

# **DOCUMENTATION FAUST & BELA**

# Index

MANUAL.....	3
Using FAUST with BELA, manual.....	3
Faust2bela script.....	7
Script for copy of files into BELA.....	8
Installing BELA.....	9
The Driver on the host computer :.....	9
BELA – SD card install :.....	9
MEASURES.....	10
Latency measures.....	10
FAUST performance on BELA.....	14
Compilation options.....	16
DEMO FILES :.....	17
Synthesizers :.....	17
AdditiveSynth.dsp 78% CPU (6 voices, without FX).....	17
FMSynth2.dsp 60% CPU (8 voices, with FX).....	17
simpleSynth.dsp 59% CPU (8 voices, with FX).....	18
WaveSynth.dsp 70% CPU (8 voices, with FX).....	18
L'effect for the synthesizers :.....	19
simpleFX.dsp.....	19
Effects :.....	19
crossDelay2.dsp 28% CPU.....	19
FXChaine2.dsp 64% CPU.....	20
granulator.dsp 23% CPU.....	20
GrainGenerator.dsp 33% CPU.....	21
repeater.dsp 13% CPU.....	21
Some small FAUST functions :.....	21
Scanner :.....	21
WaveTable generator :.....	21
RS Latch :.....	22
counterUpReset :.....	22

# MANUAL

## Using FAUST with BELA, manual

### 1- FAUST program.

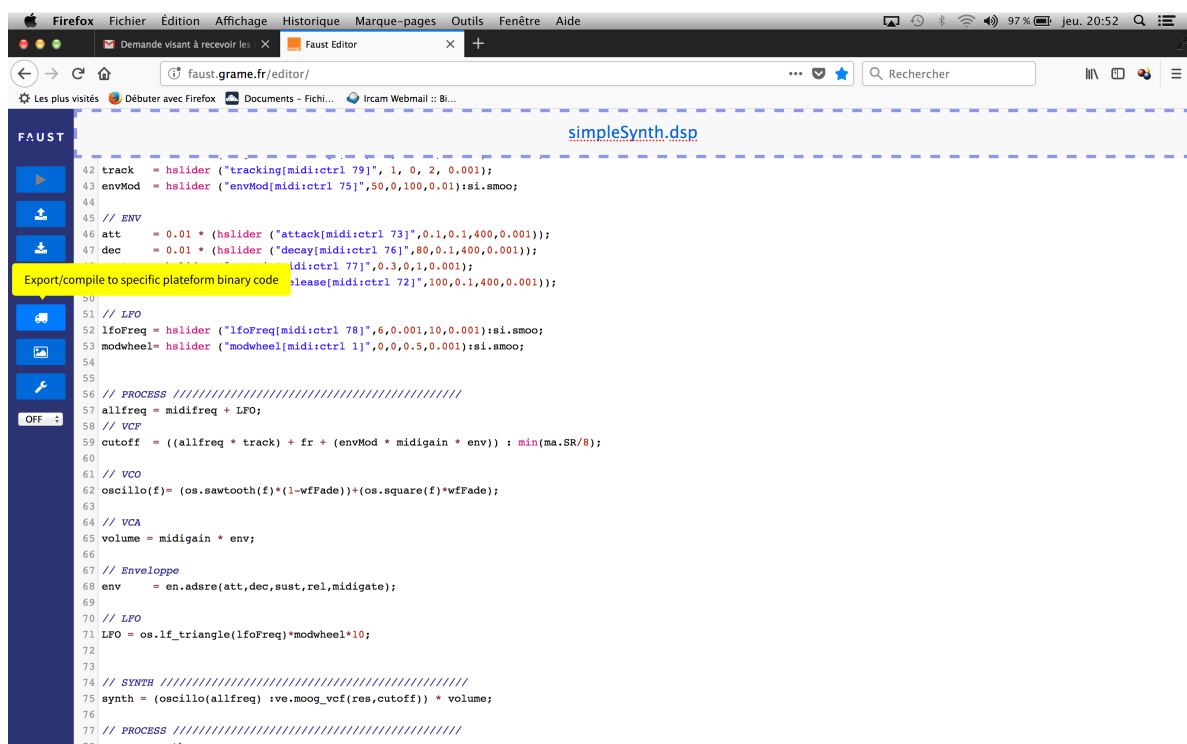
Go to the online editor : <http://faust.grame.fr/editor/> and write the FAUST code that will have to be inserted in the BELA.

To indicate that an analog input of the BELA must control a parameter, it is necessary to add in the construction of the widget [BELA: ANALOG\_'number'].

Example :

```
frequence = hslider("frequence[BELA: ANALOG_0]", 440, 20, 10000, 0.1);
```

(It's possible, for example, to check the result by compiling for a platform other than BELA).



Faust Editor web page, with export button

### 2- Export.

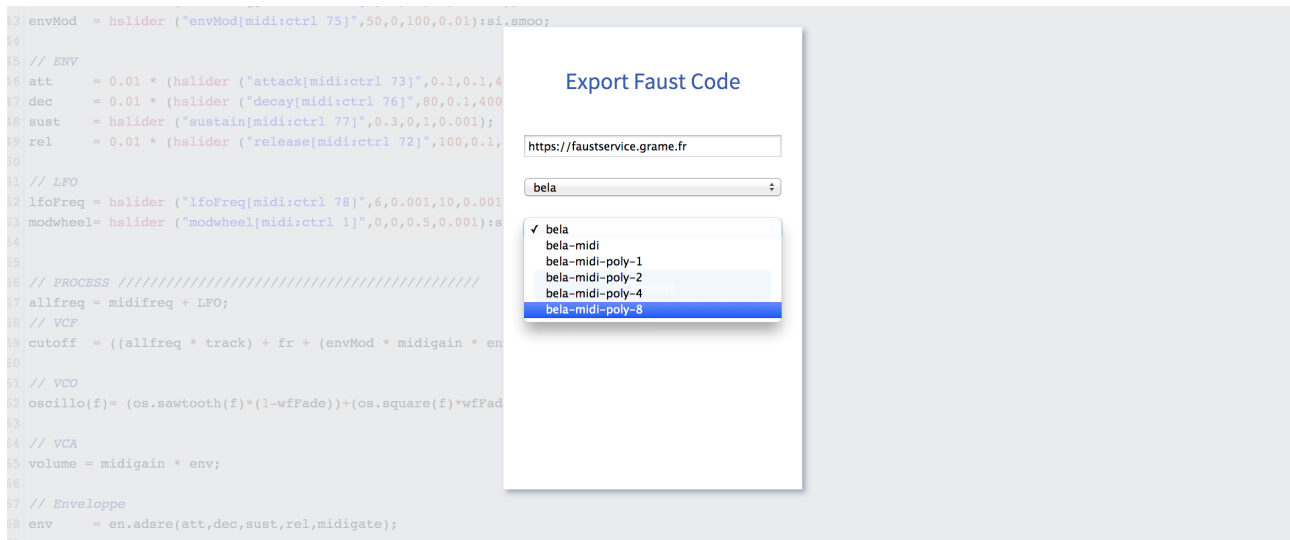
Once the FAUST program is complete and functional, choose *Export*.

#### 2a- Select target : BELA.

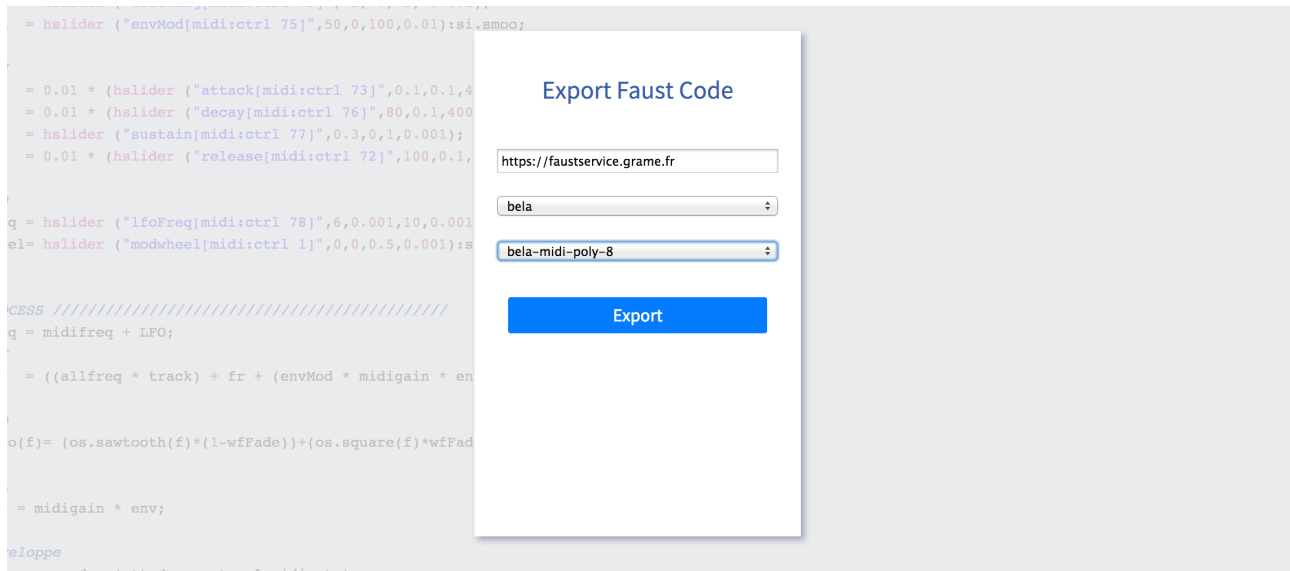
#### 2b- Choose Compilation option, according to the need of FAUST program:

- *BELA*  
For a project that doesn't need MIDI.
- *BELA + MIDI*  
For a project that needs MIDI control, but doesn't need the keyboard (it's not a synthesizer)
- *BELA + number of voices*, from 1 to 8. For a synthesizer, with the choice of the number of polyphonic voices.

The analog inputs of the BELA are always available. Similarly, all parameters are controllable by OSC without additional operations.



## 2c- Click on **EXPORT**:



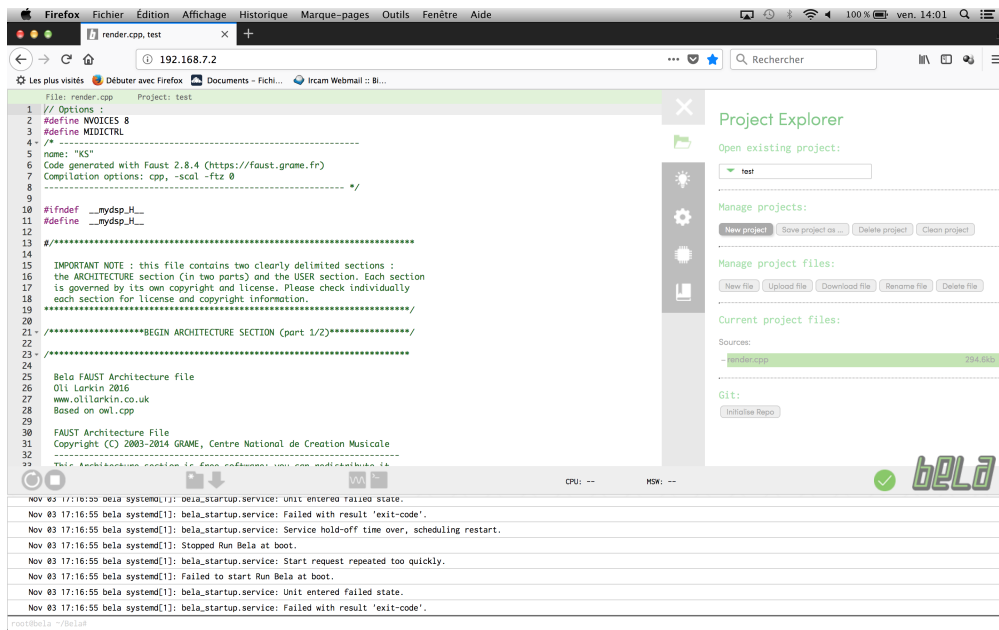
## 3- Download the .ZIP file.

Once unzipped, we have a render.cpp file and possibly an effect.cpp file.

## 4- Copy file(s) into the BELA :

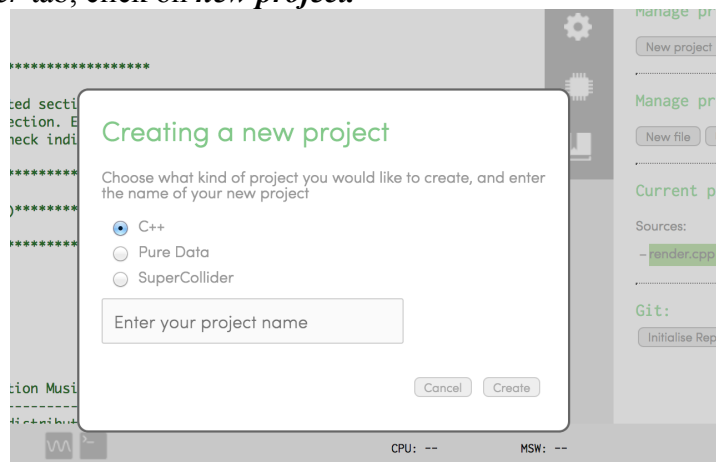
Open a WEB browser, preferably Firefox or Chrome, to the address **<http://192.168.7.2>**

This is the server address contained in the BeagleBone system, it gives access to the programming environment.



#### 4a- Create a new project.

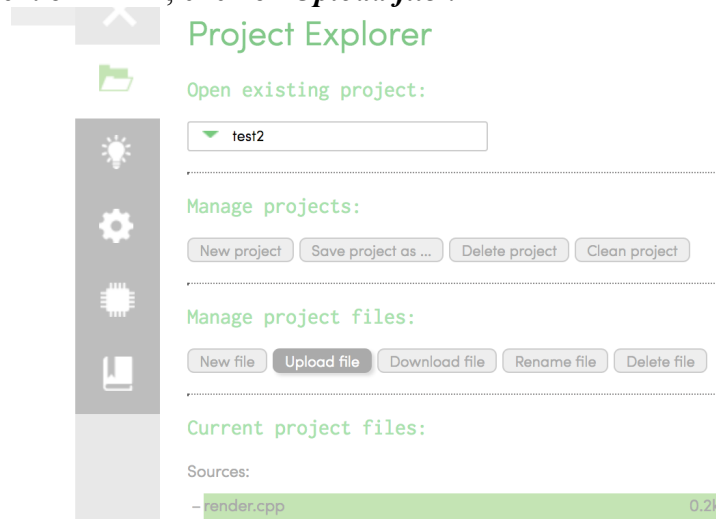
In the *Project Explorer* tab, click on **new project**.



In the window that appears, choose C ++, and enter a name for the project  
Click on **Create**.

#### 4b- Upload the C++ file(s) into the BELA.

Back in the environment of BELA, click on **Upload file** :

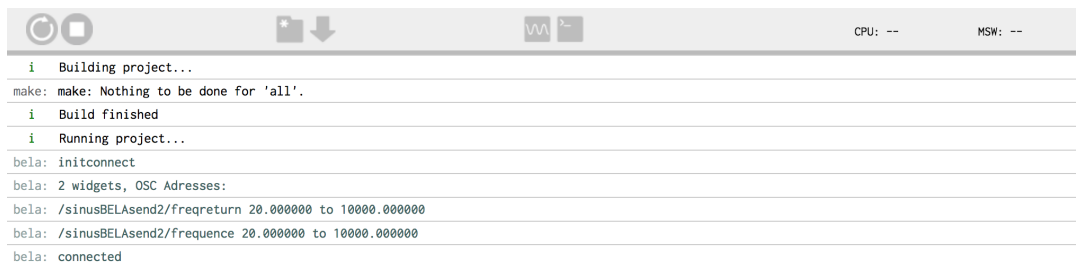


Select the render.cpp (and any additional files like effect.cpp) created previously from the FAUST program. (*overwrite* render.cpp : choose *yes*)

#### 4c- RUN

Click on ***RUN*** (The circular arrow icon on the bottom left).

The compilation may take several minutes. When finished, the last lines of the console summarize the available OSC addresses. Each address also informs about the operating range: the minimum and maximum values of the parameter.



```
i Building project...
make: make: Nothing to be done for 'all'.
i Build finished
i Running project...
bela: initconnect
bela: 2 widgets, OSC Adresses:
bela: /sinusBELAsend2/freqreturn 20.000000 to 10000.000000
bela: /sinusBELAsend2/frequence 20.000000 to 10000.000000
bela: connected
```

## ***Faust2bela script.***

If the FAUST distribution is installed on the computer, one can generate the C ++ code, without going through the online editor, using the **faust2bela** script. The script can also send code files into the BELA, compile and run it.

### **Options :**

-tobela            Send files into the BELA, compile and run it.  
(without this option, files are generated, the user have to add them into BELA manually)  
-osc              OSC activation.  
-midi             MIDI activation.  
-nvoices n       define a synthesizer of 'n' polyphony voices. Involves activating MIDI too.

-effect           Extension for synthesizers: adds an effect after the synthesizer part

There are 2 solutions :

-effect file.dsp       The effect is defined in the file  
-effect auto           The effect is defined in the file that contains the synthesizer.

As a reminder, in this case, in the FAUST code, the effect part is defined by:

effect = FAUST code of the effect chain;

'effect' replace 'process'.

Thus, in the .dsp file, there is a 'process' which defines the synthesizer, AND an 'effect' which defines the effect.

The use of the BELA analog inputs is done in the FAUST code.

Example:

```
frequence = hslider("frequence[BELA: ANALOG_0]", 440, 20, 10000, 0.1);
```

### **Some examples :**

An 8-voice synthesizer contained in the *Synth-file.dsp* file, which is followed by an effect contained in the *FX-file.dsp* file.

```
faust2bela -midi -nvoices 8 -effect FX-file.dsp Synth-file.dsp
```

A 8 voices synthesizer. The *Synth-file-withFX.dsp* file contains the synth and effect code.

```
faust2bela -midi -nvoices 8 -effect auto Synth-file-withFX.dsp
```

A simple 8 voices synthesizer without effects contained in the *Synth-file.dsp* file.

```
faust2bela -midi -nvoices 8 Synth-file.dsp
```

MIDI controllable effect.

```
faust2bela -midi FX-withMIDI-CC.dsp
```

Effect that doesn't need MIDI :

```
faust2bela FX.dsp
```

OSC activation.

```
faust2bela -osc SomethingWithOSC.dsp
```

The script generates a folder *faust.qqc* in which there is another folder with the name of the .dsp used. Inside is the file (s) to copy in the BELA.

## ***Installing BELA***

### **The Driver on the host computer :**

<https://github.com/BelaPlatform/Bela/wiki/Getting-started-with-Bela#macos-users>

### **BELA – SD card install :**

<https://github.com/BelaPlatform/Bela/wiki/Manage-your-sd-card#flashing-the-bela-image>

It needs a SD card of a minimum 4GB, the operation is done on a computer, and not on the beagleBone.

1- Find the last version :

<https://github.com/BelaPlatform/bela-image-builder/releases/>

2- Uncompress to have a .img (it needs KEKA)

<https://www.keka.io/en/>

3- Open a Terminal : `diskutil list`

Find the name of the SD (`/dev/disk1` for example)

4- Unmount the card : `sudo diskutil unmountDisk /dev/disk1`

You need to be Administrator of the computer.

The card disappears.

5- Write the .img into the SD card :

`sudo dd if=/path/to/inputFile.img of=/dev/disk1 bs=1024k`

`/path/to/inputFile.img` = path to the .img unzipped above :

`sudo dd if=/path/to/img/bela_image_v0.3.2.img of=/dev/disk1 bs=1024k`

`/path/to/img` = path to the disk image.

(Note: it's a bit long and we do not see anything... We have to go to the activity monitor, in Disc, we see a line 'dd')

At the end, the card reappears on the desktop, and is called BELABOOT.

6- It's done !

Insert the SD card into the Beaglebone.



# MEASURES

## *Latency measures*

It may be interesting to measure the performance of BELA.

It's possible to define the size of the audio buffer in number of frames (or samples). From 2 minimum to 128 maximum, with a default value of 16.

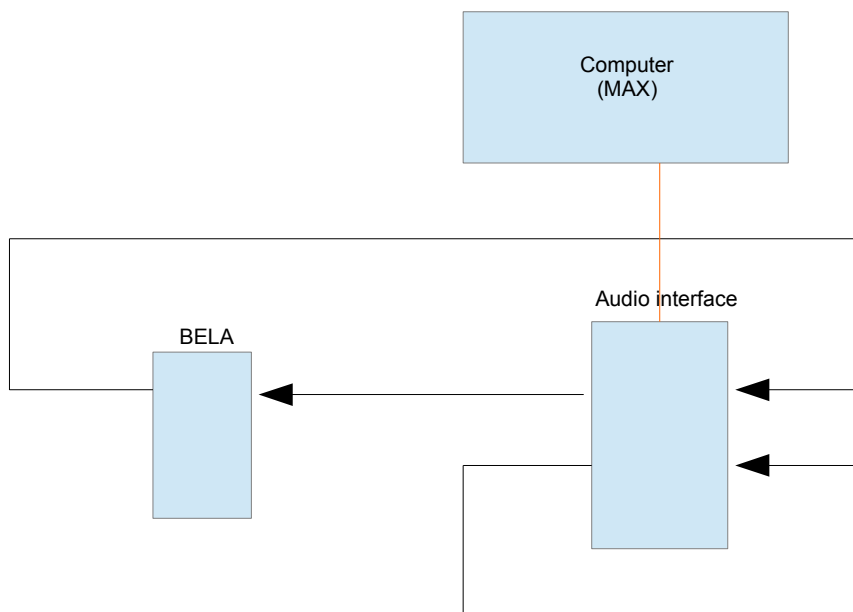
On the other hand, the sampling frequency and the number of bits are fixed at 44.1KHz 16bits.

The full latency of BELA comes down to:

DAC → Buffer audio IN → Buffer audio OUT → ADC

The size of the buffers is known, but it may be interesting to check that this size is well respected.

He remains the unknown of the 2 conversions



To measure this latency, a very simple program that crosses the audio signal from the inputs to the outputs is compiled in the BELA. A 'scope' (signal display option in the BELA, like an oscilloscope) has been added to check the signal in the BELA. An audio signal is sent into one of the inputs of the BELA, and outputted by an audio interface.

The C++ code used into the BELA.

```
#include <Bela.h>
#include <Scope.h>

Scope scope;

bool setup(BelaContext *context, void *userData)
{
    scope.setup(2, context->audioSampleRate);
    // For this example we need the same amount of audio and analog input and output channels
    if (context->audioInChannels != context->audioOutChannels ||
        context->analogInChannels != context->analogOutChannels) {
        printf("Error: for this project, you need the same number of input and output channels.\n");
        return false;
    }
    return true;
}
```

```

void render(BelaContext *context, void *userData)
{
    // Simplest possible case: pass inputs through to outputs
    for (unsigned int n = 0; n < context->audioFrames; n++) {
        float out = audioRead(context, n, 0);
        float out2 = audioRead(context, n, 1);

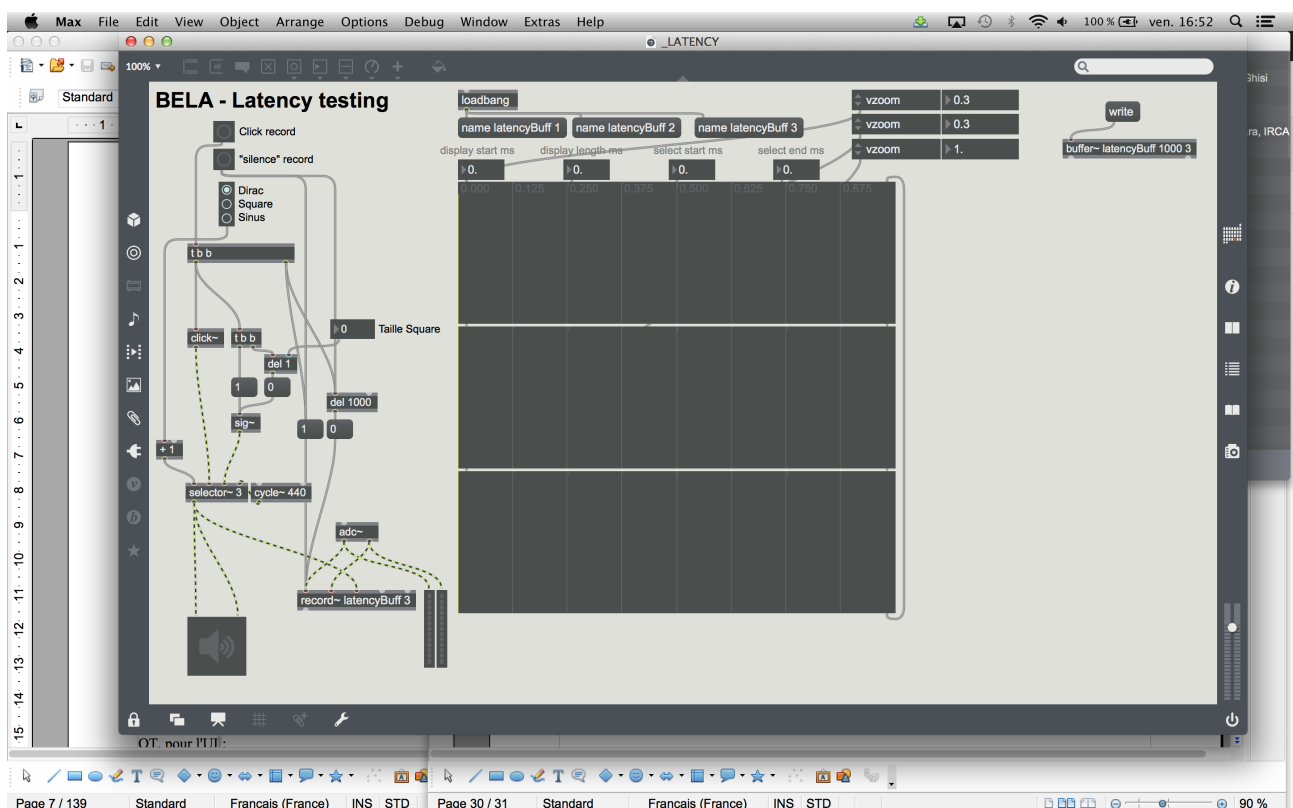
        scope.log(out, out2);

        audioWrite(context, n, 0, out);
        audioWrite(context, n, 1, out2);
    }
}

void cleanup(BelaContext *context, void *userData)
{
}

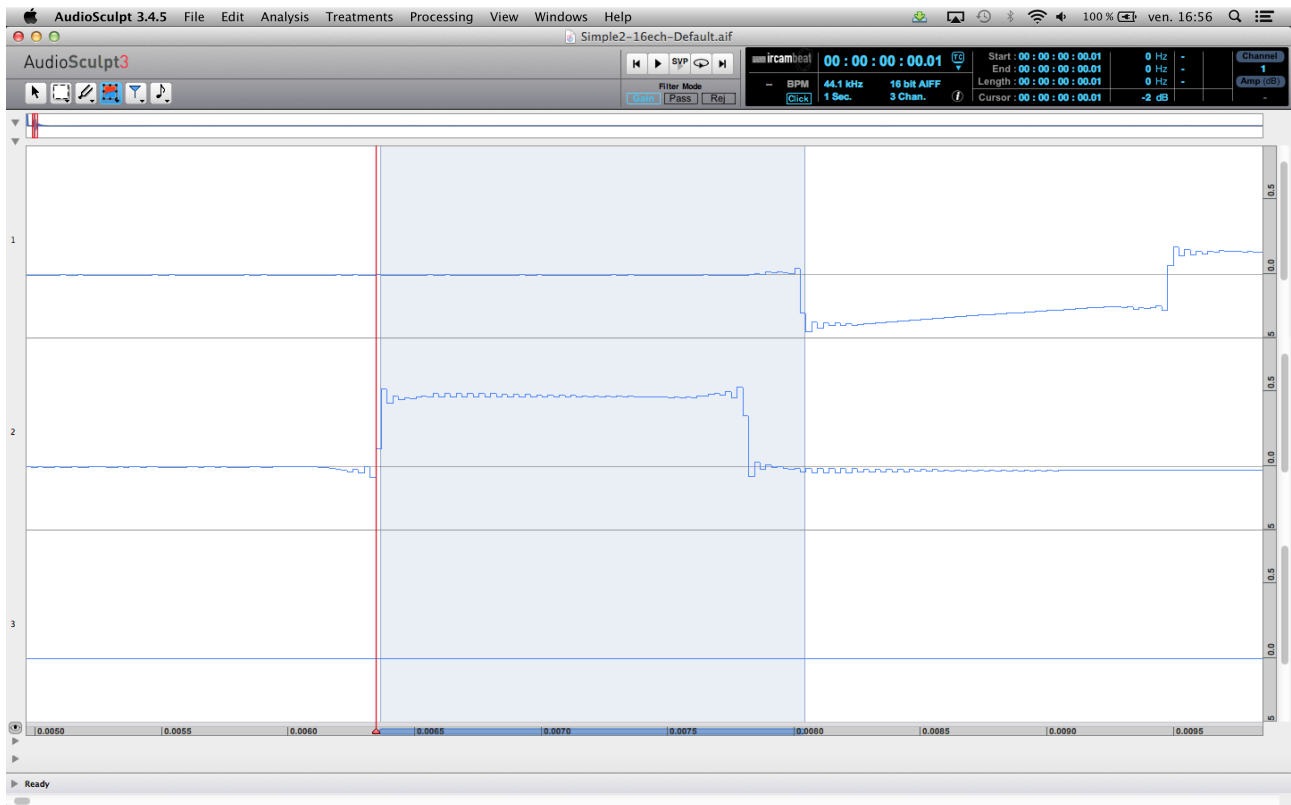
```

It's a MAX patch that takes care of generating the signal, sending it and recovering it. But the computer itself (+ audio interface) generates latency. It must be known when exactly the signal enters the BELA and comes out. For this, the same signal is sent to another output of the sound card, and is directly returned to an input. Thus, by comparing with the signal received by the BELA it is possible to find only the delay induced by the BELA. MAX records the 2 signals on the same file, with the pure signal as 3rd reference, as a reference.



Of course, you must first check that the signal is sent and received perfectly at the same time in the 2 outputs. Just use the same system without the BELA in the loop. Audio interfaces do not normally have this problem of different latency between their inputs and outputs. In the same way, nothing guarantees a priori that MAX correctly records 2 different signals

while respecting their arrival. This is the case here, where the 2 signals are perfectly identical sample to sample. This is after all logical, a shift, even a single sample, would have adverse consequences on the quality of the audio!



The signal sent is a square pulse. The measurement will be on the rising edge of the signal. Of course, the perfectly straight front will be deformed by the analog stages. Effect multiplied on the BELA because of the 2 additional conversions. This induces that the square signal has its rising and falling edges less steep, and that a small oscillation can appear, all hindering the estimation of the passage of the rising edge.

The choice will be made on the highest sample, in the end, the error induced here is negligible, and as we will see, can explain the difference in measurement found with the values given by BELA.

3 measurements are made, with 3 different buffer sizes : 2 (min), 16 (default), 128 (max)

The audio file is compared in an editing software. I used Audiosculpt, because it is present on my machine, and allows to zoom to the sample while having a displayed scale smaller than the millisecond of millisecond.

The latency is therefore the measure of the arrival of the rising edge in the interface less this one coming out of the BELA (by the 2nd input of the interface).

**BELA(ms)** indicates when the signal arrived in the BELA, compared to the beginning of the audio file.

**Interface (ms)** indicates the same value, but concerning the signal passing directly in the interface.

**Latency (ms)** is the difference of the 2 previous signals, and gives the total latency of BELA.

**Latency** gives the same value expressed as the number of samples.

Knowing the size of the buffers, it is possible to find the latency induced by the 2 conversions.

	BELA (ms)	Interface (ms)	Latency (ms)	Latency	Latency with buffers	Lat. without buffers
<b>2 samp.</b>	7.39229	6.34922	1.04307	46 ech.	42 ech.	0.952381ms
<b>16 samp.</b>	8.04985	6.37190	1.67801	74 ech.	42 ech.	0.952381ms
<b>128 samp.</b>	13.12925	6.37188	6.75737	298 ech.	42 ech.	0.952381ms

The values are consistent, regardless of the size of the buffers, the latency of the conversions are identical.

For its part, BELA gives these measures :

<https://blog.bela.io/2016/11/07/bela-analogue-vs-audio/>

	Bits	Sample Rate (kHz)	Level	Coupling	Conversion latency	Internal aliasing filter?
Audio In	16	44.1	+/- 1V (gain available)	AC	<b>0.39ms</b>	yes
Audio Out	16	44.1	+/- 1V (gain available)	AC	<b>0.47ms</b>	yes
Analog In	16	22.05 x 8 44.1 x 4 88.2 x 2	0-4.096V	DC	< 10us	no
Analog Out	16	22.05 x 8 44.1 x 4 88.2 x 2	0-5V	DC	< 10us	no

Therefore, give  $0.39 + 0.47$  or  $0.86$  ms AD + DA conversion. Compared to  $0.95$  measured. There is a difference of almost  $0.1$  ms. This difference may be due to the choice of the measuring point of the rising edge, chosen a little too late (?). This is an error of 4 samples, which seems consistent when looking at the waveform. And the choice certainly wrong at first to choose the highest point.

$0.86$  ms, this is equivalent to 38 samples.

Note that the signal is reversed at the exit of BELA:

By checking the oscilloscope function of the BELA interface, which gives the shape of the signal before its output, it is the analog output stage which inverts the signal.

Would there be an error in the setup used, or is the signal actually reversed?

The study of the electronic schema does not seem to allow such an inversion, except to imagine that the fault is the DAC ?

## FAUST performance on BELA

It's pertinent to ask the question of the interest of using FAUST to program the BELA. Certainly, the first interest is that by using FAUST, it is no longer necessary to dive into the C++ code. It is possible to do the same with PureData: create its PD patch, and deposit it in the BELA which is responsible for compiling it. But this solution is deemed not to use optimally the characteristics of BELA. What about FAUST?

As an approach to the possibilities of BELA, one can look for how many oscillators it is possible to generate before the processor generates an error.

The tests were done with default buffers of only 16 samples.

BELA already has some examples.

The most impressive is a code based on a library written in assembly, which can generate up to 500 oscillators! This number is not attainable other than by this special library.

A first test series was done with FAUST a non optimized oscillator:

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

phasor(f) = f/ma.SR : (+,1.0:fmod) ~ _ ;
osc(f, vol) = (phasor(f) * 6.28318530718 : sin)*vol;
freq = hslider("frequence[BELA: ANALOG_0]", 440, 20, 10000, 0.1);

process = par(i, 40, osc(freq*(i+1), 0.8/i)) :> _ ;
```

And 2 examples of BELA, with the classical math library (cmath) and with the optimized library (neon).

Oscillators	BELA(neon)	BELA(cmath)	FAUST
20	26%	38%	55%
30	33.2%	51.5%	73.5%
32	34.6% (59% à 2 ech.)	55.5%	87%
40	41%	65.5%	-
50	49%	81%	-
80	72%	-	-
100	87%	-	-

It's possible to generate 32 oscillators with this version of FAUST, and 100 with the BELA and its optimized library. One of the reasons for this low score is certainly due to the difference in algorithm chosen between the oscillator code of BELA and that of FAUST.

In particular, the FAUST oscillator use a Modulo :

```
phasor(f) = f/ma.SR : (+,1.0:fmod) ~ _ ;
```

Whereas the BELA oscillator instead uses a simple value comparison with an 'if'.

The modulo function use a lot of CPU.

But FAUST offers several types of oscillators. Here are tested only the sine oscillators. The number is set at **20** to allow comparison. For Promising Versions, the maximum number was searched.

The FAUST program:

```
import("stdfaust.lib");
freq = hslider("frequence[BELA: ANALOG_0]", 440, 20, 10000, 0.1);
process = par(i, 20, (os.oscrs(freq*(i+1))*(0.8/(i+1)))) :> _;
```

**oscrs** is replaced by the name of the oscillator being tested.

The sum of the sines is supposed to reproduce a sawtooth signal (SAW).

#### WaveTable bases osc :

(os.)oscsin 27%

(os.)osci 37%

#### Filter-BasedOscillators

(os.)oscb 16%

The fastest but does not give a SAW (phases?). And not constant amplitude.

(os.)oscrs 17.5% Looks stable.

50 osc : 24.5%

120 osc : 44.5%

**200 osc : 69% (with some Underrun...)**

(os.)oscr 17.5% Don't looks like a saw (phases?)

(os.)osc 16% Don't looks like a saw (phases?).

And have some error when frequency is too high.

(os.)oscs 16% Idem osc.

#### Waveguide-Resonator-BasedOscillators

(os.)oscw 20% Don't looks like a saw (phases?).

And have some error when frequency is too high.

(os.)oscws 20% Have some error when frequency is too high.

**With os.oscrs, it's possible to have more than 200 oscillators. That is more than double what the BELA can do at most (using NEON, but without the special class made in assembler).**

#### Notes :

These measurements were made with the FAUST architecture file for BELA (bela.cpp) not optimized. The improvement of code brings only a small increase of the available CPU resources (a few %).

For the chosen oscillator, oscrs, the CPU load increases with the frequency:

At low frequency, it is possible to have well over 200 oscillators, and much less than 200 at high frequency. It seems that this type of oscillator works poorly at frequencies beyond the Nyquist frequency.

A question may arise: is the stability of the oscillators identical?

Indeed, the way of generating a sine is very different between the reading of a table and an oscillator resulting from a filter.

The most efficient oscillator comes from a filter. Is it as stable as a conventional oscillator for reading a wave table?

The comparison between **oscrs** and **oscsin** does not make it possible to distinguish the 2 signals.  
Setup :

In FAUST, one oscillator on the right channel, the other on the left channel.  
Observation by ear, and in MAX with yin ~ for the detection of the pitch, and an oscilloscope.  
The 2 signals are also summed to highlight small variations and coherence of the phases.  
The phase is identical on both signals, and there is no oscillation.

## Compilation options

The Beaglebone has 2 compilers: clang 3.8 and gcc / g++ 4.9

It's **clang** which is used by default. The command line for compiling C++ projects from BELA is:

**-O3 -march=armv7-a -mtune=cortex-a8 -mfloat-abi=hard -mfpu=neon -ftree-vectorize -ffast-math**

It is possible to change the compilation options. Some tests have been done, including changing the optimization option -O.

The difference in performance is very low: + 1% for -O2 and +2 or 3% for -O1 compared to -O3.

More complete tests have already been done by the BELA team, the results are on this page:

<https://forum.bela.io/d/60-c-11/9>

```
gcc full-opt: -O3 -march=armv7-a -mtune=cortex-a8 -mfloat-abi=hard -mfpu=neon --fast-math -ftree-  
vectorize -ftree-vectorizer-verbose=1
```

```
clang full-opt: -O3 -march=armv7-a -mtune=cortex-a8 -mfloat-abi=hard -mfpu=neon -ftree-vectorize
```

```
_main.pd(techno world) heavy:
```

```
gcc-4.6 -O2 49%
```

```
gcc-4.6 -O3 49%
```

```
gcc-4.6 full-opt 48%
```

```
gcc-4.9 -O2 48.5%
```

```
gcc-4.9 -O3 48.5%
```

```
gcc-4.9 full-opt 43.8%
```

```
gcc-5.0 full-opt 44%
```

```
clang -O2 53%
```

```
clang -O3 53%
```

```
clang full-opt 26%
```

```
_main.pd(techno world) libpd:
```

```
gcc-4.9 fullopt 53.2%
```

```
clang full-opt 52.8%
```

```
airharp (C++ project)
```

```
clang -O3 69%
```

```
clang full-opt 63.5%
```

```
gcc-4.9 -O3 68.7%
```

```
gcc-4.9 fullopt 63%
```

```
Supercollider
```

```
buildnativeclangdefault
```

```
idle 16%
```

```
120sinewaves 49.3%
```

```
buildnativeclangvectorize
```

```
idle 15.8%
```

```
120sinewaves 50%
```

```
_____buildnativegcc49default_____
```

```
idle 21%
```

```
120sinewaves 51%
```

```
_____buildnativegcc49vectorizeffastmath_____
```

```
idle 16%
```

```
120sinewaves 49.5%
```

```
_____buildcrossgccdefault_____
```

```
idle 17.3%
```

```
120sinewaves 48.3%
```

```
_____buildcrossgccvectorizefastmath_____
```

```
idle 17.7%
```

```
120sinewaves 47.5%
```

## DEMO FILES :

Some FAUST examples built and tested for BELA.

### Synthesizers :

They are made to be controllable in MIDI.

The additive synthesizer, illustrates the use of OSC messages.

They are made in 2 versions:

- Purely MIDI
- Using Analog inputs. 4 parameters are devolved to the synth, and 4 others for the effects. (Except the AdditiveSynth which is not powerful enough to have effects).

The synthesizer `synthName+FX<_Analog>.dsp` included in the same file synthesizer and effects, for the online editor.

### AdditiveSynth.dsp    78% CPU (6 voices, without FX)

Additive synthesizer with 8 harmonics, a volume envelope by harmonic.

The polyphony is reduced to 4 voices, and there is not enough CPU to add FX.

(It is especially the envelopes that make performance worse).

The setting of the parameters is done by OSC.

The Analog version allows the control of the volume of the 8 harmonics by the Analog inputs of BELA.

*OSC messages (see BELA console for precise adress)*

*For each harmonics ( %rang indicate harmonic number, starting at 0 ) :*

*vol%rang                    : General Volume                    (vol0 control the volum of the fundamental)*

*(replaced by Analog\_0 to Analog\_7 for \_analog version of this program)*

*A%rang                    : Attack*

*D%rang                    : Decay*

*S%rang                    : Sustain*

*R%rang                    : Release*

### FMSynth2.dsp            60% CPU (8 voices, with FX)

Simple FM with 2 operators. The modulator can also be modulated by itself (feedback).

Analogue version for the control of the parameters by the Analog inputs of BELA

*MIDI IMPLEMENTATION:*

*CC 1    : FM feedback on modulant oscillator.*

*CC 14 : Modulator frequency ratio.*

*CC 73 : Attack*

*CC 76 : Decay*

*CC 77 : Sustain*

*CC 72 : Release*

*For Analog version :*



#### *ANALOG IMPLEMENTATION:*

*ANALOG\_0 : Modulator frequency ratio*

*ANALOG\_1 : Attack*

*ANALOG\_2 : Decay/Release*

*ANALOG\_3 : Sustain*

### **simpleSynth.dsp      59% CPU (8 voices, with FX)**

Simple classic subtractive synthesis with an oscillator.

The waveform of the oscillator can vary from sawtooth to square.

The maximum of parameters are controllable by MIDI, respecting as much as possible the recommendations of the standard.

Analogue version for the control of the parameters by the Analog inputs of BELA

#### *MIDI IMPLEMENTATION:*

*CC 70 : waveform (Saw to square)*

*CC 71 : Filter resonance (Q)*

*CC 74 : Filter Cutoff frequency*

*CC 79 : Filter keyboard tracking (0 to X2, default 1)*

*CC 75 : Filter Envelope Modulation*

*Envelope :*

*CC 73 : Attack*

*CC 76 : Decay*

*CC 77 : Sustain*

*CC 72 : Release*

*CC 78 : LFO frequency (0.001Hz to 10Hz)*

*CC 1 : LFO Amplitude (Modulation)*

*for Analog version :*

#### *ANALOG IMPLEMENTATION:*

*ANALOG\_0 : waveform (Saw to square)*

*ANALOG\_1 : Filter Cutoff frequency*

*ANALOG\_2 : Filter resonance (Q)*

*ANALOG\_3 : Filter Envelope Modulation*

### **WaveSynth.dsp      70% CPU (8 voices, with FX)**

Wavetable synthesizer test: using a MIDI controller, it is possible to travel between 4 waveforms (see PPG / Waldorf Wave, Seq-Circuit Prophet VS ...)

The 4 waveforms are calculated by a small special code described below.

Analogue version for the control of the parameters by the Analog inputs of BELA

#### *MIDI IMPLEMENTATION:*

*CC 1 : LFO Depth (wave travel modulation)*

*CC 14 : LFO Frequency*

*CC 70 : Wave travelling*

*CC 73 : Attack*

*CC 76 : Decay*  
*CC 77 : Sustain*  
*CC 72 : Release*

*ANALOG IMPLEMENTATION:*

*ANALOG\_0 : Wave travelling*  
*ANALOG\_1 : LFO Frequency*  
*ANALOG\_2 : LFO Depth (wave travel modulation)*  
*ANALOG\_3 : Release*

***Effect for the synthesizers :***

**simpleFX.dsp**

Designed to work with synthesizers, it also uses MIDI control.  
The chain of effect consists of:  
Distorsion – Flanger – Reverberation.  
In addition, the general setting of volume and panning.

*MIDI IMPLEMENTATION:*

*CC 7 : Volume*  
*CC 10 : Pan*

*CC 92 : Distortion Drive*

*CC 13 : Flanger Delay*  
*CC 93 : Flanger Dry/Wet*  
*CC 94 : Flanger Feedback*

*CC 12 : Reverberation Room size*  
*CC 91 : Reverberation Dry/Wet*  
*CC 95 : Reverberation Damp*  
*CC 90 : Reverberation Stereo Width*

*ANALOG IMPLEMENTATION:*

*ANALOG\_4 : Distortion Drive*  
*ANALOG\_5 : Flanger Dry/Wet*  
*ANALOG\_6 : Reverberation Dry/Wet*  
*ANALOG\_7 : Reverberation Room size*

***Effects :***

These 2 examples are intended to work with the 8 analog control inputs.  
Input and output signals are stereo.

**crossDelay2.dsp      28% CPU**

A stereo delay with 2 feedback: a classic, a crosstalk: the signal coming out of the left channel is injected into the right delay, and vice versa. To improve the effect, a delay without feedback is provided as input.

Finally, a pitch shifter is inserted into the feedback loop.

*ANALOG IN:*

*ANALOG 0 : Pre-Delay L*  
*ANALOG 1 : Pre-Delay R*  
*ANALOG 2 : Delay L*  
*ANALOG 3 : Delay R*  
*ANALOG 4 : Cross feedback*  
*ANALOG 5 : Feedback*  
*ANALOG 6 : Pitchshifter L*  
*ANALOG 7 : Pitchshifter R*

*Available by OSC : (see BELA console for precise adress)*

*Feedback filter:*

*crossLF : Crossfeedback Lowpass*  
*crossHF : Crossfeedback Highpass*  
*feedbLF : Feedback Lowpass*  
*feedbHF : Feedback Highpass*

## **FXChaine2.dsp      64% CPU**

More complete effect chain than the simpleFX

Chorus – Phaser – Delay – Reverberation

With only 8 parameters, the settings are simplified to a Dry / Wet and a main parameter by effect of the chain.

*ANALOG IN:*

*ANALOG 0 : Chorus Depth*  
*ANALOG 1 : Chorus Delay*  
*ANALOG 2 : Phaser Dry/Wet*  
*ANALOG 3 : Phaser Frequency ratio*  
*ANALOG 4 : Delay Dry/Wet*  
*ANALOG 5 : Delay Time*  
*ANALOG 6 : Reverberation Dry/Wet*  
*ANALOG 7 : Reverberation Room size*

*Available by OSC : (see BELA console for precise adress)*

*Rate : Chorus LFO modulation rate (Hz)*  
*Deviation : Chorus delay time deviation.*

*InvertSum : Phaser inversion of phaser in sum. (On/Off)*  
*VibratoMode : Phaser vibrato Mode. (On/Off)*  
*Speed : Phaser LFO frequency*  
*NotchDepth : Phaser LFO depth*  
*Feedback : Phaser Feedback*  
*NotchWidth : Phaser Notch Width*  
*MinNotchI : Phaser Minimal frequency*  
*MaxNotchI : Phaser Maximal Frequency*

*Damp : Reverberation Damp*  
*Stereo : Reverberation Stereo Width*

## **granulator.dsp      23% CPU**

From the Faust Audio Playground example.

*ANALOG IN:*

*ANALOG 0 : Grain Size*

*ANALOG 1 : Speed*

*ANALOG 2 : Probability*

*(others analog inputs are not used)*

## **GrainGenerator.dsp 33% CPU**

"In progress" version of granular synthesis, ready for future evolutions (more grains, random ...)

*ANALOG IN:*

*ANALOG 0 : Population: 0=almost nothing. 1=Full grain*

*ANALOG 1 : Depth of each grain, in ms.*

*ANALOG 2 : Position in the table = delay*

*ANALOG 3 : Speed = pitch change of the grains*

*ANALOG 4 : Feedback*

*(others analog inputs are not used)*

## **repeater.dsp 13% CPU**

Example of a somewhat complex code. Repeat a fragment of the incoming sound every 'n' ms.

*ANALOG IN:*

*ANALOG 0 : Duration (ms) between 2 repeat series (500 to 2000 ms)*

*ANALOG 1 : Duration of one repeat (2 to 200 ms)*

*ANALOG 2 : Number of repeat*

*(others analog inputs are not used)*

## **Some small FAUST functions :**

### **FAUST-complement.dsp**

To build the wavetable synth, 2 functions have been created :

### **Scanner :**

It allows you to travel between an arbitrary number of entries.

Like a selector, but with an interpolation between the inputs.

2 versions are available :

scanner4(nb, position)      Simplified that works from 2 to 4 entries

scanner(nb, position)      Complete for 2 to more than 4 entries.

### **WaveTable generator :**

As sinwaveform or coswaveform.

For the wavetable synthesizer or waveshapping.

**RS Latch :**

RS latch : an input to set the output to 0, an input to set the output to 1.

**counterUpReset :**

A counter that counts the input pulses up to a certain number, and stops.

A Reset input to reset the counter and allow it to re-count.