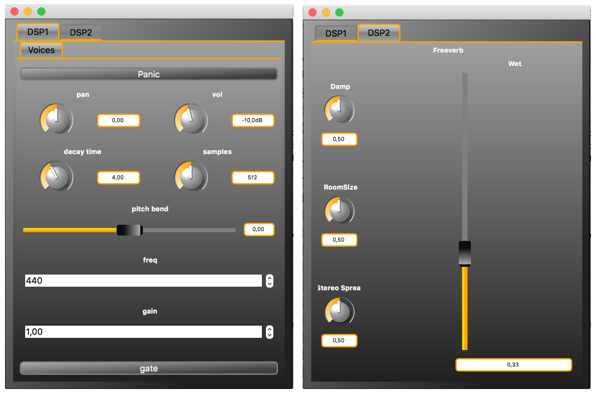

Figure 5: Polyphonic instrument with output effect GUI interface: left tab window shows the polyphonic instrument with its *Voices* group only, right tab window shows the output effect.

5 MIDI Control

MIDI control connects DSP parameters with MIDI messages (in both directions), and can be used to trigger polyphonic instruments.

5.1 MIDI Messages Description in the DSP Source Code

MIDI control messages are described as metadata in UI elements. They are decoded by a new `MidiUI` class, subclass of UI, which parses incoming MIDI messages and updates the appropriate control parameters, or sends MIDI messages when the UI elements (sliders, buttons...) are moved.

5.2 Defined Standard MIDI messages

A special `[midi:xxx yyy...]` metadata needs to be added in the UI element. Here is the description of three common MIDI messages:

- `[midi:keyon pitch]` in a slider or bargraph will map the UI element value to keyon velocity in the (0, 127) range. When used with a button or checkbox, 1 will be mapped to 127, 0 will be mapped to 0,
- `[midi:keyoff pitch]` in a slider or bargraph will map the UI element value to keyoff velocity in the (0,127) range. When used with a button or checkbox, 1 will be mapped to 127, 0 will be mapped to 0,

- `[midi:ctrl num]` in a slider or bargraph will map the UI element value to (or from) (0, 127) range. When used with a button or checkbox, 1 will be mapped to 127, 0 will be mapped to 0.

The full description of supported MIDI messages is now part of the FAUST documentation.

5.3 MIDI Clock Synchronization

MIDI clock based synchronization can be used to slave a given FAUST program, using the sample accurate control mechanism described in section 3. The following three messages have to be used:

- `[midi:start]` in a button or checkbox will trigger a value of 1 when a start MIDI message is received
- `[midi:stop]` in a button or checkbox will trigger a value of 0 when a stop MIDI message is received
- `[midi:clock]` in a button or checkbox will deliver a sequence of successive 1 and 0 values each time a clock MIDI message is received, seen by FAUST code as a square command signal, to be used to compute higher level information.

A typical FAUST program will then use the MIDI clock command signal to possibly compute the Beat Per Minutes (BPM) information, or for any synchronization need it may have.

Here is a simple example of a sinusoid generated which a frequency controlled by the MIDI clock stream[14], and starting/stopping when receiving the MIDI start/stop messages:

```
import("stdfaust.lib");

// square signal (1/0), changing state
// at each received clock
clocker = checkbox("MIDI clock[midi:clock]");

// ON/OFF button controlled
// with MIDI start/stop messages
play = checkbox("On/Off [midi:start][midi:stop]");

// detect front
front(x) = (x-x') != 0.0;

// count number of peaks during one second
freq(x) = (x-x@ma.SR) : + ~ _;

process = os.osc(8*freq(front(clocker))) * play;
```

[14]Using an external MIDI clock generator and changing its tempo allow to precisely control the sinusoid frequency.

Note that the described sample accurate MIDI clock synchronization model can currently only be used at input level. Because of the simple memory zone based connection point between the control interface and the DSP code, output controls (like bargraph) cannot generate a stream of control values. Thus a reliable MIDI clock generator cannot be implemented with the current approach.

5.4 MIDI Classes

A `midi` base class defining MIDI messages decoding/encoding methods has been developed. A `midi_hander` subclass implements actual decoding. Several concrete implementations based on native API have been written (Figure 6) and can be found in the *faust/midi* folder.

Depending on the used native MIDI API, event time-stamps are either expressed in absolute time or in frames. They are converted to offsets expressed in samples relative to the beginning of the audio buffer.

Connected with the new `MidiUI` class, subclass of UI, they allow a given DSP to be controlled with incoming MIDI messages or possibly send MIDI messages when its internal control state changes.

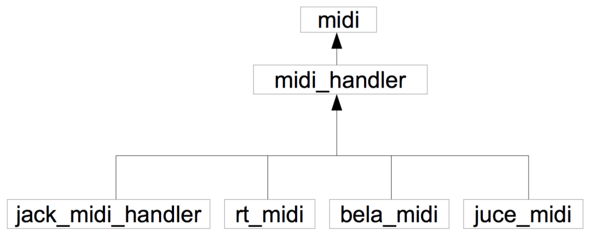

Figure 6: MIDI classes diagram

In the following piece of code, a `MidiUI` object is created and connected to a `rt_midi` [5] MIDI message handler, then given as parameter to the standard `buildUserInterface` to control the DSP parameters:

```
rt_midi midi_handler("MIDI");
MidiUI midiinterface(&midi_handler);
DSP->buildUserInterface(&midiinterface);
```

6 Deployment

The extended architecture files have been presented and used in the context of *statically generated and compiled DSP*, that is generating C++ code from FAUST, then compiling the resulting code in executable applications or plugins. They have been deployed in several *faust2xx* scripts and especially in *faust2api* presented in [6].

Note that they can also be used with dynamically *libfaust* generated DSP[15] as in particular in FaustLive [3] standalone just-in-time FAUST compiler, or in *faustgen~* Max/MSP external object.

7 Conclusion

The sample accurate control model could easily be adapted to work with MIDI controllable plugins like LV2 instruments[16], so that MIDI clock synchronization could be used.

Expanding the polyphonic and sample accurate control model over the network in the *libfaustremote* [4] library is still in progress.

As a general concluding remark, a deeper rethinking of the control/DSP connection model in the FAUST compiled code will have to be done. As explained in section 3, control and DSP computation interaction is somewhat limited in the current model of the generated code.

The described solution stays at the architecture layer level with some limitations. Although sample accurate control for inputs can be done using the presented *slices based DSP computation*, this strategy does not help to properly retrieve the stream of control output values.

A cleaner approach would be to extend the model of control signals to be *a list of time-stamped values*, so that the `compute` would handle a slice of time-stamped input controls (kept from the previous block), and possibly produces a slice of time-stamped output controls. Having this more general strategy at the code generation level still has to be developed.

Acknowledgments

This work has been done under the FEEVER project [ANR-13-BS02-0008] supported by the “Agence Nationale pour la Recherche".

References

[1] D. Fober, Y. Orlarey, and S. Letz, “FAUST Architectures Design and OSC Support", *IRCAM*, (Ed.): Proc. of the 14th Int. Conference on Digital Audio Effects (DAFx-11), pp. 231-216, 2011.

[2] Orlarey, Y., Fober, D., and Letz, S. (2009), “FAUST: an efficient functional approach to DSP programming." New Computational Paradigms for Computer Music, 290.

[3] S. Denoux, S. Letz, Y. Orlarey and D. Fober, “FAUSTLIVE Just-In-Time Faust Compiler... and much more." Linux Audio Conference, 2014.

[4] S. Letz, S. Denoux and Y. Orlarey, “Audio Rendering/Processing and Control Ubiquity ? a Solution Built Using the Faust Dynamic Compiler and JACK/NetJack." ICMC/SMC, Athenes 2014.

[5] “RtMidi framework online documentation" `http://www.music.mcgill.ca/~gary/rtmidi/`

[6] R.Michon, J.Smith, C.Chafe, S. Letz and Y. Orlarey, “faust2api: a Comprehensive API Generator for Android and iOS." Linux Audio Conference, 2017.

[7] J.McCartney, “Rethinking the Computer Music Language: SuperCollider." Computer Music Journal, Winter 2002.

[8] P.Donat-Bouillud, JL.Giavitto, A.Cont, N.Schmidt and Y.Orlarey, “Embedding native audio-processing in a score following system with quasi sample accuracy." ICMC, Utrecht 2016.

[9] G. Wang, P. R. Cook, and S. Salazar, “Chuck: A strongly timed computer music language." Computer Music Journal, 2016.

[15] Dynamically libfaust generated DSP are objects of llvm_dsp or interpreter_dsp types, subclasses of the dsp root class with the same API.

[16] `http://lv2plug.in/doc/html/`