faust2sam
Faust Integration with the Analog Devices SHARC Audio Module Platform

Gregory Pat Scandalis
support@moforte.com
Document Revision R1.4

Document History:

GPS	9/14/2017	Document Started
GPS	12/09/2017	First revision (R1.1) completed
GPS	01/02/2018	Revisions
GPS	04/28/2018	Final revisions
GPS	06/02/2018	Redid MIDI assignment tables.

1.0 faust2sam Overview

1.1 Faust

Faust (Functional Audio Stream) is an open-source, functional programming language, specifically designed for real-time audio signal processing and synthesis. Faust generates C++, as well as other target languages, for signal processing applications.

In addition to generating efficient inner loops in C++, Faust also:

- Can generate test GUIs for prototyping algorithms
- Can generate easy-to-read hierarchical block diagrams directly from the Faust source, which graphically illustrate signal flow and processing.
- Provide a runtime system that supports MIDI for both voice allocation and parameter control.

Below is an example of the complete Faust code for a simple MIDI controlled sawtooth synthesizer:

```
File Edit Options Buffers Tools Help
 1
 2  import("stdfaust.lib");
 3
 4  // Simple MIDI controlled sawtooth synth
 5
 6  normMIDI(mv)   = mv/127.0;
 7  vol  = normMIDI(hslider("Ctrl Value IN (Ctrl 1) [midi:ctrl 1]", 60, 0, 127, 1)) ;
 8
 9
10  f = nentry("freq",200,40,2000,0.01);
11  bend = nentry("bend",1,0,10,0.01) : si.polySmooth(t,0.999,1);
12  g = nentry("gain",1,0,1,0.01);
13  t = button("gate");
14  freq = f*bend;
15  envelope = t*g*vol : si.smoo;
16
17  process = os.sawtooth(freq)*envelope <: _,_;


-UU-:**--F1  sawtooth_synth.dsp   All L5    Git:master  (FAUST mode) ----------------------
```

Chapter 10 will go into more detail about this example, its MIDI control and its block diagram.

1.2 faust2sam

Faust programs may be targeted for many different platforms via what is known as "an architecture". The script **`faust2sam`** calls the Faust compiler using an architecture that is specific to the SHARC Audio Module platform. When an algorithm is compiled using **`faust2sam`**, three C++ source code files are generated for the SHARC Audio Module platform. These files may then be inserted into a Cross Core Embedded Studio (CCES) workspace that can be compiled into an algorithm that runs on the SHARC Audio Module platform.

1.3 Useful Faust References

This document will **not** go into detail about the Faust language. There are a number of good references and tutorials for Faust. Here are some useful references for learning more about Faust.

- The main Faust website: http://faust.grame.fr
- Faust Documentation: http://faust.grame.fr/Documentation/
- Dr. Julius O. Smith III's site about Faust: https://ccrma.stanford.edu/~jos/spf/
- Romain Michon's Faust tutorials: https://ccrma.stanford.edu/~rmichon/faustTutorials/
- Romain's online Faust course: https://ccrma.stanford.edu/~rmichon/faustWorkshops/course2015/
- The Wikipedia entry: https://en.wikipedia.org/wiki/FAUST_(programming_language)

1.4 The Faust Compiler

At the present time, Faust must be downloaded to a host computer, compiled and installed. All prototyping is done on the host computer. Currently, Faust is easily compiled and run on Mac/OS X machines and Linux machines. This document describes how to download, compile and install Faust on a Mac/OS X machine.

In the very near future the Faust compiler will be able to run on a remote server. All prototyping will be done in a web browser and the browser will be able to export C++ targeted for the SHARC Audio Module platform. Currently the Faust team is testing this new technology.

1.5 Prerequisites (Mac/OS X)

A number of prerequisites are needed in order to compile Faust and develop with Faust.

- xcode needs to be installed and the command line tools need to be installed so that Faust can be compiled.
- Before Faust is downloaded macports should be installed along with a number of useful packages. macports is a package manager for ports of Linux tools for the Mac. The most important package to install from macports is the QT5 package, which provides libraries for Faust to build GUIs used for prototyping algorithms.
- Faust will need to be downloaded as well. Faust will be compiled and installed on the system.
- A set of additions to Faust to support **`faust2sam`** will need to be installed. These are installed with the **`add_faust_additions.tcsh`** script.
- Optionally a **`faust-mode`** elisp file can be installed for emacs to support visual editing of Faust code using emacs.

1.6 Workflow

The typical workflow using Faust would be:

- Create algorithm in Faust. MIDI control can be attached to the algorithm in the Faust code using the MIDI metadata mechanism (see chapter 7).
- Use **`faust2caqt`** to create a running GUI app on the host computer (Mac/OS X) so that the algorithm can be tested. This GUI app can process audio (on the Mac host) as well as MIDI.
- Optionally the developer can use **`faust2firefox`** to generate a hierarchical block diagram for the algorithm to examine the algorithm's signal flow.
- Once the developer is satisfied with the algorithm **`faust2sam`** can be used to generate a set of C++ files for the algorithm that are intended to be used with a CCES framework.
- These 3 source files can be copied to the Faust directory in the CCES framework. The framework can then be compiled and downloaded to the platform.

2.0 Installing MacPorts

MacPorts is a package manager for ports of Linux tools for the Mac. Before Faust is downloaded MacPorts should be installed along with a number of useful packages. The most important package to install from macports is the QT5 package, which provides libraries for Faust to build GUIs used for prototyping algorithms.

The main site for macports is: https://www.macports.org

The installation page is: https://www.macports.org/install.php

Once macports is installed QT5 should be installed with the command

```
% sudo port install qt5
```

This command will install/compile QT 5 along with its dependencies. This install can take some time, as much as an hour or more.

3.0 Downloading, Compiling and Installing Faust

3.1 Downloading Faust

The latest Faust can be cloned from github. Typically this would be cloned into ~/Developer/faust:

```
% cd ~/Developer
% git clone https://github.com/grame-cncm/faust.git
```

3.2 Compiling Faust

Faust can be compiled like this:

```
% cd ~/Developer/faust
% make
```

The compile will run for about 5 minutes.

3.3 Installing Faust

Faust can be installed on your system with this:

```
% cd ~/Developer/faust
% sudo make install
```

You can test that Faust is installed by invoking the **`faust –version`** command.

```
% faust --version
FAUST : DSP to C, C++, Rust, LLVM IR, JAVA, JavaScript,
asm.js, WebAssembly (wast/wasm), Interpreter compiler,
Version 2.5.15
Copyright (C) 2002-2018, GRAME - Centre National de
Creation Musicale. All rights reserved.
```

4.0 Installing The faust2sam Additions

4.1 Cloning the SHARC Audio Module Faust Repository

The SHARC Audio Module Faust repo contains additions to be added to an existing Faust tree/installation as well as a number of Faust examples. This repo may be cloned with:

```
% cd ~/Developer
% git clone https://github.com/moforte/sam-faust.git
```

Some important items found in this repo are:

```
buildAllFaustExamples.tcsh - This is a script that will build the
                             SHARC Audio Module examples.
faust-additions            - These are additions to be added to an
                             existing Faust distribution with the
                             install_faust2sam.tcsh script.
faust-examples             - Example faust algorithms (.dsp) files
                             along with pre-compiled into C++ versions
                           of the algorithms.
   16-channel-volume      - A 16 channel MIDI controlled volume example
   chorus                 - A chorus stompbox
   echo                   - An echo stomp box
   effects                - An effects chain of
                            echo : flanger : chorus : reverb
   flanger                - A flanger stompbox
   freeverb               - A reverb
   virtualAnalog          - The virtualAnalog Synth
   sawtooth_synth         - A simple MIDI controlled sawtooth synth
   sine_synth             - A simple MIDI controlled sine synth
   volume                 - A simple MIDI controlled volume
```

4.2 Adding faust2sam Additions to the Faust Build Tree

In order to add the **faust2sam** additions to the Faust build tree, use the **add_faust_additions.tcsh** script.

```
% cd ~/Developer/sam-Faust/faust-additions
% add_faust_additions.tcsh ~/Developer/faust
```

4.3 Installing the faust2sam additions.

Once the additions have been added to the Faust tree you will need to install the additions on your system.

```
% cd ~/Developer/Faust
% sudo make -f Makefile.sam install
% cd ~/Developer/faust/tools/faust2appls
% sudo make -f Makefile.sam install
```

You can test that **faust2sam** is installed by invoking the **faust2sam** command.

```
% faust2sam


faust2sam can be used to generate faust-based dsp objects for the
ADI SHARC Audio Module board


Global options:
   -midi
   -nvoices N : creates a polyphonic object with N voices.
   -effect <effect.dsp>: adds an effect to the polyphonic synth
(this option is ignored if -nvoices is not specified).
```

5.0 The CCES Bare Metal Project

The bare metal framework is a part of the Analog Devices SHARC Audio Module platform installer, which contains a CCES project workspace that can be used to build SHARC Audio Module platform.

Information on how to get the Bare Metal Framework can be found here:

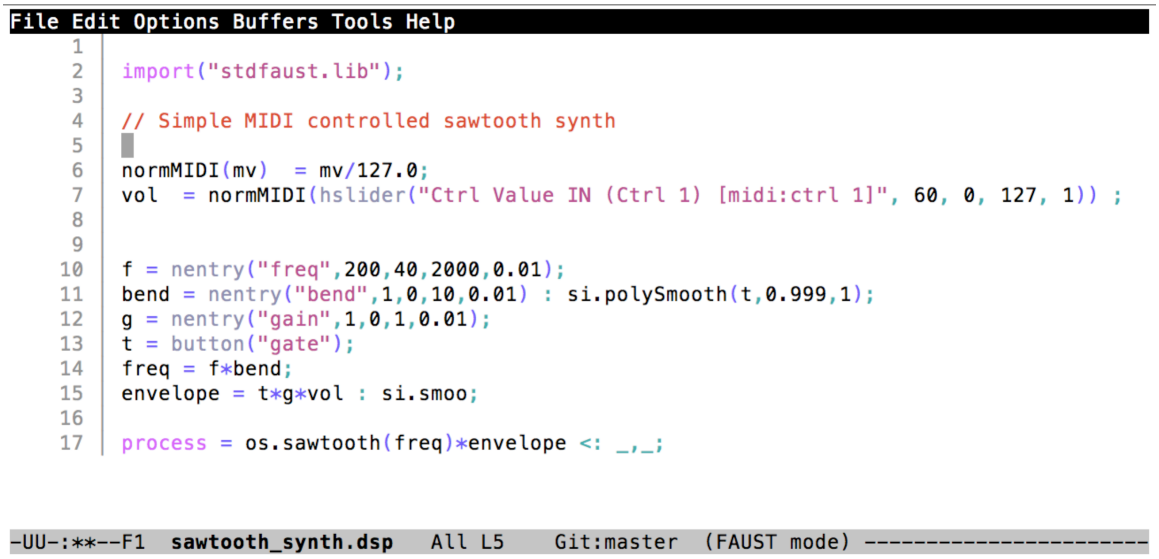
https://wiki.analog.com/resources/tools-software/sharc-audio-module/baremetal

Faust algorithms can be added to this framework in the “faust” directory, which will be explained in section 8.5 (below)

6.0 Setting up Faust Editing Mode in emacs

Optionally for users of emacs, a Faust editing mode can be setup in the. emacs file.

Below is an example Faust program in emacs Faust editing mode.



The elisp file for Faust editing mode and instructions for set it can be found here:

https://github.com/rukano/emacs-faust-mode

7.0 MIDI in Faust

7.1 Mapping MIDI messages to Faust Control

Faust has a number of meta data conventions for mapping MIDI messages into Faust control. Below is the simple MIDI controlled sawtooth synth, which illustrates how this control mapping is down. In the example below nentry() is a numerical entry object that can be mapped to receive specific MIDI values. A number of metadata values are reserved to have specific mapping functions.

- freq – If a MIDI noteOn event is received it's MIDI keyNumber is mapped to a frequency.
- bend – if a MIDI pitchBend message is received it is mapped to a bend value.
- gain – if a MIDI noteOn message is received it's velocity value is mapped to a gain value which ranges from [0 .. 1.0]
- gate – if MIDI noteOn/noteOff messages are received they are mapped to a gate value (0/1)
- [midi:ctrl 1] – A Faust control (slider, etc.) can be mapped to listen to a MIDI continuous controller.

```
File Edit Options Buffers Tools Help
 1
 2  import("stdfaust.lib");
 3
 4  // Simple MIDI controlled sawtooth synth
 5
 6  normMIDI(mv)   = mv/127.0;
 7  vol  = normMIDI(hslider("Ctrl Value IN (Ctrl 1) [midi:ctrl 1]", 60, 0, 127, 1)) ;
 8
 9
10  f = nentry("freq",200,40,2000,0.01);
11  bend = nentry("bend",1,0,10,0.01) : si.polySmooth(t,0.999,1);
12  g = nentry("gain",1,0,1,0.01);
13  t = button("gate");
14  freq = f*bend;
15  envelope = t*g*vol : si.smoo;
16
17  process = os.sawtooth(freq)*envelope <: _,_;


-UU-:**--F1  sawtooth_synth.dsp   All L5    Git:master  (FAUST mode) ----------------------
```

7.2 MIDI Conventions for the Pots and Push Buttons on the DIY Board

The pots and push buttons on the DIY board can be used to control key algorithm parameters. By default the pots are mapped to MIDI CC-2,3,4 and the push button switches are mapped to MIDI CC-102,103,104,105.

The algorithm examples that are provided use these conventions. For example, the effects algorithm is “echo : flange : chorus : reverb”. For this algorithm each of the four push buttons turns on a different effects unit. The first pot is the echo feedback, the second pot is the reverb room size and the third pot is the reverb damping.

8.0 Example Workflow – MIDI Controlled Volume on Core 1

A first example to demonstrate the workflow is a simple MIDI controlled stereo volume control that runs on Core 1.

8.1 The Faust Code for the MIDI Controlled Volume

Here is the Faust code for the MIDI controlled stereo volume. Note that the metadata string "[midi:ctrl 2]" is used to map MIDI continuous controller 2 (CC-2) to control the gain slider.

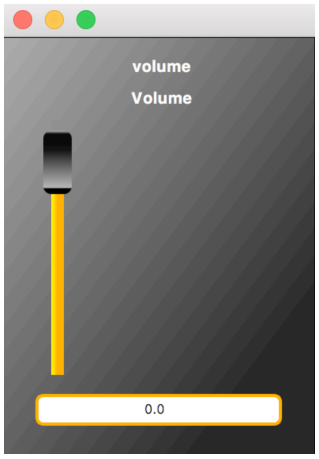
```
File Edit Options Buffers Tools Help
1 //-----------------------------------------------------
2 // Volume control in dB with MIDI control (CC-1, modWheel)
3 //-----------------------------------------------------
4
5 import("stdfaust.lib");
6
7 gain            =  vslider("Volume[midi:ctrl 2] [tooltip CC-1]", 0, -70, +4, 0.1) : ba.db2linear : si.smoo;
8
9 process         = _,_: *(gain), *(gain);


-UU-:----F1  volume.dsp     All L1    Git:master  (FAUST mode) ----------------------------------------------------
```

8.2 Building a Test GUI for the MIDI Controlled Volume

A test GUI for this Faust program can be built with the **`faust2caqt`** program:

```
% faust2caqt –midi volume.dsp
% open volume.app
```

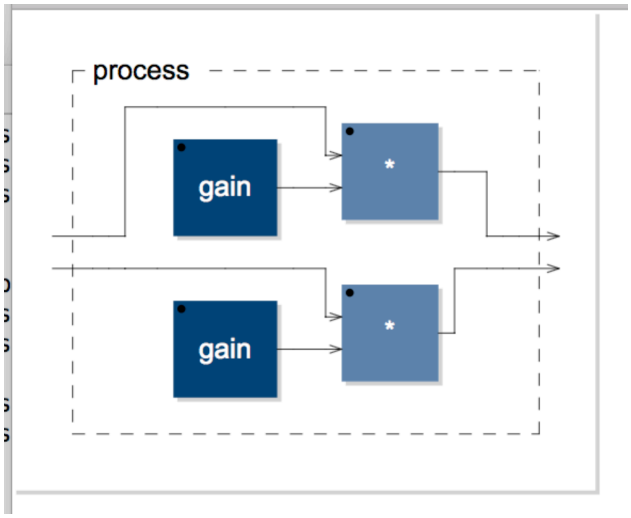
Here is what this test GUI looks like. This test program supports audio in/out and MIDI in/out and can be used to test the algorithm before committing to a SHARC Audio Module build.



8.3 Looking at the Block Diagram for the MIDI Controlled Volume

The **`faust2firefox`** command can be used to generate a hierarchical block diagram for an algorithm.

```
% faust2firefox volume.dsp
```



8.4 Running faust2sam

`faust2sam` can be run as follows:

```
% faust2sam –midi volume.dsp
% open *.zip
```

`faust2sam` will generate the following three C++ source files, which is the algorithm.

```
fast_pow2.h
samFaustDSP.cpp
samFaustDSP.h
```

8.5 The CCES Baremetal Framework

The CCES Baremetal Framework has a subproject directory for each DSP core:

```
sam_baremetal_framework_Core1
sam_baremetal_framework_Core2
```

For each project directory there is a directory where these three source files should be placed. Note that each core can be running a different Faust algorithm.

```
sam_baremetal_framework_Core1/src/faust
sam_baremetal_framework_Core2/src/faust
```

In addition there is a header file that is common across all cores called **`audio_system_config.h.`** In this file the following pre-processor variables should be set in the following way. The example below indicates that a Faust algorithm will only be running on Core1 and that Core2 will be simply passing audio to the codec.

```
#define SAM_DIY_MIDI_BOARD_PRESENT       TRUE

…

#define FAUST_INSTALLED                  TRUE

…

#define USE_FAUST_ALGORITHM_CORE1        TRUE
#define USE_FAUST_ALGORITHM_CORE2        FALSE
```

8.6 Brief Notes on Compiling and Running the Algorithm in CCES

This document will **not** go into extensive detail on how to use CCES. However here are some brief notes on how to work with this algorithm.

- Copy the Faust C++ source files to the "faust" directory for the core that the algorithm will run on.
- Set the **`audio_system_config.h`** preprocessor variables as described.
- Open the workspace with CCES.
- Use "File> Import> General> Existing Projects into Workspace" to import the projects for each core into the workspace.
- Compile with "Project> Clean…"
- Run the project on the SHARC Audio Module. Be sure that the host computer is connected to the SHARC Audio Module with an ICE-2000 interface.
- Create a Debug Configuration per the CCES instructions.
- Run the compiled algorithm on the SHARC Audio Module with Run> Debug
    - Connect a MIDI controller to the DIY board MIDI In connector
    - Connect an audio source to line-in and a speaker to line-out
    - Send MIDI continuous controller 2. The volume of the audio source should range from silent to max volume in a linear fashion.

9.0 Example Workflow – MIDI Controlled Reverb on Core 1

This second example is a MIDI controlled reverb on Core 1.

9.1 The Faust Code for the MIDI Controlled Reverb

Here is a subset the Faust code for the MIDI controlled reverb.

```
File Edit Options Buffers Tools Help
 73
 74  comb(dt, fb, damp) = (+:@(dt)) ~ (*(1-damp) : (+ ~ *(damp)) : *(fb));
 75
 76
 77  // Reverb components
 78  //-------------------
 79
 80  monoReverb(fb1, fb2, damp, spread)
 81      = _ <:  comb(combtuningL1+spread, fb1, damp),
 82              comb(combtuningL2+spread, fb1, damp),
 83              comb(combtuningL3+spread, fb1, damp),
 84              comb(combtuningL4+spread, fb1, damp),
 85              comb(combtuningL5+spread, fb1, damp),
 86              comb(combtuningL6+spread, fb1, damp),
 87              comb(combtuningL7+spread, fb1, damp),
 88              comb(combtuningL8+spread, fb1, damp)
 89          +>
 90              allpass (allpasstuningL1+spread, fb2)
 91          :   allpass (allpasstuningL2+spread, fb2)
 92          :   allpass (allpasstuningL3+spread, fb2)
 93          :   allpass (allpasstuningL4+spread, fb2)
 94          ;
 95
 96  monoReverbToStereo(fb1, fb2, damp, spread)
 97      = + <: monoReverb(fb1, fb2, damp, 0) <: _,_;
 98  stereoReverb(fb1, fb2, damp, spread)
 99      = + <:  monoReverb(fb1, fb2, damp, 0), monoReverb(fb1, fb2, damp, spread);
100  monoToStereoReverb(fb1, fb2, damp, spread)
101      = _ <: monoReverb(fb1, fb2, damp, 0), monoReverb(fb1, fb2, damp, spread);
102
103  // fxctrl : add an input gain and a wet-dry control to a stereo FX
104  //-----------------------------------------------------------------
105
106  fxctrl(g,w,Fx) =  _,_ <: (*(g),*(g) : Fx : *(w),*(w)), *(1-w), *(1-w) +> _,_;
107
108  rbp = 1-int(rsg(vslider("[0] Enable [midi:ctrl 102][style:knob]",0,0,1,1)));
-UU-:----F1  freeverb.dsp   54% L92   Git:master  (FAUST mode) ----------------------------------------------------------
```

9.2 Building a Test GUI for the MIDI Controlled Reverb

A test GUI for this Faust program can be built with the **`faust2caqt`** program:

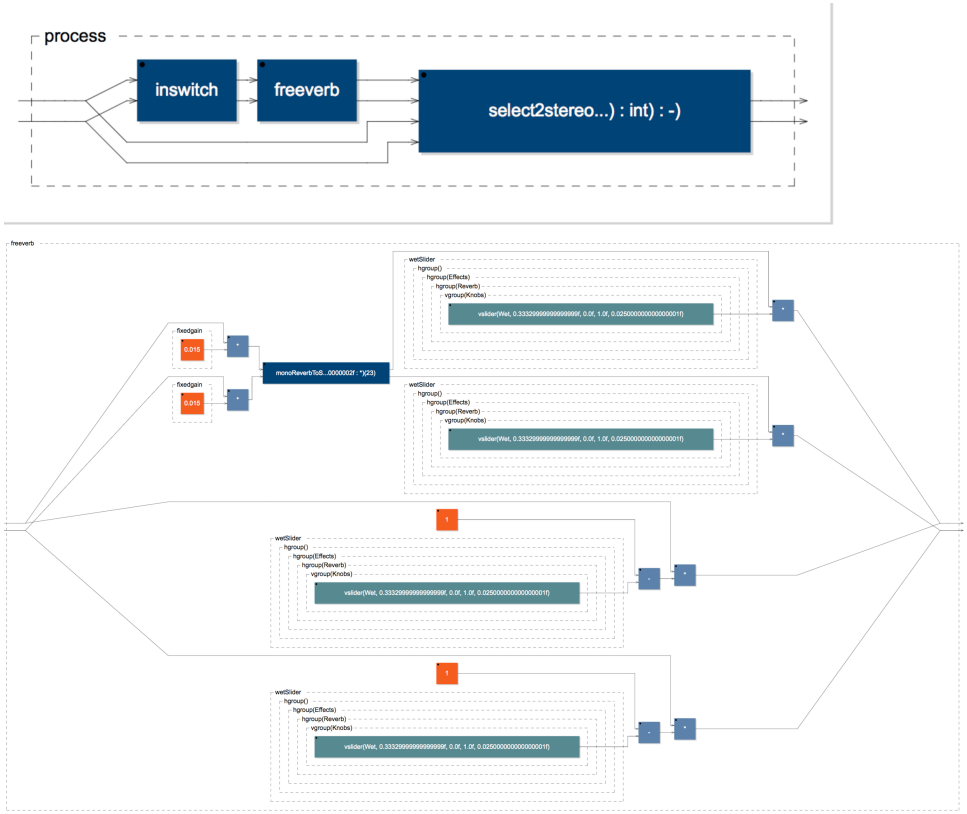
```
% faust2caqt –midi freeverb.dsp
% open freeverb.app
```

Here is what this test GUI looks like. This test program supports audio in/out and MIDI in/out and can be used to test the algorithm before committing to a SHARC Audio Module build.

9.3 Looking at the Block Diagram for the MIDI Controlled Reverb

The **`faust2firefox`** command can be used to generate a block diagram for an algorithm. This block diagram is hierarchical. Here are a few of the hierarchical levels:

```
% faust2firefox freeverb.dsp
```



9.4 Running faust2sam

`faust2sam` can be run as follows:

```
% faust2sam –midi freeverb.dsp
% open *.zip
```

`faust2sam` will generate the following three C++ source files, which is the algorithm.

```
fast_pow2.h
samFaustDSP.cpp
samFaustDSP.h
```

10.0 Example Workflow – MIDI Sawtooth Synth (Core 1)

This third example is a MIDI controlled sawtooth synth on Core 1.

10.1 The Faust Code for the MIDI Controlled Sawtooth Synth

Here is the Faust code for the MIDI controlled sawtooth synth. Notice the use of the metadata elements:

- freq – If a MIDI noteOn event is received it's MIDI keyNumber is mapped to a frequency.
- bend – if a MIDI pitchBend message is received it is mapped to a bend value.
- gain – if a MIDI noteOn message is received it's velocity value is mapped to a gain value which ranges from [0 .. 1.0]
- gate – if MIDI noteOn/noteOff messages are received they are mapped to a gate value (0/1)
- [midi:ctrl 1] – A Faust control (slider, etc) can be mapped to listen to a MIDI continuous controller.

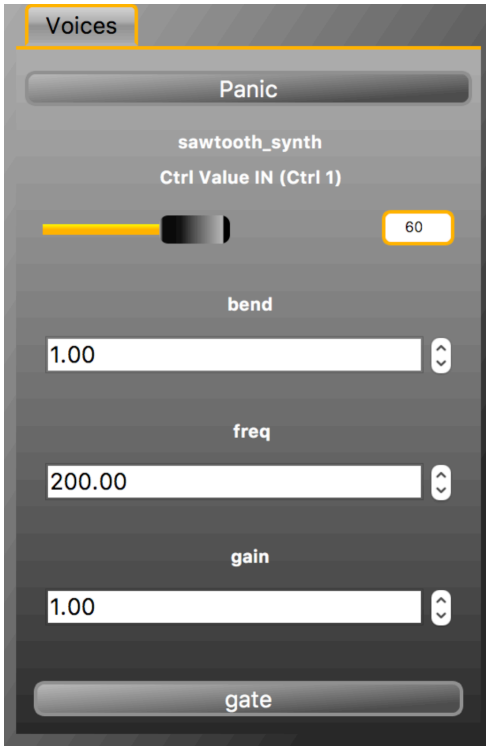
```
File Edit Options Buffers Tools Help
 1
 2  import("stdfaust.lib");
 3
 4  // Simple MIDI controlled sawtooth synth
 5
 6  normMIDI(mv)   = mv/127.0;
 7  vol  = normMIDI(hslider("Ctrl Value IN (Ctrl 1) [midi:ctrl 1]", 60, 0, 127, 1)) ;
 8
 9
10  f = nentry("freq",200,40,2000,0.01);
11  bend = nentry("bend",1,0,10,0.01) : si.polySmooth(t,0.999,1);
12  g = nentry("gain",1,0,1,0.01);
13  t = button("gate");
14  freq = f*bend;
15  envelope = t*g*vol : si.smoo;
16
17  process = os.sawtooth(freq)*envelope <: _,_;


-UU-:**--F1  sawtooth_synth.dsp   All L5    Git:master  (FAUST mode) ----------------------
```

10.2 Building a Test GUI for the MIDI Controlled Sawtooth Synth

A test GUI for this Faust program can be built with the **`faust2caqt`** program:

```
% faust2caqt –midi –nvoices 6 sawtooth_synth.dsp
% open sawtooth_synth.app
```

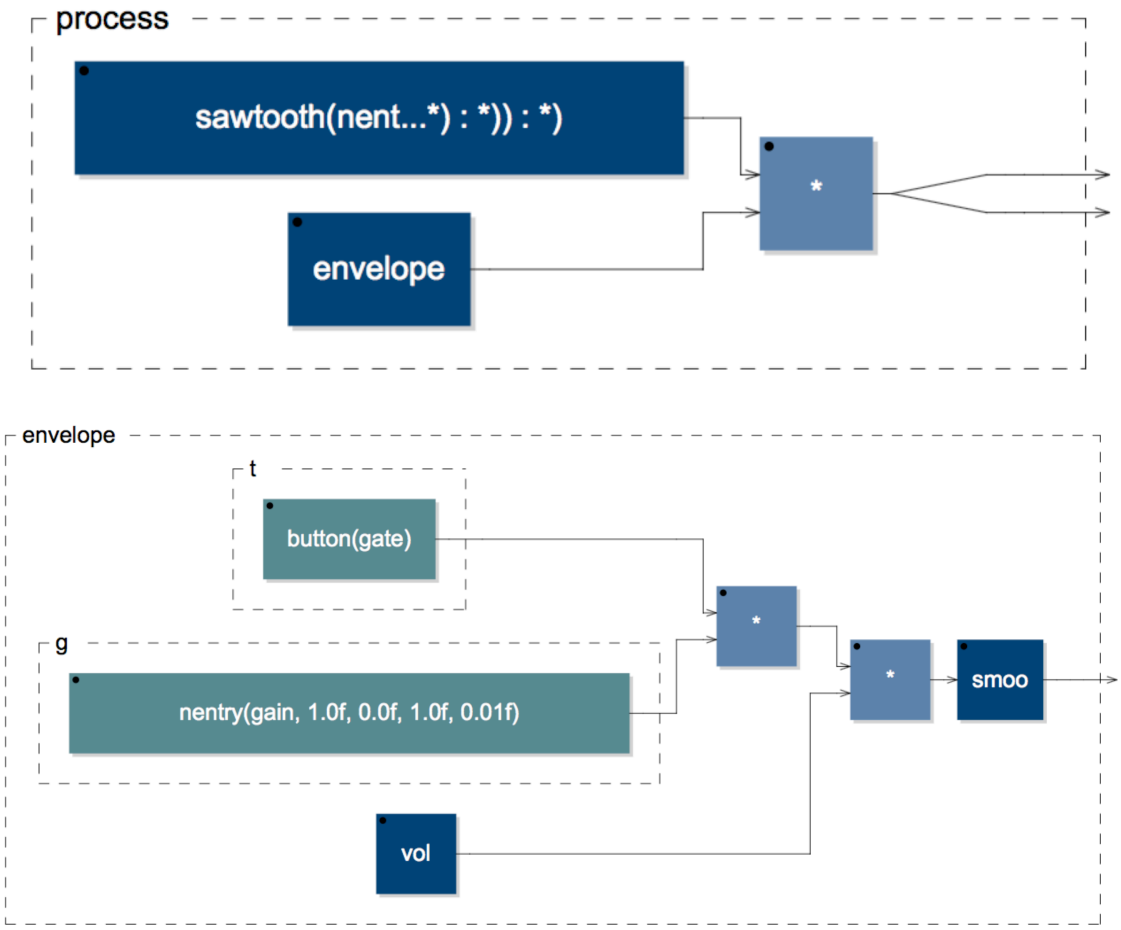
Here is what this test GUI looks like. This test program supports audio in/out and MIDI in/out and can be used to test the algorithm before committing to a SHARC Audio Module build.



10.3 Looking at the Block Diagram for the MIDI Controlled Sawtooth

The **`faust2firefox`** command can be used to generate a hierarchical block diagram for an algorithm. This block diagram is hierarchical. Here are a few of the hierarchical levels:

```
% faust2firefox sawtooth_synth.dsp
```



10.4 Running faust2sam

faust2sam can be run as follows:

```
% faust2sam –midi –nvoices 6 sawtooth_synth.dsp
% open *.zip
```

faust2sam will generate the following three C++ source files, which is the algorithm.

```
fast_pow2.h
samFaustDSP.cpp
samFaustDSP.h
```

11.0 Example Workflow – MIDI Virtual Analog Synth (Core1) /Effects (Core 2)

For this final example the full virtual analog synth is run on Core1 and the effects chain is run on Core 2.

11.1 The Faust Code for the Virtual Analog Synth (Core 1)

Here is a subset of the Faust code for the virtual analog synth:

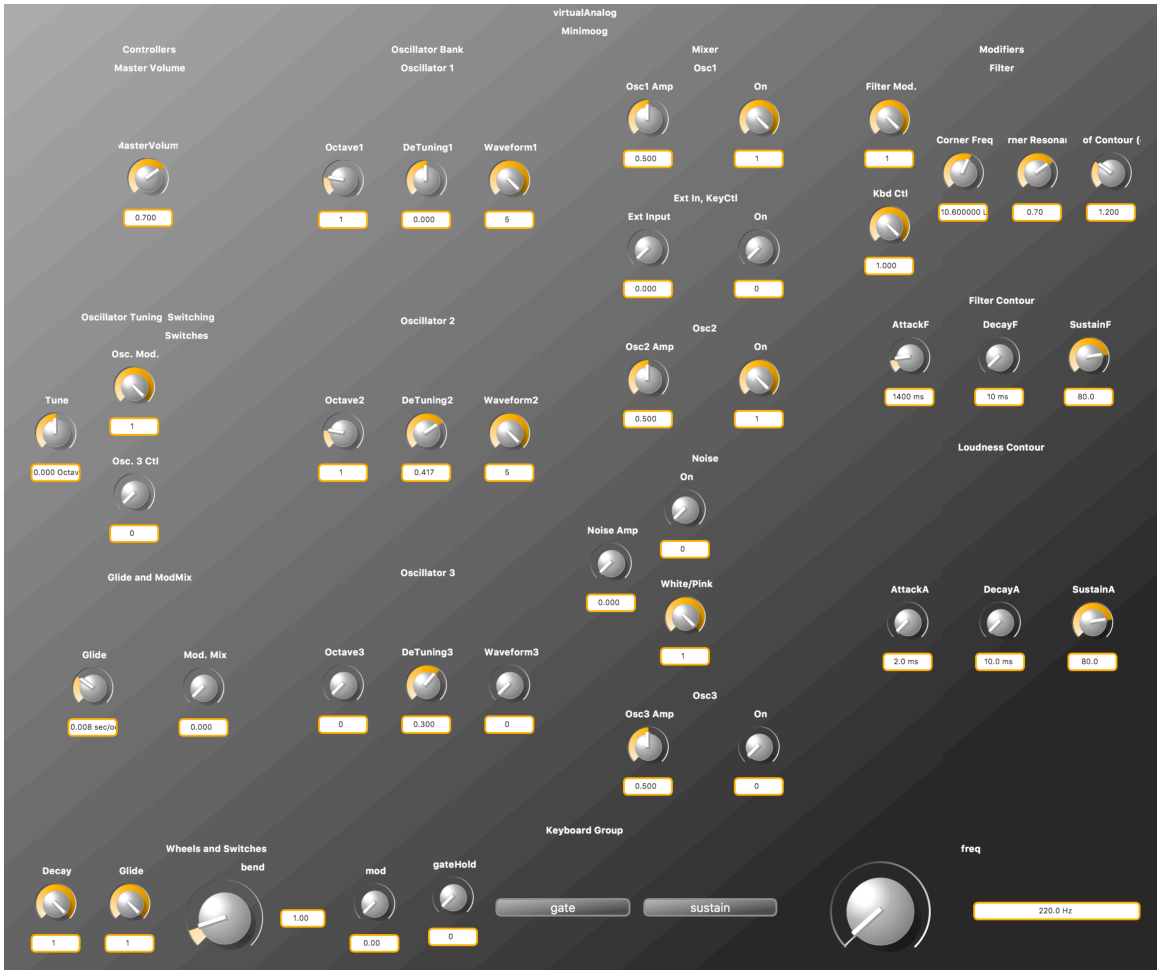
```
File Edit Options Buffers Tools Help
    waveforms(i) = (tri(i), bent(i), saw(i), sq(i), ptm(i), ptn(i));

    // compute oscillator frequency scale factor, staying in lg(Hz) as much as possible:
    modWheelShift = 1.5*modWheel; // Manual says 0 to 1.5 octaves
    modulationCenterShift = 0; // Leave this off until triangle-wave modulation is debugged
    modulationShift = select2(oscModEnable, 0.0,
                              modWheelShift * ( modulationCenterShift + (1.0-modulationCenterShift) * oscNoiseModulation ));
    octaveShift(i) = -2+int(octaveSelect(i));
    osc3FixedFreq = 369.994; // F# a tritone above middle C
    keyFreqGlidedMaybe = select2(osc3Control,osc3FixedFreq,keyFreqGlided);
    keyFreqModulatedShifted(3) = keyFreqGlidedMaybe; // osc3 not allowed to FM itself
    keyFreqModulatedShifted(i) = keyFreqGlided * pow(2.0, modulationShift); // i=1,2

    // When disconnected from the keyboard, Osc3 can detune 3 octaves up or down (Pat video):
    detuneBoost(3) = select2(osc3Control,3.0,1.0);
    detuneBoost(i) = 1.0; // i=1,2
    detuneOctavesFinal(i) = detuneOctaves(i)*detuneBoost(i);

    fBase(i) = keyFreqModulatedShifted(i) * pow(2.0, (masterTuneOctaves+octaveShift(i)+detuneOctavesFinal(i)))
                    : si.smooth(ba.tau2pole(0.016));
    fLFOBase(i) = 3.0 * pow(2.0, detuneOctavesFinal(i)); // used when osc3 (only) is in LFO mode
    lfoMode(i) = (octaveSelect(i) == 0);
    f(i) = select2(lfoMode(i), fBase(i), fLFOBase(i)); // lowest range setting is LFO mode for any osc

    // i is 1-based:
    osc(i) = ba.selectn(6, int(waveSelect(i)), tri(i), bent(i), saw(i), sq(i), ptm(i), ptn(i));
    tri(i) = select2(lfoMode(i),
              os.triangle(f(i)),
              os.lf_triangle(f(i)));
    bent(i) = 0.5*tri(i) + 0.5*saw(i); // from Minimoog manual
    saw(i) = select2(lfoMode(i),
              os.sawtooth(f(i)),
              os.lf_saw(f(i)));
    sq(i) = select2(lfoMode(i),
-UU-:----F1  virtualAnalog.dsp   26% L69    (FAUST mode) ----------------------------------------------------------------------------
```

11.2 Building a Test GUI for the Virtual Analog Synth

A test GUI for this Faust program can be built with the **`faust2caqt`** program:

```
% faust2caqt –midi –nvoices 1 virtualAnalog.dsp
% open virtualAnalog.app
```

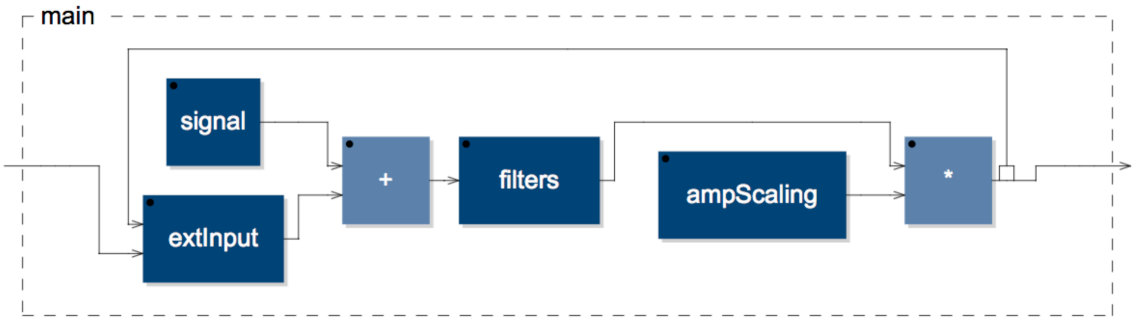
Here is what this test GUI looks like. This test program supports audio in/out and MIDI in/out and can be used to test the algorithm before committing to a SHARC Audio Module build.

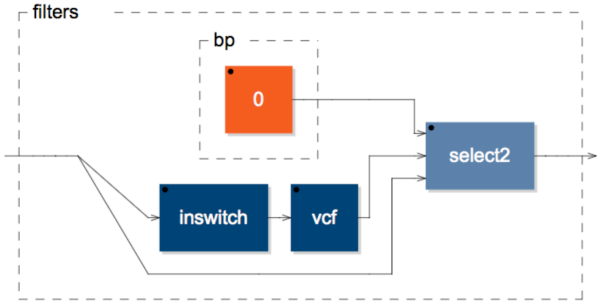


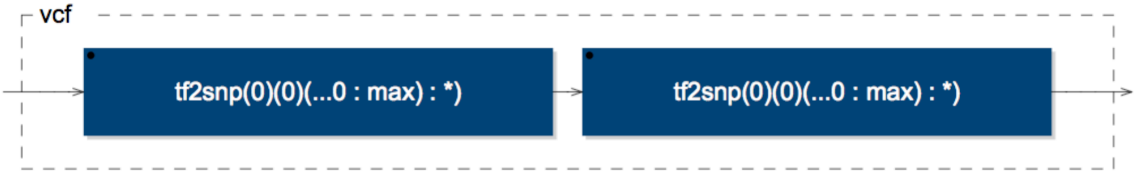
11.3 Looking at the Block Diagram for the Virtual Analog Synth

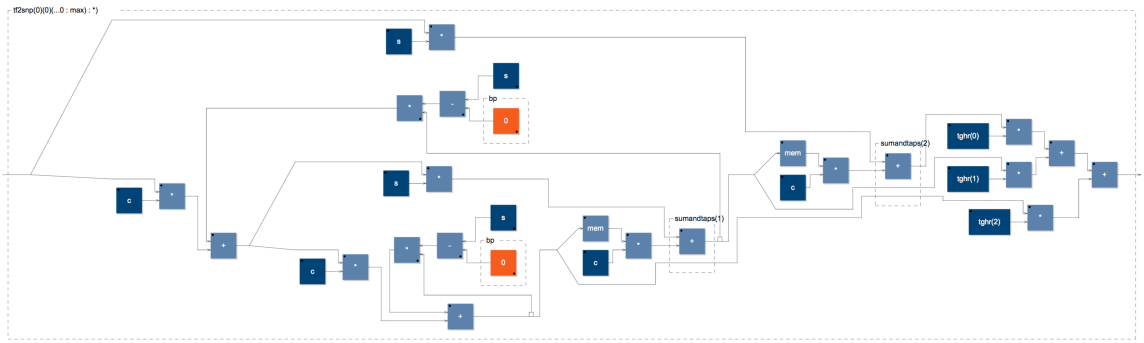
The **`faust2firefox`** command can be used to generate a hierarchical block diagram for an algorithm. This block diagram is hierarchical. Here are a few of the hierarchical levels:

```
% faust2firefox virtualAnalog.dsp
```









11.4 Running faust2sam

faust2sam can be run as follows:

```
% faust2sam –midi –nvoices 1 virtualAnalog.dsp
% open *.zip
```

faust2sam will generate the following three C++ source files, which is the algorithm.

```
fast_pow2.h
samFaustDSP.cpp
samFaustDSP.h
```

11.5 The Faust Code for the Effects Chain (Core 2)

The effects chain (echo: flanger: chorus: reverb) is run on Core 2.

Here is a subset of the Faust code for the effects chain (echo: flanger: chorus: reverb).

```
File Edit Options Buffers Tools Help
 1
 2  // All effects used by minimoog.dsp
 3
 4  import("stdfaust.lib");
 5  import("layout2.dsp");
 6
 7  process = _,_ : +
 8          : component("echo.dsp")
 9          : component("flanger.dsp")
10          : component("chorus.dsp")
11          : component("freeverb.dsp");
```

```
-UU-:**--F1  effects.dsp     All L2     (FAUST mode) ------------
```

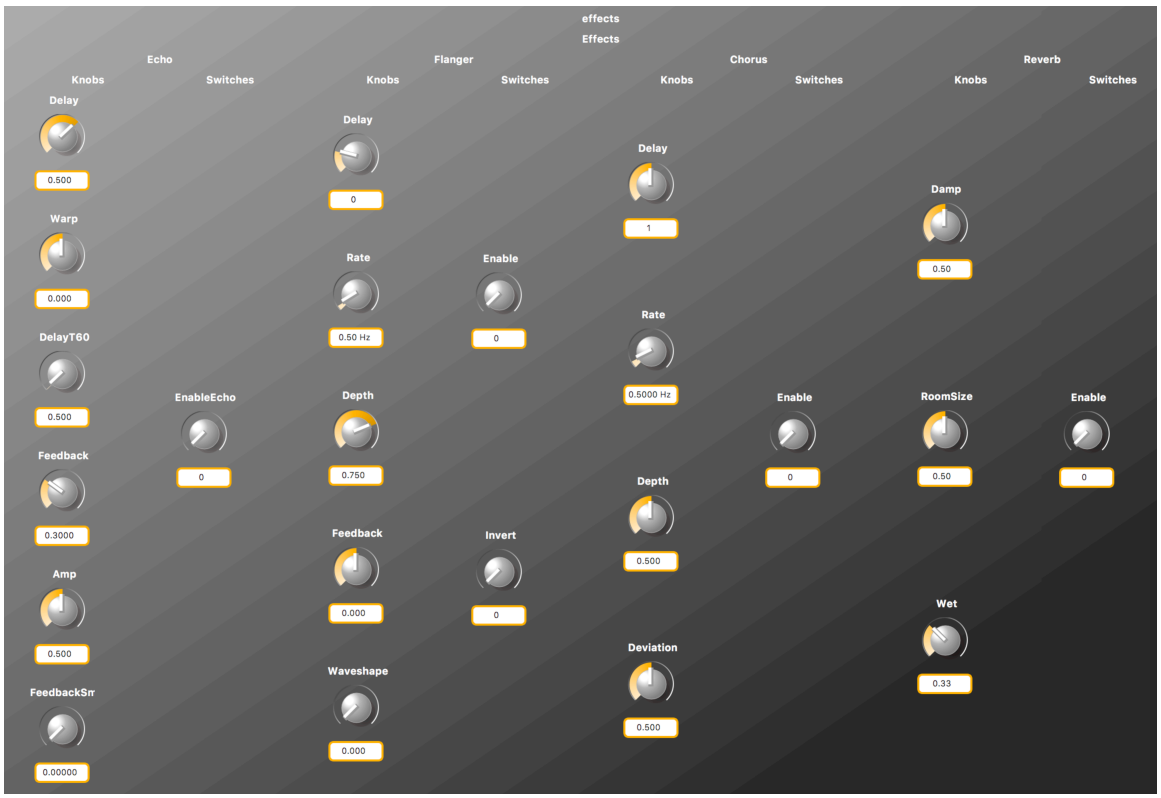
```
File Edit Options Buffers Tools Help
 1  // imported by echo.dsp and echomt.dsp
 2
 3  import("stdfaust.lib");
 4  import("layout2.dsp");
 5
 6  echo_group(x) = x; // Let layout2.dsp lay us out
 7     knobs_group(x) = ekg(x);
 8     switches_group(x)  = esg(x);
 9
10  dmax = 32768; // one and done
11  dmaxs = float(dmax)/44100.0;
12
13  Nnines = 1.8; // Increase until you get the desired maximum amount of smoothing when fbs==1
14  fastpow2 = ffunction(float fastpow2(float), "fast_pow2.h", "");
15  fbspr(fbs) = 1.0 - fastpow2(-3.33219*Nnines*fbs); // pole radius of feedback smoother
16  inputSelect(gi) = _,0 : select2(gi);
17  echo_mono(dmax,curdel,tapdel,fb,fbspr,gi) = inputSelect(gi) : (+:si.smooth(fbspr)
18                                      <: de.fdelay(dmax,curdel),
19                                         de.fdelay(dmax,tapdel))
20                                         ~(*(fb),!) : !,_;
21
22  tau2pole(tau) = ba.if(tau>0, exp(-1.0/(tau*ma.SR)), 0.0);
23  t60smoother(dEchoT60) = si.smooth(tau2pole(dEchoT60/6.91));
24
-UU-:----F1  echo.dsp       Top L2     (FAUST mode) ----------------------------------------------
```

11.6 Building a Test GUI for the Effects Chain

A test GUI for this Faust program can be built with the **`faust2caqt`** program:

```
% faust2caqt –midi effects.dsp
% open effects.app
```

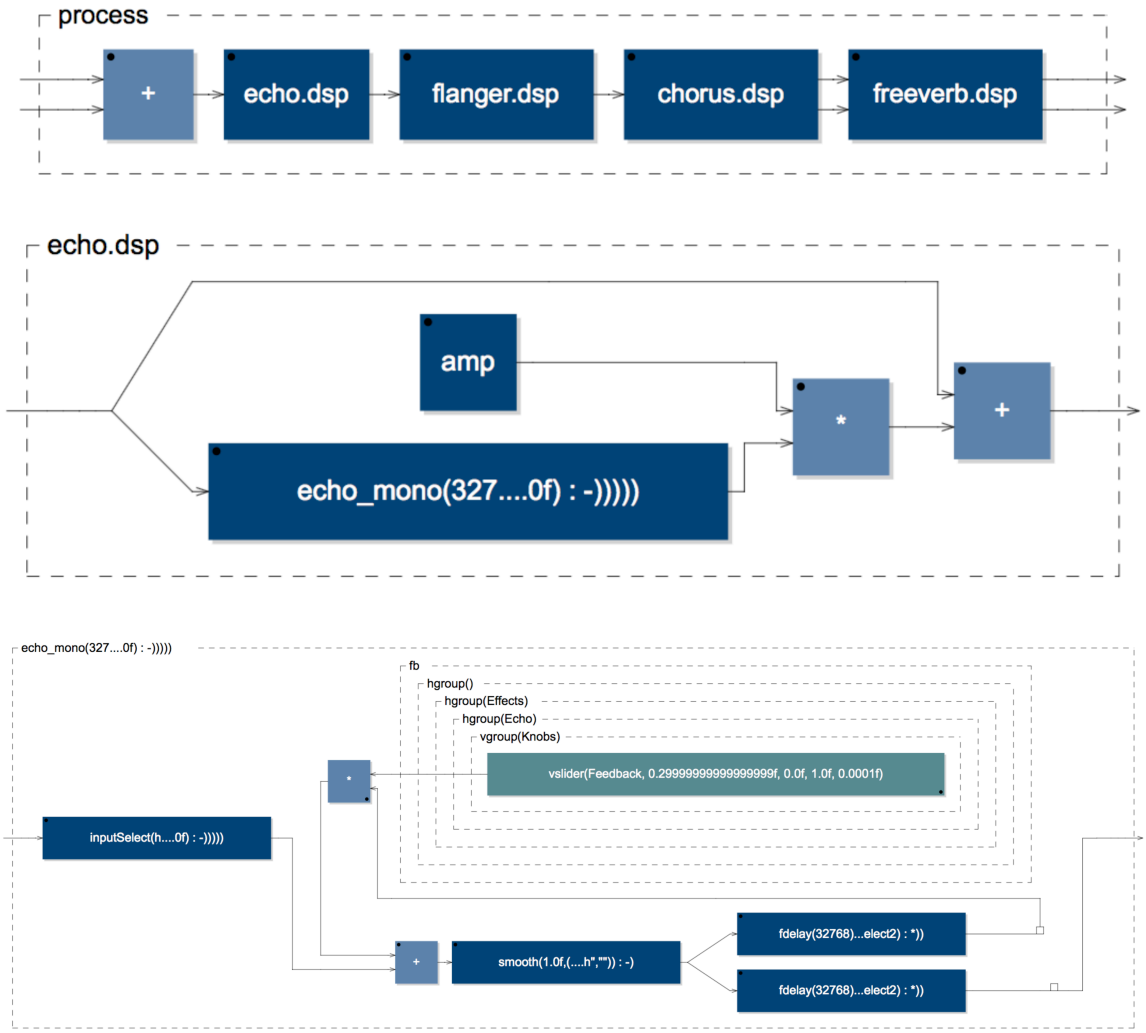
Here is what this test GUI looks like. This test program supports audio in/out and MIDI in/out and can be used to test the algorithm before committing to a SHARC Audio Module build.



11.7 Looking at the Block Diagram for the Effects Chain

The **`faust2firefox`** command can be used to generate a hierarchical block diagram for an algorithm. This block diagram is hierarchical. Here are a few of the hierarchical levels:

```
% faust2firefox effects.dsp
```



11.8 Running faust2sam

faust2sam can be run as follows:

```
% faust2sam –midi effects.dsp
% open *.zip
```

faust2sam will generate the following three C++ source files, which is the algorithm.

```
fast_pow2.h
samFaustDSP.cpp
samFaustDSP.h
```

11.9 The CCES Baremetal Framework

The CCES Baremetal Framework has a subproject directory for each DSP core:

```
sam_baremetal_framework_Core1/src/faust
sam_baremetal_framework_Core2/src/faust
```

The virtualAnalog C++ files should be placed in the "faust" directory for Core 1 and the effects chain C++ files should be placed in the "faust" directory for Core 2.

In **audio_system_config.h.** In this file the following pre-processor variables should be set. This indicates that Faust algorithms will be running on Core1 **and** Core2.

```
#define SAM_DIY_MIDI_BOARD_PRESENT       TRUE

…

#define FAUST_INSTALLED                  TRUE

…

#define USE_FAUST_ALGORITHM_CORE1        TRUE
#define USE_FAUST_ALGORITHM_CORE2        FALSE
```

11.10 MIDI Assignments for the Virtual Analog Algorithm

Function	MIDI CC	Module	Type	Notes
Tune	47	1 - Controllers	knob	Master tuning
Glide	5	1 - Controllers	knob	Portamento time
Modulation Mix	48	1 - Controllers	knob	Modulation mix between OSC3 and Noise
Oscillator Modulation	22	2 - Oscillator Bank	switch	Enable modulation control of OSC frequencies
OSC 1 Range	23	2 - Oscillator Bank	knob	OSC 1 range
OSC 1 Detune	24	2 - Oscillator Bank	knob	OSC 1 detuning
OSC 1 Waveform	25	2 - Oscillator Bank	knob	OSC 1 waveform shape
OSC 2 Range	28	2 - Oscillator Bank	knob	OSC 2 range
OSC 2 Detune	29	2 - Oscillator Bank	knob	OSC 2 detuning
OSC 2 Waveform	30	2 - Oscillator Bank	knob	OSC 2 waveform shape
OSC 3 Range	33	2 - Oscillator Bank	knob	OSC 3 range
OSC 3 Detune	34	2 - Oscillator Bank	knob	OSC 3 detuning
OSC 3 Waveform	35	2 - Oscillator Bank	knob	OSC 3 waveform shape
OSC 3 Control	9	2 - Oscillator Bank	switch	OSC 3 as a control signal or as an audio source
OSC 1 Amp	26	3 - Mixer	knob	OSC 1 gain
Osc1 mixer switch	12	3 - Mixer	switch	OSC 1 enable
OSC 2 Amp	31	3 - Mixer	knob	OSC 2 gain
Osc2 mixer switch	14	3 - Mixer	switch	OSC 2 enable
OSC3 Amp	36	3 - Mixer	knob	OSC 3 gain
OSC3 mixer switch	17	3 - Mixer	switch	OSC 3 enable
External Input amp	27	3 - Mixer	knob	Extrenal input gain
External Input mixer switch	13	3 - Mixer	switch	External input enable
Noise Amp	32	3 - Mixer	knob	Noise gain
Noise mixer switch	15	3 - Mixer	switch	Noise Enable
White/pink toggle	16	3 - Mixer	switch	Noise pink/white

Filter Modulation Enable	19	4 - Filter	switch	Enable modulation control of the filter cutoff frequency
Keyboard Range	38	4 - Filter	knob	Add keyboard control of the filter cutoff frequency
Cutoff Frequency	74	4 - Filter	knob	Filter cutoff frequency
Emphasis	37	4 - Filter	knob	Filter Resonance (Q)
Amount of Contour	39	4 - Filter	knob	Amount of envelope generator
Attack Time	40	4 - Filter	knob	VCF envelope generator attack time
Decay Time	41	4 - Filter	knob	VCF envelope generator decay time
Sustain Level	42	4 - Filter	knob	VCF envelope generator sustain level
Attack Time	43	5 - Loudness Contour	knob	VCA envelope generator attack time
Decay Time	44	5 - Loudness Contour	knob	VCA envelope generator decay time
Sustain Level	45	5 - Loudness Contour	knob	VCA envelope generator sustain level
Decay	20	6 - Keyboard	switch	Enables using the decay stage as a release stage
Glide	65	6 - Keyboard	switch	Enable portamento
Pitch Wheel	pitchWheel	6 - Keyboard	knob	Pitch Wheel
Mod Wheel	1	6 - Keyboard	knob	Modulation Wheel control
Sustain	64	midi sustain foot pedal	switch	Note sustain, preempts the VCF/VCA release stage
Master Volume	7	5 - Output	knob	Master volume

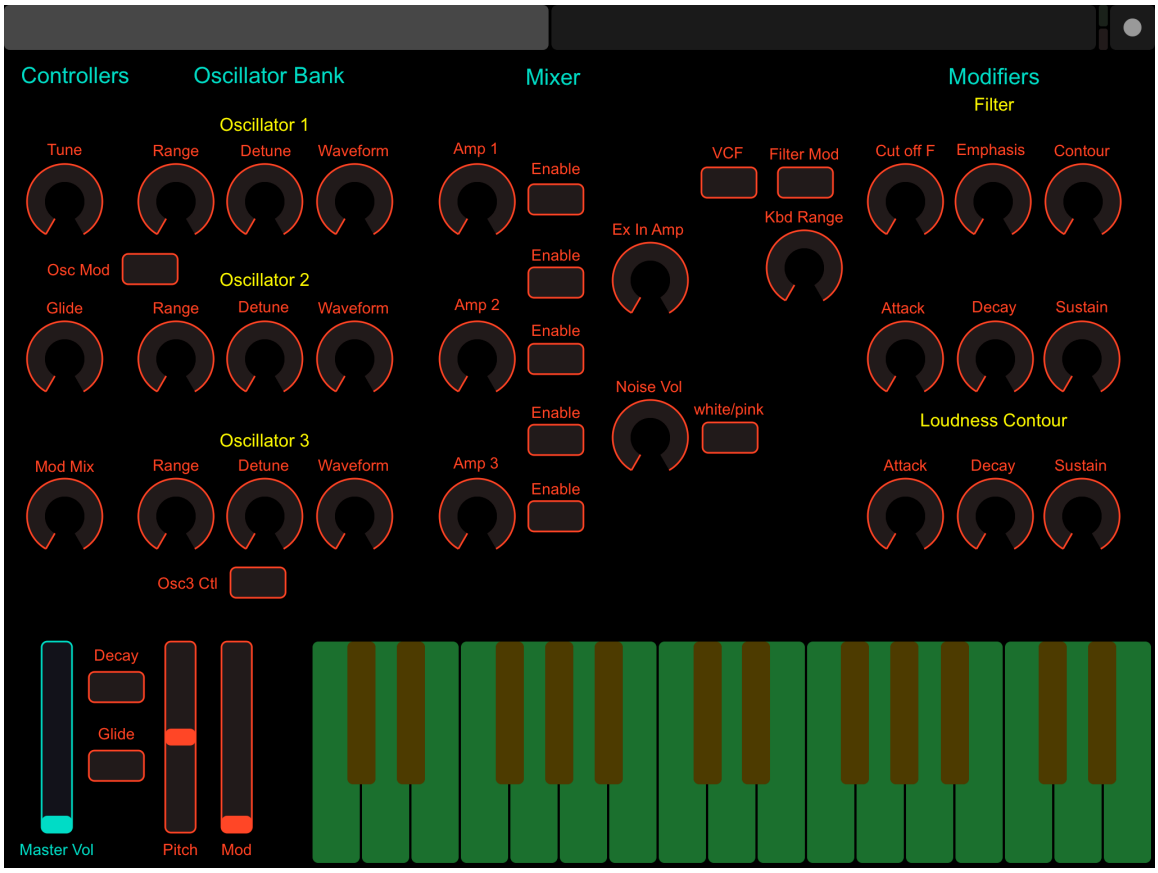
11.11 MIDI Assignments for the Effects Chain Algorithm

Function	MIDI CC	Module	Type
Invert	49	Flanger	knob
Enable	102	Flanger	switch
Delay	50	Flanger	knob
Rate	51	Flanger	knob
Depth	52	Flanger	knob
Feedback	53	Flanger	knob
Wave Shape	54	Flanger	knob
Delay	55	Chorus	knob
Enable	103	Chorus	switch
Rate	56	Chorus	knob
Depth	57	Chorus	knob
Deviation	58	Chorus	knob
enable	105	Echo	switch
Delay Portamento	60	Echo	knob
Delay	61	Echo	knob
Warp	62	Echo	knob
Feedback	2	Echo	knob
Amp	75	Echo	knob
Feedback sm?	76	Echo	knob
Damp	3	Reverb	knob
Enable	104	Reverb	switch
Room Size	4	Reverb	knob
Wet Dry	79	Reverb	knob

11.12 The Virtual Analog/Effects Chain TouchOSC UI

TouchOSC is a mobile app that can be used to create arbitrary GUIs that send MIDI values. TouchOSC is available for both iOS and Android. A TouchOSC configuration is provided for the Virtual Analog and Effects chain algorithms (virtualAnalog.touchOSC)

The Virtual Analog Page:



The Effects Chain Page:

12.0 Conclusion

The Faust library provides a rich set of audio DSP objects that can be used in creating DSP algorithms. Using Faust, it is possible to quickly create large algorithms that take advantage of the computing power available on the SHARC Audio Module platform.