FluidSynth Plugin

Simple Wrappers Around the ***`FluidSynth`*** *Library as DAW Plugin and Pedantic Command Line Processor*

Author:	Dr. Thomas Tensi
Date:	2023-01-09
Version:	0.3
Platforms:	VST3 on Windows x64, VST3/AU on MacOSX (x86_64) VST3 on Linux (x86_64)

Contents

Chapter 1

Introduction

1.1 Overview

`FluidSynth` [FluidSynthDOC] is one of the most prominent open source MIDI players. It is reasonably flexible, delivers a good audio quality and is available for the typical platforms. A common scenario is to use it for either rendering live MIDI data on some audio device or converting MIDI files into audio files by command-line batch processing.

Basis of `FluidSynth` are the so-called *soundfonts*. Soundfonts contain sampled instruments together with envelope and modulation definitions and other descriptive settings. It is easy to find really usable ones in the internet and also several of those cover all general MIDI instruments (for example, the FluidR3_GM.sf2).

So when using `FluidSynth` in a command-line driven context all is well. But when you want play around with settings for `FluidSynth` interactively in a DAW, you need some DAW plugin rendering audio from MIDI as close as possible to the original command-line fluidsynth.

So far there is no such thing to exactly emulate FluidSynth in a DAW context.

There were some previous efforts like Alexey Zhelezov's FluidSynthVST [FluidSynthVST] or Birch Labs' JuicySFPlugin [JuicySFPlugin], but those are a bit tricky to use and support for them is unclear. But the main point is that even though they rely on the `FluidSynth` library, they **do not exactly reproduce an external fluidsynth rendition of some MIDI data**. Reasons for that will be explained below.

The reason for being picky about the exact rendering is as follows: my scenario is a command-line based rendering of notation videos for a band (the LilypondToBandVideoConverter [LTBVC]). Part of that chain is fluidsynth, but I want to experiment interactively with settings in a DAW to optimize the audio and then have a faithful reproduction of the external rendering pipeline within the DAW.

So the first component of this package is a DAW plugin called "`FluidSynth-Plugin`". It has a simplistic interface where you specify a soundfont, several fluidsynth settings and possibly a MIDI program to be selected by putting text data in a text field. Then you are able to convert an incoming MIDI stream in a DAW to audio using the `FluidSynth` library.

But when playing around with that plugin some inexplicable differences to the command-line `FluidSynth` occured. Even when using innocent soundfonts (without chorus and other modulators), sample playback in the plugin and the command-line player were not absolutely identical. Analysis and contact with the `FluidSynth` team revealed that in that program MIDI events are quantized onto some processing raster in the millisecond range while the plugin quantizes them onto the smallest time unit: the sample raster itself.

So, for example, for a sample rate of 44.1kHz this 64 sample offset might lead to a time difference of more than 1ms between events in the DAW and in an external tool chain. You cannot hear this, but of course this leads to significant differences in the rendered audio (for example, when doing a spectrum analysis). In section 3.5 we will see that although the plugin feeds the events with sample raster precision to the `FluidSynth` library some inevitable internal rasterization happens there.

Another tool mitigates the rasterization by the player of `FluidSynth`. That second component of this package is a simplistic but pedantic command-line converter called "`PedanticFluidSynthConverter`". It converts a MIDI file into a WAV file, is also based on the fluidsynth library and does the same sample-exact event feeding into that library as the plugin. Hence it should produce identical results when some circumstances are guaranteed (see section 3.5).

When using both components (command-line and DAW) on the same MIDI data they produce audio output with a difference of less than -200dBFS in a spectrum analysis.

Those components are available for the x86-64 architecture in Windows, MacOS and Linux with the plugin having either VST3 (Windows, Linux) or AU format (MacOS).

All the code is open-source; hence you can check and adapt it to your needs (see chapter 5).

1.2 Acknowledgements

This project is a derivative work based on the foundations laid by the FluidSynth community.

My thanks go to the FluidSynth team: Peter Hanappe, Conrad Berhörster, Antoine Schmitt, Pedro López-Cabanillas, Josh Green, David Henningsson and Tom Moebert. Without your effort this would not have been possible!

Chapter 2

Installation of the FluidSynth-Plugins

The installation is as follows:

1. Copy the plugins from the appropriate subdirectory for your platform of _DISTRIBUTION/targetPlatforms directory in [FluidSynthPlugin] into the directory for VST or AU plugins of your DAW.
2. If helpful, you can put this documentation pdf file contained in subdirectory doc and test files in subdirectory test (see section 4) somewhere.
3. When installing the plugins on MacOSX, note that those are **not signed**; so you have to explicitly remove the quarantine flag from them (e.g. by applying the command `sudo xattr -rd com.apple.quarantine «vstPath»`).
4. When installing the plugin and program on Windows, they require the so-called Microsoft Visual C++ Redistributable library **??**. Very often this is already installed on your system; if not, you have to install it from the Microsoft site.
5. Restart your DAW and rescan the plugins. You should now be able to select the `FluidSynthPlugin`.
6. The command-line version `PedanticFluidSynthConverter` can be put in an arbitrary location for executables. Ensure that the dynamic libraries in its directory are also placed appropriately.

Note that the plugin directory and the directory of the `PedanticFluidSynthConverter` contain a dynamically linked library version of `FluidSynth` as well as related libraries. You can easily exchange those libraries for more current versions.

Chapter 3

Description of the FluidSynth-Plugins

3.1 General Remarks

As mentioned in the introduction this package provides tools written in C++ for emulating `FluidSynth` bit-exactly: a plugin called “`FluidSynthPlugin`” and a very precise command-line clone of `FluidSynth` called “`Pedantic-FluidSynthConverter`”.

The term “bit-exactly” is a bit misleading: because the tools considered (including `FluidSynth`) have different execution environments (with some data conversions and roundings involved), absolute identical outputs are unreasonable. But what can be achieved is that they are identical to a degree that cannot be heard. If you analyze the spectrum of the difference signal and this difference lies below some assumed threshold, this will be good enough. The threshold here is -200dBFS which is about 100dB below the noise level of a CD.

So both tools render MIDI data similarly to `FluidSynth` and identical to each other. And when there is no difference in rasterization the difference between the audio from those tools is identical — up to that precision – to `FluidSynth`.

So why is there a difference between `FluidSynthPlugin` and `FluidSynth` in some cases? The differences come from the fact that `FluidSynth` shifts MIDI events onto some broader raster for audio samples while both programs presented here place an event onto a sample position. This means that there is a slight delay in audio in `FluidSynth` and original audio and emulated do not cancel out when subtracted.

Figure 1 shows the problem: when an event occurs the waveform is started in `FluidSynthPlugin` at the next sample position while it starts in `FluidSynth`

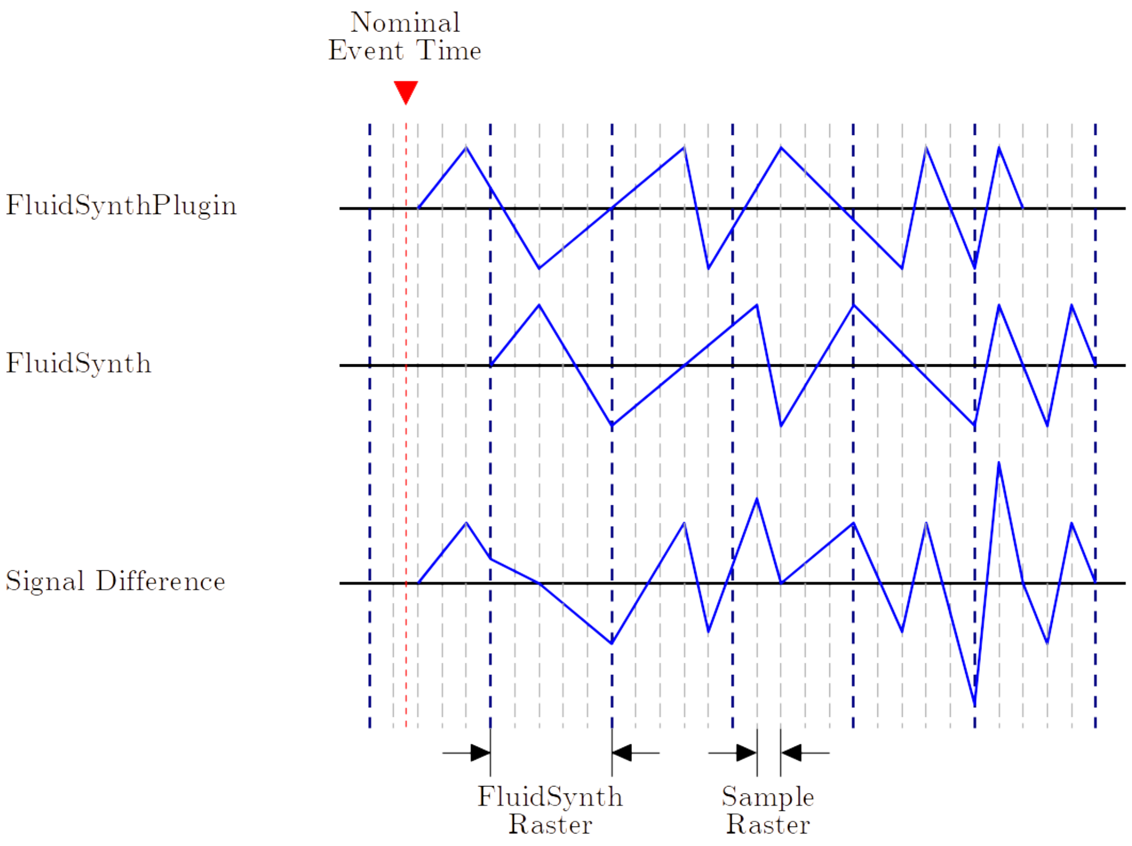


Figure 1: Rasterization Problem in `FluidSynth`

at the 64 sample raster. As can be easily seen, the difference is not zero, but some other complex waveform.

Note that this is not the complete truth as will be discussed in section 3.5: some shifting will still be done in the underlying `FluidSynth` library.

3.2 Supported `FluidSynth` Settings

Both programs support a subset of settings from `FluidSynth`. It is a subset, because, for example, all settings related to driver selection are omitted: those do not make any sense in the context of these programs.

The supported settings are shown in figures 2, 3 and 4 (with short explanations taken from from [FluidSynthSettings]).

Besides the standard settings from `FluidSynth` there are two special settings available: soundfont and program.

Note that the specified soundfont path must be an *absolute path*, because it is impossible for the plugin to find out the path of the enclosing project in the

Parameter	**Description**	**Type**
`synth.chorus.active`	tells whether chorus will be added to the output signal	boolean
`synth.chorus.depth`	specifies the modulation depth of the chorus	float
`synth.chorus.level`	specifies the output amplitude of the chorus signal	float
`synth.chorus.nr`	sets the voice count of the chorus	integer
`synth.chorus.speed`	sets the modulation speed in Hz	float
`synth.default-soundfont`	default soundfont file to use by the program	string
`synth.dynamic-sample-loading`	tells whether samples are loaded to and unloaded from memory whenever presets are being selected or unselected for a MIDI channel	boolean

Figure 2: Common Synthesizer Settings for `FluidSynthPlugin` and `PedanticFluidSynthConverter` (Part 1)

DAW and then use a relative path specification. Normally this should not be a problem — because soundfonts are often located in a specific directory in a system —, but it somewhat impedes portability of a DAW project containing this plugin.

But you can use environment variables in the path specification enclosed by `${` and `}`, for example `${soundFontDirectory}`.

Parameter	**Description**	**Type**
`synth.gain`	gain applied to the final or master output of the synthesizer	float
`synth.midi-bank-select`	defines how the synthesizer interprets bank select messages	string
`synth.min-note-length`		integer
`synth.overflow.age`	tells how event age is accounted for in a voice overflow situation	float
`synth.overflow.important`	another parameter for voice overflow handling	float
`synth.overflow.important-channels`	comma-separated list of MIDI channel numbers that should be treated as “important” by the overflow calculation	string
`synth.overflow.percussion`	overflow priority score to be added to voices on a percussion channel	float
`synth.overflow.released`	overflow priority score added to voices that have already received a note-off event	float
`synth.overflow.sustained`	overflow priority score added to currently sustained voices	float
`synth.overflow.volume`	overflow priority score added to voices based on their current volume	float

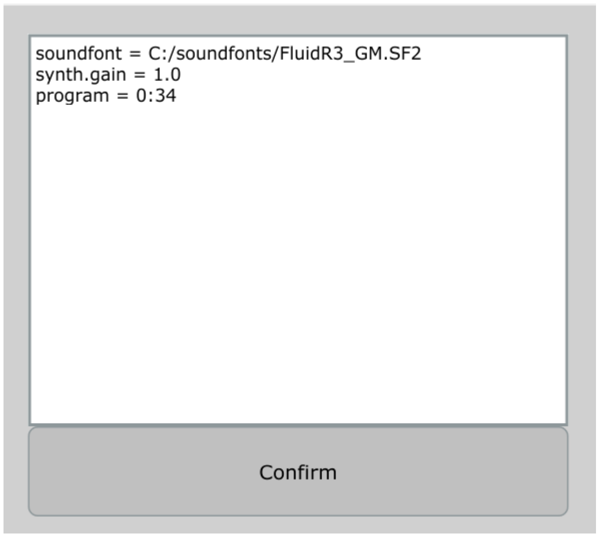
Figure 3: Common Synthesizer Settings for `FluidSynthPlugin` and `PedanticFluidSynthConverter` (Part 2)

Parameter	**Description**	**Type**
`synth.polyphony`	defines how many voices can be played in parallel	integer
`synth.reverb.active`	tells whether reverb will be added to the output signal	boolean
`synth.reverb.damp`	sets the amount of reverb damping	float
`synth.reverb.level`	sets the reverb output amplitude	float
`synth.reverb.room-size`	sets the room size (i.e. amount of wet) reverb	float
`synth.reverb.width`	sets the stereo spread of the reverb signal	float
`synth.sample-rate`	sample rate of the audio generated by the synthesizer	float
`synth.verbose`	tells whether synthesizer will print out information about the received MIDI events to stdout	boolean

Figure 4: Common Synthesizer Settings for `FluidSynthPlugin` and `PedanticFluidSynthConverter` (Part 3)

Parameter	**Description**	**Type**
`soundfont`	gives the path to the soundfont and may contain environment variables enclosed by `${` and }	string
`program`	gives the program to use as the default; format is bankNumber:programNumber where counting for both numbers starts at zero	string

Figure 5: Additional Settings in `FluidSynthPlugin` and `PedanticFluidSynthConverter`



```
soundfont = C:/soundfonts/FluidR3_GM.SF2
synth.gain = 1.0
program = 0:34
```

Figure 6: `FluidSynthPlugin` User Interface

3.3 Using the `FluidSynthPlugin`

The “`FluidSynthPlugin`” is a MIDI instrument in a DAW converting incoming MIDI input into an outgoing audio stream.

The configuration of the plugin is done via a very simplistic interface: a multiline edit field can be used for command entry.

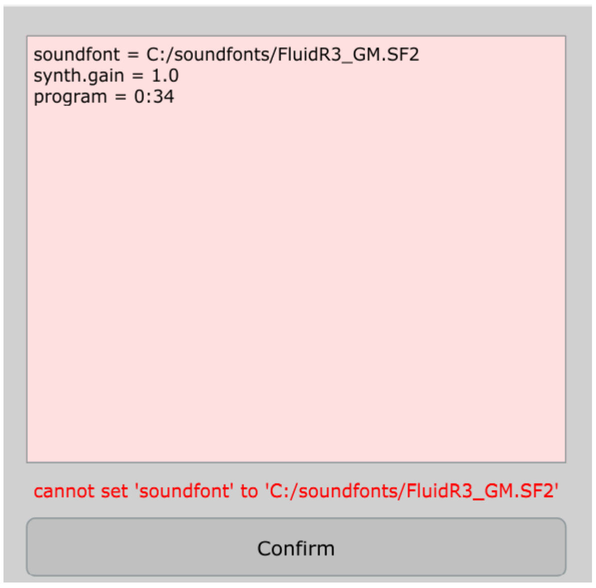
In this multiline field the `FluidSynth` commands from figures 2 to 5 can be written. Figure 6 shows how the user interface of the plugin looks like.

Each line may contain a `FluidSynth` setting or a definition for soundfont or program. A setting consists of a key string (like e.g. synth.reverb.active), an equal sign and a value appropriate for the setting. Leading and trailing blanks are ignored; strings are given without quotes and float values must have a decimal point.

When data is entered in the multiline edit field, it changes its background from grey to white. This signifies that the data has not yet been registered by the plugin. To achieve this the Confirm button has to be pressed: the data is checked and then used by the underlying `FluidSynth` synthesizer.

When the check fails, an error message is given and the edit field is still in edit mode. Note that only the first error encountered is reported, so you have to incrementally correct the settings. For example, in figure 7 the soundfont path seems to be incorrect: this has to be corrected before any other error will be analyzed.

Note that also some error message is displayed, when the fluidsynth library cannot be found by the plugin. In that case please make sure that the



```
soundfont = C:/soundfonts/FluidR3_GM.SF2
synth.gain = 1.0
program = 0:34
```

cannot set 'soundfont' to 'C:/soundfonts/FluidR3_GM.SF2'

Figure 7: Error Message in `FluidSynthPlugin`

installation has been correctly done (the plugin expects its dynamic libraries in the directory of the plugin vst3 file).

3.4 Using the `PedanticFluidSynthConverter`

The `PedanticFluidSynthConverter` is merely a functionally reduced clone of `FluidSynth`, but with a special property: it places MIDI events onto the raster given by the sample rate.

Hence similarly to `FluidSynth`, the `PedanticFluidSynthConverter` is a command-line program. But the converter does not have to do real-time processing, so the list of its options is reduced. On top of that it only supports a conversion from a MIDI file into an audio WAVE file.

The supported command line options are shown in figure 8 and 9. Any parameter not belonging to an option is interpreted as a file name. Files with their names ending in .sf2 or .sf3 are considered to be sound fonts, files with their names ending in .mid are considered to be MIDI files.

Hence the command line is

```
fluidSynthFileConverter [options] midifile soundfontfile
                        -F wavefile
```

So, for example, the command line

```
fluidSynthFileConverter -g 1.0 -R 0 test.mid FluidR3\_GM.sf2
                        -F test.wav
```

produces a wave file “test.wav” from MIDI file “test.mid” using sound font “FluidR3_GM.sf2” with reverb turned off and gain set to unity.

Option	Description

`-a`
`--audio-driver=«name»`
	audio driver to use (IGNORED)

`-C` `--chorus`	turn the chorus on or off [0|1|yes|no, default = on]

`-c`
`--audio-bufcount=«count»`
	number of audio buffers (IGNORED)

`-d` `--dump`	dump incoming and outgoing MIDI events to stdout (IGNORED)
`-E` `--audio-file-endian`	audio file endian (IGNORED: always little endian)
`-f` `--load-config`	load and execute a configuration file

`-F`
`--fast-render=«file»`
	name of target WAVE file (REQUIRED)

`-G` `--audio-groups`	define the number of LADSPA audio nodes (IGNORED)
`-g` `--gain`	set the master gain $0 < \text{gain} < 10$, default = 0.2
`-h` `--help`	print out help summary
`-i` `-no-shell`	don't read commands from the shell (IGNORED)
`-j` `-connect-jack-outputs`	connect jack outputs to the physical ports (IGNORED)

`-K`
`-midi-channels=«num»`
	number of midi channels [default = 16] (IGNORED)

`-L`
`-audio-channels=«num»`
	number of stereo audio channels [default = 1] (IGNORED)

`-l` `-disable-lash`	don't connect to LASH server (IGNORED)

`-m`
`-midi-driver=«label»`
	name of the midi driver to use (IGNORED)

Figure 8: Command Line Options for `PedanticFluidSynthConverter` (Part 1)

Option	**Description**
`-n` `-no-midi-in`	don't create midi driver to read MIDI input events (IGNORED)
`-O` `-audio-file-format`	audio file format for fast rendering [double|float|s8|s16|s24|s32, default = s16]
`-o`	define a setting, -o name=value; see FluidSynth for details

`-p`
`-portname=«label»`
	set MIDI port name (IGNORED)

`-q` `-quiet`	do not print informational output
`-R` `-reverb`	turn reverb on or off [0|1|yes|no, default = on]
`-r` `-sample-rate`	set the sample rate
`-s` `-server`	start FluidSynth as a server process (IGNORED)
`-T` `-audio-file-type`	audio file type for fast rendering (IGNORED: always WAV)
`-v` `-verbose`	print out verbose info about midi events (synth.verbose=1)
`-V` `-version`	show version of program

`-z`
`-audio-bufsize=«size»`
	size of each audio buffer (IGNORED)

Figure 9: Command Line Options for `PedanticFluidSynthConverter` (Part 2)

3.5 Restrictions

No Timelocking Available

Often audio effects produce the same output for the same input, but sometimes effects behave differently in time, technically they are *time-variant*.

An example of the former is a filter: it does not care *when* a signal arrives, it always reacts in the same way. An example of the latter is a modulated effect like e.g. a tremolo: it produces a different sound for different event times because the modulation is normally in another phase.

Hence when looking at the behaviour at a specific point in time, those time-variant effects would behave differently when the effect start time is varied.

Of course, a sample player — like `FluidSynth` — very often is also time-variant. It is not, when only a sample playback is triggered, because the audio will be the same whenever you start its playback. But when there is some modulation happening (for example, caused by a chorus effect) the effect is time-variant: the audio output produced will not be the same for different playback start times unless the modulation is in some way synchronized.

So for some externally generated audio snippet with modulation at an arbitrary position in a DAW project, a modulation by a corresponding plugin would only be congruent by accident: typically it is out of phase. The reason for that is that plugins normally start their modulation when playback begins. This means technically the phase 0° of the modulation is on the time of playback start.

Figure 10 show that situation for an example. We assume that an amplitude modulation occurs in a soundfont and we have inserted an externally rendered audio snippet (e.g. generated by `FluidSynth`) into the DAW starting at time $t_{fragment}$ into an audio track. Now the playback in the DAW is assumed to start at time t_{play}. As one can easily see, the modulation for the externally processed fragment (that just puts a modulation on the raw sample data) has its phase 0° exactly at time $t_{fragment}$. However, the internal effect in the DAW has its phase 0° at time t_{play} (see also the red dots on the respective tracks marking the phase 0° positions). This would lead to massive differences between externally and internally generated audio.

But it could be rectified by defining for the internal effect *at which point in time the modulation phase should be* 0° (so-called "time-locking"). If you set this time offset parameter to $t_{fragment}$, the modulations will be synchronous (as you can see when comparing the second with the lowest track). Of course, the effect starts at t_{play}, but its modulation phase is shifted such that it reaches phase 0° exactly at $t_{fragment}$.

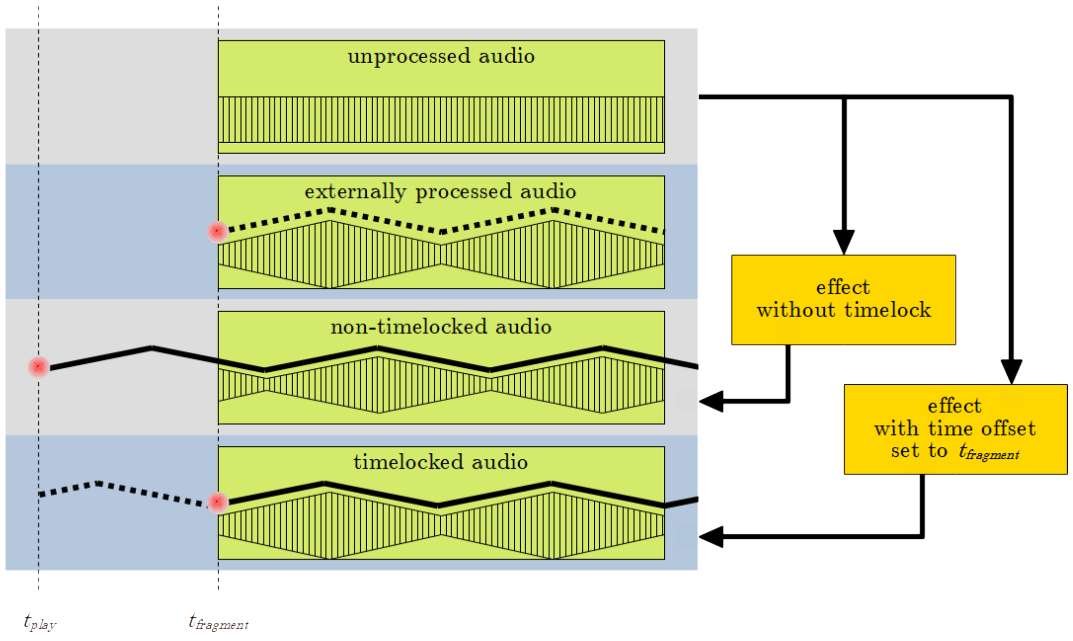


Figure 10: Deviation in Modulation between External and Internal Generation and Timelocking

While this method would lead to perfect reproduction of the external rendering, it is not feasible for the `FluidSynthPlugin`. There is no direct way to set the modulators in the underlying `FluidSynth` library to a specific phase. As a workaround when $t_{fragment} < t_{play}$ one could reset all modulators at playback start and first silently have the synthesizer process samples for a duration of $t_{play} - t_{fragment}$ to bring its modulators to the correct phases before then putting out the "real" samples. But this is tedious, takes a lot of processing time and also does not solve the problem of having the playback starting before the fragment start time.

So there is no good solution for that.

When you need a bit-exact reproduction of externally rendered audio by the `FluidSynthPlugin`, some workaround has to be made as follows:

- The selected instrument(s) in the soundfont must not contain any (free-running) modulators.
- Chorus must be deactivated (e.g. by setting synth.chorus.active to "0").

Forced Rasterization by `FluidSynth`

As mentioned in the introduction the important difference between using the standard `FluidSynth` from the command-line versus from a DAW is that there is a forced rasterization to 64 samples' intervals.

Unfortunately this rasterization is not just done by `FluidSynth` when communicating with audio drivers or its file renderer, but also done internally by the fluidsynth synthesizer in the library. The length of the smallest unit for which `FluidSynth` can make state changes and does buffering is a constant called FLUID_BUFSIZE and this is fixed to the value 64.

So what can we do?

- We could recompile the `FluidSynth` library and set this value to 1. This would on the one hand lead to a performance penalty, but would on the other hand provide sample-exact processing.

  I did not choose that option, because I wanted to use the *stock* `FluidSynth` library on all platforms.

- As an alternative we could do some intelligent buffering to adapt in the DAW to the 64-sample raster of the external rendering regardless of the start time. I played around with that, but it also did not work out well: for example, when looping in the DAW there is no way to flush the buffer within the `FluidSynth` library and to reset the synthesizer: this is just not a use-case typical for applications of `FluidSynth` and hence it has not been provided in its API.

So we are out of luck.

But there is a workaround that helps in many situations: when your DAW allows to change the loop interval and also the play head position via its API one can *nudge all those positions onto the 64-sample raster.*

Since this heavily depends on the scripting facilities of a DAW, I have only provided a simple Lua script for the Reaper DAW in the misc subdirectory of the distribution called ForceToFluidSynthRaster.lua. When executed, it shifts the selection boundaries and the play head position onto the required raster.

It is even possible to provide some integral shift offset (e.g. when the externally rendered audio files do not start at time 0 in the DAW). This is done by setting the variable sampleOffset in the project notes to some integer value, e.g. by writing the following text:

```
sampleOffset = 20
```

Chapter 4

Regression Test

To test that the `FluidSynthPlugin` is really bit-identical to the `Pedantic-FluidSynthConverter` (and at least similar to the output of `FluidSynth`), a little test suite has been set up for checking DAW versus the command-line.

The suite assumes that command-line `FluidSynth` is installed in the search path of your operating system.

If so, a simple batch script generates — externally via the command line — audio files from three MIDI files and some simple sound font both with `PedanticFluidSynthConverter` and also `FluidSynth`. The batch script can be found in the test subdirectory and is called makeTestFiles.bat (for Windows) or makeTestFiles.sh (for MacOS and Linux).

Since there are so many DAWs available, it is hard to provide a test project for each of those. The distribution just contains a Reaper project referencing those rendered audio files in autonomous tracks (see figure 11). Adaption to other DAWs should be straightforward.

Besides the six externally rendered tracks there are three other tracks containing the MIDI file data and having a single `FluidSynthPlugin` effect converting MIDI to audio. Those instrument effects are configured with the exactly the same parameters as given in the batch file and hence correspondingly applied to the raw MIDI data.

When subtracting the rendered audio in Reaper and the externally rendered audio files from the `PedanticFluidSynthConverter`, they (almost) cancel out, because they use exactly the same scheduling of the MIDI events. This can be checked by a spectrum analyser in the master channel, which is shown in figure 12. It shows a noise floor of typically less than -100dB (also depending on the audio file bit depth).

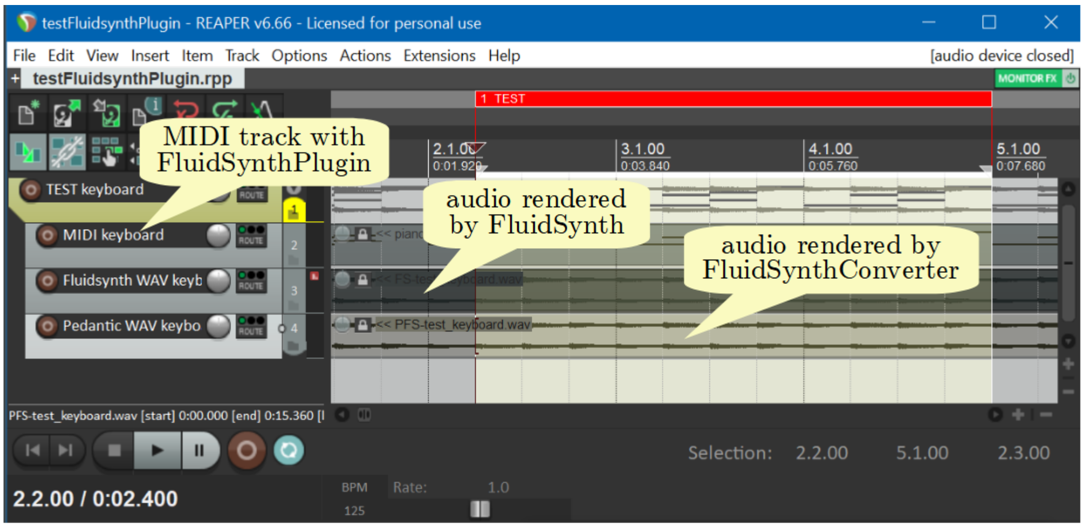

testFluidsynthPlugin - REAPER v6.66 - Licensed for personal use
File Edit View Insert Item Track Options Actions Extensions Help
[audio device closed]
testFluidsynthPlugin.rpp
MIDI track with FluidSynthPlugin
audio rendered by FluidSynth
audio rendered by FluidSynthConverter
TEST keyboard
MIDI keyboard
Fluidsynth WAV keyb
Pedantic WAV keybo
<< PFS-test_keyboard.wav
PFS-test_keyboard.wav [start] 0:00.000 [end] 0:15.360
Selection: 2.2.00 5.1.00 2.3.00
2.2.00 / 0:02.400

Figure 11: Regression Test Setup in Reaper

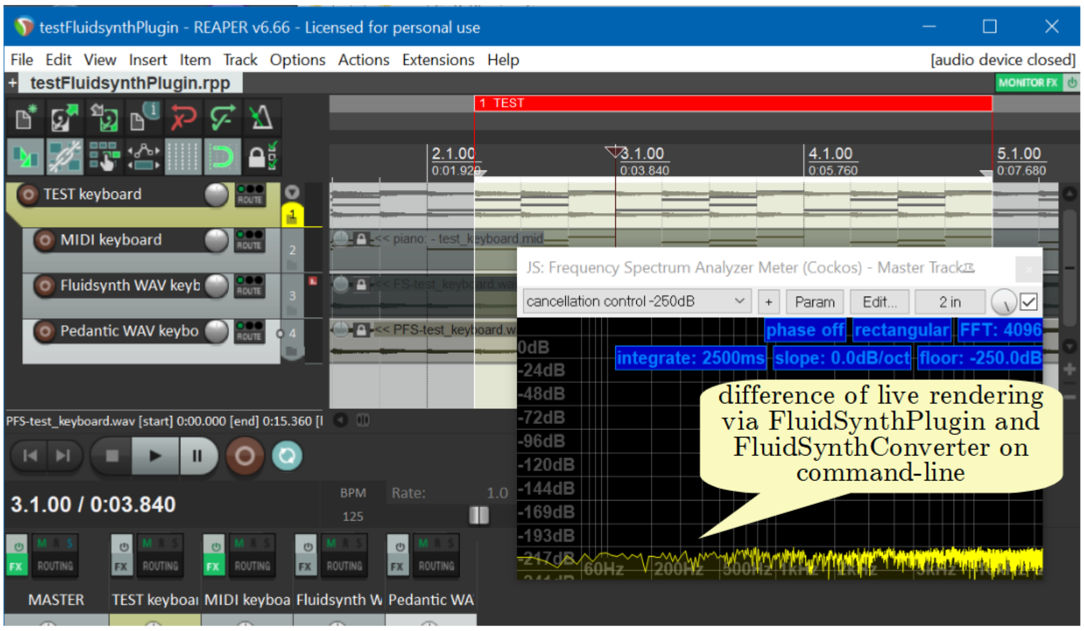

testFluidsynthPlugin - REAPER v6.66 - Licensed for personal use
File Edit View Insert Item Track Options Actions Extensions Help
[audio device closed]
testFluidsynthPlugin.rpp
JS: Frequency Spectrum Analyzer Meter (Cockos) - Master Track
cancellation control -250dB
Param
Edit...
2 in
phase off
rectangular
FFT: 4096
integrate: 2500ms
slope: 0.0dB/oct
floor: -250.0dB
difference of live rendering via FluidSynthPlugin and FluidSynthConverter on command-line
0dB
-24dB
-48dB
-72dB
-96dB
-120dB
-144dB
-169dB
-193dB
-217dB
60Hz
200Hz
3.1.00 / 0:03.840
MASTER
TEST keyboa
MIDI keyboa
Fluidsynth W
Pedantic WA

Figure 12: Example Noise Floor for Regression Test in Reaper

Chapter 5

Notes on the Implementation

5.1 Overview

The implementation of the `FluidSynthPlugin` and the `PedanticFluidSynthConverter` is done in C++. The former relies on the JUCE library [JUCE] and both — of course — use the `FluidSynth` library. The sequencer from that library is not used, because it also has some event rasterization (independent from the inevitable internal rasterization of the library).

The *bit-exact reproduction* of the `PedanticFluidSynthConverter` (as well as `FluidSynth` itself) by the `FluidSynthPlugin` in a DAW is almost achieved (see section 4), but some restrictions have to be adhered to.

The complete source code of the `FluidSynthPlugin` and the `PedanticFluidSynthConverter` is open-source for easy review and adaptation. Currently there is only a tool chain for VST3 plugins under Windows 10, VST3 and AU plugins under MacOSX and VST3 under Linux, but in principle the code is easily portable to other plugin formats or platforms.

5.2 Building the Plugins

Preliminaries

In the GIT-project of `FluidSynthPlugin` (at [FluidSynthPlugin]) there is a build file for CMAKE to build the plugins for different platforms.

Minimum prerequisites for building are:

- a clone of the GIT-project at https://github.com/prof-spock/FluidSynthPlugin
- an installation of the audio framework JUCE [JUCE] with version 5 or

later,

- some C++ compiler suite for your platform (e.g. Visual Studio, XCode, clang or gcc), and
- an installation of the build automation platform CMAKE [CMAKE] with version 3.10 or later

For documentation generation you can *optionally* install:

- a LaTeX installation — like e.g. MikTeX — (for the manual), and
- doxygen [DOXYGEN] and graphviz [GRAPHVIZ] for the internal program documentation

Doing the Build

The full build process is started via CMAKE. It is recommended to do a so-called out-of-source-build for the FluidSynth-Plugins, that means, you define some build directory where all build activity is done.

The steps are as follows:

1. Define some build directory (lets say _BUILD) and change to it.
2. Find the path of the CMakeList.txt configuration file. Adapt the file LocalConfiguration.cmake accordingly to reflect the location of LaTeXas well as the JUCE and the doxygen/graphviz installation.
3. Configure the build process via

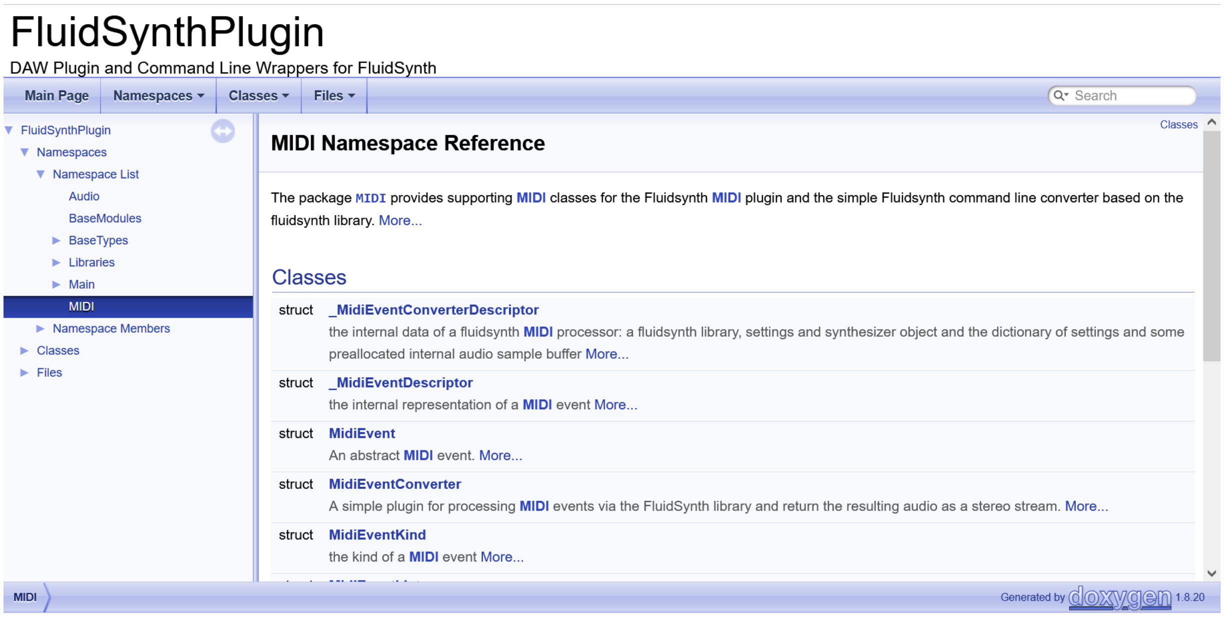
   ```
   cmake -S <pathTo>/CMakeList.txt -B . --config Release
   ```

4. Build all the plugins via

   ```
   cmake --build . --config Release
   ```

5. Install the plugins into a architecture-specific subfolder in the _DISTRIBUTION/targetPlatforms directory and install also the documentation into the _DISTRIBUTION directory via

   ```
   cmake --build . --config Release --target install
   ```


FluidSynthPlugin

DAW Plugin and Command Line Wrappers for FluidSynth

Main Page | Namespaces | Classes | Files | Search

- FluidSynthPlugin
  - Namespaces
    - Namespace List
      - Audio
      - BaseModules
      - BaseTypes
      - Libraries
      - Main
      - MIDI
    - Namespace Members
  - Classes
  - Files

Classes

MIDI Namespace Reference

The package MIDI provides supporting MIDI classes for the Fluidsynth MIDI plugin and the simple Fluidsynth command line converter based on the fluidsynth library. More...

Classes

struct _MidiEventConverterDescriptor
the internal data of a fluidsynth MIDI processor: a fluidsynth library, settings and synthesizer object and the dictionary of settings and some preallocated internal audio sample buffer More...

struct _MidiEventDescriptor
the internal representation of a MIDI event More...

struct MidiEvent
An abstract MIDI event. More...

struct MidiEventConverter
A simple plugin for processing MIDI events via the FluidSynth library and return the resulting audio as a stereo stream. More...

struct MidiEventKind
the kind of a MIDI event More...

MIDI

Generated by doxygen 1.8.20

Figure 13: Example Namespace Page for Plugin from doxygen

5.3 Internal Documentation

In the github repository there is an extensive doxygen documentation available for the inner workings of the plugins at

https://github.com/prof-spock/FluidSynthPlugin
/tree/master/internalDocumentation/html

with entry point

https://github.com/prof-spock/FluidSynthPlugin
/tree/master/internalDocumentation/html/index.html.

Every public and private feature of all classes and data types is documented and can be analyzed in an HTML browser. Figure 13 gives an impression how such an HTML page looks like for the namespaces in FluidSynth-Plugins.

If you want to regenerate this documentation from the code, you need an installation of doxygen [DOXYGEN] and ideally also graphviz [GRAPHVIZ] on your computer. If you have that available, the generation can be done via the CMAKE chain as target doxygenDocumentation in the build directory:

```
cmake --build . --config Release \
      --target internalDocumentation
```

If the command completes, the documentation in the internalDocumentation subdirectory of the project is updated.

Target Name	**Description**
`documentation`	the complete project documentation
`<-- internalDocumentation`	the HTML doxygen documentation for the code
`<-- pdfDocumentation`	the PDF manual for the plugins
`FluidSynthFileConverter`	the program for the command line `PedanticFluidSynthConverter`
`FluidSynthPlugins`	the static libraries for the `FluidSynthPlugin`
`<-- FluidSynthPlugins_Effect`	the static effect library for the `FluidSynthPlugin`
`<-- FluidSynthPlugins_VST`	the VST3 library for the `FluidSynthPlugin`
`<-- FluidSynthPlugins_AU`	the AU libraries for the `FluidSynthPlugin` (only on MacOSX)
`SupportLibraries`	the static libraries supporting the effects
`<-- JuceFramework`	the static library with utility classes from the JUCE framework

Figure 14: Available Build Targets for CMAKE

To trigger regeneration, it suffices to delete the file internalDocumentation/html/index.html.

5.4 Available Build Targets

Figure 14 shows the available CMAKE targets. They can be used as

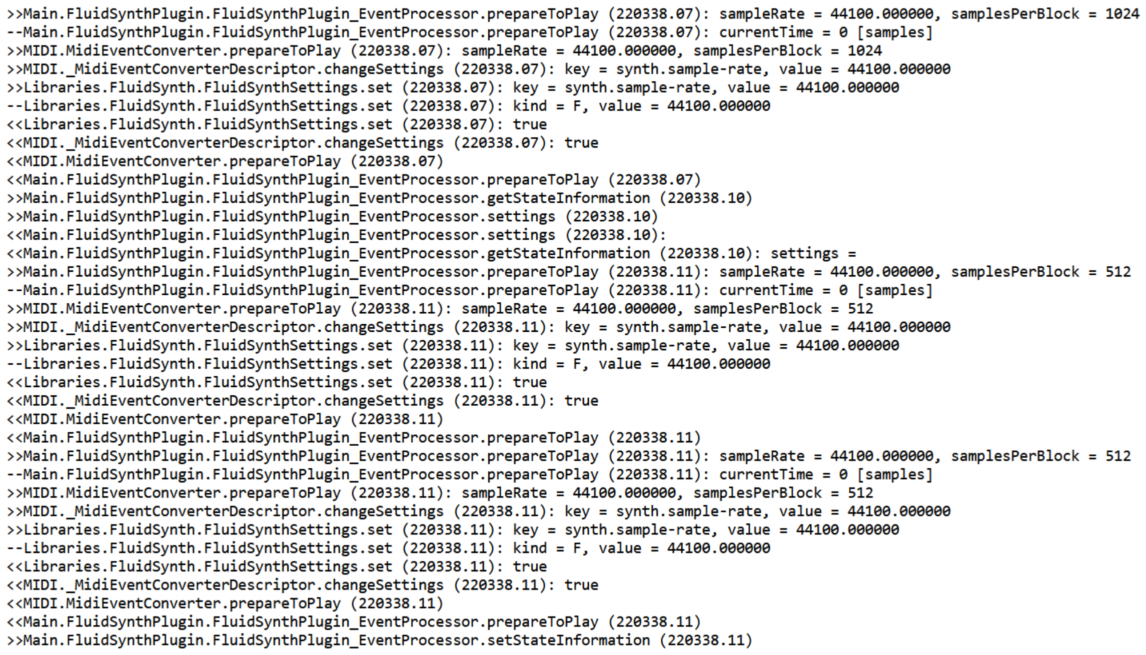
```
cmake --build . --config Release --target XXX
```

where XXX is the target name.

5.5 Debugging

For debugging purposes, for both the `FluidSynthPlugin` and the `PedanticFluidSynthConverter` a debugging version is also available that does an extensive entry-exit-logging into the temp directory. Note that this debugging slows down processing extremely and produces large log files, but it helps to understand problems in case of errors. Figure 15 shows how a logging file looks like.

Every non-trivial function is logged there at least twice with timestamps: "»" indicates the entry of that function (possibly with informationon the



```
>>Main.FluidSynthPlugin.FluidSynthPlugin_EventProcessor.prepareToPlay (220338.07): sampleRate = 44100.000000, samplesPerBlock = 1024
--Main.FluidSynthPlugin.FluidSynthPlugin_EventProcessor.prepareToPlay (220338.07): currentTime = 0 [samples]
>>MIDI.MidiEventConverter.prepareToPlay (220338.07): sampleRate = 44100.000000, samplesPerBlock = 1024
>>MIDI._MidiEventConverterDescriptor.changeSettings (220338.07): key = synth.sample-rate, value = 44100.000000
>>Libraries.FluidSynth.FluidSynthSettings.set (220338.07): key = synth.sample-rate, value = 44100.000000
--Libraries.FluidSynth.FluidSynthSettings.set (220338.07): kind = F, value = 44100.000000
<<Libraries.FluidSynth.FluidSynthSettings.set (220338.07): true
<<MIDI._MidiEventConverterDescriptor.changeSettings (220338.07): true
<<MIDI.MidiEventConverter.prepareToPlay (220338.07)
<<Main.FluidSynthPlugin.FluidSynthPlugin_EventProcessor.prepareToPlay (220338.07)
>>Main.FluidSynthPlugin.FluidSynthPlugin_EventProcessor.getStateInformation (220338.10)
>>Main.FluidSynthPlugin.FluidSynthPlugin_EventProcessor.settings (220338.10)
<<Main.FluidSynthPlugin.FluidSynthPlugin_EventProcessor.settings (220338.10):
<<Main.FluidSynthPlugin.FluidSynthPlugin_EventProcessor.getStateInformation (220338.10): settings =
>>Main.FluidSynthPlugin.FluidSynthPlugin_EventProcessor.prepareToPlay (220338.11): sampleRate = 44100.000000, samplesPerBlock = 512
--Main.FluidSynthPlugin.FluidSynthPlugin_EventProcessor.prepareToPlay (220338.11): currentTime = 0 [samples]
>>MIDI.MidiEventConverter.prepareToPlay (220338.11): sampleRate = 44100.000000, samplesPerBlock = 512
>>MIDI._MidiEventConverterDescriptor.changeSettings (220338.11): key = synth.sample-rate, value = 44100.000000
>>Libraries.FluidSynth.FluidSynthSettings.set (220338.11): key = synth.sample-rate, value = 44100.000000
--Libraries.FluidSynth.FluidSynthSettings.set (220338.11): kind = F, value = 44100.000000
<<Libraries.FluidSynth.FluidSynthSettings.set (220338.11): true
<<MIDI._MidiEventConverterDescriptor.changeSettings (220338.11): true
<<MIDI.MidiEventConverter.prepareToPlay (220338.11)
<<Main.FluidSynthPlugin.FluidSynthPlugin_EventProcessor.prepareToPlay (220338.11)
>>Main.FluidSynthPlugin.FluidSynthPlugin_EventProcessor.prepareToPlay (220338.11): sampleRate = 44100.000000, samplesPerBlock = 512
--Main.FluidSynthPlugin.FluidSynthPlugin_EventProcessor.prepareToPlay (220338.11): currentTime = 0 [samples]
>>MIDI.MidiEventConverter.prepareToPlay (220338.11): sampleRate = 44100.000000, samplesPerBlock = 512
>>MIDI._MidiEventConverterDescriptor.changeSettings (220338.11): key = synth.sample-rate, value = 44100.000000
>>Libraries.FluidSynth.FluidSynthSettings.set (220338.11): key = synth.sample-rate, value = 44100.000000
--Libraries.FluidSynth.FluidSynthSettings.set (220338.11): kind = F, value = 44100.000000
<<Libraries.FluidSynth.FluidSynthSettings.set (220338.11): true
<<MIDI._MidiEventConverterDescriptor.changeSettings (220338.11): true
<<MIDI.MidiEventConverter.prepareToPlay (220338.11)
<<Main.FluidSynthPlugin.FluidSynthPlugin_EventProcessor.prepareToPlay (220338.11)
>>Main.FluidSynthPlugin.FluidSynthPlugin_EventProcessor.setStateInformation (220338.11)
```

Figure 15: Example for Logging File

argument values), "«" the exit of that function (possibly with the return value) and "--" indicates some intermediate information during the function processing. The logging data is hierarchical, hence you can see the function call structure in this file precisely.

All logging files go to the directory specified by the `temp` environment variable.

Bibliography

[CMAKE] Kitware, Inc.
CMAKE Build Automation System.
http://cmake.org

[DOXYGEN] Dimitri van Heesch.
Doxygen - Generate Documentation from Source Code.
https://www.doxygen.nl

[FluidSynthDOC] Tom Moebert et al.:
FluidSynth.
http://www.fluidsynth.org

[FluidSynthSettings] Tom Moebert et al.:
FluidSynth - Synthesizer Settings Documentation.
https://www.fluidsynth.org/api/settings_synth.html

[FluidSynthVST] Alexey Zhelezov.
FluidSynth VST Plugin.
https://github.com/AZSlow3/FluidSynthVST

[FluidSynthPlugin] Thomas Tensi.
FluidSynth-Plugins
https://github.com/prof-spock/FluidSynthPlugin

[GRAPHVIZ] Ellson, John; Gansner, Emden; Hu, Yifan; North, Stephen et al.:
Graphviz - Open Source Graph Visualization Software.
https://graphviz.org/

[JUCE] Raw Material Software Limited.
JUCE Audio Framework.
https://www.juce.com

[JuicySFPlugin] Alex and Jamie Birch.
Juicy SF Plugin.
https://github.com/Birch-san/juicysfplugin

[LTBVC] Thomas Tensi.
LilypondToBandVideoConverter — Converter from Written Music to Notation Videos.
https://github.com/prof-spock/LilypondToBandVideoConverter

[REAPER] Cockos Incorporated.
Reaper Digital Audio Workstation.
https://www.reaper.fm

[VCCLib] Microsoft.
Visual C++ Redistributable.
https://learn.microsoft.com/cpp/windows/latest-supported-vc-redist