FAUST Quick Reference
(version 2.72.13)

GRAME
Centre National de Création Musicale

March 31, 2024

Contents

Chapter 1

Introduction

FAUST (*Functional Audio Stream*) is a functional programming language specifically designed for real-time signal processing and synthesis. FAUST targets high-performance signal processing applications and audio plug-ins for a variety of platforms and standards.

1.1 Design Principles

Various principles have guided the design of FAUST:

- FAUST is a *specification language*. It aims at providing an adequate notation to describe *signal processors* from a mathematical point of view. FAUST is, as much as possible, free from implementation details.

- FAUST programs are fully compiled, not interpreted. The compiler translates FAUST programs into equivalent C++ programs taking care of generating the most efficient code. The result can generally compete with, and sometimes even outperform, C++ code written by seasoned programmers.

- The generated code works at the sample level. It is therefore suited to implement low-level DSP functions like recursive filters. Moreover the code can be easily embedded. It is self-contained and doesn't depend of any DSP library or runtime system. It has a very deterministic behavior and a constant memory footprint.

- The semantic of FAUST is simple and well defined. This is not just of academic interest. It allows the FAUST compiler to be *semantically driven*. Instead of compiling a program literally, it compiles the mathematical function it denotes. This feature is useful for example to promote components reuse while preserving optimal performance.

- FAUST is a textual language but nevertheless block-diagram oriented. It actually combines two approaches: *functional programming* and *algebraic block-diagrams*. The key idea is to view block-diagram construction as function composition. For that purpose, FAUST relies on a *block-diagram algebra* of five composition operations (`:` `,` `~` `<:` `:>`).
- Thanks to the notion of *architecture*, FAUST programs can be easily deployed on a large variety of audio platforms and plugin formats without any change to the FAUST code.

1.2 Signal Processor Semantic

A FAUST program describes a *signal processor*. The role of a *signal processor* is to transforms a (possibly empty) group of *input signals* in order to produce a (possibly empty) group of *output signals*. Most audio equipments can be modeled as *signal processors*. They have audio inputs, audio outputs as well as control signals interfaced with sliders, knobs, vu-meters, etc.

More precisely :

FAUST considers two type of signals: *integer signals* ($s : \mathbb{Z} \rightarrow \mathbb{Z}$) and *floating point signals* ($s : \mathbb{Z} \rightarrow \mathbb{Q}$) Exchanges with the outside world are, by convention, made using floating point signals. The full range is represented by sample values between -1.0 and +1.0.

- A *signal* s is a discrete function of time $s : \mathbb{Z} \rightarrow \mathbb{R}$. The value of a signal s at time t is written $s(t)$. The values of signals are usually needed starting from time 0. But to take into account *delay operations*, negative times are possible and are always mapped to zeros. Therefore for any FAUST signal s we have $\forall t < 0, s(t) = 0$. In operational terms this corresponds to assuming that all delay lines are signals initialized with 0s.
- The set of all possible signals is $\mathbb{S} = \mathbb{Z} \rightarrow \mathbb{R}$.
- A group of n signals (a *n*-tuple of signals) is written $(s_1, \dots, s_n) \in \mathbb{S}^n$. The *empty tuple*, single element of $\mathbb{S}^0$ is notated $()$.
- A *signal processors* p, is a function from *n*-tuples of signals to *m*-tuples of signals $p : \mathbb{S}^n \rightarrow \mathbb{S}^m$. The set $\mathbb{P} = \bigcup_{n,m} \mathbb{S}^n \rightarrow \mathbb{S}^m$ is the set of all possible signal processors.

As an example, let's express the semantic of the FAUST primitive `+`. Like any FAUST expression, it is a signal processor. Its signature is $\mathbb{S}^2 \rightarrow \mathbb{S}$. It takes two input signals X_0 and X_1 and produce an output signal Y such that $Y(t) = X_0(t) + X_1(t)$.

Numbers are signal processors too. For example the number 3 has signature $\mathbb{S}^0 \rightarrow \mathbb{S}$. It takes no input signals and produce an output signal Y such that $Y(t) = 3$.

Chapter 2

Compiling and installing Faust

The Faust source distribution `faust-2.72.13.tar.gz` can be downloaded from GitHub (https://github.com/grame-cncm/faust/releases).

2.1 Organization of the distribution

The first thing is to decompress the downloaded archive.

```
tar xzf faust-2.72.13.tar.gz
```

The resulting `faust-2.72.13/` folder should contain the following elements:

`architecture/`	Faust libraries and architecture files
`benchmark`	tools to measure the efficiency of the generated code
`compiler/`	sources of the Faust compiler
`examples/`	examples of Faust programs
`syntax-highlighting/`	support for syntax highlighting for several editors
`documentation/`	Faust's documentation, including this manual
`tools/`	tools to produce audio applications and plugins
`COPYING`	license information
`Makefile`	Makefile used to build and install Faust
`README`	instructions on how to build and install Faust

2.2 Compilation

FAUST has no dependencies outside standard libraries. Therefore the compilation should be straightforward. There is no configuration phase, to compile the FAUST compiler simply do :

```
cd faust-2.72.13/
make
```

If the compilation was successful you can test the compiler before installing it:

```
[cd faust-2.72.13/]
./build/bin/faust -v
```

It should output:

```
FAUST Version 2.72.13
Embedded backends:
   DSP to C
   DSP to C++
   DSP to Cmajor
   DSP to Codebox
   DSP to CSharp
   DSP to DLang
   DSP to Java
   DSP to JAX
   DSP to Julia
   DSP to old C++
   DSP to Rust
   DSP to WebAssembly (wast/wasm)
Copyright (C) 2002-2024, GRAME - Centre National de
   Creation Musicale. All rights reserved.
```

Then you can also try to compile one of the examples :

```
[cd faust-2.72.13/]
./build/bin/faust examples/generator/noise.dsp
```

It should produce some C++ code on the standard output.

2.3 Installation

You can install FAUST with:

```
[cd faust-2.72.13/]
sudo make install
```

or

```
[cd faust-2.72.13/]
su
make install
```

depending on your system.

2.4 Compilation of the examples

Once Faust correctly installed, you can have a look at the provided examples in the `examples/` folder.

You can use any of the `faust2...` script installed on your system (go in `/tools/faust2appls` to get an exhaustive list) to compile the Faust codes available in this folder. For example, if you're a Mac user and you want to turn `filtering/vcfWahLab.dsp` into a standalone CoreAudio application with a QT interface, just run:

```
faust2caqt filtering/vcfWahLab.dsp
```

Chapter 3

Faust syntax

This section describes the syntax of Faust. Figure 3.1 gives an overview of the various concepts and where they are defined in this section.

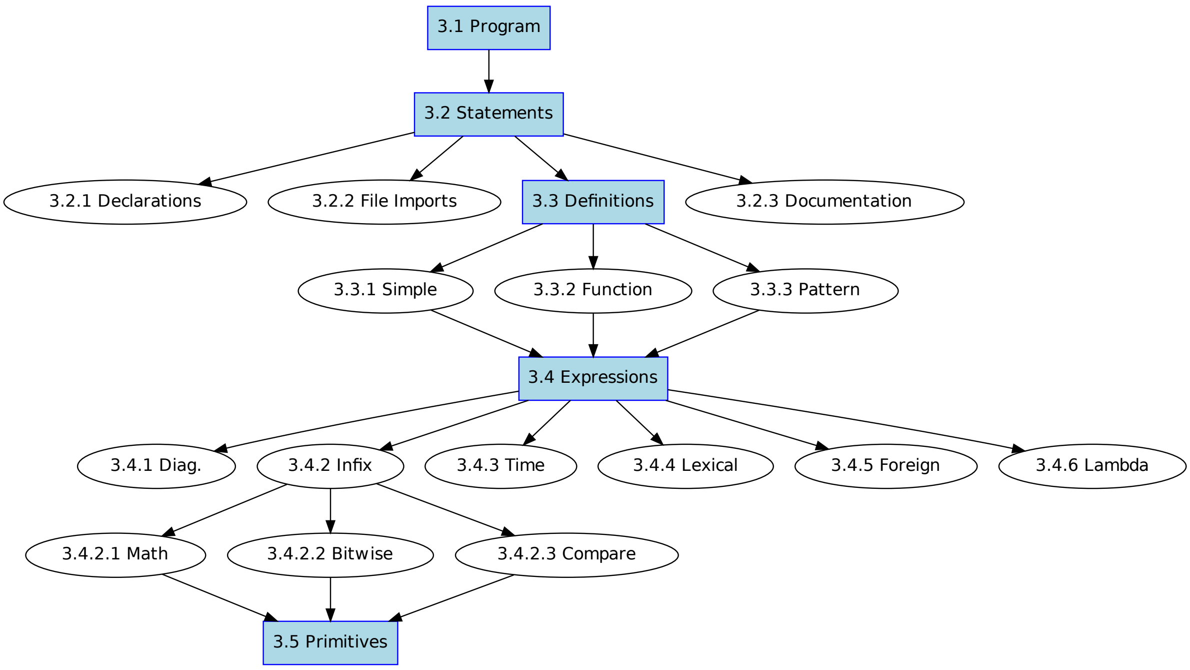


Figure 3.1: Overview of Faust syntax

As we will see, *definitions* and *expressions* have a central role.

3.1 Faust program

A Faust program is essentially a list of *statements*. These statements can be *declarations*, *imports*, *definitions* and *documentation tags*, with optional C++ style (//... and /*...*/) comments.

3.1.1 A Simple Program

Here is a short Faust program that implements of a simple noise generator. It exhibits various kind of statements: two *declarations*, an *import*, a *comment* and a *definition*. We will see later on *documentation* statements (3.2.3).

```
declare name        "noise";
declare copyright   "(c)GRAME 2006";

import("music.lib");

// noise level controlled by a slider
process = noise * vslider("volume", 0, 0, 1, 0.1);
```

The keyword `process` is the equivalent of `main` in C/C++. Any Faust program, to be valid, must at least define `process`.

3.2 Statements

The *statements* of a Faust program are of four kinds: *metadata declarations*, *file imports*, *definitions* and *documentation*. All statements but documentation end with a semicolon (`;`).

3.2.1 Declarations

Meta-data declarations (for example `declare name "noise";`) are optional and typically used to document a Faust project.

Contrary to regular comments, these declarations will appear in the C++ code generated by the compiler. A good practice is to start a FAUST program with some standard declarations:

```
declare name "MyProgram";
declare author "MySelf";
declare copyright "MyCompany";
declare version "1.00";
declare license "BSD";
```

3.2.2 Imports

File imports allow to import definitions from other source files.

For example `import("maths.lib");` imports the definitions of the `maths.lib` library, a set of additional mathematical functions provided as foreign functions.

3.2.3 Documentation

Documentation statements are optional and typically used to control the generation of the mathematical documentation of a FAUST program. This documentation system is detailed chapter **??**. In this section we will essentially describe the documentation statements syntax.

A documentation statement starts with an opening <mdoc> tag and ends with a closing </mdoc> tag. Free text content, typically in LaTeX format, can be placed in between these two tags.

Moreover, optional sub-tags can be inserted in the text content itself to require the generation, at the insertion point, of mathematical *equations*, graphical *block-diagrams*, FAUST source code *listing* and explanation *notice*.

The generation of the mathematical equations of a FAUST expression can be requested by placing this expression between an opening `<equation>` and a closing `</equation>` tag. The expression is evaluated within the lexical context of the FAUST program.

Similarly, the generation of the graphical block-diagram of a FAUST expression can be requested by placing this expression between an opening `<diagram>` and a closing `</diagram>` tag. The expression is evaluated within the lexical context of the FAUST program.

The `<metadata>` tags allow to reference FAUST metadatas (cf. declarations), calling the corresponding keyword.

The `<notice />` empty-element tag is used to generate the conventions used in the mathematical equations.

The `<listing />` empty-element tag is used to generate the listing of the FAUST program. Its three attributes `mdoctags`, `dependencies` and `distributed` enable or disable respectively `<mdoc>` tags, other files dependencies and distribution of interleaved faust code between `<mdoc>` sections.

3.3 Definitions

A *definition* associates an identifier with an expression it stands for.

Definitions are essentially a convenient shortcut avoiding to type long expressions. During compilation, more precisely during the evaluation stage, identifiers are replaced by their definitions. It is therefore always equivalent to use an identifier or directly its definition. Please note that multiple definitions of a same identifier are not allowed, unless it is a pattern matching based definition.

3.3.1 Simple Definitions

The syntax of a simple definition is:

For example here is the definition of `random`, a simple pseudo-random number generator:

```
random = +(12345) ~ *(1103515245);
```

3.3.2 Function Definitions

Definitions with formal parameters correspond to functions definitions.

For example the definition of `linear2db`, a function that converts linear values to decibels, is :

```
linear2db(x) = 20*log10(x);
```

Please note that this notation is only a convenient alternative to the direct use of *lambda-abstractions* (also called anonymous functions). The following is an equivalent definition of `linear2db` using a lambda-abstraction:

```
linear2db = \(x).(20*log10(x));
```

3.3.3 Definitions with pattern matching

Moreover, formal parameters can also be full expressions representing patterns.

This powerful mechanism allows to algorithmically create and manipulate block diagrams expressions. Let's say that you want to describe a function to duplicate an expression several times in parallel:

```
duplicate(1,x) = x;
duplicate(n,x) = x, duplicate(n-1,x);
```

Please note that this last definition is a convenient alternative to the more verbose :

```
duplicate = case {
               (1,x) => x;
               (n,x) => duplicate(n-1,x);
             };
```

Here is another example to count the number of elements of a list. Please note that we simulate lists using parallel composition : (1,2,3,5,7,11). The main limitation of this approach is that there is no empty list. Moreover lists of only one element are represented by this element :

```
count((x,xs)) = 1+count(xs);
count(x) = 1;
```

If we now write `count(duplicate(10,666))` the expression will be evaluated to `10`.

Please note that the order of pattern matching rules matters. The more specific rules must precede the more general rules. When this order is not respected, as in :

```
count(x) = 1;
count((x,xs)) = 1+count(xs);
```

the first rule will always match and the second rule will never be called.

Please note that number arguments in pattern matching rules are typically *constant numerical expressions*, so can be the result of more complex expressions involving computations done at compile-time.

3.3.4 Variants

Some statements (imports, definitions) can be preceded by a variantlist, composed of variants which can be *singleprecision*, *doubleprecision*, *quadprecision* or *fixedpointprecision* This allows some imports and definitions to be effective only for a (or several) specific float precision option in the compiler (that is either `-single`, `-double`, `-quad` or `-fx` respectively). A typical use-case is the definition of floating point constants in the maths.lib library with the following lines:

```
singleprecision MAX = 3.402823466e+38;
doubleprecision MAX = 1.7976931348623158e+308;
```

3.4 Expressions

Despite its textual syntax, FAUST is conceptually a block-diagram language. FAUST expressions represent DSP block-diagrams and are assembled from primitive ones using various *composition* operations. More traditional *numerical* expressions in infix notation are also possible. Additionally FAUST provides time based expressions, like delays, expressions related to lexical environments, expressions to interface with foreign function and lambda expressions.

3.4.1 Constant Numerical Expressions

Some language primitives (like *rdtable*, *rwtable*, *hslider* etc.) take constant numbers as some of their parameters. This is the case also for expressions using pattern matching techniques. Those numbers can be directly given in the code, but can also be computed by more complex expressions which have to produce numbers at compile time. We will refer to them as *constant numerical expressions* in the documentation.

3.4.2 Diagram Expressions

Diagram expressions are assembled from primitive ones using either binary composition operations or high level iterative constructions.

Diagram composition operations

Five binary *composition operations* are available to combine block-diagrams: *recursion*, *parallel*, *sequential*, *split* and *merge* composition. One can think of each of these composition operations as a particular way to connect two block diagrams.

To describe precisely how these connections are done, we have to introduce some notation. The number of inputs and outputs of a bloc-diagram A are notated $\mathrm{inputs}(A)$ and $\mathrm{outputs}(A)$. The inputs and outputs themselves are respectively notated : $[0]A$, $[1]A$, $[2]A$, $\ldots$ and $A[0]$, $A[1]$, $A[2]$, etc..

For each composition operation between two block-diagrams A and B we will describe the connections $A[i] \rightarrow [j]B$ that are created and the constraints on their relative numbers of inputs and outputs.

The priority and associativity of this five operations are given table 3.1. Please note that a higher priority value means a higher priority in the evaluation order. There is a companion table 3.3 that gives the associativity of each numerical operator in infix expressions.

Syntax	**Pri.**	**Assoc.**	**Description**
expression ~ *expression*	4	left	recursive composition
expression , *expression*	3	right	parallel composition
expression : *expression*	2	right	sequential composition
expression <: *expression*	1	right	split composition
expression :> *expression*	1	right	merge composition

Table 3.1: Block-Diagram composition operation priorities

Parallel Composition The *parallel composition* `(A,B)` (figure 3.2) is probably the simplest one. It places the two block-diagrams one on top of the other, without connections. The inputs of the resulting block-diagram are the inputs of `A` and `B`. The outputs of the resulting block-diagram are the outputs of `A` and `B`.

Parallel composition is an associative operation : `(A,(B,C))` and `((A,B),C)` are equivalents. When no parenthesis are used : `A,B,C,D`, FAUST uses right associativity and therefore build internally the expression `(A,(B,(C,D)))`. This organization is important to know when using pattern matching techniques on parallel compositions.

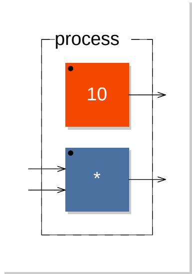


Figure 3.2: Example of parallel composition `(10,*)`

Sequential Composition The *sequential composition* `A:B` (figure 3.3) expects:

$$\text{outputs}(A) = \text{inputs}(B) \tag{3.1}$$

It connects each output of A to the corresponding input of B:

$$A[i] \rightarrow [i]B \tag{3.2}$$

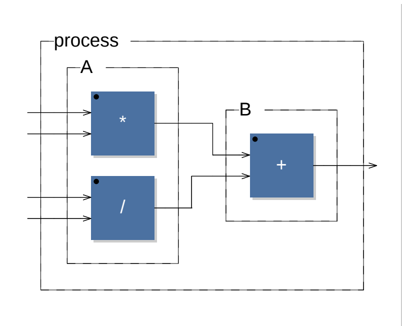


Figure 3.3: Example of sequential composition `((*,/):+)`

Sequential composition is an associative operation : `(A:(B:C))` and `((A:B):C)` are equivalents. When no parenthesis are used, like in `A:B:C:D`, Faust uses right associativity and therefore build internally the expression `(A:(B:(C:D)))`.

Split Composition The *split composition* `A<:B` (figure 3.4) operator is used to distribute the outputs of A to the inputs of B.

For the operation to be valid the number of inputs of B must be a multiple of the number of outputs of A :

$$\text{outputs}(A).k = \text{inputs}(B) \tag{3.3}$$

Each input i of B is connected to the output $i \bmod k$ of A :

$$A[i \bmod k] \rightarrow [i]B \tag{3.4}$$

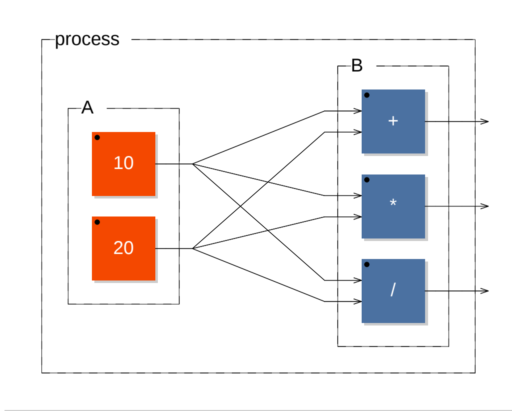


Figure 3.4: example of split composition `((10,20)<: (+,*,/))`

Merge Composition The *merge composition* `A:>B` (figure 3.5) is the dual of the *split composition*. The number of outputs of A must be a multiple of the number of inputs of B :

$$\text{outputs}(A) = k.\text{inputs}(B) \tag{3.5}$$

Each output i of A is connected to the input $i \bmod k$ of B :

$$A[i] \rightarrow \ [i \bmod k]B \tag{3.6}$$

The k incoming signals of an input of B are summed together.

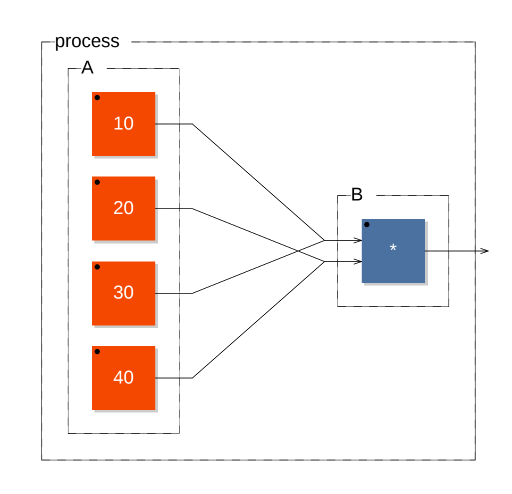


Figure 3.5: example of merge composition `((10,20,30,40):> *)`

Recursive Composition The *recursive composition* `A~B` (figure 3.6) is used to create cycles in the block-diagram in order to express recursive computations. It is the most complex operation in terms of connections.

To be applicable it requires that :

$$\text{outputs}(A) \geq \text{inputs}(B) and \text{inputs}(A) \geq \text{outputs}(B) \tag{3.7}$$

Each input of B is connected to the corresponding output of A via an implicit 1-sample delay :

$$A[i] \stackrel{Z^{-1}}{\rightarrow} [i]B \tag{3.8}$$

and each output of B is connected to the corresponding input of A:

$$B[i] \rightarrow [i]A \tag{3.9}$$

The inputs of the resulting block diagram are the remaining unconnected inputs of A. The outputs are all the outputs of A.

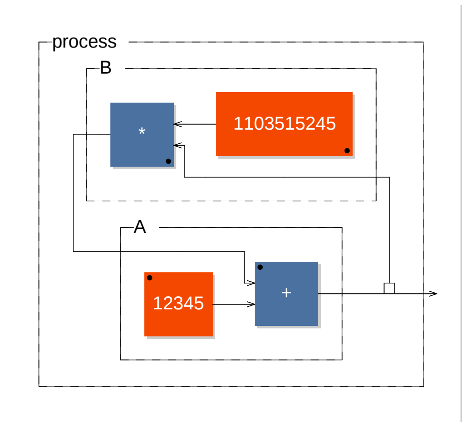


Figure 3.6: example of recursive composition `+(12345)~ *(1103515245)`

Inputs and outputs of an expression

These two constructions can be used to know at compile time the number of inputs and outputs of any Faust expression.

They are useful to define high order functions and build algorithmically complex block-diagrams. Here is an example to automatically reverse the order of the outputs of an expression.

```
Xo(expr) = expr <: par(i,n,selector(n-i-1,n))
          with { n=outputs(expr); };
```

And the inputs of an expression :

```
Xi(expr) = bus(n) <: par(i,n,selector(n-i-1,n)) : expr
          with { n=inputs(expr); };
```

For example `Xi(-)` will reverse the order of the two inputs of the substraction.

Iterations

Iterations are analogous to `for(...)` loops and provide a convenient way to automate some complex block-diagram constructions.

The following example shows the usage of `seq` to create a 10-bands filter:

```
process  =  seq(i, 10,
                vgroup("band %i",
                    bandfilter( 1000*(1+i) )
                )
            );
```

The number of iterations must be a constant expression.

3.4.3 Infix Notation

Besides its algebra-based core syntax, Faust provides some syntax extensions, in particular the familiar *infix notation*. For example if you want to multiply two numbers, let say `2` and `3`, you can write directly `2*3` instead of the equivalent core-syntax expression `2,3:*`.

Infix notation is commonly used in mathematics. It consists in placing the operand between the arguments as in $2 + 3$

`2-3`	≡	`2,3 : -`
`2*3`	≡	`2,3 : *`
`_@7`	≡	`_,7 : @`
`_/2`	≡	`_,2 : /`
`A<B`	≡	`A,B : <`

Table 3.2: Infix and core syntax equivalences

The *infix notation* is not limited to numbers or numerical expressions. Arbitrary expressions `A` and `B` can be used, provided that `A,B` has exactly two outputs. For example you can write `_/2`. It is equivalent to `_,2:/` which divides the incoming signal by `2`.

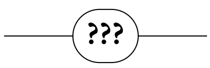
Examples of equivalences are given table 3.2.

In case of doubts on the meaning of an infix expression, for example `_*_`, it is useful to translate it to its core syntax equivalent, here `_,_:*`, which is equivalent to `*`.

Built-in primitives that can be used in infix notation are called *infix operators* and are listed in this section. Please note that a more detailed description of these operators is available section 3.5.

Comparison Operators

Comparison operators compare two signals and produce a signal that is 1 when the comparison is true and 0 when the comparison is false.



The priority and associativity of the comparison operators is given table 3.3.

Syntax	**Pri.**	**Assoc.**	**Description**
expression `<` *expression*	5	left	less than
expression `<=` *expression*	5	left	less or equal
expression `!=` *expression*	5	left	different
expression `>=` *expression*	5	left	greater or equal
expression `>` *expression*	5	left	greater than

Table 3.3: Comparison operators priorities in infix expressions

Math Operators

Math operators combine two signals and produce a resulting signal by applying a numerical operation on each sample.

The priority and associativity of the math operators is given table 3.4.

Syntax	**Pri.**	**Assoc.**	**Description**
expression `+` *expression*	6	left	addition
expression `-` *expression*	6	left	subtraction
expression `*` *expression*	7	left	multiplication
expression `/` *expression*	7	left	division
expression `%` *expression*	7	left	modulo
expression `^` *expression*	8	left	power

Table 3.4: Math operators priorities in infix expressions

Bitwise operators

Bitwise operators combine two signals and produce a resulting signal by applying a bitwise operation on each sample.

The priority and associativity of the bitwise operators is given table 3.5.

Syntax	**Pri.**	**Assoc.**	**Description**
expression `|` *expression*	6	left	bitwise or
expression `&` *expression*	7	left	bitwise and
expression `xor` *expression*	7	left	bitwise xor
expression `<<` *expression*	7	left	bitwise left shift
expression `>>` *expression*	7	left	bitwise right shift

Table 3.5: Bitwise operators priorities in infix expressions

Delay operators

Delay operators combine two signals and produce a resulting signal by applying a bitwise operation on each sample.

The delay operator `@` allows to delay left handside expression by the amount defined by the right handside expression. The unary operator `'` delays the left handside expression by one sample.

The priority and associativity of the delay operators is given table 3.6.

Syntax	Pri.	Assoc.	Description
expression `@` *expression*	9	left	variable delay
expression'	10	left	one-sample delay

Table 3.6: Delay operators priorities in infix expressions

3.4.4 Prefix Notation

Beside *infix notation*, it is also possible to use *prefix notation*. The *prefix notation* is the usual mathematical notation for functions $f(x, y, z, \ldots)$, but extended to *infix operators*.

`*(2,3)`	≡	`2,3 : *`
`@(_,7)`	≡	`_,7 : @`
`/(_,2)`	≡	`_,2 : /`
`<(A,B)`	≡	`A,B : <`

Table 3.7: Prefix to core syntax translation rules

It consists in first having the operator, for example `/`, followed by its arguments between parentheses: `/(2,3)` (see table 3.7).

Partial Application

The *partial application* notation is a variant of *prefix notation* in which not all arguments are given. For instance `/(2)` (divide by 2), `^(3)` (rise to the cube) and `@(512)` (delay by 512 samples) are examples of partial applications where only one argument is given. The result of a partial application is a function that "waits" for the remaining arguments.

When doing partial application with an *infix operator*, it is important to note that the supplied argument is not the first argument, but always the second one, as summarized table 3.8.

`+(C)`	≡	`_,C:*`
`-(C)`	≡	`_,C:-`
`<(C)`	≡	`_,C:<`
`/(C)`	≡	`_,C:/`

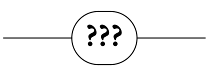
Table 3.8: Partial application of infix operators

For commutative operations that doesn't matter. But for non-commutative ones, it is more "natural" to fix the second argument. We use divide by 2 (`/(2)`) or rise to the cube (`^(3)`) more often than the other way around.

Please note that this rule only applies to infix operators, not other primitives or functions. If you partially apply a regular function to a single argument, it will correspond to the first parameter.

3.4.5 Time expressions

Time expressions are used to express delays. The notation `X@10` represent the signal `X` delayed by 10 samples. The notation `X'` represent the signal X delayed by one sample and is therefore equivalent to `X@1`.



The delay (automatically promoted to *int*) don't have to be fixed, but it must be positive and bounded. The values of a slider are perfectly acceptable as in the following example:

```
process = _ @ hslider("delay",0, 0, 100, 1);
```

3.4.6 Environment expressions

FAUST is a lexically scoped language. The meaning of a FAUST expression is determined by its context of definition (its lexical environment) and not by its context of use.

To keep their original meaning, FAUST expressions are bounded to their lexical environment in structures called *closures*. The following constructions allow to explicitly create and access such environments. Moreover they provide powerful means to reuse existing code and promote modular design.

With

The `with` construction allows to specify a *local environment*, a private list of definition that will be used to evaluate the left hand expression

In the following example :

```
pink = f : + ~ g with {
    f(x) = 0.04957526213389*x
         - 0.06305581334498*x'
         + 0.01483220320740*x'';
    g(x) = 1.80116083982126*x
         - 0.80257737639225*x';
};
```

the definitions of `f(x)` and `g(x)` are local to `f : + ~ g`.

Please note that `with` is left associative and has the lowest priority:

- `f : + ~ g with {...}` is equivalent to `(f : + ~ g) with {...}`.
- `f : + ~ g with {...} with {...}` is equivalent to `((f : + ~ g) with {...}) with {...}`.

Letrec

The `letrec` construction is somehow similar to `with`, but for *difference equations* instead of regular definitions. It allows to easily express groups of mutually recursive signals, for example:

$$\begin{aligned} x(t) &= y(t-1)+10; \\ y(t) &= x(t-1)-1; \end{aligned}$$

as `E letrec { 'x = y+10; 'y = x-1; }`

The syntax is defined by the following rules:

Please remarks the special notation `'x=y+10` instead of `x=y'+10`. It makes syntactically impossible to write non-sensical equations like `x=x+1`.

Here is a more involved example. Let say we want to define an envelop generator with an attack time, a release time and a gate signal. A possible definition is the following:

```
ar(a,r,g) = v
  letrec {
    'n = (n+1) * (g<=g');
    'v = max(0, v + (n<a)/a - (n>=a)/r) * (g<=g');
  };
```

With the following semantics for $n(t)$ and $v(t)$:

$$\begin{aligned} n(t) &= (n(t-1)+1)*(g(t) <= g(t-1)) \\ v(t) &= max(0, v(t-1)+(n(t-1) < a(t))/a(t) \\ &- (n(t-1) >= a(t))/r(t))*(g(t) <= g(t-1)) \end{aligned}$$

In order to factor some expressions common to several recursive definitions, we can use the clause `where` followed by one or more definitions. These definitions will only be visible to the recursive equations of the `letrec`, but not to the outside world, unlike the recursive definitions themselves.

For instance in the previous example we can factorize `(g<=g)` leading to the following expression:

```
    ar(a,r,g) = v
      letrec {
        'n = (n+1) * c;
        'v = max(0, v + (n<a)/a - (n>=a)/r) * c;
        where
          c = g<=g';
      };
```

Please note that `letrec` is essentially syntactic sugar.

Here is an example of 'letrec':

```
    x,y  letrec {
          x = defx;
          y = defy;
          z = defz;
        where
          f = deff;
          g = defg;
      };
```

and its translation as done internally by the compiler:

```
    x,y  with {
          x = BODY : _,!,!;
          y = BODY : !,_,!;
          z = BODY : !,!,_;
          BODY = \(x,y,z).((defx,defy,defz) with {f=deff;
             g=defg;}) ~ (_,_,_);
      };
```

Environment

The `environment` construction allows to create an explicit environment. It is like a `with`, but without the left hand expression. It is a convenient way to group together related definitions, to isolate groups of definitions and to create a name space hierarchy.

???

In the following example an `environment` construction is used to group together some constant definitions :

```
constant = environment {
    pi = 3.14159;
    e = 2.718;
    ...
};
```

The `.` construction allows to access the definitions of an environment (see next paragraph).

Access

Definitions inside an environment can be accessed using the '.' construction.

For example `constant.pi` refers to the definition of `pi` in the above `constant` environment.

Please note that environment don't have to be named. We could have written directly `environment{pi = 3.14159; e = 2.718; ... }.pi`

Library

The `library` construct allows to create an environment by reading the definitions from a file.

For example `library("miscfilter.lib")` represents the environment obtained by reading the file "miscfilter.lib". It works like `import("miscfilter.lib")` but all the read definitions are stored in a new separate lexical environment. Individual definitions can be accessed as described in the previous paragraph. For example `library("miscfilter.lib").lowpass` denotes the function `lowpass` as defined in the file `"miscfilter.lib"`.

To avoid name conflicts when importing libraries it is recommended to prefer `library` to `import`. So instead of :

```
import("miscfilter.lib");
  ...
...lowpass....
    ...
};
```

the following will ensure an absence of conflicts :

```
fl = library("miscfilter.lib");
  ...
...fl.lowpass....
    ...
};
```

Component

The `component(...)` construction allows to reuse a full FAUST program as a simple expression.

For example `component("freeverb.dsp")` denotes the signal processor defined in file "freeverb.dsp".

Components can be used within expressions like in:

```
 ...component("karplus32.dsp"):component("freeverb.dsp")
    ...
```

Please note that `component("freeverb.dsp")` is equivalent to `library("freeverb.dsp").process`.

Explicit substitution

Explicit substitution can be used to customize a component or any expression with a lexical environment by replacing some of its internal definitions, without having to modify it.

For example we can create a customized version of `component("freeverb.dsp")`, with a different definition of `foo(x)`, by writing :

```
...component("freeverb.dsp")[foo(x) = ...;]...
};
```

3.4.7 Foreign expressions

Reference to external (foreign) C *functions*, *variables* and *constants* can be introduced using the *foreign expressions* mechanism.

Foreign function declaration

An external C function is declared by indicating its name and signature as well as the required include file. The file `"maths.lib"` of the Faust distribution contains several foreign function definitions, for example the inverse hyperbolic sine function `asinh` is defined as follows

```
    asinh = ffunction(float asinhf|asinh|asinhl|asinfx(
       float), <math.h>, "");
```

The signature part of a foreign function, `float asinhf|asinh|asinhl|asinfx(float )` in our previous example, describes the prototype of the C function: its return type, function names and list of parameter types. Because the name of the foreign function can possibly depend on the floating point precision in use (float, double, quad and fixed-point), it is possible to give a different function name for each floating point precision using a signature with up to four function names.

In our example, the `asinh` function is called `asinhf` in single precision, `asinh` in double precision, `asinhl` in quad precision, and `asinhfx` in fixed-point precision. This is why the four names are provided in the signature.

Foreign functions generally expect a precise type: (*int* or *float*) for their parameters. Note that currently only numerical functions involving scalar parameters are allowed. No vectors, tables or data structures can be passed as parameters or returned. During the compilation if the type of an argument is not the same as the type of the parameter, it is automatically converted to the expected one.

Some foreign functions are polymorphic and can accept either *int* or *float* arguments. In this case, the polymorphism can be indicated by using the type *any* instead or int or float. Here is as an example the C function *sizeof* that returns the size of its argument:

```
    sizeof = ffunction(int sizeof(any), "","");
```

Foreign functions with input parameters are considered pure math functions. They are therefore considered free of side effects and called only when their parameters change (that is at the rate of the fastest parameter).

Exceptions are functions with no input parameters. A typical example is the C `rand()` function. In this case the compiler generates code to call the function at sample rate.

Foreign variables and constants

External variables and constants can also be declared with a similar syntax. In the same `"maths.lib"` file, the definition of the sampling rate constant `SR` and the definition of the block-size variable `BS` can be found :

```
SR     = min(192000.0,
           max(1.0,
             fconstant(int fSamplingFreq, <math.h>)));
BS     = fvariable(int count, <math.h>);
```

Foreign constants are not supposed to vary. Therefore expressions involving only foreign constants are computed once, during the initialization period.

Foreign variables are considered to vary at block speed. This means that expressions depending of external variables are computed every block.

Include file

In declaring foreign functions one has also to specify the include file. It allows the FAUST compiler to add the corresponding `#include...` in the generated code.

Library file

In declaring foreign functions one can possibly specify the library where the actual code is located. It allows the FAUST compiler to (possibly) automatically link the library. Note that this feature is only used with the LLVM backend in 'libfaust' dynamic library model.

3.4.8 Applications and Abstractions

Abstractions and *applications* are fundamental programming constructions directly inspired by the Lambda-Calculus. These constructions provide powerful ways to describe and transform block-diagrams algorithmically.

Abstractions

Abstractions correspond to functions definitions and allow to generalize a block-diagram by *making variable* some of its parts.

Let's say you want to transform a stereo reverb, `freeverb` for instance, into a mono effect. You can write the following expression:

```
_ <: freeverb :> _
```

The incoming mono signal is splitted to feed the two input channels of the reverb, while the two output channels of the reverb are mixed together to produce the resulting mono output.

Imagine now that you are interested in transforming other stereo effects. It can be interesting to generalize this principle by making `freeverb` a variable:

```
\(freeverb).(_ <: freeverb :> _)
```

The resulting abstraction can then be applied to transform other effects. Note that if `freeverb` is a perfectly valid variable name, a more neutral name would probably be easier to read like:

```
\(fx).(_ <: fx :> _)
```

Moreover it could be convenient to give a name to this abstraction:

```
mono = \(fx).(_ <: fx :> _);
```

Or even use a more traditional, but equivalent, notation:

```
mono(fx) = _ <: fx :> _;
```

Applications

Applications correspond to function calls and allow to replace the variable parts of an abstraction with the specified arguments.

For example you can apply the previous abstraction to transform your stereo harmonizer:

```
mono(harmonizer)
```

The compiler will start by replacing `mono` by its definition:

```
\(fx).(_ <: fx :> _)(harmonizer)
```

Whenever the Faust compiler find an application of an abstraction it replaces the *variable part* with the argument. The resulting expression is as expected:

Replacing the *variable part* with the argument is called β-reduction in Lambda-Calculus

```
(_ <: harmonizer :> _)
```

Note that the arguments given to the primitive or function in applications are reduced to their *block normal form* (that is the flat equivalent block) before the actual application. Thus if the number of outputs of the argument block does not mach the needed number of arguments, the application will be treated as *partial application* and the missing arguments will be replaced by one or several `_` (to complete the number of missing arguments).

Unapplied abstractions

Usually, lambda abstractions are supposed to be applied on arguments, using beta-reduction in Lambda-Calculus. Functional languages generally treat them as first-class values[1] which give these languages high-order programming capabilities.

Another way of looking at abstractions in Faust is as a means of routing or placing blocks that are given as parameters. For example, the following abstraction `repeat(fx)= fx : fx;` could be used to duplicate an effect and route input signals to be successively processed by that effect:

```
import("stdfaust.lib");
repeat(fx) = fx : fx;
process = repeat(dm.zita_light);
```

In Faust, a proper semantic has been given to *unapplied abstractions*: when a lambda-abstraction is not applied to parameters, it indicates *how to route input signals*. This is a convenient way to work with signals by *explicitly naming them*, to be used in the lambda abstraction body *with their parameter name*.

For instance a stereo crossing block written in the core syntax:

```
process = _,_ <: !,_,_,!;
```

can be simply defined as:

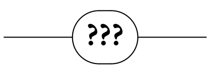
```
process = \(x,y).(y,x);
```

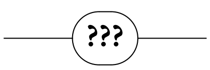
which is actually equivalent to:

```
process(x,y) = y,x;
```

Pattern Matching

Pattern matching rules provide an effective way to analyze and transform block-diagrams algorithmically.





[1] https://en.wikipedia.org/wiki/First-class_function

For example `case{ (x:y)=> y:x; (x)=> x; }` contains two rules. The first one will match a sequential expression and invert the two part. The second one will match all remaining expressions and leave it untouched. Therefore the application:

```
case{(x:y) => y:x; (x) => x;}(freeverb:harmonizer)
```

will produce:

```
    (harmonizer:freeverb)
```

Please note that patterns are evaluated before the pattern matching operation. Therefore only variables that appear free in the pattern are binding variables during pattern matching.

3.5 Primitives

The primitive signal processing operations represent the built-in functionalities of FAUST, that is the atomic operations on signals provided by the language. All these primitives denote *signal processors*, functions transforming *input signals* into *output signals*.

3.5.1 Numbers

FAUST considers two types of numbers: *integers* and *floats*. Integers are implemented as signed 32-bits integers, and floats are implemented either with a simple, double or extended precision depending of the compiler options. Floats are available in decimal or scientific notation.

Like any other FAUST expression, numbers are signal processors. For example the number 0.95 is a signal processor of type $\mathbb{S}^0 \rightarrow \mathbb{S}^1$ that transforms an empty tuple of signals $()$ into a 1-tuple of signals (y) such that $\forall t \in \mathbb{N}, y(t) = 0.95$.

Operations on *integer* numbers follow the standard C semantic for `+, -, *` operations and can possibly overflow if the result cannot be represented as a 32-bits integer. The `/` operation is treated separately and cast both of its arguments to floats before doing the division, and thus the result takes the float type.

3.5.2 Route Primitive

The `route` primitive facilitates the routing of signals in Faust. It has the following syntax:

```
route(A,B,a,b,c,d,...)
route(A,B,(a,b),(c,d),...)
```

where:

- `A` is the number of input signals, as an integer *constant numerical expression*, automatically promoted to *int*
- `B` is the number of output signals, as an integer *constant numerical expression*, automatically promoted to *int*
- `a,b / (a,b)` is an input/output pair, as integers *constant numerical expressions*, automatically promoted to *int*

Inputs are numbered from `1` to `A` and outputs are numbered from `1` to `B`. There can be any number of input/output pairs after the declaration of `A` and `B`.

For example, crossing two signals can be carried out with:

```
process = route(2,2,1,2,2,1);
```

In that case, `route` has 2 inputs and 2 outputs. The first input (1) is connected to the second output (2) and the second input (2) is connected to the first output (1).

Note that parenthesis can be optionally used to define a pair, so the previous example can also be written as:

```
process = route(2,2,(1,2),(2,1));
```

More complex expressions can be written using algorithmic constructions, like the following one to cross N signals:

```
// cross 10 signals:
// input 0 -> output 10,
// input 1 -> output 9,
// ...,
// input 9 -> output 0

N = 10;
r = route(N,N,par(i,N,(i+1,N-i)));

process = r;
```

3.5.3 Waveform Primitive

A waveform is a fixed periodic signal defined by a list of samples as literal numbers. A waveform has two outputs. The first output is constant and indicates the size (number of samples) of the period. The second output is the periodic signal itself.

For example `waveform {0,1,2,3}` produces two outputs, the constant signal 4 and the periodic signal `0,1,2,3,0,1...`.

Please note that `waveform` works nicely with `rdtable`. Its first output, known at compile time, gives the size of the table, while the second signal gives the content of the table. Here is an example:

```
process = waveform {10,20,30,40,50,60,70}, %(7)~+(3) :
   rdtable;
```

3.5.4 Soundfile Primitive

The `soundfile("label[url:\{`path1';'path2';'path3';}]", n)' primitive allows access to a list of externally defined sound resources, described as a label followed by the list of their filename, or complete paths (possibly using the '%i' syntax, as in the label part). The `soundfile("label[url:path]", n)`, or `soundfile("label", n)` (where label is used as the soundfile path) simplified syntax allows to use a single file. All sound resources are concatenated in a single data structure, and each item can be accessed and used independently.

A soundfile has:

- two inputs: the sound number (as a integer between 0 and 255, automatically promoted to *int*), and the read index in the sound (automatically promoted to *int*, which will access the last sample of the sound if the read index is greater than the sound length)
- two fixed outputs: the first one is the length in samples of the currently accessed sound, the second one is the nominal sample rate in Hz of the currently accessed sound
- `n` several more outputs for the sound channels themselves, as a integer *constant numerical expression*

If more outputs than the actual number of channels in the soundfile are used, the sound channels will be automatically duplicated up to the wanted number of outputs (so for instance if a stereo sound is used with four output channels, the same group of two channels will be duplicated).

If the soundfile cannot be loaded for whatever reason, a default sound with one channel, a length of 1024 frames and null outputs (with samples of value 0) will be used. Note also that soundfiles are entirely loaded in memory by the architecture file, so that the read index signal can access any sample.

Specialized architecture files are responsible to load the actual soundfile. The `SoundUI` C++ class located in the `faust/gui/SoundUI.h` file implements the `void addSoundfile (label, url, sf_zone)` method, which loads the actual soundfiles using the *libsndfile* library, or possibly specific audio file loading code (in the case of the JUCE framework for instance), and set up the `sf_zone` sound memory pointers.

Note that a special architecture file can perfectly decide to access and use sound resources created by another means (that is, not directly loaded from a soundfile). For instance a mapping between labels and sound resources defined in memory could be used, with some additional code in charge to actually setup all sound memory pointers when `void addSoundfile(label, url, sf_zone)` is called by the `buidUserInterface` mechanism.

3.5.5 C-equivalent primitives

Most FAUST primitives are analogue to their C counterpart but adapted to signal processing. For example `+` is a function of type $\mathbb{S}^2 \rightarrow \mathbb{S}^1$ that transforms a pair of signals

(x_1, x_2) into a 1-tuple of signals (y) such that $\forall t \in \mathbb{N}, y(t) = x_1(t) + x_2(t)$. The function `-` has type $\mathbb{S}^2 \rightarrow \mathbb{S}^1$ and transforms a pair of signals (x_1, x_2) into a 1-tuple of signals (y) such that $\forall t \in \mathbb{N}, y(t) = x_1(t) - x_2(t)$.

Please be aware that the unary `-` only exists in a limited form. It can be used with numbers: `-0.5` and variables: `-myvar`, but not with expressions surrounded by parenthesis, because in this case it represents a partial application. For instance `-(a * b)` is a partial application. It is syntactic sugar for `_,(a * b): -`. If you want to negate a complex term in parenthesis, you'll have to use `0 - (a * b)` instead.

Warning: unlike other programming languages the unary operatior `-` only exists in limited form in Faust

The primitives may use the `int` type for their arguments, but will automatically use the `float` type when the actual computation requires it. For instance `1/2` using `int` type arguments will correctly result in `0.5` in `float` type. Logical and shift primitives use the `int` type.

Syntax	Type	Description
n	$\mathbb{S}^0 \rightarrow \mathbb{S}^1$	integer number: $y(t) = n$
$n.m$	$\mathbb{S}^0 \rightarrow \mathbb{S}^1$	floating point number: $y(t) = n.m$
`_`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	identity function: $y(t) = x(t)$
`!`	$\mathbb{S}^1 \rightarrow \mathbb{S}^0$	cut function: $\forall x \in \mathbb{S}, (x) \rightarrow ()$
`int`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	cast into an int signal: $y(t) = (int)x(t)$
`float`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	cast into an float signal: $y(t) = (float)x(t)$
`+`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	addition: $y(t) = x_1(t) + x_2(t)$
`-`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	subtraction: $y(t) = x_1(t) - x_2(t)$
`*`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	multiplication: $y(t) = x_1(t) * x_2(t)$
$\wedge$	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	power: $y(t) = x_1(t)^{x_2(t)}$
`/`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	division: $y(t) = x_1(t)/x_2(t)$
`%`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	modulo: $y(t) = x_1(t)\%x_2(t)$
`&`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	bitwise AND: $y(t) = x_1(t)\&x_2(t)$
`|`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	bitwise OR: $y(t) = x_1(t)\|x_2(t)$
`xor`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	bitwise XOR: $y(t) = x_1(t) \wedge x_2(t)$
`<<`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	arith. shift left: $y(t) = x_1(t) << x_2(t)$
`>>`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	arith. shift right: $y(t) = x_1(t) >> x_2(t)$
`<`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	less than: $y(t) = x_1(t) < x_2(t)$
`<=`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	less or equal: $y(t) = x_1(t) <= x_2(t)$
`>`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	greater than: $y(t) = x_1(t) > x_2(t)$
`>=`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	greater or equal: $y(t) = x_1(t) >= x_2(t)$
`==`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	equal: $y(t) = x_1(t) == x_2(t)$
`!=`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	different: $y(t) = x_1(t) != x_2(t)$

3.5.6 `math.h`-equivalent primitives

Most of the C `math.h` functions are also built-in as primitives (the others are defined as external functions in file `maths.lib`). The primitives may use the `int` type for their arguments, but will automatically use the `float` type when the actual computation requires it.

Syntax	Type	Description
`acos`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	arc cosine: $y(t) = \mathrm{acosf}(x(t))$
`asin`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	arc sine: $y(t) = \mathrm{asinf}(x(t))$
`atan`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	arc tangent: $y(t) = \mathrm{atanf}(x(t))$
`atan2`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	arc tangent: $y(t) = \mathrm{atan2f}(x_1(t), x_2(t))$
`cos`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	cosine: $y(t) = \mathrm{cosf}(x(t))$
`sin`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	sine: $y(t) = \mathrm{sinf}(x(t))$
`tan`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	tangent: $y(t) = \mathrm{tanf}(x(t))$
`exp`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	base-e exponential: $y(t) = \mathrm{expf}(x(t))$
`log`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	base-e logarithm: $y(t) = \mathrm{logf}(x(t))$
`log10`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	base-10 logarithm: $y(t) = \mathrm{log10f}(x(t))$
`pow`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	power: $y(t) = \mathrm{powf}(x_1(t), x_2(t))$
`sqrt`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	square root: $y(t) = \mathrm{sqrtf}(x(t))$
`abs`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	absolute value (int): $y(t) = \mathrm{abs}(x(t))$
		absolute value (float): $y(t) = \mathrm{fabsf}(x(t))$
`min`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	minimum: $y(t) = \min(x_1(t), x_2(t))$
`max`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	maximum: $y(t) = \max(x_1(t), x_2(t))$
`fmod`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	float modulo: $y(t) = \mathrm{fmodf}(x_1(t), x_2(t))$
`remainder`	$\mathbb{S}^2 \rightarrow \mathbb{S}^1$	float remainder: $y(t) = \mathrm{remainderf}(x_1(t), x_2(t))$
`floor`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	largest int $\leq$: $y(t) = \mathrm{floorf}(x(t))$
`ceil`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	smallest int $\geq$: $y(t) = \mathrm{ceilf}(x(t))$
`rint`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	closest int using the current rounding mode: $y(t) = \mathrm{rintf}(x(t))$
`round`	$\mathbb{S}^1 \rightarrow \mathbb{S}^1$	nearest int value, regardless of the current rounding mode: $y(t) = \mathrm{rintf}(x(t$

3.5.7 Delay, Table, Selector primitives

The following primitives allow to define fixed delays, read-only and read-write tables and 2 or 3-ways selectors (see figure 3.7).

Syntax	**Type**	**Description**
`mem`	$\mathbb{S}^1 \to \mathbb{S}^1$	1-sample delay: $y(t+1) = x(t), y(0) = 0$
`prefix`	$\mathbb{S}^2 \to \mathbb{S}^1$	1-sample delay: $y(t+1) = x_2(t), y(0) = x_1(0)$
`@`	$\mathbb{S}^2 \to \mathbb{S}^1$	fixed delay: $y(t + x_2(t)) = x_1(t), y(t < x_2(t)) = 0$
`rdtable`	$\mathbb{S}^3 \to \mathbb{S}^1$	read-only table: $y(t) = T[r(t)]$
`rwtable`	$\mathbb{S}^5 \to \mathbb{S}^1$	read-write table: $T[w(t)] = c(t); y(t) = T[r(t)]$
`select2`	$\mathbb{S}^3 \to \mathbb{S}^1$	select between 2 signals: $T[] = \{x_0(t), x_1(t)\}; y(t) = T[s(t)]$
`select3`	$\mathbb{S}^4 \to \mathbb{S}^1$	select between 3 signals: $T[] = \{x_0(t), x_1(t), x_2(t)\}; y(t) = T[s(t)]$

The size input of *rdtable* and *rwtable* are integer *constant numerical expressions* automatically promoted to *int*, and the read and write indexes are also automatically promoted to *int*. The delay value is automatically promoted to *int*.

3.5.8 User Interface Elements

Faust user interface widgets allow an abstract description of the user interface from within the Faust code. This description is independent of any GUI toolkits. It is based on *buttons*, *checkboxes*, *sliders*, etc. that are grouped together vertically and horizontally using appropriate grouping schemes.

All these GUI elements produce signals. A button for example (see figure 3.8) produces a signal which is 1 when the button is pressed and 0 otherwise. These signals can be freely combined with other audio signals.

Syntax	**Example**
`button(`*str*`)`	`button("play")`
`checkbox(`*str*`)`	`checkbox("mute")`
`vslider(`*str*`,`*cur*`,`*min*`,`*max*`,`*step*`)`	`vslider("vol",50,0,100,1)`
`hslider(`*str*`,`*cur*`,`*min*`,`*max*`,`*step*`)`	`hslider("vol",0.5,0,1,0.01)`
`nentry(`*str*`,`*cur*`,`*min*`,`*max*`,`*step*`)`	`nentry("freq",440,0,8000,1)`
`vgroup(`*str*`,`*block-diagram*`)`	`vgroup("reverb", ...)`
`hgroup(`*str*`,`*block-diagram*`)`	`hgroup("mixer", ...)`
`tgroup(`*str*`,`*block-diagram*`)`	`tgroup("parametric", ...)`
`vbargraph(`*str*`,`*min*`,`*max*`)`	`vbargraph("input",0,100)`
`hbargraph(`*str*`,`*min*`,`*max*`)`	`hbargraph("signal",0,1.0)`
`attach`	`attach(x, vumeter(x))`

All numerical parameters (like *cur*, *min*, *max*, *step*) are *constant numerical expressions*.

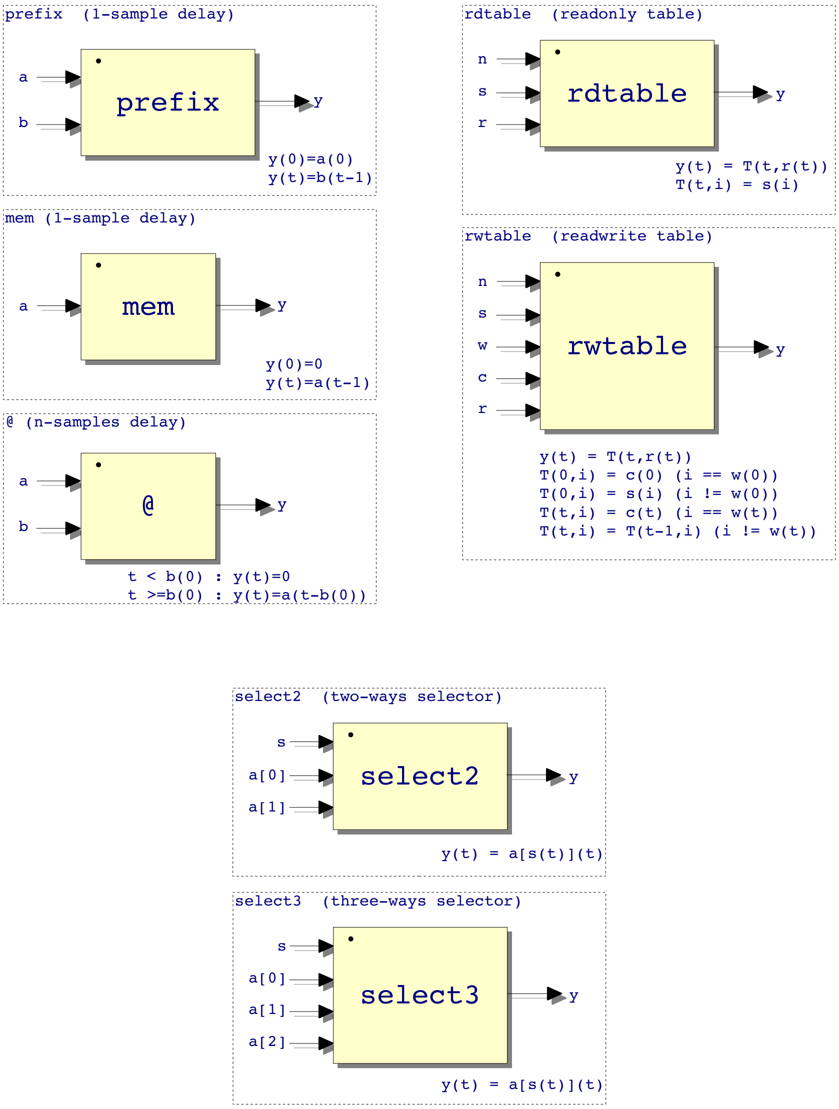


Figure 3.7: Delays, tables and selectors primitives

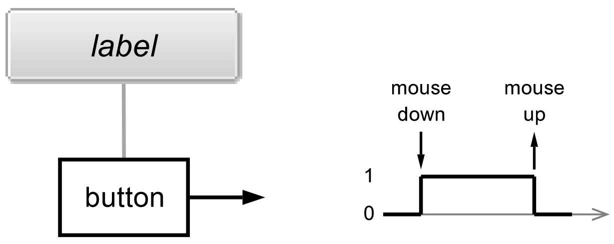


Figure 3.8: User Interface Button

Labels

Every user interface widget has a label (a string) that identifies it and informs the user of its purpose. There are three important mechanisms associated with labels (and coded inside the string): *variable parts*, *pathnames* and *metadata*.

Variable parts. Labels can contain variable parts. These variable parts are indicated by the sign '%' followed by the name of a variable. During compilation each label is processed in order to replace the variable parts by the value of the variable. For example `par(i,8,hslider("Voice %i", 0.9, 0, 1, 0.01))` creates 8 different sliders in parallel :

```
hslider("Voice 0", 0.9, 0, 1, 0.01),
hslider("Voice 1", 0.9, 0, 1, 0.01),
...
hslider("Voice 7", 0.9, 0, 1, 0.01).
```

while `par(i,8,hslider("Voice", 0.9, 0, 1, 0.01))` would have created only one slider and duplicated its output 8 times.

The variable part can have an optional format digit. For example `"Voice %2i"` would indicate to use two digits when inserting the value of i in the string.

An escape mechanism is provided. If the sign `%` is followed by itself, it will be included in the resulting string. For example `"feedback (%%)"` will result in `"feedback (%)"`.

The variable name can be enclosed in curly brackets to clearly separate it from the rest of the string, as in `par(i,8,hslider("Voice %{i}", 0.9, 0, 1, 0.01))`.

Pathnames. Thanks to horizontal, vertical and tabs groups, user interfaces have a hierarchical structure analog to a hierarchical file system. Each widget has an associated *pathname* obtained by concatenating the labels of all its surrounding groups with its own label.

In the following example:

```
hgroup("Foo",
    ...
    vgroup("Faa",
        ...
        hslider("volume",...)
        ...
    )
    ...
)
```

the volume slider has pathname `/h:Foo/v:Faa/volume`.

In order to give more flexibility to the design of user interfaces, it is possible to explicitly specify the absolute or relative pathname of a widget directly in its label.

In our previous example the pathname of:

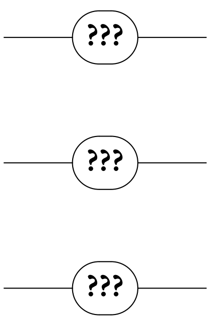
```
    hslider("../volume",...)
```

would have been `"/h:Foo/volume"`, while the pathname of:

```
    hslider("t:Fii/volume",...)
```

would have been: `"/h:Foo/v:Faa/t:Fii/volume"`.

The grammar for labels with pathnames is the following:



Metadata Widget labels can contain metadata enclosed in square brackets. These metadata associate a key with a value and are used to provide additional information to the architecture file. They are typically used to improve the look and feel of the user interface. The FAUST code:

```
process = *(hslider("foo [key1: val 1][key2: val 2]",
                    0, 0, 1, 0.1));
```

will produce and the corresponding C++ code:

```
class mydsp : public dsp {
    ...
    virtual void buildUserInterface(UI* interface) {
      interface->openVerticalBox("m");
      interface->declare(&fslider0, "key1", "val 1");
      interface->declare(&fslider0, "key2", "val 2");
      interface->addHorizontalSlider("foo",
        &fslider0, 0.0f, 0.0f, 1.0f, 0.1f);
      interface->closeBox();
    }
...
};
```

All the metadata are removed from the label by the compiler and transformed in calls to the `UI::declare()` method. All these `UI::declare()` calls will always take place before the `UI::AddSomething()` call that creates the User Interface element. This allows the `UI::AddSomething()` method to make full use of the available metadata.

It is the role of the architecture file to decide what to do with these metadata. The `jack-qt.cpp` architecture file for example implements the following:

1. `"...[style:knob]..."` creates a rotating knob instead of a regular slider or nentry.
2. `"...[style:led]..."` in a bargraph's label creates a small LED instead of a full bargraph
3. `"...[unit:dB]..."` in a bargraph's label creates a more realistic bargraph with colors ranging from green to red depending of the level of the value
4. `"...[unit:xx]..."` in a widget postfixes the value displayed with xx
5. `"...[tooltip:bla bla]..."` add a tooltip to the widget
6. `"...[osc:/address min max]..."` Open Sound Control message alias

Moreover starting a label with a number option like in `"[1]..."` provides a convenient means to control the alphabetical order of the widgets.

Attach

The `attach` primitive takes two input signals and produce one output signal which is a copy of the first input. The role of `attach` is to force its second input signal to be compiled with the first one. From a mathematical point of view `attach(x,y)` is equivalent to `1*x+0*y`, which is in turn equivalent to `x`, but it tells the compiler not to optimize-out `y`.

To illustrate this role let say that we want to develop a mixer application with a vumeter for each input signals. Such vumeters can be easily coded in FAUST using an envelop detector connected to a bargraph. The problem is that these envelop signals have no role in the output signals. Using `attach(x,vumeter(x))` one can tell the compiler that when `x` is compiled `vumeter(x)` should also be compiled.

3.5.9 Widget Modulation

Widget modulation acts on the widgets of an existing FAUST expression, but without requiring any manual modifications of the expression's code. This operation is done directly by the compiler, according to a list of *target widgets* and associated *modulators*. Target widgets are specified by their label, as used in the graphical user interface. Modulators are FAUST expressions that describe how to transform the signal produced by widgets. The syntax of a widget modulation is the following:

Here is a very simple example of widget modulation, assuming freeverb is a fully functional reverb with a `"Wet"` slider:

```
["Wet" -> freeverb]
```

The resulting circuit will have three inputs instead of two. The additional input is for the `"Wet"` widget. It acts on the values produced by the widget inside the freeverb expression. By default, the additional input signal, and the widget signal are multiplied together. In the following example, an external LFO is connected to this additional input:

```
lfo(10, 0.5), _, _ : ["Wet" -> freeverb]
```

Target Widgets

Target widgets are specified by their label. Of course, this presupposes knowing the names of the widgets. But as these names appear on the user interface, it's easy enough. If several widgets have the same name, adding the names of some (not necessarily all) of the surrounding groups, as in `"h:group/h:subgroup/label"` can help distinguish them.

Multiple targets can be indicated in the same widget modulation expression as in:

```
["Wet", "Damp", "RoomSize" -> freeverb]
```

Modulators

Modulators are Faust expressions, with exactly one output and at most two inputs that describe how to transform the signals produced by widgets. By default, when nothing is specified, the modulator is a multiplication. This is why our previous example is equivalent to:

```
["Wet": * -> freeverb]
```

Please note that the ':' sign used here is just a visual separator, it is not the sequential composition operator.

To indicate that the modulation signal should be added, instead of multiplied, one could write:

```
["Wet": + -> freeverb]
```

Multiplications and addition are examples of `2->1` modulators, but two other types are allowed: `0->1` and `1->1`.

Modulators with no inputs Modulators with no inputs `0->1` completely replace the target widget (it won't appear anymore in the user interface). Let's say that we want to remove the `"Damp"` slider and replace it with the constant `0.5`, we can write:

```
    ["Damp": 0.5 -> freeverb]
```

Modulators with one input A `1->1` modulator transforms the signal produced by the target widget without the help of an external input. Our previous example could be written as:

```
["Wet": *(lfo(10, 0.5)) -> freeverb]
```

If `lfo` had its user interface, it would be added to the freeverb interface, at the same level as the `"Wet"` slider.

Modulators with two inputs Modulators with two inputs, like `*` or `+`, are used to combine the signal produced by the target widget with an external signal. The first input is connected to the widget, the second one is connected to the external signal. As we have already seen, our example could be written as:

```
    lfo(10, 0.5), _, _ : ["Wet": * -> freeverb]
```

The main difference with the previous case is that if `lfo` had a user interface, it would be added outside of the freeverb interface. Please note that only `2->1` modulators result in additional inputs.

Chapter 4

Invoking the Faust compiler

The Faust compiler is invoked using the `faust` command. It translate Faust programs into C++ code. The generated code can be wrapped into an optional *architecture file* allowing to directly produce a fully operational program.

???

For example `faust noise.dsp` will compile `noise.dsp` and output the corresponding C++ code on the standard output. The option `-o` allows to choose the output file: `faust noise.dsp -o noise.cpp`. The option `-a` allows to choose the architecture file: `faust -a alsa-gtk.cpp noise.dsp`.

To compile a Faust program into an ALSA application on Linux you can use the following commands:

```
faust -a alsa-gtk.cpp noise.dsp -o noise.cpp
g++ -lpthread -lasound ‘pkg-config --cflags --libs
    gtk+-2.0‘ noise.cpp -o noise
```

4.1 Structure of the generated code

A Faust DSP C++ class derives from the base *dsp* class defined as below (a similar structure is used for languages other than C++):

```
class dsp {

   public:
```

```
        dsp() {}
        virtual ~dsp() {}

        // Returns the number of inputs of the Faust program
        virtual int getNumInputs() = 0;

        // Returns the number of outputs of the Faust program
        virtual int getNumOutputs() = 0

        // This method can be called to retrieve
        // the UI description of the Faust program
        // and its associated fields
        virtual void buildUserInterface(UI* ui_interface) =
            0;

        // Returns the current sampling rate
        virtual int getSampleRate() = 0;

        // Init methods
        virtual void init(int sample_rate) = 0;
        virtual void instanceInit(int sample_rate) = 0;
        virtual void instanceConstants(int sample_rate) = 0;
        virtual void instanceResetUserInterface() = 0;
        virtual void instanceClear() = 0;

        // Returns a clone of the instance
        virtual dsp* clone() = 0;

        // Retrieve the global metadata of the Faust program
        virtual void metadata(Meta* m) = 0;

        // Compute one audio buffer
        virtual void compute(int count, FAUSTFLOAT** inputs,
            FAUSTFLOAT** outputs) = 0;

        // Compute a time-stamped audio buffer
        virtual void compute(double date_usec, int count,
            FAUSTFLOAT** inputs, FAUSTFLOAT** outputs)
        {
             compute(count, inputs, outputs);
        }
};
```

Here is the class generated with the command `faust noise.dsp`. Methods are filled by the compiler with the actual code.

Several fine-grained initialization methods are available. The `instanceInit` method calls several additional initialization methods. The `instanceConstants` method sets the instance constant state. The `instanceClear` method resets the instance dynamic state (delay lines...). The `instanceResetUserInterface` method resets all control value to their default state. All of those methods can be used individually on an allocated instance to reset part of its state.

The `classInit` static method will initialize static tables that are shared between all instances of the class, and is typically supposed to be called once.

Finally the `init` method combines class static state and instance initialization.

When using a single instance, then calling `init` is the simplest way to do what is needed. When using several instances, then all of them can be initialized using `instanceInit`, whith a single call to `classInit` to initialize the static shared state.

The `compute` method takes the number of frames to process, and `inputs` and `outputs` buffers as arrays of separated mono channels. Note that by default `inputs` and `outputs` buffers are supposed to be distinct memory zones, so one cannot safely write `compute(count, inputs, inputs)`. The `-inpl` compilation option can be used for that, but only in scalar mode for now.

By default the generated code process `float` type samples. This can be changed using the `-double` option (or even `-quad` in some backends). The `FAUSTFLOAT` type used in the `compute` method is defined in architecture files, and can be `float` or `double`, depending of the audio driver layer. Sample adaptation may have to be used between the DSP sample type and the audio driver sample type.

```
class mydsp : public dsp {

  private:
      FAUSTFLOAT fslider0;
      int iRec0[2];
      int fSampleRate;

  public:
     virtual void metadata(Meta* m) {
        m->declare("name", "Noise");
        m->declare("version", "1.1");
        m->declare("author", "Grame");
        m->declare("license", "BSD");
        m->declare("copyright", "(c)GRAME 2009");
    }

    virtual int getNumInputs() { return 0; }
    virtual int getNumOutputs() { return 1; }
    static void classInit(int sample_rate) {
    }
    virtual void instanceConstants(int sample_rate) {
        fSampleRate = sample_rate;
    }
```

```
    virtual void instanceResetUserInterface() {
        fslider0 = 0.5f;
    }
    virtual void instanceClear() {
        for (int i=0; i<2; i++) iRec0[i] = 0;
    }
    virtual void init(int sample_rate) {
        classInit(sample_rate);
        instanceInit(sample_rate);
    }
    virtual void instanceInit(int sample_rate) {
        instanceConstants(sample_rate);
        instanceResetUserInterface();
        instanceClear();
    }
    virtual mydsp* clone() {
        return new mydsp();
    }
    virtual int getSampleRate() {
        return fSampleRate;
    }
    virtual void buildUserInterface(UI* ui_interface) {
        ui_interface->openVerticalBox("Noise");
        ui_interface->declare(&fslider0, "style", "knob");
        ui_interface->addVerticalSlider("Volume", &fslider0,
           0.5f, 0.0f, 1.0f, 0.1f);
        ui_interface->closeBox();
    }
    virtual void compute (int count, FAUSTFLOAT** input,
       FAUSTFLOAT** output) {
        float fSlow0 = (4.656613e-10f * float(fslider0));
        FAUSTFLOAT* output0 = output[0];
        for (int i=0; i<count; i++) {
            iRec0[0] = ((1103515245 * iRec0[1]) + 12345);
            output0[i] = (FAUSTFLOAT)(fSlow0 * iRec0[0]);
            // post processing
            iRec0[1] = iRec0[0];
        }
    }
};
```

4.2 Compilation options

Compilation options are listed in the following table :

Short	Long	Description
-h	-help	print the help message
-v	-version	print version information
-d	-details	print compilation details
-tg	-task-graph	draw a graph of all internal computation loops as a .dot (graphviz) file.
-sg	-signal-graph	draw a graph of all internal signal expressions as a .dot (graphviz) file.
-ps	-postscript	generate block-diagram postscript files
-svg	-svg	generate block-diagram svg files
-blur	-shadow-blur	add a blur to boxes shadows
-sd	-simplify-diagrams	simplify block-diagram before drawing them
-f *n*	-fold *n*	max complexity of svg diagrams before splitting into several files (default 25 boxes)
-mns *n*	-max-name-size *n*	max character size used in svg diagram labels
-sn	-simple-names	use simple names (without arguments) for block-diagram (default max size : 40 chars)
-xml	-xml	generate an additional description file in xml format
-uim	-user-interface-macros	add user interface macro definitions to the C++ code
-flist	-file-list	list all the source files and libraries implied in a compilation
-norm	-normalized-form	prints the internal signals in normalized form and exits
-lb	-left-balanced	generate left-balanced expressions
-mb	-mid-balanced	generate mid-balanced expressions (default)
-rb	-right-balanced	generate right-balanced expressions
-lt	-less-temporaries	generate less temporaries in compiling delays
-mcd *n*	-max-copy-delay *n*	threshold between copy and ring buffer delays (default 16 samples)

continued on next page

Short	**Long**	**Description**
`-vec`	`-vectorize`	generate easier to vectorize code
`-vs` *n*	`-vec-size` *n*	size of the vector (default 32 samples) when -vec
`-lv` *n*	`-loop-variant` *n*	loop variant [0:fastest (default), 1:simple] when -vec
`-dfs`	`-deepFirstScheduling`	schedule vector loops in deep first order when -vec
`-omp`	`-openMP`	generate parallel code using OpenMP (implies -vec)
`-sch`	`-scheduler`	generate parallel code using threads directly (implies -vec)
`-g`	`-groupTasks`	group sequential tasks together when -omp or -sch is used
`-single`	`-single-precision-floats`	use floats for internal computations (default)
`-double`	`-double-precision-floats`	use doubles for internal computations
`-quad`	`-quad-precision-floats`	use extended for internal computations
`-mdoc`	`-mathdoc`	generates the full mathematical description of a FAUST program
`-mdlang` *l*	`-mathdoc-lang` *l*	choose the language of the mathematical description (l = en, fr, ...)
`-stripmdoc`	`-strip-mdoc-tags`	remove documentation tags when printing FAUST listings
`-cn` *name*	`-class-name` *name*	name of the dsp class to be used instead of 'mydsp'
`-t` *time*	`-timeout` *time*	time out of time seconds (default 600) for the compiler to abort
`-a` *file*		architecture file to use
`-o` *file*		C++ output file

Chapter 5

Embedding the Faust compiler using libfaust

The dynamic compilation chain allows developers to embed the Faust compiler technology directly in their application or plugins. Thanks to the awesome LLVM technology combined with libfaust, the library version of the Faust compiler, Faust DSP programs can directly be compiled and executed on the fly at full speed.

5.1 Dynamic compilation chain

The Faust compiler uses an intermediate FIR representation (Faust Imperative Representation), which can be translated to several output languages. The FIR language describes the computation performed on the samples in a generic manner. It contains primitives to read and write variables and arrays, do arithmetic operations, and define the necessary control structures (for and while loops, if structure etc.).

To generate various output languages, several backends have been developed: for C, C++, Java, JavaScript, asm.js, and LLVM IR. The native LLVM based compilation chain is particularly interesting: it provides direct compilation of a DSP source into executable code in memory, bypassing the external compiler requirement.

5.2 LLVM

LLVM (formerly Low Level Virtual Machine) is a compiler infrastructure, designed for compile-time, link-time, run-time optimization of programs written in arbitrary programming languages. Executable code is produced dynamically using a *Just In Time* compiler from a specific code representation, called LLVM IR. Clang, the LLVM native C/C++/Objective- C compiler is a front-end for LLVM Compiler. It can, for instance, convert a C or C++ source file into LLVM IR code. Domain-specific languages like FAUST can easily target the LLVM IR. This has been done by developing an LLVM IR backend in the FAUST compiler.

5.3 Compiling in memory

The complete chain goes from the FAUST DSP source code, compiled in LLVM IR using the LLVM backend, to finally produce the executable code using the LLVM JIT. All steps take place in memory, getting rid of the classical file based approaches. Pointers to executable functions can be retrieved from the resulting LLVM module and the code directly called with the appropriate parameters.

The FAUST compiler has been packaged as an embeddable library called libfaust, published with an associated API that imitates the concept of oriented-object languages, like C++. Given a FAUST source code (as a file or a string), calling the `createDSPFactoryXXX` function runs the compilation chain (FAUST + LLVM JIT) and generates the *prototype* of the class, as a `llvm_dsp_factory` pointer.

```
class llvm_dsp_factory {

 public:

    /* Return Factory name */
    std::string getName();

    /* Return Factory LLVM target */
    std::string getTarget();

    /* Return Factory SHA key */
    std::string getSHAKey();

    /* Return Factory expanded DSP code */
    std::string getDSPCode();

    /* Create a new DSP instance, to be deleted with C++
       'delete' */
    llvm_dsp* createDSPInstance();

    /* Set a custom memory manager to be used when
       creating instances */
```

```
    void setMemoryManager(dsp_memory_manager* manager);

    /* Return the currently set custom memory manager */
    dsp_memory_manager* getMemoryManager();
};
```

Note that the library keeps an internal cache of all allocated factories so that the compilation of the same DSP code, that is same source code and same set of *normalized* (= sorted in a canonical order) compilations options, will return the same (reference counted) factory pointer. You will have to explicitly use `deleteDSPFactory` to properly decrement the reference counter when the factory is no more needed. You can get a unique SHA1 key of the created factory using its `getSHAKey` method.

Next, the `createDSPInstance` function, corresponding to the `new className` of C++, instantiates a `llvm_dsp` pointer to be used through its interface, connected to the audio chain and controller interfaces. When finished, use `delete` to destroy the dsp instance.

```
class llvm_dsp : public dsp {

 public:

    int getNumInputs();
    int getNumOutputs();

    void buildUserInterface(UI* ui_interface);

    int getSampleRate();

    void init(int sample_rate);
    void instanceInit(int sample_rate);
    void instanceConstants(int sample_rate);
    void instanceResetUserInterface();
    void instanceClear();

    llvm_dsp* clone();

    void metadata(Meta* m);

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

Since `llvm_dsp` is a subclass of the dsp base class, an object of this type can be used with all already available audio and UI classes, in essence reusing all architecture files already developed for the static C++ class compilation scheme (like OSCUI, httpdUI interfaces etc.), look at Developing a new architecture file section.

5.4 Saving/restoring the factory

After the DSP factory has been compiled, your application or plugin may want to save/restore it in order to save Faust to LLVM IR compilation or even JIT compilation time at next use. To get the internal factory compiled code, several functions are available:

- `writeDSPFactoryToIR` allows to get the DSP factory LLVM IR (in textual format) as a string, `writeDSPFactoryToIRFile` allows to save the DSP factory LLVM IR (in textual format) in a file,
- `writeDSPFactoryToBitcode` allows to get the DSP factory LLVM IR (in binary format) as a string, `writeDSPFactoryToBitcodeFile` allows to save the DSP factory LLVM IR (in binary format) in a file,
- `writeDSPFactoryToMachine` allows to get the DSP factory executable machine code as a string, `writeDSPFactoryToMachineFile` allows to save the DSP factory executable machine code in a file.

To re-create a DSP factory from a previously saved code, several functions are available:

- `readDSPFactoryFromIR` allows to create a DSP factory from a string containing the LLVM IR (in textual format), `readDSPFactoryFromIRFile` allows to create a DSP factory from a file containing the LLVM IR (in textual format),
- `readDSPFactoryFromBitcode` allows to create a DSP factory from a string containing the LLVM IR (in binary format), `readDSPFactoryFromBitcodeFile` allows to create a DSP factory from a file containing the LLVM IR (in binary format),
- `readDSPFactoryFromMachine` allows to create a DSP factory from a string containing the executable machine code, `readDSPFactoryFromMachineFile` allows to create a DSP factory from a file containing the executable machine code.

5.5 Additional functions

Some additional functions are available in the libfaust API:

`expandDSPFromString/expandDSPFromFile` creates a self-contained DSP source string where all needed librairies have been included. All compilations options are normalized and included as a comment in the expanded string, `generateAuxFilesFromString/generateAuxFilesFromFile`: from a DSP source string or file, generates auxiliary files: SVG, XML, ps... depending of the argv parameters.

5.6 Using the libfaust library

The libfaust library is part of the FAUST tree. You'll have to compile and install it. Then look at the installed `faust/dsp/llvm-dsp.h` header for a complete description of the API. Note that `faust/dsp/llvm-c-dsp.h` is a pure C version of the same API. The additional functions are available in the `faust/dsp/libfaust.h` header and their C version is in `faust/dsp/libfaust-c.h`.

5.7 Use case examples

The dynamic compilation chain has been used in several projects:

- FaustLive, an integrated IDE for FAUST development
- Faustgen, an external object for Cycling Max/MSP language
- Csound6, see this demo video
- LibAudioStream, a framework to manipulate audio ressources through the concept of streams
- Oliver Larkin JUCE framework integration and pMix2 project
- an experimental version of Antescofo
- FaucK: the combination of the Chuck language and FAUST

Chapter 6

Architecture files

A FAUST program describes a *signal processor*, a pure computation that maps *input signals* to *output signals*. It says nothing about audio drivers or GUI toolkits. This missing information is provided by *architecture files*.

An *architecture file* describes how to relate a FAUST program to the external world, in particular the audio drivers and the user interface to be used. This approach allows a single FAUST program to be easily deployed to a large variety of audio standards (Max/MSP externals, PD externals, VST plugins, CoreAudio applications, Jack applications, iPhone, etc.).

The architecture to be used is specified at compile time with the `-a` options. For example `faust -a jack-gtk.cpp foo.dsp` indicates to use the Jack GTK architecture when compiling `foo.dsp`.

The main available architecture files are listed table 6.1. Since FAUST 0.9.40 some of these architectures are a modular combination of an *audio module* and one or more *user interface modules*. Among these user interface modules OSCUI provide supports for Open Sound Control allowing FAUST programs to be controlled by OSC messages.

6.1 Audio architecture modules

An *audio architecture module* typically connects a FAUST program to the audio drivers. It is responsible for allocating and releasing the audio channels and for calling the FAUST `dsp::compute` method to handle incoming audio buffers and/or to produce audio output. It is also responsible for presenting the audio as non-interleaved float data, normalized between -1.0 and 1.0.

A FAUST audio architecture module derives an *audio* class defined as below:

File name	Description
alchemy-as.cpp	Flash - ActionScript plugin
ca-qt.cpp	CoreAudio QT4 standalone application
jack-gtk.cpp	JACK GTK standalone application
jack-qt.cpp	JACK QT4 standalone application
jack-console.cpp	JACK command line application
jack-internal.cpp	JACK server plugin
alsa-gtk.cpp	ALSA GTK standalone application
alsa-qt.cpp	ALSA QT4 standalone application
oss-gtk.cpp	OSS GTK standalone application
pa-gtk.cpp	PortAudio GTK standalone application
pa-qt.cpp	PortAudio QT4 standalone application
max-msp.cpp	Max/MSP external
vst.cpp	VST plugin
vst2p4.cpp	VST 2.4 plugin
vsti-mono.cpp	VSTi mono instrument
vsti-poly.cpp	VSTi polyphonic instrument
ladspa.cpp	LADSPA plugin
q.cpp	Q language plugin
supercollider.cpp	SuperCollider Unit Generator
snd-rt-gtk.cpp	Snd-RT music programming language
csound.cpp	CSOUND opcode
puredata.cpp	PD external
sndfile.cpp	sound file transformation command
bench.cpp	speed benchmark
octave.cpp	Octave plugin
plot.cpp	Command line application
sndfile.cpp	Command line application

Table 6.1: Some of the available architectures.

```
class audio {
 public:
            audio() {}
  virtual ~audio() {}
  virtual bool init(const char*, dsp*) = 0;
  virtual bool start() = 0;
  virtual void stop() = 0;
  virtual void shutdown(shutdown_callback cb, void* arg)  {}
  virtual int getBufferSize() = 0;
  virtual int getSampleRate() = 0;

  virtual int getNumInputs() = 0;
  virtual int getNumOutputs() = 0;

  virtual float getCPULoad() { return 0.f; }
};
```

The API is simple enough to give a great flexibility to audio architectures implementations. The `init` method should initialize the audio. At `init` exit, the system should be in a safe state to recall the `dsp` object state.

Table 6.2 gives some of the audio architectures currently available for various operating systems.

Audio system	Operating system
Alsa	Linux
CoreAudio	Mac OS X, iOS
JACK	Linux, Mac OS X, Windows
PortAudio	Linux, Mac OS X, Windows
OSC	Linux, Mac OS X, Windows
VST	Mac OS X, Windows
Max/MSP	Mac OS X, Windows
Csound	Linux, Mac OS X, Windows
SuperCollider	Linux, Mac OS X, Windows
PureData	Linux, Mac OS X, Windows
Pure	Linux, Mac OS X, Windows

Table 6.2: Some of FAUST audio architectures.

6.2 UI architecture modules

A UI architecture module links user actions (via graphic widgets, command line parameters, OSC messages, etc.) with the FAUST program to control. It is responsible for associating program parameters to user interface elements and to update parameter's values according to user actions. This association is triggered by the `dsp::buildUserInterface` call, where the `dsp` asks a UI object to build the DSP module controllers.

Since the interface is basically graphic oriented, the main concepts are *widget* based: a UI architecture module is semantically oriented to handle active widgets, passive widgets and widgets layout.

A Faust UI architecture module derives an *UI* class (Figure 6.1).

6.2.1 Active widgets

Active widgets are graphical elements that control a parameter value. They are initialized with the widget name and a pointer to the linked value, using the FAUSTFLOAT macro type (defined at compile time as either float or double). The widget currently considered are `Button`, `CheckButton`, `VerticalSlider`, `HorizontalSlider` and `NumEntry`.
A GUI architecture must implement a method
`addXxx(const char* name, FAUSTFLOAT* zone, ...)` for each active widget. Additional parameters are available for `Slider` and `NumEntry`: the `init`, `min`, `max` and `step` values.

6.2.2 Passive widgets

Passive widgets are graphical elements that reflect values. Similarly to active widgets, they are initialized with the widget name and a pointer to the linked value. The widget currently considered are `HorizontalBarGraph` and `VerticalBarGraph`.
A UI architecture must implement a method
`addXxx(const char* name, FAUSTFLOAT* zone, ...)` for each passive widget. Additional parameters are available, depending on the passive widget type.

6.2.3 Widgets layout

Generally, a GUI is hierarchically organized into boxes and/or tab boxes. A UI architecture must support the following methods to setup this hierarchy :

```
openTabBox(const char* label)
openHorizontalBox(const char* label)
openVerticalBox(const char* label)
closeBox(const char* label)
```

Note that all the widgets are added to the current box.

```
#ifndef FAUSTFLOAT
#define FAUSTFLOAT float
#endif

class UI
{
 public:
           UI() {}
  virtual ~UI() {}

     -- widget layouts
  virtual void openTabBox(const char* l)  = 0;
  virtual void openHorizontalBox(const char* l) = 0;
  virtual void openVerticalBox(const char* l) = 0;
  virtual void closeBox() = 0;

   -- active widgets
  virtual void addButton(const char* l, FAUSTFLOAT* z)
     = 0;
  virtual void addCheckButton(const char* l, FAUSTFLOAT* z)
     = 0;

  virtual void addVerticalSlider(const char* l,
          FAUSTFLOAT* z,
          FAUSTFLOAT init, FAUSTFLOAT min,
          FAUSTFLOAT max, FAUSTFLOAT step) = 0;

  virtual void addHorizontalSlider(const char* l,
          FAUSTFLOAT* z,
          FAUSTFLOAT init, FAUSTFLOAT min,
          FAUSTFLOAT max, FAUSTFLOAT step) = 0;

  virtual void addNumEntry(const char* l, FAUSTFLOAT* z,
          FAUSTFLOAT init, FAUSTFLOAT min,
          FAUSTFLOAT max, FAUSTFLOAT step)  = 0;

   -- passive widgets
  virtual void addHorizontalBargraph(const char* l,
                       FAUSTFLOAT* z, FAUSTFLOAT min,
                       FAUSTFLOAT max) = 0;

  virtual void addVerticalBargraph(const char* l,
                       FAUSTFLOAT* z, FAUSTFLOAT min,
                       FAUSTFLOAT max) = 0;

   -- metadata declarations
  virtual void declare(FAUSTFLOAT*, const char*, const char*)
      {}
};
```

Figure 6.1: UI, the root user interface class.

UI	Comment
console	a textual command line UI
GTKUI	a GTK-based GUI
QTGUI	a multi-platform QT-based GUI
FUI	a file-based UI to store and recall modules states
OSCUI	OSC control (see section 7)
httpdUI	HTTP control (see section 8)
...	...

Table 6.3: Some of the available UI architectures.

6.2.4 Metadata

The FAUST language allows widget labels to contain metadata enclosed in square brackets as key/value pairs. These metadata are handled at GUI level by a `declare` method taking as argument, a pointer to the widget associated zone, the metadata key and value:

```
declare(FAUSTFLOAT* zone, const char* key, const char* value)
```

Here is the table of currently supported general medatada (look at section 7 for OSC specific metadata and section 9 for MIDI specific metadata):

Key	Value
tooltip	actual string content
hidden	`0` or `1`
size	actual value
unit	`Hz` or `dB`
scale	`log` or `exp`
style	`knob` or `led` or `numerical`
style	`radio{'label1':v1;'label2':v2...}`
style	`menu{'label1':v1;'label2':v2...}`
acc	`axe curve amin amid amax`
gyr	`axe curve amin amid amax`
screencolor	`red` or `green` or `blue` or `white`

Table 6.4: Supported medatada.

Some typical example where several metadata are defined could be:

```
nentry("freq [unit:Hz][scale:log][acc:0 0 -30 0 30][style:menu{'white
noise':0;'pink noise':1;'sine':2}][hidden:0]", 0, 20, 100, 1)
```

or:

```
vslider("freq [unit:dB][style:knob][gyr:0 0 -30 0 30]", 0, 20, 100,
1)
```

Note that medatada are not supported in all architecture files. Some of them like (`acc` or `gyr` for exemple) only make sense on platforms with accelerometers or gyroscopes sensors. The set of medatada may be extended in the future.

6.3 Developing a new architecture file

Developing a new architecture file typically means writing a generic C++ file, that will be populated with the actual output of the Faust compiler, in order to produce a complete C++ file, ready to be compiled as a standalone application or plugin.

The architecture to be used is specified at compile time with the `-a` option. It must contain the `<<includeIntrinsic>>` and `<<includeclass>>` lines that will be looked at by the Faust compiler, and replaced by the generated C++ class.

Look at the `minimal.cpp` example located in the architecture folder:

```
#include "faust/gui/PrintUI.h"
#include "faust/gui/meta.h"
#include "faust/audio/dummy-audio.h"

using std::max;
using std::min;

//-------------------------------------------------
//  FAUST generated signal processor
//-------------------------------------------------

<<includeIntrinsic>>

<<includeclass>>

int main(int argc, char *argv[])
{
    mydsp DSP;
    PrintUI ui;

    // Activate the UI
    // (here that only prints the control paths)
    DSP.buildUserInterface(&ui);

    // Allocate the audio driver to render
    // 5 buffers of 512 frames
    dummyaudio audio(5);
```

```
    audio.init("Test", &DSP);

    // Render buffers...
    audio.start();

    audio.stop();
}
```

Calling `faust -a minimal.cpp noise.dsp -a noise.cpp` will produce a ready to compile `noise.cpp` file:

```
#include "faust/gui/PrintUI.h"
#include "faust/gui/meta.h"
#include "faust/audio/dummy-audio.h"

using std::max;
using std::min;

//-----------------------------------------------------
//  FAUST generated signal processor
//-----------------------------------------------------

#ifndef FAUSTFLOAT
#define FAUSTFLOAT float
#endif

#ifndef FAUSTCLASS
#define FAUSTCLASS mydsp
#endif

class mydsp : public dsp {
  private:
    FAUSTFLOAT fslider0;
    int iRec0[2];
    int fSamplingFreq;

  public:
    virtual void metadata(Meta* m) {
        m->declare("name", "Noise");
        m->declare("version", "1.1");
        m->declare("author", "Grame");
        m->declare("license", "BSD");
        m->declare("copyright", "(c)GRAME 2009");
    }

    virtual int getNumInputs() { return 0; }
    virtual int getNumOutputs() { return 1; }
    static void classInit(int samplingFreq) {
    }
```

```
    virtual void instanceConstants(int samplingFreq) {
        fSamplingFreq = samplingFreq;
    }
    virtual void instanceResetUserInterface() {
        fslider0 = 0.5f;
    }
    virtual void instanceClear() {
        for (int i=0; i<2; i++) iRec0[i] = 0;
    }
    virtual void init(int samplingFreq) {
        classInit(samplingFreq);
        instanceInit(samplingFreq);
    }
    virtual void instanceInit(int samplingFreq) {
        instanceConstants(samplingFreq);
        instanceResetUserInterface();
        instanceClear();
    }
    virtual mydsp* clone() {
        return new mydsp();
    }
    virtual int getSampleRate() {
        return fSamplingFreq;
    }
    virtual void buildUserInterface(UI* ui_interface) {
        ui_interface->openVerticalBox("Noise");
        ui_interface->declare(&fslider0, "acc", "0 0 -10 0 10
           ");
        ui_interface->declare(&fslider0, "style", "knob");
        ui_interface->addVerticalSlider("Volume", &fslider0,
           0.5f, 0.0f, 1.0f, 0.1f);
        ui_interface->closeBox();
    }
    virtual void compute (int count, FAUSTFLOAT** input,
       FAUSTFLOAT** output) {
        float fSlow0 = (4.656613e-10f * float(fslider0));
        FAUSTFLOAT* output0 = output[0];
        for (int i=0; i<count; i++) {
            iRec0[0] = ((1103515245 * iRec0[1]) + 12345);
            output0[i] = (FAUSTFLOAT)(fSlow0 * iRec0[0]);
            // post processing
            iRec0[1] = iRec0[0];
        }
    }
};

int main(int argc, char* argv[])
{
    mydsp DSP;
```

```
    PrintUI ui;

    // Activate the UI
    // (here that only prints the control paths)
    DSP.buildUserInterface(&ui);

    // Allocate the audio driver to render
    // 5 buffers of 512 frames
    dummyaudio audio(5);
    audio.init("Test", &DSP);

    // Render buffers...
    audio.start();

    audio.stop();
}
```

You can possibly add the `-i` option to actually inline all `#include "faust/xxx/yyy"` headers (all files starting with "faust"). Then you will have to write a `faust2xxx` script that will chain the FAUST compilation step and the C++ compilation one. Look at scripts in the tools/faust2appls folder for real examples.

Developing the adapted C++ file may require "aggregating" the generated `mydsp` class (subclass of `dsp` base class defined in faust/dsp/dsp.h header) in your specific class, or "subclassing" and extend it. So you will have to write something like:

```
class my_class : public base_interface {

  private:

    mydsp fDSP;

  public:

    my_class()
    {
        // Do something specific
    }

    virtual ~my_class()
    {
        // Do something specific
    }

    // Do something specific

    void my_compute(int count,
        FAUSTFLOAT** inputs,
        FAUSTFLOAT** outputs,....)
```

```
    {
        // Do something specific
        fDSP.compute(count, inputs, outputs);
    }

    // Do something specific
};
```

or:

```
class my_class : public mydsp {

  private:

    // Do something specific

  public:

    my_class()
    {
        // Do something specific
    }

    virtual ~my_class()
    {
        // Do something specific
    }

    // Do something specific

    void my_compute(int count,
        FAUSTFLOAT** inputs,
        FAUSTFLOAT** outputs,....)
    {
        // Do something specific
        compute(count, inputs, outputs);
    }

    // Do something specific
};
```

This way your architecture file will be adapted to any "shape" of the generated code. That is, depending if you generate purely scalar, or vector code (using the `-vec` option), or any other option, the generated `mydsp` class will always be correctly inserted in the final C++ file. Look for instance at `csound.cpp` and `unity.cpp` architecture files in the architecture folder for real examples.

Chapter 7

OSC support

Most FAUST architectures provide Open Sound Control (OSC) support [1]. This allows FAUST applications to be remotely controlled from any OSC capable application, programming language, or hardware device. OSC support can be activated using the `-osc` option when building the application with the appropriate `faust2xxx` command. The following table (table 7.1) lists FAUST's architectures which provide OSC support.

7.1 A simple example

To illustrate how OSC support works let's define a very simple noise generator with a level control: `noise.dsp`

```
process = library("music.lib").noise
        * hslider("level", 0, 0, 1, 0.01);
```

We are going to compile this example as a standalone Jack QT application with OSC support using the command:

```
faust2jaqt -osc noise.dsp
```

When we start the application from the command line:

```
./noise
```

we get various information on the standard output, including:

```
Faust OSC version 0.93 application 'noise' is running
    on UDP ports 5510, 5511, 5512
```

[1]The implementation is based internally on the *oscpack* library by Ross Bencina

Audio system	Environment	OSC support
Linux		
Alsa	GTK, Qt, Console	yes
Jack	GTK, Qt, Console	yes
Netjack	GTK, Qt, Console	yes
PortAudio	GTK, Qt	yes
Mac OS X		
CoreAudio	Qt	yes
Jack	Qt, Console	yes
Netjack	Qt, Console	yes
PortAudio	Qt	yes
Windows		
Jack	Qt, Console	yes
PortAudio	Qt	yes

Table 7.1: Faust architectures with OSC support.

As we can see the OSC module makes use of three different UDP ports:

- `5510` is the listening port number: control messages should be addressed to this port.
- `5511` is the output port number: control messages sent by the application and answers to query messages are sent to this port.
- `5512` is the error port number: used for asynchronous error notifications.

Note that if a `declare name "Foo";` line is present in the DSP program, `Foo` will be used as the OSC root name, otherwise the DSP filename will be used instead.

These OSC parameters can be changed from the command line using one of the following options:

- `-port number` sets the port number used by the application to receive messages.
- `-outport number` sets the port number used by the application to transmit messages.
- `-errport number` sets the port number used by the application to transmit error messages.

- `-desthost host` set sthe destination host for the messages sent by the application. Note that the destination address can be changed with the first incoming message: first received packet from another host sets the destination address to this host.

- `-xmit 0|1|2` turns transmission OFF, ALL, or ALIAS (default OFF). When transmission is OFF, input elements can be controlled using their addresses or aliases (if present). When transmission is ALL, input elements can be controlled using their addresses or aliases (if present), user's actions and output elements (bargraph) are transmitted as OSC messages as well as aliases (if present). When transmission is ALIAS, input elements can only be controlled using their aliases, user's actions and output elements (bargraph) are transmitted as aliases only.

- `-xmitfilter path` allows to filter output messages. Note that 'path' can be a regular expression (like "/freeverb/Reverb1/*").

- `-bundle 0|1` activates the bundle mode where all transmitted values are sent in a single message, to be used together with the `-xmit` option.

For example:

```
./noise -xmit 1 -desthost 192.168.1.104 -outport 6000
```

will run noise with transmission mode ON, using 192.168.1.104 on port 6000 as destination.

7.2 Automatic port allocation

In order to address each application individually, only one application can be listening on a single port at one time. Therefore when the default incoming port 5510 is already opened by some other application, an application will automatically try increasing port numbers until it finds an available port. Let's say that we start two applications `noise` and `mixer` on the same machine, here is what we get:

```
$ ./noise &
...
Faust OSC version 0.93 application 'noise' is running
    on UDP ports 5510, 5511, 5512
$ ./mixer
...
Faust OSC version 0.93 application 'mixer' is running
    on UDP ports 5513, 5511, 5512
```

The mixer application fails to open the default incoming port 5510 because it is already opened by noise. Therefore it tries to find an available port starting from 5513 and open it. Please note that the two outcoming ports 5511 and 5512 are shared by all running applications.

7.3 Discovering OSC applications

oscsend *hostname port address types values*: send OpenSound Control message via UDP. *types* is a string, the letters indicates the type of the following values: i=integer, f=float, s=string,...

oscdump *port* : receive OpenSound Control messages via UDP and dump to standard output

The commands oscsend Send OpenSound Control message via UDP. and oscdump from the liblo package provide a convenient mean to experiment with OSC control. For the experiment let's use two additional terminals. The first one will be used to send OSC messages to the noise application using oscsend. The second terminal will be used to monitor the messages sent by the application using oscdump. We will indicate by T1$ the command types on terminal T1 and by T2: the messages received on terminal T2. To monitor on terminal T2 the OSC messages received on UDP port 5511 we will use oscdump:

```
T2$ oscdump 5511
```

Once set we can use the hello message to scan UDP ports for FAUST applications. For example:

```
T1$ oscsend localhost 5510 "/*" s hello
```

gives us the root message address, the network and the UDP ports used by the noise application:

```
T2: /noise siii "192.168.1.102" 5510 5511 5512
```

7.4 Discovering the OSC interface of an application

Once we have an application we can discover its OSC interface (the set of OSC messages we can use to control it) by sending the get message to the root:

```
T1$ oscsend localhost 5510 /noise s get
```

As an answer of the osc messages understood by the application, a full description is available on terminal T2:

```
T2: /noise sF "xmit" #F
T2: /noise ss "desthost" "127.0.0.1"
T2: /noise si "outport" 5511
T2: /noise si "errport" 5512
T2: /noise/level fff 0.000000 0.000000 1.000000
```

The root of the osc interface is `/noise`. Transmission is OFF, `xmit` is set to false. The destination host for sending messages is `"127.0.0.1"`, the output port is `5511` and the error port is `5512`. The application has only one user interface element: `/noise/level` with current value `0.0`, minimal value `0.0` and maximal value `1.0`.

7.5 Widget's OSC address

Each widget of an application has a unique OSC address obtained by concatenating the labels of it's surrounding groups with its own label. Here is as an example `mix4.dsp`, a very simplified monophonic audio mixer with 4 inputs and one output. For each input we have a mute button and a level slider:

There are potential conflicts between widget's labels and the OSC address space. An OSC symbolic name is an ASCII string consisting of a restricted set of printable characters. Therefore to ensure compatibility spaces are replaced by underscores and some other characters (asterisk, comma, forward, question mark, open bracket, close bracket, open curly brace, close curly brace) are replaced by hyphens.

```
input(v) = vgroup("input %v", *(1-checkbox("mute")) :
    *(vslider("level", 0, 0, 1, 0.01)));
process = hgroup("mixer", par(i, 4, input(i)) :> _);
```

If we query this application:

```
T1$ oscsend localhost 5510 "/*" s get
```

We get a full description of its OSC interface on terminal T2:

```
T2: /mixer sF "xmit" #F
T2: /mixer ss "desthost" "127.0.0.1"
T2: /mixer si "outport" 5511
T2: /mixer si "errport" 5512
T2: /mixer/input_0/level fff 0.0000 0.0000 1.0000
T2: /mixer/input_0/mute  fff 0.0000 0.0000 1.0000
T2: /mixer/input_1/level fff 0.0000 0.0000 1.0000
T2: /mixer/input_1/mute  fff 0.0000 0.0000 1.0000
T2: /mixer/input_2/level fff 0.0000 0.0000 1.0000
T2: /mixer/input_2/mute  fff 0.0000 0.0000 1.0000
T2: /mixer/input_3/level fff 0.0000 0.0000 1.0000
T2: /mixer/input_3/mute  fff 0.0000 0.0000 1.0000
```

As we can see each widget has a unique OSC address obtained by concatenating the top level group label "mixer", with the "input" group label and the widget label. Please note that in this operation whites spaces are replaced by underscores and metadata are removed.

All addresses must have a common root. This is the case in our example because there is a unique horizontal group "mixer" containing all widgets. If a common root is missing as in the following code:

```
    input(v) = vgroup("input %v", *(1-checkbox("mute")) :
        *(vslider("level", 0, 0, 1, 0.01)));
    process = par(i, 4, input(i)) :> _;
```

then a default vertical group is automatically create by the Faust compiler using the name of the file `mix4` as label:

```
    T2: /mix4 sF "xmit" #F
    T2: /mix4 ss "desthost" "127.0.0.1"
    T2: /mix4 si "outport" 5511
    T2: /mix4 si "errport" 5512
    T2: /mix4/input_0/level fff 0.0000 0.0000 1.0000
    T2: /mix4/input_0/mute  fff 0.0000 0.0000 1.0000
    T2: /mix4/input_1/level fff 0.0000 0.0000 1.0000
    T2: /mix4/input_1/mute  fff 0.0000 0.0000 1.0000
    T2: /mix4/input_2/level fff 0.0000 0.0000 1.0000
    T2: /mix4/input_2/mute  fff 0.0000 0.0000 1.0000
    T2: /mix4/input_3/level fff 0.0000 0.0000 1.0000
    T2: /mix4/input_3/mute  fff 0.0000 0.0000 1.0000
```

7.6 Controlling the application via OSC

We can control any user interface element of the application by sending one of the previously discovered messages. For example to set the noise level of the application to `0.2` we send:

```
    T1$ oscsend localhost 5510 /noise/level f 0.2
```

If we now query `/noise/level` we get, as expected, the value `0.2`:

```
    T1$ oscsend localhost 5510 /noise/level s get
    T2: /noise/level fff 0.2000 0.0000 1.0000
```

7.7 Turning transmission ON

The xmit message at the root level is used to control the realtime transmission of OSC messages corresponding to user interface's actions. For examples:

```
    T1$ oscsend localhost 5510 /noise si xmit 1
```

turns transmission in ALL mode. Now if we move the level slider we get a bunch of messages:

```
T2: /noise/level f 0.024000
T2: /noise/level f 0.032000
T2: /noise/level f 0.105000
T2: /noise/level f 0.250000
T2: /noise/level f 0.258000
T2: /noise/level f 0.185000
T2: /noise/level f 0.145000
T2: /noise/level f 0.121000
T2: /noise/level f 0.105000
T2: /noise/level f 0.008000
T2: /noise/level f 0.000000
```

This feature can be typically used for automation to record and replay actions on the user interface, or to remote control from one application to another. It can be turned OFF any time using:

```
T1$ oscsend localhost 5510 /noise si xmit 0
```

Use the ALIAS (xmit = 2) mode if you need restricted access to your program: when ALIAS is mode is used, only aliases of input elements (sliders, buttons...) can be used to control them, and output elements (bargraph) will only emit on their aliases.

7.8 Filtering OSC messages

When the transmission of OSC messages is ON, all the user interface elements are sent through the OSC connection.

```
T2: /harpe/level f 0.024000
T2: /harpe/hand f 0.1
T2: /harpe/level f 0.024000
T2: /harpe/hand f 0.25
T2: /harpe/level f 0.024000
T2: /harpe/hand f 0.44
T2: /noise/level f 0.145000
T2: /harpe/hand f 0.78
T2: /noise/level f 0.145000
T2: /harpe/hand f 0.99
```

We can choose to filter the unwanted parameters (or group of parameters). For example:

```
T1$ oscsend localhost 5510 /harpe si xmit 1
   xmitfilter /harpe/level
```

As a result, we will receive:

```
T2: /harpe/hand f 0.1
T2: /harpe/hand f 0.25
T2: /harpe/hand f 0.44
T2: /harpe/hand f 0.78
```

To reset the filter, send:

```
T1$ oscsend localhost 5510 /harpe si xmit 1
   xmitfilter
```

7.9 Using OSC aliases

Aliases are a convenient mechanism to control a FAUST application from a preexisting set of OSC messages.

Let's say we want to control our noise example with touchOSC on Android. The first step is to configure TouchOSC host to 192.168.1.102 (the host running our noise application) and outgoing port to 5510.

Then we can use oscdump 5510 (after quitting the noise application in order to free port 5510) to visualize the OSC messages sent by TouchOSC. Let's use for that the left slider of simple layout. Here is what we get:

```
T2: /1/fader1 f 0.000000
T2: /1/fader1 f 0.004975
T2: /1/fader1 f 0.004975
T2: /1/fader1 f 0.008125
T2: /1/fader1 f 0.017473
T2: /1/fader1 f 0.032499
T2: /1/fader1 f 0.051032
T2: ...
T2: /1/fader1 f 0.993289
T2: /1/fader1 f 1.000000
```

We can associate this OSC message to the noise level slider by inserting the metadata [osc:/1/fader1 0 1] into the slider's label:

Several osc aliases can be inserted into a single label allowing the same widget to be controlled by several OSC messages.

```
process = library("music.lib").noise * hslider("level
   [osc:/1/fader1 0 1]",0,0,1,0.01);
```

Because here the range of /1/fader1 is 0 to 1 like the level slider we can remove the range mapping information and write simply :

```
process = library("music.lib").noise * hslider("level
   [osc:/1/fader1]", 0, 0, 1, 0.01);
```

TouchOSC can also send accelerometer data by enabling Settings/Options/Accelerometer. Using again oscdump 5510 we can visualize the messages send by TouchOSC:

```
T2: ...
T2: /accxyz fff -0.147842 0.019752 9.694721
T2: /accxyz fff -0.157419 0.016161 9.686341
T2: /accxyz fff -0.167594 0.012570 9.683948
T2: ...
```

As we can see TouchOSC send the x, y and z accelerometers in a single message, as a triplet of values ranging approximatively from -9.81 to 9.81. In order to select the appropriate accelerometer we need to concatenate to /accxyz a suffix /0, /1 or /2. For example /accxyz/0 will correspond to x, /accxyz/1 to y, etc. We also need to define a mapping because the ranges are different:

```
process = library("music.lib").noise * hslider("level[
   osc:/accxyz/0 0 9.81]",0,0,1,0.01);
```

alias	description
[osc:/1/rotary1 0 1]	top left rotary knob
[osc:/1/rotary2 0 1]	middle left rotary knob
[osc:/1/rotary3 0 1]	bottom left rotary knob
[osc:/1/push1 0 1]	bottom left push button
[osc:/1/push2 0 1]	bottom center left push button
[osc:/1/toggle1 0 1]	top center left toggle button
[osc:/1/toggle2 0 1]	middle center left toggle button
[osc:/1/fader1 0 1]	center left vertical fader
[osc:/1/toggle3 0 1]	top center right toggle button
[osc:/1/toggle4 0 1]	middle center right toggle button
[osc:/1/fader2 0 1]	center right vertical toggle button
[osc:/1/rotary4 0 1]	top right rotary knob
[osc:/1/rotary5 0 1]	middle right rotary knob
[osc:/1/rotary6 0 1]	bottom right rotary knob
[osc:/1/push3 0 1]	bottom center right push button
[osc:/1/push4 0 1]	bottom right push button
[osc:/1/fader3 0 1]	bottom horizontal fader
[osc:/accxyz/0 -10 10]	x accelerometer
[osc:/accxyz/1 -10 10]	y accelerometer
[osc:/accxyz/2 -10 10]	z accelerometer

Table 7.2: Examples of OSC message aliases for TouchOSC (layout Mix2).

7.10 OSC cheat sheet

Default ports

`5510`	default listening port
`5511`	default transmission port
`5512`	default error port
`5513...`	alternative listening ports

Command line options

`-port` *n*	set the port number used by the application to receive messages
`-outport` *n*	set the port number used by the application to transmit messages
`-errport` *n*	set the port number used by the application to transmit error messages
`-desthost` *h*	set the destination host for the messages sent by the application
`-xmit 0|1|2`	turn transmission OFF, ALL or ALIAS (default OFF)
`-xmitfilter` *s*	filter the Faust paths at emission time

Discovery messages

`oscsend` *host port* `"/*" s hello`	discover if any OSC application is listening on port *port*
`oscsend` *host port* `"/*" s get`	query OSC interface of application listening on port *port*
`oscsend` *host port* `"/*" s json`	query JSON description of application listening on port *port*

Control messages

`oscsend` *host port* `"/*" si xmit 0|1|2`	set transmission mode
`oscsend` *host port widget* `s get`	get widget's value
`oscsend` *host port widget* `f` *v*	set widget's value

Alias

`"...[osc:` *address lo hi* `]..."`	alias with *lo*→*min*, *hi*→*max* mapping
`"...[osc:` *address*`]..."`	alias with *min*, *max* clipping

7.11 DSP with polyphonic support

When the DSP code is compiled in polyphonic mode, the generated program will create a more complex hierarchy to possibly access and control individual voices.

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

/Organ/vol f -10.0
/Organ/pan f 0.0
```

```
// Polyphonic mode

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

Note that to save space on the screen, the `/Polyphonic/VoiceX/xxx` syntax is used when the number of allocated voices is less than 8, then the `/Polyphonic/VX/xxx` syntax is used when more voices are used.

Chapter 8

HTTP support

Similarly to OSC, several FAUST architectures also provide HTTP support. This allows FAUST applications to be remotely controlled from any Web browser using specific URLs. Moreover OSC and HTTPD can be freely combined.

While OSC support is installed by default when FAUST is build, this is not the case for HTTP. That's because it depends on GNU *libmicrohttpd* library which is usually not installed by default on the system. An additional `make httpd` step is therefore required when compiling and installing FAUST:

```
make httpd
make
sudo make install
```

Note that `make httpd` will fail if *libmicrohttpd* is not available on the system.

The HTTP support can be activated using the `-httpd` option when building the audio application with the appropriate `faust2xxx` command. The following table (table 8.1) lists FAUST's architectures which provide HTTP support.

8.1 A simple example

To illustrate how HTTP support works let's reuse our previous `mix4.dsp` example, a very simplified monophonic audio mixer with 4 inputs and one output. For each input we have a mute button and a level slider:

```
input(v) = vgroup("input %v", *(1-checkbox("mute"))
         : *(vslider("level", 0, 0, 1, 0.01)));
process  = hgroup("mixer", par(i, 4, input(i)) :> _);
```

Audio system	Environment	HTTP support
Linux		
Alsa	GTK, Qt, Console	yes
Jack	GTK, Qt, Console	yes
Netjack	GTK, Qt, Console	yes
PortAudio	GTK, Qt	yes
Mac OS X		
CoreAudio	Qt	yes
Jack	Qt, Console	yes
Netjack	Qt, Console	yes
PortAudio	Qt	yes
Windows		
Jack	Qt, Console	yes
PortAudio	Qt	yes

Table 8.1: Faust architectures with HTTP support.

We are going to compile this example as a standalone Jack QT application with HTTP support using the command:

```
faust2jaqt -httpd mix4.dsp
```

Th effect of the `-httpd` is to embed a small Web server into the application, which purpose is to serve an HTML page representing its user interface. This page makes use of JavaScript and SVG and is quite similar to the native QT interface.

When we start the application from the command line:

```
./mix4
```

we get various information on the standard output, including:

```
Faust httpd server version 0.72 is running on TCP port
   5510
```

As we can see the embedded Web server is running by default on TCP port 5510. The entry point is http://localhost:5510. It can be open from any recent browser and it produces the page reproduced figure 8.1.

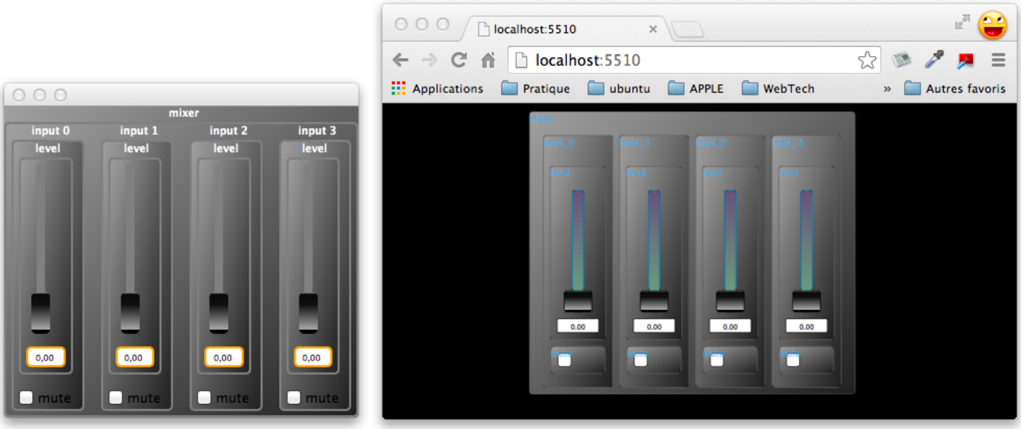


Figure 8.1: User interface of mix4.dsp in a Web browser

8.2 JSON description of the user interface

The communication between the application and the Web browser is based on several underlying URLs. The first one is http://localhost:5510/JSON that return a json description of the user interface of the application. This json description is used internally by the JavaScript code to build the graphical user interface. Here is (part of) the json returned by `mix4`:

```
{
  "name": "mix4",
  "address": "YannAir.local",
  "port": "5511",
  "ui": [
    {
      "type": "hgroup",
      "label": "mixer",
      "items": [
        {
          "type": "vgroup",
          "label": "input_0",
          "items": [
            {
              "type": "vslider",
              "label": "level",
              "address": "/mixer/input_0/level",
              "init": "0", "min": "0", "max": "1",
              "step": "0.01"
            },
```

```
            {
              "type": "checkbox",
              "label": "mute",
              "address": "/mixer/input_0/mute",
              "init": "0", "min": "0", "max": "0",
              "step": "0"
            }
          ]
        },

        ...

      ]
    }
  ]
}
```

8.3 Quering the state of the application

Each widget has a unique "address" field that can be used to query its value. In our example here the level of the input 0 has the address `/mixer/input_0/level`. The address can be used to forge an URL to get the value of the widget: http://localhost:5510/mixer/input_0/level, resulting in:

```
/mixer/input_0/level 0.00000
```

Multiple widgets can be query at once by using an address higher in the hierarchy. For example to get the values of the level and the mute state of input 0 we use http://localhost:5510/mixer/input_0, resulting in:

```
/mixer/input_0/level 0.00000
/mixer/input_0/mute  0.00000
```

To get the all the values at once we simply use http://localhost:5510/mixer, resulting in:

```
/mixer/input_0/level 0.00000
/mixer/input_0/mute  0.00000
/mixer/input_1/level 0.00000
/mixer/input_1/mute  0.00000
/mixer/input_2/level 0.00000
/mixer/input_2/mute  0.00000
/mixer/input_3/level 0.00000
/mixer/input_3/mute  0.00000
```

8.4 Changing the value of a widget

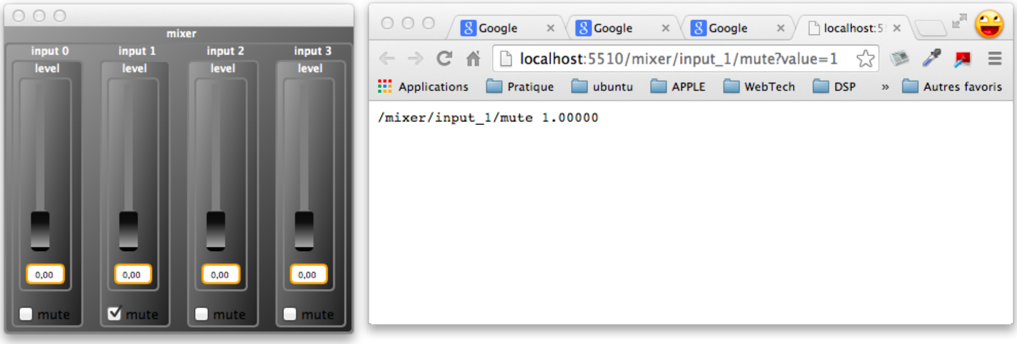


Figure 8.2: Muting input 1 by forging the appropriate URL

Let's say that we want to mute input 1 of our mixer. We can use for that purpose the URL http://localhost:5510/mixer/input_1/mute?value=1 obtained by concatenating ?value=1 at the end of the widget URL.

All widgets can be controlled similarly. For example http://localhost:5510/mixer/input_3/level?value=0.7 will sets the input 3 level to 0.7.

8.5 Proxy control access to the Web server

A control application may want to access and control the running DSP using its Web server, but without using the delivered HTML page in a browser. Since the complete json can be retrieved, control applications can purely be developed in C/C++, then build a *proxy* version of the use interface, and set and get parameters using HTTP requests.

This mode can be started dynamically using the *-server URL* parameter. Assuming an application with HTTP support is running remotely on the given URL, the control application will fetch its json description, use it to dynamically build the user interface, and allow to access the remote parameters.

8.6 HTTP cheat sheet

Here is a summary of the various URLs used to interact with the application's Web server.

Default ports

`5510`	default TCP port used by the application's Web server
`5511...`	alternative TCP ports

Command line options

`-port` *n*	set the TCP port number used by the application's Web server
`-server` *URL*	start a proxy control application accessing the remote application running on the give

URLs

`http://`*host*`:`*port*	the base URL to be used in proxy control access mode
`http://`*host*`:`*port*`/JSON`	get a json description of the user interface
`http://`*host*`:`*port*`/`*address*	get the value of a widget or a group of widgets
`http://`*host*`:`*port*`/`*address*`?value=`*v*	set the value of a widget to v

JSON

Top level

The json describes the name, host and port of the application and a hierarchy of user interface items:

```
{
  "name": <name>,
  "address": <host>,
  "port": <port>,
  "ui": [ <item> ]
}
```

An `<item>` is either a group (of items) or a widget.

Groups

A group is essentially a list of items with a specific layout:

```
{
    "type": <type>,
    "label": <label>,
    "items": [ <item>, <item>,...]
}
```

The `<type>` defines the layout. It can be either `"vgroup"`, `"hgroup"` or `"tgroup"`

Widgets

```
{
    "type": <type>,
    "label": <label>,
    "address": <address>,
    "meta": [ { "key": "value"},... ],
    "init": <num>,
    "min": <num>,
    "max": <num>,
    "step": <num>
},
```

Widgets are the basic items of the user interface. They can be of different `<type>` : `"button"`, `"checkbox"`, `"nentry"`, `"vslider"`, `"hslider"`, `"vbargraph"` or `"hbargraph"`.

Chapter 9

MIDI support

Similarly to OSC, several Faust architectures also provide MIDI support. This allows Faust applications to be controlled from any MIDI device (or to control MIDI devices). MIDI is also the preferable way to control Polyphonic instruments.

9.1 MIDI messages description in the dsp source code

MIDI control messages are described as metadata in UI elements. They are decoded by a special architecture *MidiUI* class that will parse incoming MIDI messages and update the appropriate control parameters, or send MIDI messages when the UI elements (sliders, buttons...) are moved.

9.2 Description of the possible standard MIDI messages

Below, when a 7-bit MIDI parameter is used to drive a button or checkbox, its maximum value 127 maps to 1 ("on") while its minimum value 0 maps to 0 ("off").

A special `[midi:xxx yyy...]` metadata needs to be added in the UI element description. The more usual MIDI messages can be used as described here:

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

- `[midi:keyon pitch]` in a slider or bargraph will register the UI element's state-variable to be driven by MIDI note-on velocity (an integer between 0 and 127) of the specified key between 0 and 127. 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 register the UI element's state-variable to be driven by MIDI note-off velocity (an integer between 0 and 127) of the specified key between 0 and 127. When used with a button or checkbox, 1 will be mapped to 127, 0 will be mapped to 0,

- `[midi:key pitch]` in a slider or bargraph will register the UI element's state-variable to be driven by MIDI note-on velocity (an integer between 0 and 127) of the specified key between 0 and 127. When used with a button or checkbox, 1 will be mapped to 127, 0 will be mapped to 0. Note-on and note-off events will be handled,

- `[midi:keypress pitch]` in a slider or bargraph will register the UI element's state-variable to be driven by the MIDI key-pressure (an integer between 0 and 127) from MIDI key,

- `[midi:pgm num]` in a slider or bargraph will map the UI element value to the progchange value, so *progchange* message with the same *num* value will be sent. When used with a button or checkbox, 1 will send the *progchange* message with *num* value, 0 will send nothing,

- `[midi:chanpress num]` in a slider or bargraph will map the UI element value to the chanpress value, so *chanpress* message with the same *num* value will be sent. When used with a button or checkbox, 1 will send the *chanpress* message with *num* value, 0 will send nothing,

- `[midi:pitchwheel]` in a slider or bargraph will map the UI element value to (0,16383) range. When used with a button or checkbox, 1 will be mapped to 16383, 0 will be mapped to 0.

9.3 A simple example

An example with a *volume* slider controlled with MIDI ctrlchange 7 messages :

```
//-----------------------------------------------
//  Volume MIDI control in dB
//-----------------------------------------------

import("music.lib");

smooth(c) = *(1-c) : +~*(c);
gain = vslider("Volume [midi:ctrl 7]", 0, -70, +4, 0.1) :
    db2linear : smooth(0.999);
```

```
process = *(gain);
```

A complete testing example named *midi_tester.dsp* is available in the FAUST distribution *examples* folder.

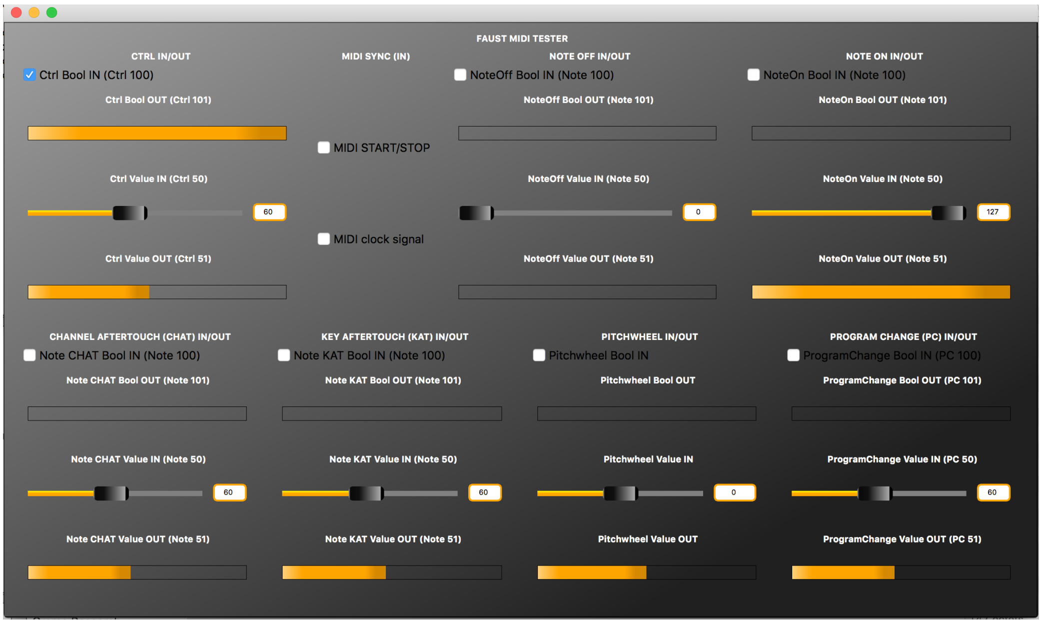


Figure 9.1: MIDI messages testing example

The MIDI support can be activated using the `-midi` option when building the audio application with the appropriate `faust2xxx` command. The following table (table 9.1) lists FAUST's architectures which provide MIDI support.

Audio system	**Environment**	**HTTP support**
Linux		
Alsa	Qt	yes
Jack	Qt	yes
Mac OS X		
CoreAudio	Qt	yes
Jack	Qt	yes

Table 9.1: FAUST architectures with HTTP support.

9.4 MIDI synchronization

MIDI clock based synchronization can be used to slave a given Faust program. The following three messages need 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 stream to possibly compute the BPM information, or for any synchronization need it may have. Here is a simple example of a sinus generated which a frequency controlled by the MIDI clock stream, and starting/stopping when receiving the MIDI start/stop messages:

```
import("music.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@SR) : + ~ _;

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

Chapter 10

Polyphonic support

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[1].

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

10.1 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.

10.2 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:

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

- 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[2], followed by one tab for each voice. Graphical User Interface components will then reflect the multi-voices structure of the new polyphonic DSP (Figure 10.1).

Figure 10.1: 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

[2] An internal control grouping mechanism has been defined to automatically dispatch a user interface action done on the master component on all linked voices, except for the *freq*, *gain* and *gate* controls.

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[3], the desired number of voices, the *dynamic voice allocation* state[4], and the *group* state which controls if separated voices are displayed or not (Figure 10.1):

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

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);
```

10.3 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.

[3]The DSP object will be automatically cloned in the mydsp_poly class to create all needed voices.
[4]Voices may be always running, or dynamically started/stopped in case of MIDI control.

10.4 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[5]. 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.

10.5 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 10.2).

`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.

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

10.5.1 Integrated global output effect

Starting with the 2.5.17 version, a new convention has been defined to directly integrate a global output effect inside the DSP source code itself. The effect has simply to be declared in a `effect = effect_code;` line in the source. Here is a more complete source code example:

```
import("stdfaust.lib");
process = pm.clarinet_ui_MIDI <: _,_;
effect = dm.freeverb_demo;
```

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

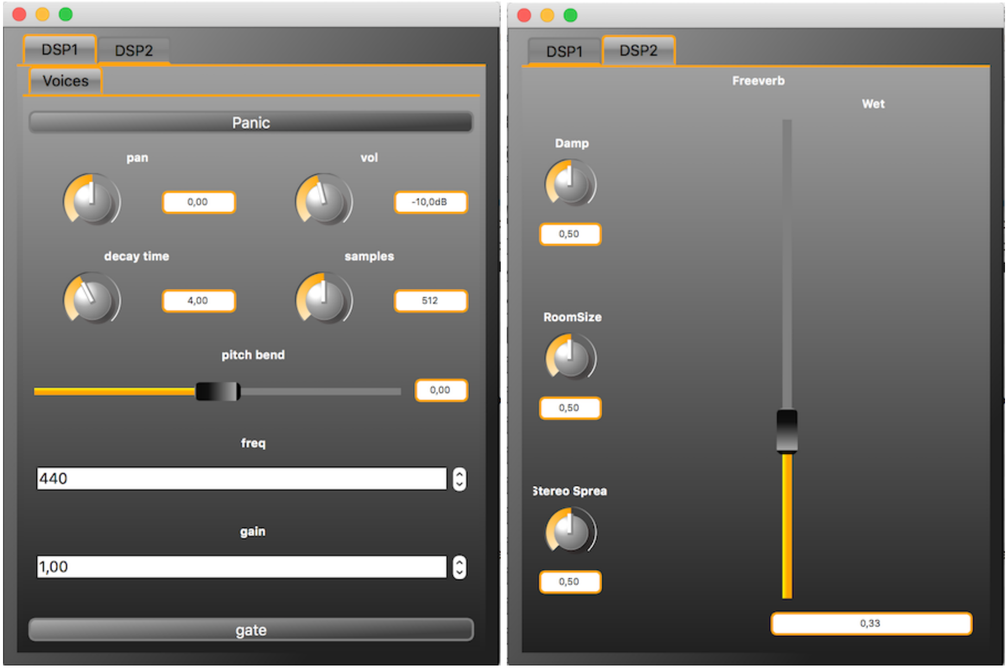


Figure 10.2: Polyphonic instrument with output effect GUI interface: left tab window shows the polyphonic instrument with its *Voices* group only, right tab window shows the output effect.

The architecture script then separates the instrument description itself (the `process = ...` definition) from the effect definition (the `effect = ...` definition), possibly adapts the instrument number of outputs to the effect number of inputs, compiles each part separately, and combines them with the `dsp_sequencer` object.

A new `auto` parameter to be used in *faust2xx* script has been defined, as in the `faustcaqt inst.dsp -effect auto` line for example.

10.5.2 Integrated global output effect and libfaust

For developers using the libfaust library, an helper file named `faust/dsp/poly-dsp-tools.h` is available. It defines an API to automatically create a polyphonic instrument with an output effect, starting from a DSP source file using the effect `effect = ...` convention. The function `createPolyDSPFactoryFromString` or `createPolyDSPFactoryFromFile` must be used to create the polyphonic DSP factory. Next, the `createPolyDSPInstance` function creates the polyphonic object (a subclass of `dsp_poly` type) to be used like a regular `dsp` type object.

After the DSP factory has been compiled, your application or plugin may want to save/restore it in order to save Faust to LLVM IR compilation or even JIT compilation time at next use. To get the internal factory compiled code, several functions are available:

- `writePolyDSPFactoryToIRFile` allows to save the polyphonic factory LLVM IR (in textual format) in a file,
- `writePolyDSPFactoryToBitcodeFile` allows to save the polyphonic factory LLVM IR (in binary format) in a file,
- `writePolyDSPFactoryToMachineFile` allows to save the polyphonic factory executable machine code in a file.

To re-create a DSP factory from a previously saved code, several functions are available:

- `readPolyDSPFactoryFromIRFile` allows to create a polyphonic DSP factory from a file containing the LLVM IR (in textual format),
- `readPolyDSPFactoryFromBitcodeFile` allows to create a polyphonic factory from a file containing the LLVM IR (in binary format),
- `readPolyDSPFactoryFromMachineFile` allows to create a polyphonic DSP factory from a file containing the executable machine code.

Chapter 11

Controlling the code generation

Several options of the Faust compiler allow to control the generated C++ code. By default the computations are done sample by sample in a single loop. But the compiler can also generate *vector* and *parallel* code.

11.1 Vector code generation

Modern C++ compilers are able to do autovectorization, that is to use SIMD instructions to speedup the code. These instructions can typically operate in parallel on short vectors of 4 simple precision floating point numbers thus leading to a theoretical speedup of $\times 4$. Autovectorization of C/C++ programs is a difficult task. Current compilers are very sensitive to the way the code is arranged. In particular too complex loops can prevent autovectorization. The goal of the vector code generation is to rearrange the C++ code in a way that facilitates the autovectorization job of the C++ compiler. Instead of generating a single sample computation loop, it splits the computation into several simpler loops that communicates by vectors.

The vector code generation is activated by passing the `--vectorize` (or `-vec`) option to the Faust compiler. Two additional options are available: `--vec-size <n>` controls the size of the vector (by default 32 samples) and `--loop-variant 0/1` gives some additional control on the loops: `--loop-variant 0` generates fixed-size sub-loops with a final sub-loop that processes the last samples, `--loop-variant 1` generates sub-loops of variable vector size.

To illustrate the difference between scalar code and vector code, let's take the computation of the RMS (Root Mean Square) value of a signal. Here is the Faust code that computes the Root Mean Square of a sliding window of 1000 samples:

```
// Root Mean Square of n consecutive samples
RMS(n) = square : mean(n) : sqrt;

// Square of a signal
square(x) = x * x;

// Mean of n consecutive samples of a signal
// (uses fixpoint to avoid the accumulation of
// rounding errors)
mean(n) = float2fix : integrate(n) :
          fix2float : /(n);

// Sliding sum of n consecutive samples
integrate(n,x) = x - x@n : +~_;

// Convertion between float and fix point
float2fix(x) = int(x*(1<<20));
fix2float(x) = float(x)/(1<<20);

// Root Mean Square of 1000 consecutive samples
process = RMS(1000);
```

The compute() method generated in scalar mode is the following:

```
virtual void compute (int count,
                      float** input,
                      float** output)
{
  float* input0 = input[0];
  float* output0 = output[0];
  for (int i=0; i<count; i++) {
    float fTemp0 = input0[i];
    int iTemp1 = int(1048576*fTemp0*fTemp0);
    iVec0[IOTA&1023] = iTemp1;
    iRec0[0] = ((iVec0[IOTA&1023] + iRec0[1])
                - iVec0[(IOTA-1000)&1023]);
    output0[i] = sqrtf(9.536744e-10f *
                       float(iRec0[0]));
    // post processing
    iRec0[1] = iRec0[0];
    IOTA = IOTA+1;
  }
}
```

The `-vec` option leads to the following reorganization of the code:

```
virtual void compute (int fullcount,
                      float** input,
                      float** output)
{
  int iRec0_tmp[32+4];
  int* iRec0 = &iRec0_tmp[4];
  for (int index=0; index<fullcount; index+=32)
  {
    int count = min (32, fullcount-index);
    float* input0 = &input[0][index];
    float* output0 = &output[0][index];
    for (int i=0; i<4; i++)
      iRec0_tmp[i]=iRec0_perm[i];
    // SECTION : 1
    for (int i=0; i<count; i++) {
      iYec0[(iYec0_idx+i)&2047] =
               int(1048576*input0[i]*input0[i]);
    }
    // SECTION : 2
    for (int i=0; i<count; i++) {
      iRec0[i] = ((iYec0[i] + iRec0[i-1]) -
               iYec0[(iYec0_idx+i-1000)&2047]);
    }
    // SECTION : 3
    for (int i=0; i<count; i++) {
      output0[i] = sqrtf((9.536744e-10f *
                 float(iRec0[i])));
    }
    // SECTION : 4
    iYec0_idx = (iYec0_idx+count)&2047;
    for (int i=0; i<4; i++)
      iRec0_perm[i]=iRec0_tmp[count+i];
  }
}
```

While the second version of the code is more complex, it turns out to be much easier to vectorize efficiently by the C++ compiler. Using Intel icc 11.0, with the exact same compilation options: `-O3 -xHost -ftz -fno-alias -fp-model fast=2`, the scalar version leads to a throughput performance of 129.144 MB/s, while the vector version achieves 359.548 MB/s, a speedup of x2.8 !

The vector code generation is built on top of the scalar code generation (see figure 11.1). Every time an expression needs to be compiled, the compiler checks if it requires a separate loop or not. It applies some simple rules for that. Expressions that are shared

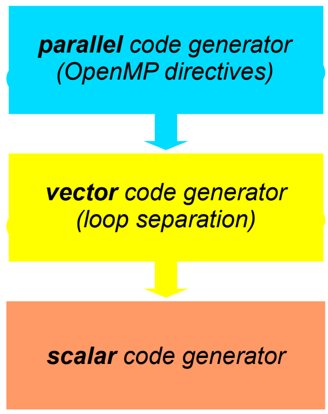


Figure 11.1: FAUST's stack of code generators

(and are complex enough) are good candidates to be compiled in a separate loop, as well as recursive expressions and expressions used in delay lines.

The result is a directed graph in which each node is a computation loop (see Figure 11.2). This graph is stored in the klass object and a topological sort is applied to it before printing the code.

11.2 Parallel code generation

The parallel code generation is activated by passing either the `--openMP` (or `-omp`) option or the `--scheduler` (or `-sch`) option. It implies the `-vec` options as the parallel code generation is built on top of the vector code generation.

11.2.1 The OpenMP code generator

The `--openMP` (or `-omp`) option given to the FAUST compiler will insert appropriate OpenMP directives in the C++ code. OpenMP (http://wwww.openmp.org) is a well established API that is used to explicitly define direct multi-threaded, shared memory parallelism. It is based on a fork-join model of parallelism (see figure 11.3). Parallel regions are delimited by `#pragma omp parallel` constructs. At the entrance of a parallel region a team of parallel threads is activated. The code within a parallel region is executed by each thread of the parallel team until the end of the region.

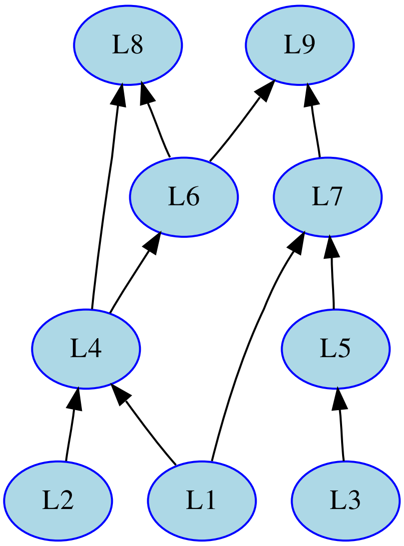


Figure 11.2: The result of the -vec option is a directed acyclic graph (DAG) of small computation loops

```
#pragma omp parallel
{
  // the code here is executed simultaneously by
  // every thread of the parallel team
  ...
}
```

In order not to have every thread doing redundantly the exact same work, OpemMP provides specific *work-sharing* directives. For example `#pragma omp sections` allows to break the work into separate, discrete sections, each section being executed by one thread:

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {
      // job 1
    }
```

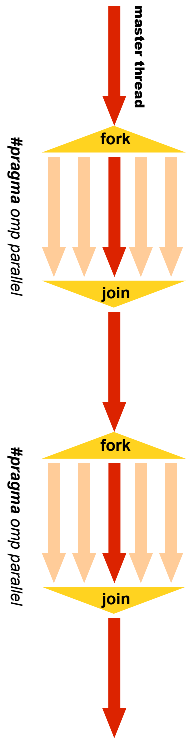


Figure 11.3: OpenMP is based on a fork-join model

```
    #pragma omp section
    {
      // job 2
    }
    ...
  }

  ...
}
```

11.2.2 Adding OpenMP directives

As said before the parallel code generation is built on top of the vector code generation. The graph of loops produced by the vector code generator is topologically sorted in order to detect the loops that can be computed in parallel. The first set S_0 (loops $L1$, $L2$ and $L3$ in the DAG of Figure 11.2) contains the loops that don't depend on any other loops, the set S_1 contains the loops that only depend on loops of S_0, (that is loops $L4$ and $L5$), etc..

As all the loops of a given set S_n can be computed in parallel, the compiler will generate a `sections` construct with a `section` for each loop.

```
#pragma omp sections
{
  #pragma omp section
  for (...) {
    // Loop 1
  }
  #pragma omp section
  for (...) {
    // Loop 2
  }
  ...
}
```

If a given set contains only one loop, then the compiler checks to see if the loop can be parallelized (no recursive dependencies) or not. If it can be parallelized, it generates:

```
#pragma omp for
for (...) {
 // Loop code
}
```

otherwise it generates a `single` construct so that only one thread will execute the loop:

```
#pragma omp single
for (...) {
 // Loop code
}
```

11.2.3 Example of parallel OpenMP code

To illustrate how Faust uses the OpenMP directives, here is a very simple example, two 1-pole filters in parallel connected to an adder (see figure 11.4 the corresponding block-diagram):

```
filter(c) = *(1-c) : + ~ *(c);
process = filter(0.9), filter(0.9) : +;
```

The corresponding compute() method obtained using the `-omp` option is the following:

```
virtual void compute (int fullcount,
                      float** input,
```

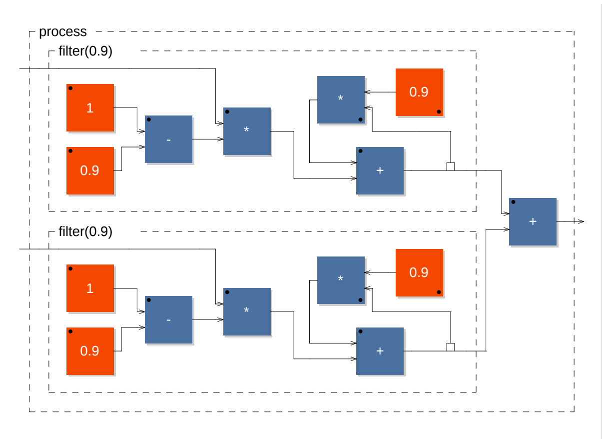


Figure 11.4: two filters in parallel connected to an adder

```
                      float** output)
{
  float   fRec0_tmp[32+4];
  float   fRec1_tmp[32+4];
  float*  fRec0 = &fRec0_tmp[4];
  float*  fRec1 = &fRec1_tmp[4];
  #pragma omp parallel firstprivate(fRec0,fRec1)
  {
    for (int index = 0; index < fullcount;
                                index += 32)
    {
      int count = min (32, fullcount-index);
      float* input0 = &input[0][index];
      float* input1 = &input[1][index];
      float* output0 = &output[0][index];
      #pragma omp single
      {
        for (int i=0; i<4; i++)
          fRec0_tmp[i]=fRec0_perm[i];
        for (int i=0; i<4; i++)
          fRec1_tmp[i]=fRec1_perm[i];
      }
      // SECTION : 1
      #pragma omp sections
      {
        #pragma omp section
        for (int i=0; i<count; i++) {
```

```
          fRec0[i] = ((0.1f * input1[i])
                      + (0.9f * fRec0[i-1]));
        }
        #pragma omp section
        for (int i=0; i<count; i++) {
          fRec1[i] = ((0.1f * input0[i])
                      + (0.9f * fRec1[i-1]));
        }
      }
      // SECTION : 2
      #pragma omp for
      for (int i=0; i<count; i++) {
        output0[i] = (fRec1[i] + fRec0[i]);
      }
      // SECTION : 3
      #pragma omp single
      {
        for (int i=0; i<4; i++)
          fRec0_perm[i]=fRec0_tmp[count+i];
        for (int i=0; i<4; i++)
          fRec1_perm[i]=fRec1_tmp[count+i];
      }
    }
  }
}
```

This code requires some comments:

1. The parallel construct `#pragma omp parallel` is the fundamental construct that starts parallel execution. The number of parallel threads is generally the number of CPU cores but it can be controlled in several ways.

2. Variables external to the parallel region are shared by default. The pragma `firstprivate (fRec0,fRec1)` indicates that each thread should have its private copy of fReco and fRec1. The reason is that accessing shared variables requires an indirection and is quite inefficient compared to private copies.

3. The top level loop `for (int index = 0;...)...` is executed by all threads simultaneously. The subsequent work-sharing directives inside the loop will indicate how the work must be shared between the threads.

4. Please note that an implied barrier exists at the end of each work-sharing region. All threads must have executed the barrier before any of them can continue.

5. The work-sharing directive `#pragma omp single` indicates that this first section will be executed by only one thread (any of them).

6. The work-sharing directive `#pragma omp sections` indicates that each corresponding `#pragma omp section`, here our two filters, will be executed in parallel.

7. The loop construct `#pragma omp for` specifies that the iterations of the associated loop will be executed in parallel. The iterations of the loop are distributed across the parallel threads. For example, if we have two threads, the first one can compute indices between 0 and count/2 and the other one between count/2 and count.

8. Finally `#pragma omp single` in section 3 indicates that this last section will be executed by only one thread (any of them).

11.2.4 The scheduler code generator

With the `--scheduler` (or `-sch`) option given to the Faust compiler, the computation graph is cut into separated computation loops (called "tasks"), and a "Work Stealing Scheduler" is used to activate and execute them following their dependencies. A pool of worked threads is created and each thread uses it's own local WSQ (Work Stealing Queue) of tasks. A WSQ is a special queue with a Push operation, a "private" LIFO Pop operation and a "public" FIFO Pop operation.

Starting from a ready task, each thread follows the dependencies, possibly pushing ready sub-tasks into it's own local WSQ. When no more tasks can be activated on a given computation path, the thread pops a task from it's local WSQ. If the WSQ is empty, then the thread is allowed to "steal" tasks from other threads WSQ.

The local LIFO Pop operation allows better cache locality and the FIFO steal Pop "larger chuck" of work to be done. The reason for this is that many work stealing workloads are divide-and-conquer in nature, stealing one of the oldest task implicitly also steals a (potentially) large subtree of computations that will unfold once that piece of work is stolen and run.

Compared to the OpenMP model (`-omp`) the new model is worse for simple Faust programs and usually starts to behave comparable or sometimes better for "complex enough" Faust programs. In any case, since OpenMP does not behave so well with GCC compilers (only quite recent versions like GCC 4.4 start to show some improvements), and is unusable on OSX in real-time contexts, this new scheduler option has it's own value. We plan to improve it adding a "pipelining" idea in the future.

11.2.5 Example of parallel scheduler code

To illustrate how FAUST generates the scheduler code, here is a very simple example, two 1-pole filters in parallel connected to an adder (see figure 11.4 the corresponding block-diagram):

```
filter(c) = *(1-c) : + ~ *(c);
process = filter(0.9), filter(0.9) : +;
```

When `-sch` option is used, the content of the additional *architecture/scheduler.h* file is inserted in the generated code. It contains code to deal with WSQ and thread management. The `compute()` and `computeThread()` methods are the following:

```
virtual void compute (int fullcount,
                      float** input,
                      float** output)
{
    GetRealTime();
    this->input = input;
    this->output = output;
    StartMeasure();
    for (fIndex = 0; fIndex < fullcount; fIndex += 32) {
        fFullCount = min (32, fullcount-fIndex);
        TaskQueue::Init();
        // Initialize end task
        fGraph.InitTask(1,1);
        // Only initialize tasks with inputs
        fGraph.InitTask(4,2);
        fIsFinished = false;
        fThreadPool.SignalAll(fDynamicNumThreads - 1);
        computeThread(0);
        while (!fThreadPool.IsFinished()) {}
    }
    StopMeasure(fStaticNumThreads,
        fDynamicNumThreads);
}
void computeThread (int cur_thread) {
    float*  fRec0 = &fRec0_tmp[4];
    float*  fRec1 = &fRec1_tmp[4];
    // Init graph state
    {
        TaskQueue taskqueue;
        int tasknum = -1;
        int count = fFullCount;
```

```
        // Init input and output
        FAUSTFLOAT* input0 = &input[0][fIndex];
        FAUSTFLOAT* input1 = &input[1][fIndex];
        FAUSTFLOAT* output0 = &output[0][fIndex];
        int task_list_size = 2;
        int task_list[2] = {2,3};
        taskqueue.InitTaskList(task_list_size, task_list,
            fDynamicNumThreads, cur_thread, tasknum);
        while (!fIsFinished) {
            switch (tasknum) {
                case WORK_STEALING_INDEX: {
                    tasknum = TaskQueue::GetNextTask(
                       cur_thread);
                    break;
                }
                case LAST_TASK_INDEX: {
                    fIsFinished = true;
                    break;
                }
                // SECTION : 1
                case 2: {
                    // LOOP 0x101111680
                    // pre processing
                    for (int i=0; i<4; i++) fRec0_tmp[i]=
                       fRec0_perm[i];
                    // exec code
                    for (int i=0; i<count; i++) {
                        fRec0[i] = ((1.000000e-01f * (
                           float)input1[i]) + (0.9f *
                           fRec0[i-1]));
                    }
                    // post processing
                    for (int i=0; i<4; i++) fRec0_perm[i
                       ]=fRec0_tmp[count+i];

                    fGraph.ActivateOneOutputTask(
                       taskqueue, 4, tasknum);
                    break;
                }
                case 3: {
                    // LOOP 0x1011125e0
                    // pre processing
                    for (int i=0; i<4; i++) fRec1_tmp[i]=
```

```
                       fRec1_perm[i];
                    // exec code
                    for (int i=0; i<count; i++) {
                        fRec1[i] = ((1.000000e-01f * (
                           float)input0[i]) + (0.9f *
                           fRec1[i-1]));
                    }
                    // post processing
                    for (int i=0; i<4; i++) fRec1_perm[i
                       ]=fRec1_tmp[count+i];

                    fGraph.ActivateOneOutputTask(
                       taskqueue, 4, tasknum);
                    break;
                }
                case 4: {
                    // LOOP 0x101111580
                    // exec code
                    for (int i=0; i<count; i++) {
                        output0[i] = (FAUSTFLOAT)(fRec1[i
                           ] + fRec0[i]);
                    }

                    tasknum = LAST_TASK_INDEX;
                    break;
                }
            }
        }
    }
}
```

Chapter 12

Mathematical Documentation

The FAUST compiler provides a mechanism to produce a self-describing documentation of the mathematical semantic of a FAUST program, essentially as a pdf file. The corresponding options are `-mdoc` (short) or `--mathdoc` (long).

12.1 Goals of the mathdoc

There are for main goals, or uses, of this mathematical documentation:

1. to preserve the DSP source code with all the needed libraries, so that the DSP can be compiled with a more recent version of the compiler and produce the same resulting program. This is the way we allow the libraries themselves to evolve (even without maintaining compatibility with older versions), while still allowing an older program to be compiled with newer versions of the compiler;

2. to preserve signal processors, independently from any computer language but only under a mathematical form;

3. to bring some help for debugging tasks, by showing the formulas as they are really computed after the compilation stage;

4. to give a new teaching support, as a bridge between code and formulas for signal processing.

12.2 Installation requirements

- `faust`, of course!

- `svg2pdf` (from the Cairo 2D graphics library), to convert block-diagrams, as LaTeX doesn't eat SVG directly yet...
- `breqn`, a LaTeX package to handle automatic breaking of long equations,
- `pdflatex`, to compile the LaTeX output file.

12.3 Generating the mathdoc

The easiest way to generate the complete mathematical documentation is to call the `faust2mathdoc` script on a FAUST file, as the `-mdoc` option leave the documentation production unfinished. For example:

```
faust2mathdoc noise.dsp
```

12.3.1 Invoking the -mdoc option

Calling directly `faust -mdoc` does only the first part of the work, generating:

- a top-level directory, suffixed with "`-mdoc`",
- 5 subdirectories (`cpp/`, `pdf/`, `src/`, `svg/`, `tex/`),
- a LaTeX file containing the formulas,
- SVG files for block-diagrams.

At this stage:

- `cpp/` remains empty,
- `pdf/` remains empty,
- `src/` contains all the used FAUST sources (so all needed libraries to have a self-contained DSP),
- `svg/` contains SVG block-diagram files,
- `tex/` contains the generated LaTeX file.

12.3.2 Invoking faust2mathdoc

The `faust2mathdoc` script calls `faust --mathdoc` first, then it finishes the work:

- moving the output C++ file into `cpp/`,
- converting all SVG files into pdf files (you must have `svg2pdf` installed, from the Cairo 2D graphics library),
- launching `pdflatex` on the LaTeX file (you must have both `pdflatex` and the `breqn` package installed),
- moving the resulting pdf file into `pdf/`.

12.3.3 Online examples

To get an idea of the results of this mathematical documentation, which captures the mathematical semantic of FAUST programs, you can look at two pdf files online:

- http://faust.grame.fr/pdf/karplus.pdf (automatic documentation),
- http://faust.grame.fr/pdf/noise.pdf (manual documentation).

You can also generate all *mdoc* pdfs at once, simply invoking the `make mathdoc` command inside the `examples/` directory:

- for each `%.dsp` file, a complete `%-mdoc` directory will be generated,
- a single `allmathpdfs/` directory will gather all the generated pdf files.

12.4 Automatic documentation

By default, when no `<mdoc>` tag can be found in the input FAUST file, the `-mdoc` option automatically generates a LaTeX file with four sections:

1. "**Equations of process**", gathering all formulas needed for `process`,
2. "**Block-diagram schema of process**", showing the top-level block-diagram of `process`,
3. "**Notice of this documentation**", summing up generation and conventions information,
4. "**Complete listing of the input code**", listing all needed input files (including libraries).

12.5 Manual documentation

You can specify yourself the documentation instead of using the automatic mode, with five xml-like tags. That permits you to modify the presentation and to add your own comments, not only on `process`, but also about any expression you'd like to. Note that as soon as you declare an `<mdoc>` tag inside your FAUST file, the default structure of the automatic mode is ignored, and all the LaTeX stuff becomes up to you!

12.5.1 Six tags

Here are the six specific tags:

- `<mdoc></mdoc>`, to open a documentation field in the FAUST code,
  - `<equation></equation>`, to get equations of a FAUST expression,
  - `<diagram></diagram>`, to get the top-level block-diagram of a FAUST expression,
  - `<metadata></metadata>`, to reference FAUST metadatas (cf. declarations), calling the corresponding keyword,
  - `<notice />`, to insert the "adaptive" notice all formulas actually printed,
  - `<listing [attributes] />`, to insert the listing of FAUST files called.

The `<listing />` tag can have up to three boolean attributes (set to `"true"` by default):

- `mdoctags` for `<mdoc>` tags;
- `dependencies` for other files dependencies;
- `distributed` for the distribution of interleaved FAUST code between `<mdoc>` sections.

12.5.2 The mdoc top-level tags

The `<mdoc></mdoc>` tags are the top-level delimiters for FAUST mathematical documentation sections. This means that the four other documentation tags can't be used outside these pairs (see section **??**).

In addition of the four inner tags, `<mdoc></mdoc>` tags accept free LaTeX text, including its standard macros (like `\section`, `\emph`, etc.). This allows to manage the presentation of resulting tex file directly from within the input FAUST file.

The complete list of the LaTeX packages included by FAUST can be found in the file `architecture/latexheader.tex`.

12.5.3 An example of manual mathdoc

```
<mdoc>
\title{<metadata>name</metadata>}
\author{<metadata>author</metadata>}
\date{\today}
\maketitle

\begin{tabular}{ll}
    \hline
    \textbf{name}       & <metadata>name</metadata> \\
    \textbf{version}    & <metadata>version</metadata> \\
    \textbf{author}     & <metadata>author</metadata> \\
    \textbf{license}    & <metadata>license</metadata> \\
    \textbf{copyright}  & <metadata>copyright</metadata>
       \\
    \hline
\end{tabular}
\bigskip
</mdoc>
//
   -----------------------------------------------------------------

// Noise generator and demo file for the Faust math
   documentation
//
   -----------------------------------------------------------------


declare name        "Noise";
declare version     "1.1";
declare author      "Grame";
declare author      "Yghe";
declare license     "BSD";
declare copyright   "(c)GRAME 2009";

<mdoc>
\section{Presentation of the "noise.dsp" Faust program}
This program describes a white noise generator with an
   interactive volume, using a random function.

\subsection{The random function}
</mdoc>
```

```
random  = +(12345)~*(1103515245);

<mdoc>
The \texttt{random} function describes a generator of
   random numbers, which equation follows. You should
   notice hereby the use of an integer arithmetic on 32
   bits, relying on integer wrapping for big numbers.
<equation>random</equation>

\subsection{The noise function}
</mdoc>

noise   = random/2147483647.0;

<mdoc>
The white noise then corresponds to:
<equation>noise</equation>

\subsection{Just add a user interface element to play
   volume!}
</mdoc>

process = noise * vslider("Volume[style:knob]", 0, 0, 1,
   0.1);

<mdoc>
Endly, the sound level of this program is controlled by a
    user slider, which gives the following equation:
<equation>process</equation>

\section{Block-diagram schema of process}
This process is illustrated on figure 1.
<diagram>process</diagram>

\section{Notice of this documentation}
You might be careful of certain information and naming
   conventions used in this documentation:
<notice />

\section{Listing of the input code}
The following listing shows the input Faust code, parsed
   to compile this mathematical documentation.
```

```
<listing mdoctags="false" dependencies="false"
   distributed="true" />
</mdoc>
```

The following page which gathers the four resulting pages of `noise.pdf` in small size. might give you an idea of the produced documentation.

12.5.4 The -stripmdoc option

As you can see on the resulting file `noisemetadata.pdf` on its pages 3 and 4, the listing of the input code (section 4) contains all the mathdoc text (here colored in grey). As it may be useless in certain cases (see Goals, section 12.1), we provide an option to strip mathdoc contents directly at compilation stage: `-stripmdoc` (short) or `--strip-mdoc-tags` (long).

12.6 Localization of mathdoc files

By default, texts used by the documentator are in English, but you can specify another language (French, German and Italian for the moment), using the `-mdlang` (or `--mathdoc-lang`) option with a two-letters argument (`en`, `fr`, `it`, etc.).

The `faust2mathdoc` script also supports this option, plus a third short form with `-l`:

```
faust2mathdoc -l fr myfaustfile.dsp
```

If you would like to contribute to the localization effort, feel free to translate the mathdoc texts from any of the `mathdoctexts-*.txt` files, that are in the `architecture` directory (`mathdoctexts-fr.txt`, `mathdoctexts-it.txt`, etc.). As these files are dynamically loaded, just adding a new file with an appropriate name should work.

Noise

Grame, Yghe

March 9, 2010

name	Noise
version	1.1
author	Grame, Yghe
license	BSD
copyright	(c)GRAME 2009

```
//-----------------------------------------------------------------
// Noise generator and demo file for the Faust math documentation
//-----------------------------------------------------------------

declare name      "Noise";
declare version   "1.1";
declare author    "Grame";
declare author    "Yghe";
declare license   "BSD";
declare copyright  "(c)GRAME 2009";
```

1 Presentation of the "noise.dsp" Faust program

This program describes a white noise generator with an interactive volume, using a random function.

1.1 The random function

```
random = +(int(12345))~*(int(1103515245));
```

The `random` function describes a generator of random numbers, which equation follows. You should notice hereby the use of an integer arithmetic on 32 bits, relying on integer wrapping for big numbers.

1. Output signal y such that

$$y(t) = r_1(t)$$

2. Input signal (none)

3. Intermediate signal r_1 such that

$$r_1(t) = 12345 \oplus 1103515245 \odot r_1(t-1)$$

1.2 The noise function

```
noise  = (int(random))/(int(random+1));
```

The white noise then corresponds to:

1. Output signal y such that

$$y(t) = s_1(t)$$

2. Input signal (none)

3. Intermediate signal s_1 such that

$$s_1(t) = \mathrm{int}\,(r_1(t)) \oslash \mathrm{int}\,(1 \oplus r_1(t))$$

1.3 Just add a user interface element to play volume!

```
process = noise * vslider("Volume[style:knob]", 0, 0, 1, 0.1);
```

Endly, the sound level of this program is controlled by a user slider, which gives the following equation:

1. Output signal y such that

$$y(t) = u_{s1}(t) \cdot s_1(t)$$

2. Input signal (none)

3. User-interface input signal u_{s1} such that

$$\text{"Volume"} \quad u_{s1}(t) \in [\,0, 1\,] \quad (\text{default value} = 0)$$

2 Block-diagram schema of process

This process is illustrated on figure 1.

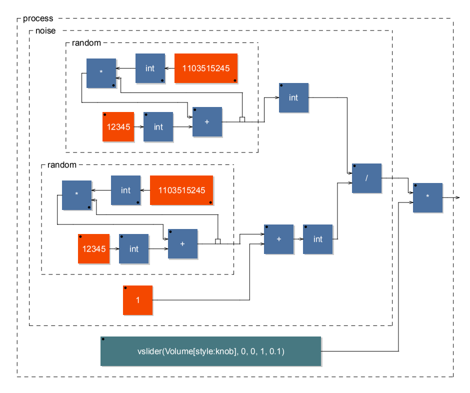


Figure 1: Block diagram of `process`

3 Notice of this documentation

You might be careful of certain information and naming conventions used in this documentation:

- This document was generated using Faust version 0.9.13 on March 09, 2010.
- The value of a Faust program is the result of applying the signal transformer denoted by the expression to which the `process` identifier is bound to input signals, running at the f_S sampling frequency.
- Faust (*Functional Audio Stream*) is a functional programming language designed for synchronous real-time signal processing and synthesis applications. A Faust program is a set of bindings of identifiers to expressions that denote signal transformers. A signal s in S is a function mapping[1] times $t \in \mathbb{Z}$ to values $s(t) \in \mathbb{R}$, while a signal transformer is a function

[1] Faust assumes that $\forall\, s \in S, \forall\, t \in \mathbb{Z}, s(t) = 0$ when $t < 0$.

from S^n to S^m, where $n, m \in \mathbb{N}$. See the Faust manual for additional information (http://faust.grame.fr).

- Every mathematical formula derived from a Faust expression is assumed, in this document, to having been normalized (in an implementation-dependent manner) by the Faust compiler.
- A block diagram is a graphical representation of the Faust binding of an identifier I to an expression E; each graph is put in a box labeled by I. Subexpressions of E are recursively displayed as long as the whole picture fits in one page.
- $\forall\, x \in \mathbb{R}$,

$$\mathrm{int}(x) = \begin{cases} \lfloor x \rfloor & \text{if } x > 0 \\ \lceil x \rceil & \text{if } x < 0 \\ 0 & \text{if } x = 0 \end{cases}.$$

- This document uses the following integer operations:

operation	*name*	*semantics*
$i \oplus j$	integer addition	$\mathrm{normalize}(i+j)$, in $\mathbb{Z}$
$i \odot j$	integer multiplication	$\mathrm{normalize}(i \cdot j)$, in $\mathbb{Z}$
$i \oslash j$	integer division	$\mathrm{normalize}(\mathrm{int}(i/j))$, in $\mathbb{Q}$

Integer operations in Faust are inspired by the semantics of operations on the n-bit two's complement representation of integer numbers; they are internal composition laws on the subset $[\,-2^{n-1}, 2^{n-1}-1\,]$ of $\mathbb{Z}$, with $n = 32$. For any integer binary operation $\times$ on $\mathbb{Z}$, the $\otimes$ operation is defined as: $i \otimes j = \mathrm{normalize}(i \times j)$, with

$$\mathrm{normalize}(i) = i - N \cdot \mathrm{sign}(i) \cdot \left\lfloor \frac{|i| + N/2 + (\mathrm{sign}(i)-1)/2}{N} \right\rfloor,$$

where $N = 2^n$ and $\mathrm{sign}(i) = 0$ if $i = 0$ and $i/|i|$ otherwise. Unary integer operations are defined likewise.

- The `noisemetadata-mdoc/` directory may also include the following subdirectories:
  - `cpp/` for Faust compiled code;
  - `pdf/` which contains this document;
  - `src/` for all Faust sources used (even libraries);
  - `svg/` for block diagrams, encoded using the Scalable Vector Graphics format (http://www.w3.org/Graphics/SVG/);
  - `tex/` for the LaTeX source of this document.

4 Listing of the input code

The following listing shows the input Faust code, parsed to compile this mathematical documentation.

Listing 1: `noisemetadata.dsp`

```
//-----------------------------------------------------------------
// Noise generator and demo file for the Faust math documentation
//-----------------------------------------------------------------

declare name      "Noise";
declare version   "1.1";
declare author    "Grame";
declare author    "Yghe";
declare license   "BSD";
declare copyright  "(c)GRAME 2009";


random = +(int(12345))~*(int(1103515245));


noise  = (int(random))/(int(random+1));


process = noise * vslider("Volume[style:knob]", 0, 0, 1, 0.1);
```

12.7 Summary of the mathdoc generation steps

1. First, to get the full mathematical documentation done on your faust file, call `faust2mathdoc myfaustfile.dsp`.
2. Then, open the pdf file `myfaustfile-mdoc/pdf/myfaustfile.pdf`.
3. That's all !

Chapter 13

Acknowledgments

Many persons are contributing to the FAUST project, by providing code for the compiler, architecture files, libraries, examples, documentation, scripts, bug reports, ideas, etc. We would like in particular to thank:

- Fons Adriaensen
- Karim Barkati
- Jérôme Barthélemy
- Tim Blechmann
- Tiziano Bole
- Alain Bonardi
- Thomas Charbonnel
- Raffaele Ciavarella
- Julien Colafrancesco
- Damien Cramet
- Sarah Denoux
- Étienne Gaudrin
- Olivier Guillerminet
- Pierre Guillot
- Albert Gräf
- Pierre Jouvelot
- Stefan Kersten

- Victor Lazzarini
- Matthieu Leberre
- Mathieu Leroi
- Fernando Lopez-Lezcano
- Kjetil Matheussen
- Hermann Meyer
- Romain Michon
- Rémy Muller
- Eliott Paris
- Reza Payami
- Laurent Pottier
- Sampo Savolainen
- Nicolas Scaringella
- Anne Sedes
- Priyanka Shekar
- Stephen Sinclair
- Travis Skare
- Julius Smith
- Mike Solomon
- Michael Wilson

as well as our colleagues at GRAME:

- Dominique Fober
- Christophe Lebreton
- Stéphane Letz
- Romain Michon

We would like also to thank for their financial support:

- the French Ministry of Culture
- the Rhône-Alpes Region
- the City of Lyon
- the French National Research Agency (ANR)