PedalPi - PluginsManager Documentation

Release 1

SrMouraSilva

Nov 17, 2019

Contents

Pythonic management of LV2 audio plugins with mod-host.

Documentation: http://pedalpi-pluginsmanager.readthedocs.io/

Code: https://github.com/PedalPi/PluginsManager

Python Package Index: https://pypi.org/project/PedalPi-PluginsManager

License: Apache License 2.0

CHAPTER 1

Install

Plugin Manager has dependencies that must be installed before installing the library. Among the dependencies are lv2ls to check the installed audio plugins and PortAudio for information on the audio interfaces through PyAudio.

On Debian-based systems, run:

```
sudo apt-get install -y portaudio19-dev python-all-dev lilv-utils --no-install-
↪recommends
```

Of course, for PluginsManager to manage Lv2 audio plugins, it is necessary that they have installed audio plugins to be managed. The Guitarix and Calf Studio projects provide some audio plugins. To install them:

```
pip install PedalPi-PluginsManager
```

CHAPTER 2

Example

Note: Other examples are in the examples folder in the repository.

This examples uses Calf and Guitarix audio plugins.

Download and install mod-host. For more information, check the ModHost section.

Start audio process

```
# In this example, is starting a Zoom g3 series audio interface
jackd -R -P70 -t2000 -dalsa -dhw:Series -p256 -n3 -r44100 -s &
mod-host
```

Play!

```
from pluginsmanager.banks_manager import BanksManager
from pluginsmanager.observer.mod_host.mod_host import ModHost

from pluginsmanager.model.bank import Bank
from pluginsmanager.model.pedalboard import Pedalboard
from pluginsmanager.model.connection import Connection

from pluginsmanager.model.lv2.lv2_effect_builder import Lv2EffectBuilder

from pluginsmanager.model.system.system_effect import SystemEffect
```

Creating a bank

```
# BanksManager manager the banks
manager = BanksManager()

bank = Bank('Bank 1')
manager.append(bank)
```

Connecting with mod_host. Is necessary that the mod_host process already running

```
mod_host = ModHost('localhost')
mod_host.connect()
manager.register(mod_host)
```

Creating pedalboard

```
pedalboard = Pedalboard('Rocksmith')
bank.append(pedalboard)
# or
# bank.pedalboards.append(pedalboard)
```

Loads pedalboard. All changes in pedalboard are reproduced in mod_host

```
mod_host.pedalboard = pedalboard
```

Add effects in the pedalboard

```
builder = Lv2EffectBuilder()

reverb = builder.build('http://calf.sourceforge.net/plugins/Reverb')
fuzz = builder.build('http://guitarix.sourceforge.net/plugins/gx_fuzz_#fuzz_')
reverb2 = builder.build('http://calf.sourceforge.net/plugins/Reverb')

pedalboard.append(reverb)
pedalboard.append(fuzz)
pedalboard.append(reverb2)
# or
# pedalboard.effects.append(reverb2)
```

For obtains automatically the sound card inputs and outputs, use *SystemEffectBuilder*. It requires a *JackClient* instance, that uses JACK-Client.

```
from pluginsmanager.jack.jack_client import JackClient
client = JackClient()

from pluginsmanager.model.system.system_effect_builder import SystemEffectBuilder
sys_effect = SystemEffectBuilder(client).build()
```

For manual input and output sound card definition, use:

```
sys_effect = SystemEffect('system', ['capture_1', 'capture_2'], ['playback_1',
↪'playback_2'])
```

Note: NOT ADD sys_effect in any Pedalboard

Connecting:

```
pedalboard.connect(sys_effect.outputs[0], reverb.inputs[0])

pedalboard.connect(reverb.outputs[0], fuzz.inputs[0])
pedalboard.connect(reverb.outputs[1], fuzz.inputs[0])
pedalboard.connect(fuzz.outputs[0], reverb2.inputs[0])
pedalboard.connect(reverb.outputs[0], reverb2.inputs[0])

pedalboard.connect(reverb2.outputs[0], sys_effect.inputs[0])
pedalboard.connect(reverb2.outputs[0], sys_effect.inputs[1])
```

Connecting using `ConnectionList`:

```
pedalboard.connections.append(Connection(sys_effect.outputs[0], reverb.inputs[0]))

pedalboard.connections.append(Connection(reverb.outputs[0], fuzz.inputs[0]))
pedalboard.connections.append(Connection(reverb.outputs[1], fuzz.inputs[0]))
pedalboard.connections.append(Connection(fuzz.outputs[0], reverb2.inputs[0]))
pedalboard.connections.append(Connection(reverb.outputs[0], reverb2.inputs[0]))

pedalboard.connections.append(Connection(reverb2.outputs[0], sys_effect.inputs[0]))
pedalboard.connections.append(Connection(reverb2.outputs[0], sys_effect.inputs[1]))
```

Set effect status (enable/disable bypass) and param value

```
fuzz.toggle()
# or
# fuzz.active = not fuzz.active

fuzz.params[0].value = fuzz.params[0].minimum / fuzz.params[0].maximum
```

Removing effects and connections:

```
pedalboard.effects.remove(fuzz)

for connection in list(pedalboard.connections):
    pedalboard.disconnect(connection)
    # or
    #pedalboard.connections.remove(connection)

for effect in list(pedalboard.effects):
    pedalboard.effects.remove(effect)
# or
# for index in reversed(range(len(pedalboard.effects))):
    # del pedalboard.effects[index]
```

CHAPTER 3

Observer

`ModHost` is an **observer** (see `UpdatesObserver`). It is informed about all changes that occur in some model instance (`BanksManager`, `Bank`, `Pedalboard`, `Effect`, `Param`, ...), allowing it to communicate with the `mod-host` process transparently.

It is possible to create observers! Some ideas are:

- Allow the use of other hosts (such as Carla);
- Automatically persist changes;
- Automatically update a human-machine interface (such as LEDs and displays that inform the state of the effects).

How to implement and the list of Observers implemented by this library can be accessed in the Observer section.

CHAPTER 4

Maintenance

4.1 Makefile

Execute `make help` for see the options

4.2 Generate documentation

This project uses Sphinx + Read the Docs.

CHAPTER 5

Changelog

5.1 Changelog

5.1.1 Version 0.8.0 – released 11/16/19

- Issue #46 - Notify changes: Pedalboard Name, Pedalboard Data, Bank Data
- Issue #95 - Update readme

Note: In the future, only python version 3.4 or higher will be supported in favor of typing (Issue #50)

5.1.2 Version 0.7.2 – released 11/10/19

- Issue #103 - Now other developers can informs custom messages notification about **changes_**

5.1.3 Version 0.7.1 – released 03/15/18

- Issue #97 - Support tornado 5 and other scripts that uses Python asyncio

5.1.4 Version 0.7.0 – released 02/17/18

- Issue #83 - Add EffectsList for raises errors:
  - Error when add twice the effect instance;
  - Error when add any `effect.is_unique_for_all_pedalboards == True` (e.g.: *`SystemEffect`*);
- Issue #80 - Add ConnectionsList for raises errors:

  - Error when add twice an equals connection;
  - Error when add a connection where the input or output is a effect that not in pedalboard if `effect.is_unique_for_all_pedalboards == False` (e.g.: is not *SystemEffect*).
- Issue #64 - Start process to add Carla support:
  - Create *HostObserver* that *ModHost* and *Carla* extends;
- Defined pattern: Access json metadata (liblilv) from lv2 classes using lv2instance.data attribute.
- Issue #90 - Reverts Issue #66 - Param now informs minimum and maximum in `__dict__()`

5.1.5 Version 0.6.0 – released 11/30/17

- Add makefile. Now is possible run tests and generate docs easily (*make help*);
- Improve *SystemInput* and `SystemOutputs` documentation;
- Issue #57 - Implementing midi support:
  - Now *Effect* list yours *midi_inputs* and *midi_outputs*;
  - *SystemEffect* now supports *midi_inputs* and *midi_outputs*;
  - *Lv2Effect* now supports *midi_inputs* and *midi_outputs*;
  - Created *MidiPort*, *MidiInput*, *MidiOutput*;
  - Created *SystemMidiInput*, *SystemMidiOutput*;
  - Created `Lv2MidiInput`, `Lv2MidiOutput`;
  - *SystemEffectBuilder* now creates *SystemEffect* with your midi outputs and midi inputs;
- Fix autosaver_test creation folder. Now is more easily configure test workspace;
- Refactored *Input*, *Output*: Created *Port* for remove duplicated code;
- Refactored *SystemInput*, *SystemOutput*: Created *SystemPortMixing* for remove duplicated code;
- Refactored *Lv2Input*, *Lv2Output*: Created `Lv2PortMixing` for remove duplicated code;
- `JackClient` - Add attributes: *audio_inputs*, *audio_outputs*, *midi_inputs*, *midi_outputs*;
- Break change: Removed `Output.connect()` and `Output.disconnect()` *Output* methods. Use instead *connect()*, *disconnect()* *Pedalboard* methods;
- Issue #67 - Created *connect()*, *disconnect()* *Pedalboard* methods;
- Fixed Changelog: Now is possible see it in the documentation;
- Issue #38 - Raise erros then add sys_effect in any Pedalboard;
- Issue #65 - Fix documentation bug *SystemEffectBuilder(client).build()* instead *SystemEffectBuilder(client)*;
- Issue #68 - Remove current mod-host pedalboard don't removes systems connection (system.output to system.input);
- Issue #66 - JSON effect improviments: Add plugin version. Remove *min* and *max*;
- Issue #62 - Create a converter MOD pedalboard -> PluginsManager pedalboard;
- Issue #77 - Fix MidiConnection bugs (`SystemMidiInput` and `SystemMidiOutput` doesn't works in `ModHost`);
- Issue #78 - Improve lv2 effect builder error message when plugin not exists;

- *Lv2EffectBuilder* - Add parameter *ignore_unsupported_plugins* for ignore audio plugins errors if it doesn't installed in the system. The previous versions raises error if a audio plugin hasn't installed in the system. Now, is possible use it if *plugins_json* parameter contains your metadata. Observes that, how the audio plugin aren't installed, your use with mod-host or other host will raises errors.

5.1.6 Version 0.5.1 – released 08/16/17

- Issue #52 - *Autosaver* - Change connection with *SystemInput* and *SystemOutput* causes error;
- **Issue #53 - *Autosaver* - Remove effect with connections breaks.** Disable connections removed notification when a effect with connections has removed;
- *Autosaver* - Add `Observable.real_list` attribute for access the list of *ObservableList*;
- Issue #54 - `Mod-host` - Fix feedback_socket optional problem.

5.1.7 Version 0.5.0 – released 05/29/17

- Issue #29 - List audio interfaces
- Issue #32 - Add method to starts mod-host in ModHost instance
- Add banks iterator for PluginsManager
- Improve documentation (Issue #3)
  - Improve Readme: Add lib requirements
  - Add examples folder
  - Informs the changes in Readme (index.html)
- Issue #39 - Add *ObservableList.move()* method (to change order of pedalboards in a bank and banks in a banks manager)
- Issue #44 - Add thread support for observer scope. **Break changes:**
  - Moved `pluginsmanager.model.updates_observer` → `pluginsmanager.observer.updates_observer`
  - Moved `pluginsmanager.model.observer_type` → `pluginsmanager.observer.updates_observer`
  - Moved `pluginsmanager.util.observable_list` → `pluginsmanager.observer.observable_list`
  - Moved `pluginsmanager.modhost` → `pluginsmanager.modhost.observer`
- Created *BanksManager.unregister()* method

5.1.8 Version 0.4.0 – released 05/17/17

- Improve coverage code
- Remove deprecated files (mod-host auto connect)
- Issue #23 - Add method for secure close in mod-host
- Issue #22 - Fastest load pedalboard
- Issue #19 - x-run callback. Create `pluginsmanager.jack.jack_client.JackClient`

5.1.9 Version 0.3.2 – released 05/12/17

- Fix `pluginsmanager.util.builder`: Add folder in pip

5.1.10 Version 0.3.1 – released 05/10/17

- Add class method `Lv2EffectBuilder.plugins_json_file()`

5.1.11 Version 0.3.0 – released 05/08/17

- Add lilvlib support: - Add object Lv2EffectBuilder method - *Lv2EffectBuilder.lv2_plugins_data()*: Scan and generate the lv2 plugins metadata - Add object Lv2EffectBuilder attribute - `Lv2EffectBuilder.plugins()`: List plugins - Add object Lv2EffectBuilder method - *Lv2EffectBuilder.reload()*: Load lv2 metadata

5.1.12 Version 0.2.1 – released 05/07/17

- Refactor *util.persistence_decoder.PersistenceDecoder*: Using now design pattern;
- Fix *ModHost*: Bug when changing value of a parameter from a plugin;
- observable_list: Add method *ObservableList.pop()*.

5.1.13 Version 0.2.0 – released 03/31/17

- Initial release

CHAPTER 6

API

Contents:

6.1 PedalPi - PluginsManager - Jack

6.1.1 pluginsmanager.jack.jack_client.JackClient

6.1.2 pluginsmanager.jack.jack_interface.JackInterfaces

6.1.3 pluginsmanager.jack.jack_interface.AudioInterface

6.2 PedalPi - PluginsManager - Host

6.2.1 PedalPi - PluginsManager - HostObserver

HostObserver

class pluginsmanager.observer.host_observer.host_observer.**HostObserver**

HostObserver contains the basis for Host implementations, like *ModHost* or *Carla*.

It is an `UpdatesObserver`. With it, can be apply the current pedalboard changes transparently.

HostObserver contains an algorithm for improve the change of the current pedalboard. Also, HostObserver process the updates and define abstract methods that hosts needs to implements, usually only with the important part.

__del__()

Calls *close()* method for remove the audio plugins loaded and closes connection with the host.

__init__()

Initialize self. See help(type(self)) for accurate signature.

close()
Remove the audio plugins loaded and closes connection with the host.

connect()
Connect with the host

on_bank_updated(*bank*, *update_type*, ***kwargs*)
Called when changes occurs in any *Bank*

Parameters

- **bank** (Bank) – Bank changed.
- **update_type** (UpdateType) – Change type
- **index** (*int*) – Bank index (or old index if update_type == UpdateType.DELETED)
- **origin** (BanksManager) – BanksManager that the bank is (or has) contained
- **Bank** – Contains the old bank occurs a *UpdateType.UPDATED*

on_connection_updated(*connection*, *update_type*, *pedalboard*, ***kwargs*)
Called when changes occurs in any *pluginsmanager.model.connection.Connection* of Pedalboard (adding, updating or removing connections)

Parameters

- **connection** (pluginsmanager.model.connection.Connection) – Connection changed
- **update_type** (UpdateType) – Change type
- **pedalboard** (Pedalboard) – Pedalboard that the connection is (or has) contained

on_effect_status_toggled(*effect*, ***kwargs*)
Called when any *Effect* status is toggled

Parameters **effect** (Effect) – Effect when status has been toggled

on_effect_updated(*effect*, *update_type*, *index*, *origin*, ***kwargs*)
Called when changes occurs in any *Effect*

Parameters

- **effect** (Effect) – Effect changed
- **update_type** (UpdateType) – Change type
- **index** (*int*) – Effect index (or old index if update_type == UpdateType.DELETED)
- **origin** (Pedalboard) – Pedalboard that the effect is (or has) contained

on_param_value_changed(*param*, ***kwargs*)
Called when a param value change

Parameters **param** (Param) – Param with value changed

on_pedalboard_updated(*pedalboard*, *update_type*, ***kwargs*)
Called when changes occurs in any *Pedalboard*

Parameters

- **pedalboard** (Pedalboard) – Pedalboard changed
- **update_type** (UpdateType) – Change type
- **index** (*int*) – Pedalboard index (or old index if update_type == UpdateType.DELETED)

- **origin** (Bank) – Bank that the pedalboard is (or has) contained
- **old** (Pedalboard) – Contains the old pedalboard when occurs a *UpdateType.UPDATED*

pedalboard
Currently managed pedalboard (current pedalboard)

Getter Current pedalboard - Pedalboard loaded by mod-host

Setter Set the pedalboard that will be loaded by mod-host

Type *Pedalboard*

start()
Invokes the process.

HostError

class pluginsmanager.observer.host_observer.host_observer.**HostError**

6.2.2 PedalPi - PluginsManager - ModHost

About *mod-host*

mod-host is a LV2 host for Jack controllable via socket or command line. With it you can load audio plugins, connect, manage plugins.

For your use, is necessary download it

```
git clone https://github.com/moddevices/mod-host
cd mod-host
make
make install
```

Then boot the JACK process and start the *mod-host*. Details about "JACK" can be found at https://help.ubuntu.com/community/What%20is%20JACK

```
# In this example, is starting a Zoom g3 series audio interface
jackd -R -P70 -t2000 -dalsa -dhw:Series -p256 -n3 -r44100 -s &
mod-host
```

You can now connect to the mod-host through the Plugins Manager API. Create a ModHost object with the address that is running the *mod-host* process. Being in the same machine, it should be *'localhost'*

```
mod_host = ModHost('localhost')
mod_host.connect()
```

Finally, register the mod-host in your BanksManager. Changes made to the current pedalboard will be applied to *mod-host*

```
manager = BanksManager()
# ...
manager.register(mod_host)
```

To change the current pedalboard, change the *pedalboard* parameter to *mod_host*. Remember that for changes to occur in *mod-host*, the *pedalboard* must belong to some *bank* of *banks_manager*.

```
mod_host.pedalboard = my_awesome_pedalboard
```

ModHost

class pluginsmanager.observer.mod_host.mod_host.**ModHost**(*address='localhost'*, *port=5555*)

Python port for mod-host Mod-host is a LV2 host for Jack controllable via socket or command line.

This class offers the mod-host control in a python API:

```
# Create a mod-host, connect and register it in banks_manager
mod_host = ModHost('localhost')
mod_host.connect()
banks_manager.register(mod_host)

# Set the mod_host pedalboard for a pedalboard that the bank
# has added in banks_manager
mod_host.pedalboard = my_awesome_pedalboard
```

The changes in current pedalboard (`pedalboard` attribute of *mod_host*) will also result in mod-host:

```
driver = my_awesome_pedalboard.effects[0]
driver.active = False
```

Note: For use, is necessary that the mod-host is running, for use, access

- Install dependencies
- Building mod-host
- Running mod-host

For more JACK information, access Demystifying JACK – A Beginners Guide to Getting Started with JACK

Example:

In this example, is starting a Zoom G3 series audio interface. Others interfaces maybe needs others configurations.

```
# Starting jackdump process via console
jackd -R -P70 -t2000 -dalsa -dhw:Series -p256 -n3 -r44100 -s &
# Starting mod-host
mod-host &
```

Parameters

- **address** (*string*) – Computer mod-host process address (IP). If the process is running on the same computer that is running the python code uses *localhost*.
- **port** (*int*) – Socket port on which mod-host should be running. Default is *5555*

__del__()

Calls `close()` method for remove the audio plugins loaded and closes connection with mod-host.

```
>>> mod_host = ModHost()
>>> del mod_host
```

Note: If the mod-host process has been created with `start()` method, it will be finished.

__init__(*address='localhost'*, *port=5555*)
Initialize self. See help(type(self)) for accurate signature.

close()
Remove the audio plugins loaded and closes connection with mod-host.

Note: If the mod-host process has been created with `start()` method, it will be finished.

connect()
Connect the object with mod-host with the _address_ parameter informed in the constructor method (`__init__()`)

start()
Invokes the mod-host process.

mod-host requires JACK to be running. mod-host does not startup JACK automatically, so you need to start it before running mod-host.

Note: This function is experimental. There is no guarantee that the process will actually be initiated.

ModHost internal

The classes below are for internal use of mod-host

Connection

class pluginsmanager.observer.mod_host.connection.**Connection**(*socket_port=5555*, *address='localhost'*)
Class responsible for managing an API connection to the mod-host process via socket

__init__(*socket_port=5555*, *address='localhost'*)
Initialize self. See help(type(self)) for accurate signature.

close()
Closes socket connection

send(*message*)
Sends message to *mod-host*.

Note: Uses *ProtocolParser* for a high-level management. As example, view *Host*

Parameters **message** (*string*) – Message that will be sent for *mod-host*

Host

class pluginsmanager.observer.mod_host.host.**Host**(*address='localhost'*, *port=5555*)
Bridge between *mod-host* API and *mod-host* process

__init__(*address='localhost'*, *port=5555*)
Initialize self. See help(type(self)) for accurate signature.

add(*effect*)
Add an LV2 plugin encapsulated as a jack client

Parameters **effect** (Lv2Effect) – Effect that will be loaded as LV2 plugin encapsulated

close()
Quit the connection with mod-host

connect(*connection*)
Connect two effect audio ports

Parameters **connection** (pluginsmanager.model.connection.Connection) – Connection with the two effect audio ports (output and input)

disconnect(*connection*)
Disconnect two effect audio ports

Parameters **connection** (pluginsmanager.model.connection.Connection) – Connection with the two effect audio ports (output and input)

quit()
Quit the connection with mod-host and stop the mod-host process

remove(*effect*)
Remove an LV2 plugin instance (and also the jack client)

Parameters **effect** (Lv2Effect) – Effect that your jack client encapsulated will removed

set_param_value(*param*)
Set a value to given control

Parameters **param** (Lv2Param) – Param that the value will be updated

set_status(*effect*)
Toggle effect processing

Parameters **effect** (Lv2Effect) – Effect with the status updated

ProtocolParser

class pluginsmanager.observer.mod_host.protocol_parser.**ProtocolParser**
Prepare the objects to mod-host string command

static add(*effect*)
`add <lv2_uri> <instance_number>`

add a LV2 plugin encapsulated as a jack client

e.g.:

```
add http://lv2plug.in/plugins/eg-amp 0
```

instance_number must be any value between 0 ~ 9999, inclusively

Parameters **effect** (Lv2Effect) – Effect will be added

static bypass(*effect*)

bypass <instance_number> <bypass_value>

toggle plugin processing

e.g.:

```
bypass 0 1
```

- if bypass_value = 1 bypass plugin
- if bypass_value = 0 process plugin

Parameters **effect** (Lv2Effect) – Effect that will be active the bypass or disable the bypass

static connect(*connection*)

connect <origin_port> <destination_port>

connect two plugin audio ports

e.g.:

```
connect system:capture_1 plugin_0:in
```

Parameters **connection** (pluginsmanager.model.connection.Connection) – Connection with a valid *Output* and *Input*

static disconnect(*connection*)

disconnect <origin_port> <destination_port>

disconnect two plugin audio ports

e.g.:

```
disconnect system:capture_1 plugin_0:in
```

Parameters **connection** (pluginsmanager.model.connection.Connection) – Connection with a valid *Output* and *Input*

static help()

help

show a help message

static load(*filename*)

load <file_name>

load a history command file dummy way to save/load workspace state

e.g.:

```
load my_setup
```

Note: Not implemented yet

static midi_learn(*self*, *plugin*, *param*)

`midi_learn <instance_number> <param_symbol>`

This command maps starts MIDI learn for a parameter

e.g.:

```
midi_learn 0 gain
```

Note: Not implemented yet

static midi_map(*plugin*, *param*, *midi_chanel*, *midi_cc*)

`midi_map <instance_number> <param_symbol> <midi_channel> <midi_cc>`

This command maps a MIDI controller to a parameter

e.g.:

```
midi_map 0 gain 0 7
```

Note: Not implemented yet

static midi_unmap(*plugin*, *param*)

`midi_unmap <instance_number> <param_symbol>`

This command unmaps the MIDI controller from a parameter

e.g.:

```
unmap 0 gain
```

Note: Not implemented yet

static monitor()

`monitor <addr> <port> <status>`

open a socket port to monitoring parameters

e.g.:

```
monitor localhost 12345 1
```

- if `status = 1` start monitoring
- if `status = 0` stop monitoring

Note: Not implemented yet

static param_get(*param*)

`param_get <instance_number> <param_symbol>`

get the value of the request control

e.g.:

```
param_get 0 gain
```

Parameters **param** (Lv2Param) – Parameter that will be get your current value

static param_monitor()
param_monitor <instance_number> <param_symbol> <cond_op> <value>

do monitoring a plugin instance control port according given condition

e.g.:

```
param_monitor 0 gain > 2.50
```

Note: Not implemented yet

static param_set(*param*)
param_set <instance_number> <param_symbol> <param_value>

set a value to given control

e.g.:

```
param_set 0 gain 2.50
```

Parameters **param** (Lv2Param) – Parameter that will be updated your value

static preset_load()
preset_load <instance_number> <preset_uri>

load a preset state to given plugin instance

e.g.:

```
preset_load 0 "http://drobilla.net/plugins/mda/presets#JX10-moogcury-lite"
```

Note: Not implemented yet

static preset_save()
preset_save <instance_number> <preset_name> <dir> <file_name>

save a preset state from given plugin instance

e.g.:

```
preset_save 0 "My Preset" /home/user/.lv2/my-presets.lv2 mypreset.ttl
```

Note: Not implemented yet

static preset_show()
preset_show <instance_number> <preset_uri>

show the preset information of requested instance / URI

e.g.:

```
preset_show 0 http://drobilla.net/plugins/mda/presets#EPiano-bright
```

Note: Not implemented yet

static quit()
quit

bye!

static remove(*effect*)
remove <instance_number>

remove a LV2 plugin instance (and also the jack client)

e.g.:

```
remove 0
```

Parameters **effect** (Lv2Effect) – Effect will be removed

static save(*filename*)
save <file_name>

saves the history of typed commands dummy way to save/load workspace state

e.g.:

```
save my_setup
```

Note: Not implemented yet

6.2.3 PedalPi - PluginsManager - Carla

It is in alpha, some methods aren't implemented, as effects connection and disconnection.

About *Carla*

In development

Carla

class pluginsmanager.observer.carla.carla.**Carla**(*path*)

Python port for carla Carla is a fully-featured audio plugin host, with support for many audio drivers and plugin formats. It's open source and licensed under the GNU General Public License, version 2 or later.

This class offers the Carla control in a python API:

```
# Create a carla, connect and register it in banks_manager
host = Carla('localhost')
host.connect()
banks_manager.register(host)

# Set the carla.pedalboard for a pedalboard that the bank
# has added in banks_manager
host.pedalboard = my_awesome_pedalboard
```

The changes in current pedalboard (`pedalboard` attribute of *carla*) will also result in carla host:

```
driver = my_awesome_pedalboard.effects[0]
driver.active = False
```

Note: For use, is necessary that the carla is running, for use, access

- Install dependencies
- Building Carla
- Running Carla

For more JACK information, access Demystifying JACK – A Beginners Guide to Getting Started with JACK

Example:

In this example, is starting a Zoom G3 series audio interface. Others interfaces maybe needs others configurations.

```
# Starting jackdump process via console
jackd -R -P70 -t2000 -dalsa -dhw:Series -p256 -n3 -r44100 -s &
# Starting carla host
# FIXME
```

Parameters **path** (*Path*) – Path that carla are persisted.

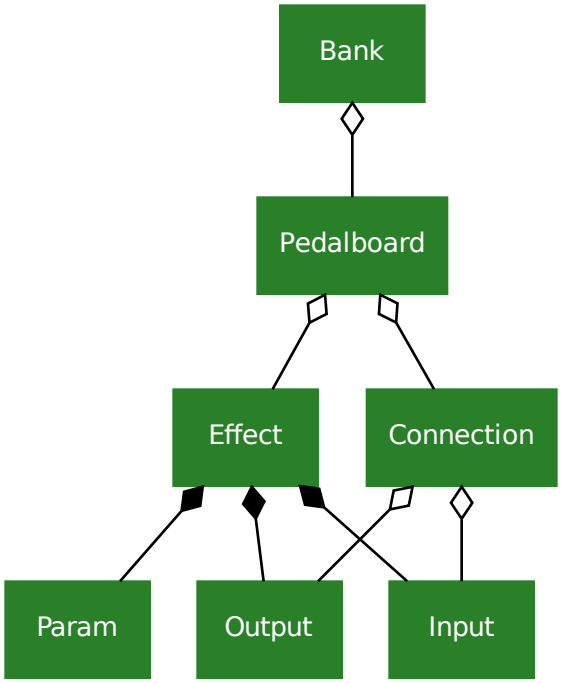
__init__(*path*)
Initialize self. See help(type(self)) for accurate signature.

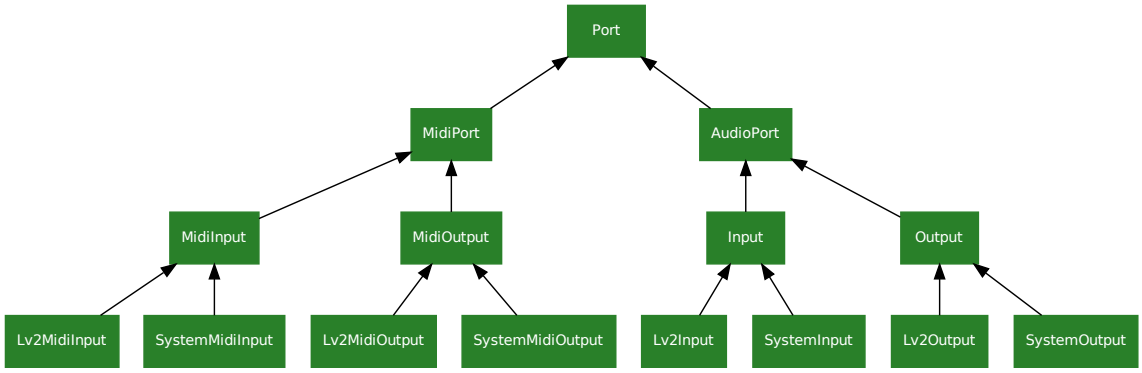
CarlaError

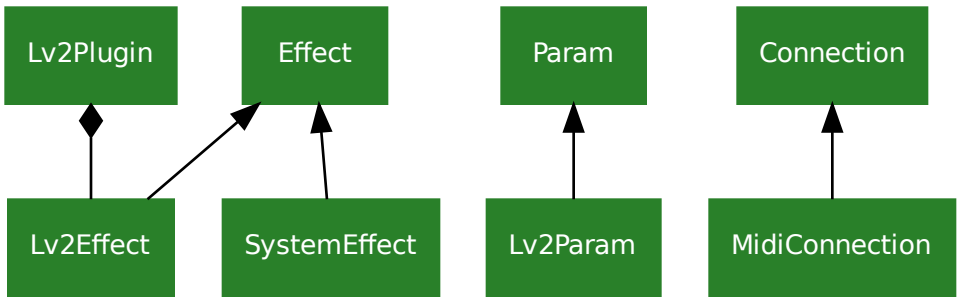
class pluginsmanager.observer.carla.carla.**CarlaError**

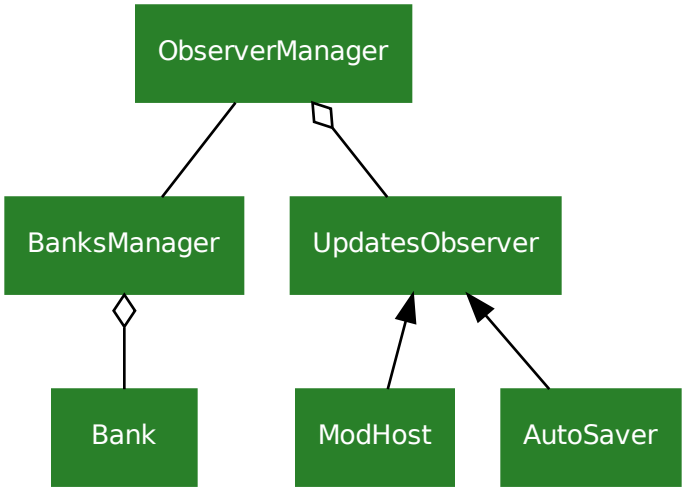
6.3 PedalPi - PluginsManager - Models

This page contains the model classes.


Bank
Pedalboard
Effect
Connection
Param
Output
Input


Port
MidiPort
AudioPort
MidiInput
MidiOutput
Input
Output
Lv2MidiInput
SystemMidiInput
Lv2MidiOutput
SystemMidiOutput
Lv2Input
SystemInput
Lv2Output
SystemOutput





6.3.1 BanksManager

class pluginsmanager.banks_manager.**BanksManager**(*banks=None*)

BanksManager manager the banks. In these is possible add banks, obtains the banks and register observers for will be notified when occurs changes (like added new pedalboard, rename bank, set effect param value or state)

For use details, view Readme.rst example documentation.

Parameters **banks** (*list[Bank]*) – Banks that will be added in this. Useful for loads banks previously loaded, like banks persisted and recovered.

__init__(*banks=None*)

Initialize self. See help(type(self)) for accurate signature.

__iter__()

Iterates banks of the banksmanager:

```
>>> banks_manager = BanksManager()
>>> for index, bank in enumerate(banks_manager):
...     print(index, '-', bank)
```

Returns Iterator for banks list

append(*bank*)
Append the bank in banks manager. It will be monitored, changes in this will be notified for the notifiers.

Parameters **bank** (Bank) – Bank that will be added in this

enter_scope(*observer*)
Informs that changes occurs by the observer and isn't necessary informs the changes for observer

Parameters **observer** (UpdatesObserver) – Observer that causes changes

exit_scope()
Closes the last observer scope added

observers

Returns Observers registered in BanksManager instance

register(*observer*)
Register an observer for it be notified when occurs changes.

For more details, see *UpdatesObserver*

Parameters **observer** (UpdatesObserver) – Observer that will be notified then occurs changes

unregister(*observer*)
Remove the observers of the observers list. It will not receive any more notifications when occurs changes.

Parameters **observer** (UpdatesObserver) – Observer you will not receive any more notifications then occurs changes.

6.3.2 Bank

class pluginsmanager.model.bank.**Bank**(*name*)
Bank is a data structure that contains *Pedalboard*. It's useful for group common pedalboards, like "Pedalboards will be used in the Sunday show"

A fast bank overview:

```
>>> bank = Bank('RHCP')
>>> californication = Pedalboard('Californication')
```

```
>>> # Add pedalboard in bank - mode A
>>> bank.append(californication)
>>> californication.bank == bank
True
```

```
>>> bank.pedalboards[0] == californication
True
```

```
>>> # Add pedalboard in bank - mode B
>>> bank.pedalboards.append(Pedalboard('Dark Necessities'))
>>> bank.pedalboards[1].bank == bank
True
```

```
>>> # If you needs change pedalboards order (swap), use pythonic mode
>>> bank.pedalboards[1], bank.pedalboards[0] = bank.pedalboards[0], bank.
↪pedalboards[1]
>>> bank.pedalboards[1] == californication
True
```

```
>>> # Set pedalboard
>>> bank.pedalboards[0] = Pedalboard("Can't Stop")
>>> bank.pedalboards[0].bank == bank
True
```

```
>>> del bank.pedalboards[0]
>>> bank.pedalboards[0] == californication # Pedalboard Can't stop rermoved,␣
↪first is now the californication
True
```

You can also toggle pedalboards into different banks:

```
>>> bank1.pedalboards[0], bank2.pedalboards[2] = bank2.pedalboards[0], bank1.
↪pedalboards[2]
```

Parameters **name** (*string*) – Bank name

__init__ (*name*)
Initialize self. See help(type(self)) for accurate signature.

append (*pedalboard*)
Add a *Pedalboard* in this bank

This works same as:

```
>>> bank.pedalboards.append(pedalboard)
```

or:

```
>>> bank.pedalboards.insert(len(bank.pedalboards), pedalboard)
```

Parameters **pedalboard** (Pedalboard) – Pedalboard that will be added

index
Returns the first occurrence of the bank in your PluginsManager

json
Get a json decodable representation of this bank

Return dict json representation

name
Bank name

Getter Bank name

Setter Set the Bank name

Type string

simple_identifier
Simple identifier for index file human comprehension and reordering :return string:

6.3.3 Pedalboard

class pluginsmanager.model.pedalboard.**Pedalboard**(*name*)
Pedalboard is a patch representation: your structure contains *Effect* and *Connection*:

```
>>> pedalboard = Pedalboard('Rocksmith')
>>> bank.append(pedalboard)

>>> builder = Lv2EffectBuilder()
>>> pedalboard.effects
[]
>>> reverb = builder.build('http://calf.sourceforge.net/plugins/Reverb')
>>> pedalboard.append(reverb)
>>> pedalboard.effects
[<Lv2Effect object as 'Calf Reverb'  active at 0x7f60effb09e8>]

>>> fuzz = builder.build('http://guitarix.sourceforge.net/plugins/gx_fuzzfacefm_#_
↪fuzzfacefm_')
>>> pedalboard.effects.append(fuzz)

>>> pedalboard.connections
[]
>>> pedalboard.connections.append(Connection(sys_effect.outputs[0], fuzz.
↪inputs[0])) # View SystemEffect for more details
>>> pedalboard.connections.append(Connection(fuzz.outputs[0], reverb.inputs[0]))
>>> # It works too
>>> pedalboard.connect(reverb.outputs[1], sys_effect.inputs[0])
>>> pedalboard.connections
[<Connection object as 'system.capture_1 -> GxFuzzFaceFullerMod.In' at␣
↪0x7f60f45f3f60>, <Connection object as 'GxFuzzFaceFullerMod.Out -> Calf Reverb.
↪In L' at 0x7f60f45f57f0>, <Connection object as 'Calf Reverb.Out R -> system.
↪playback_1' at 0x7f60f45dacc0>]

>>> pedalboard.data
{}
>>> pedalboard.data = {'my-awesome-component': True}
>>> pedalboard.data
{'my-awesome-component': True}
```

For load the pedalboard for play the songs with it:

```
>>> mod_host.pedalboard = pedalboard
```

All changes[1] in the pedalboard will be reproduced in mod-host. [1] Except in data attribute, changes in this does not interfere with anything.

Parameters **name** (*string*) – Pedalboard name

__init__(*name*)
Initialize self. See help(type(self)) for accurate signature.

append(*effect*)

Add a *Effect* in this pedalboard

This works same as:

```
>>> pedalboard.effects.append(effect)
```

or:

```
>>> pedalboard.effects.insert(len(pedalboard.effects), effect)
```

Parameters **effect** (Effect) – Effect that will be added

connect(*output_port*, *input_port*)

Connect two *Effect* instances in this pedalboard. For this, is necessary informs the output port origin and the input port destination:

```
>>> pedalboard.append(driver)
>>> pedalboard.append(reverb)
>>> driver_output = driver.outputs[0]
>>> reverb_input = reverb.inputs[0]
>>> Connection(driver_output, reverb_input) in driver.connections
False
>>> pedalboard.connect(driver_output, reverb_input)
>>> Connection(driver_output, reverb_input) in driver.connections
True
```

Parameters

- **output_port** (Port) – Effect output port
- **input_port** (Port) – Effect input port

connections

Return the pedalboard connections list

Note: Because the connections is an *ObservableList*, it isn't settable. For replace, del the connections unnecessary and add the necessary connections

---

data

Custom information about pedalboard that is necessary to be persisted. Example is effects disposition in a visual modelling pedalboard.

Note: This operation only will notifies changes if the setter is called:

```
>>> data = {'level': 50}
>>> # This call the observer on_custom_change(CustomChange.PEDALBOARD_DATA,␣
↪UpdateType.UPDATED, pedalboard=pedalboard)
>>> pedalboard.data = data
>>> # This doesn't call the observer on_custom_change(...)
>>> data['level'] = 80
>>> # But this call, even though the object is the same.
>>> pedalboard.data = data
```

Getter Data

Setter Set the data

Type dict

disconnect(*output_port*, *input_port*)

Remove a connection between (two ports of) *Effect* instances. For this, is necessary informs the output port origin and the input port destination:

```
>>> pedalboard.append(driver)
>>> pedalboard.append(reverb)
>>> driver_output = driver.outputs[0]
>>> reverb_input = reverb.inputs[0]
>>> pedalboard.connect(driver_output, reverb_input)
>>> Connection(driver_output, reverb_input) in driver.connections
True
>>> pedalboard.disconnect(driver_output, reverb_input)
>>> Connection(driver_output, reverb_input) in driver.connections
False
```

Parameters

- **output_port** (*Port*) – Effect output port
- **input_port** (*Port*) – Effect input port

effects

Return the effects presents in the pedalboard

Note: Because the effects is an *ObservableList*, it isn't settable. For replace, del the effects unnecessary and add the necessary effects

---

index

Returns the first occurrence of the pedalboard in your bank

json

Get a json decodable representation of this pedalboard

Return dict json representation

name

Pedalboard name

Getter Pedalboard name

Setter Set the pedalboard name

Type string

6.3.4 Connection

class pluginsmanager.model.connection.**Connection**(*output_port*, *input_port*)

Connection represents a connection between two distinct effects by your *AudioPort* (effect *Output* with effect *Input*):

```
>>> from pluginsmanager.model.pedalboard import Pedalboard
>>> californication = Pedalboard('Californication')
>>> californication.append(driver)
>>> californication.append(reverb)
```

```
>>> guitar_output = sys_effect.outputs[0]
>>> driver_input = driver.inputs[0]
>>> driver_output = driver.outputs[0]
>>> reverb_input = reverb.inputs[0]
>>> reverb_output = reverb.outputs[0]
>>> amp_input = sys_effect.inputs[0]
```

```
>>> # Guitar -> driver -> reverb -> amp
>>> californication.connections.append(Connection(guitar_output, driver_input))
>>> californication.connections.append(Connection(driver_output, reverb_input))
>>> californication.connections.append(Connection(reverb_output, amp_input))
```

Another way to use implicitly connections:

```
>>> californication.connect(guitar_output, driver_input)
>>> californication.connect(driver_output, reverb_input)
>>> californication.connect(reverb_output, amp_input)
```

Parameters

- **output_port** (Output) – Audio output port that will be connected with audio input port
- **input_port** (Input) – Audio input port that will be connected with audio output port

__eq__(*other*)
Return self==value.

__hash__()
Return hash(self).

__init__(*output_port*, *input_port*)
Initialize self. See help(type(self)) for accurate signature.

__repr__()
Return repr(self).

input

Return Output Input connection port

json
Get a json decodable representation of this effect

Return dict json representation

output

Return Output Output connection port

ports_class

Return class Port class that this connection only accepts

6.3.5 MidiConnection

class pluginsmanager.model.midi_connection.**MidiConnection**(*output_port*, *input_port*)

MidiConnection represents a connection between two distinct effects by your *MidiPort* (effect *MidiOutput* with effect *MidiInput*):

```
>>> californication = Pedalboard('Californication')
>>> californication.append(driver)
>>> californication.append(reverb)
```

```
>>> output_port = cctonode1.midi_outputs[0]
>>> input_port = cctonode2.midi_inputs[0]
```

```
>>> californication.connections.append(MidiConnection(output_port, input_port))
```

Another way to use implicitly connections:

```
>>> californication.connect(output_port, input_port)
```

Parameters

- **output_port** (MidiOutput) – MidiOutput port that will be connected with midi input port
- **input_port** (MidiInput) – MidiInput port that will be connected with midi output port

ports_class

Return class Port class that this connection only accepts

6.3.6 ConnectionsList

class pluginsmanager.model.connections_list.**ConnectionsList**(*pedalboard*)

ConnectionsList contains a *ObservableList* and checks the effect instance unity restrictions

__init__(*pedalboard*)

Initialize self. See help(type(self)) for accurate signature.

6.3.7 Effect

class pluginsmanager.model.effect.**Effect**

Representation of a audio plugin instance - LV2 plugin encapsulated as a jack client.

Effect contains a *active* status (off=bypass), a list of *Param*, a list of *Input* and a list of *Connection*:

```
>>> reverb = builder.build('http://calf.sourceforge.net/plugins/Reverb')
>>> pedalboard.append(reverb)
>>> reverb
<Lv2Effect object as 'Calf Reverb' active at 0x7fd58d874ba8>

>>> reverb.active
True
>>> reverb.toggle()
>>> reverb.active
```

(continues on next page)

(continued from previous page)

```
False
>>> reverb.active = True
>>> reverb.active
True

>>> reverb.inputs
(<Lv2Input object as In L at 0x7fd58c583208>, <Lv2Input object as In R at␣
↪0x7fd58c587320>)
>>> reverb.outputs
(<Lv2Output object as Out L at 0x7fd58c58a438>, <Lv2Output object as Out R at␣
↪0x7fd58c58d550>)
>>> reverb.params
(<Lv2Param object as value=1.5 [0.4000000059604645 - 15.0] at 0x7fd587f77908>,
↪<Lv2Param object as value=5000.0 [2000.0 - 20000.0] at 0x7fd587f7a9e8>,
↪<Lv2Param object as value=2 [0 - 5] at 0x7fd587f7cac8>, <Lv2Param object as␣
↪value=0.5 [0.0 - 1.0] at 0x7fd587f7eba8>, <Lv2Param object as value=0.25 [0.0 -␣
↪2.0] at 0x7fd58c576c88>, <Lv2Param object as value=1.0 [0.0 - 2.0] at␣
↪0x7fd58c578d68>, <Lv2Param object as value=0.0 [0.0 - 500.0] at 0x7fd58c57ae80>,
↪ <Lv2Param object as value=300.0 [20.0 - 20000.0] at 0x7fd58c57df98>, <Lv2Param␣
↪object as value=5000.0 [20.0 - 20000.0] at 0x7fd58c5810f0>)
```

Parameters **pedalboard** (*Pedalboard*) – Pedalboard where the effect lies.

__init__()
Initialize self. See help(type(self)) for accurate signature.

__repr__()
Return repr(self).

active
Effect status: active or bypass

Getter Current effect status

Setter Set the effect Status

Type bool

connections

Return list[Connection] Connections that this effects is present (with input or output port)

index
Returns the first occurrence of the effect in your pedalboard

inputs

Return list[Input] Inputs of effect

is_possible_connect_itself
return bool: Is possible connect the with it self?

is_unique_for_all_pedalboards

return bool: Is unique for all pedalboards? Example: *SystemEffect* is unique for all pedalboards

json
Get a json decodable representation of this effect

Return dict json representation

midi_inputs

Return list[MidiInput] MidiInputs of effect

midi_outputs

Return list[MidiOutput] MidiOutputs of effect

outputs

Return list[Output] Outputs of effect

params

Return list[Param] Params of effect

toggle()
Toggle the effect status: `self.active = not self.active`

use_real_identifier
Instances of audio plugins are dynamically created, so the effect identifier for the jack can be set.

However, SystemEffect correspond (mostly) to the audio interfaces already present in the computational system. The identifier for their jack has already been set.

return bool: For this audio plugin, is necessary use the real effect identifier? Example: *Lv2Effect* is False Example: *SystemEffect* is True

version

Return string Effect version

6.3.8 EffectsList

class pluginsmanager.model.effects_list.**EffectsList**
EffectsList contains a *ObservableList* and checks the effect instance unity restrictions

6.3.9 Port

class pluginsmanager.model.port.**Port**(*effect*)
Port is a parent abstraction for inputs and outputs

Parameters **effect** (Effect) – Effect that contains port

__init__(*effect*)
Initialize self. See help(type(self)) for accurate signature.

__repr__()
Return repr(self).

connection_class

Returns Class used for connections in this port

effect

Returns Effect that this port is related

index

Returns Index in the effect related based in your category. As example, if this port is a `input`, the index returns your position in the inputs ports.

json
Get a json decodable representation

Return dict json representation

symbol

Returns Identifier for this port

6.3.10 AudioPort

class pluginsmanager.model.audio_port.**AudioPort**(*effect*)

Port is a parent abstraction for audio inputs and audio outputs

connection_class

Return Connection Class used for connections in this port

6.3.11 Input

class pluginsmanager.model.input.**Input**(*effect*)

Input is the medium in which the audio will go into effect to be processed.

Effects usually have a one (mono) or two inputs (stereo L + stereo R). But this isn't a rule: Some have only *Output*, like audio frequency generators, others have more than two.

For obtains the inputs:

```
>>> my_awesome_effect
<Lv2Effect object as 'Calf Reverb' active at 0x7fd58d874ba8>
>>> my_awesome_effect.inputs
(<Lv2Input object as In L at 0x7fd58c583208>, <Lv2Input object as In R at␣
↪0x7fd58c587320>)

>>> effect_input = my_awesome_effect.inputs[0]
>>> effect_input
<Lv2Input object as In L at 0x7fd58c583208>

>>> symbol = effect_input.symbol
>>> symbol
'in_l'

>>> my_awesome_effect.inputs[symbol] == effect_input
True
```

For connections between effects, see *connect()* and *disconnect() Pedalboard* class methods.

Parameters **effect** (*Effect*) – Effect of input

index

Returns Input index in the your effect

6.3.12 Output

class pluginsmanager.model.output.**Output**(*effect*)

Output is the medium in which the audio processed by the effect is returned.

Effects usually have a one (mono) or two outputs (stereo L + stereo R). .

For obtains the outputs:

```
>>> my_awesome_effect
<Lv2Effect object as 'Calf Reverb' active at 0x7fd58d874ba8>
>>> my_awesome_effect.outputs
(<Lv2Output object as Out L at 0x7fd58c58a438>, <Lv2Output object as Out R at␣
↪0x7fd58c58d550>)

>>> output = my_awesome_effect.outputs[0]
>>> output
<Lv2Output object as Out L at 0x7fd58c58a438>

>>> symbol = my_awesome_effect.outputs[0].symbol
>>> symbol
'output_l'

>>> my_awesome_effect.outputs[symbol] == output
True
```

For connections between effects, see *connect()* and *disconnect() Pedalboard* class methods.

Parameters **effect** (Effect) – Effect that contains the output

index

Returns Output index in the your effect

6.3.13 MidiPort

class pluginsmanager.model.midi_port.**MidiPort**(*effect*)

Port is a parent abstraction for midi inputs and midi outputs

connection_class

Return MidiConnection Class used for connections in this port

6.3.14 MidiInput

class pluginsmanager.model.midi_input.**MidiInput**(*effect*)

MidiInput is the medium in which the midi input port will go into effect to be processed.

For obtains the inputs:

```
>>> cctonode
<Lv2Effect object as 'CC2Note'  active at 0x7efe5480af28>
>>> cctonode.midi_inputs
(<Lv2MidiInput object as MIDI In at 0x7efe54535dd8>,)

>>> midi_input = cctonode.midi_inputs[0]
>>> midi_input
<Lv2MidiInput object as MIDI In at 0x7efe54535dd8>

>>> symbol = midi_input.symbol
>>> symbol
'midiin'

>>> cctonode.midi_inputs[symbol] == midi_input
True
```

For connections between effects, see *connect()* and *disconnect() Pedalboard* class methods.

Parameters **effect** (Effect) – Effect of midi input

index

Returns MidiInput index in the your effect

6.3.15 MidiOutput

class pluginsmanager.model.midi_output.**MidiOutput**(*effect*)

MidiOutput is the medium in which the midi output processed by the effect is returned.

For obtains the outputs:

```
>>> cctonode
<Lv2Effect object as 'CC2Note'  active at 0x7efe5480af28>
>>> cctonode.outputs
(<Lv2MidiOutput object as MIDI Out at 0x7efe5420eeb8>,)

>>> midi_output = cctonode.midi_outputs[0]
>>> midi_output
<Lv2Output object as Out L at 0x7fd58c58a438>

>>> symbol = midi_output.symbol
>>> symbol
'midiout'

>>> cctonode.midi_outputs[symbol] == midi_output
True
```

For connections between effects, see *connect()* and *disconnect() Pedalboard* class methods.

Parameters **effect** (Effect) – Effect that contains the output

index

Returns Output index in the your effect

6.3.16 Param

class pluginsmanager.model.param.**Param**(*effect*, *default*)

Param represents an Audio Plugin Parameter:

```
>>> my_awesome_effect
<Lv2Effect object as 'Calf Reverb' active at 0x7fd58d874ba8>
>>> my_awesome_effect.params
(<Lv2Param object as value=1.5 [0.4000000059604645 - 15.0] at 0x7fd587f77908>,
↪<Lv2Param object as value=5000.0 [2000.0 - 20000.0] at 0x7fd587f7a9e8>,
↪<Lv2Param object as value=2 [0 - 5] at 0x7fd587f7cac8>, <Lv2Param object as␣
↪value=0.5 [0.0 - 1.0] at 0x7fd587f7eba8>, <Lv2Param object as value=0.25 [0.0 -␣
↪2.0] at 0x7fd58c576c88>, <Lv2Param object as value=1.0 [0.0 - 2.0] at␣
↪0x7fd58c578d68>, <Lv2Param object as value=0.0 [0.0 - 500.0] at 0x7fd58c57ae80>,
↪ <Lv2Param object as value=300.0 [20.0 - 20000.0] at 0x7fd58c57df98>, <Lv2Param␣
↪object as value=5000.0 [20.0 - 20000.0] at 0x7fd58c5810f0>)

>>> param = my_awesome_effect.params[0]
>>> param
```

(continues on next page)

(continued from previous page)

```
<Lv2Param object as value=1.5 [0.4000000059604645 - 15.0] at 0x7fd587f77908>

>>> param.default
1.5
>>> param.value = 14

>>> symbol = param.symbol
>>> symbol
'decay_time'
>>> param == my_awesome_effect.params[symbol]
True
```

Parameters

- **effect** (Effect) – Effect in which this parameter belongs
- **default** – Default value (initial value parameter)

__init__(*effect*, *default*)
Initialize self. See help(type(self)) for accurate signature.

__repr__(**args*, ***kwargs*)
Return repr(self).

default
Default parameter value. Then a effect is instanced, the value initial for a parameter is your default value.

Getter Default parameter value.

effect

Returns Effect in which this parameter belongs

json
Get a json decodable representation of this param

Return dict json representation

maximum

Returns Greater value that the parameter can assume

minimum

Returns Smaller value that the parameter can assume

symbol

Returns Param identifier

value
Parameter value

Getter Current value

Setter Set the current value

6.4 PedalPi - PluginsManager - Model - Lv2

6.4.1 Lv2EffectBuilder

class pluginsmanager.model.lv2.lv2_effect_builder.**Lv2EffectBuilder**(*plugins_json=None*, *ignore_unsupported_plugins=True*)

Generates lv2 audio plugins instance (as *Lv2Effect* object).

Note: In the current implementation, the data plugins are persisted in *plugins.json*.

Parameters

- **plugins_json** (*Path*) – Plugins json path file
- **ignore_unsupported_plugins** (*bool*) – Not allows instantiation of uninstalled or unrecognized audio plugins?

__init__(*plugins_json=None*, *ignore_unsupported_plugins=True*)
Initialize self. See help(type(self)) for accurate signature.

build(*lv2_uri*)
Returns a new *Lv2Effect* by the valid lv2_uri

Parameters **lv2_uri** (*string*) –

Return Lv2Effect Effect created

lv2_plugins_data()
Generates a file with all plugins data info. It uses the lilvlib library.

PluginsManager can manage lv2 audio plugins through previously obtained metadata from the lv2 audio plugins descriptor files.

To speed up usage, data has been pre-generated and loaded into this piped packet. This avoids a dependency installation in order to obtain the metadata.

However, this measure makes it not possible to manage audio plugins that were not included in the list.

To work around this problem, this method - using the lilvlib library - can get the information from the audio plugins. You can use this data to generate a file containing the settings:

```
>>> builder = Lv2EffectBuilder()
>>> plugins_data = builder.lv2_plugins_data()

>>> import json
>>> with open('plugins.json', 'w') as outfile:
>>>     json.dump(plugins_data, outfile)
```

The next time you instantiate this class, you can pass the configuration file:

```
>>> builder = Lv2EffectBuilder(os.path.abspath('plugins.json'))
```

Or, if you want to load the data without having to create a new instance of this class:

```
>>> builder.reload(builder.lv2_plugins_data())
```

Warning: To use this method, it is necessary that the system has the lilv in a version equal to or greater than 0.22.0. Many linux systems currently have previous versions on their package lists, so you need to compile them manually.

In order to ease the work, Pedal Pi has compiled lilv for some versions of linux. You can get the list of .deb packages in https://github.com/PedalPi/lilvlib/releases.

```
# Example
wget https://github.com/PedalPi/lilvlib/releases/download/v1.0.0/python3-
↪lilv_0.22.1.git20160613_amd64.deb
sudo dpkg -i python3-lilv_0.22.1+git20160613_amd64.deb
```

If the architecture of your computer is not contemplated, moddevices provided a script to generate the package. Go to https://github.com/moddevices/lilvlib to get the script in its most up-to-date version.

Return list lv2 audio plugins metadata

plugins_json_file = '/home/docs/checkouts/readthedocs.org/user_builds/pedalpi-pluginsma
Informs the path of the *plugins.json* file. This file contains the lv2 plugins metadata info

reload(*metadata*, *ignore_unsupported_plugins=True*)
Loads the metadata. They will be used so that it is possible to generate lv2 audio plugins.

Parameters

- **metadata** (*list*) – lv2 audio plugins metadata
- **ignore_unsupported_plugins** (*bool*) – Not allows instantiation of uninstalled or unrecognized audio plugins?

6.4.2 Lv2Effect

class pluginsmanager.model.lv2.lv2_effect.**Lv2Effect**(*plugin*)
Representation of a Lv2 audio plugin instance.

For general effect use, see *Effect* class documentation.

It's possible obtains the *Lv2Plugin* information:

```
>>> reverb
<Lv2Effect object as 'Calf Reverb'  active at 0x7f60effb09e8>
>>> reverb.plugin
<Lv2Plugin object as Calf Reverb at 0x7f60effb9940>
```

Parameters **plugin** (Lv2Plugin) –

__init__(*plugin*)
Initialize self. See help(type(self)) for accurate signature.

__repr__()
Return repr(self).

__str__()
Return str(self).

version

Return string Version of plugin of effect

6.4.3 Lv2Input

class pluginsmanager.model.lv2.lv2_input.**Lv2Input**(*effect*, *data*)
Representation of a Lv2 input audio port instance.

For general input use, see *Input* class documentation.

Parameters

- **effect** (Lv2Effect) – Effect that contains the input
- **data** (*dict*) – *input audio port* json representation

__init__(*effect*, *data*)
Initialize self. See help(type(self)) for accurate signature.

data

Return dict Metadata used for provide the required information

in this object

6.4.4 Lv2Output

class pluginsmanager.model.lv2.lv2_output.**Lv2Output**(*effect*, *data*)
Representation of a Lv2 output audio port instance.

For general input use, see *Output* class documentation.

Parameters

- **effect** (Lv2Effect) – Effect that contains the output
- **data** (*dict*) – *output audio port* json representation

__init__(*effect*, *data*)
Initialize self. See help(type(self)) for accurate signature.

data

Return dict Metadata used for provide the required information

in this object

6.4.5 Lv2Param

class pluginsmanager.model.lv2.lv2_param.**Lv2Param**(*effect*, *data*)
Representation of a Lv2 input control port instance.

For general input use, see *Param* class documentation.

Parameters

- **effect** (Lv2Effect) – Effect that contains the param
- **data** (*dict*) – *input control port* json representation

__init__(*effect*, *data*)
Initialize self. See help(type(self)) for accurate signature.

maximum

Returns Greater value that the parameter can assume

minimum

Returns Smaller value that the parameter can assume

symbol

Returns Param identifier

6.4.6 Lv2Plugin

class pluginsmanager.model.lv2.lv2_plugin.**Lv2Plugin**(*json*)

__getitem__(*key*)

Parameters **key** (*string*) – Property key

Returns Returns a Plugin property

__init__(*json*)
Initialize self. See help(type(self)) for accurate signature.

__repr__()
Return repr(self).

__str__()
Return str(self).

data

Returns Json decodable representation of this plugin based in moddevices lilvlib.

json

Returns Json decodable representation of this plugin based in moddevices lilvlib.

6.5 PedalPi - PluginsManager - Model - System

6.5.1 SystemEffectBuilder

class pluginsmanager.model.system.system_effect_builder.**SystemEffectBuilder**(*jack_client*)
Automatic system physical ports detection.

Maybe the midi ports not will recognize. In these cases, you need to start a2jmidid to get MIDI-ALSA ports automatically mapped to JACK-MIDI ports.

Parameters **jack_client** (*JackClient*) – JackClient instance that will get the information to generate *SystemEffect*

__init__(*jack_client*)
Initialize self. See help(type(self)) for accurate signature.

6.5.2 SystemEffect

class pluginsmanager.model.system.system_effect.**SystemEffect**(*representation*, *outputs=None*, *inputs=None*, *midi_outputs=None*, *midi_inputs=None*)

Representation of the system instance: audio and/or midi interfaces.

System output is equivalent with audio input: You connect the instrument in the audio card input and it captures and send the audio to *SystemOutput* for you connect in a input plugins.

System input is equivalent with audio output: The audio card receives the audio processed in your *SystemInput* and send it to audio card output for you connects in amplifier, headset.

Because no autodetection of existing ports in audio card has been implemented, you must explicitly inform in the creation of the SystemEffect object:

```
>>> sys_effect = SystemEffect('system', ('capture_1', 'capture_2'), ('playback_1',
↪ 'playback_2'))
```

Unlike effects that should be added in the pedalboard, SystemEffects MUST NOT:

```
>>> builder = Lv2EffectBuilder()
```

```
>>> pedalboard = Pedalboard('Rocksmith')
>>> reverb = builder.build('http://calf.sourceforge.net/plugins/Reverb')
>>> pedalboard.append(reverb)
```

However the pedalboard must have the connections:

```
>>> pedalboard.connect(sys_effect.outputs[0], reverb.inputs[0])
```

An bypass example:

```
>>> pedalboard = Pedalboard('Bypass example')
>>> sys_effect = SystemEffect('system', ('capture_1', 'capture_2'), ('playback_1',
↪ 'playback_2'))
>>> pedalboard.connect(sys_effect.outputs[0], sys_effect.inputs[0])
>>> pedalboard.connect(sys_effect.outputs[1], sys_effect.inputs[1])
```

You can create multiple SystemEffect for multiple audio/midi interfaces. In the following example, exists Jack provides audio system ports and two midi ports are added by I/O ports:

```
>>> audio_system = SystemEffect('system', inputs=['playback_1', 'playback_2'])
>>> midi_system = SystemEffect('ttymidi', midi_outputs=['MIDI_in'], midi_inputs=[
↪'MIDI_out'])
>>> pedalboard = Pedalboard('MDA-EP')
>>> ep = builder.build('http://moddevices.com/plugins/mda/EPiano')
>>> pedalboard.connect(ep.outputs[0], audio_system.inputs[0])
>>> pedalboard.connect(ep.outputs[1], audio_system.inputs[1])
>>> pedalboard.connect(audio_system.midi_outputs[0], ep.midi_inputs[0])
```

You can check the audio/midi ports defined in your environment using jack_lsp:

```
root@zynthian:~ # As example in Zynthian project: http://zynthian.org
root@zynthian:~ jack_lsp -A
system:playback_1
```

(continues on next page)

(continued from previous page)

```
   alsa_pcm:hw:0:in1
system:playback_2
   alsa_pcm:hw:0:in2
ttymidi:MIDI_in
ttymidi:MIDI_out
Zyncoder:output
Zyncoder:input
```

If you prefer, you can use a unique SystemEffect if *alias* the ports:

```
localhost@localdomain:~ jack_alias system:midi_capture1 ttymidi:MIDI_in
localhost@localdomain:~ jack_alias system:midi_playback1 ttymidi:MIDI_out
```

```
>>> sys_effect = SystemEffect(
...     'system',
...     inputs=['playback_1', 'playback_2'],
...     midi_outputs=['midi_capture1'],
...     midi_inputs=['midi_playback1']
... )
>>> pedalboard = Pedalboard('MDA-EP')
>>> ep = builder.build('http://moddevices.com/plugins/mda/EPiano')
>>> pedalboard.connect(ep.outputs[0], sys_effect.inputs[0])
>>> pedalboard.connect(ep.outputs[1], sys_effect.inputs[1])
>>> pedalboard.connect(sys_effect.midi_outputs[0], ep.midi_inputs[0])
```

Parameters

- **representation** (*string*) – Audio card representation. Usually 'system'
- **outputs** (*tuple(string)*) – Tuple of outputs representation. Usually a output representation starts with `capture_`
- **inputs** (*tuple(string)*) – Tuple of inputs representation. Usually a input representation starts with `playback_`
- **midi_outputs** (*tuple(string)*) – Tuple of midi outputs representation.
- **midi_inputs** (*tuple(string)*) – Tuple of midi inputs representation.

__init__(*representation*, *outputs=None*, *inputs=None*, *midi_outputs=None*, *midi_inputs=None*)
Initialize self. See help(type(self)) for accurate signature.

__str__()
Return str(self).

is_possible_connect_itself
return bool: Is possible connect the with it self?

is_unique_for_all_pedalboards

return bool: Is unique for all pedalboards? Example: *SystemEffect* is unique for all pedalboards

use_real_identifier
return bool: For this audio plugin, is necessary use the real effect identifier?

6.5.3 SystemInput

class pluginsmanager.model.system.system_input.**SystemInput**(*effect*, *symbol*)
Representation of a System input audio port instance.

For general input use, see *Input* class documentation.

Parameters

- **effect** (SystemEffect) – Effect that contains the input
- **symbol** (*string*) – *input audio port* symbol identifier

__init__(*effect*, *symbol*)
Initialize self. See help(type(self)) for accurate signature.

symbol

Returns Identifier for this port

6.5.4 SystemOutput

class pluginsmanager.model.system.system_output.**SystemOutput**(*effect*, *symbol*)
Representation of a System output audio port instance.

For general input use, see *Output* class documentation.

Parameters

- **effect** (SystemEffect) – Effect that contains the input
- **symbol** (*string*) – *output audio port* symbol identifier

__init__(*effect*, *symbol*)
Initialize self. See help(type(self)) for accurate signature.

symbol

Returns Identifier for this port

6.5.5 SystemMidiInput

class pluginsmanager.model.system.system_midi_input.**SystemMidiInput**(*effect*, *symbol*)
Representation of a System midi input port instance.

For general input use, see *Input* and *MidiInput* classes documentation.

Parameters

- **effect** (SystemEffect) – Effect that contains the input
- **symbol** (*string*) – *midi input port* symbol identifier

__init__(*effect*, *symbol*)
Initialize self. See help(type(self)) for accurate signature.

symbol

Returns Identifier for this port

6.5.6 SystemMidiOutput

class pluginsmanager.model.system.system_midi_output.**SystemMidiOutput**(*effect*, *symbol*)

Representation of a System midi output port instance.

For general input use, see *Output* and *MidiOutput* classes documentation.

Parameters

- **effect** (*SystemEffect*) – Effect that contains the input
- **symbol** (*string*) – *midi output port* symbol identifier

__init__(*effect*, *symbol*)
Initialize self. See help(type(self)) for accurate signature.

symbol

Returns Identifier for this port

6.5.7 SystemPortMixing

class pluginsmanager.model.system.system_port_mixing.**SystemPortMixing**(**args*, ***kwargs*)

Contains the default implementation of System ports: *SystemInput*, *SystemOutput*, *SystemMidiInput* and *SystemMidiInput*

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__str__()
Return str(self).

6.6 PedalPi - PluginsManager - Observers

An observer is a class that receives notifications of changes in model classes (*Bank*, *Pedalboard*, *Effect*, *Param* ...).

6.6.1 Implementations

Some useful *UpdatesObserver* classes have been implemented. They are:

- *Autosaver*: Allows save the changes automatically in *json* data files.
- ModHost: Allows use mod-host, a LV2 host for Jack controllable via socket or command line

6.6.2 Using

For use a observer, it's necessary register it in *BanksManager*:

```
>>> saver = Autosaver()  # Autosaver is a UpdatesObserver
>>> banks_manager = BanksManager()
>>> banks_manager.register(saver)
```

For access all observers registered, use *BanksManager.observers*:

```
>>> saver in banks_manager.observers
True
```

For remove a observer:

```
>>> banks_manager.unregister(saver)
```

6.6.3 Creating a observer

It is possible to create observers! Some ideas are:

- Allow the use of other hosts (such as Carla);
- Automatically persist changes;
- Automatically update a human-machine interface (such as LEDs and displays that inform the state of the effects).

For create a observer, is necessary create a class that extends *UpdatesObserver*:

```
class AwesomeObserver(UpdatesObserver):
    ...
```

UpdatesObserver contains a number of methods that must be implemented in the created class. These methods will be called when changes occur:

```
class AwesomeObserver(UpdatesObserver):

    def on_bank_updated(self, bank, update_type, index, origin, **kwargs):
        pass

    def on_pedalboard_updated(self, pedalboard, update_type, index, origin, **kwargs):
        pass

    def on_effect_status_toggled(self, effect, **kwargs):
        pass

    def on_effect_updated(self, effect, update_type, index, origin, **kwargs):
        pass

    def on_param_value_changed(self, param, **kwargs):
        pass

    def on_connection_updated(self, connection, update_type, pedalboard, **kwargs):
        pass
```

Use the `update_type` attribute to check what type of change occurred:

```
class AwesomeObserver(UpdatesObserver):
    """Registers all pedalboards that have been deleted"""

    def __init__(self):
        super(AwesomeObserver, self).__init__()
        self.pedalboards_removed = []

    ...
```

(continues on next page)

(continued from previous page)

```
    def on_pedalboard_updated(self, pedalboard, update_type, index, origin, **kwargs):
        if update_type == UpdateType.DELETED:
            self.pedalboards_removed.append(update_type)

    ...
```

6.6.4 Scope

Notification problem

There are cases where it makes no sense for an observer to be notified of a change. Usually this occurs in interfaces for control, where through them actions can be performed (activate an effect when pressing on a footswitch). Control interfaces need to know of changes that occur so that their display mechanisms are updated when some change occurs through another control interface.

Note that it does not make sense for an interface to be notified of the occurrence of any change if it was the one that performed the action.

A classic example would be an interface for control containing footswitch and a led. The footswitch changes the state of an effect and the led indicates whether it is active or not. If another interface to control (a mobile application, for example) changes the state of the effect to off, the led should reverse its state:

```
class MyControllerObserver(UpdatesObserver):

    ...

    def on_effect_status_toggled(self, effect, **kwargs):
        # Using gpiozero
        # https://gpiozero.readthedocs.io/en/stable/recipes.html#led
        self.led.toggle()
```

However, in this situation, when the footswitch changes the effect state, it is notified of the change itself. What can lead to inconsistency in the led:

```
def pressed():
    effect.toggle()
    led.toggle()

# footswitch is a button
# https://gpiozero.readthedocs.io/en/stable/recipes.html#button
footswitch.when_pressed = pressed
```

In this example, pressing the button:

1. `pressed()` is called;
2. The effect has its changed state (`effect.toggle()`);
3. `on_effect_status_toggled(self, effect, ** kwargs)` is called and the led is changed state (`self.led.toggle()`);
4. Finally, in `pressed()` is called `led.toggle()`.

That is, `led.toggle()` will be **called twice instead of one**.

Scope solution

Using `with` keyword, you can indicate which observer is performing the action, allowing the observer not to be notified of the updates that occur in the `with` scope:

```
>>> with observer1:
>>>     del manager.banks[0]
```

Example

Note: The complete example can be obtained from the examples folder of the repository. observer_scope.py

Consider an Observer who only prints actions taken on a bank:

```
class MyAwesomeObserver(UpdatesObserver):

    def __init__(self, message):
        super(MyAwesomeObserver, self).__init__()
        self.message = message

    def on_bank_updated(self, bank, update_type, **kwargs):
        print(self.message)

    ...
```

We will create two instances of this observer and perform some actions to see how the notification will occur:

```
>>> observer1 = MyAwesomeObserver("Hi! I am observer1")
>>> observer2 = MyAwesomeObserver("Hi! I am observer2")

>>> manager = BanksManager()
>>> manager.register(observer1)
>>> manager.register(observer1)
```

When notification occurs outside a `with` scope, all observers are informed of the change:

```
>>> bank = Bank('Bank 1')
>>> manager.banks.append(bank)
"Hi! I am observer1"
"Hi! I am observer2"
```

We'll now limit the notification by telling you who performed the actions:

```
>>> with observer1:
>>> with observer1:
...     del manager.banks[0]
"Hi! I am observer2"
>>> with observer2:
...     manager.banks.append(bank)
"Hi! I am observer1"
```

If there is `with` inside a `with` block, the behavior will not change, ie it will not be cumulative

```
1 with observer1:
2     manager.banks.remove(bank)
3     with observer2:
4         manager.banks.append(bank)
```

Line 2 will result in `Hi! I am observer2` and line 4 in `Hi! I am observer1`

6.6.5 Base API

UpdateType

class pluginsmanager.observer.update_type.**UpdateType**

Enumeration for informs the change type.

See *UpdatesObserver* for more details

CREATED = 0

Informs that the change is caused by the creation of an object

DELETED = 2

Informs that the change is caused by the removal of an object

UPDATED = 1

Informs that the change is caused by the update of an object

UpdatesObserver

class pluginsmanager.observer.updates_observer.**UpdatesObserver**

The *UpdatesObserver* is an abstract class definition for treatment of changes in some class model. Your methods are called when occurs any change in *Bank*, *Pedalboard*, *Effect*, etc.

To do this, it is necessary that the *UpdatesObserver* objects be registered in some manager, so that it reports the changes. An example of a manager is *BanksManager*.

__init__()

Initialize self. See help(type(self)) for accurate signature.

on_bank_updated(*bank*, *update_type*, *index*, *origin*, ***kwargs*)

Called when changes occurs in any *Bank*

Parameters

- **bank** (Bank) – Bank changed.
- **update_type** (UpdateType) – Change type
- **index** (*int*) – Bank index (or old index if update_type == UpdateType.DELETED)
- **origin** (BanksManager) – BanksManager that the bank is (or has) contained
- **Bank** – Contains the old bank occurs a *UpdateType.UPDATED*

on_connection_updated(*connection*, *update_type*, *pedalboard*, ***kwargs*)

Called when changes occurs in any *pluginsmanager.model.connection.Connection* of Pedalboard (adding, updating or removing connections)

Parameters

- **connection** (pluginsmanager.model.connection.Connection) – Connection changed

- **update_type** (UpdateType) – Change type
- **pedalboard** (Pedalboard) – Pedalboard that the connection is (or has) contained

on_custom_change(*identifier*, **args*, ***kwargs*)
Called in specific changes that do not fit the other methods. See CustomChange for more details.

Also, called when any changes not officially supported by the PluginsManager library are made.

Developers can implement libraries on PluginsManager to control different equipment. For example, you can implement a host that will communicate with a GT100 or Zoom G3 boss pedal. If the device has any features not officially supported by the PluginsManager library, this method is useful.

For example: Zoom G3 pedalboards have level. If someone changes the level value, this method can be used to communicate this information to all other observers.

Parameters **identifier** – Unique identifier to informs that which informs the type of change in which it is being informed

on_effect_status_toggled(*effect*, ***kwargs*)
Called when any *Effect* status is toggled

Parameters **effect** (Effect) – Effect when status has been toggled

on_effect_updated(*effect*, *update_type*, *index*, *origin*, ***kwargs*)
Called when changes occurs in any *Effect*

Parameters

- **effect** (Effect) – Effect changed
- **update_type** (UpdateType) – Change type
- **index** (*int*) – Effect index (or old index if update_type == UpdateType.DELETED)
- **origin** (Pedalboard) – Pedalboard that the effect is (or has) contained

on_param_value_changed(*param*, ***kwargs*)
Called when a param value change

Parameters **param** (Param) – Param with value changed

on_pedalboard_updated(*pedalboard*, *update_type*, *index*, *origin*, ***kwargs*)
Called when changes occurs in any *Pedalboard*

Parameters

- **pedalboard** (Pedalboard) – Pedalboard changed
- **update_type** (UpdateType) – Change type
- **index** (*int*) – Pedalboard index (or old index if update_type == UpdateType.DELETED)
- **origin** (Bank) – Bank that the pedalboard is (or has) contained
- **old** (Pedalboard) – Contains the old pedalboard when occurs a *UpdateType.UPDATED*

pluginsmanager.observer.observable_list.ObservableList

class pluginsmanager.observer.observable_list.**ObservableList**(*lista=None*)
Detects changes in list.

In append, in remove and in setter, the *observer* is callable with changes details

Based in https://www.pythonsheets.com/notes/python-basic.html#emulating-a-list

__contains__(*item*)
See `list.__contains__()` method

__delitem__(*sliced*)
See `list.__delitem__()` method

Calls observer `self.observer(UpdateType.DELETED, item, index)` where **item** is *self[index]*

__getitem__(*index*)
See `list.__getitem__()` method

__init__(*lista=None*)
Initialize self. See help(type(self)) for accurate signature.

__iter__()
See `list.__iter__()` method

__len__()
See `list.__len__()` method

__repr__()
See `list.__repr__()` method

__setitem__(*index*, *val*)
See `list.__setitem__()` method

Calls observer `self.observer(UpdateType.UPDATED, item, index)` if `val != self[index]`

__str__()
See `list.__repr__()` method

append(*item*)
See `list.append()` method

Calls observer `self.observer(UpdateType.CREATED, item, index)` where **index** is *item position*

index(*x*)
See `list.index()` method

insert(*index*, *x*)
See `list.insert()` method

Calls observer `self.observer(UpdateType.CREATED, item, index)`

move(*item*, *new_position*)
Moves a item list to new position

Calls observer `self.observer(UpdateType.DELETED, item, index)` and observer `self.observer(UpdateType.CREATED, item, index)` if `val != self[index]`

Parameters

- **item** – Item that will be moved to new_position
- **new_position** – Item's new position

pop(*index=None*)
See `list.pop()` method

Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list.

Parameters **index** (*int*) – element index that will be removed

Returns item removed

remove(*item*)

See `list.remove()` method

Calls observer `self.observer(UpdateType.DELETED, item, index)` where **index** is *item position*

6.6.6 Implementations API

6.6.7 pluginsmanager.observer.autosaver.autosaver.Autosaver

class pluginsmanager.observer.autosaver.autosaver.**Autosaver**(*data_path*, *auto_save=True*)

The UpdatesObserver *Autosaver* allows save any changes automatically in json data files. Save all plugins changes in json files in a specified path.

It also allows loading of saved files:

```
>>> system_effect = SystemEffect('system', ('capture_1', 'capture_2'), ('playback_
↪1', 'playback_2'))
>>>
>>> autosaver = Autosaver('my/path/data/')
>>> banks_manager = autosaver.load(system_effect)
```

When loads data with *Autosaver*, the autosaver has registered in observers of the banks_manager generated:

```
>>> autosaver in banks_manager.observers
True
```

For manual registering in *BanksManager* uses *register()*:

```
>>> banks_manager = BanksManager()
>>> autosaver = Autosaver('my/path/data/')
>>> autosaver in banks_manager.observers
False
>>> banks_manager.register(autosaver)
>>> autosaver in banks_manager.observers
True
```

After registered, any changes in *Bank*, *Pedalboard*, *Effect*, *Connection* or *Param* which belong to the structure of *BanksManager* instance are persisted automatically by *Autosaver*:

```
>>> banks_manager = BanksManager()
>>> banks_manager.register(autosaver)
>>> my_bank = Bank('My bank')
>>> banks_manager.append(my_bank)
>>> # The bank will be added in banksmanger
>>> # and now is observable (and persisted) by autosaver
```

It's possible disables autosaver for saves manually:

```
>>> autosaver.auto_save = False
>>> autosaver.save(banks_manager)   # save() method saves all banks data
```

Parameters

- **data_path** (*string*) – Path that banks will be saved (each bank in one file)
- **auto_save** (*bool*) – Auto save any change?

__init__(*data_path*, *auto_save=True*)
Initialize self. See help(type(self)) for accurate signature.

load(*system_effect*)
Return a *BanksManager* instance contains the banks present in data_path

Parameters **system_effect** (SystemEffect) – SystemEffect used in pedalboards

Return BanksManager *BanksManager* with banks persisted in data_path

on_bank_updated(*bank*, *update_type*, *index*, *origin*, ***kwargs*)
Called when changes occurs in any *Bank*

Parameters

- **bank** (Bank) – Bank changed.
- **update_type** (UpdateType) – Change type
- **index** (*int*) – Bank index (or old index if update_type == UpdateType.DELETED)
- **origin** (BanksManager) – BanksManager that the bank is (or has) contained
- **Bank** – Contains the old bank occurs a *UpdateType.UPDATED*

on_connection_updated(*connection*, *update_type*, *pedalboard*, ***kwargs*)
Called when changes occurs in any *pluginsmanager.model.connection.Connection* of Pedalboard (adding, updating or removing connections)

Parameters

- **connection** (pluginsmanager.model.connection.Connection) – Connection changed
- **update_type** (UpdateType) – Change type
- **pedalboard** (Pedalboard) – Pedalboard that the connection is (or has) contained

on_effect_status_toggled(*effect*, ***kwargs*)
Called when any *Effect* status is toggled

Parameters **effect** (Effect) – Effect when status has been toggled

on_effect_updated(*effect*, *update_type*, *index*, *origin*, ***kwargs*)
Called when changes occurs in any *Effect*

Parameters

- **effect** (Effect) – Effect changed
- **update_type** (UpdateType) – Change type
- **index** (*int*) – Effect index (or old index if update_type == UpdateType.DELETED)
- **origin** (Pedalboard) – Pedalboard that the effect is (or has) contained

on_param_value_changed(*param*, ***kwargs*)
Called when a param value change

Parameters **param** (Param) – Param with value changed

on_pedalboard_updated(*pedalboard*, *update_type*, *index*, *origin*, ***kwargs*)
Called when changes occurs in any *Pedalboard*

Parameters

- **pedalboard** (Pedalboard) – Pedalboard changed
- **update_type** (UpdateType) – Change type
- **index** (*int*) – Pedalboard index (or old index if update_type == UpdateType.DELETED)
- **origin** (Bank) – Bank that the pedalboard is (or has) contained
- **old** (Pedalboard) – Contains the old pedalboard when occurs a *UpdateType.UPDATED*

save(*banks_manager*)
Save all data from a banks_manager

Parameters **banks_manager** (BanksManager) – BanksManager that your banks data will be persisted

6.7 PedalPi - PluginsManager - Util

6.7.1 DictTuple

class pluginsmanager.util.dict_tuple.**DictTuple**(*elements*, *key_function*)
Dict tuple is a union with dicts and tuples. It's possible obtains an element by index or by a key

The key is not been a int or long instance

Based in http://jfine-python-classes.readthedocs.io/en/latest/subclass-tuple.html

Parameters

- **elements** (*iterable*) – Elements for the tuple
- **key_function** (*lambda*) – Function mapper: it obtains an element and returns your key.

__contains__(*item*)
Return key in self.

__getitem__(*index*)
Return self[key].

__init__(*elements*, *key_function*)
Initialize self. See help(type(self)) for accurate signature.

static __new__(*cls*, *elements*, *key_function*)
Create and return a new object. See help(type) for accurate signature.

6.7.2 ModPedalboardConverter

class pluginsmanager.util.mod_pedalboard_converter.**ModPedalboardConverter**(*mod_ui_path*, *builder*, *ignore_errors=False*)

ModPedalboardConverter is a utility to convert MOD[1] pedalboards structure in plugins manager pedalboard.

For use, is necessary that the computer system contains the mod_ui with your codes compiled[2] and the pedalboard:

```
>>> path = Path('/home/user/git/mod/mod_ui/')
>>> builder = Lv2EffectBuilder()
>>> converter = ModPedalboardConverter(path, builder)
>>> pedalboard_path = Path('/home/user/.pedalboards/pedalboard_name.pedalboard')
>>> system_effect = SystemEffect('system', ['capture_1', 'capture_2'], ['playback_
↪1', 'playback_2'])
>>> pedalboard = converter.convert(pedalboard_path, system_effect)
```

ModPedalboardConverter can try discover the *system_pedalboard* by the pedalboard:

```
>>> path = Path('/home/user/git/mod/mod_ui/')
>>> builder = Lv2EffectBuilder()
>>> converter = ModPedalboardConverter(path, builder)
>>> pedalboard_path = Path('/home/user/.pedalboards/pedalboard_name.pedalboard')
>>> pedalboard = converter.convert(pedalboard_path)
```

If you needs only obtain the system effect:

```
>>> path = Path('/home/user/git/mod/mod_ui/')
>>> builder = Lv2EffectBuilder()
>>> converter = ModPedalboardConverter(path, builder)
>>> pedalboard_path = Path('/home/user/.pedalboards/pedalboard_name.pedalboard')
>>> pedalboard_info = converter.get_pedalboard_info(pedalboard_path)
>>> system_effect = converter.discover_system_effect(pedalboard_info)
```

Parameters

- **mod_ui_path** (*Path*) – path that mod_ui has in the computer.
- **builder** (Lv2EffectBuilder) – Builder for generate the lv2 effects
- **ignore_errors** (*bool*) – Ignore pedalboard problems like connections with undefined ports

__init__(*mod_ui_path*, *builder*, *ignore_errors=False*)

Initialize self. See help(type(self)) for accurate signature.

convert(*pedalboard_path*, *system_effect=None*)

Parameters

- **pedalboard_path** (*Path*) – Path that the pedalboard has been persisted. Generally is in format *path/to/pedalboard/name.pedalboard*
- **system_effect** (SystemEffect) – Effect that contains the audio interface outputs and inputs or None for **auto discover**

[1] MOD, an awesome music enterprise, create the mod-ui, a visual interface that enable create pedalboards in a simple way.
[2] See the docs: https://github.com/moddevices/mod-ui#install

Return Pedalboard Pedalboard loaded

discover_system_effect(*pedalboard_info*)
Generate the system effect based in pedalboard_info

Parameters **pedalboard_info** (*dict*) – For obtain this, see `get_pedalboard_info()`

Return SystemEffect SystemEffect generated based in pedalboard_info

get_pedalboard_info(*path*)

Parameters **path** (*Path*) – Path that the pedalboard has been persisted. Generally is in format `path/to/pedalboard/name.pedalboard`

Return dict pedalboard persisted configurations

6.7.3 PairsList

class pluginsmanager.util.pairs_list.**PairsList**(*similarity_key_function*)
Receives two lists and generates a result list of pairs of equal elements

Uses *calculate* method for generate list

Parameters **similarity_key_function** – Function that receives a element and returns your identifier to do a mapping with elements from another list

__init__(*similarity_key_function*)
Initialize self. See help(type(self)) for accurate signature.

6.7.4 PairsListResult

class pluginsmanager.util.pairs_list.**PairsListResult**

__init__()
Initialize self. See help(type(self)) for accurate signature.

6.7.5 persistence_decoder

class pluginsmanager.util.persistence_decoder.**PersistenceDecoderError**

class pluginsmanager.util.persistence_decoder.**PersistenceDecoder**(*system_effect*)

__init__(*system_effect*)
Initialize self. See help(type(self)) for accurate signature.

class pluginsmanager.util.persistence_decoder.**Reader**(*system_effect*)

__init__(*system_effect*)
Initialize self. See help(type(self)) for accurate signature.

class pluginsmanager.util.persistence_decoder.**BankReader**(*system_effect*)

class pluginsmanager.util.persistence_decoder.**PedalboardReader**(*system_effect*)

class pluginsmanager.util.persistence_decoder.**EffectReader**(*system_effect*)

__init__(*system_effect*)
Initialize self. See help(type(self)) for accurate signature.

class pluginsmanager.util.persistence_decoder.**ConnectionReader**(*pedalboard*, *system_effect*)

__init__(*pedalboard*, *system_effect*)
Initialize self. See help(type(self)) for accurate signature.

generate_builder(*json*, *audio_port*)
:return AudioPortBuilder

6.7.6 pluginsmanager.util.restriction_list.RestrictionList

class pluginsmanager.util.restriction_list.**RestrictionList**
List with validation when add a element

__contains__(*item*)
See *__contains__()* method

__delitem__(*sliced*)
See *__delitem__()* method

__getitem__(*index*)
See *__getitem__()* method

__init__()
Initialize self. See help(type(self)) for accurate signature.

__iter__()
See *__iter__()* method

__len__()
See *__len__()* method

__repr__()
See *__repr__()* method

__setitem__(*index*, *val*)
See *__setitem__()* method

Swap doesn't works:

```
>>> builder = Lv2EffectBuilder()
>>> effects = EffectsList()
>>> effects.append(builder.build('http://calf.sourceforge.net/plugins/Reverb
↪'))
>>> effects.append(builder.build('http://guitarix.sourceforge.net/plugins/gx_
↪fuzzfacefm_#_fuzzfacefm_'))
>>> effects[0], effects[1] = effects[1], effects[0]
pluginsmanager.model.effects_list.AlreadyAddedError: The effect
↪'GxFuzzFaceFullerMod' already added
```

__str__()
See *__repr__()* method

append(*item*)
See *append()* method

index(*x*)
See *index()* method

insert(*index*, *x*)
See *insert()* method

move(*item*, *new_position*)
See *move()* method

pop(*index=None*)
See *pop()* method

remove(*item*)
See *remove()* method

remove_silently(*item*)
Remove item and not notify

Index

Symbols

A

B

C

D

E

G

H

I

J

L

M

N

O

P

V