The **Songlib** Tab Processor

written by: Brandon Bonds

Revision Date: January 8, 2013

The **songlib** Tab Processor, *processtab*, parses a *tab* file into **songlib**-based source code. You can run *processtab* as follows:

```
processtab [filename].tab
```

This produces a C file, [*filename*].*c*, that builds against **songlib**, and a *makefile*. (View more options by using the `-h` switch.) Running:

```
make
```

will compile [*filename*].*c* into a binary (simply named [filename] with no extension). The makefile will then run the binary to produce [filename].rra. By default, make will immediately play the RRA file.

Tab Format Overview

A *tab* file is text. To start, let's define a very simple tablature:

```
|SN|o-o-|oooo|
|  |1234|1234|
```

This is a single *stave* of tablature containing two measures of snare drum beats. The first measure has notes at beats 1 and 3 and rests at beats 2 and 4. The second measure has notes at all four beats.

However, this file is not quite complete. *processtab* does not natively understand what "SN" represents. We must define it using an *attribute*. The following is a complete tab file which contains a definition for "SN" using the `voice` attribute:

```
# Voices

voice:
{
    "name": "SN",
    "instrumentPath": "/usr/local/share/samples/drums/",
    "instrumentBaseName": "hera_",
    "notes":
    [
        {
            "name": "o",
            "offset": 57
        }
```

```
    ]
}

# Tab

|SN|o-o-|oooo|
|  |1234|1234|
```

For now, it is sufficient to understand that a voice is defined by a name ("voice"), followed by a colon (:), followed by JSON-formatted details inside curly braces.

In addition to staves and attributes, tab files may also contain comments which are not parsed. A comment begins with the hash character (#) and lasts until the end of the line.

Tab Defaults

By default, the following values are used:

stride	0.05
sustain	0.99995
primary emphasis	1 (no emphasis)
secondary emphasis	1 (no emphasis)
tempo	80
time signature	4/4
use random sampling	true

More on Staves

A tab contains one or more *staves*. Staves define the notes to be played. The example above contains a single stave defining notes for voice "SN".

Each stave ends with a single line called the *timing course*. This line defines the beat. In the example, both measures have a basic four-beat timing course. More complex timing courses are possible. The example could have been written as either of the following:

```
|SN|o---o---|o-o-o-o-|
|  |1&2&3&4&|1&2&3&4&|

|SN|o-------o-------|o---o---o---o---|
|  |1e&a2e&a3e&a4e&a|1e&a2e&a3e&a4e&a|
```

The first stave subdivides the timing into eighth notes, and the second stave subdivides the timing into sixteenth notes. Numbers indicate down beats, ampersands (`&`) represent upbeats, and "e" and "a" represent sixteenth divisions. This makes more intricate rhythms possible, such as the following:

```
|SN|o--oo---|o-o-o-o-oo-oo-o-|
|  |1&2&3&4&|1e&a2e&a3e&a4e&a|
```

You may mix timings, even in the same measure:

```
|SN|o--oo-oooo|
|  |1&2&3&4e&a|
```

You may insert dots (.) in the timing course to subdivide into triplets or practically any other possible timing (such as sextuplets, quintuplets, etc.):

```
|SN|o--o--oooo--|o--o--ooooooo--|
|  |1..2..3..4..|1..2..3.....4..|
```

Each beat is subdivided into triplets, except beat three of the second measure, which is subdivided into sextuplets.

Multiple voice lines (voice courses) may exist for each stave. Everything should line up vertically. Spaces are optional around the name, and may be useful for lining up courses whose names are different lengths. Notes that are played at the same time must line up vertically. All pipe characters (—) must line up vertically.

For instance, the following stave must include an extra space for the "B" voice to make it line up with the "SN" voice:

```
|SN|--o---o-|o-o-o-o-oo-oo-o-|
|B |o--oo---|o---o---o---o---|
|  |1&2&3&4&|1e&a2e&a3e&a4e&a|
```

Notes and Rests

Inside a measure, a dash character (-) indicates a rest and a space indicates that the previous note is held out during that duration. The following two measures have the same rhythm, but differ in how long each note is held out:

```
|SN|o o o o |o-o-o-o-|
|  |1&2&3&4&|1&2&3&4&|
```

In the first measure, each note holds out a quarter-note length. In the second measure, each note holds out an eighth-note length, followed by an eighth-note rest.

"o" has been used to indicate playing a note in each of these examples. Any character other than space or dash may be used to represent a note, and is defined by the voice attribute. In the next example, we will use "x" to represent a closed hi-hat, "o" to represent and open hi-hat, and "@" to represent a rim click:

```
|HH|XxxxXxxo|XxxxXxxo|
|SN|--@---@-|--@---@-|
|  |1&2&3&4&|1&2&3&4&|
```

Capitalized characters typically represent accents.

More on Attributes

An attribute is defined on one or more lines. It lists the name of the attribute (such as `voice`), followed by a colon character (:), followed by details in JSON (JavaScript Object Notation) format. JSON is a common hierarchical text format like XML, but with less bloat.

There are three built-in attribute types: `voice`, `tempo`, and `amplitude`. *processtab* also supports user-defined attribute processors.

Voice

The voice processor defines each voice, or instrument, that can play notes in the tab. It configures the voice according to the following attributes:

name (string): This is the unique name that also appears in the first column of a voice course, before measure and note definitions. In the above examples, "SN" was defined for a snare drum.

instrumentPath (string): This is the directory (folder) on disk which contains the samples. See the first argument for *readScale* in **songlib**.

instrumentBaseName (string): This is the base name for each sample. See the second argument for *readScale* in **songlib**.

amplitudeMultiplier (float): This sets the base amplitude for all notes in this instrument. See *setAmplitude* in **songlib**. Note that the final amplitude for a note may be adjusted by other parameters, and may be adjusted by other processors such as *amplitude*.

instrumentCount (integer): When set higher than one, **songlib**will simulate multiple instrumentalists by playing the note on top of itself several times. This is affected by the values of *amplitudeDeviation*, *pitchDeviation*, and *timingDeviation*. The amplitude of each individual instrument in this set is divided by instrumentCount.

amplitudeDeviation (float): Sets the randomization of the amplitude. Higher values help simulate instrument players with less skill. The entire song will be amplitude-adjusted, up or down, at a randomized value between 1 and *amplitudeDeviation*. When *instrumentCount* is greater than one, each individual instrument has a separate randomized amplitude throughout the song.

pitchDeviation (float): Sets the randomization of the pitch. Higher values can simulate the natural difference in the pitch of percussion instruments, or help simulate tuned instrument players with less skill. The entire song will be pitch-adjusted, up or down, at a randomized value between 1 and *pitchDeviation*. When *instrumentCount* is greater than one, each individual instrument has a separate randomized pitch throughout the song.

timingDeviation (float): Sets the randomization of timing. Higher values can simulate mud or slop. Each individual note will be offset by up to timingDeviation beats from its intended location.

notes (array): Contains several individual note definitions, each with the following attributes:

- **name** (character): A single character used to define notes in the *tab*. In the examples above, the "o" note is defined.
- **offset** (integer): A value indicating the offset used to play the note. See the third parameter for *nplay* in **songlib**.
- **accent** (boolean): If "true", this note will always be played at double the amplitude of a normal note. (Typically, the *name* is capitalized for accented notes.)
- **delay** (integer): The number of samples to delay the note. Typically this value is negative, causing the note to begin early. This is useful for aligning the attack of a note.

The following is a full example of a voice attribute with two notes defined:

```
voice:
{
"name": "SN",
"instrumentPath": "/usr/local/share/samples/drums-snare/",
"instrumentBaseName": "pearlFF_",
"amplitudeMultiplier": 16.0,
```

```
"instrumentCount": 4,
"amplitudeDeviation": 1.5,
"pitchDeviation": 4.0,
"timingDeviation": 0.125,
"notes":
[
    {"name": "o", "offset": 57, "delay": -1234},
    {"name": "O", "offset": 57, "delay": -1234, "accent": true}
]
}
```

This voice defines a "pearlFF" snare drum from the "drums-snare" sample pack. The overall amplitude is adjusted louder by a factor of 16. Each note is played 4 times simultaneously (due to *instrumentCount*). Each individually played note has a randomly adjusted amplitude, pitch, and timings to simulate 4 real players.

Two notes are defined: lowercase "o" plays sample 57 (note A4) with a delay of -1234 samples (which makes it start playing a bit early), and uppercase "O" plays the same note with an accent.

Tempo

Tempo is a simple processor that defines the tempo beginning at that location in the tab. Tempo may be redefined multiple times throughout a song as required.

```
# Tempo example
tempo: { "": 164 }
```

This will set the music tempo to 164 beats per minute. See *setTempo* in **songlib**.

Amplitude

Amplitude is a simple processor that defines the base amplitude for all voices and notes beginning at that location in the tab. This value multiplies against the explicit values defined by voice attributes. Amplitude may be redefined multiple times throughout a song as required.

```
# Amplitude example
amplitude: { "": 0.015 }
```

This will set the base amplitude for all instruments to 0.015. The "voice" attribute will then multiply this value by any other amplitude variations it sets. See *setAmplitude* in **songlib**.

Custom processors

Custom processors are an advanced way to alter the output of *processtab*. Custom processors are written in C and are used like built-in processors.

Writing custom processors

We will create an example custom processor that calls the **songlib**function *setSkipSeconds*. We will name this processor `skipseconds`.

Each custom processor must define a *.c* file and a *.h* file. These files must be named the same as the processor name. These files must define a minimum of the following five *hooks* (replace *XYZ* with your processor name):

- `void setupProcessor_`*XYZ*`(int, int, json_value *)`

- `void preProcess_`*XYZ*`(int, Part *)`
- `void postProcess_`*XYZ*`(int, Part *)`
- `void preRenderNote_`*XYZ*`(int, Part *, int)`
- `void renderNote_`*XYZ*`(int, Part *, int)`

Hooks are functions called by other code. In our case, these hooks are called by the output of *processtab*.

This is our *skipseconds.h* file:

```
#ifndef INCLUDED_SKIPSECONDS_H
#define INCLUDED_SKIPSECONDS_H

/*
 * Defines a skip seconds processor
 */

#include <tab/tabproc.h>

extern void setupProcessor_skipseconds(int, int, json_value *);

/* Pre- and post- processing */

extern void preProcess_skipseconds(int, Part *);
extern void postProcess_skipseconds(int, Part *);

/* Note rendering */

extern void preRenderNote_skipseconds(int, Part *, int);
extern void renderNote_skipseconds(int, Part *, int);

#endif /* INCLUDED_SKIPSECONDS_H */
```

All of these functions must be implemented in the *.c* file, but any hook that is not needed may contain an empty implementation.

In the case of `skipseconds`, we will only implement `setupProcessor`. This is our *skipseconds.c* file, including the functions that have an empty implementation:

```
/*
 * Defines a skip seconds processor
 */

#include "skipseconds.h"

/* Processor setup */

void
setupProcessor_skipseconds(int parameterIndex, int sectionNumber, json_value *parameter)
{
if (getSkipSeconds() != 0.0)
{
Fatal("skipseconds may only be set once\n");
```

```
return;
}
double seconds = get_json_double(parameter, "", 0);
setSkipSeconds(seconds);
}

/* Pre- and post- processing */

void
preProcess_skipseconds(int parameterIndex, Part *part)
{
}

void
postProcess_skipseconds(int parameterIndex, Part *part)
{
}

/* Note rendering */

void
preRenderNote_skipseconds(int parameterIndex, Part *part, int noteNumber)
{
}

void
renderNote_skipseconds(int parameterIndex, Part *part, int noteNumber)
{
}
```

`setupProcessor` will extract the number of seconds, as a `double`, from the JSON-formatted attribute (using the *get_json_double* function). It will then pass that value into the **songlib**function *setSkipSeconds*. And we will add a check using **songlib**function *getSkipSeconds* to ensure that the value is only set once.

Using custom processors

Custom processors are used just like any built-in processor, by adding an attribute at the appropriate location in your tab file. The *skipseconds* processor may be invoked as follows:

```
skipseconds: { "" : 120 }
```

processtab automatically recognizes when a tab uses custom processors (any attribute name other than `voice`, `tempo`, and `amplitude`), and outputs a makefile that references the *.c* and *.h* you create for your custom processor.

More custom processing details

This section explains, in detail, the five hooks that are used in custom attribute processing.

(It may help to visualize this process as it is being described. To do so, run *processtab* on a simple tablature, and read the *main()* function in the output *.c* file. You can see each of these hooks being called as described below.)

setupProcessor `setupProcessor` is called first for each attribute. This is called before *songInit*, before defaults are set, and before the *.rra* file is open for output. *setupProcessor* is the only location that has

access to the processed JSON information.

Because of this, we may need to store any data for later use by the other four functions. The first integer passed into `setupProcessor` is *parameterIndex*, which should be associated with that data. That same integer is passed into the other hooks to reference that particular data.

The second parameter is the section number. This allows attributes to affect different areas of the song. In this context, a *section* is equivalent to the stave number above which the attribute is defined. All attributes defined before the first stave are in section 0, and after the first stave but before the second are in section 1, and so on.

As you can see from the example *skipseconds* processor, it is possible to ignore these two values for very simple processors.

The third parameter is the data contained within the attribute. It is pre-parsed using the *json-parser* library (see https://github.com/udp/json-parser, and for more examples see the source code to the built-in attribute processors). The data is passed in as a *json_value* data structure.

Other functions The rest of the functions take an integer *parameterIndex*, which is used to reference data stored by *setupProcessor*. They also take in a *Part*, which roughly represents a voice.

But remember, `voice` is just another processor written the same way that a custom processor is written. It is a complex processor that ultimately calls the **songlib***rplay* function. You can write a custom processor and use that instead of `voice`, or you can copy and make changes to the `voice` processor to suit your needs.

So, a *Part* represents the notes that are to be played by a certain instrument. A part is defined by its name such as "SN" or "B".

Each part is processed at once. `preProcess` is called before processing the notes, and `postProcess` is called after processing all the notes. Between, each individual note is processed in order by first calling `preRenderNote` for every attribute, and then `renderNote` for every attribute.

`preRenderNote` and `renderNote` have a third paramter, *noteNumber*, which can be used to index into the *Part*.

Parser Limitations

The *processtab* program currently requires 4/4 time signature for all tabs. It requires that no spaces come before or after stave courses, or else a parsing error will occur. Also, you may come across some parsing bugs related to whitespace between staves or between voices, staves, and comments.

All quarter note timings must be indicated. This is partially due to the need to make the tab language unambiguous. Otherwise, the following invalid tab could be interpreted multiple ways:

```
# Invalid tab

|SN|oooooo|
|  |1&3&4&|
```

Here, is the first `&` the upbeat of beat one or the upbeat of beat two?

If "e" and/or "a" are indicated, `&` must also be indicated. However, "e" may be indicated without "a", and "a" without "e", of the same beat.