JACK Audio Connection Kit (JACK) Client for Python

Release 0.2.0

Matthias Geier

June 11, 2015

Contents

This Python module provides bindings for the JACK[1] library.

Documentation: http://jackclient-python.rtfd.org/

Code: http://github.com/spatialaudio/jackclient-python/

Python Package Index: http://pypi.python.org/pypi/JACK-Client/

1 Requirements

Python: Of course, you'll need Python[2]. Any version where CFFI (see below) is supported should work. If you don't have Python installed yet, you should get one of the distributions which already include CFFI (and many other useful things), e.g. Anaconda[3].

CFFI: The C Foreign Function Interface for Python[4] is used to access the C-API of the JACK library from within Python. It supports CPython 2.6, 2.7, 3.x; and is distributed with PyPy[5] 2.0 beta2 or later. You should install it with your package manager (if it's not installed already), or you can get it with pip[6]:

```
pip install cffi --user
```

---

[1] http://jackaudio.org/

[2] http://www.python.org/

[3] http://docs.continuum.io/anaconda/

[4] http://cffi.readthedocs.org/

[5] http://pypy.org/

[6] http://www.pip-installer.org/en/latest/installing.html

JACK library: The JACK[7] library must be installed on your system (and CFFI must be able to find it). Again, you should use your package manager to install it. Make sure you install the JACK daemon (called `jackd`). This will also install the JACK library package. If you prefer, you can of course also download the sources and compile everything locally.

setuptools: This is needed for the installation of the Python module. Most systems will have this installed already, but if not, you can install it with your package manager or you can get it with pip[8]:

```
    pip install setuptools --user
```

2 Installation

Once you have installed the above-mentioned dependencies, you can use pip[9] to download and install the latest release with a single command:

```
pip install JACK-Client --user
```

If you want to install it system-wide for all users (assuming you have the necessary rights), you can just drop the `--user` option.

To un-install, use:

```
pip uninstall JACK-Client
```

If you prefer, you can also download the package from PyPI[10], extract it, change to the main directory and install it using:

```
python setup.py install --user
```

If you want to get the newest development version from Github[11]:

```
git clone https://github.com/spatialaudio/jackclient-python.git
cd jackclient-python
python setup.py develop --user
```

This way, your installation always stays up-to-date, even if you pull new changes from the Github repository.

If you prefer, you can also replace the last command with:

```
pip install --user -e .
```

... where `-e` stands for `--editable`.

If you want to avoid installation altogether, you can simply copy `jack.py` to your working directory (or to any directory in your Python path).

3 Usage

First, import the module:

```
import jack
```

Then, you most likely want to create a new JACK client:

```
client = jack.Client("MyGreatClient")
```

---

[7] http://jackaudio.org/

[8] http://www.pip-installer.org/en/latest/installing.html

[9] http://www.pip-installer.org/en/latest/installing.html

[10] http://pypi.python.org/pypi/JACK-Client/

[11] http://github.com/spatialaudio/jackclient-python/

You probably want to create some input and output ports, too:

```
client.inports.register("input_1")
client.outports.register("output_1")
```

These functions return the newly created port, so you can save it for later:

```
in2 = client.inports.register("input_2")
out2 = client.outports.register("output_2")
```

To see what you can do with the returned objects, have a look at the documentation of the class `jack.OwnPort`.

You can also check what other JACK ports are available:

```
portlist = client.get_ports()
```

If you want, you can also set all kinds of callback functions, for details see the API documentation for the class `jack.Client`.

Once you are ready to run, you should activate your client:

```
client.activate()
```

Once the client is activated, you can make connections (this isn't possible before activating the client):

```
client.connect("system:capture_1", "MyGreatClient:input_1")
client.connect("MyGreatClient:output_1", "system:playback_1")
```

You can also use the port objects from before instead of port names:

```
client.connect(out2, "system:playback_2")
in2.connect("system:capture_2")
```

You can also disconnect ports, there are again several possibilities:

```
client.disconnect("system:capture_1", "MyGreatClient:input_1")
client.disconnect(out2, "system:playback_2")
# disconnect all connections with in2:
in2.disconnect()
```

If you don't need your ports anymore, you can un-register them:

```
in2.unregister()
# unregister all output ports:
client.outports.clear()
```

Finally, you can de-activate your JACK client and close it:

```
client.deactivate()
client.close()
```

4 API Documentation

JACK Client for Python.

http://jackclient-python.rtfd.org/

`jack.`**`CALL_AGAIN`** **= 0**

Possible return value for process callback.

`jack.`**`STOP_CALLING`** **= 1**

Possible return value for process callback.

`jack.`**`SUCCESS`** **= 0**

Possible return value for several callbacks.

`jack.`**`FAILURE`** **= 1**

Possible return value for several callbacks.

class `jack.`**`Client`**(*name*, *use_exact_name=False*, *no_start_server=False*, *servername=None*, *session_id=None*)

Create a new JACK client.

Parameters **name** (*str*) – The desired client name of at most `client_name_size()` characters. The name scope is local to each server. Unless forbidden by the *use_exact_name* option, the server will modify this name to create a unique variant, if needed.

Other Parameters

- **use_exact_name** (*bool*) – Whether an error should be raised if *name* is not unique. See `Status.name_not_unique`.
- **no_start_server** (*bool*) – Do not automatically start the JACK server when it is not already running. This option is always selected if `JACK_NO_START_SERVER` is defined in the calling process environment.
- **servername** (*str*) – Selects from among several possible concurrent server instances. Server names are unique to each user. If unspecified, use `"default"` unless `JACK_DEFAULT_SERVER` is defined in the process environment.
- **session_id** (*str*) – Pass a SessionID Token. This allows the sessionmanager to identify the client again.

`status` **= None**

JACK status. See `Status`.

`name`

The name of the JACK client (read-only).

`samplerate`

The sample rate of the JACK system (read-only).

`blocksize`

The JACK block size (must be a power of two).

The current maximum size that will ever be passed to the process callback. It should only be queried *before* `activate()` has been called. This size may change, clients that depend on it must register a callback with `set_blocksize_callback()` so they will be notified if it does.

Changing the blocksize stops the JACK engine process cycle, then calls all registered callback functions (see `set_blocksize_callback()`) before restarting the process cycle. This will cause a gap in the audio flow, so it should only be done at appropriate stopping points.

`realtime`

Whether JACK is running with `-R` (`--realtime`).

`frames_since_cycle_start`

Time since start of audio block.

The estimated time in frames that has passed since the JACK server began the current process cycle.

`frame_time`

The estimated current time in frames.

This is intended for use in other threads (not the process callback). The return value can be compared with the value of `last_frame_time` to relate time in other threads to JACK time.

`last_frame_time`

The precise time at the start of the current process cycle.

This may only be used from the process callback (see `set_process_callback()`), and can be used to interpret timestamps generated by `frame_time` in other threads with respect to the current process cycle.

This is the only jack time function that returns exact time: when used during the process callback it always returns the same value (until the next process callback, where it will return that value + `blocksize`, etc). The return value is guaranteed to be monotonic and linear in this fashion unless an xrun occurs (see `set_xrun_callback()`). If an xrun occurs, clients must check this value again, as time may have advanced in a non-linear way (e.g. cycles may have been skipped).

`xrun_delayed_usecs`

Delay in microseconds due to the most recent XRUN occurrence.

This probably only makes sense when queried from a callback defined using `set_xrun_callback()`.

`inports`

A list of audio input `Ports`.

New ports can be created and added to this list with the `register()` method. When `unregister()` is called on one of the items in this list, this port is removed from the list. The `clear()` method can be used to unregister all audio input ports at once.

See also:

`Ports`, `OwnPort`

`outports`

A list of audio output `Ports`.

New ports can be created and added to this list with the `register()` method. When `unregister()` is called on one of the items in this list, this port is removed from the list. The `clear()` method can be used to unregister all audio output ports at once.

See also:

`Ports`, `OwnPort`

`midi_inports`

A list of MIDI input `Ports`.

New MIDI ports can be created and added to this list with the `register()` method. When `unregister()` is called on one of the items in this list, this port is removed from the list. The `clear()` method can be used to unregister all MIDI input ports at once.

See also:

`Ports`, `OwnMidiPort`

`midi_outports`

A list of MIDI output `Ports`.

New MIDI ports can be created and added to this list with the `register()` method. When `unregister()` is called on one of the items in this list, this port is removed from the list. The `clear()` method can be used to unregister all MIDI output ports at once.

See also:

`Ports`, `OwnMidiPort`

`owns`(*port*)

Check if a given port belongs to *self*.

Parameters port (*str or Port*) – Full port name or `Port`, `MidiPort`, `OwnPort` or `OwnMidiPort` object.

`activate`()

Activate JACK client.

Tell the JACK server that the program is ready to start processing audio.

`deactivate`(*ignore_errors=True*)

De-activate JACK client.

Tell the JACK server to remove *self* from the process graph. Also, disconnect all ports belonging to it, since inactive clients have no port connections.

`cpu_load``()`

Return the current CPU load estimated by JACK.

This is a running average of the time it takes to execute a full process cycle for all clients as a percentage of the real time available per cycle determined by the *`blocksize`* and *`samplerate`*.

`close``(`*ignore_errors=True*`)`

Close the JACK client.

`connect``(`*source*, *destination*`)`

Establish a connection between two ports.

When a connection exists, data written to the source port will be available to be read at the destination port.

The port types must be identical.

Parameters

- **source** (*str or Port*) – One end of the connection. Must be an output port.
- **destination** (*str or Port*) – The other end of the connection. Must be an input port.

`disconnect``(`*source*, *destination*`)`

Remove a connection between two ports.

Parameters **source, destination** (*str or Port*) – See *`connect()`*.

`transport_start``()`

Start JACK transport.

`transport_stop``()`

Stop JACK transport.

`transport_locate``(`*frame*`)`

Reposition the JACK transport to a new frame number.

Parameters **frame** (*int*) – Frame number.

`set_freewheel``(`*onoff*`)`

Start/Stop JACK's "freewheel" mode.

When in "freewheel" mode, JACK no longer waits for any external event to begin the start of the next process cycle.

As a result, freewheel mode causes "faster than realtime" execution of a JACK graph. If possessed, real-time scheduling is dropped when entering freewheel mode, and if appropriate it is reacquired when stopping.

IMPORTANT: on systems using capabilities to provide real-time scheduling (i.e. Linux kernel 2.4), if onoff is zero, this function must be called from the thread that originally called *`activate()`*. This restriction does not apply to other systems (e.g. Linux kernel 2.6 or OS X).

Parameters **onoff** (*bool*) – If `True`, freewheel mode starts. Otherwise freewheel mode ends.

See also:

`set_freewheel_callback()`

`set_shutdown_callback``(`*callback*, *userdata=None*`)`

Register shutdown callback.

Register a function (and optional argument) to be called if and when the JACK server shuts down the client thread. The function must be written as if it were an asynchonrous POSIX signal handler — use only async-safe functions, and remember that it is executed from another thread. A typical function might set a flag or write to a pipe so that the rest of the application knows that the JACK client thread has shut down.

Note: clients do not need to call this. It exists only to help more complex clients understand what is going on. It should be called before `activate()`.

Note: application should typically signal another thread to correctly finish cleanup, that is by calling `close()` (since `close()` cannot be called directly in the context of the thread that calls the shutdown callback).

Parameters

- **callback** (*callable*) – User-supplied function that is called whenever the JACK daemon is shutdown. It must have this signature:

```
callback(status:Status, reason:str, userdata) -> None
```

  The argument *status* is of type `jack.Status`.

  Note that after server shutdown, *self* is *not* deallocated by libjack, the application is responsible to properly use `close()` to release client ressources.

  Warning: `close()` cannot be safely used inside the shutdown callback and has to be called outside of the callback context.

- **userdata** (*anything*) – This will be passed as third argument when *callback* is called.

`set_process_callback`(*callback*, *userdata=None*)

Register process callback.

Tell the JACK server to call *callback* whenever there is work be done, passing *userdata* as the second argument.

The code in the supplied function must be suitable for real-time execution. That means that it cannot call functions that might block for a long time. This includes malloc, free, printf, pthread_mutex_lock, sleep, wait, poll, select, pthread_join, pthread_cond_wait, etc, etc.

Note: This function cannot be called while the client is activated (after `activate()` has been called).

Parameters

- **callback** (*callable*) – User-supplied function that is called by the engine anytime there is work to be done. It must have this signature:

```
callback(frames:int, userdata) -> int
```

  The argument *frames* specifies the number of frames that have to be processed in the current audio block. It will be the same number as `blocksize` and it will be a power of two. The *callback* must return zero on success (if *callback* shall be called again for the next audio block) and non-zero on error (if *callback* shall not be called again). You can use `CALL_AGAIN` and `STOP_CALLING`, respectively.

- **userdata** (*anything*) – This will be passed as second argument whenever *callback* is called.

`set_freewheel_callback`(*callback*, *userdata=None*)

Register freewheel callback.

Tell the JACK server to call *callback* whenever we enter or leave "freewheel" mode, passing *userdata* as the second argument. The first argument to the callback will be `True` if JACK is entering freewheel mode, and `False` otherwise.

All "notification events" are received in a separated non RT thread, the code in the supplied function does not need to be suitable for real-time execution.

Note: this function cannot be called while the client is activated (after *`activate()`* has been called).

Parameters

- **callback** (*callable*) – User-supplied function that is called whenever jackd starts or stops freewheeling. It must have this signature:

```
callback(starting:bool, userdata) -> None
```

- **userdata** (*anything*) – This will be passed as second argument whenever *callback* is called.

See also:

`set_freewheel()`

`set_blocksize_callback`(*callback*, *userdata=None*)

Register blocksize callback.

Tell JACK to call *callback* whenever the size of the the buffer that will be passed to the process callback is about to change. Clients that depend on knowing the buffer size must supply a *callback* before activating themselves.

All "notification events" are received in a separated non RT thread, the code in the supplied function does not need to be suitable for real-time execution.

Note: this function cannot be called while the client is activated (after *`activate()`* has been called).

Parameters

- **callback** (*callable*) – User-supplied function that is invoked whenever the JACK engine buffer size changes. It must have this signature:

```
callback(blocksize:int, userdata) -> int
```

  The *callback* must return zero on success and non-zero on error. You can use *`SUCCESS`* and *`FAILURE`*, respectively.

  Although this function is called in the JACK process thread, the normal process cycle is suspended during its operation, causing a gap in the audio flow. So, the *callback* can allocate storage, touch memory not previously referenced, and perform other operations that are not realtime safe.

- **userdata** (*anything*) – This will be passed as second argument whenever *callback* is called.

`set_samplerate_callback`(*callback*, *userdata=None*)

Register samplerate callback.

Tell the JACK server to call *callback* whenever the system sample rate changes.

All "notification events" are received in a separated non RT thread, the code in the supplied function does not need to be suitable for real-time execution.

Note: this function cannot be called while the client is activated (after `activate()` has been called).

Parameters

- **callback** (*callable*) – User-supplied function that is called when the engine sample rate changes. It must have this signature:

```
callback(samplerate:int, userdata) -> int
```

  The argument *samplerate* is the new engine sample rate. The *callback* must return zero on success and non-zero on error. You can use `SUCCESS` and `FAILURE`, respectively.

- **userdata** (*anything*) – This will be passed as second argument whenever *callback* is called.

`set_client_registration_callback`(*callback*, *userdata=None*)

Register client registration callback.

Tell the JACK server to call *callback* whenever a client is registered or unregistered, passing *userdata* as a parameter.

All "notification events" are received in a separated non RT thread, the code in the supplied function does not need to be suitable for real-time execution.

Note: this function cannot be called while the client is activated (after `activate()` has been called).

Parameters

- **callback** (*callable*) – User-supplied function that is called whenever a client is registered or unregistered. It must have this signature:

```
callback(name:str, register:bool, userdata) -> None
```

  The first argument contains the client name, the second argument is `True` if the client is being registered and `False` if the client is being unregistered.

- **userdata** (*anything*) – This will be passed as third argument whenever *callback* is called.

`set_port_registration_callback`(*callback*, *userdata=None*)

Register port registration callback.

Tell the JACK server to call *callback* whenever a port is registered or unregistered, passing *userdata* as a parameter.

All "notification events" are received in a separated non RT thread, the code in the supplied function does not need to be suitable for real-time execution.

Note: this function cannot be called while the client is activated (after `activate()` has been called).

Parameters

- **callback** (*callable*) – User-supplied function function that is called whenever a port is registered or unregistered. It must have this signature:

```
callback(port:Port, register:bool, userdata) -> None
```

The first argument is a `Port`, `MidiPort`, `OwnPort` or `OwnMidiPort` object, the second argument is `True` if the port is being registered, `False` if the port is being unregistered.

- **userdata** (*anything*) – This will be passed as third argument whenever *callback* is called.

set_port_connect_callback(*callback*, *userdata=None*)

Register port connect callback.

Tell the JACK server to call *callback* whenever a port is connected or disconnected, passing *userdata* as a parameter.

All “notification events” are received in a separated non RT thread, the code in the supplied function does not need to be suitable for real-time execution.

Note: this function cannot be called while the client is activated (after `activate()` has been called).

Parameters

- **callback** (*callable*) – User-supplied function that is called whenever a port is connected or disconnected. It must have this signature:

```
callback(a:Port, b:Port, connect:bool, userdata) -> None
```

The first and second arguments contain `Port`, `MidiPort`, `OwnPort` or `OwnMidiPort` objects of the ports which are connected or disconnected. The third argument is `True` if the ports were connected and `False` if the ports were disconnected.

- **userdata** (*anything*) – This will be passed as fourth argument whenever *callback* is called.

set_port_rename_callback(*callback*, *userdata=None*)

Register port rename callback.

Tell the JACK server to call *callback* whenever a port is renamed, passing *userdata* as a parameter.

All “notification events” are received in a separated non RT thread, the code in the supplied function does not need to be suitable for real-time execution.

Note: this function cannot be called while the client is activated (after `activate()` has been called).

Parameters

- **callback** (*callable*) – User-supplied function that is called whenever the port name has been changed. It must have this signature:

```
callback(port:Port, old:str, new:str, userdata) -> int
```

The first argument is the port that has been renamed (a `Port`, `MidiPort`, `OwnPort` or `OwnMidiPort` object); the second and third argument is the old and new name, respectively. The *callback* must return zero on success and non-zero on error. You can use `SUCCESS` and `FAILURE`, respectively.

- **userdata** (*anything*) – This will be passed as fourth argument whenever *callback* is called.

`set_graph_order_callback`(*callback*, *userdata=None*)

Register graph order callback.

Tell the JACK server to call *callback* whenever the processing graph is reordered, passing *userdata* as a parameter.

All “notification events” are received in a separated non RT thread, the code in the supplied function does not need to be suitable for real-time execution.

Note: this function cannot be called while the client is activated (after *`activate()`* has been called).

Parameters

- **callback** (*callable*) – User-supplied function that is called whenever the processing graph is reordered. It must have this signature:

```
callback(userdata) -> int
```

  The *callback* must return zero on success and non-zero on error. You can use *`SUCCESS`* and *`FAILURE`*, respectively.

- **userdata** (*anything*) – This will be passed as argument whenever *callback* is called.

`set_xrun_callback`(*callback*, *userdata=None*)

Register xrun callback.

Tell the JACK server to call *callback* whenever there is an xrun, passing *userdata* as a parameter.

All “notification events” are received in a separated non RT thread, the code in the supplied function does not need to be suitable for real-time execution.

Note: this function cannot be called while the client is activated (after *`activate()`* has been called).

Parameters

- **callback** (*callable*) – User-supplied function that is called whenever an xrun has occured. It must have this signature:

```
callback(userdata) -> int
```

  The *callback* must return zero on success and non-zero on error. You can use *`SUCCESS`* and *`FAILURE`*, respectively.

- **userdata** (*anything*) – This will be passed as argument whenever *callback* is called.

See also:

`xrun_delayed_usecs`

`get_uuid_for_client_name`(*name*)

Get the session ID for a client name.

The session manager needs this to reassociate a client name to the session ID.

`get_client_name_by_uuid`(*uuid*)

Get the client name for a session ID.

In order to snapshot the graph connections, the session manager needs to map session IDs to client names.

`get_port_by_name` (*name*)

Get port by name.

Given a full port name, this returns a `Port`, `MidiPort`, `OwnPort` or `OwnMidiPort` object.

`get_all_connections` (*port*)

Return a list of ports which the given port is connected to.

This differs from `OwnPort.connections` (also available on `OwnMidiPort`) in two important respects:

1. You may not call this function from code that is executed in response to a JACK event. For example, you cannot use it in a graph order callback.
2. You need not be the owner of the port to get information about its connections.

`get_ports` (*name_pattern=’‘*, *is_audio=False*, *is_midi=False*, *is_input=False*, *is_output=False*, *is_physical=False*, *can_monitor=False*, *is_terminal=False*)

Return a list of selected ports.

Parameters

- **name_pattern** (*str*) – A regular expression used to select ports by name. If empty, no selection based on name will be carried out.
- **is_audio, is_midi** (*bool*) – Select audio/MIDI ports. If neither of them is `True`, both types of ports are selected.
- **is_input, is_output, is_physical, can_monitor, is_terminal** (*bool*) – Select ports by their flags. If none of them are `True`, no selection based on flags will be carried out.

Returns *list of Port/MidiPort/OwnPort/OwnMidiPort* – All ports that satisfy the given conditions.

class `jack.`**`Port`** (*port_ptr*)

A JACK audio port.

This class cannot be instantiated directly. Instead, instances of this class are returned from `Client.get_port_by_name()`, `Client.get_ports()`, `Client.get_all_connections()` and `OwnPort.connections`. In addition, instances of this class are available in the callbacks which are set with `Client.set_port_registration_callback()`, `Client.set_port_connect_callback()` or `Client.set_port_rename_callback()`.

Note, however, that if the used `Client` owns the respective port, instances of `OwnPort` (instead of `Port`) will be created. In case of MIDI ports, instances of `MidiPort` or `OwnMidiPort` are created.

Besides being the type of non-owned JACK audio ports, this class also serves as base class for all other port classes (`OwnPort`, `MidiPort` and `OwnMidiPort`).

New JACK audio/MIDI ports can be created with the `register()` method of `Client.inports`, `Client.outports`, `Client.midi_inports` and `Client.midi_outports`.

`name`

Full name of the JACK port (read-only).

`shortname`

Short name of the JACK port, not including “client_name – ”.

Must be unique among all ports owned by a client.

May be modified at any time. If the resulting full name (including the “client_name:” prefix) is longer than `port_name_size()`, it will be truncated.

`uuid`

The UUID of the JACK port.

`is_audio`

This is always `True`.

`is_midi`
This is always `False`.

`is_input`
Can the port receive data?

`is_output`
Can data be read from this port?

`is_physical`
Does it correspond to some kind of physical I/O connector?

`can_monitor`
Does a call to *`request_monitor()`* make sense?

`is_terminal`
Is the data consumed/generated?

`request_monitor`(*onoff*)
Set input monitoring.

If *`can_monitor`* is `True`, turn input monitoring on or off. Otherwise, do nothing.

Parameters **onoff** (*bool*) – If `True`, switch monitoring on; if `False`, switch it off.

class `jack.`**`MidiPort`**(*port_ptr*)
A JACK MIDI port.

This class is derived from *`Port`* and has exactly the same attributes and methods.

New JACK audio/MIDI ports can be created with the *`register()`* method of *`Client.inports`*, *`Client.outports`*, *`Client.midi_inports`* and *`Client.midi_outports`*.

See also:

`Port`, *`OwnMidiPort`*

`is_audio`
This is always `False`.

`is_midi`
This is always `True`.

class `jack.`**`OwnPort`**(*port_ptr*, *client*)
A JACK audio port owned by a *`Client`* object.

This class is derived from *`Port`*. *`OwnPort`* objects can do everything that *`Port`* objects can, plus a lot more.

This class cannot be instantiated directly. Instead, instances of this class are returned from *`Client.get_port_by_name()`*, *`Client.get_ports()`*, *`Client.get_all_connections()`* and *`connections`*. In addition, instances of this class are available in the callbacks which are set with *`Client.set_port_registration_callback()`*, *`Client.set_port_connect_callback()`* or *`Client.set_port_rename_callback()`*.

Note, however, that if the used *`Client`* doesn't own the respective port, instances of *`Port`* (instead of *`OwnPort`*) will be created. In case of MIDI ports, instances of *`MidiPort`* or *`OwnMidiPort`* are created.

New JACK audio/MIDI ports can be created with the *`register()`* method of *`Client.inports`*, *`Client.outports`*, *`Client.midi_inports`* and *`Client.midi_outports`*.

`number_of_connections`
Number of connections to or from port.

`connections`
List of ports which the port is connected to.

`is_connected_to`(*port*)
Am I *directly* connected to *port*?

Parameters **port** (*str or Port*) – Full port name or port object.

`connect`(*port*)

Connect to given port:

Parameters **port** (*str or Port*) – Full port name or port object.

See also:

`Client.connect()`

`disconnect`(*other=None*)

Disconnect this port.

Parameters **other** (*str or Port*) – Port to disconnect from. By default, disconnect from all connected ports.

`unregister`()

Unregister port.

Remove the port from the client, disconnecting any existing connections. This also removes the port from `Client.inports`, `Client.outports`, `Client.midi_inports` or `Client.midi_outports`.

`get_buffer`()

Get buffer for audio data.

This returns a buffer holding the memory area associated with the specified port. For an output port, it will be a memory area that can be written to; for an input port, it will be an area containing the data from the port's connection(s), or zero-filled. If there are multiple inbound connections, the data will be mixed appropriately.

Caching output ports is DEPRECATED in JACK 2.0, due to some new optimization (like "pipelining"). Port buffers have to be retrieved in each callback for proper functioning.

`get_array`()

Get audio buffer as NumPy array.

See also:

`get_buffer()`

class `jack.`**`OwnMidiPort`**(**args*, ***kwargs*)

A JACK MIDI port owned by a `Client` object.

This class is derived from `OwnPort` and `MidiPort`, which are themselves derived from `Port`. It has the same attributes and methods as `OwnPort`, but `get_buffer()` and `get_array()` are disabled. Instead, it has methods for sending and receiving MIDI events (to be used from within the process callback – see `Client.set_process_callback()`).

New JACK audio/MIDI ports can be created with the `register()` method of `Client.inports`, `Client.outports`, `Client.midi_inports` and `Client.midi_outports`.

`get_buffer`()

Not available for MIDI ports.

`get_array`()

Not available for MIDI ports.

`max_event_size`

Get the size of the largest event that can be stored by the port.

This returns the current space available, taking into account events already stored in the port.

`lost_midi_events`

Get the number of events that could not be written to the port.

This being a non-zero value implies that the port is full. Currently the only way this can happen is if events are lost on port mixdown.

`incoming_midi_events`()

Return generator for incoming MIDI events.

JACK MIDI is normalised, the MIDI events yielded by this generator are guaranteed to be complete MIDI events (the status byte will always be present, and no realtime events will be interspersed with the events).

Yields

- **time** (*int*) – Time (in samples) relative to the beginning of the current audio block.
- **event** (*buffer*) – The actual MIDI event data.

`clear_buffer`()

Clear an event buffer.

This should be called at the beginning of each process cycle before calling `reserve_midi_event()` or `write_midi_event()`. This function may not be called on an input port.

`write_midi_event`(*time*, *event*)

Create an outgoing MIDI event.

Clients must write normalised MIDI data to the port - no running status and no (one-byte) realtime messages interspersed with other messages (realtime messages are fine when they occur on their own, like other messages).

Events must be written in order, sorted by their sample offsets. JACK will not sort the events for you, and will refuse to store out-of-order events.

Parameters

- **time** (*int*) – Time (in samples) relative to the beginning of the current audio block.
- **event** (*bytes or buffer or sequence of int*) – The actual MIDI event data.

Note: Buffer objects are only supported for CFFI >= 0.9.

Raises `JackError` – If MIDI event couldn't be written.

`reserve_midi_event`(*time*, *size*)

Get a buffer where an outgoing MIDI event can be written to.

Clients must write normalised MIDI data to the port - no running status and no (one-byte) realtime messages interspersed with other messages (realtime messages are fine when they occur on their own, like other messages).

Events must be written in order, sorted by their sample offsets. JACK will not sort the events for you, and will refuse to store out-of-order events.

Parameters

- **time** (*int*) – Time (in samples) relative to the beginning of the current audio block.
- **size** (*int*) – Number of bytes to reserve.

Returns *buffer* – A buffer object where MIDI data bytes can be written to. If no space could be reserved, an empty buffer is returned.

class `jack.`**`Ports`**(*client*, *porttype*, *flag*)

A list of input/output ports.

This class is not meant to be instantiated directly. It is only used as `Client.inports`, `Client.outports`, `Client.midi_inports` and `Client.midi_outports`.

The ports can be accessed by indexing or by iteration.

New ports can be added with `register()`, existing ports can be removed by calling their `unregister()` method.

`register`(*shortname*, *is_terminal=False*, *is_physical=False*)

Create a new input/output port.

The new `OwnPort` or `OwnMidiPort` object is automatically added to `Client.inports`, `Client.outports`, `Client.midi_inports` or `Client.midi_outports`.

Parameters

- **shortname** (*str*) – Each port has a short name. The port's full name contains the name of the client concatenated with a colon (:) followed by its short name. The `port_name_size()` is the maximum length of this full name. Exceeding that will cause the port registration to fail.

  The port name must be unique among all ports owned by this client. If the name is not unique, the registration will fail.
- **is_terminal** (*bool*) – For an input port: If `True`, the data received by the port will not be passed on or made available at any other port. For an output port: If `True`, the data available at the port does not originate from any other port

  Audio synthesizers, I/O hardware interface clients, HDR systems are examples of clients that would set this flag for their ports.
- **is_physical** (*bool*) – If `True` the port corresponds to some kind of physical I/O connector.

Returns *Port* – A new `OwnPort` or `OwnMidiPort` instance.

`clear`()

Unregister all ports in the list.

See also:

`OwnPort.unregister()`

class `jack.`**`Status`**(*statuscode*)

Representation of the JACK status bits.

`failure`

Overall operation failed.

`invalid_option`

The operation contained an invalid or unsupported option.

`name_not_unique`

The desired client name was not unique.

With the *use_exact_name* option of `Client`, this situation is fatal. Otherwise, the name is modified by appending a dash and a two-digit number in the range "-01" to "-99". `Client.name` will return the exact string that was used. If the specified *name* plus these extra characters would be too long, the open fails instead.

`server_started`

The JACK server was started for this `Client`.

Otherwise, it was running already.

`server_failed`

Unable to connect to the JACK server.

`server_error`

Communication error with the JACK server.

`no_such_client`

Requested client does not exist.

`load_failure`

Unable to load internal client.

`init_failure`
Unable to initialize client.

`shm_failure`
Unable to access shared memory.

`version_error`
Client's protocol version does not match.

`backend_error`
Backend error.

`client_zombie`
Client zombified failure.

exception `jack.`**`JackError`**
Exception for all kinds of JACK-related errors.

`jack.`**`version`**`()`
Get tuple of major/minor/micro/protocol version.

`jack.`**`version_string`**`()`
Get human-readable JACK version.

`jack.`**`client_name_size`**`()`
Return the maximum number of characters in a JACK client name.

This includes the final NULL character. This value is a constant.

`jack.`**`port_name_size`**`()`
Maximum length of port names.

The maximum number of characters in a full JACK port name including the final NULL character. This value is a constant.

A port's full name contains the owning client name concatenated with a colon (:) followed by its short name and a NULL character.

`jack.`**`set_error_function`**`(`*callback=None*`)`
Set the callback for error message display.

Set it to `None` to restore the default error callback function (which prints the error message plus a newline to stderr). The *callback* function must have this signature:

```
callback(message:str) -> None
```

`jack.`**`set_info_function`**`(`*callback=None*`)`
Set the callback for info message display.

Set it to `None` to restore default info callback function (which prints the info message plus a newline to stderr). The *callback* function must have this signature:

```
callback(message:str) -> None
```

`jack.`**`client_pid`**`(`*name*`)`
Return PID of a JACK client.

Parameters **name** (*str*) – Name of the JACK client whose PID shall be returned.

Returns *int* – PID of *name*. If not available, 0 will be returned.

5 Version History

Version 0.2.0 (2015-02-23):

- MIDI support (*`jack.MidiPort`*, *`jack.OwnMidiPort`*, *`jack.Client.midi_inports`*, *`jack.Client.midi_outports`*, ...)

- ignore errors in `jack.Client.deactivate()` by default, can be overridden
- optional argument to `jack.OwnPort.disconnect()`
- several examples (`chatty_client.py`, `thru_client.py`, `midi_monitor.py` and `midi_chords.py`)
- `jack.Port.is_physical`, courtesy of Alexandru Stan
- `jack.Status` class
- some bug-fixes and refactorings, some documentation improvements

Version 0.1.0 (2014-12-15): Initial release