

# ***SoundFontTools***

Tools for Conversion, In-Place-Renaming and  
Analysis of SoundFonts (v0.1)

Dr. Thomas Tensi

November 7, 2025



# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>                               | <b>7</b>  |
| 1.1      | Overview . . . . .                                | 7         |
| 1.2      | Outline of this Document . . . . .                | 7         |
| <b>2</b> | <b>Preliminaries</b>                              | <b>9</b>  |
| 2.1      | Requirements . . . . .                            | 9         |
| 2.2      | Installation . . . . .                            | 9         |
| 2.3      | Build . . . . .                                   | 9         |
| <b>3</b> | <b>Usage</b>                                      | <b>11</b> |
| 3.1      | jsonToSoundFont . . . . .                         | 11        |
| 3.2      | soundFontAnalyser . . . . .                       | 13        |
| 3.2.1    | Modulator Analysis . . . . .                      | 15        |
| 3.2.1.1  | Modulation Functions . . . . .                    | 16        |
| 3.2.1.2  | Projection to One-Dimensional Functions . . . . . | 17        |
| 3.2.1.3  | Example Analysis . . . . .                        | 18        |
| 3.3      | soundFontToJSON . . . . .                         | 20        |
| 3.4      | soundFontInPlaceRenamer . . . . .                 | 22        |
| 3.5      | Using the Tools with the Demo SoundFont . . . . . | 24        |
| <b>4</b> | <b>Debugging</b>                                  | <b>27</b> |
| <b>5</b> | <b>References</b>                                 | <b>29</b> |
| <b>A</b> | <b>Glossary</b>                                   | <b>31</b> |
| <b>B</b> | <b>Configuration File Overview</b>                | <b>33</b> |
| B.1      | Configuration File Location . . . . .             | 33        |
| B.2      | Configuration File Syntax . . . . .               | 33        |
| <b>C</b> | <b>SoundFont UML Model</b>                        | <b>37</b> |
| C.1      | Elementary_Types . . . . .                        | 37        |
| C.2      | SoundFont_Types . . . . .                         | 39        |



# List of Figures

|    |  |    |
|----|--|----|
| 1  | Mapping from Generator Kind to Unit Kind . . . . .   | 12 |
| 2  | Mapping from Unit Kind to Unit String for Instrument and<br>Preset Zones . . . . .                   | 13 |
| 3  | Concave Functions in FluidSynth and SoundFontAnalyser . .  | 16 |
| 4  | Comparison of Projected One-Dimensional Generator Value<br>Functions with 20% Error Margin . . . . . | 18 |
| 5  | Approximation of Staircase Generator Value Function . . . . .  | 20 |
| 6  | Example for Logging File Extract from SoundFontInPlaceRe-<br>namer . . . . .                         | 28 |
| 7  | UML Model — Elementary Types - Overview . . . . .  | 37 |
| 8  | UML Model — SoundFont Types - Overview . . . . .   | 39 |
| 9  | UML Model — SoundFont Types - Global SoundFont Data . .  | 40 |
| 10 | UML Model — SoundFont Types - Named and Identified<br>Elements . . . . .                             | 40 |
| 11 | UML Model — SoundFont Types - Samples . . . . .  | 41 |
| 12 | UML Model — SoundFont Types - Zoned Elements . . . . .   | 42 |
| 13 | UML Model — SoundFont Types - Generators . . . . .   | 42 |
| 14 | UML Model — SoundFont Types - Modulators . . . . .   | 43 |



# 1. Introduction

## 1.1 Overview

The *SoundFontTools* is an suite of several python scripts that allow to read, modify, analyse and write SoundFont files.

They consist of

- a converter from a SoundFont file to a JSON file plus wave files for the samples,
- a converter from a JSON file plus sample wave files to a SoundFont file,
- an in-place renaming utility for doing a pattern-based adaptation of the sample, instrument and preset names within a SoundFont file, and
- a SoundFont file analyser scanning for possible optimizations in a SoundFont file.

All those tools should help a SoundFont designer or someone analyzing existing SoundFonts.

## 1.2 Outline of this Document

This document will present how to setup and use the *SoundFontTools*.

- Chapter 2 describes the installation process.
- Chapter 3 tells how to use the (command line) python programs.
- Because things will certainly go wrong some time, chapter 4 gives some hints on how to trace the problem.
- Further bibliographic information and links to the tools are given in chapter 5.
- Appendix A gives a glossary of terms.
- Appendix B shows the syntax of a configuration file used by some of the programs. It consists of key-value-pairs; the keys are identifiers, but the values may be a bit more complicated.
- Appendix C gives an overview of the internal model of the SoundFont in the Unified Modeling Language [UML17] which is reflected in the JSON representation.

## 1.2. *OUTLINE OF THIS DOCUMENT*

---

- Appendix D lists all the release changes.



## 2. Preliminaries

### 2.1 Requirements

All the scripts are written in python and can be installed as a python package. The package requires Python 3.10 or later.

### 2.2 Installation

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

```
pip install soundfonttools
```

Once installed the program is ready for use. Make sure that the scripts directory of python (e.g. «...»/python/scripts) is in the path for executables on your platform.

In the site-packages directory you can find the python source files and two directories **demo** and **doc**. **doc** contains this documentation PDF and **demo** the demo soundfont plus the configurations files for the simple tests described in section 3.5.

### 2.3 Build

You can build the tools yourself from the repository by cloning the GIT repository from <https://github.com/prof-spock/SoundFontTools> and building it via the standard python mechanisms as follows:

```
git clone https://github.com/prof-spock/SoundFontTools
python -m build --wheel SoundFontTools
python3 -m pip install SoundFontTools/dist/xxx.whl
```



## 3. Usage

### 3.1 jsonToSoundFont

The conversion from a JSON file to a SoundFont file by the program `jsonToSoundFont` is done via the following command line:

```
jsonToSoundFont [-h] [-l loggingFilePath] [-sf i16/i24]
                 -i jsonFilePath -w waveFileDirectoryPath
                 -o soundFontFilePath
```

The options have the following meaning:

- h**  
makes the program show all the command line options and exit
- i jsonFilePath**  
defines the path of the JSON source file
- l, --logging\_file loggingFilePath**  
defines the path for the logging file and activates logging
- o soundFontFilePath**  
defines path of the SoundFont destination file to be written
- sf, --sample\_format i16/i24**  
gives the sample format for the sound font: for 'i16' samples are stored in a 16-bit integer format, for 'i24' samples are stored in 24-bit integer format (assuming that the SoundFont version is at least 2.04)
- df, --debug\_file\_path debugFilePath**  
defines an optional path to a JSON file showing the data read into internal SoundFont model from input JSON file
- w, --wave\_file\_directory waveFileDirectoryPath**  
defines the path to the source directory for wave files

The JSON input is read from the file given by the `-i` option. Note that it is assumed that the file is in ASCII encoding, because SoundFonts may only use ASCII strings.

If there are problems with the given data, error messages are written to `stderr` and *no soundfont is generated*. Otherwise the soundfont file specified as output via the `-o` option is generated.

For analyzing problems it is possible to specify a debug text file via the command line option `-df`. It is filled with the text representation of the internal SoundFont model generated from the given JSON input. Normally

### 3.1. JSONTOSOUNDFONT

| Generator Kind  | Unit Kind          | Description   |
|---|--------------------|---|
| fineTune, keynumToModEnvDecay, keynumToModEnvHold, keynumToVolEnvDecay, keynumToVolEnvHold, modEnvToFilterFc, modEnvToPitch, modLfoToFilterFc, modLfoToPitch, vibLfoToPitch   | cent               | cents (with an interpretation depending on the context)   |
| scaleTuning   | constant           | a constant applicable to both instruments and presets     |
| endAddrOffset, endAddrCoarseOffset, endLoopAddrCoarseOffset, endLoopAddrOffset, exclusiveClass, keynum, overridingRootKey, sampleID, startAddrCoarseOffset, startAddrOffset, startLoopAddrCoarseOffset, startLoopAddrOffset, velocity | constantForInst    | a constant applicable to instruments only                 |
| instrument  | constantForPrst    | a constant applicable to presets only                     |
| initialFilterQ, modLfoToVolume, sustainVolEnv   | decibel            | decibels  |
| initialAttenuation  | decibelAttenuation | decibel attenuation (may only take non-negative values)   |
| freqModLFO, freqVibLFO, initialFilterFc   | hertz              | frequency in Hz   |
| keyRange, velRange  | pair               | a pair of two byte values from 0 to 127                   |
| chorusEffectsSend, pan, reverbEffectsSend, sustainModEnv  | percent            | a percentage (typically from 0 to 100)                    |
| sampleModes   | sampleMode         | sample loop mode either given as name or number (or both) |
| attackModEnv, attackVolEnv, decayModEnv, decayVolEnv, delayModEnv, delayModLFO, delayVibLFO, delayVolEnv, holdModEnv, holdVolEnv, releaseModEnv, releaseVolEnv  | second             | seconds   |
| coarseTune  | semitone           | semitones   |

Figure 1: Mapping from Generator Kind to Unit Kind

the input and the debug file should have equivalent information, although - of course -, the sequence of information in both files may be different.

Within the generated SoundFont the chunks are ordered in canonical order as specified by the SoundFont specification [SFTSpec06].

The order of generators for a zone is key/velocity range first (if any), then the normal generators in numerical order from the specification and sample or instrument index (if any) last.

The generator amounts may have optional units. It is differentiated whether the generators are additive or multiplicative and whether they are in an instrument or preset zone.

The table in figure 1 gives the unit kind per generator kind.

The allowed strings per unit are given by the table in figure 2. Note that the

| Unit Kind          | Unit String in Instruments | Unit String in Presets |
|--------------------|----------------------------|------------------------|
| cent               | ct                         | ct                     |
| constant           | —                          | —                      |
| constantForInst    | —                          | n/a                    |
| constantForPrst    | n/a                        | —                      |
| decibel            | dB                         | dB                     |
| decibelAttenuation | dB                         | dB                     |
| hertz              | Hz                         | x                      |
| pair               | —                          | —                      |
| percent            | %                          | %                      |
| sampleMode         | n/a                        | n/a                    |
| second             | s                          | x                      |
| semitone           | st                         | st                     |

Figure 2: Mapping from Unit Kind to Unit String for Instrument and Preset Zones

units strings are case-sensitive and that `x` signifies a multiplication by the given factor. A sample mode is either given as a number or a text or both and has no unit.

## 3.2 soundFontAnalyser

The program `soundFontAnalyser` scans a SoundFont file for possible problems and optimizations. This is done via the following command line:

```
soundFontAnalyser [-h] [-l loggingFilePath] [-p]
                  [-c configurationFilePath] -i soundFontFilePath
```

The options have the following meaning:

**-c configurationFilePath**

(optional) path to the configuration file containing settings for activating or deactivating rules (with its syntax described in appendix B)

**-h**

makes the program show all the command line options and exit

**-l loggingFilePath**

(optional) path for the logging file and activates logging

**-i soundFontFilePath**

path to the SoundFont file to be checked

**-p**

tells to format output in a pretty-printed and human-readable style

The program analyses the SoundFont given and writes the analysis results to the standard output.

It is assumed that the SoundFont file is structurally correct; there are no provisions to analyse a damaged SoundFont.

The following rules are checked:

- **rules for samples:**

- `sampleOverrides`: checks whether some identical generator settings in all usages within instruments can be moved to the sample definition itself

- **rules for instruments:**

- `instrumentGlobalZone`: checks whether identical generator values from non-global zones can be moved to the instrument global zone (when they occur in more than one zone)
- `instrumentModulators`: checks whether generator values in non-global zones of an instrument can be approximated by a modulator (see below)
- `instrumentOverrides`: checks whether some identical generator settings in all usages within presets can be moved to the instrument definition itself

- **rules for presets:**

- `presetGlobalZone`: checks whether identical generator values from non-global zones can be moved to the preset global zone (when they occur in more than one zone)
- `presetModulators`: checks whether generator values in non-global zones of an instrument can be approximated by a modulator (see below)

Normally all rules are checked on all applicable kinds of generators. But one can selectively disable rules or generator kinds checked in a rule via a configuration file given by the `-c` option.

The configuration file contains two kinds of variables: variables defining the checking of a rule and variables excluding specific generator kinds per rule.

Disabling a rule is done via a boolean-valued variable called `rule_«rule name»_isActive`. For example, to deactivate the checking of the `presetModulators` rule, the configuration file should have a line:

```
rule_presetModulators_isActive = false
```

Disabling certain generator kinds for a rule is done via a list-valued variable called `rule_«rule name»_excludedGeneratorKindList`. For example, to deactivate the checking `overridingRootKey` and `startLoopAddrsOffset` in the `sampleOverrides` rule, the configuration file should have a line:

```
rule_sampleOverrides_excludedGeneratorKindList =
[ "overridingRootKey", "startLoopAddrsOffset" ]
```

The distribution contains a demo configuration file containing all rules and also the generator kinds those rules will be applied to.

### 3.2.1 Modulator Analysis

The analyser tries to find out whether data points in several zones may either be approximated by a modulator function with the velocity or with the key value as input.

There are several caveats:

- The interplay between generator values in defaults, global zones, non-global zones, instruments and presets is quite complicated and cannot be easily captured in single function analyses.
- The zones may be overlapping and/or non-consecutive and may be defined by both key range and velocity range. Although modulators may have two parameters practically only either key or velocity can be used for some modulator. Hence an arbitrary two-dimensional function cannot be modelled.
- It is not clear what the exact definition of the curve kinds “concave” and “convex” is. The specification [SFTSpec06] gives a nonsensical formula, while its images resemble circle quadrants; hence the analyser assumes such functions for the curve fitting. There is a slight difference between the log-based concave definition in FluidSynth

$$y = \begin{cases} \frac{-200 \cdot 2}{960 \cdot \log 10} \cdot \log \left( \frac{127 - x}{127} \right) & \text{if } x \in \{1, \dots, 127\} \\ 0 & \text{if } x = 0 \end{cases}$$

and the circle-quadrant-based concave definition used by the analyser

$$y = \sqrt{1 - \left(1 - \frac{x}{128}\right)^2}, \text{ if } x \in \{0, \dots, 127\}$$

(the same holds for convex). Figure 3 show the differences for the concave functions.

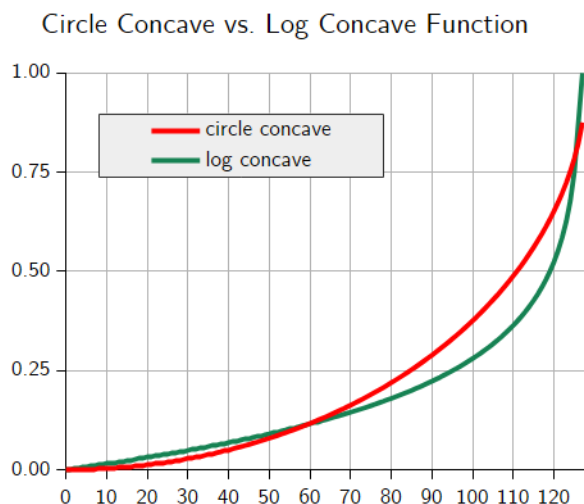


Figure 3: Concave Functions in FluidSynth and SoundFontAnalyser

Note that the sonic differences between those functions should be minor, hence the analyser sticks to the circle-quadrant interpretation of the specification.

- The analyser currently cannot handle a combination of two modulators for a single generator for key number and key velocity even for a well-behaved two-dimensional function. It assumes that some generator value either depends on the key number but not at all on the velocity or vice versa. This covers many situations in SoundFonts, but could be improved.

### 3.2.1.1 Modulation Functions

In the result file the analyser describes the modulator function by its settings `destination`, `controllerKind`, `isUnipolar`, `isAscending`, `curveKind`, `factor`, `isAbsolute` and `offset`. The first six values can directly be used in the modulator, the offset has to be added to the current value of that generator in the global zone (if possible).

Additionally a comparison is done between the values given in the zones and the values provided by the modulator in the center of those zones. Note that there typically is some deviation between those values and, of course, the modulator function is *not a staircase function*, but a continuous function on the controller value.

To ensure that only reasonable modulators are recommended by the analysis, one can specify a bound for the “normalized mean square error” *nrmse* which



is mathematically defined by

$$nrmse = \frac{mse(S, M)}{\text{Var}(S)} \text{ with } mse(S, M) = \frac{\sqrt{\sum_i (s_i - m_i)^2}}{n} \text{ with } \text{Var}(S) = \frac{\sum_i (s_i - \bar{s})^2}{n}$$

assuming that the set  $S = \{s_i\}, i \in \{1, \dots, n\}$  are the function values and the set  $M = \{m_i\}, i \in \{1, \dots, n\}$  the estimates.

In our case  $S$  are the staircase values and  $M$  is the approximation by the modulator. So to calculate the *nrmse* one considers each generator value  $i$  from zero to 127 and subtracts the staircase value and the modulator value and squares the result; then all those squares are summed up and the square root of that sum is taken. The result gives a measure of the distance between the two functions.

Finally that value is divided by the standard deviation of the staircase values. This scales the *nrmse* appropriately, because, for example, an average distance of 1.0 between staircase function and modulator would be bad for a staircase function ranging from -2.0 to 2.0 while it is totally acceptable for a staircase function ranging from -50 to 50.

It is possible to define the maximum acceptable *nrmse* in the configuration file by the variable `maximumNormalizedRootMeanSquareError`, for example, by:

```
maximumNormalizedRootMeanSquareError = 0.3
```

This allows some minor deviation of the modulator value from the manual staircase setting and is a good starting point for the analysis (and is also the default value).

Note that because the analyser analyses the staircase function given by manual values along its complete domain, the recommended modulator function approximates that function **broadly**. When the error range specified is large, individual deviations in the function values between approximation and desired value may be significant. In that case either a smaller error bound should be given or some adaptation of the staircase function should be done such that the analyser might find a more convincing fit.

### 3.2.1.2 Projection to One-Dimensional Functions

As mentioned before the analyser does not do a two-dimensional analysis of the zones with respect to both key and velocity range. Instead it finds out whether some generator value is either dependent on key num only (and independent on velocity) or vice versa.

This is done by slicing the two-dimensional generator function in each of the dimensions and comparing whether the resulting one-dimensional functions are more or less equal.

There is a configuration value for that called `comparisonMarginForOneDFunctions` that gives the maximum acceptable deviation per value.

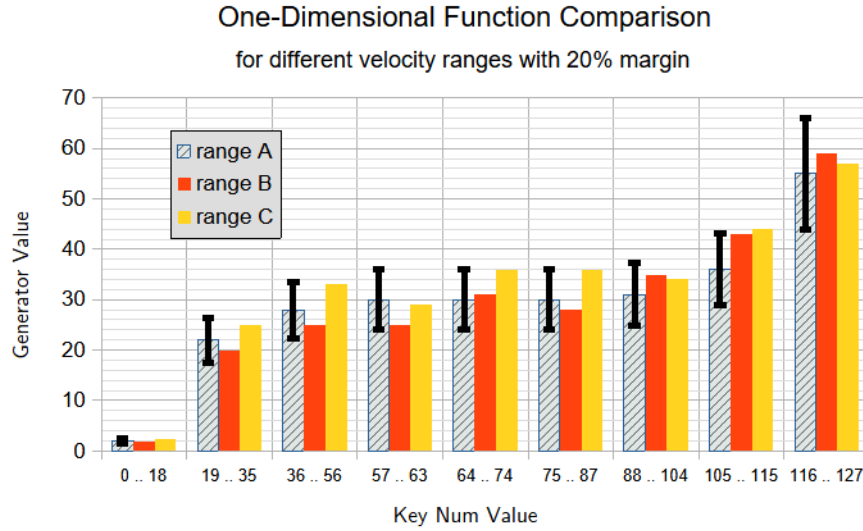


Figure 4: Comparison of Projected One-Dimensional Generator Value Functions with 20% Error Margin

So a setting of

```
comparisonMarginForOneDFunctions = 0.2
```

tells that a value may differ by at most 20% from some reference value (which is also the default).

Figure 4 shows the approach for an example: we have eight key range zones and obviously three velocity ranges are defined for those zones. The first velocity (“range A”) gives the reference value for the other two: when the generator value for an other velocity zone for the same key range differs by at most the margin given, the function values are considered “equal”. As one can see when looking at the functions values of the ranges, this holds for all intervals.

Technically all those x-intervals are expanded to single x-values before comparison; hence the analyser can cope with different intervals.

### 3.2.1.3 Example Analysis

Let us assume that a manually defined function for `generatorXYZ` is given as shown in figure 4. Previous analysis has shown that it does mostly only depend on the key number (as indicated in the figure).

So we can use the following mapping from the key num intervals to generator values as:

|              |               |               |               |               |               |                |                 |                 |
|--------------|---------------|---------------|---------------|---------------|---------------|----------------|-----------------|-----------------|
| <b>0..18</b> | <b>19..35</b> | <b>36..56</b> | <b>57..63</b> | <b>64..74</b> | <b>75..87</b> | <b>88..104</b> | <b>105..115</b> | <b>116..127</b> |
| 2            | 18            | 24            | 26            | 27            | 28            | 31             | 38              | 55              |

Analysis for that function shows that the best fit can be achieved by assuming that it is a bipolar function (with domain (-1,1)).

Linear regression for the different curve kinds give the following results (internally!):

| curve:  | linear | concave      | convex | switch |
|---------|--------|--------------|--------|--------|
| factor: | 22.029 | 42.350       | 13.685 | 9.656  |
| offset: | 25.922 | 26.081       | 25.857 | 25.750 |
| nrmse:  | 0.382  | <b>0.288</b> | 0.582  | 0.710  |

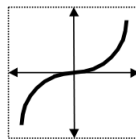
The analyser gives a diagnostic with the approximation function parameters and a comparison for the mid points of the original staircases. For that situation — slightly reformatted, because it is a single line in the analysis file — the result is:

```
data for generator kind 'generatorXYZ' can be provided by a global modulator
(controllerKind = 'key', isUnipolar = False, isAscending = True,
curveKind = concave, factor = 42.350 (=> 42), isAbsolute = False)
and offset = 26.081;
with an overall function distance of 0.2875,
new function = {
  0..18 : 5.386 (for 2.000), 19..35 : 18.286 (for 18.000),
  36..56 : 24.371 (for 24.000), 57..63 : 25.998 (for 26.000),
  64..74 : 26.210 (for 27.000), 75..87 : 27.602 (for 28.000),
  88..104 : 31.755 (for 31.000), 105..115 : 38.986 (for 38.000),
  116..127 : 49.173 (for 55.000)
```

So the concave function wins by a small margin as can be seen in figure 5. The red line represents the original staircase function, the green line the concave modulator function. The other curve kinds are not far off (especially the linear function), but the concave function captures the flat gradient in the center very well.

Now one is able to remove the manual entries for the generator and replace them by a global modulator for that destination generator as follows:

- Since the controller is identified to be `controllerKind = 'Key'`, the first modulator source is **Note-On Key Number**.
- The function transforming that modulator source into the multiplication node is characterized by `isUnipolar = False`, `isAscending = True`, `curveKind = concave`. This means we select a **bipolar, ascending and concave**<sup>1</sup> transformation curve, i.e.



<sup>1</sup>By the way note the visual similarity of the symbol to figure 5

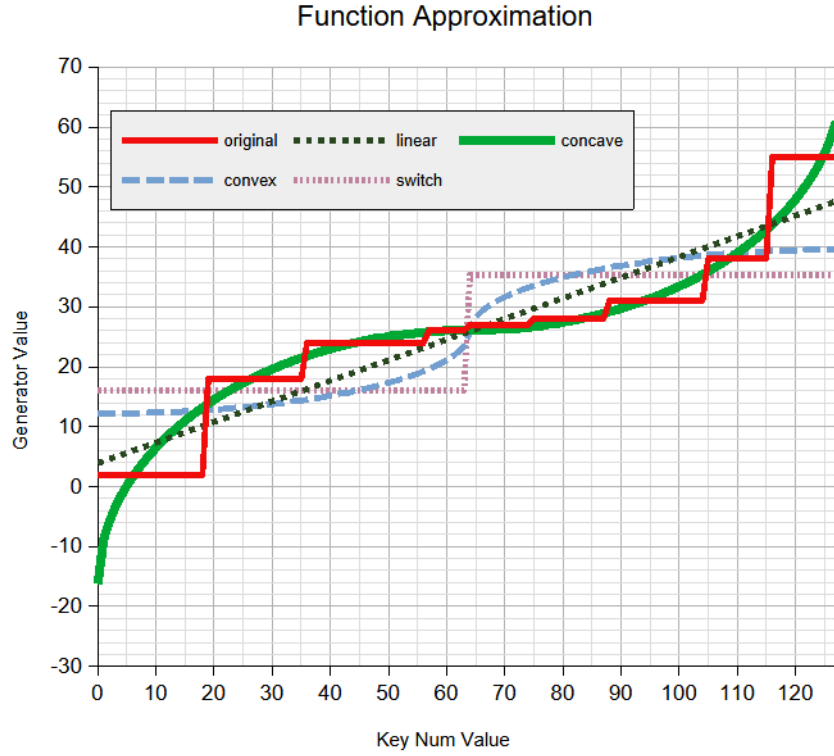


Figure 5: Approximation of Staircase Generator Value Function

- The second modulator source is a **No controller**, = **1**, providing a no-op partner for the multiplication.
- The modulation amount is given by factor = 42.350 ( $\Rightarrow$  42), hence a **42** is entered.
- Because the analysis shows that isAbsolute = False, the outgoing transformation is a **no operation**.
- The affected generator destination is given as **generatorXYZ**.
- Finally we have ensure that offset = 26.081 is taken care of: the value of 26.081 is **added to the current value in the global zone of generatorXYZ**.

### 3.3 soundFontToJSON

The conversion from a SoundFont to a JSON file by the program soundFontToJSON is done via the following command line:

```
soundFontToJson [-h] [-l loggingFilePath] [-w waveFileDirectoryPath]
                 [-idx uuid/nat] [-sn i/n] [-sf i16/i32/f32]
                 -i soundFontFilePath
```

The options have the following meaning:

**-h, --help**

makes the program show all the command line options and exit

**-i soundFontFilePath**

defines the path to the SoundFont source file to be converted

**-l, --logging\_file loggingFilePath**

defines the path for the logging file and activates logging

**-w, --wave\_file\_directory waveFileDirectoryPath**

defines the path to the destination directory for wave files; if this option is not set, only the plain JSON output is produced

**-idx, --indexing\_kind uuid/nat**

tells the kind of identification indexing kind for all objects: 'uuid' generates unique identification indices, 'nat' numbers the objects with natural numbers starting at zero

**-sn, --sample\_naming i/n**

tells the naming conventions for the wave file names (if they are generated): 'i' uses the sample identification, 'n' uses the sample name

**-sf, --sample\_format n/i16/i24/i32**

gives the sample format of the wave files (if they are generated): 'i16' is a 16-bit integer format, 'i24' is a 24-bit integer format, 'i32' a 32-bit integer format; 'n' stands for the native format (either i16 or i24 depending on the SoundFont version)

The JSON output is written to the standard output and can be redirected to a JSON file.

Within the JSON output the generators are ordered in the order of the Polyphone SoundFont Editor [Polyphone] (for an easier comparison). Also the units as given in figure 2 are used for the generator amounts.

Normally all the elements (samples, instruments, presets) are indexed numerically, which means that a sample has “SHDR” followed by a five-digit number as its identification while instruments and presets have “INST” and “PHDR” respectively.

This is fine for most purposes, but for example, when you want to merge generated JSON files somehow, it is helpful to produce unique identifications for the elements which are twelve-digit hexadecimal numbers. You can specify this by setting the `-idx` flag to `uuid`. Note that those identifications change for

### 3.4. *SOUNDFONTINPLACERENAMER*

---

each conversion, hence the same objects in a SoundFont do not have identical unique identifications across different conversions.

When the wave files are generated, their format is specified via the `-sf` flag. Default is `n` where the destination format depends on whether there is a `sm24` chunk in the SoundFont: when it is there, the sample format is 24-bit; otherwise it is 16-bit. But it is also possible to explicitly specify the sample format.

The names of the wave files are generated from the sample names. When identical names are used for different samples, the generation of wave files is skipped. Normally each sample has its own wave file, but linked samples are combined into the left and right channel of a single stereo file (when their sample rates are identical).

## 3.4 soundFontInPlaceRenamer

The in-place renaming of sample, instrument and preset names in a SoundFont by the program `soundFontInPlaceRenamer` is done via the following command line:

```
soundFontInPlaceRenamer [-h] [-l loggingFilePath]
                        -c configurationFilePath soundFontFilePath
```

The options have the following meaning:

**-h**

makes the program show all the command line options and exit

**-l loggingFilePath**

path for the logging file and activates logging

**-c configurationFilePath**

path to the configuration file containing the renaming information (with its syntax described in appendix B)

**soundFontFilePath**

path to the SoundFont file to be adapted

The configuration contains replacement definitions for sample names, instrument names and preset names and global replacements.

A replacement definition uses a regular expression (in Python notation) as a pattern and a replacement text. Those replacements are defined within four variables of the configuration file: `globalNameReplacementMap`, `sampleNameReplacementMap`, `instrumentNameReplacementMap`, and `presetNameReplacementMap`.

For example the following variable definition for instruments defines two replacements:

```
instrumentNameReplacementMap = "{"
    "'^Guitar Feedback'      : 'GTR: Feedback',"
    "'^(\w+) Sax'           : 'REE: Sax \1'"
    "'^(Tubular) (Bells)'    : 'CRP: \2 \1'"
"}
```

The first one replaces any instrument name prefix “Guitar\_Feedback” by “GTR:\_Feedback”.

The second one uses a so-called group to capture parts of the string and replaces any instrument name with suffix “\_Sax” by “REE:\_Sax\_” followed by the prefix captured in the group (signified by \1 for the **first group**). For example, “Alto\_Sax” would be replaced by “REE:\_Sax\_Alto”, because “Alto” would be captured as the first group. \w+ stands for a sequence of word characters, which are letters and digits, but no blanks.

The third one matches a string starting with “Tubular Bells”, captures the first word in group 1 and the second in group 2, prefixes them by “CRP:” and exchanges the words. Hence e.g. “Tubular Bells 1” would be replaced by “CRP: Bells Tubular 1”.

Note that the caret in all patterns is required; otherwise also occurrences of those strings **within an instrument name** would be replaced accordingly.

A more detailed description of Python patterns can be found in the Python documentation [PyRegExp].

The replacements defined in the four variables mentioned above are applied as follows to the sample, instrument and preset names and in the given order:

|                   |  |
|-------------------|--|
| sample names:     | globalNameReplacementMap, sampleNameReplacementMap     |
| instrument names: | globalNameReplacementMap, instrumentNameReplacementMap |
| preset names:     | globalNameReplacementMap, presetNameReplacementMap     |

Note that in general *the order of replacements is significant* because the algorithm checks all patterns and applies the replacement upon each match. There is only one pass through the pattern list per name string, hence the process terminates.

The distribution contains a demo replacement configuration file and a demo SoundFont file for experimentation.

## 3.5 Using the Tools with the Demo SoundFont

The following section gives some hints on how to use the tools by using the enclosed demo soundfont, which you can find in the demo directory.

Note that this soundfont is *not at all helpful for music production*: it is simply used for the demonstration of the tools.

### Doing some In-Place-Renaming in the Demo SoundFont

There is a simple configuration file for renaming a soundfont called `sFInPlaceRenamer_demo.cfg`. Because we want to use the demo soundfont for other commands, we copy it first to a soundfont for renaming e.g. called `test.sf2`.

Then the renaming command is called as:

```
soundFontInPlaceRenamer -c sFInPlaceRenamer_demo.cfg test.sf2
```

One can check whether the renamings have been done by opening `test.sf2` in a SoundFont editor, e.g. in Polyphone.

### Converting the Demo SoundFont to JSON

First you will have to prepare some directory for storing the wave files, e.g. the directory `/tmp/soundfontsamples`.

We use a 16-bit format for the sample wave files, take the sample names directly as file names and want to have the natural number indices as identifications for all objects.

```
soundFontToJSON -w /tmp/soundfontsamples -idx nat -sn n -sf i16  
-i demoSoundFont.sf2 >demoSoundFont.json
```

### Converting the JSON File to Another SoundFont

Now we reverse the process.

We use the wave files from the above wave file directory and want a 16 bit sample storage in the soundfont. For testing we also write the representation of the internal model to another JSON file.

This leads to the following command:

```
jsonToSoundFont -i demoSoundFont.json -df otherSoundFont.json  
-o otherSoundFont.sf2 -w /tmp/soundfontsamples -sf i16
```

When comparing `demoSoundFont.sf2` and `otherSoundFont.sf2` they should be identical.



Also the JSON text representations `demoSoundFont.json` and `otherSoundFont.json` should be identical.

### Analyzing the SoundFont for Problems

The demo soundfont can easily be analysed for problems via the soundfont analyser as follows (using a configuration file and with a human readable output):

```
soundFontAnalyser demoSoundFont.sf2 -c sFAnalyser_demo.cfg -p  
                  >demoSFAnalysis.txt
```

The analysis should reveal several problems. You will find out, that the modulator analysis for generator kind 'modEnvToPitch' in instrument 'Test Modulator V' is exactly the analysis as described in section 3.2.1.3.



## 4. Debugging

The script suite consists of several complex tools and typically something goes wrong. Oftentimes the script issues some error message, but how can you find out what really went wrong?

When logging had been activated via the `-l` option on the command-line, the first place to look is that logging file. It does a very fine-grained tracing of the relevant function calls; searching in that file should give you some indication about the error.

The logging module does an entry-exit-logging and although this file generation slows down processing extremely, it helps to understand problems in case of errors. Figure 6 shows how a logging file looks like.

Every non-trivial function is logged there at least twice with timestamps: “»” indicates the entry of that function (possibly with information on the argument values), “«” the exit of that function (possibly with the return value) and “–” indicates some intermediate information during the function processing. The logging data is hierarchical, hence you can see the function call structure in this file precisely.

But note that logging files might get very large, typically in the tens or even hundreds of megabytes range, depending on the size of the SoundFont involved. So any analysis using the logging files might be tedious. Logging files are normally meant as a developer tool and not as a tool for the end user.

---

```

>>ChunkTreeNode.__init__ (013813): kind = 'ifil'
<<ChunkTreeNode.__init__ (013813)
>>ChunkTreeNode.fillVersionNodeFromChunkData (013813): kind = 'ifil', data = _ByteListReader(byteList = ['\x02', '\x
00', '\x01', '\x00', 'i', 's', 'n', 'g', '\n', '\x00'], position = 32, endPosition = 36)
>>AccessDescriptor.__init__ (013813): kind = 'n', position = 32, count = 2, value = None
<<AccessDescriptor.__init__ (013813): AccessDescriptor(kind = 'n', position = 32, count = 2, value = None)
>>AccessDescriptor.readFromByteList (013813): kind = 'n', position = 32, count = 2
<<AccessDescriptor.readFromByteList (013813): AccessDescriptor(kind = 'n', position = 32, count = 2, value = 2)
>>AccessDescriptor.__init__ (013813): kind = 'n', position = 34, count = 2, value = None
<<AccessDescriptor.__init__ (013813): AccessDescriptor(kind = 'n', position = 34, count = 2, value = None)
>>AccessDescriptor.readFromByteList (013813): kind = 'n', position = 34, count = 2
<<AccessDescriptor.readFromByteList (013813): AccessDescriptor(kind = 'n', position = 34, count = 2, value = 1)
>>ChunkTreeNode.fillVersionNode (013813): majorVersion = AccessDescriptor(kind = 'n', position = 32, count = 2, valu
e = 2), minorVersion = AccessDescriptor(kind = 'n', position = 34, count = 2, value = 1)
>>ChunkTreeNode.updateMap (013813)
<<ChunkTreeNode.updateMap (013813)
<<ChunkTreeNode.fillVersionNode (013813): ChunkTreeNode(identification = TND000003, kind = 'ifil', map = {'majorVersi
on': AccessDescriptor(kind = 'n', position = 32, count = 2, value = 2), 'minorVersion': AccessDescriptor(kind = 'n',
position = 34, count = 2, value = 1)}, children = [])
>>ChunkTreeNode.fillVersionNodeFromChunkData (013813): ChunkTreeNode(identification = TND000003, kind = 'ifil', map =
{'majorVersion': AccessDescriptor(kind = 'n', position = 32, count = 2, value = 2), 'minorVersion': AccessDescripto
r(kind = 'n', position = 34, count = 2, value = 1)}, children = [])
>> _ByteListReader.isDone (013813)
<< _ByteListReader.isDone (013813): True
>> _ByteListReader.isDone (013813)
<< _ByteListReader.isDone (013813): False
>>AccessDescriptor.__init__ (013813): kind = 's', position = 36, count = 4, value = None
<<AccessDescriptor.__init__ (013813): AccessDescriptor(kind = 's', position = 36, count = 4, value = None)
>>AccessDescriptor.readFromByteList (013813): kind = 's', position = 36, count = 4
<<AccessDescriptor.readFromByteList (013813): AccessDescriptor(kind = 's', position = 36, count = 4, value = isng)

```

Figure 6: Example for Logging File Extract from SoundFontInPlaceRenamer

## 5. References

- [Polyphone] Triponney, Davy.  
*Polyphone SoundFont Editor*.  
2025, <https://www.polyphone.io/>
- [PyRegExp] Python Software Foundation.  
*Python: re - Regular expression operations*.  
2025, <https://docs.python.org/3/library/re.html>
- [SFTSpec06] E-mu Systems.  
*SoundFont® Technical Specification*.  
February 2006, <http://www.synthfont.com/sfspec24.pdf>
- [SoundFontTools] Tensi, Thomas.  
*SoundFontTools*  
<https://github.com/prof-spock/SoundFontTools>
- [UML17] Object Management Group.  
*Unified Modeling Language - v2.5.1*.  
2017, <http://www.omg.org/spec/UML/>



## A. Glossary

**chunk**

a section within a  $\rightarrow RIFF$  file describing some object or an object list, typically characterized by a four-letter code; chunks form a tree structure and hence can be used to describe complex object structures

**data point**

a single sampled value at a specific point in a time raster within a  $\rightarrow sample$

**generator**

a synthesizer parameter within a  $\rightarrow zone$  with either a fixed value or a value controlled by one or several  $\rightarrow modulators$

**global zone**

a  $\rightarrow zone$  whose generators and modulators affect all other zones within the containing object

**instrument**

a collection of  $\rightarrow zones$  which represents the sound of a single musical instrument or sound effect set

**modulator**

a function of two  $\rightarrow modulator$  sources each transformed by a simple unit-interval mapping, multiplied by a factor with an optional trailing absolute value transformation and finally changing some  $\rightarrow generator$  by the resulting value

**modulator source**

a typically varying input parameter for controlling the sound synthesis like e.g. the key number pressed, the key velocity, some MIDI controller value etc.; is used as an input of a  $\rightarrow modulator$

**non-global zone**

a  $\rightarrow zone$  referencing a  $\rightarrow sample$  or  $\rightarrow instrument$  typically activated for some key or velocity range (or both)

**preset**

an object within a  $\rightarrow SoundFont$  collecting several  $\rightarrow instruments$  together to be sounded for a particular MIDI bank and preset number; within a SoundFont it is technically represented by a PHDR  $\rightarrow chunk$

**RIFF**

a file format introduced by Microsoft and IBM and being an abbreviation for “Resource Interchange File Format”; RIFF files are binary files consisting of a hierarchy of self-identifying  $\rightarrow chunks$

---

**sample**

a wave form used for constructing  $\rightarrow instruments$  in a  $\rightarrow SoundFont$  consisting of several  $\rightarrow data\ points$  giving the amplitudes; within a SoundFont it is technically represented by a SHDR  $\rightarrow chunk$

**sample data point**

$\rightarrow data\ point$

**soundfont, SoundFont**

a registered trademark of Creative Technology, Ltd. and a specification of a file format for sample-based synthesis for MIDI containing definitions for  $\rightarrow samples$ ,  $\rightarrow instruments$  and  $\rightarrow presets$  as well as their interconnection; the file format is based on the  $\rightarrow RIFF$  specification and consists of  $\rightarrow chunkss$  describing the above objects

**zone**

a substructure of an  $\rightarrow instrument$  or a  $\rightarrow preset$  containing either global articulation data (a  $\rightarrow global\ zone$  or articulation data defined to play over certain key numbers and velocities (a  $\rightarrow non-global\ zone$ ); within a SoundFont a zone is technically represented by a PBAG or IBAG  $\rightarrow chunk$  referencing data in PMOD, PGEN, IMOD and IGEN chunks



## B. Configuration File Overview

For some of the tools processing is defined by a configuration file. The name of this file is given as a mandatory parameter for that program.

Although one can define all settings in a single configuration file, configuration files can include other files for common definitions.

### B.1 Configuration File Location

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

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

### B.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 “`--`” (double-dash) is ignored.
- 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 and signifies a variable. One may define such variables arbitrarily.
- Several physical lines are collected into a single logical value assignment line until either an empty line (with only whitespace) or a new assignment line is encountered.

## B.2. CONFIGURATION FILE SYNTAX

---

- A line may end with a continuation marker “\”. That marker is discarded and the line is combined into the previous logical assignment line (if any). That marker is only needed in specific circumstances, because the algorithms tries to combine lines intelligently.
- 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 *immediately* replaced by its associated value. If there is none, this is an error. The processing is strictly sequential: the use of an identifier *must occur after its definition*. It is okay to use an identifier in its own redefinition or to have more than one definitions of an identifier.
- 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. As a simple convention in this document we prefix auxiliary variables with an underscore (but any convention — even none — is fine).

Assume for an example the following — not very meaningful — definitions in two files “test.txt” and “config.txt”:

```
-- test.txt file to be included elsewhere
_suppressedGeneratorKinds = "vibLfoToPitch", "modEnvToPitch", "initialFilterFc"
rule_presetModulators_isActive = false
standardReplacement = "'(\w+) Bass' : 'BAS: \1',"
```

```
-- config.txt file including test file
INCLUDE "test.txt"
rule_presetModulators_isActive = true
instrumentNameReplacementMap = "{" standardReplacement "}"
rule_instrumentModulators_excludedGeneratorKindList =
[ _suppressedGeneratorKinds, "releaseVolEnv" ]
```

leads to the following overall variable settings (when all strings are expanded accordingly):

## APPENDIX B. CONFIGURATION FILE OVERVIEW

---

```
rule_instrumentModulators_excludedGeneratorKindList =  
    "[vibLfoToPitch, modEnvToPitch, initialFilterFc, releaseVolEnv]"  
rule_presetModulators_isActive = true  
standardReplacement = "'(\w+) Bass' : 'BAS: \1',"  
_suppressedGeneratorKinds = "vibLfoToPitch, modEnvToPitch, initialFilterFc"  
instrumentNameReplacementMap = "{ '(\w+) Bass' : 'BAS: \1', }"
```



## C. SoundFont UML Model

The internal model of a SoundFont is given in this chapter by a model in the Unified Modeling Language [UML17] which is directly reflected in the JSON representation.

### C.1 Elementary\_\_Types

This packages contains the elementary types for each UML model like e.g. strings or numbers.

#### Diagrams

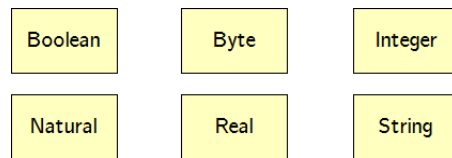


Figure 7: UML Model — Elementary Types - Overview

Elementary types ("primitive types") of the model are the following types:

- **Boolean**: for truth values that can be true or false
- **Byte**: for eight bit binary information,
- **Integer**: for positive, zero and negative whole numbers,
- **Natural**: for positive and zero whole numbers,
- **Real**: for floating point numbers (with internally defined precision), and
- **String**: for texts (of unspecified length).

## Classes

### **Boolean**

A *Boolean* is a truth value (either true or false).

### **Byte**

A *Byte* is an eight-bit number used for binary data.

### **Integer**

An *Integer* is a whole number (either positive, zero or negative).

### **Natural**

A *Natural* is a whole number (either positive or zero).

### **Real**

A *Real* is a floating point number (with internally defined precision).

### **String**

A *String* is a sequence of ASCII-characters of unspecified length.

## C.2 SoundFont\_\_Types

This package contains all soundfont specific types.

The described model is an *abstraction* of soundfonts: it does not put any constraints on the range of some variables (e.g. the sample data point size or the number of modulators per zone).

Technically those constraints are enforced by the file format used.

### Diagrams

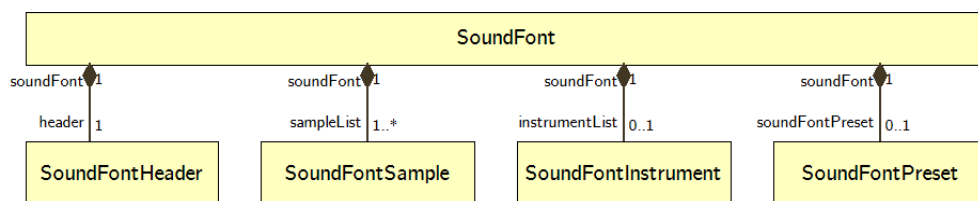


Figure 8: UML Model — SoundFont Types - Overview

The soundfont types give information about general aspects of the soundfont and the characteristic sound data like samples, instruments and presets. The top-level types are:

- **SoundFont**: the SoundFont itself with a SoundFontHeader, a list of SoundFontSamples, a list of SoundFontInstruments, and a list of SoundFontPresets,
- **SoundFontHeader**: a structure capturing all general data of the soundfont, like e.g. its name or version,
- **SoundFontSample**: a named sample referencing binary data with the sample data points,
- **SoundFontInstrument**: a named list of zones each referencing at most one SoundFontSample and defining synthesis parameters for them, and
- **SoundFontPreset**: a named list of zones each referencing at most one SoundFontInstrument and defining synthesis parameters for them.

Note that this model does not contain technical types like e.g. bags. Those types are needed for the serialization of a soundfont to a file, but they are replaced in the conceptual model by lists of partner objects.

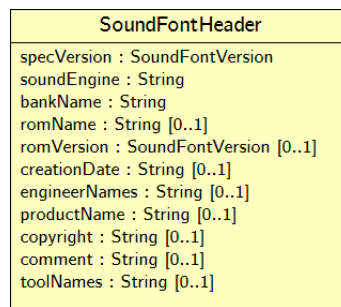


Figure 9: UML Model — SoundFont Types - Global SoundFont Data

A `SoundFontHeader` has several attributes describing the soundfont:

- `soundEngine` gives the underlying sound engine name,
- `bankName` gives the name of the soundfont,
- `romName` gives the name of the ROM for ROM-based samples (and is considered obsolete and hence is empty),
- `creationDate` is a date string for the creation date (or typically the last change date) of this soundfont,
- `engineerNames` is a name list string for the creators,
- `productName` describes a specific product for which the soundfont is intended (typically empty),
- `copyright` is a copyright notice for this soundfont,
- `comment` is some longer comment describing the soundfont further,
- `toolNames` is the name list string for the tools used for creating this soundfont,
- `specVersion` is the soundfont specification version level to which the soundfont complies (either 2.01 or 2.04),
- `romVersion` is the required ROM version for the samples in the soundfont (and is considered obsolete and hence is normally empty).

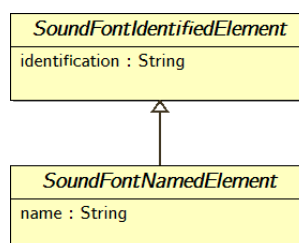


Figure 10: UML Model — SoundFont Types - Named and Identified Elements



There are two classes abstracting the fact of having an identification and a name:

- `SoundFontIdentifiedElement` is an object with a string identification.
- `SoundFontNamedElement` is an object with a name string (and an identification).

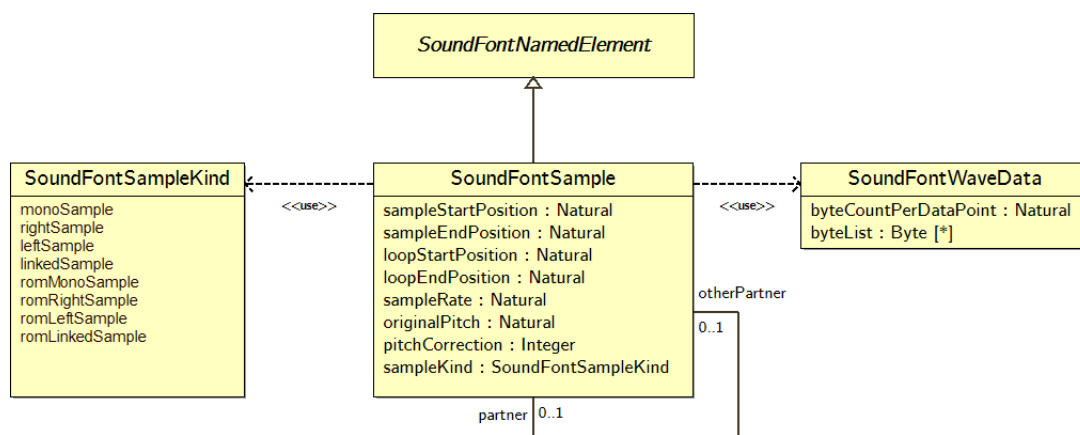


Figure 11: UML Model — SoundFont Types - Samples

The wave forms in a soundfont are represented by `SoundFontSample` objects. Those samples reference data points giving the signal amplitudes and those are stored in a `SoundFontWaveData` object.

Two attributes in the soundfont sample give the first and last data point position in the wave data list; it is also possible to specify a loop range in the sample.

Additionally information about the sample rate, the intended original pitch, a tuning correction and the kind of sample is contained. A sample may also have a partner: then each one is responsible for one channel of a stereo playback.

The single wave data object either has two or three byte data points. Those are stored in a byte list, where the data points of all samples are collected.

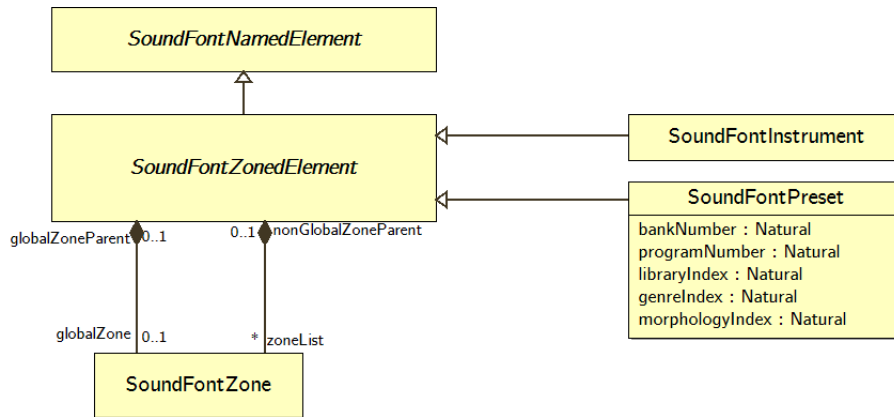


Figure 12: UML Model — SoundFont Types - Zoned Elements

A **SoundFontZonedElement** encapsulates common characteristics of both a **SoundFontInstrument** and **SoundFontPreset**. Both have an optional global zone and a list of non-global zones (referencing other soundfont elements).

Such a zoned element has a name and an identification (inherited from **SoundFontNamedElement**).

A preset object has additional parameters: its preset number (given by **bankNumber** and **programNumber**) and some (so far unused) classification information given by **libraryIndex**, **genreIndex** and **morphologyIndex**.

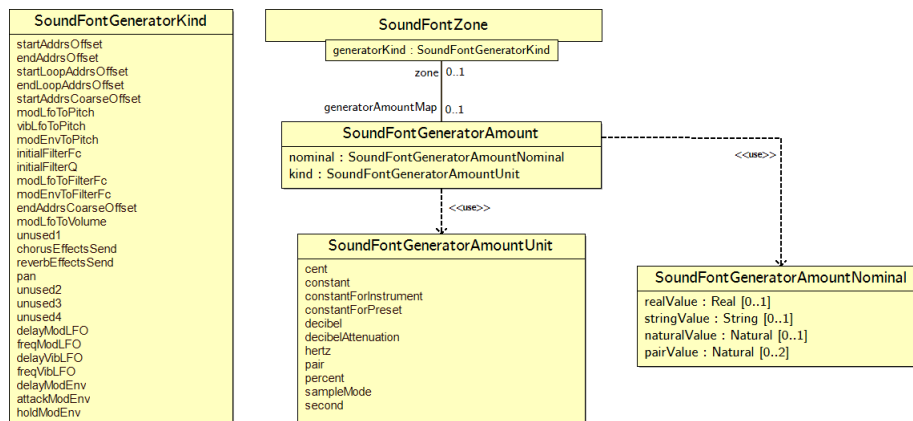


Figure 13: UML Model — SoundFont Types - Generators

The **SoundFontGeneratorKind** enumeration describes all possible synthesizer parameters within a sound font zone.

Each `SoundFontZone` has a generator map that maps generator kinds onto associated `SoundFontGeneratorAmount`. Each such amount has a nominal value (typically a real number and a unit given by a `SoundFontGeneratorAmountUnit`).

For example, the amount for `reverbEffectsSend` has a percent unit and this will be represented by a trailing "%" in the external representation, the amount for `delayModLFO` has a seconds unit with a trailing "s".

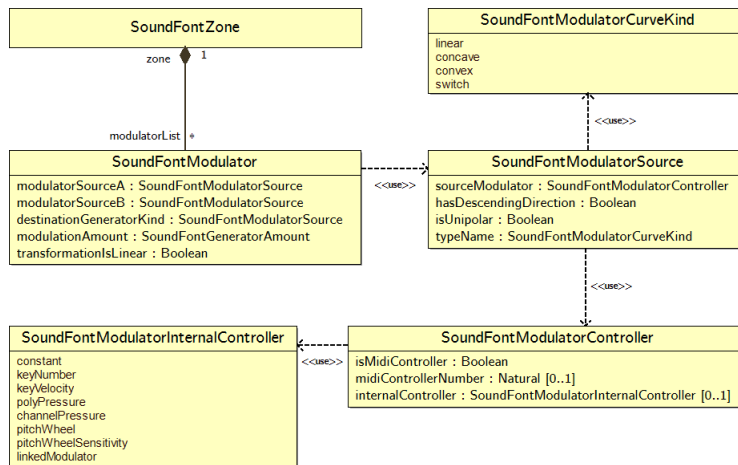


Figure 14: UML Model — SoundFont Types - Modulators

A `SoundFontModulator` object controls a generator in a sound font zone. It implements a function of two modulator sources both transformed to a range of 0 to 1 or -1 to 1 and then multiplied together. The resulting value is multiplied by a factor followed by an optional trailing absolute value transformation; this value finally changes some generator via the resulting value.

A `SoundFontModulatorSource` references some controller: this can be a MIDI controller or some internal controller like e.g. the key number. That value is transformed to a range of 0 to 1 or -1 to 1, potentially mirrored and deformed by a concave/convex or switch function.

### Classes

#### SoundFont

A *SoundFont* is the central object type and groups information about general aspects of the SoundFont ("header data" like e.g. the name and the author) and the characteristic sound data like samples, instruments and presets.

It references all its subelements via composition, i.e. it owns those elements.

#### Attributes from Associations or Queries

- header: SoundFontHeader  
the associated header of the soundfont
- sampleList: SoundFontSample [1..\*]  
the list of samples in this soundfont
- instrumentList: SoundFontInstrument [0..1]  
the list of instruments in this soundfont
- soundFontPreset: SoundFontPreset [0..1]  
the soundfont where this preset belongs to

#### SoundFontGeneratorAmount

A *SoundFontGeneratorAmount* object describes the amount of some zone generator.

The amount consists of a nominal value (which may be a real, a natural, a partner identification or a range pair) and a unit (hertz, cent, second, ...).

For example, the generator kind "delayModLFO" has a nominal value kind "real" and a unit of "second".

#### Attributes

- nominal: SoundFontGeneratorAmountNominal  
the nominal value of the generator amount
- kind: SoundFontGeneratorAmountUnit  
the kind of the generator amount

#### Attributes from Associations or Queries

- zone: SoundFontZone  
the sound font zone where this amount is used for some generator

**SoundFontGeneratorAmountNominal**

A *SoundFontGeneratorAmountNominal* is a union type combining a real value, an identification string, a natural and a pair of two natural values (for range specifications).

The nominal depends on the unit of the generator kind: e.g. a *keyRange* generator has a pair nominal value, a *chorusEffectsSend* generator has a real nominal value, an *instrument* generator has a string nominal value and a *sampleModes* generator has a natural nominal value.

Each of the possible nominal values is optional: exactly one must be defined in a valid instance.

**Attributes**

- `realValue: Real [0..1]`  
the (optional) real nominal value
- `stringValue: String [0..1]`  
the (optional) string nominal value
- `naturalValue: Natural [0..1]`  
the (optional) natural nominal value
- `pairValue: Natural [0..2]`  
the (optional) pair nominal value; only multiplicities 0 and 2 are allowed

**Rules**

- exactly one variant is active:  
exactly one of those conditions is true:  $\text{multiplicity}(\text{realValue}) = 1$ ,  $\text{multiplicity}(\text{stringValue}) = 1$ ,  $\text{multiplicity}(\text{naturalValue}) = 1$ ,  $\text{multiplicity}(\text{pairValue}) = 2$

**SoundFontGeneratorAmountUnit**

The *SoundFontGeneratorAmountUnit* enumeration type gives the possible unit for generator amounts.

For example, the amount for *reverbEffectsSend* has a percent unit, the amount for *delayModLFO* has a seconds unit.

- `cent`
- `constant`
- `constantForInstrument`
- `constantForPreset`
- `decibel`

- `decibelAttenuation`
- `hertz`
- `pair`
- `percent`
- `sampleMode`
- `second`

### **SoundFontGeneratorKind**

The *SoundFontGeneratorKind* enumeration type covers all kinds of generators in a soundfont.

They are ordered here in specification order.

- `startAddrsOffset`
- `endAddrsOffset`
- `startLoopAddrsOffset`
- `endLoopAddrsOffset`
- `startAddrsCoarseOffset`
- `modLfoToPitch`
- `vibLfoToPitch`
- `modEnvToPitch`
- `initialFilterFc`
- `initialFilterQ`
- `modLfoToFilterFc`
- `modEnvToFilterFc`
- `endAddrsCoarseOffset`
- `modLfoToVolume`
- `unused1`
- `chorusEffectsSend`
- `reverbEffectsSend`
- `pan`
- `unused2`
- `unused3`
- `unused4`
- `delayModLFO`
- `freqModLFO`
- `delayVibLFO`
- `freqVibLFO`
- `delayModEnv`
- `attackModEnv`
- `holdModEnv`
- `decayModEnv`
- `sustainModEnv`

- releaseModEnv
- keynumToModEnvHold
- keynumToModEnvDecay
- delayVolEnv
- attackVolEnv
- holdVolEnv
- decayVolEnv
- sustainVolEnv
- releaseVolEnv
- keynumToVolEnvHold
- keynumToVolEnvDecay
- instrument
- reserved1
- keyRange
- velRange
- startLoopAddrsCoarseOffset
- keynum
- velocity
- initialAttenuation
- reserved2
- endLoopAddrsCoarseOffset
- coarseTune
- fineTune
- sampleID
- sampleModes
- reserved3
- scaleTuning
- exclusiveClass
- overridingRootKey
- unused5
- endOper

### **SoundFontHeader**

A *SoundFontHeader* contains the general information about the soundfont, like e.g. the name of the soundfont, the author or the SoundFont standard version used for the file containing the data.

#### **Attributes**

- specVersion: SoundFontVersion  
the version supported by this soundfont (e.g. "2.01")
- soundEngine: String  
the underlying sound engine name (e.g. "E-mu 10K2")

- bankName: String  
the name of this soundfont
- romName: String [0..1]  
the name of the required ROM for the samples in the soundfont (if applicable); normally obsolete and hence empty
- romVersion: SoundFontVersion  
the required ROM version for the samples in the soundfont (if applicable); normally obsolete and hence empty
- creationDate: String [0..1]  
the date string for the creation date (or typically the last change date) of this soundfont
- engineerNames: String [0..1]  
the name list string for the creators
- productName: String [0..1]  
the name of a specific product for which the soundfont is intended (typically empty)
- copyright: String [0..1]  
a string specifying the copyright for the soundfont
- comment: String [0..1]  
a longer comment describing the soundfont further
- toolNames: String [0..1]  
the name list string for the tools used for creating this soundfont

### Attributes from Associations or Queries

- soundFont: SoundFont  
the soundfont where this header belongs to

### ***SoundFontIdentifiedElement***

A *SoundFontIdentifiedElement* is an element with an externally visible identification.

### Attributes

- identification: String  
a identification for that object; can be a UUID or some readable identification



**SoundFontInstrument**

(inherits from SoundFontZonedElement)

A *SoundFontInstrument* object is a zoned element referencing samples in the non-global zones.

**Attributes from Associations or Queries**

- soundFont: SoundFont  
the soundfont where this instrument belongs to

**SoundFontModulator**

A *SoundFontModulator* object represents a function of two modulator sources to a generator value.

Each modulator source is transformed by a simple unit-interval mapping (possibly non-linear), multiplied by a factor with an optional trailing absolute value transformation giving a result value for finally changing some generator by that value.

**Attributes**

- modulatorSourceA: SoundFontModulatorSource  
the first modulator source
- modulatorSourceB: SoundFontModulatorSource  
the second modulator source
- destinationGeneratorKind: SoundFontModulatorSource  
the destination generator kind for this modulator
- modulationAmount: SoundFontGeneratorAmount  
the maximum amount this modulator will change destination generator kind; the minimum amount is either -modulationAmount or 0 depending on the modulator sources
- transformationIsLinear: Boolean  
tells whether the resulting modulation amount is either passed through or has its absolute value calculated before

**Attributes from Associations or Queries**

- zone: SoundFontZone  
the sound font zone where this modulator is used for some generator

### SoundFontModulatorController

A *SoundFontModulatorController* object describes one controller in a modulator. It can be either a MIDI controller or an internal controller like the key number, the key velocity and a constant.

#### Attributes

- `isMidiController`: Boolean  
tells whether this controller is a MIDI controller (instead of an internal controller)
- `midiControllerNumber`: Natural [0..1]  
the MIDI controller number (when `isMidiController` is true)
- `internalController`: *SoundFontModulatorInternalController*  
the internal controller (when `isMidiController` is false)

### SoundFontModulatorCurveKind

The *SoundFontModulatorCurveKind* describes the different curve kinds for a modulator source transformation: linear, concave, convex and switch.

- linear
- concave
- convex
- switch

### SoundFontModulatorInternalController

The *SoundFontModulatorInternalController* object describes a non-MIDI controller in a modulator. It can be constant, the key number, the key velocity, the poly or channel pressure or the pitch wheel.

- constant
- keyNumber
- keyVelocity
- polyPressure
- channelPressure
- pitchWheel
- pitchWheelSensitivity
- linkedModulator

**SoundFontModulatorSource**

A *SoundFontModulatorSource* object represents a controller for a modulator together with a transformation function described by a curve kind, an information about polarity and on the slope of that function.

**Attributes**

- `sourceModulator: SoundFontModulatorController`  
the controller used in this modulator source
- `hasDescendingDirection: Boolean`  
tells that the transformation function is descending
- `isUnipolar: Boolean`  
tells that the transformation function domain goes from 0 to 1 (instead of -1 to 1)
- `typeName: SoundFontModulatorCurveKind`  
tells the type of transformation function (linear, concave, convex or switch)

**SoundFontNamedElement**

(inherits from *SoundFontIdentifiedElement*)

A *SoundFontNamedElement* is an element with a name (and an identification inherited from *SoundFontIdentifiedElement*).

**Attributes**

- `name: String`  
the name of that object; does not have to be unique, because identification can be used for distinction

**SoundFontPreset**

(inherits from *SoundFontZonedElement*)

A *SoundFontPreset* object is a zoned element referencing instruments in the non-global zones.

Those instruments shall be sounded together for a particular MIDI bank and preset number.

### Attributes

- `bankNumber`: `Natural`  
the bank number of that preset; practically the maximum value specified by the MIDI standard is 128 (for a drum bank), while 0 to 126 are used for normal banks and 127 is also reserved for drum banks
- `programNumber`: `Natural`  
the program number of that preset
- `libraryIndex`: `Natural`  
the library index of that preset; informally described by the SoundFont specification, but never used in practice
- `genreIndex`: `Natural`  
the genre index of that preset; informally described by the SoundFont specification, but never used in practice
- `morphologyIndex`: `Natural`  
the morphology index of that preset; informally described by the SoundFont specification, but never used in practice

### Attributes from Associations or Queries

- `soundFont`: `SoundFont`  
the soundfont where this preset belongs to

### **SoundFontSample**

(**inherits from** `SoundFontNamedElement`)

A *SoundFontSample* represents a wave form used for constructing instruments in a soundfont.

It consists of several data points giving the signal amplitudes and stored in a `SoundFontWaveData` object.

### Attributes

- `sampleStartPosition`: `Natural`  
the index of the first sample data point into the data of the wave data object
- `sampleEndPosition`: `Natural`  
the index of the first sample data point into the data of the wave data object

- `loopStartPosition`: `Natural`  
the index of the first sample loop data point into the data of the wave data object
- `loopEndPosition`: `Natural`  
the index of the last sample loop data point into the data of the wave data object
- `sampleRate`: `Natural`  
the frequency to be used for playback of the data points
- `originalPitch`: `Natural`  
the MIDI note number describing the recorded pitch of the sample (hence bounded by 127)
- `pitchCorrection`: `Integer`  
the pitch correction in cents to bring the sample to the intended pitch
- `sampleKind`: `SoundFontSampleKind`  
the sample kind of that sample

#### Attributes from Associations or Queries

- `soundFont`: `SoundFont`  
the soundfont where this sample belongs to
- `partner`: `SoundFontSample`  
the partner sample for a single channel sample representing the other channel

#### Rules

- correct sample indices:  
 $\text{sampleStartPosition} \leq \text{sampleEndPosition}$
- correct loop indices:  
 $\text{loopStartPosition} \leq \text{loopEndPosition}$   
 $\wedge \text{sampleStartPosition} \leq \text{loopStartPosition}$   
 $\wedge \text{loopEndPosition} \leq \text{sampleEndPosition}$

#### **SoundFontSampleKind**

The *SoundFontSampleKind* tells the kind of sample: either this sample is given externally or resides in ROM (where the latter is considered obsolete). For both variants a sample may be a mono sample, the left or right channel of a stereo sample (with reference to a partner) or a "linked sample" not currently defined by the specification but intended for circular playback.

- `monoSample`
- `rightSample`
- `leftSample`
- `linkedSample`
- `romMonoSample`
- `romRightSample`
- `romLeftSample`
- `romLinkedSample`

### SoundFontVersion

The *SoundFontVersion* object is a pair of two natural numbers specifying the major and minor version of some object.

#### Attributes

- `majorVersion`: Natural  
the major version part
- `minorVersion`: Natural  
the minor version part (where a two digit representation is assumed with e.g. 4 meaning '04')

### SoundFontWaveData

The singleton *SoundFontWaveData* stores the sample data points referenced by all sample objects.

Sample data points may either consist of two or three bytes per data point and those bytes are all stored in a byte list.

#### Attributes

- `byteCountPerDataPoint`: Natural  
the number of bytes per data point
- `byteList`: Byte [0..\*]  
the bytes for all data points

### SoundFontZone

A *SoundFontZone* is part of an *SoundFontZonedElement* containing either global articulation data (a *global zone*) or articulation data defined to play over certain key numbers and velocities (a *non-global zone*).

#### Attributes from Associations or Queries

- globalZoneParent: SoundFontZonedElement [0..1]  
the zoned element containing this zone as a global zone
- nonGlobalZoneParent: SoundFontZonedElement [0..1]  
the zoned element containing this zone as a non-global zone
- generatorAmountMap:  
    SoundFontGeneratorKind→SoundFontGeneratorAmount  
a partial map from generator kind to an associated amount; those are the fixed values for the generator in the current zone (but they may be modified by some modulator)
- modulatorList: SoundFontModulator [0..\*]  
the list of modulators for that zone

#### **SoundFontZonedElement**

(inherits from SoundFontNamedElement)

A *SoundFontZonedElement* is an object containing a global zone and a list of non-global zones referencing some partner object.

It is the supertype of both *SoundFontInstrument* and *SoundFontPreset*.

#### Attributes from Associations or Queries

- zoneList: SoundFontZone [0..\*]  
the list of non-global zones for this element
- globalZone: SoundFontZone [0..1]  
the associated global zone (if any)





## D. Release Changes

- Version 0.1 (2025-11): initial version