python-rtmixer

Release 0.1.4

Matthias Geier

2022-01-05

Contents

Warning: This is work in progress!

Goal: Reliable low-latency audio playback and recording with Python, using PortAudio via the sounddevice module.

The audio callback is implemented in C (and compiled with the help of CFFI) and doesn't invoke the Python interpreter, therefore avoiding waiting for things like garbage collection and the GIL.

All PortAudio platforms and host APIs are supported. Runs on any Python version where CFFI is available.

Online documentation https://python-rtmixer.readthedocs.io/

Source code repository https://github.com/spatialaudio/python-rtmixer

Somewhat similar projects

- https://github.com/nwhitehead/swmixer
- https://github.com/nvahalik/PyAudioMixer
- http://www.pygame.org/docs/ref/mixer.html

1 Features

- playback of multiple signals at the same time (that's why it's called "mixer")
- play from buffer, play from ringbuffer
- record into buffer, record into ringbuffer
- multichannel support
- NumPy arrays with data type `'float32'` can be easily used (via the buffer protocol) as long as they are C-contiguous
- fixed latency playback, (close to) no jitter (optional)
  - to be verified …
- sample-accurate playback/recording (with known offset)
  - to be verified …
- non-blocking callback function, using PortAudio ringbuffers
- all memory allocations/deallocations happen outside the audio callback

1.1 Planned Features

- meticulous reporting of overruns/underruns
- loopback tests to verify correct operation and accurate latency values
- fade in/out?
- loop?
- playlist/queue?

1.2 Out Of Scope

- reading from/writing to files (use e.g. the soundfile module instead)
- realtime signal processing inside the audio callback (ring buffers can be used as a work-around, see the signal_processing.py example)
- signal generators
- multiple mixer instances (some PortAudio host APIs only support one stream at a time)
- resampling (apart from what PortAudio does)
- fast forward/rewind
- panning/balance
- audio/video synchronization

2 Installation

On Windows, macOS and Linux you can install a precompiled "wheel" package with:

```
python3 -m pip install rtmixer
```

This will install `rtmixer` and its dependencies, including `sounddevice`.

Depending on your Python installation, you may have to use `python` instead of `python3`. If you have installed the module already, you can use the `--upgrade` flag to get the newest release.

Note: On Linux, to use `sounddevice` and `rtmixer` you will need to have PortAudio installed, e.g. via `sudo apt install libportaudio2`. On other platforms, PortAudio comes bundled with `sounddevice`.

---

3 Usage

See the list of examples on GitHub.

4 API Documentation

Mixer	PortAudio output stream for realtime mixing.
Recorder	PortAudio input stream for realtime recording.
MixerAndRecorder	PortAudio stream for realtime mixing and recording.
RingBuffer	PortAudio's single-reader single-writer lock-free ring buffer.

Common parameters that are shared by most commands:

start Desired time at which the playback/recording should be started. The actual time will be stored in the `actual_time` field of the returned action.

time Desired time at which the command should be executed. The actual time will be stored in the `actual_time` field of the returned action.

channels This can be either the desired number of channels or a list of (1-based) channel numbers that is used as a channel map for playback/recording.

allow_belated Use `False` to cancel the command in case the requested time cannot be met. The `actual_time` field will be set to `0.0` in this case. Use `True` to execute the command nevertheless. Even if the requested time was met, the `actual_time` might be slightly different due to rounding to the next audio sample.

All commands return a corresponding "action", which can be compared against the active *actions*, and can be used as input for *cancel()* and *wait()*. The fields of action objects are defined in C but can be accessed with Python (e.g. `my_action.stats.min_blocksize`) *after* the command is finished:

```
struct action
{
  const enum actiontype type;
  const PaTime requested_time;
  PaTime actual_time;  // Set != 0.0 to allow belated actions
  struct action* next;  // Used to create singly linked list of actions
  union {
    float* const buffer;
    struct PaUtilRingBuffer* const ringbuffer;
```

(continues on next page)

(continued from previous page)

```
    struct action* const action;  // Used in CANCEL
  };
  frame_t total_frames;
  frame_t done_frames;
  struct stats stats;
  // TODO: ringbuffer usage: store smallest available write/read size?
  const frame_t channels;  // Size of the following array
  const frame_t mapping[];  // "flexible array member"
};
```

The `stats` field contains some statistics collected during playback/recording (again, *after* the command is finished):

```
struct stats
{
  frame_t blocks;
  frame_t min_blocksize;
  frame_t max_blocksize;
  frame_t input_underflows;
  frame_t input_overflows;
  frame_t output_underflows;
  frame_t output_overflows;
};
```

These statistics are also collected for the whole runtime of a stream, where they are available as *stats* attribute (but only if the stream is *inactive*). The statistics of an *active* stream can be obtained (and at the same time reset) with *fetch_and_reset_stats()*.

class rtmixer.**Mixer**(***kwargs*)

PortAudio output stream for realtime mixing.

Takes the same keyword arguments as `sounddevice.OutputStream`, except *callback* (a callback function implemented in C is used internally) and *dtype* (which is always `'float32'`).

Uses default values from `sounddevice.default` (except *dtype*, which is always `'float32'`).

Has the same methods and attributes as `sounddevice.OutputStream` (except `write()` and `write_available`), plus the following:

property actions

The set of active "actions".

cancel(*action*, *time=0*, *allow_belated=True*)

Initiate stopping a running action.

This creates another action that is sent to the callback in order to stop the given *action*.

This function typically returns before the *action* is actually stopped. Use *wait()* (on either one of the two actions) to wait until it's done.

fetch_and_reset_stats(*time=0*, *allow_belated=True*)

Fetch and reset over-/underflow statistics of the stream.

The statistics will be available in the `stats` field of the returned action.

play_buffer(*buffer*, *channels*, *start=0*, *allow_belated=True*)

Send a buffer to the callback to be played back.

After calling this, the *buffer* must not be written to anymore.

play_ringbuffer(*ringbuffer*, *channels=None*, *start=0*, *allow_belated=True*)

Send a *RingBuffer* to the callback to be played back.

By default, the number of channels is obtained from the ring buffer's *elementsize*.

property stats
Get over-/underflow statistics from an *inactive* stream.

To get statistics from an `active` stream, use *`fetch_and_reset_stats()`*.

wait(*action=None*, *sleeptime=10*)
Wait for *action* to be finished.

Between repeatedly checking if the action is finished, this waits for *sleeptime* milliseconds.

If no *action* is given, this waits for all actions.

class rtmixer.**Recorder**(***kwargs*)
PortAudio input stream for realtime recording.

Takes the same keyword arguments as `sounddevice.InputStream`, except *callback* (a callback function implemented in C is used internally) and *dtype* (which is always `'float32'`).

Uses default values from `sounddevice.default` (except *dtype*, which is always `'float32'`).

Has the same methods and attributes as *`Mixer`*, except that *`play_buffer()`* and *`play_ringbuffer()`* are replaced by:

record_buffer(*buffer*, *channels*, *start=0*, *allow_belated=True*)
Send a buffer to the callback to be recorded into.

record_ringbuffer(*ringbuffer*, *channels=None*, *start=0*, *allow_belated=True*)
Send a *`RingBuffer`* to the callback to be recorded into.

By default, the number of channels is obtained from the ring buffer's *`elementsize`*.

class rtmixer.**MixerAndRecorder**(***kwargs*)
PortAudio stream for realtime mixing and recording.

Takes the same keyword arguments as `sounddevice.Stream`, except *callback* (a callback function implemented in C is used internally) and *dtype* (which is always `'float32'`).

Uses default values from `sounddevice.default` (except *dtype*, which is always `'float32'`).

Inherits all methods and attributes from *`Mixer`* and *`Recorder`*.

class rtmixer.**RingBuffer**(*elementsize*, *size=None*, *buffer=None*)
PortAudio's single-reader single-writer lock-free ring buffer.

C API documentation: http://portaudio.com/docs/v19-doxydocs-dev/pa__ringbuffer_8h.html

Python wrapper: https://github.com/spatialaudio/python-pa-ringbuffer

Instances of this class can be used to transport data between Python code and some compiled code running on a different thread.

This only works when there is a single reader and a single writer (i.e. one thread or callback writes to the ring buffer, another thread or callback reads from it).

This ring buffer is *not* appropriate for passing data from one Python thread to another Python thread. For this, the `queue.Queue` class from the standard library can be used.

Parameters

- **elementsize** (*int*) – The size of a single data element in bytes.
- **size** (*int*) – The number of elements in the buffer (must be a power of 2). Can be omitted if a pre-allocated buffer is passed.
- **buffer** (*buffer*) – optional pre-allocated buffer to use with RingBuffer. Note that if you pass a read-only buffer object, you still get a writable RingBuffer; it is your responsibility not to write there if the original buffer doesn't expect you to.

advance_read_index(*size*)
Advance the read index to the next location to be read.

Parameters **size** (*int*) – The number of elements to advance.

Returns The new position.

Note: This is only needed when using *get_read_buffers()*, the methods *read()* and *readinto()* take care of this by themselves!

---

advance_write_index(*size*)
Advance the write index to the next location to be written.

Parameters **size** (*int*) – The number of elements to advance.

Returns The new position.

Note: This is only needed when using *get_write_buffers()*, the method *write()* takes care of this by itself!

---

property elementsize
Element size in bytes.

flush()
Reset buffer to empty.

Should only be called when buffer is **not** being read or written.

get_read_buffers(*size*)
Get buffer(s) from which we can read data.

When done reading, use *advance_read_index()* to make the memory available for writing again.

Parameters **size** (*int*) – The number of elements desired.

Returns

- The number of elements available for reading (which might be less than the requested *size*).
- The first buffer.
- The second buffer.

Return type tuple (int, buffer, buffer)

get_write_buffers(*size*)
Get buffer(s) to which we can write data.

When done writing, use *advance_write_index()* to make the written data available for reading.

Parameters **size** (*int*) – The number of elements desired.

Returns

- The room available to be written or the given *size*, whichever is smaller.
- The first buffer.
- The second buffer.

Return type tuple (int, buffer, buffer)

read(*size=-1*)
Read data from the ring buffer into a new buffer.

This advances the read index after reading; calling *advance_read_index()* is *not* necessary.

Parameters **size** (*int, optional*) – The number of elements to be read. If not specified, all available elements are read.

Returns A new buffer containing the read data. Its size may be less than the requested *size*.

property read_available
Number of elements available in the ring buffer for reading.

readinto(*data*)
Read data from the ring buffer into a user-provided buffer.

This advances the read index after reading; calling *advance_read_index()* is *not* necessary.

Parameters **data** (*CData pointer or buffer*) – The memory where the data should be stored.

Returns The number of elements read, which may be less than the size of *data*.

write(*data*, *size=-1*)
Write data to the ring buffer.

This advances the write index after writing; calling *advance_write_index()* is *not* necessary.

Parameters

- **data** (*CData pointer or buffer or bytes*) – Data to write to the buffer.
- **size** (*int, optional*) – The number of elements to be written.

Returns The number of elements written.

property write_available
Number of elements available in the ring buffer for writing.

5 Contributing

If you find bugs, errors, omissions or other things that need improvement, please create an issue or a pull request at https://github.com/spatialaudio/python-rtmixer/. Contributions are always welcome!

5.1 Development Installation

Instead of pip-installing the latest release from PyPI, you should get the newest development version (a.k.a. "master") with Git:

```
git clone https://github.com/spatialaudio/python-rtmixer.git --recursive
cd python-rtmixer
python3 -m pip install -e .
```

… where `-e` stands for `--editable`.

When installing this way, you can quickly try other Git branches (in this example the branch is called "another-branch"):

```
git checkout another-branch
```

If you want to go back to the "master" branch, use:

```
git checkout master
```

To get the latest changes from Github, use:

```
git pull --ff-only
```

If you used the `--recursive` option when cloning, the `portaudio` submodule (which is needed for compiling the module) will be checked out automatically. If not, you can get the submodule with:

```
git submodule update --init
```

5.2 Building the Documentation

If you make changes to the documentation, you should create the HTML pages locally using Sphinx and check if they look OK.

Initially, you might need to install a few packages that are needed to build the documentation:

```
python3 -m pip install -r doc/requirements.txt
```

To (re-)build the HTML files, use:

```
python3 setup.py build_sphinx
```

The generated files will be available in the directory `build/sphinx/html/`.

5.3 Creating a New Release

New releases are made using the following steps:

1. Bump version number in `src/rtmixer.py`
2. Update `NEWS.rst`
3. Commit those changes as "Release x.y.z"
4. Create an (annotated) tag with `git tag -a x.y.z`
5. Push the commit and the tag to Github
6. Wait 10 minutes for the PyPI packages to be automagically uploaded
7. On Github, add release notes containing a link to PyPI and the bullet points from `NEWS.rst`
8. Check that the new release was built correctly on RTD and select the new release as default version

6 Version History

Version 0.1.3 – 2021-05-05 – PyPI – diff

- Add Python 3.9 support by adding entries to `azure-pipelines.yml` for `cibuildwheel`

Version 0.1.2 – 2020-08-20 – PyPI – diff

- Clean up `cibuildwheel`

Version 0.1.1 – 2020-06-19 – PyPI – diff

- Add Python 3.8 support by adding entries to `azure-pipelines.yml` for `cibuildwheel`

Version 0.1.0 – 2019-09-05 – PyPI Initial release

Python Module Index

r