pylsdj Documentation

Release 2.3.3

Alex Rasmussen

February 16, 2015

Contents

Contents:

CHAPTER 1

Introduction

1.1 What is LSDJ?

Little Sound DJ (or LSDJ) is a program for the Nintendo Game Boy that turns the humble Game Boy into a music workstation. More information about LSDJ can be found at the LSDJ website and the LSDJ wiki.

1.2 What is pylsdj?

pylsdj is a suite of tools for reading, writing and editing LSDJ's save data, which includes the user's saved songs and instruments.

1.3 Why?

Before pylsdj, the suite of tools available for interacting with LSDJ's save data was sparse and fragmented. People who wanted to share and re-use instruments between songs or move songs between saves were met with partial solutions at best. pylsdj endeavors to be a one-stop solution for save data reading, writing and editing.

1.4 How Can I Help?

First and foremost, use it! You can also try out LSMC, which is really just a GUI on top of many of pylsdj's functions.

Second, if you find a bug, file it. I know I haven't hit all the potential use cases for this in tests, and your input will help me find and squash bugs.

Third, if you're a developer, write some tests. If you find a feature pylsdj doesn't have and you want to take a crack at it, fork the code and send me a pull request. I'm ready and willing to receive contributions from the community.

1.5 Known Limitations

pylsdj only works on save data for LSDJ versions 3.0.0 and above. Given that version 3.0.0 came out back in 2006 and marked a significant change in file structure, I feel like this is a reasonable point at which to freeze backwards-compatibility.

There are parts of the codebase that are much more messy than I'd like them to be, but perfect is the enemy of done.

CHAPTER 2

Terminology Primer

LSDJ's save data is stored in the Game Boy's battery RAM. LSDJ's file manager can hold up to 32 songs. Songs are stored in the file manager in a compressed form and a song is expanded into memory when it's being worked on.

The Game Boy has four audio channels: two pulse wave generators, a PCM 4-bit wave sample, and a noise generator. A song consists of a sequence of **chains**, one for each channel. Each chain consists of a sequence of **phrases**, and a phrase contains up to 16 notes.

The sound of each note is determined by the **instrument** used to play the note. This instrument controls the sound produced by one of the channels when the note is played. While the sound produced by each channel is simple, LSDJ provides a variety of means to vary the sound over time, allowing for a wide range of timbres as well as effects like arpeggio and vibrato.

CHAPTER 3

Compression and Decompression

Game Boys don't have a lot of RAM (128KB tops); in order for LSDJ to deal with such a limited amount of space, it has to pack files in pretty tight. To do this, it uses an algorithm referred to on the LSDJ wiki as the "file pack algorithm".

pylsdj includes functions that will compress and decompress lists of bytes using the file pack algorithm.

Typically you don't need to do this: .sav files compress themselves on save and decompress themselves on load automatically. If your application needs to do something fancy with LSDJ's filesystem, however, you can use the compression and decompression functions by themselves.

3.1 Usage Examples

```
from pylsdj import filepack

# Here's a list of bytes
bytes = [0x12, 0x12, 0x12, 0x15, 0x15, 0x10]

# We can compress those bytes
compressed = filepack.compress(bytes)

# ... and then decompress them again
decompressed = filepack.decompress(compressed)
```

3.2 API Documentation

pylsdj.filepack.**compress**(*raw_data*)
Compress raw bytes with the filepack algorithm.

Parameters **raw_data** – an array of raw data bytes to compress

Return type a list of compressed bytes

pylsdj.filepack.**decompress**(*compressed_data*)
Decompress data that has been compressed by the filepack algorithm.

Parameters **compressed_data** – an array of compressed data bytes to decompress

Return type an array of decompressed bytes

CHAPTER 4

.sav Files

pylsdj manipulates .sav files through a `pylsdj.SAVFile` object. This object can be used to load, store, and edit the contents of LSDJ's SRAM file.

4.1 Loading and Saving

If you're writing an application using pylsdj, you'll probably want to load and store it.

4.1.1 Callback Functions

Several methods of `pylsdj.SAVFile` take a progress callback function that callers can use to notify callers of how far the operation has progressed. Callback functions take four arguments

- `message`: a message explaining what the step is doing
- `step`: the step that the operation is currently on
- `total_steps`: the total number of steps in the operation
- `continuing`: True if the operation is going to continue

4.2 Accessing and Editing a .sav File's Projects

A .sav file's `project_list` field contains an ordered list of that file's projects. You can insert, modify and delete `pylsdj.Project` objects in this list to modify the .sav file's contents. Note that changes to the .sav file will not persist unless it is saved.

4.3 Usage Examples

```
from pylsdj import SAVFile

# Load .sav file from lsdj.sav
sav = SAVFile('lsdj.sav')

# Load a .sav file, passing loading progress to a callback
def my_callback(message, step, total_steps, continuing):
  print '%(m)s: %(s)d/%(t)d complete!' % { m: message, s: step, t: total_steps }
```

```
sav = SAVFile('lsdj.sav', my_callback)

# Get the file's project map (maps slot number to Project)
projects = sav.projects

# Save a savfile as lsdj_modified.sav, passing the same progress callback
# from the above example
sav.save('lsdj_modified.sav', my_callback)
```

4.4 API Documentation

class pylsdj.**SAVFile**(*filename*, *callback=<function _noop_callback at 0x7ffea471d668>*)

project_list
The list of pylsdj.Project s that the .sav file contains

save(*filename*, *callback=<function _noop_callback at 0x7ffea471d668>*)
Save this file.

Parameters

- **filename** (*str*) – the file to which to save the .sav file
- **callback** (*function*) – a progress callback function

CHAPTER 5

Projects

A project is a wrapper around a song that gives the song a name and a version.

In addition to the `pylsdj.Project` object itself, the `pylsdj.projects` module contains functions for loading projects from `.srm` and `.lsdsng` files.

5.1 Usage Examples

```
from pylsdj import Project, load_srm, load_lsdsng

# Load a .srm file
srm_proj = load_srm("test1.srm")

# Load a .lsdsng file
lsdsng_proj = load_srm("test2.lsdsng")

# Convert the .srm project to .lsdsng
srm_proj.save_lsdsng("test1_conv.lsdsng")

# Get the srm project's song
song = srm_proj.song
```

5.2 API Documentation

pylsdj.**load_lsdsng**(*filename*)
Load a Project from a `.lsdsng` file.

Parameters **filename** – the name of the file from which to load

Return type `pylsdj.Project`

pylsdj.**load_srm**(*filename*)
Load a Project from an `.srm` file.

Parameters **filename** – the name of the file from which to load

Return type `pylsdj.Project`

class pylsdj.**Project**(*name*, *version*, *size_blks*, *data*)

name = None
the project's name

save (*filename*)
Save a project in .lsdsng format to the target file.

Parameters **filename** – the name of the file to which to save

Deprecated use `save_lsdsng(filename)` instead

save_lsdsng (*filename*)
Save a project in .lsdsng format to the target file.

Parameters **filename** – the name of the file to which to save

save_srm (*filename*)
Save a project in .srm format to the target file.

Parameters **filename** – the name of the file to which to save

size_blks = None
the size of the song in filesystem blocks

song
the song associated with the project

version = None
the project's version (incremented on every save in LSDJ)

CHAPTER 6

Songs

A song contains all the information about its notes and the instruments that control how the notes sound. It also contains settings related to how LSDJ should play the song.

6.1 Notes

A song is defined by its *sequence*. A sequence consists of a number of sequence **steps**. Each step specifies a chain for each of the Game Boy's four audio channels (pulse 1, pulse 2, wave, and noise). See *Chains* for more information on how Chains are structured.

A song's sequence is stored in its `sequence` field as a two-dimensional dictionary of chains.

6.1.1 Usage Examples

```
from pylsdj import Sequence

# Get the chain in step $3 of PU2 from the sequence
curr_chain = song.sequence[Sequence.PU2][0x3]

# Get that same chain from the global chains table
curr_chain_another_way = song.chains[curr_chain.index]

# Get chain $2D from the global chain table
chain_two_d = song.chains[0x2d]
```

6.2 Instruments

The sound of a note is determined by an instrument. An instrument can also refer to a synth or a macro table to control how it behaves over time.

A song contains global tables for instruments, synths and macro tables. These are stored in the song's `instruments`, `synths` and `tables` fields, resp.

6.3 Appearance and Playback Behavior

Songs also have a number of fields that control the appearance of LSDJ and its synchronization setting. A complete overview of what all these settings do is out of this document's scope; see the API documentation below for a list of supported settings.

6.4 API Documentation

class `pylsdj.`**`Song`**(*song_data*)

A song consists of a sequence of chains, one per channel.

`bookmarks`
list of screen bookmarks

`chains`
the song's chain table, represented as a list of Chain objects

`clock`
the amount of time LSDJ has been used since the last memory reset, represented as a Clock object

`clone`
chain cloning depth; one of `"deep"`, `"slim"`

`colorset`
the selected LSDJ colorset

`file_changed`
`1` if the file has changed since last save, `0` otherwise

`font`
the selected LSDJ font

`global_clock`
the amount of time LSDJ has been used total, represented as a Clock object

`grooves`
the song's groove table

`instruments`
the song's instrument table, represented as a list of Instrument objects

`key_delay`
the delay before key repeat is activated for Game Boy buttons

`key_repeat`
the key repeat speed for Game Boy buttons

`phrases`
the song's phrase table, represented as a list of Phrase objects

`prelisten`
if non-zero, play notes and instruments while entering them

`sequence`
the song's sequence, showing the order in which chains are played on each of the four channels

`song_version`
the song's version number

speech_instrument
the song's speech instrument settings, represented as a SpeechInstrument object

sync_setting
LSDJ's sync setting; one of `"off"`, `"slave"`, `"master"`, `"midi"`, `"nano"`, and `"keyboard"`

tables
the song's table of macro tables, represented as Table objects

tempo
the song's tempo

CHAPTER 7

Chains

A chain is a list of phrases for a single channel.

A chain can have up to 16 phrases, each of which is associated with a transpose (how much the phrase's notes should be shifted up).

The `pylsdj.chain.Chain` class is a convenience wrapper around one of a song's chains.

7.1 Usage Examples

```
# Access the second transpose in chain $05
song.chains[0x05].transposes[1]

# Access the fifth phrase in chain $53
song.chains[0x53].phrase[4]
```

7.2 API Reference

class `pylsdj.chain.`**`Chain`**(*song*, *index*)

A chain is a sequence of phrases for a single channel. Each phrase can be transposed by a number of semitones.

CHAPTER 8

Phrases

Each phrase consists of a sequence of notes. Each note can have a parameterized effect and instrument associated with it.

8.1 API Documentation

class pylsdj.**Phrase**(*song*, *index*)

A phrase is a sequence of notes for a single channel.

fx
a list of the phrase's effects, one byte per effect

fx_val
a list of the phrase's effect parameters, one byte per effect

index
the phrase's index within its parent song's phrase table

instruments
a list of Instruments, None where no instrument is defined

notes
a list of the phrase's notes, one byte per note

song
a reference to the phrase's parent song

CHAPTER 9

Clocks

LSDJ's clocks serve as a way to track the amount of time the current song has been worked on. The global clock also has a checksum, which provides a check against file corruption.

`pylsdj.clock.TotalClock` is a wrapper around a song's global clock data. Modifying any of the clock's fields (`days`, `hours`, or `minutes`) will also update the checksum accordingly.

`pylsdj.clock.Clock` is a wrapper around a song's local clock data. It only tracks hours and minutes.

9.1 Usage Examples

Get a clock's days

```
>>> clock.days
5
```

Set a clock's hours:

```
>>> clock.hours = 6
```

9.2 API Documentation

class `pylsdj.clock.`**`TotalClock`**(*clock_data*)

`checksum`
the clock's checksum (days + hours + minutes)

`days`
The total number of days on the clock.

`hours`
The total number of hours on the clock.

`minutes`
The total number of minutes on the clock.

class `pylsdj.clock.`**`Clock`**(*clock_data*)

`hours`
The total number of hours on the clock.

minutes
The total number of minutes on the clock.

CHAPTER 10

Instruments

`pylsdj.Instrument` is a wrapper class allowing manipulation of a project's instrument. It is typically accessed by looking up the instrument in its parent song's `instruments` field.

10.1 Importing and Exporting Instruments

Instruments export in what I'm calling `lsdinst` format, which is really just a JSON encoding of the instrument's data.

Importing an instrument is handled by its parent song, so that it can do the necessary bookkeeping if the instrument's type changes.

class `pylsdj.`**`Instruments`**(*song*)

`import_from_file`(*index*, *filename*)
Import this instrument's settings from the given file. Will automatically add the instrument's synth and table to the song's synths and tables if needed.

Note that this may invalidate existing instrument accessor objects.

Parameters

- **index** – the index into which to import
- **filename** – the file from which to load

Raises ImportException if importing failed, usually because the song doesn't have enough synth or table slots left for the instrument's synth or table

class `pylsdj.`**`Instrument`**(*song*, *index*)

`export_to_file`(*filename*)
Export this instrument's settings to a file.

Parameters **filename** – the name of the file

10.1.1 Usage Examples

```
# Editing a song's instrument $06
instrument = song.instruments[0x06]
```

```
# Change the instrument's name
instrument.name = "ABCDE"

# Export the instrument to a file
instrument.export_to_file("my_instrument.lsdinst")

# Import the instrument, overwriting instrument $09
song.instruments.import_from_file(0x09, "my_instrument.lsdinst")
```

10.2 Instrument Fields

All instrument types have the following fields:

- `name`: the instrument's name
- `type`: the instrument's type (pulse, wave, noise, or kit)

Different instruments have different additional fields, corresponding to the fields that an instrument has in LSDJ. These fields are described below.

10.2.1 Vibrato

The pulse, wave, and kit instrument types all have a vibrato control, accessed through their `vibrato` fields, which has the following structure:

class `pylsdj.`**`Vibrato`**(*data*)

`direction`
'down' or 'up'

`type`
`hf` (for high frequency sine), `sawtooth`, `saw` or `square`

10.2.2 Pulse Instruments

class `pylsdj.`**`PulseInstrument`**(*song*, *index*)

`automate`
if True, automation is on

`envelope`
the noise instrument's volume envelope (8-bit integer)

`name`
the instrument's name (5 characters, zero-padded)

`phase_finetune`
detune pulse channel 1 down, channel 2 up; in LSDJ, this is `PU FINE` (4-bit integer)

`phase_transpose`
detune pulse channel 2 this many semitones; in LSDJ, this is `PU2 TUNE` (8-bit integer)

`sound_length`
the instrument sound's length, a 6-bit integer or `unlimited` if the sound plays forever

sweep
modulates the sound's frequency; only works on pulse 1 (8-bit integer)

table
a `'pylsdj.Table'` referencing the instrument's table, or None if the instrument doesn't have a table

type
the instrument's type (`pulse`, `wave`, `kit` or `noise`)

vibrato
instrument's vibrato settings

wave
the pulse's wave width; `12.5%`, `25%`, `50%` or `75%`

10.2.3 Wave Instruments

class `pylsdj.`**`WaveInstrument`**(*song*, *index*)

automate
if True, automation is on

name
the instrument's name (5 characters, zero-padded)

play_type
how to play the synth sound; `once`, `loop`, `ping-pong`, or `manual`

repeat
the synth sound's repeat point (4-bit integer)

speed
how fast the sound should be played back (4-bit integer)

steps
length of the synth sound (4-bit integer)

synth
the wave's synth settings

table
a `'pylsdj.Table'` referencing the instrument's table, or None if the instrument doesn't have a table

type
the instrument's type (`pulse`, `wave`, `kit` or `noise`)

vibrato
instrument's vibrato settings

volume
the sound's volume; 0 through 3

10.2.4 Noise Instrument Fields

class `pylsdj.`**`NoiseInstrument`**(*song*, *index*)

automate
if True, automation is on

envelope
the noise instrument's volume envelope (8-bit integer)

name
the instrument's name (5 characters, zero-padded)

s_cmd
`free` or `stable`. When free, altering noise shape with the S command can sometimes mute the sound. When stable, sound will never be muted by accident. My understanding is that this setting exists for backwards-compatibility of behavior in old LSDJ instruments

sound_length
the instrument sound's length, a 6-bit integer or `unlimited` if the sound plays forever

sweep
modulates the sound's frequency; only works on pulse 1 (8-bit integer)

table
a `'pylsdj.Table'` referencing the instrument's table, or None if the instrument doesn't have a table

type
the instrument's type (`pulse`, `wave`, `kit` or `noise`)

10.2.5 Kit Instrument Fields

class `pylsdj.`**`KitInstrument`**(*song*, *index*)

automate
if True, automation is on

dist_type
algorithm used when two kits are mixed together; `clip`, `shape`, `shap2` or `wrap`

half_speed
if true, play samples at half their normal speed

keep_attack_1
loop sample in kit 1 and start playing from beginning

keep_attack_2
loop sample in kit 2 and start playing from beginning

kit_1
the index of the first kit in LSDJ's kit list

kit_2
the index of the second kit in LSDJ's kit list

length_1
the length of kit 1's sound (0 means 'always play the sample to the end' and is displayed as AUT in LSDJ)

length_2
the length of kit 2's sound (0 means 'always play the sample to the end' and is displayed as AUT in LSDJ)

loop_1
loop sample in kit 1 and start playing from an offset

loop_2
loop sample in kit 2 and start playing from an offset

name
the instrument's name (5 characters, zero-padded)

offset_1
kit 1's loop start point (if loop_1 is True and keep_attack_1 is False)

offset_2
kit 2's loop start point (if loop_2 is True and keep_attack_2 is False)

pitch
sample pitch shift (8-bit integer)

table
a `pylsdj.Table` referencing the instrument's table, or None if the instrument doesn't have a table

type
the instrument's type (`pulse`, `wave`, `kit` or `noise`)

vibrato
instrument's vibrato settings

volume
the kit's volume; 0 to 3

CHAPTER 11

Synths

Synths are used by the wave channel to define the shape of its waveform and how that shape changes over time.

A synth is defined by a sequence of 16 waveforms. In LSDJ waveforms can be drawn by hand, or can be generated by tweaking the softsynth's parameters.

If you update the synth's waves or its parameters, the synth's *wave synth overwrite lock* in its parent song will be updated appropriately. Modifying the wave frames manually will enable the lock, while modifying its parameters will disable the lock.

11.1 Usage Examples

```
# Get the raw waveforms for synth $3
waves = song.synths[0x3].waves

# Get the end volume for synth $9
vol = song.synths[0x9].end.volume
```

11.2 API Reference

class `pylsdj.`**`Synth`**(*song*, *index*)

`distortion`
use `"clip"` or `"wrap"` distortion

`end`
parameters for the end of the sound, represented as a SynthSoundParams object

`filter_resonance`
boosts the signal around the cutoff frequency, to change how bright or dull the wave sounds

`filter_type`
the type of filter applied to the waveform; one of `"lowpass"`, `"highpass"`, `"bandpass"`, `"allpass"`

`index`
the synth's index within its parent song's synth table

`phase_type`
compresses the waveform horizontally; one of `"normal"`, `"resync"`, `"resync2"`

song
the synth's parent Song

start
parameters for the start of the sound, represented as a SynthSoundParams object

wave_synth_overwrite_lock
if True, the synth's waveforms override its synth parameters; if False, its synth parameters override its waveforms

waveform
the synth's waveform type; one of `"sawtooth"`, `"square"`, `"sine"`

waves
a list of the synth's waveforms, each of which is a list of bytes

class `pylsdj.`**SynthSoundParams**(*params*, *overwrite_lock*)

filter_cutoff
the filter's cutoff frequency

phase_amount
the amount of phase shift, `0` = no phase, `0x1f` = maximum phase

vertical_shift
the amount to shift the waveform vertically

volume
the wave's volume

CHAPTER 12

Tables

Tables define the behavior of an instrument over time.

Each table consists of a list of volume envelopes, transposes and effect commands. How (or whether) these commands are applied to an instrument depends on that instrument's settings.

12.1 Usage Examples

```
# Get table $4 from the song's table of tables
table = song.tables[0x4]

# Alternatively, get it from instrument $b
table = song.instruments[0xb].table

# Set the envelope in row $5 to $A6
table.envelopes[0x5] = 0xa6

# Set the value of the first effect's parameter to 5 in row $6
table.fx1[0x6].value = 5
```

12.2 API Reference

class pylsdj.**Table**(*song*, *index*)

Each table is a sequence of transposes, commands, and amplitude changes that can be applied to any channel.

envelopes

a list of the table's volume envelopes

fx1

a list of the table's first effects, represented as TableFX objects

fx2

a list of the table's first effects, represented as TableFX objects

index

the table's index within its parent song's table of macro tables

song

the table's parent Song

transposes
a list of the table's volume transposes

class pylsdj.**TableFX**(*params*, *table_index*, *fx_index*)

command
the effect's command

value
the command's parameter

CHAPTER 13

Speech Instrument

LSDJ has a speech instrument (loaded into instrument slot $40) that can synthesize words from allophones.

If you're looking for a way to break down words into allophones, the CMU Pronouncing Dictionary is a good place to start.

13.1 Speech Instrument Structure

The speech instrument consists of a list of **words**. Each word has a name, and a list of **sounds**. Each sound consists of an allophone and a length.

13.2 Usage Examples

```
# Get word $5 defined in the speech instrument
word = song.speech_instrument.words[0x5]

# Extract the word's allophones
allophones = [sound.allophone for sound in word.sounds]

# Change the fifth allophone to 'OY'
word.sounds[4].allophone = 'OY'

# Change the length of the 10th allophone to 5
word.sounds[9].length = 5

# Change the 3rd word's name to 'WORD'
word.sounds[2].name = 'WORD'
```

13.3 API Documentation

class pylsdj.**SpeechInstrument**(*song*)

song
the speech instrument's parent song

words
a list of the speech instrument's defined words, as Word objects

class `pylsdj.`**`Word`**(*song*, *index*)

`name`
the word's name

`sounds`
a list of the sounds that make up the word; each sound has an `allophone` and a `length`

CHAPTER 14

Utilities

pylsdj.utils.**name_without_zeroes**(*name*)

Return a human-readable name without LSDJ's trailing zeroes.

Parameters **name** – the name from which to strip zeroes

Return type the name, without trailing zeroes

CHAPTER 15

Indices and tables

- *genindex*
- *modindex*
- *search*

p