LilypondToBandVideoConverter - Automated Generation of Notation Videos with Backing Tracks

Dr. Thomas Tensi

January 4, 2019

Contents

1. Introduction

1.1 Overview

The *LilypondToBandVideoConverter* is an application built from several python scripts that orchestrate standard command-line tools to convert a music piece (a song) written in the lilypond notation to

- a PDF score of the whole song,
- several PDF voice extracts,
- a MIDI file with all voices (with some preprocessing applied for humanization),
- audio mix files with several subsets of voices (specified by configuration), and
- video files for several output targets visualizing the score notation pages and having the mixes as mutually selectable audio tracks as backing tracks.

The central aim is to finally have a video file with several audio tracks containing mixes of different voice subsets to be used as selectable backing tracks. The video itself shows a score with "pages" turned at the right time and an indication of the current measure as a subtitle.

So one might have a score video to be displayed on some device (like a tablet) that synchronously plays, for example, a backing track without vocals, guitar and keyboard, but with bass and drums. Hence a (partial) band can play the missing voices live (reading the score) and have the other voices coming from the backing track.

For processing a song one must have

- a lilypond include file with the score information containing specific lilypond identifiers, and
- a configuration file giving details like the voices occuring in the song, their associated midi instrument, target audio volume, list of mutable voices for the audio tracks etc.

Based on those files the python scripts – together with some open-source command-line software like ffmpeg – produce all the target files either incrementally or altogether.

In principle, all this could also be done with standard lilypond files using command line tools. But the LilypondToBandVideoConverter application automates a lot of that: based on data given in a song-dependent configuration file plus the lilypond fragment file for the notes of the voices, it adds boilerplate lilypond code, parametrizes the tool chain and calls the necessary programs automatically. And the process is completely unattended: once your configuration and lilypond notation files are set up the process runs on its own. Additionally the audio generation can be tweaked by defining midi humanization styles and command chains ("sound styles") for the audio postprocessing.

This document assumes that you have an adequate knowledge of the following underlying software:

lilypond:
for the notation specification,

sox:
for postprocessing the audio files

1.2 Outline of this Document

This document will present how to setup a lilypond fragment file and an associated configuration file for processing with LilypondToBandVideoConverter.

- Chapter 2 describes the installation requirements and defines some terminology used in this document.
- Chapter 4 tells how the (command line) program is used and what kind of processing phases are available. There is also some dependency between the artifacts of the phases that is presented there.
- Chapter 5 gives an overview of the syntax of a LilypondToBandVideoConverter configuration file. It consists of key-value-pairs; the keys are identifiers, but the values may be a bit more complicated.
- Chapter 6 tells how the lilypond fragment file should look. Of course, the syntax is given by the lilypond program, but — since we have fragments with external boilerplate code — we discuss what kind of information must be provided in those files.
- Chapter 7 discusses in detail each configuration file variable needed by going through all the processing phases in sequence.

- Chapter 8 gives an example by showing all the lilypond macros and all required configuration settings for a simple two-verse blues song with three instruments. It shows that some initial effort is needed, but normally you can reuse things once you have understood how to make it work.
- Because things will certainly go wrong some time, chapter 9 gives some hints on how to trace the problem.
- Appendix A gives an overview table of all configuration file commands and appendix C shows the used bibliography references.

2. Preliminaries

2.1 Requirements

All the scripts are written in python and can be installed as a python package. The package requires either Python 2.7 or Python 3.3 or later and relies on the python package mutagen.

Additionally the following software must be available:

lilypond:
for generating the score pdf, voice extract pdfs, the raw midi file and the score images used in the video files [LILY],

ffmpeg:
for video generation and video postprocessing [FFMPEG],

fluidsynth:
for generation of voice audio files from a midi file [FLUID] plus some soundfont (e.g. FluidR3_GM.sf3 at [SOUNDFONT]), and

sox:
for instrument-specific postprocessing of audio files for the target mix files as well as the mixdown [SOX]

Both "fluidsynth" and "sox" may be replaced by other software that does similar file transformations. "fluidsynth" can be substituted by a command-line program transforming MIDI to WAV, "sox" by one doing command-line audio processing on WAV files. In both cases the corresponding configuration has to be adapted accordingly.

The following software is optional:

aac:
an AAC-encoder for the final audio mix file compression (for example [AAC]), and

mp4box:
the MP4 container packaging software mp4box [MP4BOX]

The location of all those commands as well as a few other settings has to be defined in a global configuration file for the LilypondToBandVideoConverter (cf. overall configuration file syntax)

2.2 Installation

The program is available via the Python platform PyPi, the Python package index.

```
pip install lilypondToBVC
```

Once installed the program is ready for use. Make sure that the scripts directory of python is in the path for executables on your platform.

3. Terminology

Because the different programs do not completely agree in their terminology, a single terminology defined here is used throughout the document. Appendix B gives a detailed description of the all terms used in this document.

The most important terms are:

voice:
: a polyphonic part of a composition belonging to a single instrument to be notated in one or several musical staffs

song:
: a collection of several parallel voices forming a musical piece

album:
: a collection of several related songs (for example, related by year, artist, etc.)

audio track:
: the audio rendering of a subset of all song voices (typically within the final notation video)

4. Usage

The LilypondToBandVideoConverter is a commandline program with the following syntax:

```
lilypondToBVC [-h] [-k] --phases PHASELIST [--voices VOICELIST]
              configurationFilePath
```

The options have the following meaning:

-h
: makes the program show all the commandline options and exit

-k
: force the program to keep intermediate files

–phases PHASELIST
: specifies the processing phases or combination of processing phases to be applied; is a slash-separated identifier list from the set {all, preprocess, postprocess, extract, score, midi, silentvideo, rawaudio, refinedaudio, mixdown, finalvideo}

–voices VOICELIST
: gives the slash-separated list of voices where current phase should be done on (for example, only on vocals and on drums); those voice names should be a subset of the list of voices given in the configuration file and in the associated lilypond fragment file; this option is optional: when it is not given, all voices are used; only applies to phases “extract”, “rawaudio” and “refinedaudio”

configurationFilePath
: gives the path to the configuration file specifying all information about the song to be processed

The several processing phases of LilypondToBandVideoConverter produce the several outputs incrementally. Those phases have the following meanings:

extract:
: generates PDF notation files for single voices as extracts (might use compacted versions if specified),

score:
: generates a single PDF file containing all voices as a score,

midi:
: generates a MIDI file containing all voices with specified instruments, pan positions and volumes,

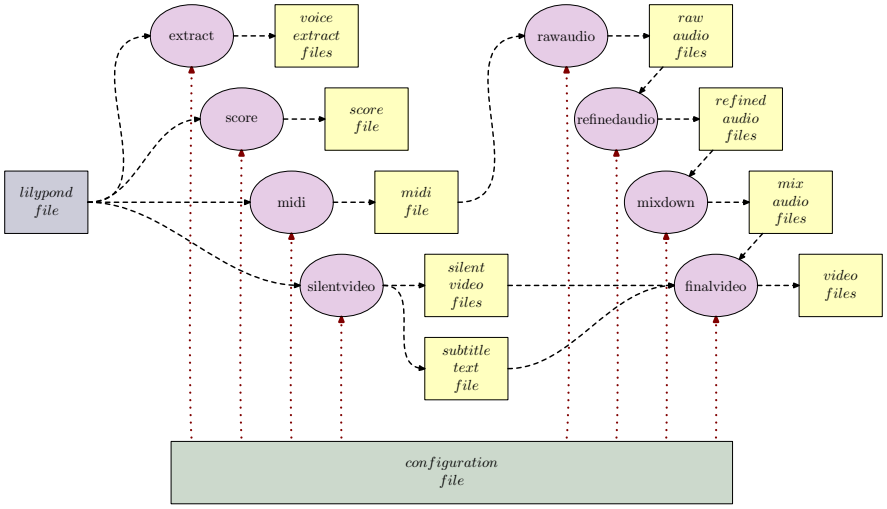

Figure 1: Dependencies between Generation Phases

silentvideo:
generates (intermediate) silent videos containing the score pages for several output video file kinds (with configurable resolution and size),

rawaudio:
generates unprocessed (intermediate) audio files for all the instrument voices from the midi tracks,

refinedaudio:
generates (intermediate) audio files for all the instrument voices with additional sound processing applied,

mixdown:
generates final compressed audio files with submixes of all instruments voices based on the refined audio files with specified volume balance (where the submix variants are configurable), and

finalvideo:
generates a final video file with all submixes as selectable audio tracks and with a measure indication as subtitle

Of course, those phases are not independent. Several phases rely on results produced by other phases. Figure 1 shows how the phases depend on each other. The files (in yellow) are generated by the phases (in magenta), the configuration file (in green) and the lilypond fragment file (in blue) are the only manual inputs into the processing chain.

For example, the phase rawaudio needs a midi file as input containing all voices to be rendered as audio files. When using combining phases (see

below) or when specifying several phases for a single run of the LilypondToBandVideoConverter application, the phases are processed in a correct order, but when doing a manual selection of phases, you have to make sure that the dependencies given are obeyed.

In the following we shall use the color coding for the files as given in figure 1: parts from the configuration file have a green background, parts from the lilypond fragment file have a blue background.

There are also some combining phase available as follows:

preprocess:
: combining all the phases extract, score, midi and silentvideo for generation of voice extract PDFs and score PDF, MIDI file as well the silent videos for all video file kinds

postprocess:
: combining all the phases rawaudio, refinedaudio, mixdown and finalvideo for generation of the intermediate raw and refined WAV files, the submixes as compressed audios and the final videos for all video file kinds

all:
: full processing via phase groups preprocess and postprocess

So for example

```
lilypondToBVC --phases voice/score
              --voices vocals/strings/drums config.txt
```

will generate the voice extracts for vocals, strings and drums as well as a song score with those three voices specified in file config.txt. The vertical order within the score as well as other layout parameters are given by the order of voice descriptions and specific variables in the configuration file.

5. Configuration File Overview

Variables controlling the song processing have to be defined in the configuration file for a song. The name of this file is given as a mandatory parameter for the application.

Note that typically there is not a single configuration file, but several. Often a song configuration file includes others with global definitions (like, for example, defining the location of the ffmpeg command or some style of audio postprocessing).

Although there is some internal program logic separating the variables into different domains for global setup variables, album related variables and song variables, this is somewhat academical: a variable definition can be given at any place and a later definition overrides a previous one.

5.1 Configuration File Location

The configuration file(s) are searched for in the following locations in the given order:

- the current directory
- the directory /.ltbvc within the user's home directory
- the directory config and ../config relative to the directory of the python program files

5.2 Configuration File Syntax

Each configuration file has a simple line-oriented syntax as follows:

- Leading and trailing whitespace in a line is ignored. Other whitespace is only interpreted as token separator.
- A line starting with a comment marker "--" or completely empty is ignored.
- A line ending with a continuation marker "␣\" is combined with the following line.
- Each relevant line starts with an identifier followed by an equal sign and the associated value. The associated value may be an integer, a decimal, a boolean or a string. By this assignment the value is associated with

the variable given by the identifier. A subsequent assignment to the same variable will replace that value.

- An identifier is a sequence of lower- and uppercase letters or underscores.
- One may define such variables arbitrarily.
- An integer literal is a digit sequence, a decimal value is a digit sequence with at most one decimal point, a boolean value is either the string "true" or "false" and a string value is a character sequence enclosed by double quotes. Two double quotes within a string are interpreted as a double quote character.
- When a variable identifier occurs on the right hand side of an assignment, it is replaced by its associated value. If there is none, this is an error. The processing is strictly sequential: the use of an identifier must come after its definition. It is okay to use an identifier in its own redefinition.
- A sequence of adjacent string literals or variables with string contents are concatenated into a single string value.
- A line starting with "INCLUDE" followed by a string specifies the name of a file to be included in place.
- As a convention sets have comma-separated string values and maps are strings with a leading and trailing brace and key and values separated by a colon. White space within those strings is not significant except when it is itself part of a value string enclosed in single quotation marks.
- It is helpful to distinguish auxiliary variables from those used by the program. In this document we prefix auxiliary variables with an underscore (but any convention — even none — is fine).

Assume for an example the following definitions in two files "test.text" and "config.txt":

```
-- test.txt file to be included elsewhere
voiceNameList = "vocals, guitar, drums"
humanizedVoiceNameSet = "vocals"
_initialTempo = "90"
year = 2017
```

```
-- config.txt file including test file
INCLUDE "test.txt"
voiceNameList = "vocals, guitar"
humanizedVoiceNameSet = humanizedVoiceNameSet ", drums"
measureToTempoMap = "{ 1 : " _initialTempo ", 20 : 67 }"
```

leads to the following overall variable settings:

```
_initialTempo          = "90"
year                   = 2017
voiceNameList          = "vocals, guitar"
humanizedVoiceNameSet = "vocals, drums"
measureToTempoMap      = "{ 1 : 90, 20 : 67 }"
```

6. Lilypond Fragment File Overview

The lilypond fragment file used for a song contains lilypond macros. At least there must be definitions for the following items:

keyAndTime:
tells the key and time of the song and assumes that this applies to all voices

«voice»XXX:
for each voice given in the configuration file containing the musical expression to be used in an extract, in a score, in the midi file or in the video; here "XXX" depends on the target, so you might have different macros for a voice for the different targets it occurs in (extract, score, midi, video).

The names of all voices are given by the configuration variable voiceNameList. Because lilypond only allows letters in macro names, those voice names must consist of small and capital letters only (no blanks, no digits, no special characters!) and they are case sensitive. And they should not clash with predefined lilypond macros [1].

The above looks quite complicated because you need macros for each voice and each processing phase. But often you will reuse lilypond macros and typically the MIDI macro «voice»Midi is the same as the score macro «voice» only with *all repetitions unfolded*. You do not have to do this by yourself: for midi output this unfolding is done by the generator.

There is even another automatism: if the generator looks for some voice macro with some extension it also accepts the plain macro for the voice (if available). For example, if the macro guitarMidi cannot be found, the generator looks for the macro guitar and automatically applies necessary lilypond transformations (like unfolding repeats).

Some variables in the configuration file make other lilypond macros "mandatory". The table in figure 2 gives the configuration variable, the corresponding lilypond macro(s) and a short description. The dependency is not strict, because some default settings are done, but in general the logic described in the figure is a good orientation. Video voice names are not specified in a single variable, but via video target and video file kind definitions (see section 7.4.1.3).

For example, assume we have three voices in the song called "vocals", "drums" and "guitar". We also assume that we shall have all voices in the midi file,

[1] Like drums, but because this is a common voice name it is automatically mapped to myDrums by the generator.

Config. Variable	Description	Lilypond Var.
audioVoiceNameSet	for each voice given in the set the lilypond macro gives the musical expression for the voice to be rendered as an audio file with the voice name	«voice»Midi
extractVoiceNameSet	for each voice given in the list the lilypond macro gives the musical expression for a voice to be rendered in the corresponding voice extract	«voice»Extract
midiVoiceNameList	for each voice given in the list the lilypond macro gives the musical expression for the voice to be rendered in the *midi file* and rendered as an audio file with the voice name; the list is the order of the voices in the file	«voice»Midi
scoreVoiceNameList	for each voice given in the list the lilypond macro gives the musical expression for the voice to be rendered in the *midi file*, the list is the order of the voices in the score from top to bottom	«voice»Score

Figure 2: Dependency of Lilypond Macros on Configuration Variables

vocals in an extract, drums and guitar in the score and vocals and guitar in the video.

So the configuration file for the song contains the following definitions:

```
...
voiceNameList       = "vocals, drums, guitar"
extractVoiceNameSet = "vocals"
scoreVoiceNameList  = "guitar, drums"
midiVoiceNameList   = "vocals, guitar, drums"
...
```

Note that the midiVoiceNameList could be omitted, because the default is to use the voices from the overall voice list voiceNameList and the "wrong" order of voices does not really matter in the midi file. The audio variable audioVoiceNameSet has been omitted: it defaults to the setting of midiVoiceNameList, so we have audio for "vocals", "guitar" and "drums" (that means, all voices).

For the given configuration we must have the following macros in the lilypond fragment file:

```
keyAndTime = {...}

vocalsExtract = {...}
vocalsScore   = {...}
vocalsMidi    = {...}

guitarScore   = {...}
guitarMidi    = {...}
guitarVideo   = {...}

myDrumsScore  = {...}
myDrumsMidi   = {...}
```

Again some simplification is possible: when some global macros like guitar is introduced, the associated variants can be omitted.

6.1 Chords

Because the software is used in a band context, chord symbols may also be used. Chords may depend on voice and very often depend on the processing target, because the voice formatting may be different per target.

The configuration file variable responsible for chords is voiceNameToChordsMap and tells where chords are shown and for which voices.

All voices with chords are mentioned as keys and mapped onto a slash separated list of single character abbreviations for the targets. We have "e" for the extract, "s" for the score and "v" for the video. There are no chords for the midi file.

So for the configuration file line

```
voiceNameToChordsMap = "{ vocals: v/s, guitar: e }"
```

the chords are shown for the vocals in video and score and for guitar in its extract. This means the lilypond fragment file must contain the following definitions in \chordmode:

```
guitarChordsExtract = {...}
vocalsChordsScore   = {...}
vocalsChordsVideo   = {...}
```

Again there is a default: when some chord macro is missing, either the plain chords macro for the voice or even the chords for all voices are used.

So for example, for a missing guitarChordsExtract the search is first for guitarChords and finally for allChords (the latter as a catch-all since chords is a keyword in lilypond).

6.2 Lyrics

Also lyrics may be attached to voices. Lyrics may occur in voice extracts, in the score and in the video. The difference to chords is that multiple lyrics lines (for example, for stanzas) may be attached to a single voice, hence we need an additional count information.

It is assumed that each lyrics line is always valid for all the notes in the voice, hence you have to provide appropriate padding (at least leading padding).

The syntax is similar to chords, hence we have a voiceNameToLyricsMap, but it also contains a count of parallel lyrics lines directly following the target letter ("e" for the extract, "s" for the score and "v" for the video).

So for the configuration file line

```
voiceNameToLyricsMap = "{ vocals: e2/s2/v, bgVocals: e3 }"
```

the lyrics are shown for the vocals in extract, video and score and for the background vocals only in its extract. The lyrics line macros have capital letters as suffices (A, B, ...) and hence are confined to 26 parallel lines per voice.

This means the lilypond fragment file must contain the following definitions in \lyricmode:

```
vocalsLyricsExtractA = {...}
vocalsLyricsExtractB = {...}
vocalsLyricsScoreA   = {...}
vocalsLyricsScoreB   = {...}
vocalsLyricsVideoA   = {...}

bgVocalsLyricsExtractA = {...}
bgVocalsLyricsExtractB = {...}
bgVocalsLyricsExtractC = {...}
```

Again there is a default: when some lyrics macro is missing, the macro for the voice without the target (but with the appropriate suffix) is used. So for example, for a missing vocalsLyricsScoreB an existing vocalsLyricsB is used. Additionally for the first line the suffix may be totally omitted, so vocalsLyricsScoreA can be replaced by vocalsLyricsScore or even vocalsLyrics.

6.3 Things Not to Put in the Lilypond Fragment File

Because the different phases add their own boilerplate code, the following lilypond code must not occur in the lilypond fragment file:

- a \score block, and
- staff definitions

The following should not occur in the fragment, unless you want to override the presets from the program:

- a \header block,
- a \paper block, and
- a setting of the global-staff-size

Note that settings overriding presets above might interfere with some phases: e.g. the videos use their own paper and resolution settings and those would be shadowed by conflicting definitions in the fragment.

7. Configuration File Settings

In the following we show all the settings of the configuration file in detail and what to put in an associated lilypond music fragment file.

In principle one only needs a *single* configuration file and a single lilypond fragment file. For systematic reasons the information can be divided for didactic reasons and must then be combined into a single configuration file by INCLUDE statements.

7.1 Overall Configuration

In this section the configuration file settings are discussed that define the locations of programs and files used. Note that paths use the Unix forward slash as a separator. If a relative path is used, it is relative to the current directory where the program call is made.

Some variables define the program locations and global program parameters and are shown in figure 3. For example, ffmpegCommand tells the path of the ffmpeg command (you wouldn't have guessed that, would you?).

Three entries are special: aacCommandLine and soxCommandLinePrefix.

- The aac command line specifies the complete line for an aac encoding command with ${infile} and ${outfile} as placeholders for the input and output file name. If empty, ffmpeg is used for aac encoding.

- The audio refinement command line specifies the complete line for refining audio files with ${infile}, ${outfile} and ${commands} as placeholders for the input and output file name and the refinement commands from a sound style. If empty the variable soxCommandLinePrefix *must* be set.

- The sox command line prefix specifies the prefix for the audio refinement command sox with command name and global options (like buffering) for refinement, audio shifting and mixdown. If this value is not set, slow internal routines will be used instead for audio shifting and mixdown and also the variable audioRefinementCommandLine *must* be set.

So an example setting in the configuration file for the global configuration variables could look like that:

Variable	Description	Example
aacCommandLine	aac encoder command line with parameters for input (${infile}) and output (${outfile}) (optional, if not defined ffmpeg is used for aac encoding)	"/pathto/qaac -V100 -i ${infile} -o ${outfile}"
audioRefinementCommandLine	audio refinement command line with parameters for input (${infile}) and output (${outfile}); this commandline is optional when soxCommandLinePrefix is set, otherwise it is mandatory	"/pathto/sox ${infile} ${outfile} ${commands}"
ffmpegCommand	location of ffmpeg command	"/pathto/ffmpeg"
lilypondCommand	location of lilypond command	"/pathto/lilypond"
lilypondVersion	the version string for lilypond	"2.18.2"
midiToWavRendering-CommandLine	command line for rendering command from MIDI file to WAV audio file (typically "fluidsynth")	"/pathto/fluidsynth"
mp4boxCommand	location of mp4box command	"/pathto/mp4box"
soxCommandLinePrefix	command line prefix for sox audio filter with global options (like buffering or multithreading settings); if missing, (slow) internal routines will be used for audio shifting and mixdown	"/pathto/sox"

Figure 3: Global Configuration Variables for Programs

```
aacCommandLine     = "/usr/local/qaac -V100 -i $1 -o $2"
ffmpegCommand      = "/usr/local/ffmpeg"
lilypondCommand    = "/usr/local/lilypond"
lilypondVersion    = "2.18.2"
midiToWavRenderingCommandLine = \
    "/usr/local/fluidsynth ${infile} ${outfile}"
soxCommandLinePrefix = \
    "/usr/local/sox --buffer 100000 --multi-threaded"
```

Note that we use sox for audio refinement. Other variables shown in figure 4 define file and path locations. Very important is the path where the logging file ltvbc.log is located: sometimes it is the only way to find out what went wrong.

Temporary files go to intermediateFileDirectoryPath. By default, all temp files go to the current directory and the phase-internal files are deleted at the end of a phase (but you can prevent that, see 9).

An example setting in the configuration file for file path configuration variables could look like that:

Variable	Description	Example
intermediateFile-DirectoryPath	path of directory where intermediate files go that are either used for processing within a phase or as information between phases	"temp"
loggingFilePath	path of file containing the processing log	"/pathto/ltbvc.log"
targetDirectoryPath	path of directory where all generated files go (except for audio and video files)	"generated"
tempAudioDirectoryPath	path of directory for temporary audio files	"/pathto/audiofiles"
tempLilypondFilePath	path of temporary lilypond file	"temp.ly"

Figure 4: Global Configuration Variables for File Paths

Variable	Description	Example
albumName	album for song group (embedded as “album” in audio and video files)	"Best of Fredo"
artistName	artist of that song group (embedded as “artist” and “album artist” in audio and video files)	"Fredo"

Figure 5: Song Group Related Configuration File Variables

```
intermediateFileDirectoryPath = "temp"
loggingFilePath = "/var/logs/ltbvc.log"
targetDirectoryPath = "generated"
tempAudioDirectoryPath = "~/ltbvc_audiofilesdir"
tempLilypondFilePath = "temp.ly"
```

7.2 Song Group Configuration

Very often several songs are combined into a song group, for example, into an album.

A song group is characterized by two parameters in the configuration file as shown in figure 5.

7.3 Song Configuration

The song is characterized by some very simple parameters in the configuration file shown in figure 6. The most important variable is fileNamePrefix because it is used in the file names of the generated files; all the other variables may

Variable	Description	Example
composerText	composer text to be shown in voice extracts and score	"arranged by Fred, 2017"
fileNamePrefix	file name prefix used for all generated files for this song	"wonderful_song"
includeFilePath	path for the music include file containing all fragments for lilypond processing; if unset, defaults to fileNamePrefix plus "-music.ly"	"wonderful_song-music.ly"
keepIntermediateFiles	boolean telling whether temporary files are kept	False
measureToTempoMap	map defining the tempo for measure in bpm until another tempo setting is given; the time signature as a fraction may be appended after a vertical bar (4/4 is default)	"{ 1 : 60|3/4, 20 : 100 }"
trackNumber	track number within album	22
title	human visible title of song used as tag in the target audio file and as header line in the notation files	"Wonderful Song"
year	year of arrangement	2017

Figure 6: Song Related Configuration File Variables

be missing and are set to some reasonable default.

The lilypond include file containing all fragments can be specified via includeFilePath, but if unset defaults to fileNamePrefix plus "-music.ly".

7.4 Configuration of the Processing Phases

7.4.1 Preprocessing Phases

All preprocessing phases rely on the configuration and the lilypond fragment file, while the postprocessing phase start from the generated midi file and the silent videos.

In each preprocessing phase some boilerplate lilypond file is generated including the lilypond fragment file with the music and puts it through the notation typesetter lilypond.

Figure 7 shows the connection between the inputs and the outputs for the phases. Both lilypond fragment file and configuration file serve as manual input into the processing chain, the other files are generated.

For the "extract" and "score" phases this is all there is to do, but the "midi" and "silentvideo" phases do further processing:

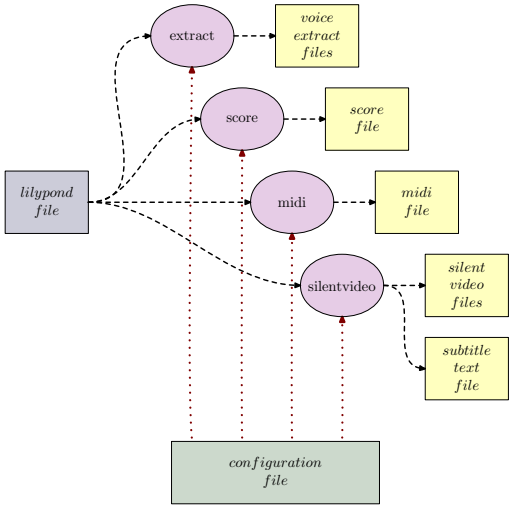

Figure 7: Information Flow for the Preprocessing Phases

midi:
the midi file produced by lilypond has humanization applied to the voices, and

silentvideo:
the image files produced by lilypond are combined into a correctly timed video and a subtitle file in SRT format is produced

If you *really* want to fiddle with lilypond, the processing phase is provided as the lilypond macro ltbvcProcessingPhase with values "extract", "score", "midi" or "silentvideo". You can use that for conditional processing, layout changes etc., because the fragment file is included into the boilerplate file at a very late position. Be warned that the whole generation might fail, because the generator assumes a simple-structured lilypond include file.

7.4.1.1 Notation Generation: "extract" and "score" Phase

Preliminaries

The central settings in the configuration file define the characteristics of the voices. Each voice is given by its name (an identifier) in the variable voiceNameList.

Note that the order in the voice name list is significant, because later on variable in other phases rely on that order. For example, the audio volumes for phase "mixdown" in variable audioVolumeList have the same order as the voiceNameList. So the lines

```
voiceNameList   = "vocals, guitar, drums"
audioVolumeList = "   0.9,    0.7,   1.0"
```

associate "vocals" with volume 0.9, "guitar" with volume 0.7 etc. A simple table logic: and it is fine to align the data in different entries with blanks.

The staff layout is specified by several variables that map voice names into several kinds of staff-related layout information. Because this might be phase-dependent, another mapping layer is added, mapping the phase onto the voice name to staff info map.

phaseAndVoiceNameToStaffListMap tells the staff to use for the voice in extract, score and video for a given processing phase. Default is "Staff", special staffs like "DrumStaff" may be defined in the map. The mapping goes from phase name to a map from voice name to staff names.

To reduce the mental complexity we first define a map from voice name to staff by the following configuration file lines

```
_voiceNameToStaffListMap = \
    "{ drums      : DrumStaff,"    \
     " keyboard   : PianoStaff,"   \
     " percussion : DrumStaff }"
```

that are reused in the mapping from phase name

```
phaseAndVoiceNameToStaffListMap = \
    "{ extract : " _voiceNameToStaffListMap ","  \
     " midi    : " _voiceNameToStaffListMap ","  \
     " score   : " _voiceNameToStaffListMap ","  \
     " video   : " _voiceNameToStaffListMap "}"
```

Very often the different phases use exactly identical definitions, so the technique shown above is often fine (with individual definitions per phase if necessary). Note that only phaseAndVoiceNameToStaffListMap is used by the generator, _voiceNameToStaffListMap is just an auxiliary variable.

It is also allowed to have more than one staff as the target of a voice. In that case the staff names are slash-separated and are filled from several voice macros in the lilypond fragment file. For two systems the macros are «voice»Top and «voice»Bottom with the phase target name appended, for three systems we have «voice»Top, «voice»Middle and «voice»Bottom. For example, a keyboard with a piano staff in a score references the macros keyboardTopScore and keyboardBottomScore.

Some replacement is done: if, for example, «voice»MiddleExtract does not exist, «voice»Middle and finally «voice» are taken instead.

So for a guitar with a tab the following definition in the configuration file is fine and it either reuses the guitar macro in the lilypond fragment file for both staffs or you can define special guitarTop/guitarBottom macros to

differentiate:

```
...
"guitar"   : "Staff/TabStaff",
...
```

When reusing the same voice data in different staffs, be careful with respect to the midi generation. Normally you only want the voice notes *once* in the midi file, hence you will have to adapt the phaseAndVoiceNameToStaffListMap definition and only include one staff in the midi file.

A similar logic as for the staffs applies to the mapping from voice name to clef. The standard clef is "G", others have to be defined explicitely. Especially this applies to multi-system-staffs like the "PianoStaff": here at least the "xxxBottom" must have a special clef definition (it must be a bass clef).

A typical definition might be given as follows:

```
_voiceNameToClefMap = \
    "{ bass"           : 'bass_8', "  \
     " drums"          : '',"         \
     " guitar"         : 'G_8',"      \
     " keyboardBottom" : 'bass',"     \
     " percussion"     : '' }"
```

Here bass and guitar have the transposed clef (as their traditional notation), drums and percussion have none and the lower part of a piano staff is notated in a bass clef.

Again the above is only an auxiliary definition. The relevant variable is phaseAndVoiceNameToClefMap shown below. In our case — as above — the mapping is identical for all phases, but, of course, individual definitions per phase are possible.

```
phaseAndVoiceNameToClefMap = \
    "{ extract : " _voiceNameToClefMap ","  \
     " midi    : " _voiceNameToClefMap ","  \
     " score   : " _voiceNameToClefMap ","  \
     " video   : " _voiceNameToClefMap "}"
```

Figure 8 shows all notation related configuration variables discussed in the current section.

"extract" Phase

Once everything is set up as described above, the "extract" phase generates an extract for each voice given in extractVoiceNameSet. The processing order of the voices is undefined.

For each voice an extract pdf file is put into the directory given by targetDirectoryPath with name fileNamePrefix, a dash, the voice name and the extension

Variable	Description	Example
phaseAndVoiceName-ToClefMap	mapping from processing phase to maps from voice name to lilypond clef	see text
phaseAndVoiceName-ToStaffListMap	mapping from processing phase to maps from voice name to slash-separated lilypond staff names	see text
voiceNameToChordsMap	mapping from voice names to phase abbreviations where chords are shown for that voice system	"{vocals: v/s, guitar: e}"
voiceNameToLyricsMap	mapping from voice name to a count of parallel lyrics lines directly following the target letter ("e" for the extract, "s" for the score and "v" for the video)	"{vocals: e2/s2/v}"

Figure 8: Notation Generation Configuration File Variables

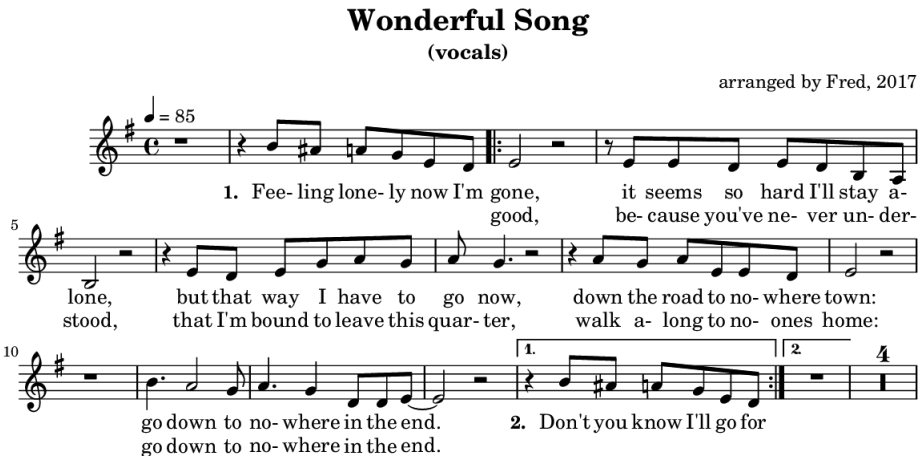

Figure 9: Example Layout of an Extract File

".pdf".

The headings in the extract are set as follows: the song name from the title variable is the extract title, the voice name is the extract subtitle, and the contents of composerText is the text for the composer part.

Figure 9 shows how the first page of an extract might look like and figure 10 shows the specific configuration variables for voice extracts.

"score" Phase

In the "score" phase the generator produces a single score with the voices given in scoreVoiceNameList in the order given and with default layout parameters.

Variable	Description	Example
extractVoiceNameSet	set of voices to be rendered as a voice extract	"vocals, drums"

Figure 10: Extract Generation Configuration File Variables

The score pdf file is put into the directory given by targetDirectoryPath with name fileNamePrefix followed by "_score" and the extension ".pdf".

Headings in the score are set as follows: the song name from the title variable is the score title and the contents of composerText is the text for the composer part.

Because voice names might be long, there is a mapping that provides a short name for each voice to be used in the score as the system identification by filling the variable voiceNameToScoreNameMap. A possible setting is:

```
voiceNameToScoreNameMap = \
    "{ bass           : bs,"   \
     " bgVocals       : bvc,"  \
     " drums          : dr,"   \
     " guitar         : gtr,"  \
     " keyboard       : kb,"   \
     " keyboardSimple : kb,"   \
     " organ          : org,"  \
     " percussion     : prc,"  \
     " strings        : str,"  \
     " synthesizer    : syn,"  \
     " vocals         : voc }"
```

With the settings above, the "bass" voice has a "bs" name in the score. You do not have to use that mechanism: the default is just to use the original voice name for staff identification in the score.

Figure 11 shows how the first page of a score might look like, figure 12 shows the specific configuration variables for scores.

7.4.1.2 Midi File Generation: "midi" Phase

The lilypond fragment file normally does not contain any further macros for MIDI because the voices used for the score are often fine for the MIDI file.

Nevertheless it could happen that you need special processing here. Examples are

- A voice has different notes or is transposed in the MIDI and audio rendering than in the notation. This can be achieved by having a different «voice»Midi macro.
- Some hidden voice occurs in MIDI and audio output, for example, a

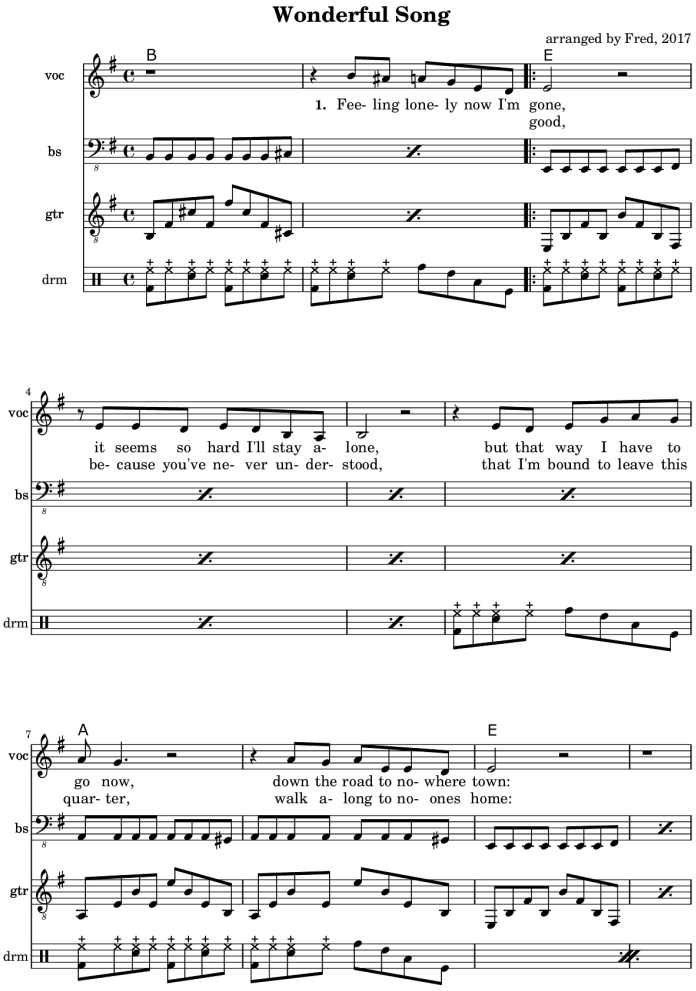

Figure 11: Example Layout of a Score File

voice delayed or transposed relative to some other voice (to enhance the sound of the original voice). This can be achieved by adding a voice to the voiceNameList macro, but excluding it from extracts, score and video.

The "midi" processing phase unfolds all repeats in the given voices and generates corresponding midi streams. Those streams are generated for all voices specified in the configuration variable midiVoiceNameList and stored in a single file in the directory given by targetDirectoryPath with name fileNamePrefix plus "-std" and extension ".mid".

All those voices have specific settings defined by several list variables, that align with the list voiceNameList and are shown in figure 13.

For example, the following settings in the configuration file

Variable	Description	Example
scoreVoiceNameList	list of voices to be rendered in order given into the score	"vocals, guitar, drums"
voiceNameToScore-NameMap	mapping from voices name to short score name at the beginning of a system	"{ vocals : voc, bass : bs }"

Figure 12: Score Generation Configuration File Variables

Variable	Description	Example
midiVoiceNameList	list of voices to be rendered in order given into the MIDI file	"guitar, drums"
midiChannelList	list of midi channels per voice each between 1 and 16 (10 for a drum voice)	see text
midiInstrumentList	list of midi instrument programs per voice each as an integer between 0 and 127; each entry may be prefixed by a bank number (0 to 127) followed by a colon	see text
midiVolumeList	list of midi volumes per voice each as an integer between 0 and 127	see text
midiPanList	list of pan positions per voice as a decimal value between 0 and 1 with suffix "R" or "L" (for right/left) or the character "C" (for center)	see text

Figure 13: Midi Related Configuration File Variables

```
voiceNameList      = "vocals, guitar, drums"
midiChannelList    = "   1,      2,     10 "
midiInstrumentList = "  54,   2:29,     16 "
midiVolumeList     = "  90,     60,    110 "
midiPanList        = "   C,   0.5L,    0.1R"
```

define vocals to be a synth vox in the center with 3/4 volume, the guitar to be an overdrive guitar (in bank 2), located half left with medium volume, and the drums to be a power set, located slightly right with almost full volume.

Nevertheless the midi phase not only transforms lilypond to plain midi, but does further processing by adding *humanization*. The variable humanized-VoiceNameSet tells what voices shall be humanized, the others are left untouched.

Humanization is done by adding random variations in timing and velocity to the notes in a voice. This is not completely random, but depends on voice, position within measure and on the style of the song.

The voice- (or instrument-specific) variation is global and defined by the

configuration variable voiceNameToVariationFactorMap. Each voice name is mapped onto a slash-separated pair of two numbers with the first giving the velocity, the second the timing variation percentage.

For a standard band instrument set, we take the variations of the drum as the reference in a humanization style. Hence drums should have an instrument-specific variation factor of 1.0 each which means that the calculated variation for some note is taken directly for drums. Other voices like, for example, vocals are slightly more loose and might have a value of 1.5 for velocity and 1.2 for timing which means that the calculated variation for those parameters is scaled accordingly. Of course, the velocity values are adjusted to their ranges after the variation, because there is a maximum and minimum velocity.

Our example would result in

```
voiceNameToVariationFactorMap = "{ drums: 1.0/1.0," \
                                 " vocals: 1.5/1.2}"
```

The *humanization style* of a song tells individual variations based on the position of a note within a measure. Hence it gives timing and velocity variations for the main beats and all other notes.

A *timing variation* is a positive decimal number and tells how much a note can be shifted in $1/32^{nd}$ notes (where 0 means no shift at all, 1 means a shift by at most a $1/32^{nd}$ etc.). A *velocity variation* tells the standard velocity level of a note at this position and the slack gives the maximum variation.

When specifying a style, the note positions within a measure are given as decimal fractions of a semibreve giving the offset to the measure start. For example, the first beat in a measure has offset 0, the third beat an offset of 0.5. Additionally each style specifies a raster size r, for example 0.125 for an eight note raster. When a measure position is given by an offset o, all notes in the open interval $(o - \frac{r}{2}, o + \frac{r}{2})$ will be handled by the given humanization definition.

The algorithmic logic for a note humanization is as follows:

1. Assume that the given note has time t_i and velocity v_i. Further assume that length of a thirtysecond note in time units is ℓ and that the instrument-specific adjustments from the table are adj_t and adj_v.

2. Pick two random numbers r_t and r_v both in the interval $[0, 1[$ from a quadratic probability distribution (which favours smaller numbers).

3. Depending on t_i find the note position p_i within its measure. Calculate the note offset within the measure and convert it to a fraction of a semibreve giving o_i. If o_i lies in some interval $(p - \frac{r}{2}, p + \frac{r}{2})$ — where r is the raster size specified in the style —, then the position p_i is given as p, otherwise the position is “OTHER”.

4. For the timing take the offset $\tau(p_i)$ given by the timing map for the current position p_i and multiply it by r_t and by the length of a thirty-second note and by the instrument-specific adjustment adj_t giving Δ_t. If the offset has a "B"(ehind) prefix, set the factor f_i to 1, because the note may only be behind the position; if the offset has an "A"(head) prefix, set the factor f_i to -1, because the note may only be ahead of the position; otherwise with each 50% probability set the factor f_i to either -1 or 1.

   Finally we have

   $$t'_i := t_i + f_i \cdot \Delta_t = t_i + f_i \cdot \tau(p_i) \cdot r_t \cdot \ell \cdot adj_t$$

   The timing of notes in a voice starting simultaneously is changed identically.

5. For the velocity take the associated velocity emphasis value $\sigma(p_i)$ given by the velocity map for the current position and the global slack in the velocity map ψ. The velocity is first scaled by the emphasis value $\sigma(p_i)$ (to accentuate beats) then randomly adjusted by slack ψ and instrument-specific adjustment adj_v and finally capped to the MIDI velocity interval $[0, 127]$.

   Finally we have

   $$v'_i := min(127, max(0, v_i \cdot (\sigma(p_i) + \psi \cdot r_v \cdot adj_v)))$$

   If the velocity already varies within a measure, no emphasis will *not* be applied, but only the slack. This means that whenever the voice already has some nontrivial accentuation, only some random velocity variation is applied.

The idea behind the approach for the velocity is to accent some beats in a measure. For example, a rock style would favour the 2 and 4, a march the 1. Timing may be varied or even be dragged or hurried.

So altogether a single style definition is a map telling about the velocity and the timing for positions in a measure plus information about position raster and velocity slack.

Let us take a rock style with steady beats on two and four (so no time variation here) and some emphasis on the second beat. In the configuration file it might look like

```
humanizationStyleRockHard  = \
    "{ 0.00:    1/0.2, 0.25: 1.15/0," \
    "  0.50: 0.95/0.2, 0.75:  1.1/0," \
    "  OTHER: 0.9/B0.25," \
    "  RASTER : 0.03125, SLACK : 0.1 }"
```

Variable	Description	Example
countInMeasureCount	number of count-in measures for the song (which defines the time before the first measure)	2
humanizedVoiceNameSet	set of voice names to be humanized by random variations of timing and velocity	"vocals, drums, keyboard"
measureToHumanizationStyleNameMap	map of measure number to humanization style name used from this position onward for humanized voices	" 1: styleXXX, 5: styleYYY "
humanizationStyle«name»	map that tells the initial count-in measures, the variation in timing and velocity for several positions within a measure	see text
voiceNameToVariationFactorMap	map from voice name to a pair of decimal factors characterizing the timing and velocity variation for this kind of voice to be applied additional to the humanization style	see text

Figure 14: Midi Humanization Related Configuration File Variables

All available humanization styles in the configuration file must have a "humanizationStyle" prefix in their names to be elegible.

Note that because all those definitions go anywhere in the configuration files, humanization styles could even be song-specific. On the other hand it is helpful to just reuse those styles, because humanization normally should not depend on the song, but on the style of the song only.

The song itself defines the styles to be applied as a style map from measure number to style starting here. Styles apply to all humanized instruments simultaneously, it is not possible to have, for example, a reggae on drums against a rumba on bass.

So the style map in the configuration file might look like

```
measureToHumanizationStyleNameMap =   \
"{   1 : humanizationStyleRockHard,"  \
  " 45 : humanizationStyleBeat}"
```

and tells that the "rock hard" style defined above is used at the beginning and that the style switches to a "beat" style in measure 45.

All humanization variables discussed above are shown summarized in the table in figure 14.

Variable	Description
height	height of device and video (in dots)
width	width of device and video (in dots)
resolution	resolution of the device (in dpi)
topBottomMargin	margin for video on top and bottom (in millimeters)
leftRightMargin	margin for video on left and right side (in millimeters)
systemSize	size of lilypond system (in lilypond units, cf. lilypond system size)
scalingFactor	the factor by which width and height are multiplied for lilypond image rendering to be downscaled accordingly by the video renderer (an integer)
frameRate	the frame rate of the video (in frames per second)
ffmpegPresetName	a specific ffmpeg preset for the current video target device (a string, a missing value defaults to a baseline level 3 profile)
mediaType	the Quicktime media type of the video (for example "TV Show")
subtitleColor	color of overlayed subtitle in final video for measure display (as integer for 16bit alpha/red/green/blue)
subtitleFontSize	height of subtitle (in pixels)
subtitlesAreHardcoded	flag to tell whether subtitles are burnt into the video or are available as a separate subtitle track

Figure 15: Parameters for Video Target in videoTargetMap Variable

7.4.1.3 Video Generation: "silentvideo" Phase

The video from the lilypond fragment file is produced by combining rendered images from lilypond in an intelligent fashion. "silentvideo" just renders the video without sound, later on the "finalvideo" phase in the postprocessing combines the silent video with the rendered audio tracks.

For the video rendering we need the characteristics of the video target, for example, the size and resolution of the device used. Additionally there is data as the rendering directory or the suffix used for the video files.

Because it might happen that several video renderings have similar video target properties, the information is split: a video rendering relies on a specific video target and gives details such as the directory where the video file goes or the names of the displayed voices.

So we have two configuration file variables:

- videoTargetMap provides video device dependent properties of notation videos, but also some device independent parameters (like, for example, the subtitle font size).

  This variable is a map from "target name" to a target descriptor. A target descriptor is itself a map with the several fields as shown in figure 15. Some of the variables like resolution, height or width describe

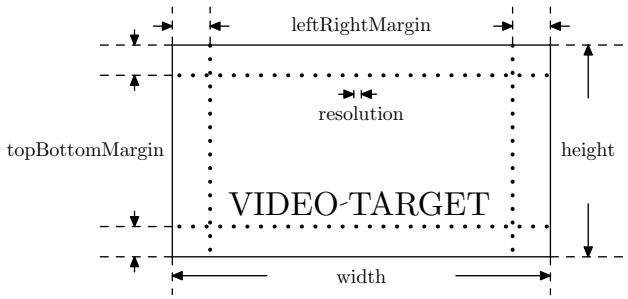

Figure 16: Target Parameters for Video Generation

Variable	Description
target	name of associated video target that is used when rendering video files of that kind
directoryPath	directory where final videos for that target go
fileNameSuffix	suffix to be used for the video file names for that target
voiceNameList	list of voice names to be rendered in order to audio files via the phase "silentvideo"

Figure 17: Parameters for Video File Kind in videoFileKindMap Variable

"hardware" parameters (because normally the video should have the appropriate size), others like topBottomMargin the layout of the video.

Figure 16 shows how some of the parameters for video generation are connected to the physical output device and the video target in general.

- videoFileKindMap provides further details on the rendering (like, for example, the list of voices to be shown).

  This variable is a map from a "video file kind name" to a video file kind descriptor. A video file kind descriptor is itself a map with the several fields as shown in figure 17. There is information about the target file given by videoDirectoryPath and fileNameSuffix and the list of the voices in those video files.

So a video target definition for a single midrange tablet could look like this:

```
videoTargetMap = \
  "{" \
    " tablet:" \
      " { fileNameSuffix:           '-i-v'," \
        " targetVideoDirectoryPath: '/pathto/tablet'," \
        " resolution:               132," \
        " height:                   1024," \
        " width:                    768," \
        " topBottomMargin:          5," \
        " leftRightMargin:          10," \
        " systemSize:               25," \
        " ffmpegPresetName:         'mydevice'," \
        " scalingFactor:            4," \
        " frameRate:                10," \
        " mediaType:                'TV Show'," \
        " subtitleColor:            2281766911," \
        " subtitleFontSize:         20," \
        " subtitlesAreHardcoded:    false }" \
  "}"
```

The above defines a target called "tablet" having a video with 1024x768 pixels, a resolution of 132dpi, a margin of 5mm at top and bottom, a margin of 10mm left and right, slightly enlarged systems (lilypond standard system size is 20), a yellow semi-transparent subtitle with size 20 pixels. The video is encoded by ffmpeg with an ffmpeg preset called "mydevice" at a frame rate of 10fps (which is ample for a more or less static video and ensures that the time resolution for page turning and subtitle changes is 0.1s) and lilypond produces images 4 times wider and higher than needed to be downscaled by the video renderer for better video image quality. The quicktime media type is "TV Show" and subtitles in the final video are on a separate track.

Based on the video target definition given above a video file kind definition could look like this:

```
videoFileKindMap = \
  "{" \
    " tabletVocGtr:" \
      " { target:          tablet,"            \
        " fileNameSuffix:  '-i-v',"            \
        " directoryPath:   '/pathto/xyz',"     \
        " voiceNameList:   'vocals, guitar' }" \
  "}"
```

The above defines a single file kind for output. The target characteristics are those of a "tablet", those videos contain a score with vocals plus guitar and all the files have suffix '-i-v' (followed by '.mp4', of course).

So the silent video generation produces an MP4 video file for each video file kind specified. Each video displays a score with all voices specified in the configuration variable videoFileKind.voiceNameList with automatic page turning at the right points in time. That video is stored in a single file in the

Variable	Description	Example
videoTargetMap	mapping from video target name to video target descriptor with several parameters for specific video file generation	see text
videoFileKindMap	mapping from video file kind name to video file kind descriptor with several parameters for specific video file generation referencing a video target that gives overall video parameters	see text

Figure 18: Video Configuration File Variables

directory given by videoFileKind.directoryPath with name fileNamePrefix plus "_noaudio" and the videoFileKind.fileNameSuffix from the file kind specification and extension ".mp4".

Additionally a subtitle file with all measure numbers is generated in the directory given by targetDirectoryPath with name fileNamePrefix plus "_subtitle" and extension ".srt".

This means that a song with file name prefix "wonderful_song" and a target file name suffix "-tablet" leads to a silent video file of "wonderful_song_noaudio-tablet.mp4" and a subtitle file of "wonderful_song_subtitle.srt". Note that the subtitle file is independent of the video target, because it only gives the time intervals of each measure and those do not depend on the video.

If you *really* want to fiddle with the video generation, the video target name is provided as the lilypond macro ltbvcVideoTargetName and has the values specified as keys in the list videoTargetMap. You can use this for conditional processing, video layout changes etc., because the file inclusion into the boilerplate file is done at a very late position. Be warned that the whole video generation might fail, because the generator assumes that it has to handle a simple-structured lilypond include file.

There is only a single configuration file variable for video as shown in figure 18 that defines all video targets that are used in the generation.

Because the algorithm for finding the page breaks in the video relies on data scraping of a postscript file produced by lilypond, some restrictions apply for the notation videos: the bar numbers are activated for the line starts only and those bar numbers as well as the bar lines will be black.

7.4.2 Postprocessing Phases

All postprocessing phases rely on the configuration file, the generated midi file and the silent videos; the lilypond fragment file is not used any longer.

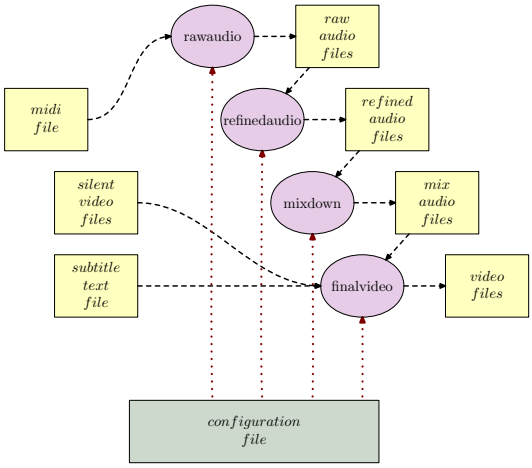

Figure 19: Information Flow for the Postprocessing Phases

Figure 19 shows the connection between the inputs and the outputs for the phases. Only the configuration file serves as manual input into the processing chain, the other files are generated from files coming from the preprocessing phases in section 7.4.1.

The following processing is done:

rawaudio:
the midi file is rendered via fluidsynth and sound fonts into plain audio files for each relevant audio voice,

refinedaudio:
based on voice-specific sound definitions each plain audio file is refined typically by sox processing for each relevant audio voice into a refined audio file,

mixdown:
mixed versions of the voice audio files are generated with sox grouped into audio groups from the configuration file (for later selection as audio track), and

finalvideo:
the still videos and the subtitle file produced from the lilypond fragment file are combined with the grouped audio files to video files with selectable audio tracks and either selectable or burnt in

7.4.2.1 Audio Generation: "rawaudio" and "refinedaudio" Phase

Each voice in audioVoiceNameSet is rendered to audio files via the phases "rawaudio" and "refinedaudio" based on the humanized midi file from sec-

tion 7.4.1.2. The audioVoiceNameSet variable is an (unordered) list of voices names that are a subset of those occuring in the midiVoiceNameList.

"rawaudio" Phase

The "rawaudio" phase simply takes each voice given in the audio voice name set and converts the humanized midi stream into a wave file using midiToWavRenderingCommandLine typically using the fluidsynth program. This command line relies on soundfont files specified in that string. The name order of the soundfonts (of type sf2 or sf3) give the order of matching a given midi instrument number: the first match is accepted.

Note that the midi volume is not used by this phase: any midi volume changes are suppressed and only the velocity is used.

For each voice the resulting wave file after generation is stored in directory tempAudioDirectoryPath as an intermediate file for further processing. The naming convention is to use the voice name with a ".wav" extension (for example, "bass.wav" stores the result for a bass voice).

"refinedaudio" Phase

Normally the sounds produced by soundfonts need some beefing up. This is done in the "refinedaudio" phase where the audio file from the previous phase are postprocessed by the sound processor sox.

sox is a commandline program where chains of effects are applied to audio input files producing audio output files. For example, the command

```
sox input.wav output.wav highpass 80 2q reverb 50
```

applies a double-pole highpass filter at 80Hz with a width of 2q followed by a medium reverb to file input.wav and stores the result in file output.wav.

sox has a lot of those filters and all those can be used for sound shaping. In this document we cannot go into details, but a thorough information can be found in the sox documentation [SOX].

Note that you can also use another command-line audio processor by setting the variable audioRefinementCommandLine appropriately and adapting the refinement commands for the voices for the tool used. But this is an expert solution beyond the scope of this documentation; hence you are on your own...

Each audio voice is transformed depending on voice-specific settings in the configuration file. Because the input file comes from the previous "rawaudio" phase (for example "bass.wav") and the output file name for the "refinedaudio" phase is also well-defined (for example as "bass-processed.wav"), we only have to specify the sox commands for the transformation itself.

Those commands depend on the voice/instrument and on the style of the playing and this is combined in a so-called *sound style* variable.

The name of sound style variables is constructed as follows: the prefix "soundStyle" is followed by the voice name with initial caps (for example "Bass") and by the style variant — a single word — capitalized as suffix ("Hard"). When following this convention, a hard bass has a sound style name "soundStyleBassHard".

Very often a sound style is not defined on its own, but relies on other definitions. Let us assume we have some standard postprocessing for a bass. This consists of a normalization with 24dB headroom (to prevent distortion in the following steps), an enhancement of the 150Hz band by 10dB and a 6dB cutoff of high frequencies above 600Hz. In the configuration file this could look as follows:

```
_bassPostprocess = \
    " norm -24" \
    " equalizer  150 4o +10" \
    " lowpass -2 600 1.2o"
```

Based on that definition above the actual sound style can be defined as follows (referencing the definition by name):

```
soundStyleBassHard = \
    " highpass -2 40"  \
    " lowpass -2 2k"   \
    " norm -6 "        \
    " tee"             \
    " overdrive 12 0 " \
    _bassPostprocess
```

The sound style definition uses a low- and highpass followed by an overdrive and the final equalization. Note that the name is *not* in double quotes: this distinguishes it from plain text (as explained in section 5.2).

There are four things to note:

1. As demonstrated sound styles may rely on other definitions; so you can build a hierarchy of effect chains.

2. The special effect "tee" is not part of sox. When debugging is active, this "effect" writes out the audio data available at that position in the chain into a temporary file in the target audio directory called "«voice»X.wav" where X stands for a hex number. Multiple "tee" commands are possible, so you can do an audio debugging of your chain.

3. Normally processing is purely sequential with a single signal path (which is standard sox behaviour). But it is possible to add parallel signal paths and to combine them (e.g. for New York parallel compression etc.).

4. Reverb may be specified in the chain or — for really simple applications — is automatically applied with default parameters and an intensity defined by the configuration variable reverbLevelList to the final audio.

   If that simple reverb is not good enough and specific settings are needed, you can set the reverb level for some voice to 0 and add a more elaborate reverb effect to the sound style. If you leave off the reverbLevelList altogether, all voices have no automatic reverb applied.

So how do we apply the specific sound style and some reverb to our bass? The settings in the song configuration file are as follows

```
voiceNameList    = "...,  bass, ..."
reverbLevelList  = "...,   0.4, ..."
soundVariantList = "...,  hard, ..."
```

As above reverbLevelList and soundVariantList are lists with elements in the same order as voiceNameList. There is a special sound variant called copy that just takes the raw audio file and applies the specified reverb to it.

The sound variant may be given in any letter case, because it is automatically adapted for the selection of the sound style. Combined with the above sound style this leads to the following sox commands — when debugging is active — (note the command split at the tee effect and the added final reverb with 100· reverbLevel):

```
sox bass.wav bassA.wav highpass -2 40 lowpass -2 2k norm -6
sox bassA.wav bass-processed.wav overdrive 12 0 norm -24 \
     equalizer 150 4o +10 lowpass -2 600 1.2o reverb 40
```

In general, sound styles can be defined per song or globally. I prefer the latter, because I use a few bread-and-butter sounds per instrument and adapt them only by using different midi instruments, audio volumes and reverb levels in the voice configuration; hence the sound styles itself are not adapted. But in principle you can fine-tune the voice sounds per song, which I find tedious.

For the bread-and-butter sound approach, it is helpful to use a simple set of variant names that apply to all voices, for example, “STD” (for a normal sound), “HARD” (for some heavier sound), “EXTREME” (for an ultra-hard sound) etc.

So finally each audio voice has its processed wav version in targetDirectoryPath called “«voice»-processed.wav” for later mixdown.

Parallel Paths

Parallel signal paths cannot be handled directly by sox and are emulated by LilypondToBandVideoConverter. They can be specified as follows:

- Parallel chains are specified by using chain separators using the char-

acter token ":".

- For each chain its (single) source and target are each given by an identifier that is immediately preceeded or followed by "->". So a chain target might be specified as "->xxx", a chain source might be specified as "yyy->". When no identifier is given for a source, the raw audio file is used.

  Note that, of course, the name of a chain source must occur as a preceeding chain target.

  The first chain has "->" (the raw audio file) as its chain source, the last chain has the refined audio file as target.

- A chain may consist of a special "mix" command that does a weighted mix of several sources into a single target. E.g. the chain

  ```
  mix 1.0 -> 0.3 A-> 0.5 B-> ->C
  ```

  mixes 100% of the raw audio file, 30% of A and 50% of B into C.

  Very often, the last chain is a mix of several sources into the refined audio file as the target.

As an example let us enhance a bass part by adding a copy pitched down by an octave and having some parallel compression added. We assume that the bass is pre-processed by "soundStyleBassStd" and we simple add the postprocessing as follows:

```
soundStyleBassStd ->A \
: A-> pitch -1200 ->B \
: mix 1.0 A-> 0.75 B-> ->C \
: C-> compand 0.04,0.5 6:-25,-20,-5 -6 -90 0.02 ->D \
: mix 1.0 C-> 0.4 D->
```

"A" contains the preprocessed audio, "B" the pitched down version, "C" the enriched bass sound, "D" the compressed version of it and the combined audio goes to the refined audio file.

Special Tracks

Another helpful feature of the "refinedaudio" phase is the ability to introduce other audio files into the processing. There are two cases:

1. One can override a processed track by some external audio file.
2. A parallel track in a file not related to some voice can be added.

So both cases involve external audio files to be added.

The first case is common when you want to replace a track by a real recording. For example, the vocals with midi beeps could be enhanced by having a real singer sing the track.

All those tracks are described in the configuration variable voiceNameToOverrideFileNameMap. As its name tells, it maps voice names to file names.

```
voiceNameToOverrideFileNameMap = \
    "{ vocals : 'vocals.flac'," \
      "bass   : 'mybass.wav'  }"
```

This approach replaces the processed voice files by the contents of the files given in the map. File types supported are all those supported by sox as input. Note that the overriding file has to have the length of a refined voice file, that means, it also has to contain material for the count-in measures.

In the second case no specific voice track is replaced, but some parallel track is introduced. For example, this could be used for lead-in text or audience audio.

In principle this could be handled by introducing an artificial voice only used for audio, but for convenience there is another variable called parallelTrack for a single additional track. It contains comma-separated data for an audio file name, a volume factor and offset relative to the start of the song in seconds as follows:

```
parallelTrack = " parallelFile.wav, 1.0, 2.8"
```

Note that it is only possible to have a single parallel track.

Summary of Audio Configuration Variables

Figure 20 shows all the configuration variables described for the "rawaudio" and "refinedaudio" phases.

7.4.2.2 Final Audio Generation: "mixdown" Phase

The "mixdown" phase combines the refined audio files into one or more audio file with all voices and in aac audio format.

Audio levels of the individual voices and a final attenuation factor are specified in the configuration; the audio voices are mixed with those levels and the attenuation is applied to the mix before it gets compressed into an AAC file.

The entries are specified as follows:

Variable	Description	Example
audioVoiceNameSet	set of voice names to be rendered to audio files via the phases "rawaudio" and "refinedaudio" based on voice representations in humanized midi file	"vocals, drums, bass"
reverbLevelList	list of reverb levels (as decimal values typically between 0 and 1) for the voices aligned with the list voiceNameList; those reverb levels are applied to each voice as the final refinement operation	"0.1, 1.1, 0.5, 0.0"
soundStyle«Voice»-«Variant»	sequence of (sox) refinement commands to be applied on raw audio file when this style is selected for «voice»	see text
soundVariantList	list of variant names for the sound styles of the voices aligned with the list voiceName; those style variant names are combined into a complete style name to be applied during audio refinement	"COPY, EXTREME, STD, HARD"
voiceNameToOverride-FileNameMap	map from voice name to name of file overriding that voice in the processed audio files and in the final mixdown audio files and in the target videos	see text

Figure 20: Audio Configuration File Variables

```
voiceNameList  = "...,  bass, guitar, ..."
audioLevelList = "...,   0.7,    0.5, ..."

attenuationLevel = -0.2
```

The target file is stored in the audioTargetDirectoryPath with a name concatenated from targetFileNamePrefix, fileNamePrefix and suffix "-ALL.m4a".

But: you do not want a backing track with all voices of your arrangement, but the ones to be played live should be missing and ideally one should be able to switch them on and off!

Again we specify this by several mapping variables in the configuration file.

The first variable, audioGroupToVoicesMap, specifies a partitioning of the audio voices into groups where some freely selectable audio group names are mapped onto sets of audio voice names.

Variable	Description
audioGroupList	slash-separated list of audio group names occuring as keys in audioGroupToVoicesMap
audioFileTemplate	template string defining how the audio file name of the target audio file for given list of voices is constructed from the plain audio file name (indicated by a dollar-sign)
songNameTemplate	template string defining how the song name for given list of voices is constructed from the plain song name (indicated by a dollar-sign)
albumName	name of the album of the audio file for given list of voices (where an embedded dollar-sign is replaced by the global album name)
description	description for audio track within target video (typically unsupported by video players)
languageCode	ISO language code for audio track within target video (typically supported by video players)

Figure 21: Parameters for Audio Track in audioTrackList Variable

```
audioGroupToVoicesMap = "{" \
    " base : bass/keyboard/keyboardSimple/strings," \
    " voc  : vocals/bgVocals," \
    " gtr  : guitar," \
    " drm  : drums/percussion" \
 "}"
```

The voice names in the song should be a subset of the voice names mentioned in the audio group map, missing or extraneous voice names will be ignored. When defining those settings globally for a group of songs, ensure that typical voice name variants (like, for example, “keyboardSimple”) are included in one of the lists; otherwise those voices will be missed in the mix files and videos.

The second variable, audioTrackList, specifies all tracks that will later occur as tracks in the video, but also that are rendered as compressed audio files.

Each track is described by a track descriptor with several fields as shown in figure 21. It consists of a list of the several groups to be combined, templates for the audio file and the song name, an album name, and some description and a language code for the video track.

“Language code” sounds a bit strange: why do you need that?

Unfortunately not many video players support audio track description texts for MP4 videos, but most of them allow to select audio tracks by “language”. So the audio tracks in the final video are tagged with both description and language code for some kind of identification. Of course, the selected languages are quite arbitrary, because you typically do not find a connection between a list of audio voice names and some language name. So you must be creative...

Altogether we have something like that in the configuration file:

```
audioTrackList = "{" \
  "all :   { audioGroupList    : base/voc/gtr/drm,"    \
  "          audioFileTemplate : '$',"                 \
  "          songNameTemplate  : '$ [ALL]',"           \
  "          albumName         : 'Best',"              \
  "          description       : 'all voices',"        \
  "          languageCode      : eng },"               \
  "novoc : { audioGroupList    : base/gtr/drm,"        \
  "          audioFileTemplate : '$-novoc',"           \
  "          songNameTemplate  : '$ [-V]',"            \
  "          albumName         : 'Best [no vocals]',"  \
  "          description       : 'no vocals',"         \
  "          languageCode      : deu },"               \
  ...
"}"
```

So any number of audio tracks is possible. In the example above we have two (if you ignore the ellipsis!). If we assume that the target file name prefix is "test-" and that the song has file name prefix "wonderful_song" and is called "Wonderful Song", the files have the following properties:

1. The first track has all voices, is stored in "test-wonderful_song.m4a" with title "Wonderful Song [ALL]" in album "Best" and it has description "all voices" and an English language tag.

2. The second track has all voices except for vocals and bg vocals, is stored in "test-wonderful_song-novoc.m4a" with title "Wonderful Song [-V]" in album "Best [no vocals]" and it has description "no vocals" and a German language tag.

Figure 22 shows the variables introduced in this section in summary.

7.4.2.3 Video Generation: "finalvideo" Phase

The still videos from the lilypond fragment file contain rendered score images from lilypond with appropriate display times. The "finalvideo" phase combines those silent videos with the subtitle file and the rendered audio tracks from above.

There are no big surprises here: for every video file kind in the list videoFileKindMap a video is built with the following parts:

- the file-kind-specific still video with the appropriate extension fileNameSuffix for the given target name finally located in targetDirectoryPath,

- the subtitle file located in targetDirectoryPath, and

Variable	Description	Example
attenuationLevel	decimal value in decibels telling the volume change to be applied to the final audio files; this is helpful to adjust volume levels of different songs within an album	-1.3
audioGroupToVoicesMap	mapping from freely defined voice group names to names of voices contained in that group described by a slash-separated name list	see text
audioLevelList	list of volume factors aligned with the list voiceName used for mixing the refined audio files into cumulated audio files; the factors are decimal values with 1.0 meaning that the refined voice file is taken unchanged	"0.7, 0.5, 1.2"
audioTargetDirectoryPath	path for the final AAC audio files with subsets of rendered and refined audio tracks	"/pathto/XXX"
audioTrackList	list of track descriptors defining groups of audio groups to be put on some track with naming templates for audio file, song and album name and a track description and language	see text

Figure 22: Mixdown Configuration File Variables

- the compressed audio files generated by the "mixdown" phase and located in audioTargetDirectoryPath

If subtitlesAreHardcoded is set for the target, the subtitle is burnt into the video with specified subtitleFontSize and subtitleColor. Otherwise the subtitle is put into the target video as a subtitle track (to be switched on or off). In the latter case, the rendering of the subtitle is done by the video player.

The name of the combined video is constructed as follows: the targetFileNamePrefix is concatenated with fileNamePrefix for the song, a minus character, the video file kind name suffix and ".mp4" extension. It is stored in the directory given by videoFileKind.directoryPath.

For example, by those conventions the "Wonderful Song" for the "tablet" has name "test-wonderful_song-tablet.mp4" and is stored in the directory given in the target definition.

7.5 Summary

We're done! We have achieved the following results from a lilypond fragment file with song voices and a song configuration file:

- notation extracts of selected voices as PDF files,
- a notation score of selected voices as a PDF file,
- a MIDI file with selected voices slightly humanized,
- several single voice audio files,
- audio file mixes combining voices into groups, and
- video files for different target devices containing selectable audio tracks and possibly a selectable subtitle with measure indication

8. Example

As the example we take a twelve-bar blues in E with two verses and some intro and outro. Note that this song is just an example, its musical merit is limited.

In the following we shall work with three files:

- a global configuration file containing overall settings (like for example, the path to programs),
- a song-specific configuration file containing the settings for the song (like, for example, the title of the song or the voice names), and
- a lilypond music file containing the music fragments used by the generator.

In principle one only needs a *single* configuration file and a single lilypond fragment file, but by this approach we can keep global and song-specific stuff separate.

In the following we explain the lilypond fragment file and configuration file in pieces; the complete versions are in the distribution.

8.1 Example Lilypond Fragment File

The lilypond fragment file starts with the inclusion of the note name language file:

```
\include "english.ly"
```

The first musical definition is the key and time designation of the song: it is in e major and uses common time.

```
keyAndTime = { \key e \major  \time 4/4 }
```

The chords are those of a plain blues with a very simple intro and outro. Note that the chords differ for extract and other notation renderings: for the extract and score we use a volta repeat for the verses, hence in that case all verse lyrics are stacked vertically and we only have one pass of the verse.

All chords are generic: there is no distinction by instrument.

```
chordsIntro = \chordmode { b1*2 | }
chordsOutro = \chordmode { e1*2 | b2 a2 | e1 }
chordsVerse = \chordmode { e1*4 | a1*2 e1*2 | b1 a1 e1*2 }

allChords = {
  \chordsIntro  \repeat unfold 2 { \chordsVerse }
  \chordsOutro
}

chordsExtract = { \chordsIntro  \chordsVerse  \chordsOutro }
chordsScore   = { \chordsExtract }
```

The vocals are simple with a pickup measure. Because we want to keep the notation consistent across the voices we have to use two alternate endings for the vocalsExtract and vocalsScore.

```
vocTransition = \relative c' { r4 b'8 as a g e d | }
vocVersePrefix = \relative c' {
  e2 r | r8 e e d e d b a |
  b2 r | r4 e8 d e g a g | a8 g4. r2 | r4 a8 g a e e d |
  e2 r | r1 | b'4. a2 g8 | a4. g4 d8 d e~ | e2 r |
}

vocIntro = { r1 \vocTransition }
vocVerse = { \vocVersePrefix \vocTransition }

vocals = { \vocIntro \vocVerse \vocVersePrefix R1*5 }
vocalsExtract = {
  \vocIntro
  \repeat volta 2 { \vocVersePrefix }
  \alternative {
      { \vocTransition }{ R1 }
  }
  R1*4
}
vocalsScore = { \vocalsExtract }
```

The lyrics of the demo song are really bad. Nevertheless note the lilypond separation for the syllables and the stanza marks. For the video notation the lyrics are serialized. Because of the pickup measure, the lyrics have to be juggled around.

```
vocalsLyricsBPrefix = \lyricmode {
  \set stanza = #"2. " Don't you know I'll go for }

vocalsLyricsBSuffix = \lyricmode {
  good, be- cause you've ne- ver un- der- stood,
  that I'm bound to leave this quar- ter,
  walk a- long to no- ones home:
  go down to no- where in the end. }

vocalsLyricsA = \lyricmode {
  \set stanza = #"1. "
  Fee- ling lone- ly now I'm gone,
  it seems so hard I'll stay a- lone,
  but that way I have to go now,
  down the road to no- where town:
  go down to no- where in the end.
  \vocalsLyricsBPrefix }

vocalsLyricsB = \lyricmode {
  _ _ _ _ _ _ \vocalsLyricsBSuffix }
vocalsLyrics = { \vocalsLyricsA \vocalsLyricsBSuffix }
vocalsLyricsVideo = { \vocalsLyrics }
```

The bass simply hammers out eighth notes. As before there is an extract and a score version with volta repeats and an unfolded version for the rest.

```
bsTonPhrase  = \relative c, { \repeat unfold 7 { e8  } fs8 }
bsSubDPhrase = \relative c, { \repeat unfold 7 { a'8 } gs8 }
bsDomPhrase  = \relative c, { \repeat unfold 7 { b'8 } cs8 }
bsDoubleTonPhrase = { \repeat percent 2 { \bsTonPhrase } }
bsOutroPhrase = \relative c, { b8 b b b gs a b a | e1 | }

bsIntro = { \repeat percent 2 { \bsDomPhrase } }
bsOutro = { \bsDoubleTonPhrase  \bsOutroPhrase }
bsVersePrefix = {
  \repeat percent 4 { \bsTonPhrase }
  \bsSubDPhrase \bsSubDPhrase \bsDoubleTonPhrase
  \bsDomPhrase \bsSubDPhrase \bsTonPhrase
}
bsVerse = { \bsVersePrefix \bsTonPhrase }

bass = { \bsIntro  \bsVerse \bsVerse  \bsOutro }
bassExtract = {
  \bsIntro
  \repeat volta 2 { \bsVersePrefix }
  \alternative {
    {\bsTonPhrase} {\bsTonPhrase}
  }
  \bsOutro
}
bassScore = { \bassExtract }
```

The guitar plays arpeggios. As can be seen here, very often the lilypond

macro structure is similar for different voices.

```
gtrTonPhrase  = \relative c { e,8 b' fs' b, b' fs b, fs }
gtrSubDPhrase = \relative c { a8 e' b' e, e' b e, b }
gtrDomPhrase  = \relative c { b8 fs' cs' fs, fs' cs fs, cs }
gtrDoubleTonPhrase = { \repeat percent 2 { \gtrTonPhrase } }
gtrOutroPhrase = \relative c { b4 fs' a, e | <e b'>1 | }

gtrIntro = { \repeat percent 2 { \gtrDomPhrase } }
gtrOutro = { \gtrDoubleTonPhrase | \gtrOutroPhrase }
gtrVersePrefix = {
  \repeat percent 4 { \gtrTonPhrase }
  \gtrSubDPhrase  \gtrSubDPhrase  \gtrDoubleTonPhrase
  \gtrDomPhrase  \gtrSubDPhrase  \gtrTonPhrase
}
gtrVerse = { \gtrVersePrefix \gtrTonPhrase }

guitar = { \gtrIntro  \gtrVerse  \gtrVerse  \gtrOutro }
guitarExtract = {
  \gtrIntro
  \repeat volta 2 { \gtrVersePrefix }
  \alternative {
    {\gtrTonPhrase} {\gtrTonPhrase}
  }
  \gtrOutro
}
guitarScore = { \guitarExtract }
```

Finally the drums do some monotonic blues accompaniment. We have to use the myDrums name here, because drums is a predefined name in lilypond. There is no preprocessing of the lilypond fragment file: it is just included into some boilerplate code.

```
drmPhrase = \drummode { <bd hhc>8 hhc <sn hhc> hhc }
drmOstinato = { \repeat unfold 2 { \drmPhrase } }
drmFill = \drummode { \drmPhrase tomh8 tommh toml tomfl }
drmIntro = { \drmOstinato  \drmFill }
drmOutro = \drummode {
  \repeat percent 6 { \drmPhrase } | <sn cymc>1 | }
drmVersePrefix = {
  \repeat percent 3 { \drmOstinato }  \drmFill
  \repeat percent 2 { \drmOstinato  \drmFill }
  \repeat percent 3 { \drmOstinato }
}
drmVerse = { \drmVersePrefix \drmFill }

myDrums = { \drmIntro  \drmVerse \drmVerse  \drmOutro }
myDrumsExtract = { \drmIntro
  \repeat volta 2 {\drmVersePrefix}
  \alternative {
   {\drmFill} {\drmFill}
  }
  \drmOutro }
myDrumsScore = { \myDrumsExtract }
```

So we are done with the lilypond fragment file. What we have defined are

- the song key and time,
- the chords,
- the vocal lyrics, and
- voices for vocals, bass, guitar and drums.

All those definitions take care that the notations shall differ in our case for extracts/score and other notation renderings.

8.2 Example Configuration Files

As mentioned above the configuration is split up into a file with global settings and one with the song settings.

As a convention we prefix auxiliary variable with an underscore to distinguish them from the real configuration variables.

8.2.1 Example Global Configuration

The first setup steps define the program locations. We assume that programs are located together in some directory, but this depends on the environment.

All definitions assume a Unix context, but you may also use slashes as path separators for Windows.

```
_programDirectory = "/usr/local"
aacCommandLine = \
    _programDirectory "/qaac -V100 -i ${infile} -o ${outfile}"
ffmpegCommand = _programDirectory "/ffmpeg"
fluidsynthCommand = _programDirectory "/fluidsynth"
lilypondCommand = _programDirectory "/lilypond"
lilypondVersion = "2.18.2"
soxCommandLinePrefix = _programDirectory "/sox"
```

We have not provided a definition for the mp4boxcommand because — as a default — ffmpeg can also do the MP4 container packaging. Note also that aac and sox must have more extensive definitions.

Other global settings define paths for files or directories. The generated PDF and MIDI files go to subdirectory "generated" of the current directory, audio into "/tmp/audiofiles".

```
loggingFilePath = "/tmp/logs/ltbvc.log"
targetDirectoryPath = "generated"
tempAudioDirectoryPath = "/tmp/audiofiles"
```

For the notation we ensure that drums use the drum staff and that the clefs for bass and guitar are transposed by an octave and that drums have no clef at all. Chords shall be shown for all extracts of melodic instruments and on the top voice "vocals" in the score and video.

```
_voiceNameToStaffListMap = "{ drums : DrumStaff }"
_voiceNameToClefMap = "{" \
  "bass : bass_8, drums : '', guitar : G_8" \
"}"

phaseAndVoiceNameToStaffListMap = "{"      \
  "extract :" _voiceNameToStaffListMap ","  \
  "midi    :" _voiceNameToStaffListMap ","  \
  "score   :" _voiceNameToStaffListMap ","  \
  "video   :" _voiceNameToStaffListMap "}"

phaseAndVoiceNameToClefMap = "{"      \
  "extract :" _voiceNameToClefMap ","  \
  "midi    :" _voiceNameToClefMap ","  \
  "score   :" _voiceNameToClefMap ","  \
  "video   :" _voiceNameToClefMap "}"

voiceNameToChordsMap = "{" \
  "vocals : s/v, bass : e, guitar : e" \
"}"
```

The humanization for the MIDI and audio files is quite simple: we use a rock groove with tight hits on two and four and slight variations for other measure

positions. The timing variations are very subtle as the variation is at most 0.2 $1/32^{nd}$ notes.

As the velocity variation there is a hard accent on two and a slighter accent on four while the other positions are much weaker.

We have *not* defined individual variation factors per instrument; hence all humanized instruments have similar variations in timing and velocity.

```
countInMeasureCount = 2

humanizationStyleRockHard  = \
  "{ 0.00: 1.0/0.1, 0.25: 1.15/0," \
  "  0.50: 0.95/0.1, 0.75: 1.1/0," \
  "  OTHER: 0.85/B0.2," \
  "  SLACK:0.1, RASTER: 0.03125 }"
```

The video generation is just done for a single video target called "tablet" with a portrait orientation and a classical 4:3 aspect ratio. The strange integer below for the subtitle color is a hexadecimal 8800FFFF, that is a yellow with about 45% transparency. And the videos show both vocals and guitar and are characterized as "Music Videos" in their media type.

```
videoTargetMap = "{" \
    "tablet: { resolution: 132," \
             " height: 1024," \
             " width: 768," \
             " topBottomMargin: 5," \
             " leftRightMargin: 10," \
             " scalingFactor: 4," \
             " frameRate: 10.0," \
             " mediaType: 'Music Video'," \
             " systemSize: 25," \
             " subtitleColor: 2281766911," \
             " subtitleFontSize: 20," \
             " subtitlesAreHardcoded: true } }"

videoFileKindMap = "{" \
    "tabletVocGtr: { target:         tablet,"      \
                   " fileNameSuffix: '-tblt-vg',"     \
                   " directoryPath:  './mediaFiles' ," \
                   " voiceNameList:  'vocals, guitar' } }"
```

For the transformation from midi tracks to audio files there are only two sound style definitions: an extreme bass and a crunchy guitar. Both use overdrive and some sound shaping, the guitar style also applies a bit of compression. Details of the parameters can be found in the sox documentation [SOX].

For all the other voices we shall specify later that they just use the raw audio files with some reverb added.

```
soundStyleBassExtreme = \
    " norm -12 highpass -2 40 lowpass -2 2k" \
    " norm -10 overdrive 30 0" \
    " norm -24 equalizer  150 4o +10 lowpass -2 600 1.2o"

soundStyleGuitarCrunch = \
    " highpass -1 100 norm -6" \
    " compand 0.04,0.5 6:-25,-20,-5 -6 -90 0.02" \
    " overdrive 10 40"
```

For the final audio files we have two variants: one with all voices, the other one with missing vocals and background vocals (the "karaoke version"). The song and album names have the appropriate info in brackets.

All songs and the video will go to the "mediaFiles" subdirectory of HOME and have a jpeg-file as their embedded album art. Audio and video files have "test-" as their prefix before the song name. So, for example, the audio file for "Wonderful Song" with all voices has path "./mediaFiles/test-wonderful_song.m4a".

```
targetFileNamePrefix = "test-"
audioTargetDirectoryPath = "./mediaFiles"
albumArtFilePath = "./mediaFiles/demo.jpg"

audioGroupToVoicesMap = "{" \
    " base : bass/keyboard/strings/drums/percussion," \
    " voc  : vocals/bgVocals," \
    " gtr  : guitar" \
"}"

audioTrackList = "{" \
    "all :      { audioGroupList    : base/voc/gtr,"  \
    "             audioFileTemplate : '$',"      \
    "             songNameTemplate  : '$ [ALL]',"     \
    "             albumName         : '$',"           \
    "             description       : 'all voices',"  \
    "             languageCode      : deu },"         \
    "novocals : { audioGroupList    : base/gtr,"      \
    "             audioFileTemplate : '$-v',"    \
    "             songNameTemplate  : '$ [-V]',"      \
    "             albumName         : '$ [-V]',"      \
    "             description       : 'no vocals',"   \
    "             languageCode      : eng }"          \
"}"
```

8.2.2 Example Song Configuration

There is not much left to define the song. First come the overall properties:

```
title = "Wonderful Song"
fileNamePrefix = "wonderful_song"
year = 2017
composerText = "arranged by Fred, 2017"
trackNumber = 99
artistName = "Fred"
albumName = "Best of Fred"
```

The main information about a song is given in the table of voices with the voice names, midi data, audio and reverb levels and the sound variants. As mentioned before only bass and guitar have an audio postprocessing.

```
voiceNameList      =  "vocals,    bass,  guitar,   drums"
midiChannelList    =  "     2,       3,       4,      10"
midiInstrumentList =  "    54,      35,      29,      18"
midiVolumeList     =  "   100,     120,      70,     110"
panPositionList    =  "     C,    0.3R,    0.8R,    0.1L"
audioLevelList     =  "   1.0,    0.83,    0.33,    1.48"
reverbLevelList    =  "   0.3,     0.2,     0.0,     0.4"
soundVariantList   =  "  COPY, EXTREME,  CRUNCH,    COPY"
```

We also have lyrics: two lines of lyrics in vocals extract and score, one (serialized) line in the video.

```
voiceNameToLyricsMap = "{ vocals : e2/s2/v }"
```

Humanization relies on the humanization style defined in 8.2.1. It applies to all voices except vocals and starts in measure 1.

```
styleHumanizationKind = "humanizationStyleRockHard"
humanizedVoiceNameSet = "bass, guitar, drums"
measureToHumanizationStyleNameMap = \
    "{ 1 : humanizationStyleRockHard }"
```

The overall tempo is 85bpm throughout the song.

```
measureToTempoMap = "{ 1 : 85 }"
```

8.3 Putting it All Together

Now we are set to start the tool chain. Assuming that the configuration is in file "wonderful_song-config.txt" and the lilypond stuff is in "wonderful_song-music.ly", the command to produce everything is

```
lilypondToBVC --phases all wonderful_song-config.txt
```

and it produces the following target files

- in directory "generated" the extracts "wonderful_song-bass.pdf", "wonderful_song-drums.pdf", "wonderful_song-guitar.pdf" and "wonderful_-

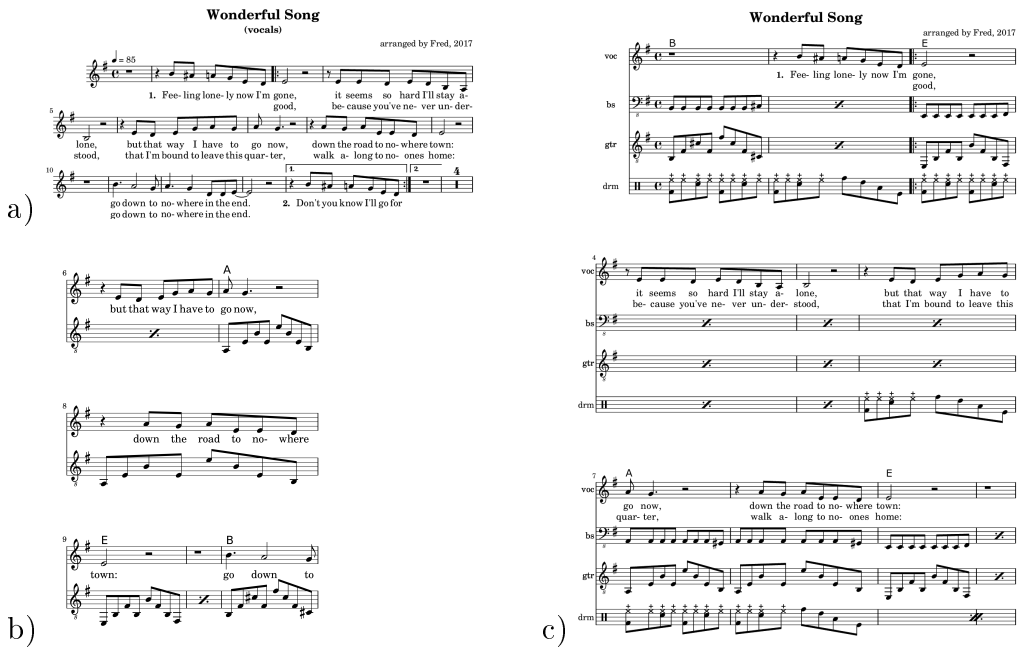

Figure 23: Examples for Target File Images

song-vocals.pdf",

- the score file "generated/wonderful_song_score.pdf",
- the midi file "generated/wonderful_song-std.mid",
- in directory " /musicFiles" the audio files "test-wonderful_song.m4a" and "test-wonderful_song-v.m4a", and
- the video file with two audio tracks " /videos/test-wonderful_song-tblt.mp4"

Figure 23 shows an extract page (a), one image of the target video (b) and the first score page (c) as an illustration.

9. Debugging

Several tools are orchestrated by the script and typically something goes wrong. The script or one of the underlying tools issues some error message, but how can you find out what really went wrong?

The first place to look is the logging file located in loggingFilePath. It does a very fine-grained tracing of the relevant function calls and the last lines should give you some indication about the error.

Note that the outputs of the called programs are not logged, but at least the commandlines to call them. This would not be helpful in itself, because typically those programs work on generated intermediate files. But you can tell ltbvc to keep the intermediate files by setting keepIntermediateFiles to true or alternatively calling the program with the "-k" flag. This only applies to the preprocessing phases, because in the postprocessing phases all files are kept as they serve as input for other phases [1].

For example, assume that the score generation phase does not produce a meaningful output. If you have set the keep-files-flag, then a file called "temp.ly" is produced and kept that contains the boiler-plate code for the score. You can then run

```
lilypond test.ly
```

and see what happens. Of course, you must be able to get by with the lilypond messages, but this is plain lilypond expertise.

Assuming default settings of the configuration variables, the following temporary files will be produced:

extract:
a single temp.ly file containing a single voice,

score:
a single temp.ly file for the complete score,

midi:
a single temp.ly file for the midi voices and a generated ".mid" file containing the voices with standard sound assignment and no humanization, and

silentvideo:
a single temp.ly file for the video voices, ".png" image files with single pages of the video and ".mp4" files containing the parts of the video showing just a single page.

[1] The silent videos and the subtitle file also go into the intermediate file directory, because they are not interesting in themselves, but must be kept.

For the postprocessing phases all intermediate files are kept as follows:

rawaudio:
each voice wave-file goes into the path specified by tempAudioDirectoryPath as "«voice».wav",

refinedaudio:
each voice wave-file goes into the path specified by tempAudioDirectoryPath as "«voice»-processed.wav",

mixdown and finalvideo:
both phases only have target files in audioTargetDirectoryPath and the target specific path in targetVideoDirectory.

Most problems in postprocessing probably occur in the "refinedaudio" phase, because sox does a lot of complex transformations. It might be helpful to insert "tee" commands in the sox processing chain in the command file to have a peek at intermediate audio stages.

Be aware that "tee" is not a standard sox command: if you execute the sox steps directly on the command line, you must take care of any intermediate files yourself.

10. Future Extensions

The following things are not contained in the current version, but are planned for future versions:

- The audio processing chain during the postprocessing phase is currently linear. It is planned to allow complex function graphs (DAGs) for audio processing.
- The sound variant list (describing a single sound variant for each voice) shall be replaced by map from voice to a map from measure to sound variant. This allows to have individual sound styles for different parts in a song (like, for example, for an instrument solo part where special sounds are required).
- The algorithm for finding the measures for the page breaks for the video is quite naive and fragile. The page breaks are currently found by scanning the Lilypond Postscript file, because to my knowledge Lilypond currently has no means for providing the location of those breaks programmatically. Some better solution must be found.
- Currently the humanization algorithm can only cope with a single time signature for the complete song and uses the same (measure-specific) humanization pattern for all voices. Similarly to the sound variants a map should be used from measure and voice to the humanization pattern.
- In professional audio productions drums are processed by handling the different drum instrument groups (e.g. kick, snare, toms, cymbals) individually. This is currently not possible: drums are simply a single audio voice. A workaround could be done, if the used midi-to-wav-converter produced a multi-channel result and the refinement stage were able to combine those parts into a final result. But possibly this must go into the workflow itself.

List of Figures

A. Table of Configuration File Variables

The following table describes all the configuration variables with their default values and the figure numbers where those variables have been mentioned first in the current document.

Variable	Description	Default	Fig.
aacCommandLine	aac encoder command line with parameters for input (${infile}) and output (${outfile}) (optional, if not defined ffmpeg is used for aac encoding)	empty	3
audioRefinementCommandLine	audio refinement command line with parameters for input (${infile}) and output (${outfile}); this commandline is optional when soxCommandLinePrefix is set, otherwise it is mandatory	empty, must be set when soxCommandLinePrefix is empty	3
albumName	album for song group (embedded as "album" in audio and video files)	"UNKNOWN ALBUM"	5
artistName	artist of that song group (embedded as "artist" and "album artist" in audio and video files)	"UNKNOWN ARTIST"	5
attenuationLevel	decimal value in decibels telling the volume change to be applied to the final audio files; this is helpful to adjust volume levels of different songs within an album	0	22
audioGroupToVoicesMap	mapping from freely defined voice group names to names of voices contained in that group described by a slash-separated name list	**MANDATORY**	22
audioLevelList	list of volume factors aligned with the list voiceName used for mixing the refined audio files into cumulated audio files; the factors are decimal values with 1.0 meaning that the refined voice file is taken unchanged	**MANDATORY**, compatible to voiceNameList	22
audioTargetDirectoryPath	path for the final AAC audio files with subsets of rendered and refined audio tracks	current directory	22
audioTrack.albumName	name of the album of the audio file for given list of voices (where an embedded dollar-sign is replaced by the global album name)	albumName	21
audioTrack.audioFileTemplate	template string defining how the audio file name of the target audio file for given list of voices is constructed from the plain audio file name (indicated by a dollar-sign)	**MANDATORY**	21
audioTrack.audioGroupList	slash-separated list of audio group names occuring as keys in audioGroupToVoicesMap	**MANDATORY**	21
audioTrack.description	description for audio track within target video (typically unsupported by video players)	empty	21
audioTrack.languageCode	ISO language code for audio track within target video (typically supported by video players)	eng	21
audioTrack .songNameTemplate	template string defining how the song name for given list of voices is constructed from the plain song name (indicated by a dollar-sign)	title	21

Variable	Description	Default	Fig.
audioTrackList	list of track descriptors defining groups of audio groups to be put on some track with naming templates for audio file, song and album name and a track description and language	**MANDATORY**	22
audioVoiceNameSet	set of voice names to be rendered to audio files via the phases "rawaudio" and "refinedaudio" based on voice representations in humanized midi file	voiceNameList	20
composerText	composer text to be shown in voice extracts and score	empty	6
countInMeasureCount	number of count-in measures for the song (which defines the time before the first measure)	0	14
extractVoiceNameSet	set of voices to be rendered as a voice extract	voiceNameList	10
ffmpegCommand	location of ffmpeg command	**MANDATORY**	3
fileNamePrefix	file name prefix used for all generated files for this song	**MANDATORY**	6
humanizationStyleXXX	map that tells the initial count-in measures, the variation in timing and velocity for several positions within a measure	empty	14
humanizedVoiceNameSet	set of voice names to be humanized by random variations of timing and velocity	empty	14
includeFilePath	path for the music include file containing all fragments for lilypond processing; if unset, defaults to fileNamePrefix plus "-music.ly"	fileNamePrefix plus "-music.ly"	4
intermediateFileDirectoryPath	path of directory where intermediate files go that are either used for processing within a phase or as information between phases	current directory	4
keepIntermediateFiles	boolean telling whether temporary files are kept	false	6
lilypondCommand	location of lilypond command	**MANDATORY**	3
lilypondVersion	the version string for lilypond	**MANDATORY**	3
loggingFilePath	path of file containing the processing log	**MANDATORY**	4
measureToHumanization-StyleNameMap	map of measure number to humanization style name used from this position onward for humanized voices	empty	14
measureToTempoMap	map defining the tempo for measure in bpm until another tempo setting is given; the time signature as a fraction may be appended after a vertical bar (4/4 is default)	**MANDATORY**	6
midiChannelList	list of midi channels per voice each between 1 and 16 (10 for a drum voice)	**MANDATORY**, compatible to voiceNameList	13
midiInstrumentList	list of midi instrument programs per voice each as an integer between 0 and 127; each entry may be prefixed by a bank number (0 to 127) followed by a colon	**MANDATORY**, compatible to voiceNameList	13
midiPanList	list of pan positions per voice as a decimal value between 0 and 1 with suffix "R" or "L" (for right/left) or the character "C" (for center)	**MANDATORY**, compatible to voiceNameList	13
midiToWavRendering-CommandLine	command line for rendering command from MIDI file to WAV audio file (typically "fluidsynth")	**MANDATORY**	3
midiVoiceNameList	list of voices to be rendered in order given into the MIDI file	voiceNameList	13

Variable	Description	Default	Fig.
midiVolumeList	list of midi volumes per voice each as an integer between 0 and 127	**MANDATORY**, compatible to voiceNameList	13
mp4boxCommand	location of mp4box command	**MANDATORY**	3
phaseAndVoiceName-ToClefMap	mapping from processing phase to maps from voice name to lilypond clef	empty	8
phaseAndVoiceName-ToStaffListMap	mapping from processing phase to maps from voice name to slash-separated lilypond staff names	empty	8
reverbLevelList	list of reverb levels (as decimal values typically between 0 and 1) for the voices aligned with the list voiceNameList; those reverb levels are applied to each voice as the final refinement operation	list of 0.0	20
scoreVoiceNameList	list of voices to be rendered in order given into the score	voiceNameList	12
soundStyleXXX	sequence of (sox) refinement commands to be applied on raw audio file when this style is selected for «voice»	empty	20
soundVariantList	list of variant names for the sound styles of the voices aligned with the list voiceName; those style variant names are combined into a complete style name to be applied during audio refinement	**MANDATORY**, compatible to voiceNameList	20
soxCommandLinePrefix	command line prefix for sox audio filter with global options (like buffering or multithreading settings); if missing, (slow) internal routines will be used for audio shifting and mixdown	empty, must be set when audioRefinementCommandLine is empty	3
targetDirectoryPath	path of directory where all generated files go (except for audio and video files)	current directory	4
tempAudioDirectoryPath	path of directory for temporary audio files	current directory	4
tempLilypondFilePath	path of temporary lilypond file	temp.ly in current directory	4
title	human visible title of song used as tag in the target audio file and as header line in the notation files	**MANDATORY**	6
trackNumber	track number within album	0	6
videoFileKind.directoryPath	directory where final videos for that target go	current directory	17
videoFileKind.fileNameSuffix	suffix to be used for the video file names for that target	**MANDATORY**	17
videoFileKind.target	name of associated video target that is used when rendering video files of that kind	**MANDATORY**	17
videoFileKind.voiceNameList	list of voice names to be rendered in order to audio files via the phase "silentvideo"	voiceNameList	17
videoTarget .ffmpegPresetName	a specific ffmpeg preset for the current video target device (a string, a missing value defaults to a baseline level 3 profile)	""	15
videoTarget.frameRate	the frame rate of the video (in frames per second)	10	15
videoTarget.height	height of device and video (in dots)	**MANDATORY**	15
videoTarget.leftRightMargin	margin for video on left and right side (in millimeters)	**MANDATORY**	15
videoTarget.mediaType	the Quicktime media type of the video (for example "TV Show")	"TV Show"	15
videoTarget.resolution	resolution of the device (in dpi)	**MANDATORY**	15

Variable	Description	Default	Fig.
videoTarget.scalingFactor	the factor by which width and height are multiplied for lilypond image rendering to be downscaled accordingly by the video renderer (an integer)	1	15
videoTarget.subtitleColor	color of overlayed subtitle in final video for measure display (as integer for 16bit alpha/red/green/blue)	(yellow)	15
videoTarget.subtitleFontSize	height of subtitle (in pixels)	10	15
videoTarget .subtitlesAreHardcoded	flag to tell whether subtitles are burnt into the video or are available as a separate subtitle track	false	15
videoTarget.systemSize	size of lilypond system (in lilypond units, cf. lilypond system size)	20 (default of lilypond)	15
videoTarget.topBottomMargin	margin for video on top and bottom (in millimeters)	**MANDATORY**	15
videoTarget.width	width of device and video (in dots)	**MANDATORY**	15
videoTargetMap	mapping from video target name to video target descriptor with several parameters for specific video file generation	**MANDATORY**	18
voiceNameToChordsMap	mapping from voice names to phase abbreviations where chords are shown for that voice system	empty	8
voiceNameToLyricsMap	mapping from voice name to a count of parallel lyrics lines directly following the target letter ("e" for the extract, "s" for the score and "v" for the video)	empty	8
voiceNameToOverride-FileNameMap	map from voice name to name of file overriding that voice in the processed audio files and in the final mixdown audio files and in the target videos	empty	20
voiceNameToScoreNameMap	mapping from voices name to short score name at the beginning of a system	empty	12
voiceNameToVariation-FactorMap	map from voice name to a pair of decimal factors characterizing the timing and velocity variation for this kind of voice to be applied additional to the humanization style	empty	14
year	year of arrangement	current year	6

B. Glossary

album
→*song group*

all (phase group)
a group of →*processing phase*s doing full processing via phase groups →*preprocess* and →*postprocess*

audio group
a group of →*voice* →*audio track*s to be mixed into a target audio file or into a single audio track in the target video files

audio track
the audio rendering of a subset of all song voices (typically within the final notation video)

(song) configuration file
a text file containing configuration information for a single →*song* (possibly including other text configuration files) that is used in generation of wrapper →*lilypond* files and parametrization of underlying generation programs; consists of key-value pairs with variable names as keys followed by an equal sign and a string, boolean or numeric value

extract (phase)
a →*processing phase* producing the extract PDF notation files for single →*voice*s using the program →*lilypond*

ffmpeg
a command-line program for producing videos from notation page images, inserting hard subtitles into them and possibly combining those silent videos with audio tracks (when →*mp4box* is not used for that)

finalvideo (phase)
a →*processing phase* generating final video files for each →*video file kind* with all submixes as selectable audio tracks and with a measure indication as subtitle using the programs →*ffmpeg* and optionally →*mp4box*

fluidsynth
a command-line program for conversion of MIDI files into WAV audio files (representing →*audio track*s) using →*sound font*s

humanization
a part of the →*midi* phase applying algorithmic and rule-based random time and volume (velocity) shifts to notes in the midi stream of →*voices*

humanization style
: the configuration information for →*humanization* of a →*song* telling individual variations based on the position of a note within a measure; gives timing and velocity variations for the main beats, the other sixteenths and all other notes; multiple styles may be given for a song for non-overlapping measure ranges

lilypond
: a typesetting program transforming text files with music notation information into PDF or MIDI files

lilypond fragment file
: a text file with fragmentary →*lilypond* typesetting information; based on a song-specific →*configuration file* the generator provides wrapping lilypond code and calls the appropriated underlying programs

midi (phase)
: a →*processing phase* producing a MIDI file containing all →*voice*s with specified instruments, pan positions and volumes using the program →*lilypond* plus some →*humanization*

mixdown (phase)
: a →*processing phase* generating final compressed audio files with submixes of all instruments →*voice*s based on the refined audio files with specified volume balance (where the submix variants are configurable) using the program →*sox*

mp4box
: a command-line program for combining the silent notation videos with →*audio track*s; used optionally instead of →*ffmpeg* for a better compatibility with Apple devices

override (of a voice audio)
: a replacement of the refined audio file for some →*voice* by an external audio file to be applied in the →*refinedaudio* phase; is normally applied when the external file has a higher quality (like, for example, with a real singer instead of a vocals instrumental rendition)

parallel track (audio)
: an additional audio file to be added in the →*mixdown* phase; this is used for a single external audio file not associated with some voice (like, for example, background sounds)

preprocess (phase group)
: a group of →*processing phase*s combining →*extract*, →*score*, →*midi* and →*silentvideo* for generation of →*voice* extract PDFs and score PDF, MIDI file as well the silent videos for all →*video file kind*s

postprocess (phase group)
: a group of →*processing phase*s combining →*rawaudio*, →*refinedaudio*, →*mixdown* and →*finalvideo* for generation of the intermediate raw and refined WAV files, the submixes as compressed audios and the final videos for all →*video file kind*s

processing phase
: a part of the generation of →*song* artifacts from given →*lilypond fragment file* and →*configuration file*; possible processing phases or processing phase groups are →*all*, →*preprocess*, →*postprocess*, →*extract*, →*score*, →*midi*, →*silentvideo*, →*rawaudio*, →*refinedaudio*, →*mixdown* and →*finalvideo*

qaac
: a command-line program for converting WAV audio files into aac encoded audio files (representing →*audio groups*); used optionally instead of →*ffmpeg* for a better encoding quality

rawaudio (phase)
: a →*processing phase* producing unprocessed (intermediate) audio files for all the instrument →*voice*s from the midi tracks using the program →*fluidsynth* plus some →*sound fonts*

refinedaudio (phase)
: a →*processing phase* producing (intermediate) audio files for all the instrument →*voice*s with additional sound processing applied by the program →*sox*

score (phase)
: a →*processing phase* producing a single PDF notation file containing all →*voice*s as a score generated by the program →*lilypond*

silentvideo (phase)
: a →*processing phase* to generate (intermediate) silent videos containing the score pages for several output →*video target*s (with configurable resolution and size) using →*ffmpeg* as the video generator from notation pages produced by →*lilypond*

song
: a collection of several parallel →*voice*s forming a musical piece

song group
: a collection of several related →*song*s (for example, related by year, artist, etc.) sharing common characteristics

sound font (file)
: a file containing data for a sample-based rendering of MIDI data as au-

dio files; the generator uses the →*fluidsynth* program for this conversion within the →*rawaudio* phase

sound style
: a (sequential) chain of →*sox* audio filters to be applied to a an audio rendering of a →*voice* in phase →*refinedaudio*; typically those sound styles are instrument specific

sox
: a program for transformation of audio files via parametrizable audio filters (like, for example, equalizers, distortions or reverbs) used in the →*refinedaudio* and →*mixdown* phases

video file kind
: the configuration information used in the →*silentvideo* and →*finalvideo* phases giving video rendering properties of notation videos extending characteristics of a →*video target* by data (like, for example, the list of voices to be shown or the video files target directory)

video target
: the configuration information used in the →*silentvideo* and →*finalvideo* phases giving video device dependent properties of notation videos (like, for example, device resolution or pixel width and height), but also some device independent parameters (like, for example, the subtitle font size)

voice
: a polyphonic part of a composition belonging to a single instrument to be notated in one or several musical staffs

C. References

[AAC] *QAAC - Quicktime AAC.*
https://sites.google.com/site/qaacpage/

[FFMPEG] *FFMPEG - Documentation.*
http://ffmpeg.org/documentation.html

[FLUID] *FluidSynth - Software synthesizer based on the SoundFont 2 specifications.*
http://fluidsynth.org

[LILY] *Lilypond - Music Notation for Everyone.*
http://lilypond.org

[MP4BOX] *GPAC - General Documentation MP4Box.*
https://gpac.wp.imt.fr/mp4box/mp4box-documentation/

[SOUNDFONT] *FluidR3_GM.sf3 SoundFont at musescore.org.*
https://github.com/musescore/MuseScore/raw/2.1/share/sound/FluidR3Mono_GM.sf3

[SOX] Chris Bagwell, Lance Norskog et al.: *SoX - Sound eXchange - Documentation.*
http://sox.sourceforge.net/Docs/Documentation