BUILDING FAUST WITH CMAKE

Dominique Fober
∗
GRAME
Lyon, France
fober@grame.fr

Yann Orlarey†
GRAME
Lyon, France
orlarey@grame.fr

Stephane Letz‡
GRAME
Lyon, France
letz@grame.fr

ABSTRACT

This paper describes the new Faust building system that is now based on CMake. This new building system preserves the previous Makefile approach as much as possible while offering far more flexibility and above all, a platform independent solution for compiling the various faust components. The paper gives practical information to address basic uses of the building system as well as for advanced and custom settings.

1. INTRODUCTION

While Faust 1 was nearly free of dependencies, the current version can be more challenging to compile, due in particular to the LLVM dependency. Moreover until version 2.5.23 the building system was based on Make and compiling Faust for non Unix systems like Windows was quite complicated. For all these reasons we decided, in November 2018, to develop a new build system for Faust, based on CMake.

CMake offers a great flexibility, both in defining the targets to be compiled as well as selecting the different backends to be included in each target. However, this flexibility is based on a set of states (cached by CMake), which can sometimes make the compilation process a bit obscure.

The goal of this paper is to give you some practical information about this new building system. The first section explains how to do a basic Faust installation, when LLVM is not required. The second section gives details on all the available options.

2. BASIC FAUST INSTALLATION

If you don’t have special needs, if you essentially want to compile Faust code to your favorite target, then the new building system is as simple to use as the old one. In this section we will explain all the needed steps to install and run Faust on a fresh Ubuntu 16.04 distribution.

2.1. Compiling the compiler

The first step is to compile Faust itself. Let’s start with the minimal requirement in terms of packages: the building tools, git and libmicrohttpd.

$ sudo apt-get update
$ sudo apt-get install -y build-essential
cmake git libmicrohttpd-dev

We can now clone the Github repository of Faust

$ git clone https://github.com/grame-cncm/ faust.git

then compile and install Faust:

$ cd faust
$ make
$ sudo make install

Once the installation has been completed, we can check it:

$ faust -v

This will give you the version of the Faust compiler with a list of the available backends:

FAUST : DSP to C, C++, Java, JavaScript, old C++, asm.js, WebAssembly (wast/wasm) compiler, Version 2.5.30
Copyright (C) 2002-2018, GRAME - Centre National de Creation Musicale. All rights reserved.

As you can note the LLVM backend is not installed in this basic setup. You will typically need LLVM if you want to compile the library version of Faust. A complete installation is described in the next section.

We don’t have any audio development package installed, but we can nevertheless use Faust to compile some of the examples to C++ code, at least to check that the compiler works:

$ faust examples/generator/noise.dsp

The above command will compile noise.dsp and generate a C++ code on the standard output.

This C++ code has to be embedded into an architecture file (that describes how to relate the audio computation to the external world) before being compiled into standalone application or an audio plugin. For that you will need to install some development packages depending of your targets.

2.2. Compiling Alsa and JACK applications

In order to compile Faust programs to Alsa or JACK applications for Linux you need to install the corresponding development packages:

$ sudo apt-get install -y libasound2-dev
libjack-jackd2-dev libgtk2.0-dev

It is now possible to compile an Alsa or JACK application with the various possible options:

$ faust2alsa -osc -httpd -midi foo.dsp
$ faust2jack -osc -httpd -midi foo.dsp
2.3. Compiling QT applications

Instead of using Gtk you may want to use QT. In this case the QT4 development package needs to be installed:

```bash
$ sudo apt-get install -y libasound2-dev libjack-jackd2-dev libqt4-dev
```

Then it is possible to compile for Alsa/QT and JACK/QT:

```bash
$ faust2alqt -midi -httpd -osc foo.dsp
$ faust2jaqt -midi -httpd -osc foo.dsp
```

2.4. Compiling LADSPA, DSSI and LV2 plugins

Linux supports several formats of plugins including LADSPA, DSSI and LV2.

```bash
$ sudo apt-get install -y ladspa-sdk
$ sudo apt-get install -y dssi-dev
$ sudo apt-get install -y lv2-dev libboost-dev
```

These plugins can now be compiled:

```bash
$ faust2ladspa good.dsp
$ faust2dssi good.dsp
$ faust2lv2 good.dsp
```

2.5. Compiling for SuperCollider

In order to compile SuperCollider UGens you need to install the SuperCollider development package. For some reason the development package requires jackd to be installed and the jackd package requires a user response! In some situation (building a Docker image) this is a problem. The solution to have a fully silent installation is to set `DEBIAN_FRONTEND` to `noninteractive`.

```bash
$ sudo DEBIAN_FRONTEND=noninteractive apt-get install -y supercollider-dev
$ faust2supercollider foo.dsp
```

2.6. Compiling for CSound

Compile for CSound only requires to install the libcsound development package.

```bash
$ sudo apt-get install -y libcsound64-dev
$ faust2csound good.dsp
```

2.7. Compiling for Pure Data

The puredata-dev package is enough to compile Pure Data externals. But they will lack the nice UI patch generated by Albert Graef’s faust2pd.

```bash
$ sudo apt-get install -y software-properties-common
$ sudo add-apt-repository -y "ppa:dr-graef/pure-lang.xenial"
$ sudo apt-get update
$ sudo apt-get install -y faust2pd faust2pd-extra
```

Now `faust2puredata` will detect `faust2pd` and:

```bash
$ faust2puredata foo.dsp
```

will produce the external object `foo~.pd_linux` and the UI patch `foo.pd`.

2.8. Compiling a VST plugin for Linux

In order to compile vst plugins for Linux you need to install the VST SDK provided by Steinberg.

```bash
$ sudo apt-get install -y unzip
$ sudo unzip vstsdk365_12_11_2015_build_67.zip -d /usr/local/include/
$ sudo mv /usr/local/include/VST3/SDK /usr/local/include/vstsdk2.4
```

Once this is done:

```bash
$ faust2faustvst foo.dsp
```

will produce `foo.so` a VST plugin for Linux.

3. ADVANCED FAUST INSTALLATION

The Faust distribution includes the Faust compiler, but also other elements that you may want to compile, in particular libfaust, the library version of the Faust compiler. Moreover, the way these elements are compiled can be configured with appropriate files.

3.1. Faust backends

The Faust compiler can produce various languages on output. Support for these languages is provided using backends that may or may not be embedded into the compiler or into the faust libraries. This is intended to simplify the compilation process: some backends (like LLVM) proved to be a bit complex to compile, some others are not supported by all compilers (like the interpreter backend). In addition, selecting only the set of backends to be used, can reduce significantly the size of the resulting binary.

3.1.1. Selecting your backends

The backends selection is described using `backends` files which are actually cmake files that simply populate the cmake cache. These files are located in the `backends` folder. They consist in a matrix where each line corresponds to a language support and where the columns select (or discard) the corresponding backend for each binary output i.e.:

- the Faust compiler.
- the libfaust static library,
• the libfaust dynamic library,
• the libfaust asmjs library,
• the libfaust wasm library

The example in figure 1 selects the ASMJS backend for the asmjs library, the cpp backend for the compiler and the faust static and dynamic libraries and discards the interpreter backend.

A BACKENDS option is provided to select a backend file using make e.g.:

done

Example:  
```
make BACKENDS=asmjs-build
```

By default the selected backends are taken from `backends.cmake`.

Note that make always looks for the backend files in the `backends` folder.

You can get similar results using direct cmake invocation:

done

Example:  
```
cd faustdir
make -C ../backends/backends.cmake
```

The configuration file `faustdir/backends.cmake` may be modified to add or remove backends. Note that make always looks for the backend files in the `backends` folder.

3.2.1. Project generation

This is the step where you choose what you want to include in your project and to compile in a second step. The Faust compiler, the OSC and HTTP libraries are included by default, but you can add (or remove) the Faust libraries (static or dynamic versions) as argument. You can also choose the form of your project: a Makefile, an Xcode or Visual Studio project, or any of the generator provided by cmake on your platform.

You may think of this step as the definition of the targets that will be available from your project. Note that at this step, you also choose the Faust backends that you want to include in the different components (compiler and faust libraries). See the `backends` subsection for more details.

3.2.2. The project form and location

Cmake provides support for a lot of development environments depending on your platform. To know what environments are supported, type `cmake --help` and you’ll get a list of the supported generators at the end of the help message.

By default, the Makefile makes use of “Unix Makefiles” (or “MSYS Makefiles” on Windows). Thus when you type `make`, it generates a makefile and then run a make command using this Makefile. To avoid overwriting the existing makefile, the project is generated in a subfolder named `faustdir` by default and created on the fly.

You can freely change these default settings make and the FAUSTDIR and GENERATOR options, that control the subfolder name and the generator to use. For example:

```
$ make GENERATOR=Xcode
```

will generate an Xcode project in the `faustdir` subfolder.

```
$ make FAUSTDIR=macos GENERATOR=Xcode
```

will generate an Xcode project in the `macos` subfolder.

You can achieve similar results using direct cmake invocation e.g.:

```
$ mkdir macos
$ cd macos
$ cmake .. -G Xcode
```

3.2.3. The project targets

By default, the generated project includes the Faust compiler and the OSC and HTTPD static libraries, but not the Faust static or dynamic libraries. The makefile provides specific targets to include these libraries in your project:

```
• make configstatic: add the libfaust static library to your projects
• make configdynamic: add the libfaust dynamic library to your projects
• make configall: add the libfaust static and dynamic libraries to your projects
• make reset: restore the default project settings.
```

Equivalent settings using direct cmake invocation. For example and to add/remove the libfaust static library to/from your project, you can run the following command from your `faustdir`:
set (ASMJS_BACKEND ASMJS CACHE STRING "Include ASMJS backend" FORCE)
set (CPP_BACKEND COMPILER STATIC DYNAMIC CACHE STRING "Include CPP backend" FORCE)
set (INTERP_BACKEND OFF CACHE STRING "Include INTERPRETER backend" FORCE)

Figure 1: Example of backend matrix configuration.

$ cmake -DINCLUDE_STATIC=[on/off] ..

You can have a look at the Makefile to see the correspondence between the make targets and the cmake equivalent call. Note that since cmake is a state machine, it’ll keep all the current settings (i.e. the values of the cmake variables) unless specified with new values.

3.2.4. Re-generate the project

The makefile includes a special target to re-generate your project. It allows to change your backends, but can also be necessary to include new source files in your project (source files are scanned at project generation and are not described explicitly). Simply type:

$ make cmake [options]

All the above options can be specified when running the cmake target (apart the GENERATOR option that can’t be changed at cmake level).

Equivalent call with cmake has the following form:

$ cd faustdir
$ cmake .. [optional cmake options]

3.2.5. Miscellaneous project configuration targets

- make verbose: activates the printing of the exact command that is run at each make step
- make silent: reverts what make verbose did.
- make universal: [MacOSX only] creates universal binaries
- make native: [MacOSX only] reverts native only binaries (default state).

3.3. Compiling using make or cmake

Once your project has been generated (see Building steps), the default is to compile all the targets that are included in the project. Thus, typing make will build the Faust compiler, the OSC static library and the HTTP static library when these 3 components are included in your project.

3.3.1. Single targets always supported

Single targets are available use make or cmake. These targets are:

- faust: to build the Faust compiler
- osc: to build the OSC library
- http: to build the HTTP library

3.3.2. Single targets that require a project configuration

- staticlib: to build libfaust library in static mode. Requires to call make configstatic first.
- dynamiclib: to build libfaust library in dynamic mode. Requires to call make configdynamic first.
- oscdynamic: to build OSC library in dynamic mode. Requires to call make configoscdynamic first.
- httpdynamic: to build HTTP library in dynamic mode. Requires to call make confighttpdynamic first.

3.3.3. Targets excluded from all

- wasmlib: to build libfaust as a Web Assembly library.
- asmjslib: to build libfaust as an ASM JS library.

These targets require the emcc compiler to be available from your path.

3.3.4. Platform specific targets

- ioslib: to build libfaust library in static mode for iOS.

3.3.5. Invoking targets from cmake

The general form to invoke a target using cmake commands is the following:

$ cmake --build <project dir> [--target target] [-- native project options]

The default cmake target is all. For example the following command builds all the targets included in your project:

$ cmake --build faustdir

Cmake takes care of the generator you used and thus, provides an universal way to build your project from the command line whether it’s Makefile based or IDE based (e.g. Xcode or Visual Studio)

The following sequence creates and build a project using Visual Studio on Windows in release mode:

$ cd your_build_folder
$ cmake -C ../backends/backends.cmake .. -G "Visual Studio 14 2015 Win64"
$ cmake --build . --config Release

For more details and options, you should refer to the cmake documentation.

3.3.6. The install and uninstall targets

Your project will always include an install target, which always installs all the components included in the project. There is no uninstall target at cmake level (not supported by cmake). It is provided by the Makefile only and is based on the
install_manifest.txt file that is generated by the install target in faustdir.

Note that cmake ensures that all the targets of your project are up-to-date before installing and thus may compile some or all of the targets. It can be annoying if you invoke sudo make install: the object files will then be property of the superuser and you can then have errors during later compilation due to write rights issues on object files. Thus it is recommended to make sure that all your targets are up-to-date by running make before running sudo make install.

4. CONCLUSIONS

We have presented Faust's new building system based on CMake. The retained approach preserves as much as possible the simplicity of Makefiles with the flexibility and power of CMake. Above all, it proposes a platform independent solution for compiling the various Faust components. We hope that this new building system will also facilitate the distribution of Faust on non Unix platforms, in particular Windows.

5. ACKNOWLEDGMENTS

We would like to thank Albert Gräf and Romain Michon for their very useful comments and bug reports during the development of this new building system.

6. USEFUL LINKS

Faust Wiki: https://github.com/grame-cnrm/faust/wiki
CMake documentation: https://cmake.org
# A. Variables Reference

## A.1. Variables that control the project generation

<table>
<thead>
<tr>
<th>Name</th>
<th>default</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAUSTDIR</td>
<td>faustdir</td>
<td>the project folder name</td>
</tr>
<tr>
<td>IOSDIR</td>
<td>iosdir</td>
<td>the project folder name used by the ioslib target only</td>
</tr>
<tr>
<td>CMAKEOPT</td>
<td>-DCMAKE_BUILD_TYPE=Release</td>
<td>the project build types see CMAKE_BUILD_TYPE in cmake documentation</td>
</tr>
<tr>
<td>GENERATOR</td>
<td>&quot;Unix Makefiles&quot;</td>
<td>on Unix like systems on Windows</td>
</tr>
<tr>
<td>BACKENDS</td>
<td>backends.cmake</td>
<td>defines the embedded backends (see section 3.1)</td>
</tr>
</tbody>
</table>

### Table 1: Variables defined at make level only

<table>
<thead>
<tr>
<th>Name</th>
<th>default</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIVERSAL</td>
<td>off</td>
<td>MacOSX only: control universal binaries generation</td>
</tr>
<tr>
<td>CMAKE_VERBOSE_MAKEFILE</td>
<td>off</td>
<td>Makefiles only: control make verbosity</td>
</tr>
<tr>
<td>INCLUDE_STATIC</td>
<td>off</td>
<td>Include libfaust static library</td>
</tr>
<tr>
<td>INCLUDE_DYNAMIC</td>
<td>off</td>
<td>Include libfaust dynamic library</td>
</tr>
<tr>
<td>INCLUDE_OSC</td>
<td>on</td>
<td>Include Faust OSC static library</td>
</tr>
<tr>
<td>INCLUDE_HTTP</td>
<td>on</td>
<td>Include Faust HTTPD library</td>
</tr>
<tr>
<td>OSC_DYNAMIC</td>
<td>off</td>
<td>Include Faust OSC dynamic library</td>
</tr>
<tr>
<td>HTTPDYNAMIC</td>
<td>off</td>
<td>Include Faust HTTPD dynamic library</td>
</tr>
<tr>
<td>USE_LLVM_CONFIG</td>
<td>on</td>
<td>makes use of llvm-config to scan LLVM settings when off, cmake try to use llvm-config.cmake (if any) to use an alternate llvm-config name or path</td>
</tr>
<tr>
<td>LLVM_CONFIG</td>
<td>llvm-config</td>
<td></td>
</tr>
</tbody>
</table>

### Table 2: Variables defined at cmake level only

## A.2. Variables that control the project compilation

<table>
<thead>
<tr>
<th>Name</th>
<th>default</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUILDOPT</td>
<td>--config Release</td>
<td>compiles in release mode depends on CMAKE_BUILD_TYPE (see CMAKEOPT above)</td>
</tr>
<tr>
<td>JOBS</td>
<td>-j 4</td>
<td>when using make with Xcode projects with MSVC projects</td>
</tr>
<tr>
<td></td>
<td>-jobs 4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/maxcpucount:4</td>
<td></td>
</tr>
</tbody>
</table>

## A.3. Variables that control the project installation

<table>
<thead>
<tr>
<th>Name</th>
<th>default</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>DESTDIR</td>
<td></td>
<td>the install destination directory</td>
</tr>
<tr>
<td>PREFIX</td>
<td>/usr/local</td>
<td>on Unix systems on Windows note that PREFIX is translated to CMAKE_INSTALL_PREFIX at cmake level</td>
</tr>
<tr>
<td></td>
<td>C:\Program Files</td>
<td></td>
</tr>
</tbody>
</table>