Contributing

In general, libraries are organised in a *stacked manner*: the base ones define functions or constants without any dependancies, and additional ones are gradually built on top of simpler ones, layer by layer. **Dependency loops must be avoided as much as possible**. The *resources* folder contains tools to build and visualise the libraries dependencies graphs.

If you wish to add a function to any of these libraries or if you plan to add a new library, make sure that you observe the following conventions:

New Functions

- All functions must be preceded by a markdown documentation header respecting the following format (open the source code of any of the libraries for an example):

```
//-----------------functionName--------------------
// Description
//
// #### Usage
//
// ```
// Usage example
// ```
//
// Where:
//
// * argument1: argument 1 description
// * argument2: argument 2 description
//
// #### Example
//
// ```
// Additional example
// ```
//
// #### Test
// ```
// functionName_test = some_dsp_code;
// ```
//
// #### References
//
// * <https://some_url1>
// * <https://some_url2>
//-------------------------------------------------
```

- Every time a new function is added, the documentation should be updated simply by running `make doclib`.
- The environment system (e.g. `os.osc`) should be used when calling a function declared in another library (see the section on Library Import).
- Try to reuse existing functions as much as possible.
- The `Usage` line must show the *input/output shape* (the number of inputs and outputs) of the function, like `gen: _` for a mono generator, `_ : filter : _` for a mono effect, etc.
- The `Example` line can be used to provide additional examples.
- The `Test` line can be used to add a DSP program to test the function. The test name must be `functionName_test`. The actual code can be extracted and independantly tested using the `-pn` compiler option (to specify the name of the dsp entry-point instead of process). The test code must import all the needed libraries, like `an = library("analyzers.lib");` if a function from analyzers.lib is used in the test code.
- The `References` line can be used to add links to references
- Some functions use parameters that are constant numerical expressions. The convention is to label them in *capital letters* and document them preferably to be *constant numerical expressions* (or *known at compile time* in existing libraries).
- Functions with several parameters should better be written by putting the *more constant parameters* (like control, setup...) at the beginning of the parameter list, and *audio signals to be processed* at the end. This allows to do partial-application. So prefer the following `clip(low, high, x) = min(max(x, low), high);` form where `clip(-1, 1)` partially applied version can be used later on in different contexts, better than `clip(x, low, high) = min(max(x, low), high);` version.

Layering UI-ready variants

Many functions benefit from two public faces so the same DSP can serve both low-level reuse and ready-to-tweak usage:

- *Core function*: exposes all parameters, no UI or side effects; best for reuse, composition, and testing.
- *UI wrapper*: fixes sensible defaults and exposes only runtime-tuned parameters as UI controls; leaves signals that must be provided externally as arguments.

Use the UI-free core for correctness and performance work; build the UI variant when you need something directly tweakable in examples or end-user contexts.

A generic core/UI pair could be:

```
// Core: parameters explicit, no UI
coreEffect(paramA, paramB, mix) =
  fooProcessing(mix, wet)
with {
  wet = *(paramA) : barProcessing(paramB); // barProcessing is your DSP
};

// UI wrapper: binds smoothed controls to the core
coreEffect_ui =
  coreEffect(paramA_ui, paramB_ui, mix_ui)
with {
  paramA_ui = hslider("Param A", 1.0, 0.0, 2.0, 0.01) : si.smoo;
  paramB_ui = hslider("Param B", 0.5, 0.0, 1.0, 0.01) : si.smoo;
  mix_ui    = hslider("Mix", 1.0, 0.0, 1.0, 0.01) : si.smoo;
};

process = coreEffect_ui;
```

This keeps the core reusable (no UI dependencies) and the wrapper ready for immediate tweaking; `process` points to the UI layer for quick testing.

Instrument-specific three-layer pattern

Instrument models often add a third, ready-to-play layer. The clarinet model is a reference:

- `pm.clarinetModel(tubeLength, pressure, reedStiffness, bellOpening)`: core DSP with every parameter explicit and no UI.
- `pm.clarinetModel_ui(pressure)`: wraps the core and adds UI sliders for tube length, reed stiffness, bell opening, and output gain; keeps `pressure` as an argument.
- `pm.clarinet_ui_MIDI`: builds a playable instrument by pairing the core with a blower/envelope plus MIDI-mapped UI (pitch bend, sustain, vibrato, gain, etc.).

When adding similar models, start with the UI-free core, add a minimal UI wrapper, then optionally provide a controller-specific wrapper (MIDI or otherwise). Keep the core independent so it remains reusable.

Variables and identifiers scoping

To avoid name clashes between libraries, keep identifiers as local as possible. Prefer defining intermediate constants and helpers inside `with { ... }` blocks or `environment { ... }` sections, and only expose the intended public entry points. This minimizes collisions when several libraries are imported together and keeps global namespace usage limited to documented, public-facing functions.

New Libraries

- Any new "standard" library should be declared in `stdfaust.lib` with its own environment (2 letters - see `stdfaust.lib`).
- Any new "standard" library must be added to `generateDoc`.
- Functions must be organized by sections.
- Any new library should at least `declare` a `name` and a `version`.
- Any new library has to use a prefix declared in the header section with the following kind of syntax: `Its official prefix is 'qu'` (look at an existing library to follow the exact syntax).
- Be sure to add the appropriate kind of `ma = library("maths.lib");` import library line, for each external library function used in the new library (for instance `ma.foo` that would be used somewhre in the code).
- The comment based markdown documentation of each library must respect the following format (open the source code of any of the libraries for an example):

```
//################ libraryName ##################
// Description
//
// * Section Name 1
// * Section Name 2
// * ...
//
// It should be used using the `[...]` environment:
//
// ```
// [...] = library("libraryName");
// process = [...].functionCall;
// ```
//
// Another option is to import `stdfaust.lib` which already contains the `[...]`
// environment:
//
// ```
// import("stdfaust.lib");
// process = [...].functionCall;
// ```
//###############################################

//================ Section Name ===============
// Description
//=============================================
```

Coding Conventions

In order to have a uniformized library system, we established the following conventions (that hopefully will be followed by others when making modifications to them).

Function Naming

[WIP]

JOS proposal: using terms used in the field of digital signal processing, as follows:

- `impulse` : ...,0,1,0,...
- `pulse` : ...,0,1,1,0,... or longer
- `impulse_train`
- `pulse_train`
- `gate` = pulse controlled externally (e.g., by NoteOn,NoteOff)
- `trigger` = impulse controlled externally (gate - gate' > 0) == gate rising edge

[/WIP]

Variable Argument List

Strictly speaking, there are no lists in Faust. But list operations can be simulated (in part) using the parallel binary composition operation `,` and pattern matching.

Thus functions expecting a variable number of arguments can use this mechanism, like a `foo` function that would be used this way: `foo((a,b,c,d))`. See fi.iir and fi.fir examples.

Documentation

- All the functions that we want to be "public" are documented.
- We used the `faust2md` "standards" for each library: `//###` for main title (library name - equivalent to `#` in markdown), `//===` for section declarations (equivalent to `##` in markdown) and `//---` for function declarations (equivalent to `####` in markdown - see `basics.lib` for an example).
- Sections in function documentation should be declared as `####` markdown title.
- Each function documentation provides a "Usage" section (see `basics.lib`).

- The full documentation can be generated using the doc/Makefile script. Use `make help` to see all possible commands. If you plan to create a pull-request, *do not commit the full generated code* but only the modified .lib files.
- Each function can have `declare author "name";`, `declare copyright "XXX";` and `declare licence "YYY";` declarations.
- Each library has a `declare version "xx.yy.zz";` semantic version number to be raised each time a modification is done. The global `version` number in `version.lib` also has to be adapted according to the change.

Library Import

To prevent cross-references between libraries, we generalized the use of the `library("")` system for function calls in all the libraries. This means that everytime a function declared in another library is called, the environment corresponding to this library needs to be called too. To make things easier, a `stdfaust.lib` library was created and is imported by all the libraries:

```
aa = library("aanl.lib");
sf = library("all.lib");
an = library("analyzers.lib");
ba = library("basics.lib");
co = library("compressors.lib");
de = library("delays.lib");
dm = library("demos.lib");
dx = library("dx7.lib");
en = library("envelopes.lib");
fd = library("fds.lib");
fi = library("filters.lib");
ho = library("hoa.lib");
it = library("interpolators.lib");
la = library("linearalgebra.lib");
ma = library("maths.lib");
mi = library("mi.lib");
ef = library("misceffects.lib");
os = library("oscillators.lib");
no = library("noises.lib");
pf = library("phaflangers.lib");
pm = library("physmodels.lib");
qu = library("quantizers.lib");
rm = library("reducemaps.lib");
re = library("reverbs.lib");
ro = library("routes.lib");
si = library("signals.lib");
so = library("soundfiles.lib");
sp = library("spats.lib");
sy = library("synths.lib");
ve = library("vaeffects.lib");
vl = library("version.lib");
wa = library("webaudio.lib");
wd = library("wdmodels.lib");
```

For example, if we wanted to use the `smooth` function which is now declared in `signals.lib`, we would do the following:

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

process = si.smooth(0.999);
```

This standard is only used within the libraries: nothing prevents coders to still import `signals.lib` directly and call `smooth` without `ro.`, etc. It means symbols and function names defined within a library **have to be unique to not collide with symbols of any other libraries**.

"Demo" Functions

"Demo" functions are placed in `demos.lib` and have a built-in user interface (UI). Their name ends with the `_demo` suffix. Each of these function have a `.dsp` file associated to them in the `/examples` folder.

Any function containing UI elements should be placed in this library and respect these standards.

"Standard" Functions

"Standard" functions are here to simplify the life of new (or not so new) Faust coders. They are declared in `/libraries/doc/standardFunctions.md` and allow to point programmers to preferred functions to carry out a specific task. For example, there are many different types of lowpass filters declared in `filters.lib` and only one of them is considered to be standard, etc.

Testing the library

Before preparing a pull-request, the new library must be carefully tested:

- all functions defined in the library must be tested by preparing a DSP test program, to be added using the `#### Test` syntax
- the compatibility library `all.lib` imports all libraries in a same namespace, so check functions names collisions using the following test program:

```
import("all.lib");
process = _;
```

Library test and deployment

For GRAME maintainers:

- global tests can be done using the `make reference` and `make check` at rool level
- regenerate the PDF documentation using `make pdf` target in the `doc` folder
- update the library submodule in faust, recompile and deploy WebAssembly libfaust in fausteditor, faustplayground and faustide
- update the library submodule in faustlive
- update the library list in this fausteditor page as well as the snippets (using the `faust2atomsnippets` tool).
- update the library list in this faustide page and LIBRARIES folder.
- update the library list in the faustgen~ code
- update the Faust Syntax Highlighting Files
- make an update PR for vscode-faust project
